DOTTORATO DI RICERCA IN
INGEGNERIA DELL'INFORMAZIONE
XVIII CICLO

Sede Amministrativa
Università degli Studi di MODENA e REGGIO EMILIA

TESI PER IL CONSEGUIMENTO DEL TITOLO DI DOTTORE DI RICERCA

Efficient provisioning and adaptation of Web-based services

Ing. Francesca Mazzoni

Relatore:
Prof. Michele Colajanni

Anno Accademico 2004 - 2005

Contents

1 Introduction **1**

I Efficient Web content generation **7**

2 Design and testing of dynamic Web-based systems **9**

2.1 Introduction 9

2.2 Related work 13

2.3 Design issues for a dynamic Web-based system 15

2.3.1 Performance constraints 15

2.3.2 Design goals 17

2.3.3 Scalability of software technologies 18

2.3.4 Scalability of hardware technologies 21

2.4 Web-based system performance testing 22

2.4.1 Workload specification 22

2.4.2 Coarse grain times in a Web-based system 25

2.4.3 Black box performance testing 27

2.4.4 White box performance testing and bottleneck removal 28

2.5 A case study for Web-based system performance testing 32

2.5.1 Workload model definition 33

2.5.2 Architectural testbed 34

2.5.3 System capacity evaluation 36

2.5.4 Bottleneck location 37

2.5.5 Bottleneck causes identification 38

2.5.6 Bottleneck removal 41
2.5.7 Impact of wide area network effects 43
2.6 Summary of the results 47

II Efficient Web content delivery 49

3 Web content caching and delivery 51
3.1 Introduction 51
3.2 Related work 54
3.3 Performance evaluation of pure protocols 57
3.3.1 Limits of query-based schemes 58
3.3.2 Limits of directory- and summary-based schemes . . . 60
3.4 Summary of the performance evaluation 63

4 Two-tier distributed architectures for Web caching 65
4.1 Introduction 65
4.2 Two-tier cooperation protocols 68
4.2.1 InterQ-IntraS cooperation protocol 69
4.2.2 InterS-IntraQ cooperation protocol 73
4.3 Prototype implementation 77
4.3.1 Implementation of the InterQ-IntraS and InterQ-IntraS-DM schemes 78
4.3.2 Implementation of the InterS-IntraQ scheme 79
4.4 Workload models for performance evaluation 79
4.5 Performance evaluation of hybrid cooperation schemes 81
4.5.1 Scalability of the cooperation protocols 81
4.5.2 Sensitivity analysis on cache capacity 87
4.5.3 Experiments on a geographic testbed 88
4.5.4 Summary of the experimental results 92

III Efficient Web content adaptation 95

5 Intermediary adaptation systems 97

5.1 Introduction 97
5.2 Related work 101
5.3 Programmable intermediary systems on the WWW 101
5.4 Scalable Intermediary Software Infrastructure (SISI) 107
5.5 Integrating SISI with new functionalities 116
5.5.1 SISI services 116
5.5.2 Easy SISI Services Deployment 121
5.5.3 HTML/HTTP Parsing Library 123
5.6 SISI Management and Configuration 124
5.6.1 SISI Visual Monitor 125
5.6.2 Users' profiles configuration 127
5.7 "Out-of-the-box" SISI Services 129
5.8 Performance Evaluation 135
5.8.1 Workload Model and Testbed 135
5.8.2 Overhead of the intermediary adaptation server 137
5.8.3 Performance Comparison 138
5.8.4 SISI Scalability 140

6 Conclusions 143

6.1 Results 143
6.2 Future directions 144

A List of Acronyms 147

Bibliography 151

List of Tables 165

List of Figures 167

Chapter 1

Introduction

The World Wide Web has nowadays become the favorite access portal to any kind of information and service. On the client side, Web users are becoming more demanding, both in terms of required service complexity and in terms of fast interaction with the services they access through the Web. Moreover, there is an increasing need of adaptation/personalization services to tailor Web contents to the user device capabilities and to their personal preferences.

On the server side, we should consider that most of present contents are generated on-the-fly on the basis of information that is dynamically provided by the user. Typically, many entities interact to build such dynamic contents: a Web server may have to interact with application and DBMS servers to get some information that it uses to build the whole Web response.

All previous characteristics augment the need for designing and building adequate server infrastructures that efficiently generate, deliver and, possibly, adapt Web-based contents and services. This thesis aims at studying and proposing novel infrastructures and solutions that are able to cope with the needs of present and future Web users.

In Figure 1.1 we evidence four main logical levels in a Web-based infrastructure for the support of modern services.

- The *Content Generation Level* denotes the infrastructure of the content providers that host the origin versions of all resources and services.

- The *Content Delivery Level* may be utilized to improve the delivery of Web-based contents through some caching mechanisms.
- The *Content Adaptation Level* deploys adaptation services to provide users with personalized content and services.
- The *Client Level* denotes the applications that request content and services from the content/service providers.

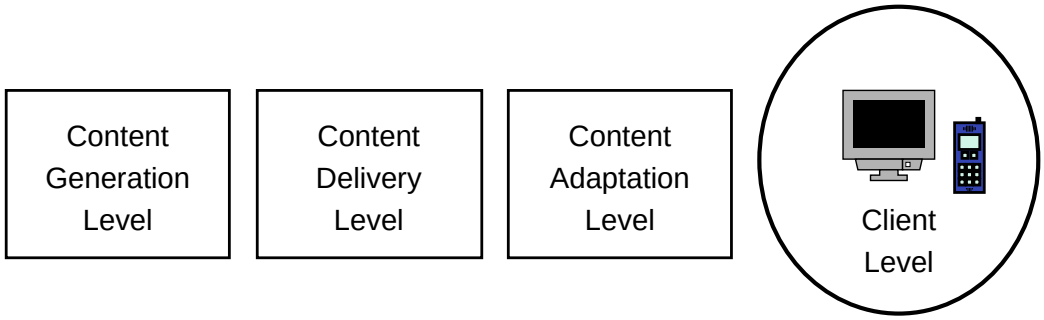

Figure 1.1: A logical view of a modern Web-based infrastructure for the support of efficient services

In this thesis, we consider and propose solutions for the first three levels by excluding client-based solutions.

In the first part, we focus on content generation; in the second part, we describe distributed architectures of cooperating cache servers, that aim at decreasing the download times for Web resources; in the third part, we focus on the adaptation level.

The main goal of our study is to propose architectures and protocols to provide the users with "good" services, both in terms of performance (low response times) and in terms of users' perception of navigation that is, to tailor Web contents and services to the client device and users' preferences.

Let us summarize the main contributions on each of the three research topics.

1. **Efficient content generation**.

   - We present an innovative methodology for the testing of modern, dynamic Web-based services and for the improvement of existing systems that must satisfy some performance constraints even in the case of unpredictable load variations. A case study describes the application of the proposed methodology to a typical consumer-oriented Web site. We show that a coarse grain analysis, that is used in most performance studies, may lead to incomplete or false deductions about the behavior of the hardware and software components supporting Web-based services. Through a fine grain performance evaluation of a cluster-based Web system, we find some interesting results that demonstrate the importance of an analysis approach that is carried out at the software function level and utilizes higher moment metrics instead of average values.

2. **Efficient content delivery**.

   - We provide the first thorough comparison of well known algorithms for efficient resource delivery through distributed Web caching systems. We evaluate the scalability of hierarchical and flat cooperation architectures, and we conclude that such protocols show poor scalability properties.
   - We propose novel architectures (namely *two-tier architectures*), algorithms and protocols for cooperative Web caching and delivery. We demonstrate that two-tier architectures provide a scalable solution for Web delivery and improve the performance stability by reducing the variance in the user response time.

3. **Efficient adaptation of Web-based contents**.

   - We propose a novel framework, namely *Scalable Intermediary Software Infrastructure (SISI)*, for the efficient adaptation and personalization of Web-based contents. One of SISI's key features is that it is able to deal with per-user profiles, provided that most existing frameworks only allow for a system wide profile. Thanks to its flexible structure, SISI shows an easy programmability that is, programmers can quickly extend SISI with new services. Despite its flexibility, SISI does not penalize the user with long response times.

The thesis, organized in six chapters, is divided into three parts: content generation, content delivery, content adaptation.

In Chapter 2 we propose a methodology for the testing of modern, dynamic Web-based services and for the improvement of existing systems that must satisfy some performance constraints even in the case of unpredictable load variations. This aims at providing the users with a good service in terms of performance. The resources are still tailored neither to the client device capabilities, nor to the users preferences.

In Chapter 3 we describe the main distributed architectures for cooperative Web caching and the main issues of content delivery through Web caching. The goal of the studies carried out in this and in the following chapter is to further improve the performance of the whole Web-based system through caching mechanisms applied before the adaptation task.

In Chapter 4 we propose novel architectures, algorithms and protocols for cooperative Web caching and delivery. The proposed protocols guarantee better scalability and stability in the user response time, than those studied in the previous chapter, thus improving the users' perception of the Web-based system in the whole.

In Chapter 5 we propose a novel framework, namely Scalable Intermediary Software Infrastructure (SISI), that aims to give an original solution to the trade-off between programmability and efficiency of content adaptation/personalization.

Chapter 6 summarizes the main results and contributions of this thesis and indicates some directions for future research.

Part I

Efficient Web content generation

Chapter 2

Design and performance testing of dynamic Web-based systems

2.1 Introduction

In the first part of the thesis we focus on the content generation level of the Web-based infrastructure shown in Figure 2.1.

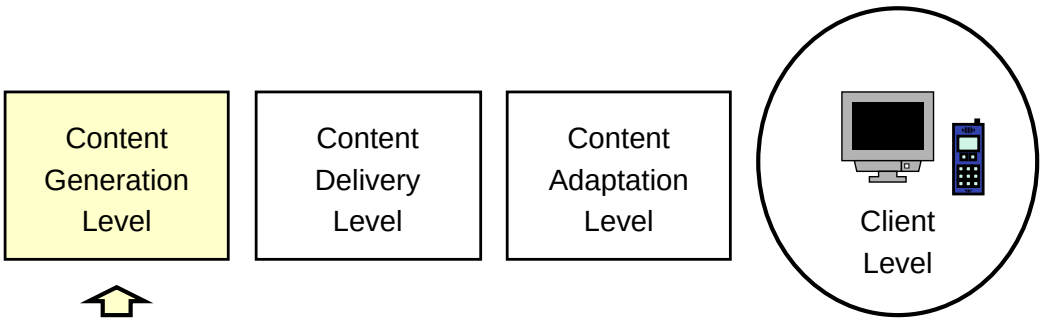

Figure 2.1: The generation level of the Web-based infrastructure

First, we give some hints on system design: we describe the main differences between the most popular solutions aiming at understanding their strengths and weaknesses. Then, we focus on system testing: we propose a novel methodology to test the performance of Web-based services to verify whether goals of the design phase have been met. Since in this chapter we are

focusing on the content generation level, for the sake of simplicity, we assume that the client is directly connected to this level, without the intervention of the delivery and adaptation levels.

System performance and scalability remain key factors for the success of any Web-based service. For example, the popularity of a Web-based system, that is perceived as too slow or that suffers availability problems, drops dramatically because navigation becomes a frustrating experience for the user. The need for complex and often critical services has led to the deployment of specific Web-based technologies for content generation that fall into three main classes:

- Single-server
- LAN-based systems (also called locally *distributed Web systems* or *Web clusters*)
- WAN-based systems (also called *geographically distributed Web systems*)

In the *single server* architecture all the tasks are carried out by one server. This approach presents some evident scalability limits because a single node running all services (Web server, application server and DMBS server) may quickly become the performance bottleneck.

LAN-based systems are used to address the risk of congestion at the Web server. LAN-based systems systems are tightly coupled architectures with a single system interface. The node hosting the interface, called *Web switch* [Car01], dispatches the client requests to the other nodes of the cluster hosting the Web servers. Clusters are organized as multi-level systems, as shown in Figure 2.2, where Web servers are the front-end level. Behind the front-end level there are one or more application and DBMS servers, that compose the application and back-end levels of the architecture [ACM06, ACLM04].

WAN-based systems replicate the Web clusters over a geographical scenario [Car01, PCL06] to avoid the risks of network congestion that may arise at the first mile and peering points.

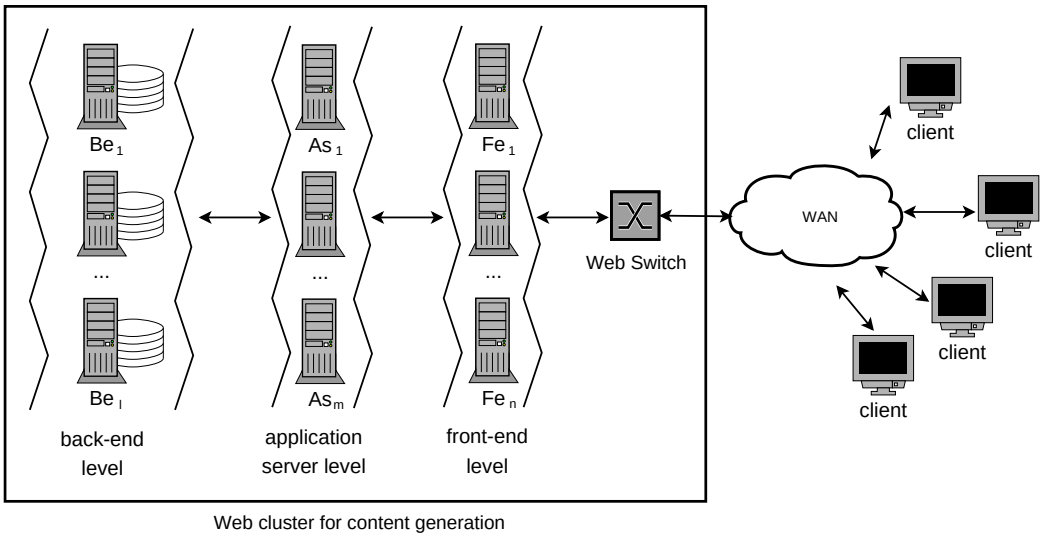

Figure 2.2: A typical LAN-based system for content generation

In this thesis we focus on LAN-based systems for content generation. Throughout this first part we investigate the issues related to the design and testing of Web systems that take into account performance constraints, even for unpredictable request frequencies. The proposed approach is conceptually valid for every Web-based services multi-level infrastructure, although here we focus on a dynamic Web-based system. Indeed, this category is widely diffused and it presents interesting design challenges.

A huge number of software and hardware technologies are available to support Web-based systems with their pros and cons in terms of complexity, performance, cost. Unlike the previous Web-based systems providing most static information, the modern Web-based services and technologies are extremely heterogeneous. Although it is impossible to identify from this universe of alternatives the solution which suits best to every possible dynamic-oriented Web-based system, we show the main invariants to design a scalable infrastructure.

The testing phase of these technologies to support modern Web-based systems seems more premature than the design phase. Indeed, the high complexity of these hardware/software infrastructures consisting of distributed components, processes and server nodes together with the heavy-tailed characteristics of the Web workload and the unpredictable nature of the expected load reaching a site make inapplicable most results of the performance literature. Although the main principles remain valid, we show that an accurate performance evaluation of dynamic Web-based systems is still a quite difficult task. For example, most performance studies use coarse grain load monitors and average performance metrics that are inadequate to the purpose of achieving an available infrastructure to support Web-based services, especially in the presence of unpredictable bursts of requests. Moreover, coarse grain load monitors are helpful just to give a first idea about the system behavior and for a preliminary bottleneck analysis. The limits of this coarse grain approach arise when it is necessary to understand the real motivations that are behind poor performance or a bottleneck, in a system that has literally hundreds of processes/threads cooperating and conflicting for the same resources. Hence, we show that it is necessary to delve into a finer grain performance analysis that yields a more detailed view about the system components and, if necessary, goes down to consider even processes and functions.

A peculiarity of most dynamic sites is the strong interaction between the Web technology and information sources for nearly every client request. In this part of the thesis we focus on the Web system components. There are three main abstract functions that are typically organized to form a so called multi-level architecture for content generation: *HTTP interface* also known as front-end level, *application logic* and *information source*, also known as back-end level. The HTTP interface handles connection requests from the clients through the standard HTTP protocol and serves static content, while it is not responsible for the generation of dynamic content. The functions offered by the *application logic* are at the heart of a Web system: they define

the logic behind the generation of dynamic content and build the HTML documents that will be sent back to the clients. Usually, the construction of a Web page requires the retrieval of further data from some *information source*. This last level provides functions for storage of critical information that is used in the generation of dynamic content. The final result of the computations is an HTML (or XML) document which is sent back to the HTTP interface that manages the delivery to the client.

In this part of the thesis, we illustrate and face the difficulties related to the design and testing of dynamic-oriented Web-based systems that are subject to some performance constraints. We show the main techniques to find the maximum capacity of a Web system and to identify bottlenecks when the system shows poor performance, especially in the presence of unpredictable bursts of requests. It is worth to premise that we remain in the domain of best-effort Web-based services, with no strict guarantees on the performance levels, similarly to those that characterize QoS-based applications [MBD01, CM02].

We consider a case study to show the effectiveness of both coarse and fine grain analysis of a dynamic Web-based system, that is built through open source software and runs on PC-like hardware. Although the results of this study cannot be immediately extended to the complexity of sites built through sophisticated proprietary suites, such as Oracle Portal [Ora06b], BEA Web Logic Server [Web06a] and IBM Web Sphere [Web06b], the spirit of the approach and the main conclusions are quite representative for the large majority of dynamic Web-based systems receiving hundreds of requests per minute that is, they are in the order of millions requests per day.

2.2 Related work

The design of scalable systems has been widely addressed in the literature on distributed systems. The main challenge here is to provide an acceptable level of performance even in the case where user access patterns are subject

to change. For example, in [IBM02, Chi00] the authors propose techniques for the design of Web system with good scalability properties. However, the design process proposed in these studies is based on the assumption that the expected request load offered to the system is a-priori knowledge. This information is not always available with a sufficient degree of precision. Studies on capacity planning, such as [ABCdO96], also focus on addressing user access load changes over time. However, as for the previous studies, a knowledge about the expected system load is the key for the system design and tuning. Our study follows a different approach to the problem of Web system design and proposes a novel approach that does not rely on any a-priori knowledge of the expected system load. We show how to design and test scalable dynamic Web systems through off-the-shelf software and hardware.

Many papers illustrate different aspects of Web-based system testing. For example, in [XBC02, HNY01] a fine grained analysis of the performance on HTTP servers is provided. These studies focus on Web-based systems serving mainly static Web resources and do not take into account the complexity and the interactions of a more complex Web-based system providing highly dynamic and personalized contents. Other studies discuss the issues behind the testing of a dynamic Web-based system. For example, in [JKB03, HY00] different technologies to implement the same e-commerce system are compared, while in [DMB01] a detailed analysis of the TPC-W model used for dynamic Web-based systems benchmarking is provided. Another interesting study by Aguilera *et al.* [AMW+03] provides an insight on the performance analysis of distributed, multi-tiered Web systems where the components cannot be instrumented to provide fine grained analysis. However, all these studies share the common trait of focusing on a coarse grain performance analysis of the systems at most at the node level. The identification of critical indexes and metrics for the performance evaluation of Web-based systems has been addressed in many contexts, but not with the goal of analyzing the consequences of the different index granularities. To the best of our knowledge,

this is one of the first studies that integrate both the coarse and the fine grain approach as a valuable aid for the testing of dynamic Web-based systems.

2.3 Design issues for a dynamic Web-based system

2.3.1 Performance constraints

In the design phase we should consider that all interactive Internet-based services are intrinsically characterized by the performance level that is considered acceptable for each service. These performance constraints may be stricter, for example in the case of real-time multimedia delivery, or more relaxed, for example in the case of non commercial Web sites, with a large spectrum of alternatives within these two extremes. In the design phase, we could even consider the possibility of having a QoS-based Web system that requires the respect of rigorous Service Level Agreements (SLAs). Actually, these systems have not yet succeeded outside of the research labs because of the difficulty of guaranteeing end-to-end performance when the Web provider controls a tiny part of the system, with no influence on the long way from his system to the user machine. There is an interesting literature on QoS-based Web sites that basically aims to give guarantees on the server response time, e.g. [IKLR03, MBD01, CM02], but in this thesis we do not refer to these systems.

Outside this QoS literature, performance constraints do not have a precise definition. A typical performance index that quantifies the behavior of the whole Web-based system is the *Web page response time* that is, the time elapsed from the user click until the arrival of the last object composing a Web resource. This index is of main interest for the users that care on the time they have to wait for the fruition of a service, but it is often expressed vaguely because of the already evidenced problem that a content provider cannot influence what he does not control that is, all Internet-related com-

ponents outside the first mile (mainly, peering points between Autonomous Systems and last mile). Often, the performance constraints define the level of *adequate performance*. For example, a previous study by IBM [IBM02] provides a ranking of parameters (ranging from unacceptable to excellent) in terms of response time for a typical Web page loaded by a dial-up user. The study concludes that a Web page response time higher than 30 seconds is unacceptable, while everything below 20 seconds is considered at least adequate. In [Nie94] the reaction of broadband users to different Web page download times is analyzed. One interesting conclusion is that 10 seconds is about the limit for keeping the user's attention focused on the browser window. Longer lasting Web page downloads lead the users to perform other tasks.

The problem is that it is quite difficult to anticipate the possible offered loads to the Web-based system with no previous experience. The large number of system- and user-oriented performance parameters (some of which are reported in Section 2.4) make even more complicated to define exact levels of adequate performance without testing the system under some representative workload models. In practice, the definition of performance expectations is an iterative process by itself.

We should consider that in critical Web-based systems, the access frequencies could be controlled through sophisticated access mechanisms, such as admission control that is adopted for QoS-based systems, but in most instances, the management would like to reject no user. Consequently, the maximum system capacity is the only parameter that can be acted upon. As a consequence, the only modifiable parameter is the maximum capacity of the system. We have to design architectures that are intrinsically scalable so that we can easily adapt them to novel performance constraints, even for highly fluctuating values of access frequencies.

2.3.2 Design goals

From the previous analyses we have outlined the three main requirements for the design of a Web-based system in terms of provided services, expected workload models and performance constraints. Starting from these inputs the design and deployment of the Web-based systems become a matter of choosing the right software and hardware technologies that can guarantee acceptable performance. But even other considerations are in order, coming from the observation that many dynamic Web-based systems might be subject to heavy bursts of client arrivals, that it is difficult if not impossible to anticipate the workloads to the Web-based system, that the workload is highly dynamic. Let us point out the three main issues that we consider important to be respected in the design and testing phase, and that we detail in remaining part of this chapter.

Scalability. It is mandatory to design a *scalable* system that is, a system that can easily increase its performance through the addition of some hardware/software components. To this purpose, it is important to know the main strengths and weaknesses of the available technologies for dynamic-oriented Web-based systems that we outline below.

Margin. In the testing phase, an acceptable system must demonstrate to have a response time well below the performance constraint level because it should be able to face even initially unexpected workload models. We remark the importance that the Web-based system works within safe margins of performance, due to unpredictable peaks in the volume and nature of client requests.

Trend. Another important goal of the testing phase is to check the performance trends of the system that is subject to different workload models. An acceptable system must not only demonstrate to have a response time well below the performance constraint level at some given points, but also not to show a tendency to increase suddenly its response time because of some close system bottlenecks. We will see that the com-

plex infrastructures supporting Web-based systems are characterized by gracefully degrading resources (e.g., CPU) and by suddenly degrading resources that can immediately lead the system to thrashing when there are no items in the available set (e.g., memory, descriptors, connections).

Below we analyze software and hardware technologies that can facilitate the design and implementation of a scalable infrastructure. In the following of the chapter we consider the main issues related to testing.

The use of three separate levels of functions (presentation, application, information) has its advantages. The most obvious is the *modularity*: if the interfaces among different abstraction levels are kept consistent, changes at one level do not influence other levels. Another advantage is the *scalability*: the separation of abstraction layers makes it easier to deploy them on different nodes. It is even possible to deploy a single level over multiple, identical nodes. Hence, the service of a dynamic request is the result of the interaction of multiple and complex functions. Each of them can be deployed through different software technologies that have their own strengths and weaknesses. Furthermore, they can be mapped in different ways to the underlying hardware. A performance-oriented design must address both issues: choosing the right software technologies and the hardware architecture for the Web-based system. This is a non-trivial task that must be solved through an extensive analysis of the main alternatives at the software and hardware level.

2.3.3 Scalability of software technologies

From the point of view of software technologies, the real design challenge resides in the choice of the appropriate *application logic* because the HTTP interface and the information layer correspond to well established technologies. For the HTTP interface, Apache has become the most popular Web server [net05], followed by other products, such as MS Internet Information Services [Mic06a], Sun Java System Web Server [Sun06], Zeus [Zeu06]. Similar considerations hold for the information layer, that consists of a database

management system (DBMS) and some storage elements. There are many alternatives in the DBMS world, even though all of them are based on the relational architecture and some SQL dialect. The most common products are MySQL [MyS04] and PostgreSQL [pos05] on the open source side, MS SQL Server [Mic06b], Oracle [Ora06a] and IBM DB2 [DB206] on the proprietary side. Hence, choosing the technology for the information layer is basically a matter of cost, management, operating system constraints, internal competences, and taste.

The application logic at the application level, instead, is at the heart of a modern dynamic Web-based system. This level computes the information which will be used to build documents that are sent over a protocol handler. There is a plethora of software technologies which implement different standards. Each of them has its advantages and drawbacks with respect to performance, modularity, scalability. The most common technologies are *scripting* or *component-based*. Scripting technologies are based on a language interpreter that is integrated in the Web server software. The interpreter processes the code that is embedded in the HTML pages and that typically accesses the database. At run-time the script code is replaced by its output, and the resulting HTML is returned to the client. Static HTML code (also called *HTML template*) is left unaltered. Examples of scripting technologies include language processors such as PHP, ASP and ColdFusion. Scripting technologies are efficient for dynamic content generation, because they are tightly coupled with the Web server and do not require the dynamic activation of another process as it was required by CGI technologies. They are ideal for medium-sized, monolithic applications that need an efficient execution environment. On the other hand, the tight coupling between the front end and the application layer limits the use of scripting languages in Web-related applications requiring high scalability.

For this reason, a good solution for large-size sites are component-based technologies that implement the application logic through software objects. These objects are instantiated within special execution environments called

containers. A popular component-based technology for dynamic Web resource generation is the Java 2 Enterprise Edition (J2EE), which includes specifications for *Java Servlets*, *Java Server Pages* (JSP), and *Enterprise Java Beans* (EJB). Java Servlets are Java classes that implement the application logic for a Web-based system. They are instantiated within a *Servlet container* (such as Tomcat [tom05]) that has an interface to the Web server. The object-oriented nature of Java Servlets enforces better modularity in the design, while the possibility to run distinct containers on different nodes facilitates a system scalability level that could not be achieved by the scripting technologies. Java Servlets represent the building block of the J2EE framework. Indeed, they only provide the low-level mechanisms to serve dynamic requests. The programmer must take care of many details, such as coding the HTML document template, and organizing the communication with external information sources. For these reasons, Java Servlets are usually integrated with other J2EE technologies, such as JSP and EJB. JSP is a standard extension defined on top of the Java Servlet API, that allows the embedding of Java code in an HTML document. JSP is usually the default choice for dynamic, component-based content generation. EJB are Java-based server-side software components that enable dynamic content generation. An EJB runs in a special environment called *EJB container*, that is analogous to a Java Servlet container. EJB provides native support for atomic transactions that are useful to preserve data consistency through commit and rollback mechanisms. Moreover, they handle persistent information across several requests. An interesting performance comparison among scripting and component-based technologies is provided in [CCE+03]. This study compares the PHP scripting technology against Java Servlets and EJB for the implementation of a simple e-commerce site. Scripting technologies tend to reach their maximum throughput before component-based technologies, because of their more efficient execution environment. Hence, component-based technologies tend to perform badly on small-to-medium sized Web-based systems, but they scale better than scripting technologies and can reach even higher throughputs.

2.3.4 Scalability of hardware technologies

Once defined the logical layers and the proper software technologies that are needed to implement the Web-based system, we have to map them onto some physical nodes. Typically, we do not have a one-to-one mapping because many logical layers may be located on the same physical node, as well as a single layer may be distributed among different nodes for the sake of performance, modularity and fault tolerance. There are two approaches to map logical layers over the physical nodes, that we call *vertical* and *horizontal* replication. In a vertical replication, each logical layer is mapped to at most one physical node. Hence, each node hosts one or more logical layers. In horizontal replication, multiple replicas of the same layer are distributed across different nodes. Horizontal and vertical replication are usually combined together to reach a scalable and reliable platform.

Vertical three-node architectures have each logical layer on a distinct node, and are thus better suited to component-based technologies. For example, the J2EE specification provides inter-layer communication mechanisms that facilitate the distribution of the front end and the application layer among nodes. Scripting technologies do not natively provide these mechanisms, so they have to be implemented from scratch if the distribution of the layers is a primary concern. Three-node architectures help improving the performance with respect to vertical two-node architectures, in which front-end and application level are placed on the same physical node. Finally, vertical four-node architectures may be deployed into J2EE systems which distribute the application level among two physical nodes: one hosting the business logic (which is encapsulated into the EJB container), and the other hosting the application functions (through the JSP Servlet Engine). Whenever the advantages of EJB components are required it is convenient to adopt this architecture because of the typical high overheads of the EJBs.

Higher performance goals motivate the tendency towards the replication of nodes at one or more logical layers. This horizontal replication is usually combined with vertical replication, and this combination helps improving

important design goals such as scalability and fault tolerance, which are crucial to obtain an adequate level of performance and reliability. In particular, horizontal replication allows the use of dispatching algorithms that tend to distribute the load among the nodes in a uniform way [CCCY02, ACM02].

2.4 Web-based system performance testing

Once the Web-based system has been deployed and its functional correctness has been verified, it is necessary to check whether the performance constraints are satisfied or not. The performance testing of a dynamic resource-oriented Web-based system is a complex activity that requires different tasks, such as workload characterization, traffic generation, data collection and analysis. Performance testing is characterized by two main goals.

1. First, we have to check whether the consider Web-based system performs adequately. This means that the performance constraints are not violated, but also that the Web-based system works within safe margins of performance and that it does not show some dangerous trend that is, the tendency to increase suddenly its response time because of some close system bottlenecks. This goal is typically achieved through an evaluation of the *system capacity* for different workload models.

2. If the three types of expectations are not satisfied, some resource prevents that the system capacity achieves the expected performance or, in other words, there is a system bottleneck. In such a case, we have to carry out three main tasks: *bottleneck location*, *bottleneck causes identification*, *bottleneck removal*.

2.4.1 Workload specification

Understanding the nature of the workload which will reach the system is a key step for the testing of any infrastructure. When we consider Web-based systems this task is complicated by several internal and external problems.

Among the internal problems, we can evidence that modern Web-based systems are usually deployed by complex interactions of multiple software components running on distributed platforms where nodes are interconnected on a local and sometimes on a geographical scale. This leads to a combinatorial explosion of the possible interactions that can be triggered by each user request.

Among the external aspects, the main problems come from the characteristics of the client requests that pre-Web systems were not used to face. Nowadays, there is a huge literature showing that workload models are characterized by heavy tailed distributions, and Web traffic exhibits self similarity properties, heavy bursts of arrivals, frequent changes during an observation period.

A common way to represent the heterogeneous nature of the Web-based services is to define the entire set of classes of provided services, e.g., $\overline{S} = \{S_1, \ldots, S_n\}$, where each service is characterized by a different impact on Web-based system resources. For example, if we consider e-commerce Web sites, the most common services refer to *searches*, *product browsing* and *purchase*.

User requests for each service class are characterized by different frequencies $\overline{\Lambda} = \{\lambda_1, \ldots, \lambda_n\}$, where λ_i denotes the access frequency for the S_i service class. It is worth noting that even the definition of the impact on the Web-based system resources of one service class is not a trivial task because of two main reasons. First, we should consider that a user request for a Web resource (i.e., a user click on a URL) originates multiple client requests to the Web-based system. These are usually for the HTML base container and for the related embedded objects, where each embedded object can correspond to a static file or to a dynamically generated object that results from the interactions of several internal components and servers. Second, many commercially important applications, especially Web-based applications, are composed by a number of communicating components. These are usually structured as distributed systems with components running on differ-

ent processes and even different server nodes. For example, in a multi-level Web-based system, a client request can flow from the front end server to the back end server passing through the application logic. The impact of a user request involving dynamic content generation in a multi-level system is not easily predictable and even impossible to predict when we consider proprietary softwares. The conclusion is that considering only the λ_i access frequency is a simplification that neglects many fundamental details of the interactions between the client and the Web-based system, and the entire pattern of activated services behind each request. Hence, each λ_i should be further decomposed in sub-components λ_{i_j} where the number and characteristics of the j's depends on the hardware/software infrastructure supporting the Web-based services.

Nevertheless, we have to denote the workload models and to this purpose we usually rely on the description of the *workload mix*, which represents the popularity of each service class in client requests, and the *workload intensity*, which represents the load offered to the system. The definition of a workload mix is a difficult task on its own. The "typical" or standard workload model that reflects the prevalent client activities tends to disappear when we pass from traditional (prevalently static) to modern (prevalently dynamic) Web-based systems. The large majority of studies in the nineties was focused on Web-based systems providing static contents (e.g., [AW97, CB97]). Few results exist about the workload characterization of dynamic Web-based systems. Indeed, the large heterogeneity of dynamic and interactive Web-based services reduces the possibility of identifying invariants that may be acceptable for a large class of Web-based systems. Let us cite some studies about e-commerce sites, such as [ACC+02, AKR01], or about publishing sites [SWCK02]. Different efforts aim to characterize Web-based services through the definition of commonly accepted benchmarking standards about dynamically-oriented sites and e-commerce, such as TPC-W [TPC04] and SPECweb2005 suites [SPE05].

The common approach is to define multiple workload mixes $W_1, \ldots, W_m$

that represent the behavior of clients when they access each service for different scenarios. Each workload mix is defined as the vector of probabilities $\overline{W} = \{p_1, \ldots, p_n\}$ that a user request issues to each service. As the workload intensity is usually expressed through the frequency of user requests λ, where $\lambda = \sum_{i=1}^{n} \lambda_i$. Knowing the workload mix $\overline{W}$ for a scenario and the workload intensity λ provides us with all the information we need to describe a workload, because the knowledge of $\overline{W}$ and λ is equivalent to the knowledge of the vector $\overline{\Lambda}$. In other words, $\lambda_i = \lambda p_i, \forall i \in [1, n]$.

We should also consider that workload mixes, although assumed static for a while, actually vary continuously over time. Sometimes the patterns are predictable, other times they are not. This is a further problem that increases the complexity of the testing process of modern Web-based systems, but that is outside the scope of this study. As a consequence, the only modifiable parameter is the maximum capacity of the system. Hence, we point out the necessity to design architectures that are intrinsically scalable at the hardware and software levels. Only through a scalable system, we can easily adapt the infrastructure to guarantee the same performance constraints even in front of unpredictable and highly fluctuating values of λ. A decade of research on high performance Web-based systems [CCCY02, ACN03, IBM02, AMW+03, CCE+03] has achieved some important results that we consider as modern invariants for the Web infrastructures: component-based distributed software, cluster-based systems, multi-level architectures, as we described in Section 2.3.

2.4.2 Coarse grain times in a Web-based system

Let us consider a LAN-based Web system for content generation. A user request for a Web resource is typically processed by the browser in terms of multiple requests to the front-end server to get the HTML base container and its embedded objects. Most of the embedded objects are static files that can be directly served by the front-end HTTP server. Other objects are generated on the fly through one or multiple interactions with the application

layer and the back end layer. Each object is explicitly requested by the client application without further interaction with the user after the first click. Let us focus on these dynamic requests that imply the most complex interactions.

The *response time* T_r for a Web object consists of two main components: the network contribution T_{net} and the Web-based system contribution T_{sys}. This latter, in its turn, may be composed by one or multiple sojourn times in the three main system components: the *front-end time* T_{fe}, the *application server time* T_{as}, the *back-end server time* T_{be}[1].

Figure 2.3 shows the temporal diagram corresponding to a request for a dynamically generated resource. It evidences the main sojourn times caused by the three main *software components* of the Web-based system. The network time is typically independent of the type of request (unless it requires a secure channel), while the important parameter is the dimension of the transmitted information: when the client sends a request to the Web-based system, the request reaches the front-end server after the time T_{net_1} that is determined by the network conditions between the client and the front-end server nodes. After the Web-based system time T_{sys}, it is necessary to consider an additional network time T_{net_2}.

When the request is for a dynamic object, the front-end server activates the application server after a processing time T_{fe_1}. We can assume that the server at the application level handles the logic associated with the request with a T_{as_1} processing time. If the application server needs some data from the back-end level, then it must issue one or more queries to the database server. This server receives the first query, processes it in a T_{be_1} time, and returns the result to the application server that processes the received data (T_{as_2} time) and possibly issues other $n-1$ queries to the back-end server. When the application server has gathered the results from the back-end server ($T_{as_{n+1}}$ time), it passes all information to the front-end server that builds the

[1]For the sake of completeness we should also consider the time spent by each node to interact with the others, but we neglect it because it has always been demonstrated irrelevant with respect to the other components.

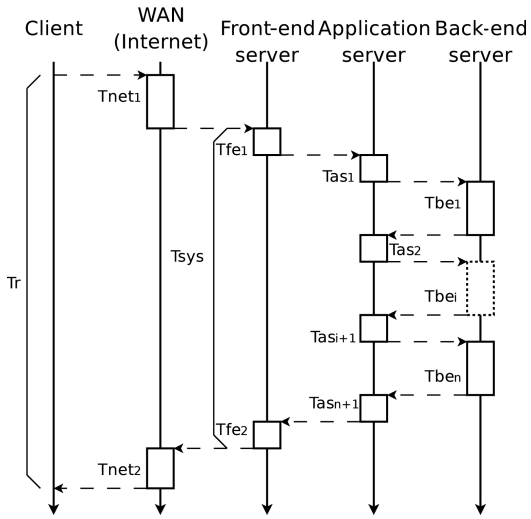

Figure 2.3: Temporal diagram of a request for a Web object to a Web-based system

HTTP response (T_{fe_2} processing time) and sends it back to the client.

Hence, the response time for a dynamic resource may be written as in Eq. 2.1, where n denotes the number of queries to the back-end server.

$$T_r = \sum_{i=1,2} T_{net_i} + \sum_{i=1,2} T_{fe_i} + \sum_{i=0,n+1} T_{as_i} + \sum_{i=0,n} T_{be_i}. \tag{2.1}$$

The total response time does not give any clue about possible system bottlenecks. But even the T_{net}, T_{fe}, T_{as}, T_{be} times considered as separate terms give a false impression of mutual independence. On the other hand, the components of the Web-based system are strictly correlated. For example, the application server relies on the back-end level to provide the necessary information to build the application logic data. If the back-end level fails or is slow, the performance of the application server may be severely degraded and, as a domino effect, the overall performance of the system drops.

2.4.3 Black box performance testing

Let us define *black box* performance testing the operations related to the evaluation of the capacity of the system that is subject to different workloads.

During the black box performance testing, we consider the whole Web-based system as seen from the outside, hence it is not necessary to analyze the internal parts. We can sample performance indexes, such as response time or throughput, outside of the Web-based system while it is exercised with each of its pre-defined workload models. The samples are typically coarse-grained and collected by the client emulator, through which we evaluate the Web-based system behavior.

The main goal of the black box performance testing is to check whether the Web-based system is able to meet the performance constraints with safe margins and safe trends for each workload model. This means that when the system is subject to the maximum expected request access frequency, it should not show signs of imminent congestion. This requirement is motivated by the observation that a Web-based system may meet all performance constraints, but with some critically utilized resource. A similar situation is unacceptable because the first burst of client arrivals may easily saturate the resource, thus slowing down the entire Web-based system. To avoid the risk of drawing false conclusions about the performance of the system, we also have to evaluate the performance trends as a function of different workload mixes and offered loads.

2.4.4 White box performance testing and bottleneck removal

During the black box testing, it is not necessary to consider the internal resources of the Web system. We can sample performance indexes outside of the Web system while it is exercised with each of the predefined workload models. On the other hand, when some expected condition is violated, understanding the motivations requires an accurate analysis of the behavior of the internal components of the Web system. We recall that addressing issues related to inadequate performance requires three main tasks:

- Bottleneck location

- Bottleneck causes identification
- Bottleneck removal

Let us define the first two tasks together as *white box* performance testing.

To pursue the goal of white box performance testing, we have to find the most appropriate *granularity level* for each resource metric. There exist many levels of granularity for each resource metric: *system*, *node*, *hardware resource*, *software component*, *process*, *function*. As a testbed example, we consider the system described in Figure 2.4, that has been implemented on two nodes by means of the PHP technology.

System-level metrics represent the overall behavior of the system and correspond to black box testing. An example is the whole response time T_r in Figure 2.4.

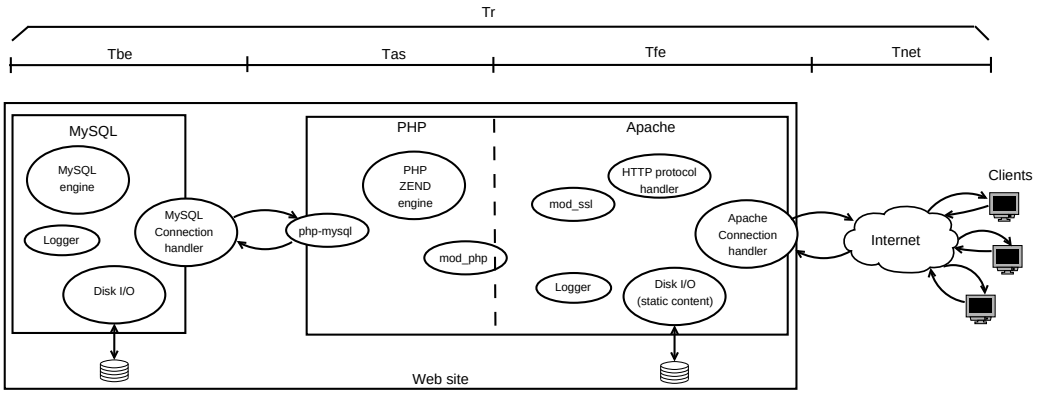

Figure 2.4: An example of PHP-based architecture of a Web-based system

Node-level metrics describe the behavior of a single, physical machine. In the architecture of Figure 2.4, T_{be} and $(T_{fe} + T_{as})$ are two examples of node-level metrics. At this granularity level, unexpected behavior at the single nodes can be spotted, but no clue is given about what is slowing down that machine.

To delve a deeper analysis, we have to consider at least the *resource-level* metrics, that are associated to the hardware and operating system resources of a node. Typical examples include: the utilization of the CPU, disk

and network interface that are gracefully degrading resources; the number of available file and socket descriptors, the amount of free memory that are token-based resources that may cause a sudden performance degradation. The vast majority of monitoring tools provide performance samples at this level of granularity. Indeed, they help to find the lacking node resource, but it is difficult or impossible to derive the motivation for sure by simply looking at resource-level metrics. For example, in a node hosting a database server, if the disk results as the node bottleneck, we can easily assume that this is due to some database operations, but we cannot know which component is causing the heavy I/O operations that are degrading the node performance. The problem becomes even worse when more software components run on the same node (which is usually the case with Apache and PHP). In these case separating the utilization of the different components is a difficult or impossible task, and we have to pass to a finer granularity.

The *component-level* metrics reflect the behavior of a software component, such as the HTTP server, the application server or the database server. The problem is that they are quite difficult to sample. Some metrics, such the response time, may even require modifications to the source code. On the other hand, component-level metrics usually give a detailed view of the system performance and permit to identify the critical components.

Even after the identification of the most critical component, when dealing with multiprocess or multithreaded applications, it is not easy to understand the motivations why this components is not performing according to the expected specifications. Often, it is necessary to go down to consider the *process-level* granularity. A typical example includes the response times of the single processes composing the application. The process-level granularity is of great help in spotting the critical areas, but the related metrics are almost impossible to collect with ordinary monitoring tools that are typically limited to the node-level granularity. The process-level metrics can be gathered by appropriately modifying the source code or by running more sophisticated performance analyzers, such as node profilers.

Some classes of software components, such as the database server, are so complex that even after the individuation of the critical area, it is difficult to *exactly* understand the hot spots in the code. To this purpose, it is necessary to resort to the finest granularity level, the so called *function level*, which relates to the main functions of each process (including the operating system). This level of granularity requires special instrumentation (node and kernel profilers [Lev04]). The latter cause some overhead on the system and yield data (call profile graphs) that are quite expensive to analyze. For this reason, the analysis of profiled data is usually performed off-line. On the positive side, the function-level metrics identify the critical areas of the software component in a very precise way.

The analysis is still not complete, because, once a bottleneck is identified, it has to be removed. The appropriate actions depend on the particular type of bottleneck. We may have a misconfigured system component that is under-performing, where a simple reconfiguration may help boosting its performance. Typical examples of *configuration bottlenecks* include the number of worker processes or the maximum number of TCP connections. We may find that one physical resource at a node (CPU, disk, network interface) is completely utilized. In this case, instead of reconfiguring the system, its capacity must be upgraded. There are two ways to improve the capacity of a component: *hardware upgrades* which simply augment the system capacity with the same number of node(s), and *hardware replications* which allow the system to distribute the computation among additional nodes.

A *software bottleneck* may be removed through a better design. The latter possibility is rather infrequent, because several application servers and the major back end servers are not shipped with the source code, thus its modification is not a viable solution.

It is clear that the broadest picture is taken when sampling all resource metrics at the function level. While the collection in itself may be executed with relatively low overhead through system kernel profilers, the real problem is the off-line analysis of call-graphs for thousands of different application-

level and OS-level functions. There is usually a trade-off between the granularity level of the resource metric and the completeness of the obtained samples. There is no need for sampling at the function-level when bottlenecks result obvious from the component-level metrics. However, in most instances it is rather easy to locate a bottleneck even at a higher level, but when we need to understand the motivations of that bottleneck we have to carry out the performance analysis at the finest granularity of the previous scale.

After the choice of the resource metrics, it is necessary to determine the most representative statistics for the considered samples. We do not want to enter into many details, but we should consider that average values are still common choices for many performance studies, even if the characteristics of the current Web-based systems (e.g., workload models characterized by quantities at different orders of magnitude, heavy-tailed distributions, burst arrivals, complex correlations between the software components) would require higher moments. The choice of cumulative distributions or percentiles instead of average values becomes even more important when we consider that some sites may be interested to provide services based on some *Service Level Agreement* (SLA). And it is quite obvious that the service levels of a complex multi-level system with large variances require statistics that are more representative than simple average values.

A new testing phase follows the bottleneck removal (involving black box and white box performance testing) to verify that performance and reliability attributes have been achieved. The whole procedure is a fix-and-test loop that may take several attempts to achieve the desired goals.

2.5 A case study for Web-based system performance testing

In this section we present a case study that illustrates the main steps to be followed when testing Web systems with performance constraints and unpre-

dictable access frequencies. We carry out black box and, when necessary, white-box performance tests on a real prototype with the goal of finding and removing system bottlenecks. Finally, we show a possible solution to remove some system bottleneck.

First, we show how the complete picture of the performance metrics can give useful information about the corrections that should be applied to the Web-based system to avoid bottlenecks or to improve its performance. Then, we evaluate the impact of wide area network dynamics on the system performance. This study allow us to show the necessity of considering in the analysis even the token-based resources that are often neglected, but that cause the worst performance problems when the pool is empty.

2.5.1 Workload model definition

The services offered by the dynamic Web-based system are modeled after the TPC-W benchmark [TPC04] of an online book store, where users are allowed to browse a product catalog and to purchase goods. In particular, we implemented the 14 different interactions $\overline{S} = \{S_1, \ldots, S_{14}\}$ that are defined in the benchmark workload model. Six services access the database in a read-only way: access to home page, listing of new products and best sellers, requests for product detail, two searches. Other eight interactions require an update of the database: user registration, updates to the shopping cart, two interactions involving purchases, two involving order inquiry and display, and two involving administrative tasks. We do not consider the Payment Gateway Emulator Service, which in TPC-W represents an external system authorizing payment of funds during interactions.

We implement the three workload mixes $W_{browsing}$, $W_{shopping}$ and $W_{ordering}$ that are built upon the services $S_1, \ldots, S_{14}$ and represent the *browsing*, *shopping* and *ordering* user activities. Table 2.1 shows the composition of workload mixes in terms of service access probabilities p_i.

A workload mix is generated through a *client emulator*, which creates a fixed number of client processes. Each process instantiates sessions consisting

Table 2.1: Composition of the workload models.

Service ID (S_i)	**Service Class**	**Browsing Mix** (p_i)	**Shopping Mix** (p_i)	**Ordering Mix** (p_i)
S_1	Buy confirm	0.007	0.01	0.10
S_2	Execute search	0.10	0.16	0.12
S_3	Search request	0.12	0.20	0.14
S_4	New products	0.11	0.05	0.005
S_5	Customer registration	0.009	0.03	0.13
S_6	Admin request	0.0009	0.0007	0.001
S_7	Order inquiry	0.003	0.008	0.003
S_8	Product detail	0.19	0.17	0.11
S_9	Admin response	0.0006	0.0007	0.001
S_{10}	Buy request	0.008	0.02	0.13
S_{11}	Order display	0.002	0.007	0.003
S_{12}	Home interaction	0.31	0.17	0.11
S_{13}	Best sellers	0.11	0.05	0.005
S_{14}	Shopping cart	0.02	0.11	0.14

of multiple page requests to the dynamic Web system. Emulated sessions last 15 minutes on average, as from the TPC-W specification. Before initiating the successive request, each emulated client waits for a specified think time of 7 seconds on average. The sequence of requests is emulated by a finite state machine that specifies the probability to pass from one Web transaction to another.

We run different tests for increasing Web page request frequencies λ's until the system shows clear sign of performance degradation. Since our main goal is to guarantee an adequate operation margin, we focus more on the performance trend rather than on absolute performance values for the performance constraints.

2.5.2 Architectural testbed

Figure 2.5 shows the architecture of the prototype system. The experiments are carried out in a clustered environment of nodes running the Linux operating system (kernel version 2.6.8). Each node is equipped with a 2.4GHz

hyperthreaded Xeon, 1GB of main memory, 80GB ATA disks (7200 RPM, transfer rate 55MB/s) and a Fast Ethernet adapter. All nodes are connected through a Fast Ethernet switch.

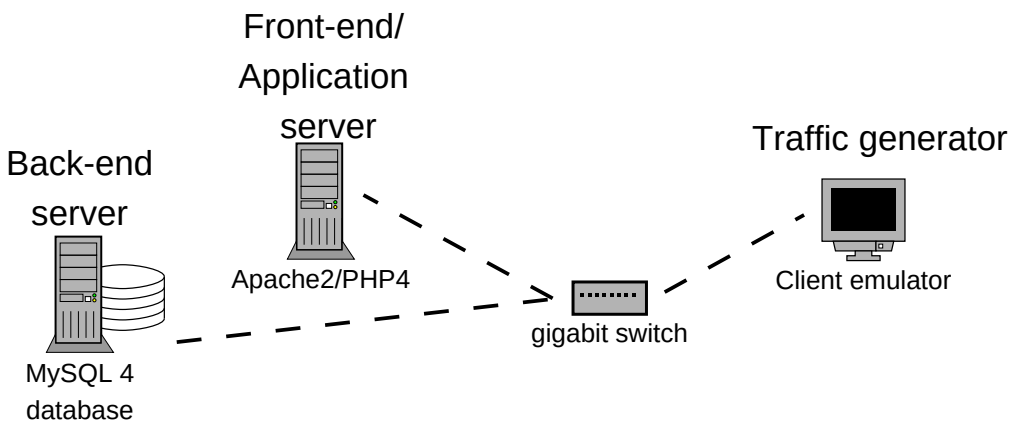

Figure 2.5: Architecture of the testbed for the experiments

One node runs both the Apache [Apa06] Web server (version 2.0.52) and the PHP4 [PHP04] engine, which is used to implement the scripts at the application layer. The MySQL database server [MyS04] (version 4.0.20) runs on a different node. To reflect a realistic workload scenario, we enabled the support for table locking and two phase commits. Data collection is performed through monitoring tools at the node level (the *system activity report* [God04]) and at the function level (oprofile [Lev04]).

To take into account the effects of wide area networks, we instrumented the *netem* packet scheduler [Hem04] that creates a virtual link between the clients and the Web-based system with the following characteristics: maximum link bandwidth 8Mbit/sec, packet delay normally distributed with $\mu = 200ms$ and $\sigma = 10ms$, packet drop probability of 1%.

The first set of experiments is carried out in a local environment without taking into account WAN effects, which will be discussed in greater detail in Section 2.5.7.

For all the experiments presented in this section we use as workload model the *browsing mix* described in Table 2.1, since in other not reported experi-

ments we found similar trends with the other two workload models.

2.5.3 System capacity evaluation

First, we focus on system level measures (system response time T_{sys} and the system throughput) to find the capacity of the system. Figure 2.6 shows the system throughput (in served objects per second) as a function of the client population. Saturation occurs around 250 clients when the system is able to serve approximately 400 objects per second.

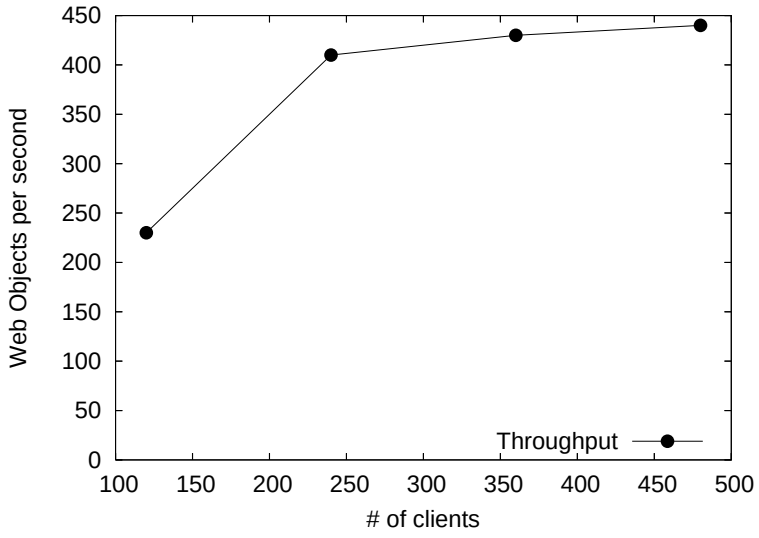

Figure 2.6: Throughput of the Web-based system

We also consider cumulative distributions and percentiles that are more representative of the system behavior in the context of Web-based information systems. For example, in Figure 2.7 we show the average and the 90-percentile of the system response time for a single Web object. The growth of the 90-percentile around the knee is much more evident (from 0.075 seconds at 240 clients to 0.68 seconds at 360 clients) than that of the average response time (from 0.0759 seconds at 240 clients to 0.322 at 360 clients). Indeed, mean values may tend to underestimate the explosion of the response time for increasing client populations.

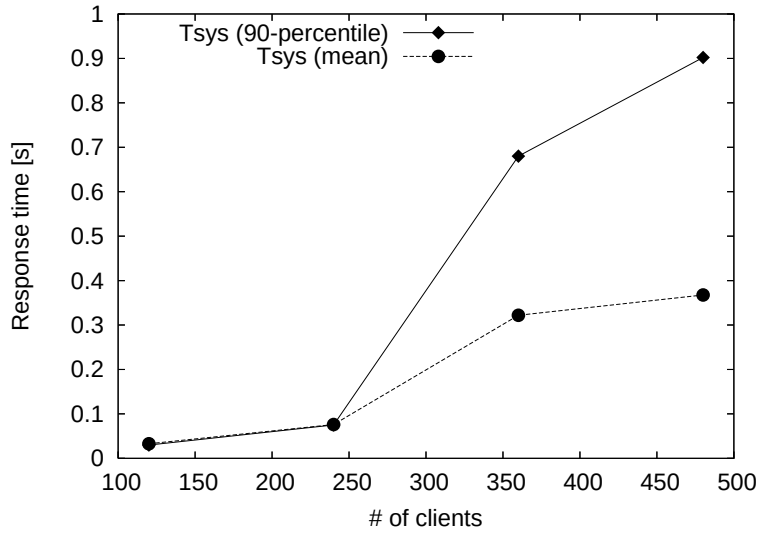

Figure 2.7: Response time of the Web-based system

Black box testing results are fairly easy to collect, but they only allow for congestion detection. For example, they do not provide any information about the causes that lead to poor performance. Thus, it is necessary to investigate the system at a finer grain level to find out where and why congestion occurs.

2.5.4 Bottleneck location

Let us pass to consider the contribution to the system response time of the two nodes that compose the considered Web-based system: one hosting the Apache Web server and the PHP component, the other running the MySQL server. Node-level measurement allows us to identify where the major part of the response time is spent. Furthermore, we can easily identify the node that gets overloaded by observing which node-level response time explodes first. Figure 2.8 shows the 90-percentile of the system response time T_{sys} and the contributions of the two nodes T_{fe-as} and T_{be} (logarithmic scale).

There is no doubt that T_{be} represents the predominant factor of the system response time T_{sys}. Hence, with the considered workload oriented to dynamic

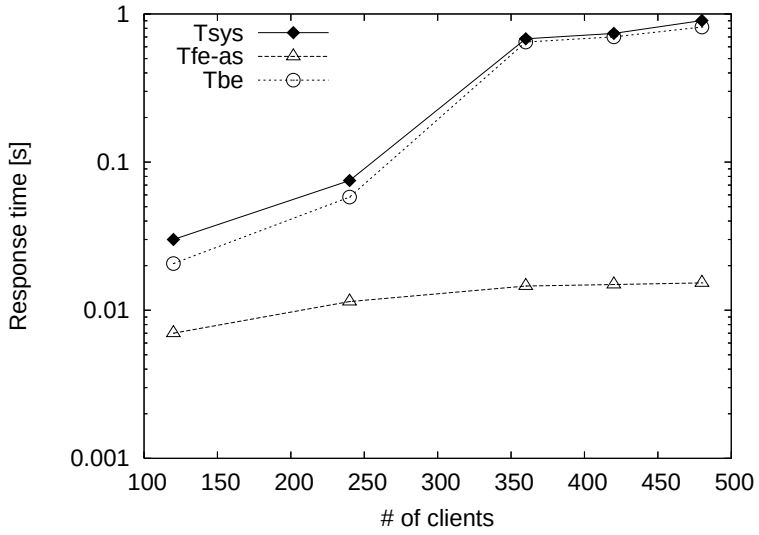

Figure 2.8: 90-percentile of the contributions to the system response time at the node level

requests, the bottleneck that limits the performance of the system is on the back-end node.

At this level of granularity we are still missing most of the information on the possible interventions on the system to avoid the congestion. A common approach to better identify the potential bottlenecks is to move to a finer granularity level such as the resource level.

2.5.5 Bottleneck causes identification

We use metrics such as CPU, disk and network utilization collected on each node that allow to evidence the hardware resource of the back-end node that is over-utilized with respect to its capacity.

Figures 2.9 and 2.10 show the CPU utilization of the back-end node during the experiments carried out with a population of 240 and 360 clients, respectively. The two horizontal lines represent the mean values. CPU utilization is bursty at 240 clients (Figure 2.9), while it bumps to 100% for the majority of time when there are 360 clients (Figure 2.10). Table 2.2 shows

the average resource utilization of the database node for different client populations. In particular, user-space and kernel-space measures are separated to better identify the source of the problem (application computations or system calls?). To complete the picture, we also report the utilization of the disk and the network interface. This table confirms that the CPU utilization is extremely high, while disk and network seem underutilized.

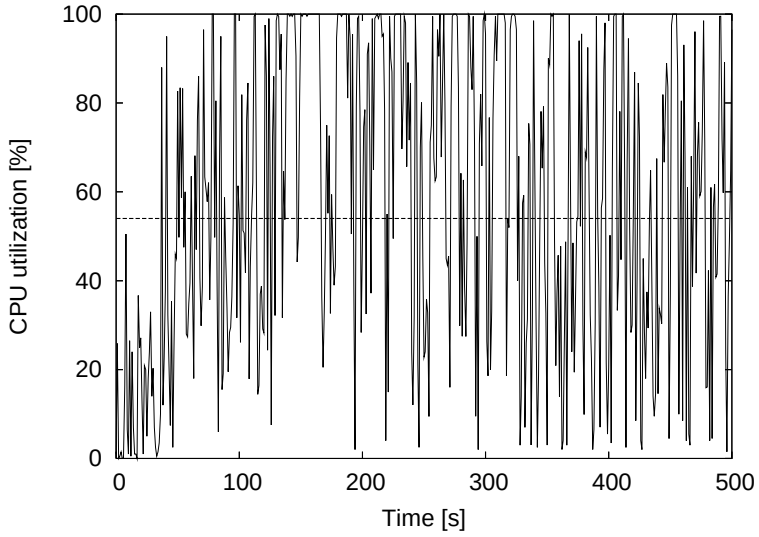

Figure 2.9: CPU utilization of the back-end node (240 clients)

The low disk activity is an initially unexpected result that we can explain due to the database size which in large part fits in the main memory of 1Gbyte. This aspect is interesting because with the hardware improvements even at the entry level, it is becoming common for medium-size e-commerce databases to fit for a large part in the main memory. As a trend result, we can conclude that the disk activities may not be the most significant component to understand the behavior of the back-end node hosting the database server. A similar result has been obtained in [ENTZ04].

If we consider the CPU utilization for 360 clients (Table 2.2), we recognize a 80-20% ratio between the time spent in the user and kernel space. This initially unexpected result suggests that the application level computations are much more intensive than the cost necessary for the system calls. As there

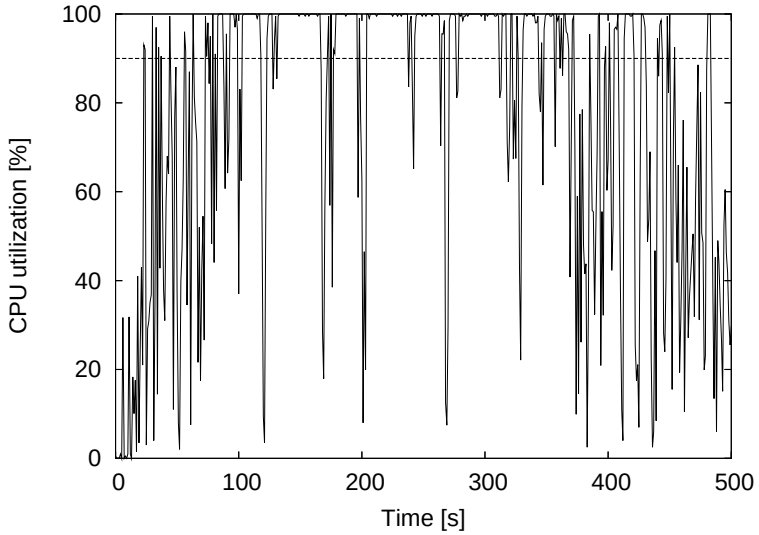

Figure 2.10: CPU utilization of the back-end node (360 clients)

# of clients	CPU utilization		disk utilization	Network utilization
	user	kernel		
120	17%	4%	0.10%	0.018%
240	51%	9%	0.12%	0.019%
360	76%	14%	0.15%	0.020%

Table 2.2: Hardware resource utilizations

is only one major process running on the back-end node, we can easily assume that the *mysqld* server process is the source of the bottleneck. However, if we limit the analysis at this granularity level, we cannot exactly motivate the high CPU utilization of the database server application.

Hence, to identify the hot spots in the database server process we pass to get measurements at the function level that represent our finest grain. The profiler output shows more than 800 *mysqld* functions, hence a detailed analysis is quite difficult and even useless. The idea is to focus on the functions the use more CPU time, while we aggregate the others that are not significant for a performance study. Figure 2.11 shows the percentages of CPU utilization that are used by the main functions of the *mysqld* process that is, tuple

management, I/O management, *buf_page_is_corrupted()*, others. The result of the pie is clear and unexpected: the function *buf_page_is_corrupted()* that checksums asynchronous I/O buffers uses almost 70% of the CPU time. This result is not obvious at all from the measurements carried out at the hardware resource level, hence some technical details may be useful. This function is part of the asynchronous I/O buffer management of MySQL. Asynchronous I/O is used to improve I/O performance by caching frequently accessed portions of the database thus bypassing the operating system disk buffer cache. To provide data consistency a checksum is calculated on every MySQL buffer through the *mysqld buf_page_is_corrupted()* function. We can conclude that the asynchronous I/O subsystem is the real bottleneck of the *mysqld* process.

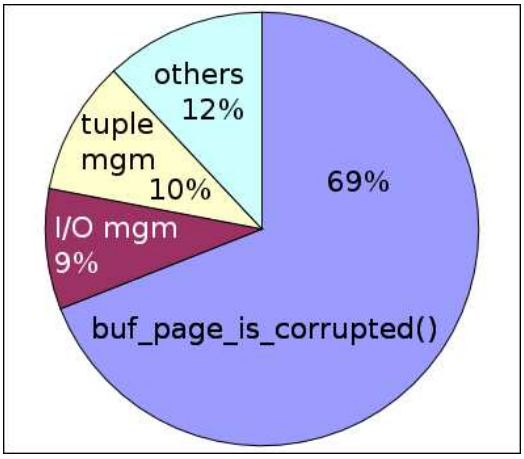

Figure 2.11: CPU utilization percentages at the function level (database process)

2.5.6 Bottleneck removal

Let us summarize the result of the bottleneck analysis: the asynchronous I/O operations on the DBMS require more CPU power than what is currently available. The goal of the bottleneck removal task is to understand which is the most appropriate solution among the possible interventions to address this issue: hardware upgrade, hardware replication and system tuning. Many

considerations may affect the choice including available budget, technical know-how, and the expected scalability improvement.

The first interesting solution is to tune the parameters of the DBMS system to reduce the asynchronous I/O activity, which is the cause of the bottleneck. For example, we can increase the size of the query cache to improve system performance by reducing I/O operations. After the tuning operations, we should carry out another black box performance testing to evaluate the effects and whether the improvement satisfies the expectations. In the negative case, another campaign of white box tests is necessary because, after system tuning, the system bottleneck may have changed. Figure 2.12 reports the cumulative probabilities of the response time of the back-end node before and after the intervention. This figure proves the validity of our deduction: the 90 percentile of T_{be} drops from the original 0.646 seconds to 0.273 seconds after the database query cache tuning. A similar improvement is reflected on the system response time T_{sys}.

Scale-up or hardware upgrade and scale-out or hardware replication are hardware interventions that become necessary when system tuning does not show significant improvements. In our test case, if we augment the CPU power and/or the CPU cache size of the back end node, we can improve performance to a limited extent because of the intrinsic upper bound of the hardware technology (even in terms of cost/performance for the Web system provider). Hence, the most effective solution from the point of view of maximum scalability of the system is the horizontal replication of the back end node. After having doubled the number of back end nodes through the *clustering* function available in the development branch (4.1) of MySQL, we found a significant performance improvement for the overall system. Indeed, the scalability of the two nodes solution is almost linear and augments the maximum capacity of the system by a factor of about 1.7. This improvement is worth the effort of doubling the back end nodes.

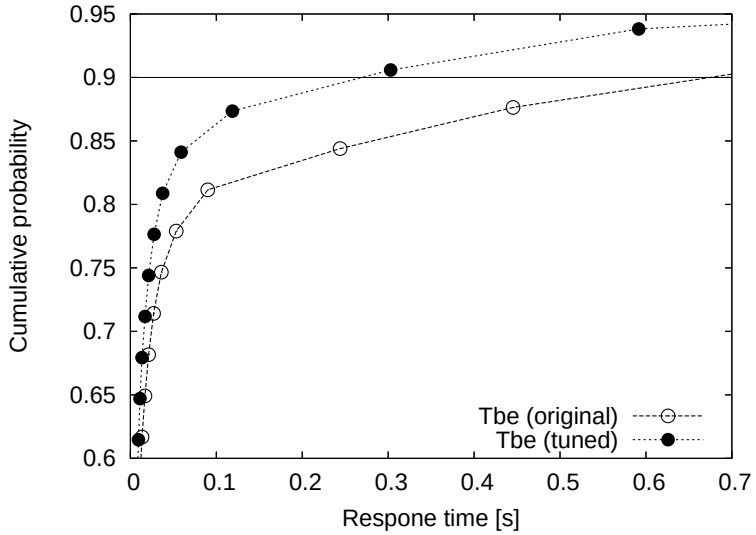

Figure 2.12: Cumulative distribution functions of the back-end response time before and after the system tuning operation

2.5.7 Impact of wide area network effects

The goal of this series of experiments is to evaluate the impact of WAN effects on the performance of the Web-based system. The initial motivation for this study comes from the results of Nahum *et al.* [NRSA01] that were among the first authors to describe the details of the wide area network effects on the behavior of Web server systems. The results of this section extends their work from a single layer Web-based system to a multi-tier Web-based system. The motivation also stems from the users' perception of navigation, which includes network delays. Since the main goal of the whole thesis is to provide the users with good quality services, we cannot neglect their point of view.

The testbed and workload model is the same of the previous experiments, with the addition of wide area network emulation between the clients and the front-end node. We focus on a client population of 240 clients and outline the impact of wide area network characteristics on the previously obtained results. We denote the scenario without and with wide area effects through the terms *no-WAN* and *WAN*, respectively.

System capacity evaluation

We first compare the response time T_r for the two network scenarios. Figure 2.13 shows the cumulative distribution of the response time of one dynamic request to the Web-based system T_r and the network delay T_{net} for the WAN scenario. The 90-percentile of $T_r(WAN)$ is equal to 2.5 seconds. From the previous experiments we also have that the 90-percentile of the system response time for the no-WAN scenario is equal to 0.075 seconds. The difference of two orders of magnitude is too large to be motivated only by the introduction of the network delays. Indeed, the network delay has a 90-percentile below 1.5 seconds, which cannot explain the 90-percentile of 2.5 seconds for $T_r(WAN)$, when the $T_r(no-WAN)$ counterpart is below 0.1 seconds.

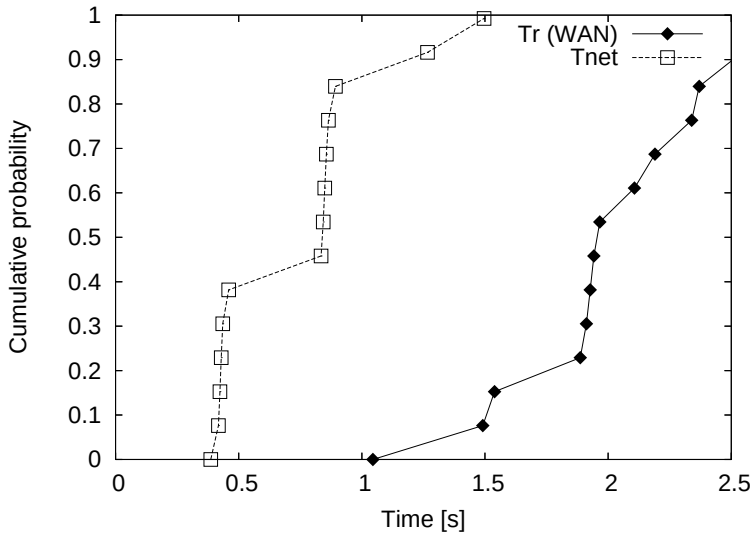

Figure 2.13: Cumulative distributions of the response time T_r and the network time T_{net} (WAN scenario)

Bottleneck location

To look for other motivations we pass to a finer grain analysis of the system. To this purpose, we show in Figures 2.14 and 2.15 the contributions of the

three main software components to the response times. In the comparison, it is necessary to consider the different scale on the x-axis.

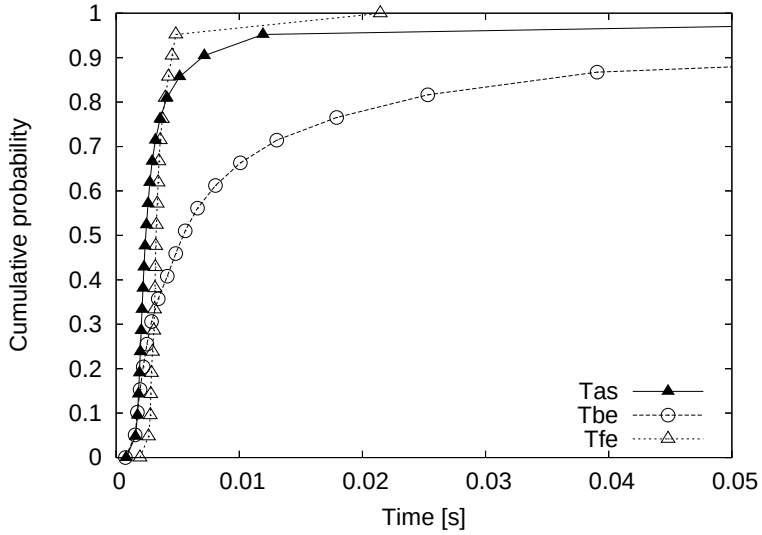

Figure 2.14: Cumulative distributions of the components of T_{sys} (no WAN scenario)

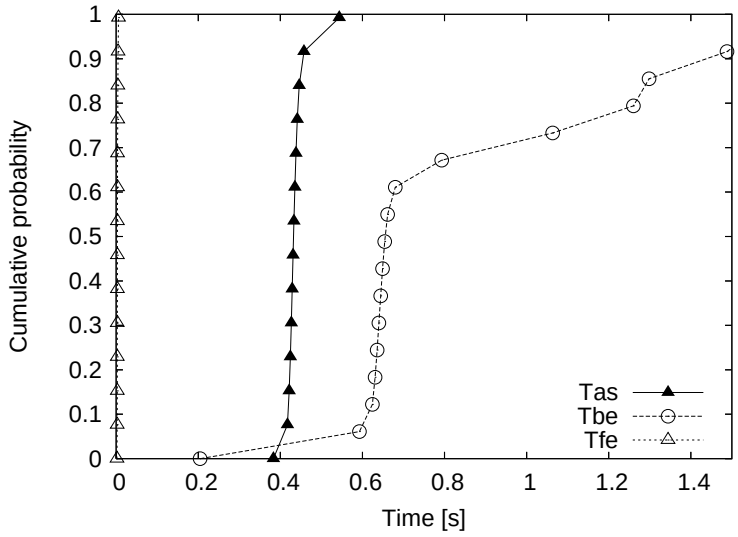

Figure 2.15: Cumulative distribution function of the components of T_{sys} (WAN scenario)

While the contribution of the front-end server T_{fe} remains low for both scenarios (the 90-percentile is equal to 0.0045 and 0.005 for the no-WAN and WAN scenario, respectively), the contributions of the application and back-end server grow substantially with the introduction of the WAN effects. In particular, the 90-percentile of T_{as} passes from 0.008 to 0.46 seconds, and the 90-percentile of T_{be} passes from 0.055 to 1.5 seconds. The poor performance of the database and the application server in a WAN scenario can be explained through a finer granularity analysis.

Bottleneck causes identification

Figure 2.16 shows the number of open sockets during the experiment in the WAN and no-WAN scenarios. From this figure, it is immediate to get two results: the number of sockets simultaneously used by the database server is three times higher in the WAN scenario (on average, 45 vs. 128); the sockets are a limited set that in this experiment is fully utilized (128 is the default number in the MySQL configuration file). The growth in socket needs is explained by considering that connections between the application server and the database server last much longer in the WAN scenario due to the network slowdown on client requests. Sockets are token based resources that are not gracefully degradable. This means that once the number of available sockets is exhausted, further requests to the database server are queued. The contention for access to the limited pool of available sockets connecting to the database further increases the concurrency level leading to an amplification of the phenomenon that is similar to a thrashing event. The macroscopic effect of socket shortage is the poor performance of the application server and back-end server components.

Bottleneck removal

In this case also a system tuning could help in gaining better scalability, for instance by augmenting the maximum number of simultaneous connection in MySQL configuration file. As previously stated, a node replication is

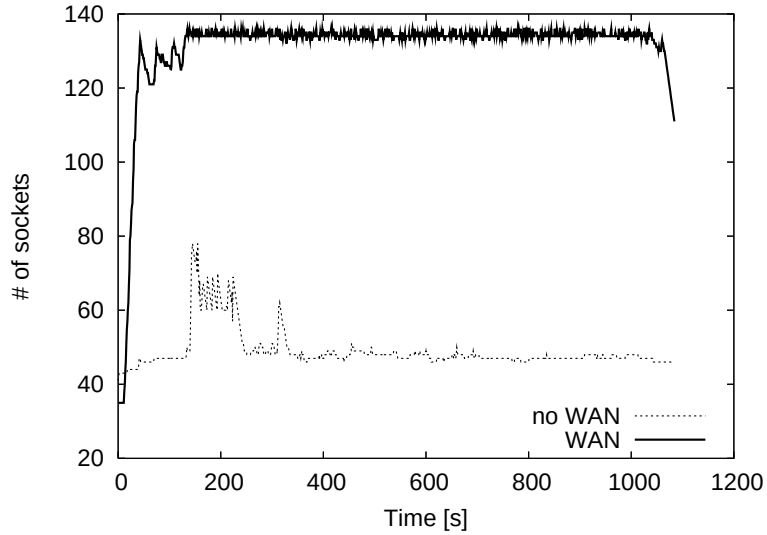

Figure 2.16: Number of utilized sockets by the back-end server in the WAN and no-WAN scenarios

probably the best solution to get a sure scalability.

2.6 Summary of the results

Due to their growing popularity, Web-based systems are vulnerable to the increase in the volume of client requests, which can hinder both performance and reliability. Thus, it is fundamental to design Web systems that are inherently scalable, so that they can face sudden flash crowds.

When testing such systems, we notice that percentiles and cumulative distribution functions are much more suited than average values because the workloads show an heavy tailed behavior.

We discuss the trade-off between analysis complexity and results richness, that is we want to give an answer to the question about when it is necessary to go further into deeper granularity levels or if the current level is enough to understand what can be done to improve performance.

Finally, in the analysis that takes into account network delays we evidence two important results. In a Web-based system there are many token-based

resources that are often neglected in the performance studies: ignoring these resources may have serious consequences. There is a high interdependence among the multiple components of a Web-based system, hence a bottleneck on one component can easily be reflected on other parts of the system with an amplification effect that sometimes makes difficult to understand the initial cause.

Part II

Efficient Web content delivery

Chapter 3

Web content caching and delivery

3.1 Introduction

In the second part of the thesis we focus on the caching infrastructure that is located between the generation level and the adaptation level, as shown in Figure 3.1. For the sake of simplicity, we assume clients directly connect to this infrastructure.

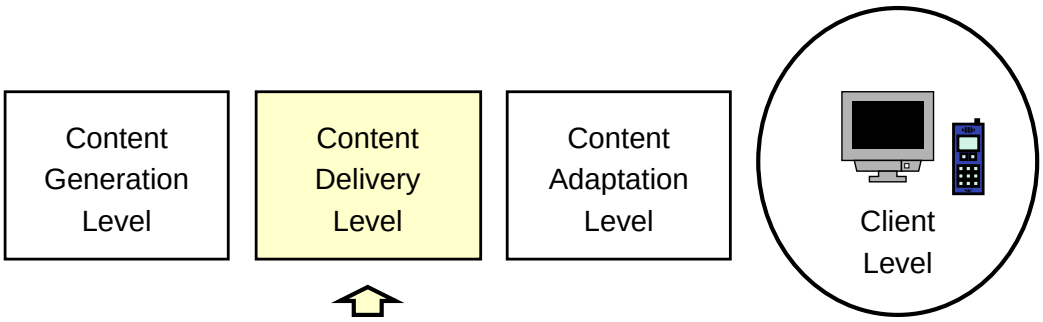

Figure 3.1: The delivery level of the Web-based infrastructure

The main goal of this level is to allow a better content delivery through the use of a caching mechanism, that brings the origin resources "closer" to the end-user.

Indeed, one of the most popular approaches to efficient Web content delivery is the use of Web caching to improve the scalability of the Web. Web Caching is very popular and widespread. Van Jacobson points out that it is the only known way to address the exponential growth of the Web [Jac95].

Caches leverage the well known principle of references locality. There are two flavors of locality [Wes01, Dav01]: *temporal* and *spatial*. Temporal locality means that some pieces of data (some Web resource, in case of Web caching) are more popular than others. Spatial locality means that requests for certain Web resources are likely to occur together. Caches use locality of references to predict future accesses based on the past accesses. When the prediction is correct there is a significant performance improvement. Some studies [Zip99, CSD+95] suggest that most data processing (including the Web) shows some form of locality.

In this thesis we focus on systems where the caching and the replication of Web resources are carried out by a third party that is independent of both the content provider and the Web user.

It is important to note that we examine distributed systems of cooperating edge servers because intermediate solutions relying on one node are inherently non scalable. For Web caching the initial idea of using one edge server shows very low cache hit rates even in a context of homogeneous users. For this reason the solutions proposed in literature [CSD+95] aim to establish interactions among various peers. *Global caching* or *cooperative caching* architectures are used mainly by public organizations (e.g., IRCache [IRC95], NLANR [NLA02]) and Internet Service Providers (ISPs, e.g., AT&T [ATT02]).

After a first period of prevalent enthusiasm towards cooperating edge servers, the research community is exploring in a more systematic way the real benefits and limitations of cooperation. While there is no doubt that Web caching improves performance when applied to a limited working set [KWZ01] (for example, successful CDN companies apply some sort of Web caching only to content of their customer sites), the debate is open over the benefits of

cooperation when applied to the entire Web content.

The most important reasons for setting up some form of cooperation among the edge servers are:

- Content lookup (*cooperative discovery*),
- Content delivery,
- Content placement (*cooperative pre-fetching*) and removal (*cooperative replacement*).

In this chapter, we focus on cooperative schemes for resource discovery and delivery that are traditionally based on two classes of "pure" protocols coming from the theory on distributed systems: *query-based* [WC97b, WC97c, WC97a] and *directory-based* protocols [RW98, FCAB98]. We claim and demonstrate that any "pure" protocol is affected by some or all of the following drawbacks:

- limited scalability,
- the cooperation overhead increases more than linearly for higher number of cooperative nodes,
- the performance (especially cache hit rates) is unstable because too much dependent on the workload/system characteristics.

In the following chapter we propose and evaluate two innovative hybrid cooperation protocols (and a variant of both of them) that are based on a semi-flat organization of the edge servers.

Any cooperation scheme among a set of distributed edge servers has to define a protocol to exchange some local state information. The two main and opposite approaches to deliver state information are well defined in literature: *query-based protocols* in which exchanges of state information occur in response to an explicit request by an edge server, and *directory-based protocols* in which state information is exchanged among the cooperating nodes in a periodic way or at the occurrence of a significant event, with many possible variants in between.

3.2 Related work

Cooperative Web caching has been studied for a long period. Cooperation can be used for different goals, but here we are mainly interested to resource discovery, retrieval and delivery. The first idea for cooperation was the use of a hierarchy based on the physical Internet topology [CSD+95]. In this approach, the phases of discovery and delivery are simultaneous. However, hierarchical cooperation shows multiple drawbacks especially for the lookup latency [RSB99], and management issues [PH97].

Some of the problems of hierarchical caching can be solved by organizing the edge servers in a flat topology. Typically, in this approach the two actions of discovery and delivery are separated, as in the systems considered in this thesis. Document retrieval for delivery is extremely simple and efficient, by requiring only an HTTP connection to the selected edge server or to the origin Web server. On the other hand, discovery can be performed according to different schemes (namely distributed lookup schemes). In pure distributed schemes, the lookup process takes place in one step and is basically founded on query-based or directory (summary)-based mechanisms. In our hybrid distributed schemes, which are described in the next chapter, the lookup process takes place in two separate steps and is founded on a combination of query-based and summary-based mechanisms.

ICP [WC97c] is the most popular query-based cooperation protocol and its pros and cons have been widely studied. The limited scalability of ICP has been documented in [WC97b] by the author of the protocol himself, while its high overhead for cooperation was one of the main motivation for the proposal of a summary-based scheme, such as Summary Cache [FCAB98]. In [RW98], Russkov et al. also note that ICP is slow in handling misses, thus motivating the proposal of Cache Digests, another well known summary-based protocol. Multicast has been proposed (for example [WC97a] for ICP) to reduce cooperation overhead. However, this technique has serious limits to be applied to a geographically distributed system, because by now multicast addresses cannot usually cross the boundaries of autonomous systems.

In our research [LMC03] we intensively tested these "pure" protocols and found a confirmation of the fact that they are not scalable and their performance are highly dependent on workload and traffic characteristics. ICP because of the high overheads it puts on the network, Cache Digests because of its low hit rates.

There are different opinions about the benefits of cooperative Web caching when applied to a large scale. Wolman et al. claim that the benefits are small because the gain in terms of hit rate increase are very small and tend to saturate [WVS+99]. On the other hand, Dykes and Robbins in [DR01] and in [DR02] claim that the most important benefits of Web caching are the reduction of the variability of user response time and the reduction of the number of pathologically long delays and this is more important than the hit rate increase. Our scalable protocols provide a viable solution to address the issues evidenced by Dykes and Robbins, which were not taken into account in [WVS+99].

As "pure" hierarchical and distributed approaches were found unsuitable for cooperation over a large scale, hybrid schemes have been proposed. Here, a hierarchical approach is combined with other forms of cooperation among sibling edge servers (that is, nodes sharing the same parent edge server). *Hybrid scheme* is a quite generic term that is usually referred to any cooperation mechanism that does not fit in the category of pure hierarchical and pure distributed cooperation. However, there are many possible combinations that derive from the lookup topology and the cooperative protocol. For example, an all-query-based scheme is proposed in [RSB01], where traditional hierarchical caching is integrated with horizontal query-based cooperation among cooperating edge servers at the same level of the hierarchy. Tewari et al. [TDVK99] suggest a summary-based hierarchical approach by means of meta-data that track where copies of files are stored in the hierarchy. A similar technique is proposed by Povey *et al.* [PH97], where only the lower level edge servers are responsible for storing documents, while upper level nodes maintain information about the contents of the lower level edge servers. The

idea of a multi-step lookup over sets of different cardinality is related to the idea of two-step cooperation over a two-tier architecture proposed in this thesis. The main difference with the results from our contribution is that the previous proposals rely on one "pure" lookup protocol, that is either query-based or directory(summary)-based, whereas we propose various innovative hybrid combinations of "pure" protocols.

CRISP is another hybrid scheme in terms of protocols, proposed by Rabinovich et al. [GRC97, GCR98]. It combines a centralized directory-based protocol with a query-based approach. On the other hand, all the schemes proposed here rely on a distributed summary-based protocol combined with a distributed query-based protocol. Indeed, the authors of CRISP observed that a centralized directory makes their architecture not scalable over a geographic network environment, although they outlined some alternative solutions for scalability [RCG98]. In particular, the two-step lookup process and node clustering choices have some similarities with our two-tier architecture, and especially with the InterQ-IntraS-DM scheme. The good performance of InterQ-IntraS-DM can be considered a demonstration of the validity of the ideas contained in [RCG98].

Another important topic related to Web caching is the definition of measures of quality for Web caching and cooperative Web caching. Unlike many other studies that use a limited set of measures, we consider all the main metrics appeared in literature, such as cache hit rate, response time, overhead for cooperation. In all previous studies, object (or byte) hit rate is the main metric used to evaluate performance of a caching architecture. More recently, several authors have pointed out the importance of considering other performance metrics. For example, Fan et al. [FCAB98] introduce network overhead as a measure for system scalability, and the impact of cooperation on central memory and CPU usage. However, present architectures do not seem anymore to represent a bottleneck for edge servers with the exception of mass storage size for nodes holding large resources such as video data [LASV02]. The user response time is another important measure of

cache performance. For example, Russkov et al. [RW98] use this parameter to demonstrate that Cache Digests is preferable to ICP. In our study the user response time is the most important metric to evaluate the performance of the hybrid protocols in a geographic environment.

Finally, we consider important to observe that our prototype experiments based on Web Polygraph [RW00] allow us the highest flexibility on the choices of workload characteristics. We carry out important sensitivity analysis that are prevented to the large majority of papers based on trace-driven simulations. Indeed, we are aware of few sensitivity analysis of cooperative caching performance to workload characteristics. Williamson et al. focus on traditional proxy caching [WB01], while some results on cooperative Web caching is in [LASV02].

3.3 Performance evaluation of pure cooperation protocols

We extensively evaluate the performance of the "pure" schemes in a flat topology under different conditions. In order to provide a better insight of the scalability characteristics of the cooperation schemes under test, we use in our performance evaluation a simplified workload. We call this workload "ideal" in the sense that it favors object caching.

In order to provide a valid workload model, we base our model on the workload used for the second cache-off by IRCache [IRC95]. Such workload model is characterized by high *recurrence* (that is, the probability that a resource is requested more than once), and small inter-arrival times for client requests. Requests are referred to a mix of content types consisting of images (65%), HTML documents (15%), binary data (0.5%), others (19.5%). The workload model defines a set of *hot* resources (15% of the working set) that receive about 15% of the requests.

In the Ideal Workload the object cacheability is always 100%, the object lifetime is assumed infinite (there is no chance of stale hit), and there is a

high probability (0.87) that an object is requested more than once.

The two main performance results considered here are the *global cache hit rate* (Global HR) and *overhead for cooperation* (expressed as an average of the additional bytes transmitted per each delivered document). There are many parameters that can influence the performance of the cooperative protocols, and most of them are considered here: number of peers, cache capacity with respect to the working set, local cache hit rate determined by the characteristics of the workload, frequency of client request arrivals to the edge servers.

3.3.1 Limits of query-based schemes

Query-based protocols are conceptually simple. When an edge server experiences a local miss, it sends a query message to all of the cooperating edge servers[1] in order to discover whether one of them caches a valid copy of the requested resource. In the positive case, the recipient edge server replies with a hit message, otherwise it may reply with a miss message or not reply at all. This query-based solution was proposed by the Harvest project [CSD+95], and then implemented in NetCache [Dan97] and Squid [Wes02]. The last two use the Internet Cache Protocol (ICP) as their query mechanism [WC97c].

Let us summarize the main pros and cons of the query-based class of cooperating protocols. On the positive side, they have a high cache hit rate that, in a local area, is nearly insensible to the number of involved edge servers, to the cache capacity, and to the frequency of client requests (the experiments of which we are reporting the results are described in great details in [LMC05]). The overhead for cooperation is the main drawback of this class of protocols that is evidenced by our experiments. The number of bytes required for cooperative document discovery increases more than linearly when the number of cooperating edge servers augments. This result occurs for any kind of workload, and it represents a limit to the scalability and stability of query-

[1] Some implementations keep records of the node state, so the query is not sent to overloaded or temporarily off-line edge servers.

based protocols. Increasing the number of cooperating edge servers leads to scalability problems because there is an increment of both the number of ICP queries and the number of recipients of each query. The scalability and stability problems of query-based protocols are even more serious when they are applied over a geographic context. For example, in [FCAB98] it has been shown that cooperation overhead may become unacceptable even with less than 10 cooperating edge servers.

The overhead for cooperation is the main drawback of query-based schemes that is evidenced by our experiments. From Table 3.1 we see that the cooperation overhead (expressed as the number of bytes required for cooperative document discovery) increases more than linearly when the number of peers augments. This result occurs for any kind of workload, and it represents a limit to the scalability and stability of query-based protocols. Increasing the number of cooperating edge servers leads to scalability problems because there is an increment of both the number of queries and the number of recipient of each query. The last column of Table 3.1 confirms this: when 30 nodes are involved, the traffic for coordination is more than one third of the traffic generated by client requests. We also carry out a simulation using a simulator derived from ns-2. Through the simulator we evaluate systems with up to 60 nodes. Our simulation results confirm the more than linear growth in overhead for query-based cooperation schemes are the number of edge servers increases. Increasing the cache size can alleviate this problem because the local hit rate augments, so less client requests starts a cooperative lookup process. However, recent studies indicate that the working set seems to increase faster than the typical cache capacity [LASV02].

The scalability and stability problems of query-based protocols are even more serious when they are applied over a geographical context. For example, in [FCAB98] it has been shown that cooperation overhead may become unacceptable even with less than 10 cooperative peers. Other limits to the scalability of query-based protocols are found in literature. One of the most interesting, described in [RW98] is related to the mechanism used to detect

Nodes	Fraction of cache capacity per node	Local HR	Global HR	Overhead [$\frac{\text{bytes}}{\text{req.}}$]
8	11%	58.92%	88.20%	474
15	5%	42.43%	88.07%	1520
30	3.5%	34.14%	87.71%	3583

Table 3.1: Cache size per node [% of working set] for query-based cooperation schemes

cache miss. During cooperative lookup, the first received hit message is used to locate the requested resource on a peer. To detect a miss, instead, the edge server must wait for a reply by all peers or for a time out. The consequence is that in query-based protocols cache misses are much slower to be detected than cache hits. Our experiments in Section 4.5.3 confirm this conclusion. Increasing the number of peers makes this problem even worse, especially if cooperation occurs among distant nodes. Moreover, the probability of packet loss or delay is increased as the number of peers augments, in a way proportional to $1-(1-P_{err1})^M$ [PP01], where P_{err1} is the probability of packet loss or delay when only one packet is sent and M is the number of peers. In a geographic environment, where packet delay and loss are more likely, the response time of query-based protocols augment because of the higher numbers of cache misses.

3.3.2 Limits of directory- and summary-based schemes

Directory-based protocols are conceptually more complex than query-based schemes, especially because they include a large class of alternatives, of which the two most important are: the presence of one centralized directory or multiple directories disseminated over the cooperating edge servers; the frequency to communicate a local change to the directory/ies. It is impossible to discuss here all the alternatives that have been the topics of many studies. We limit our analysis to distributed directory-based schemes, because it is common opinion that in a geographically distributed system any centralized

solution does not scale: the central directory server may represent a bottleneck and a single point of failure, and it does not avoid the query delays during the lookup process.

In a distributed directory-based scheme, each edge server keeps a directory of which URLs are cached in every other cooperating node, and uses the directory as a filter to reduce the number of queries. Distributing the directory among all the cooperating nodes avoids the polling of multiple edge servers during the discovery phase, and in the ideal case makes object lookup extremely efficient. However, the ideal case is affected by large traffic overheads to keep the directories up-to-date. Hence, real implementations use multiple relaxations, such as compressed directories (namely, *summary*) and less frequent information exchanges to save memory space and network bandwidth, respectively. Our experiments in [LMC05] show that the cache hit rate of the summary-based schemes is heavily dependent on many parameters. In particular, it diminishes as the number of cooperating edge servers increases, and even when the working set augments more than the cache capacities. Moreover, we found a (stronger than expected) correlation between the cache hit rate of summary-based protocols and the frequency of client request arrivals. On the positive side for this kind of protocols, we have that the overhead for cooperation is really low and almost independent of the number of nodes: the traffic generated for cooperation is even three orders of magnitude lower than that caused by query-based protocols.

Unlike query-based schemes that have rather stable cache hit rates, our performance evaluation shows that the cache hit rate of the summary-based schemes is heavily dependent on many parameters. In particular, it diminishes as the number of peers increases, and even when the working set augments more than the cache capacities. Our tests with the same Ideal workload previously described (with a frequency of 10 client requests per second) show that the global hit rate decreases from 50% to 33% when the cooperative peers pass from 8 to 30 (fourth column of Table 3.2). This instability is also due to the acceleration of the exchanges of objects between the nodes when

Nodes	Fraction of cache capacity per node	Local HR	Global HR	Overhead $[\frac{\text{bytes}}{\text{req.}}]$
8	11%	31.83%	50.23%	3.31
15	5%	20.99%	37.01%	4.34
30	3.5%	18.79%	33.80%	5.41

Table 3.2: Cache size per node [% of working set] for CD protocol

the total working set augments more than the cache capacity (second column of Table 3.2).

Moreover, we found a (stronger than expected) correlation between the cache hit rate of Cache Digest and the frequency of client request arrivals. In Table 3.3 we report the impact of capacity miss on hit rate when 11% of the content can be held in each of the 8 edge servers used in the experiments. This table shows that the reduction of the cache hit rate when the frequency of client requests increases is significant: from 76% to 50%, when all other parameters are kept constant. The dependency of the Cache Digests hit rate on the number of cooperating peers and on the frequency of client request arrivals are confirmed by all our experiments. The sensitivity of hit rate to both client request frequency and number of peers is a double effect of the same problem (out-of-date summary information) caused by capacity misses and consistency misses, respectively.

Capacity misses occur when a directory references an object that the edge server has previously discarded to make space to other objects. Larger numbers of cooperative peers augment the working set because more clients are served. If the capacity of each node remains the same, the replacement algorithm tends to be activated more often, with consequent higher numbers of capacity misses. In our experiments, each node can contain from 11% to 3.5% of the entire working set. As shown in the second column of Table 3.2, the capacity misses due to replacements reduce the hit rate when the peers augment.

Client request frequency $[\frac{\text{req.}}{\text{sec.}}]$	Global Hit Rate
10	50%
5	72%
2	74%
1	76%

Table 3.3: Global Hit Rate of Cache Digest as a function of the frequency of client requests

Consistency misses are caused by the asynchronous propagation of summary contents that further reduce the accuracy of the directories, especially when the frequency of client request arrivals is much greater than that used for summary refresh.

On the positive side for the summary-based schemes, we have that the overhead for cooperation is really low and almost independent of the number of nodes: the traffic generated for coordination is even three orders of magnitude lower than that caused by the query-based scheme (compare the last column of Table 3.2 against that of Table 3.1). Moreover, this cooperation overhead is quite stable with respect many parameters: it is mainly a function of the cache capacity, because increasing the capacity augments the size of the digests being exchanged. Dependency on other parameters, such as digests rebuild frequency and number of edge servers, is much less evident.

3.4 Summary of the performance evaluation

We can summarize the results of the previous performance evaluation as follows:

- Query-based schemes are very effective in finding hits in remote edge servers

- The overhead associated with query-based schemes is very high and increases more than linearly with the number of cooperating nodes, this reducing the scalability of query-based schemes
- Summary-based schemes present a suboptimal object hit rate because of less than perfect accuracy in the exchanged information. This effect of reduced hit rate is very sensitive to client request frequency
- The overhead of summary-based schemes is more than one order of magnitude lower than the one of query-based schemes

We demonstrated that "pure" cooperation architectures have several drawbacks, that is:

- Flat query-based schemes produce a high overhead that increases more than linearly with the number of nodes
- Flat summary-based schemes have poor cache hit rate due to out-of date information exchange. Moreover this effect is more evident as client request frequency increases

It seems then interesting to investigate whether an architecture combining both summary- and query-based schemes can provide better and more stable performance [SCCQ02, LCC03, LMC05].

Chapter 4

Two-tier distributed architectures for Web caching

4.1 Introduction

The key idea behind the proposed two-tier architecture is to provide a lookup mechanism that can carry out a discovery task over a set of edge servers without the need to contact every node in the set. We already observed that flat architectures do not follow this approach: in summary-based cooperation every node must have some knowledge about the content of the neighbor cache, while in query-based cooperation the query and response message exchange involves every node. The performance analysis carried out in the previous chapter suggests that a lookup process involving every node leads to poor scalability. On the other hand we can achieve high scalability by providing a lightweight lookup mechanism that contacts only a subset of the nodes and yet is able to locate resources also on nodes not directly involved in the lookup.

This idea brings the need of a two-tier organization of the edge servers and we also need a protocol to take advantage of this two-tier topology. This idea influences both the node topology organization and the lookup process.

- On the topological side, we first partition the set of edge servers on

the basis of physical characteristics of the node interconnections, by creating *groups* of well-connected nodes. Then, we choose representative edge servers for each group and we put these representatives all together, thus creating what we call a *representative subset*.

- The lookup process is divided in two steps that may include a group or a representative subset.

We consider solutions for resource discovery based on an architecture where edge servers are logically organized in a two-tier topology ($N = 2$) that uses hybrid cooperation protocols combining query- and summary-based schemes.

The choice of a two-tier scheme derives from the observation that typically ISPs deploy caches at the points of presence (POPs), and corporate networks deploy caches within each of their locations. Moving the lookup process beyond $N > 2$ has been demonstrated not to be convenient because more than two steps tend to lengthen the response time [TDVK99], thus making often convenient to get the resource from the origin Web server.

To describe the two-tier logical organization, let us take as example the nine edge servers in Figure 4.1.

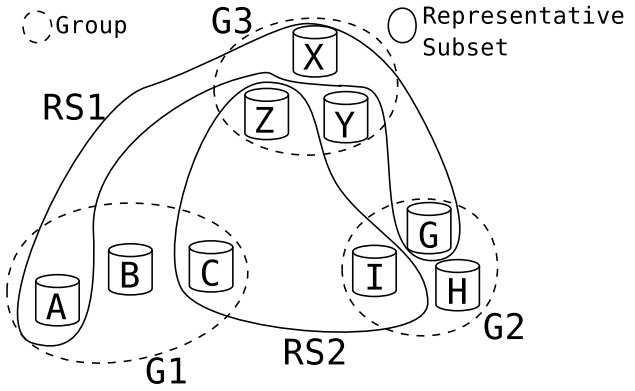

Figure 4.1: Example of a two-tier architecture: groups and representative subsets

A *group* is a subset of "well connected" edge servers in terms of distance

and network connectivity. The connectivity requirements for WAN network applications subject to variable traffic conditions are still open problems, although there are studies that demonstrate that routing in the Internet core tends to be more stable than that observed some years ago [McM99, KO00]. Independently of these issues, we can easily accept any edge server partition that achieves intra-group connections with typical lower latency, lower congestion and higher bandwidth than inter-group connections.

For instance, in Figure 4.1, we could define the following groups: $G_1 = \{A, B, C\}$, G_2=$\{G, H, I\}$ and G_3=$\{X, Y, Z\}$.

A *representative subset* is a subset of edge servers that includes one representative for each group. We create representative subsets in order to obtain an exchange of information only among the representatives of each group. That is, we do not want to have a global information exchange. We already observed that pure protocols do not follow this approach: in summary-based cooperation every edge server must have some knowledge about the content of the neighbor cooperating nodes, while in query-based cooperation the query and response message exchange involves every edge server. The previous chapter suggests that a lookup process involving every edge server leads to poor scalability. On the other hand, we can achieve high scalability by providing a lightweight lookup mechanism that contacts only a subset of the edge servers and yet is able to locate resources also on cooperating nodes that are not directly involved in the lookup. Since there are multiple nodes in a group, each of them can be chosen as a representative for its group. Hence, given a partition of the nodes in groups we may have one or multiple representative subsets. For instance, in Figure 4.1, we define two representative subsets at the same time: RS_1=$\{A, G, X\}$, RS_2=$\{C, I, Z\}$.

Cooperation protocols for two-tier topologies

Two-tier architectures allow us to split the cooperative resource lookup process in two separate phases called *lookup tiers*. In this way, the first attempt is done by contacting only a few servers, thus bringing a small overhead on

the network. Then, only if it is necessary, other edge servers are contacted as well. In any case, we want to avoid the need of contacting each cooperating server, because this would lead to poor scalability. Instead, we introduce a new way of collaboration among the edge servers, that requires to contact only the representatives to know whether any edge server owns the requested resource. *Intra-group cooperation* occurs inside a group, while *inter-group cooperation* occurs among the groups by contacting their representatives.

Many protocols can be used as first-tier and/or second-tier lookup protocols, such as query-based and summary-based. A summary-based protocol exchanges small amounts of traffic for cooperation and guarantees low latency times for lookup (just a search in a table in main memory), but we have verified that its cache hit rates are highly sensitive to network and workload characteristics. On the other hand, a query-based protocol requires larger amounts of traffic for cooperation and has higher latency times, but its cache hit rates are high and typically more stable for different workloads.

If we consider protocols that combine query- and summary-based mechanisms on a two-tier topology, it seems useless to apply the same query-based (or summary-based) protocol for both first and second tier because it would result in a "pure" query- (or summary-) based protocol, that is not of interest for our study.

The doubt that we cannot solve through the previous qualitative considerations is about the most convenient combination of query- and summary-based protocols as intra- and inter-group cooperation protocol. In the next section, we consider hybrid cooperation schemes that combine in different ways summary- and query-based protocols.

4.2 Two-tier cooperation protocols

In this section we consider two innovative hybrid protocols and a variation of both of them: InterQ-IntraS, InterQ-IntraS-DM and InterS-IntraQ with its variant. Each protocol name denotes the two schemes that are used for

intra- and inter-group cooperation. For instance, *InterQ-IntraS* denotes the protocol that uses a query-based and a summary-based schema for inter- and intra-group cooperation, respectively.

Let us introduce some definitions we use throughout the chapter. A client request reaching an edge server of a two-tier cooperative architecture may experiment different effects. It may result in a *local hit* when a valid copy of the requested resource is found in the first contacted edge server: the resource is sent to the client and no cooperation is activated. Otherwise, a *first-tier hit* occurs when the requested resource is found in an edge server by means of the first-tier lookup. A *global hit* occurs, after a *first-tier miss*, when the resource is found in an edge server by means of both tiers of cooperation. Finally, we have a *global miss*, if the resource must be retrieved from the origin Web server, because both lookup tiers fail in finding a valid copy of the resource in any edge server.

4.2.1 InterQ-IntraS cooperation protocol

The InterQ-IntraS scheme uses a summary-based protocol based on Cache Digests [RW98] for intra-group cooperation (the "Summary" part), that is inside a group, and a query-based cooperation protocol for inter-group cooperation (the "Query" part), that is when contacting the representative nodes. The main advantage of this scheme is that any edge server is informed about the content of all the cooperating edge servers of the same group. Hence, it seems convenient to limit the query-based cooperation on the second tier to one representative server for each group, called *group master*. Typically, we select the edge server offering superior computing power and better connections to the other groups to act as a master. Masters are edge servers that can be directly contacted by clients, as well as the other edge servers. To describe this protocol we consider the example of the architecture shown in Figure 4.2.

Let G_1 denote the group $\{A, B, C\}$, G_2 the group $\{G, H, I\}$ and G_3 the group $\{X, Y, Z\}$. The three nodes $\{C, I, Z\}$ compose the representative sub-

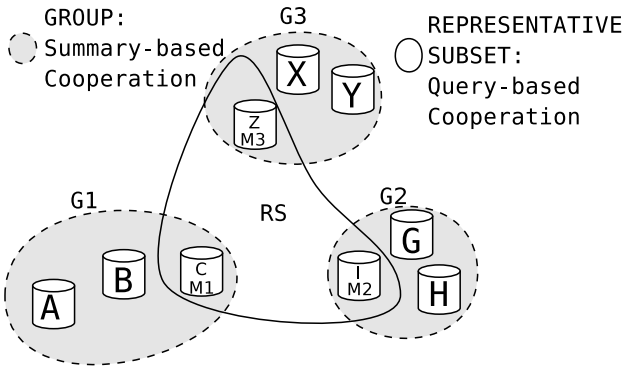

Figure 4.2: Configuration example of a system using InterQ-IntraS cooperation

set in which the second-tier lookup is carried out through a query-based protocol. To better evidence the group masters, we have added the labels M_1, M_2 and M_3 to the nodes C, I and Z, respectively. As you can imagine, there are only two scenarios:

1. A client requests a resource by contacting a non-master server.

2. A client requests a resource by contacting a master server.

The former scenario is described in Figure 4.3. The client issues an HTTP request (step 1) for a given resource to a non master server, for instance A in Figure 4.3. A searches in its own cache. If it finds the requested resource, it sends the copy to the client. This phase does not require the activation of any cooperation scheme. If A does not find a valid copy of the resource in its cache, then the first-tier lookup is activated (step 2): A looks up the digests to examine the content of the other edge servers belonging to its group. If A discovers the requested resource in B, then A sets up a TCP/IP connection with B to retrieve it (step 3). If the download is successful, A forwards the resource to the client (step 4). If B cannot return the resource or if the resource cannot be found in any edge server belonging to the A group, A activates the second-tier lookup (step 5) by sending a query to its own master (M_1 in Figure 4.3). In its turn, M_1 sends an ICP query (step 6) to each of

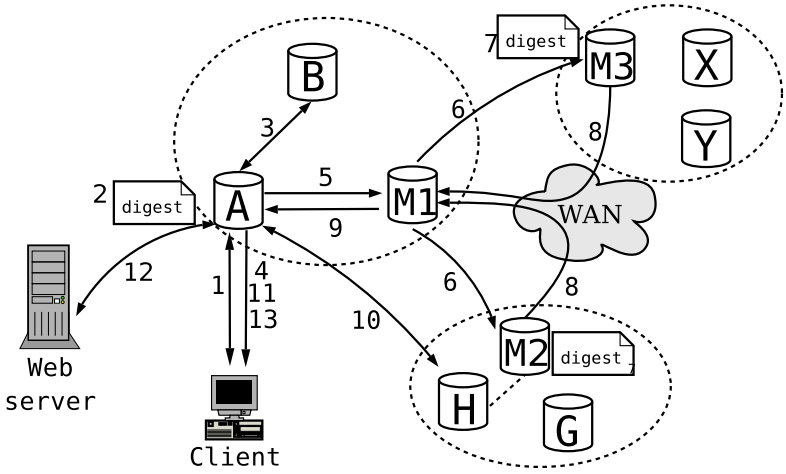

Figure 4.3: InterQ-IntraS cooperation. A client requests resources to a non master server

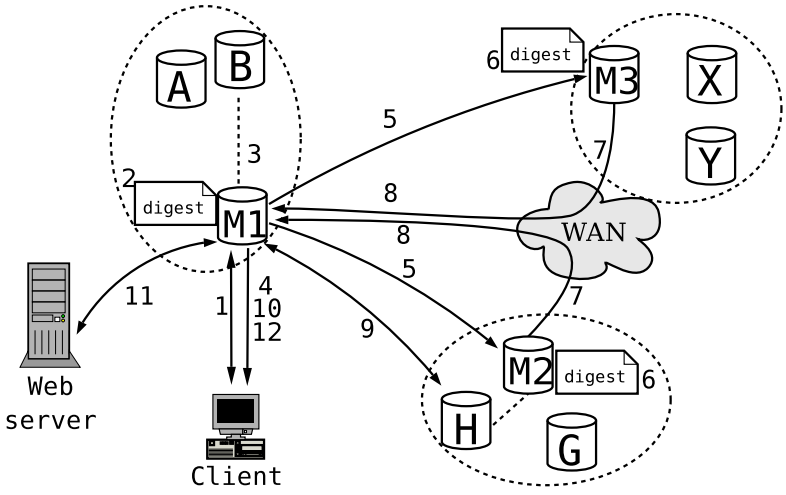

Figure 4.4: InterQ-IntraS cooperation. A client requests resources to a master server

the other masters (M_2 and M_3 in Figure 4.3). The masters look up in their digests (step 7) and answer (step 8) to the requesting master (M_1) whether the resource can be found in any edge server belonging to their group. Let us suppose, for instance, that M_3 answers with a miss message, indicating none of its group edge servers owns a copy. On the other hand, M_2 answers with a hit message, indicating that an edge server (e.g., H in Figure 4.3) owns a valid copy of the requested resource. Then M_1 sends a message to A (step 9) containing the address of the server which resulted in a hit. A sets up an HTTP connection with H (step 10) to download the requested resource. If the resource retrieval is successful, A forwards it to the client (step 11), otherwise A contacts the origin Web server (step 12) and then forwards the resource to the client (step 13).

The scenario where a client requests a resource directly to a master server (e.g. M_1) is described in Figure 4.4. In this case, M_1 looks up in its cache (step 1) a valid copy of the resource. If there is a miss in the local cache, M_1 uses the digest search algorithm of its group to look for possible remote hits (step 2). Two possibilities exist:

1. There is a possible hit in its group, for instance in B. Then M_1 sets up a TCP/IP connection with B (step 3); it fetches the resource and forwards it to the client (step 4).

2. There is no hit in its group. The master sends an ICP query to the nearest masters (step 5). The successive behavior is similar to that described in the previous scenario (Figure 4.3) from step 7 to step 13, keeping in mind that in Figure 4.4 the corresponding steps are numbered from 6 to 12.

A variant of the InterQ-IntraS protocol

The previous InterQ-IntraS scheme can achieve high cache hit rates, but it places a significant amount of work on the group masters. Each of them

has to serve client requests and, at the same time, has to act as a gateway for query-based cooperation. Especially when masters do not have computational capacity higher than that of the other edge servers, the twofold role of edge and master server can easily congest these nodes and lead the entire cooperation system to perform poorly.

To address this bottleneck issue, we propose the idea of using *dedicated masters* that do not act as edge servers. This approach denotes a different cooperation architecture, called *InterQ-IntraS-DM*, that works similarly to the previous InterQ-IntraS scheme. The main difference is that now each master acts as a gateway for its group and as a directory for the other groups, but it cannot be directly contacted by clients. The limited congestion in master nodes is done at the expenses of a reduced number of edge servers available for client service.

4.2.2 InterS-IntraQ cooperation protocol

The motivation for the *InterS-IntraQ* scheme comes from the observation that in the InterQ-IntraS schemes the large majority of traffic for cooperation (due to ICP messages) transits through the inter-group links, that are potentially slower and more expensive (for ISPs) than intra-group links. The idea behind *InterS-IntraQ* is to achieve a twofold effect: inter-group cooperation through Cache Digests that generates a minor traffic; intra-group cooperation, where nodes are connected through faster and less expensive links, based on an enhanced version of ICP messages. As a further positive effect of the InterS-IntraQ scheme, we can indicate that it bypasses the bottleneck related to the masters, and it can achieve a better load balancing because it uses multiple representatives of each group for the requests.

Figure 4.5 shows an example of this cooperation scheme by considering the same architecture shown in Figure 4.2. The three *groups* are $G_1 = \{A, B, C\}$, $G_2 = \{G, H, I\}$, and $G_3 = \{X, Y, Z\}$. We also define the following *representative subsets* for summary-based cooperation: $RS_1 = \{C, I, Z\}$, $RS_2 = \{A, H, Y\}$ and $RS_3 = \{B, G, X\}$. Each representative subset must

contain at least one member of each group. In this version of InterS-IntraQ cooperation protocol, we require that the groups G_i and the representative subsets RS_i denote a partition of the initial set of nodes. This leads to the conclusion that the maximum number of representative subsets is equal to the minimum cardinality of the groups. Additionally, the minimum number of nodes inside a representative subset RS_i is equal to the number of groups in the system, because at least a member of each group must belong to a representative subset.

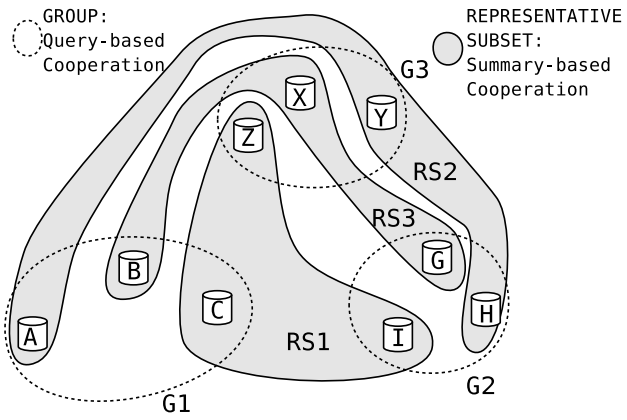

Figure 4.5: InterS-IntraQ architecture

We describe this cooperation scheme by referring to Figure 4.6. When an edge server receives an HTTP request from a client (step 1), it first searches in its local cache (step 2). If the requested resource is present, it is forwarded to the client (step 3), otherwise the first-tier lookup is activated. The edge server analyzes the digests of the nodes belonging to its representative subset (step 4). If any of the first-tier cooperating nodes owns the resource (e.g., I in Fig. 4.6), then the server fetches it (step 5) and forwards it to the client (step 6). On the other hand, if the first-tier lookup comes out as a first-tier miss, then the second-tier lookup is activated and ICP queries are sent to the group nodes (step 7). It is important to observe that these nodes do not answer only for the content of their caches. Since they also know the digests of their representative subset nodes (step 8), every node can answer for its

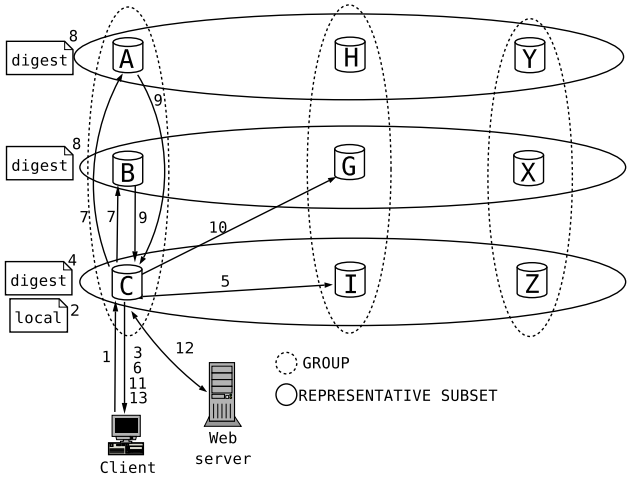

Figure 4.6: InterS-IntraQ cooperation protocol: answer to an HTTP query from a client.

whole representative subset (step 9). For example, A in Fig. 4.6 answers for itself, for H and for Y. If any of the cooperating nodes owns the resource, the edge server activates an HTTP connection (step 10) to fetch it and forwards it to the client (step 11). Otherwise, if no node has the resource, or it cannot be retrieved, then the edge server reaches the origin Web server (step 12), fetches the requested resource and forwards it to the client (step 13).

A variant of the InterS-IntraQ cooperation protocol

A possible variant of the InterS-IntraQ scheme comes from the observation that the first-tier summary-based cooperation is activated among the "far" servers of a representative subset through a Cache Digest lookup, while, if necessary, ICP queries are sent to the "near" edge servers belonging to the group of the contacted server.

We may think to an alternative protocol that first looks for the resource in the nearby edge servers through ICP, and then in the far servers through a Cache Digest lookup. The convenience of this variant with respect to the InterS-IntraQ protocol comes from the observation that when a resource is present in both a far and a near edge server, it is fetched from the closer

node.

The InterS-IntraQ protocol and its variant have the same response time in the case of local hit because no cooperation is activated, and in the case of local and group miss, because the resource has to be retrieved from a far edge server and it does not matter whether the Cache Digest lookup occurs before or after the ICP queries. The two protocols have different response times when there is a hit in the representative subset and in the group of the contacted edge server, because the InterS-IntraQ protocol fetches the resource from far servers, after a Cache Digest lookup, while the variant fetches the resource from near servers, after an ICP query.

We do not implement this variant because from our data we found that there are few cases when a valid resource is both in a near group and in a far representative subset. Moreover, the response times depend on many parameters and it is unclear whether it is convenient to pursue a reduction of the lookup time (InterS-IntraQ) or of the fetch time (variant). We carried out a sensitivity analysis to evaluate the trade-offs between the InterS-IntraQ protocol and its variant. Figure 4.7 denotes the boundaries of the region where the InterS-IntraQ protocol performs better than its variant. In the regions below the curves the InterS-IntraQ protocol performs better than the variant, but there are many parameters to be considered: the resource size (in logarithmic scale), the ICP lookup time, the average network bandwidth among the edge servers, the probability of having a valid copy of the resource in a near and far edge server (in Figure 4.7, this probability is set to 0.1, that is a value much higher than our observed average values). From this figure we have that the variant of the InterS-IntraQ protocol prevails when the resource is very large (above 500 KB), almost independently of the other parameters. On the other hand, good connections among the edge servers, long ICP lookup times, low probabilities (below 0.1) of finding the same resource in near and far servers tend to increase the region where InterS-IntraQ is preferable to its variant.

Under the observation that from our data and real traces the resource

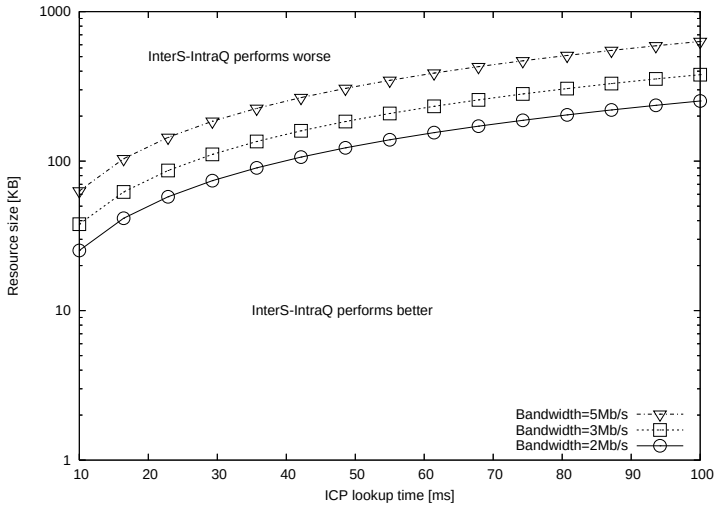

Figure 4.7: Comparison between the InterS-IntraQ protocol and its variant

size of the majority of the requests is of a few kilobytes and the hypothesis that the bandwidth among the edge servers is not so narrow even if they are "far", nowadays it seems more convenient to limit the lookup time through the Cache Digest scheme, as done by the InterS-IntraQ protocol. If in the future the average resource size increases more than the network bandwidth, it will be more convenient to adopt the variant of the InterS-IntraQ protocol.

4.3 Prototype implementation

The two-tier cooperation architecture is implemented by modifying Squid 2.4 [Squ], a well known and widely used proxy server. The main modifications to the Squid software are localized in The modules involved in cooperative resource discovery, and handling information on neighbor edge servers [LCC02].

In particular for the InterQ-IntraS and InterQ-IntraS-DM schemes we:

- modify the lookup algorithm in order to implement the two-tier lookup algorithm
- introduce a protocol (cache-to-master protocol – CMP) in order to let

edge servers communicate with their group master

- modify neighbor modules in order to allow the dynamic insertion of a new node in case of hit in a remote group

The implementation of the InterS-IntraQ scheme is more straightforward. The two step lookup algorithm is similar to the one of the InterQ-IntraS scheme and the cooperative lookup is carried out through a modification of the ICP protocol.

4.3.1 Implementation of the InterQ-IntraS and InterQ-IntraS-DM schemes

The Cache-to-Master Protocol (CMP) is used to manage the intercommunication between the two levels of the two-tier architecture. CMP is essentially a modified version of ICP, designed to be lighter and simpler than the original ICP protocol.

Both ICP and CMP have a *sender address* field. However, its use is quite different. In ICP this field is considered untrusted, and the information is usually taken from the lower level protocols. In our prototype architecture, a master edge server can report a hit referred to another cache. Hence, in both CMP and ICP responses the sender address contains the IP address of the cache holding the requested document.

The use of a master node to communicate with other groups is a two-way process: both queries and replies pass through the master. This is necessary to avoid the risk of *cache poisoning*: cache hit in unknown edge servers are trusted only if signaled by the group master, which is trusted.

The implementation of CMP requires a modification of the *neighbors* module. For security reasons, Squid does not fetch resources from unknown proxies, but this would not allow inter-cluster cooperation, because each proxy knows only the caches belonging to its cluster. With the modified module, when a CMP_HIT reply pointing to a previously unknown proxy is received, the module that manages this protocol calls a function in the

neighbors module that dynamically adds a new entry to the list of known proxies. In such a way, the Web objects are always retrieved from known peers even if they come from caches of other clusters.

4.3.2 Implementation of the InterS-IntraQ scheme

The key feature for the InterS-IntraQscheme is the use of the experimental `ICP_FLAG_POINTER` flag documented in [Wes01] (this feature was not previously implemented in Squid).

In the ICP queries exchanged in the lookup phase the `ICP_FLAG_POINTER` flag is enabled. This flag is used to express the capability of the edge server to process the additional opcode `ICP_OP_MISS_POINTER`. Response with this opcode report a miss in the contacted edge node but report a possible hit in other nodes. The list of nodes with potential hits are stored as the last fields of the ICP packet.

4.4 Workload models for performance evaluation

The difficulty of defining a "typical" workload model is a well known issue, because studies on real traces show great differences. For the experiments, we prefer to use two synthetic workload models generated by Web-Polygraph version 2.5 [RW00]. They intend to capture two "realistic" Internet scenarios, that are based on the model proposed for the second cache-off by IRCache [IRC95]. The basic workload is characterized by a high *recurrence* (that is, the probability that a resource is requested more than once), and small inter-arrival times for client requests. This latter characteristic contributes to degrade summary-based cooperation because of capacity and consistency misses, as observed in Section 3.3. We can anticipate that the chosen workload models tend to penalize the summary-based protocol with respect to query-based cooperation. Nevertheless, for a more than fair comparison

with the "pure" query-based protocols, in this study we prefer to report results of a worst-case scenario for first-tier lookup, that is always based on a summary-based protocol.

Every experiment is done twice and data measures are collected only in the second run, so to emulate a steady state scenario. It is worth to observe that an increment of the number of edge servers augments the global cache capacity, but also the size of the working set, because the number of clients is incremented proportionally. Preliminary tests were done to tune some parameters of Squid. For example, for summary-based cooperation we find more convenient to use a digest rebuild period of 60 seconds to reduce the consistency miss effects due to frequent cache object replacements (note that the Squid default value is 60 minutes).

Workload 1 represents a set of heterogeneous users with different interests. This characteristic, together with the document popularity model, leads to a high spatial locality. Client requests also show some temporal locality, so that only 30% of the workload is active at a given time. Requests are referred to a mix of content types consisting of images (65%), HTML documents (15%), binary data (0.5%), others (19.5%). The workload model defines a set of *hot* resources (1% of the working set) that receive about 10% of the requests. The heterogeneity of the user interests is represented by the fact that only 50% of the requests is taken from a "public" set of pages common to all clients, while the remaining 50% is taken from a "private" set, different for each client. Each client is configured to visit more than once only 80% of the URLs and the object cacheability is 80%. HTML resources typically contain embedded objects.

Workload 2 is a modified version of Workload 1, where the client population is more homogeneous. Here, clients share the same interests, so spatial locality is reduced, while the overall access locality is augmented. Moreover, the popularity model is chosen so to increase the size of the hot fraction of the workload. This workload takes less advantage from local cache hit rates and puts more pressure on cooperation. The modified main parameters of

Workload 1 are indicated below. The hot set of the workload is larger (15% of the access are referred to a hot set corresponding to 5% of the URL-space). This creates a popularity distribution with a heavier tail. The working set size keeps growing through the whole experiment (there is no temporal locality). The public fraction of requests is 100%, which means that every document in the workload is public, so that each resource can be requested by any client.

4.5 Performance evaluation of hybrid cooperation schemes

In this section we provide a performance analysis of the proposed two-tier architecture. All prototypes are extensively tested to verify their scalability and to compare the stability of their performance with that of the "pure" query-based and summary-based protocols. The first set of experiments, carried out with edge servers placed on the same network segment, focuses on cache hit rates, cooperation overheads, and sensitivity to other parameters, such as cache capacity and workload models. The InterQ-IntraS-DM scheme is not tested in this context because measuring response times and congestion (the main issues addressed by this cooperation scheme) would be meaningless in a LAN. With a second set of experiments we measure the response times of client requests in a geographic scenario. This is to guarantee that the stability of hybrid schemes on cooperation overhead and cache hit rates are not achieved at the expenses of user response time. These experiments also include the InterQ-IntraS-DM cooperation protocol.

4.5.1 Scalability of the cooperation protocols

In Section 3.3 we observed that "pure" cooperation protocols do not scale well over a limited number of cooperating edge servers: traffic overheads and poor cache hit rates limit query-based and summary-based protocols, respec-

tively. In this section, we aim to verify whether hybrid cooperation protocols based on a two-tier topology may provide better scalability and more stable performance than "pure" protocols do. Moreover, we are interested in finding which is the best hybrid scheme among the novel protocols proposed in this chapter. To this purpose, we compare cache hit rates and cooperation overheads for an architecture with an increasing number of edge servers that are subject to Workload 1 and Workload 2. As a comparison testbed, we also include results for an architecture with the same number of nodes that do not cooperate for document discovery (namely, No Cooperation).

Table 4.1 and Table 4.2 summarize the results regarding Workload 2 and 1, respectively. In each table, the first column reports the number of cooperating nodes, the following three columns report the cache hit rates (local, first-tier and global), while the last four columns report the traffic overhead due to cooperation, that is indicated as additional bytes exchanged for each client request.

Actually, the most interesting results to evaluate the cooperation schemes are obtained for Workload 2. Hence, we derive the main conclusions from these experiments and use the results obtained with Workload 1 to confirm these results or show some light differences. We already observed that increasing the number of edge servers leads to an increment of the working set size and a consequent reduction of the percentage of documents that can be cached in each node. For example, for Workload 2 these percentages decrease from 7.5% to 2% of the working set when the cooperating nodes pass from 8 to 30. But the most interesting aspect of the Workload 2 characteristics is that a larger number of nodes leads to a sensible reduction of the local cache hit rates. From Table 4.1 we have that the local cache hit rates go from about 40% to about 15%. Some differences dependent on the cooperation scheme were expected, because an edge server belonging to a cooperative system receives requests from its clients and other edge servers. This alters the access patterns and in practice reduces the document locality. Indeed, the best local hit rates are obtained by the No-Cooperation scheme that pre-

Table 4.1: Cache hit rates and overheads for Workload 2

Number of Nodes	Local HR	1st-tier HR	Global HR	Intra-Group Overhead per request [bytes/req.]	Inter-Group Overhead per request [bytes/req.]	Total Overhead per request [bytes/req.]	Relative Overhead per node [bytes/req.]
InterS-IntraQ							
8	38.38%	45.69%	60.16%	284.25	7.32	291.58	36.44
15	25.89%	37.87%	55.66%	427.38	18.10	445.48	29.70
30	15.13%	30.42%	51.04%	543.05	49.10	642.14	21.40
InterQ-IntraS							
8	41.00%	52.50%	60.13%	62.05	41.67	104.72	13.09
15	23.63%	36.16%	60.96%	83.40	118.87	202.36	13.49
30	13.49%	26.71%	60.26%	116.61	249.58	366.19	12.21
Cache Digest							
8	38.00%		53.05%			5.70	0.71
15	26.92%	n/a	44.29%	n/a	n/a	6.32	0.42
30	16.28%		33.77%			6.84	0.23
ICP							
8	40.54%		66.49%			780.84	97.60
15	27.07%	n/a	66.15%	n/a	n/a	1882.64	125.51
30	16.19%		65.20%			4500.42	150.14
No Cooperation							
8	49.64%		49.64%				
15	27.84%	n/a	27.84%	n/a	n/a	n/a	n/a
30	16.35%		16.35%				

serves locality at most. The same behavior is shown in Table 4.2 referring to Workload 1, although in this case the local hit rates remain higher than those observed for Workload 2.

From Table 4.1 we can observe that the No-Cooperation scheme is not able to face the reduction of the local cache hit rates, and also Cache Digest shows many limits especially for higher numbers of cooperating edge servers. On the other hand, ICP and the other hybrid cooperation schemes compensate the effects of the local hit rate reduction with a stable and always over 50% global hit rate. InterQ-IntraS is better than InterS-IntraQ, but plain ICP is even better. The motivation for these results is related to the choice of the workload models that tend to penalize summary-based cooperation. Indeed, an increment of the working set size rises the frequency of object replacement, thus reducing the accuracy of the exchanged cache digests (there is an increment of capacity and consistency misses). Cache Digests is particularly sensitive to this and its hit rate (quite low even with only 8 cooperating nodes) is further reduced when the working set size augments. ICP is not affected by the frequency of cache replacements and this explains its good performance. However, it is remarkable how well the innovative hybrid schemes are able to address the issues related to cache replacement even if the peculiarities of the workload model affect their summary cooperation component occurring on the first-tier lookup. In particular, InterQ-IntraS seems the most stable protocol, because it overcomes the drawbacks due to the low first-tier lookup rates by means of great performance on the second-tier lookup. It is remarkable the case with 30 nodes, where its cache hit rates pass from 13% to 26% to 60%, that is the same value observed for lower numbers of cooperating edge servers. On the other hand, the InterS-IntraQ scheme finds a minor benefit from the query-based cooperation on the second-tier. The consequence is a reduction of the global cache hit rate of about 15%, that however remains much better than that of Cache Digests experiencing a decrease of about 38%.

The results reported in Table 4.2 related to Workload 1 confirm the previ-

ous conclusions: InterQ-IntraS and ICP show the most stable cache hit rates, followed by InterS-IntraQ and Cache Digest. Now, because of the higher spatial locality in request patterns, the hit rates are generally higher than those shown in Table 4.1. The main contributions to these results are due to the local hit rate (see No-Cooperation performance), which is increased by the greater locality caused by the higher percentage of "private" requests, as explained in the description of the workload models. Cooperation tends to reduce locality, and this motivates the results of Cache Digest that are even poorer than those of No-Cooperation.

Table 4.2: Cache hit rates and overheads for Workload 1

Number of Nodes	Local HR	1st-tier HR	Global HR	Intra-Group Overhead per request [$\frac{\text{bytes}}{\text{req.}}$]	Inter-Group Overhead per request [$\frac{\text{bytes}}{\text{req.}}$]	Total Overhead per request [$\frac{\text{bytes}}{\text{req.}}$]	Relative Overhead per node [$\frac{\text{bytes}}{\text{req.}}$]
InterS-IntraQ							
8	55.01%	57.60%	63.84%	332.15	7.14	339.29	42.41
15	39.81%	43.82%	52.55%	317.66	10.88	328.54	21.90
30	35.80%	44.11%	54.38%	427.40	24.46	451.86	15.06
InterQ-IntraS							
8	48.24%	55.40%	62.56%	70.44	82.84	153.28	19.16
15	40.77%	50.53%	63.62%	65.54	95.09	160.64	10.71
30	31.65%	55.12%	64.06%	88.37	192.83	281.20	9.37
Cache Digest							
8	31.47%		40.25%			3.67	0.46
15	30.52%	n/a	37.82%	n/a	n/a	5.11	0.34
30	29.14%		37.80%			5.48	0.18
ICP							
8	57.22%		69.46%			532.72	66.59
15	52.16%	n/a	68.24%	n/a	n/a	1238.13	82.54
30	45.12%		67.62%			2977.00	99.23
No Cooperation							
8	57.30%		57.30%				
15	52.28%	n/a	52.28%	n/a	n/a	n/a	n/a
30	45.45%		45.45%				

If we include the overheads due to cooperation in the analysis of scalability and stability, we can observe some interesting modifications in the ranking of the cooperation schemes. Once again, we use Workload 2 (Table 4.1) as

a main reference to explain our conclusions, and report data for Workload 1 (Table 4.2) as a comparison testbed. The last four columns of these tables show cooperation overheads that are measured as the ratio between the bytes exchanged for cooperation and the number of client connections (hits and misses) received by the edge servers, normalized by the number of cooperating edge servers in the last column.

As expected, augmenting the number of edge servers increases the cooperation overhead, but not every scheme is affected in the same way. In particular, ICP generates the highest cooperation traffic even with 8 nodes, and continues to increase with respect to the number of cooperating edge servers in a much faster way than any other schemes do. It is interesting to note that with 30 nodes the ICP cooperation overhead (traffic generated only to cooperative lookup purposes) reaches 4.5 KB per requests, with an average document size of about 10 KB. On the other hand, Cache Digests shows the lowest cooperation overhead.

Hybrid schemes occupy an intermediate position, with a cooperation overhead higher than that of Cache Digest, but significantly lower than that of ICP. InterQ-IntraS halves the traffic of InterS-IntraQ, but the order of magnitude remains the same. It is more important to observe that the experimental results confirm the motivation of the InterS-IntraQ scheme. This approach is effective in reducing the usage of inter-group network resources, because it offers a summary-like traffic on the distant and more expensive links, while it moves more overhead on the intra-group links. The results observed for Workload 2 are substantially confirmed by the experiments based on Workload 1, as shown in the last four columns of Table 4.2. To appreciate the stability of the results of the cooperation schemes, we refer to the last columns of Tables 4.1 and 4.2, showing the overhead traffic per request relative to each node. It is important that for larger numbers of cooperating edge servers, InterQ-IntraS does not show practical modifications, the relative overheads of InterS-IntraQ and Cache Digests relative overheads tend to diminish, whereas continuous increments only occur for ICP.

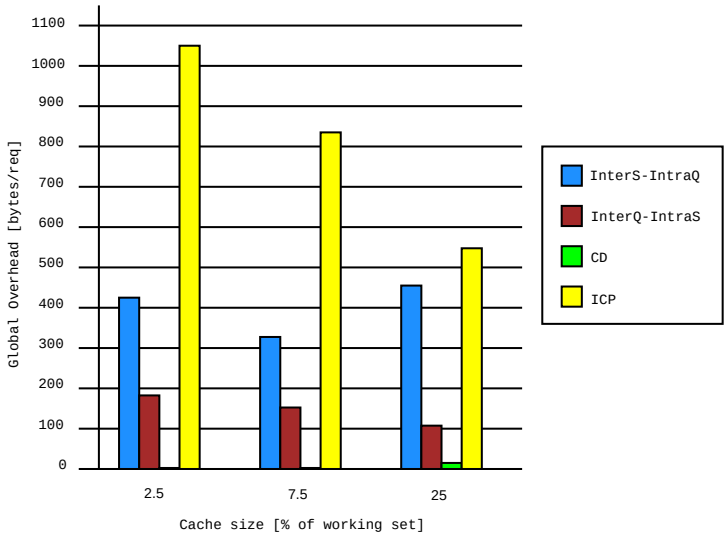

Figure 4.8: Sensitivity of traffic overhead for cooperation to cache capacity

4.5.2 Sensitivity analysis on cache capacity

Studying the sensitivity of the cooperation schemes with respect to the cache capacity evidences other interesting results, especially for the hybrid protocols. For these experiments we use the Workload 1 and only 8 nodes to reduce the advantages of the better scalability of two-tier cooperation protocols. The considered cache capacity per node is equal to 2.5%, 7.5% and 25% of the working set size. As expected, the cache hit rate of all schemes increases with the cache capacity. With the highest capacity, InterQ-IntraS, InterS-IntraQ and ICP show close performance beyond 70%. Even Cache Digest improves over 50%, but this scheme remains penalized by the frequent object replacements.

The results related to the cooperation overhead, reported in Figure 4.8, are interesting. Increasing the cache capacity brings two different effects: the local hit rate raises, but the digest size grows as well. The increment in local hit rates reduces the need of cooperation, hence the overhead related to query-based cooperation decreases, whereas the overhead related to summary-based cooperation tends to augment, because of the larger dimen-

sion of the exchanged digests. The overhead of Cache Digest remains two orders of magnitude below that of ICP. Hybrid schemes, combining both types of cooperation, have the most stable results with respect to cache capacity modifications.

4.5.3 Experiments on a geographic testbed

The last set of experiments is done in a geographic environment. The main goal is to verify whether the better scalability and stability of hybrid schemes (lower overheads than ICP with similar cache hit rates) have a negative influence on the user response times. We set up two groups of five nodes each, that are connected through some links and a geographic backbone. Ten network hops exist between the groups. We also install one Web server in each location. For this test, we report the results referring to Workload 1, because they are representative of Workload 2 as well. The experiments were performed with two different network conditions.

The first test is carried out on Sunday, with light network load, while the second test is done in conditions of heavy network load during a working day. The observed mean round-trip time is 50 ms and 300 ms, in the case of light and heavy network load, respectively.

We collect the data related to the client request service times from the Squid logs which can be used as an estimation for the user response time. We also monitor the network resource usage on the worst performing links. The worst, thus becoming the bottleneck, has a capacity of 2 Mbit/sec on one hop.

Figures 4.9 and 4.10 show the cumulative probability of the response times, for light and heavy network load, respectively. It is interesting to note the correlation between the cache hit rate and the user response time, that leads to the Bernoullian shape of the curves in Figures 4.9 and 4.10: requests are served very early or after a large amount of time.

Indeed, there is a big difference between client requests resulting in a hit (usually served in very short time) and requests occurring in a global

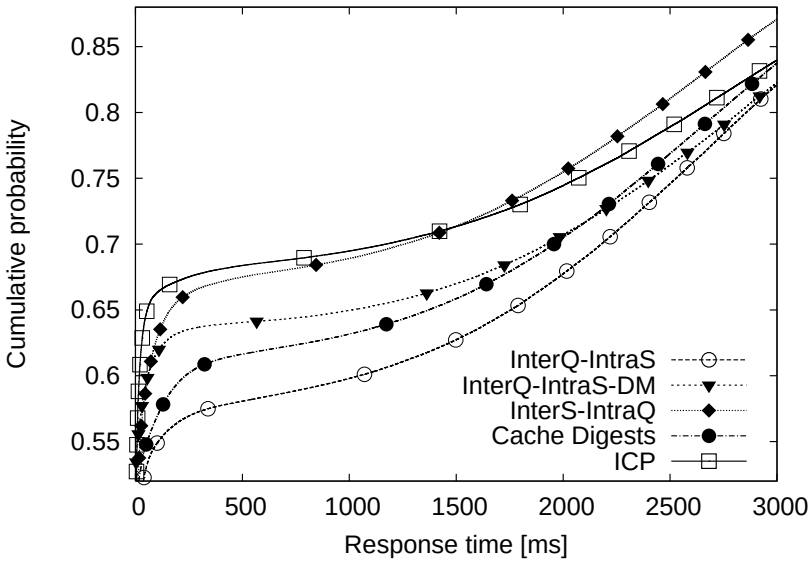

Figure 4.9: Response time (light network load)

miss (that usually experience a much higher latency). ICP shows the best response times for hit resources, but poorest results in the case of global misses because it has to wait for the slowest cooperating edge server. In Figure 4.9 the ICP curve is the highest at the beginning, but it is surpassed by that of InterS-IntraQ (around 1.5 seconds) and becomes the worst (after 4 seconds). These effects, even if already shown by our experiments, become much more evident in other not reported curves obtained for a workload model characterized by scarce locality and consequent lower hit rates. The conclusions on response times can be better appreciated by evaluating the 90-percentile of the request service time instead than looking at the curves that seem so close. As shown in Table 4.3, in the case of light network load, the 90-percentile for ICP is 3.7 seconds (the highest value), while for InterS-IntraQ it is 3.2 sec (the lowest value) and for InterQ-IntraS-DM it is 3.6 seconds. Cache Digests, because of its low hit rate tends to be slower in the first part of the curve, but the 90-percentile of its response time (3.5 seconds) is similar to that of the other cooperation protocols.

When the network is affected by heavy traffic (Figure 4.10), congestion

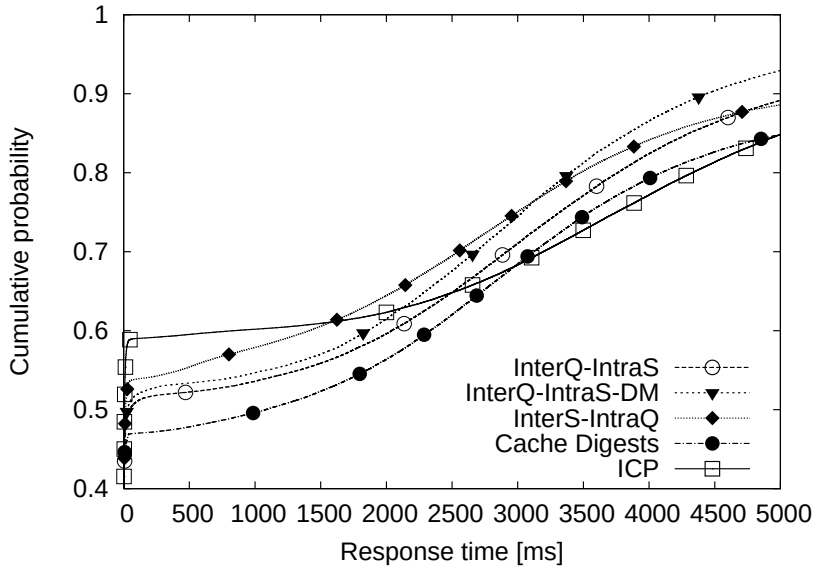

Figure 4.10: Response time (heavy network load)

Table 4.3: 90 percentile of response times in seconds

Protocol	Light network load [sec]	Heavy network load [sec]
InterQ-IntraS	3.60	5.19
InterQ-IntraS-DM	3.68	4.43
InterS-IntraQ	3.26	5.63
Cache Digests	3.51	7.30
ICP	3.70	6.23

can occur in many places of the architecture. The overall performance of the cooperative system is reduced: the percentage of client requests serviced within 100 ms is reduced from 66%-56% to 59%-47% depending on the cooperation protocol. Furthermore, the cache hit rate decreases (from 68%-47% to 61%-36%) in conditions of heavy network load. The differences among the various hybrid schemes are reduced because the congestion occurring in the query-based cooperation overpowers the congestion of InterQ-IntraS protocol. As shown in Table 4.3 also the 90-percentile of the user response time augments: ICP shows once again poor performance (6.2 seconds), far more than that of the best InterQ-IntraS-DM remaining below 4.5 seconds. Cache Digests is the worst cooperation protocol in terms of 90-percentile (7.3 sec) when the network is congested. The motivation is that its lower hit rate augments the use of congested links to contact the Web servers. The other hybrid schemes, InterQ-IntraS and InterS-IntraQ, have an intermediate result with 90-percentile equal to 5.2 and 5.6 seconds, respectively.

The InterQ-IntraS scheme, that showed the best results in the previous subsections referring to LAN-based experiments, seems the worst of the three hybrid schemes in a geographic scenario. Collecting user response time statistics in a LAN environment is senseless, while it is very useful in a WAN environment. That is why we notice the congestion occurring at the group masters only in this latter series of experiments. We can see from Figure 4.10 that, when the group masters also act as edge servers, about 15% of resources are served after a long time (up to 5 seconds). This problem is solved by the InterQ-IntraS-DM variant with dedicated group masters. Indeed, even with a smaller number of edge servers, InterQ-IntraS-DM improves the performance of the hybrid schemes: in the case of heavy network load, its 90-percentile is equal to 4.4 seconds, while the corresponding index is equal to 5.2 and 5.6 seconds for InterQ-IntraS and InterS-IntraQ, respectively, as shown in Table 4.3. Actually, this bad result for InterS-IntraQ is mainly due to its lower cache hit rate that, like for Cache Digests, is more affected by the characteristics of the workload considered in this experiment. InterQ-IntraS-DM is

the best cooperation protocol in terms of 90 percentile in the case of a heavy network load. However, it is worth to observe that especially when the cache hit rate has a lower impact on response time (as it is for the case of light network load), InterS-IntraQ is a valid alternative. Its curve is often over the others in Figure 4.9, and even better than InterQ-IntraS-DM in some circumstances (see the first part of Figure 4.10). It degrades when its low cache hit rate influences the percentage of objects that must be retrieved from the origin Web server.

4.5.4 Summary of the experimental results

The large set of experiments carried out in this and the previous chapters highlights some interesting results.

- Query-based protocols seem very effective in resource discovery because their mechanism is less subject to stale information. However, their high cache hit rates are affected by cooperation overheads that grow as the number of cooperating edge servers increases, and by network traffic especially over distant and congested links. The tests in a geographic environment evidence the poor stability of results based on query mechanisms: they have low response time in case of hits, but they are much slower than other protocols in handling misses.

- Summary-based protocols cause negligible overhead for cooperation, but their cache hit rates are very sensitive to many workload and architecture parameters. For example, when the frequency of client requests is high and/or there is a frequent object replacement, it is not convenient to use summary-based protocols for cooperative discovery. Even in a geographic environment, their performance is mainly affected by low hit rates.

- Hybrid protocols achieve intermediate results, with hit rates close to query-based protocols (especially for the InterQ-IntraS and InterQ-IntraS-DM schemes), and overheads growing much slower than those

of query-based protocols as the number of cooperating edge servers increases. This twofold effect allows us to conclude that they have the best potential scalability. Tests in the geographic scenario confirm this hypothesis: two-tier protocols show good ability in dealing with both hit and miss objects. Their stable performance in the highly variable Web scenario is the most important achievement of these hybrid schemes.

A final remark is in order about the general applicability of these results. The two main workload models considered in this study derive from a careful selection of many other results. They have been included here because they bring the most important conclusions and certainly do not favor hybrid schemes. On the other hand, they tend to favor query-based cooperation with respect to summary-based cooperation. Actually, we can say that different workload characteristics never contributed to improve ICP performance shown here.

Other workload models can only improve summary-based performance (e.g., less frequent object replacements in caches, higher inter-arrival times for client requests) or deteriorate query-based results especially for the user response time (e.g., lower recurrence of requests leading to reduced local hit rates). The real important point is that the results of the hybrid schemes have been demonstrated to be more independent of the workload models, because they can backup the defacement of a cooperation protocol in the first-tier lookup with the improvement of the other protocol in the second-tier lookup.

Part III

Efficient Web content adaptation

Chapter 5

Intermediary adaptation systems

5.1 Introduction

In the *Pervasive and Ubiquitous Computing* era the trend is to access Web content and multimedia applications by taking into account four types of important requirements: *anytime-anywhere* access to *any data* through *any device* and by using *any access network.* Nowadays, the existing and emerging wireless technologies of the 3^{rd} generation (e.g. UMTS on a geographical scale, WiFi on a local area) involve a growing proliferation of new rich, multimedia and interactive applications that may be accessed through the Web interface. On the other hand, these appealing services are requested from a growing variety of terminal devices, such as pagers, personal digital assistants (PDAs), hand-held computers, Web-phones, TV browsers, set top boxes, smart watches, car navigation systems. The capabilities of these devices widely range according to different hardware and software properties. In particular, for mobile devices relevant constraints concern storage, display capabilities (such as screen size and color depth), intermittent wireless network connections, processing power and power consumption. These constraints involve several challenges for the delivery and presentation of

complex personalized applications towards these devices, especially if there is the necessity of guaranteeing specified levels of service.

In this last part of the thesis, we focus on the adaptation level, (Figure 5.1), that adapts the contents coming from origin servers or through some caching infrastructure to match client device capabilities and/or users' preferences.

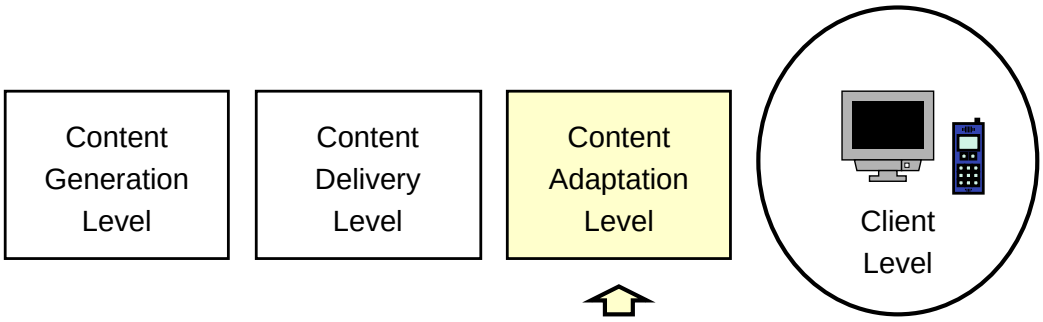

Figure 5.1: The adaptation level of the Web-based infrastructure

The most simple solution (still used by some Web portals) is to provide content that is specifically designed for desktop PC, without taking care of the troubles that could affect mobile users. Another approach is to replicate the content for two sets of devices. Both these solutions may work in a short term perspective. In fact, the effective presentation of Web content would require additional efforts, and new computation patterns must be provided to meet the requirements of the rapid and continuous evolving of pervasive and ubiquitous devices.

By allowing computation anytime, anywhere, and on any device, pervasive computing environments exhibit characteristics that are not shared by traditional computing environments, such as:

Heterogeneity of computing devices. Computing devices show a wide variability range from small mobile devices, such as smart phones and PDAs, to tablet PCs, laptops, and stationary devices such as PCs. Each of these devices has a different hardware configuration and capability, and possibly a different software platform for program execution.

Limited device capability. Many mobile devices are designed to be small for easy portability. They are small not only in terms of form factor, but also computing power, memory, display screen size and battery power. All these factors limit the type of computing that can be done on these devices.

Limited network connectivity. The wireless connection used in the mobile devices is usually bandwidth-limited and unstable when compared with the wired connection. Although new networking standards (such as GPRS and Bluetooth) exist, the comparatively slow speed is always a barrier to run large applications using mobile devices.

High mobility. Users in pervasive computing environments are highly mobile. Without the need to have a fixed association with the computing devices, users can freely move from place to place with or without any mobile device. This high mobility nature also results in a continuous change of the computing environment, or execution context, that can affect the execution.

These characteristics are essential to help identifying the requirements of pervasive computing, that we can summarize as following.

Client-device adaptability. Client devices of different capabilities have different requirements to the information or services being retrieved. Failure in satisfying these requirements results in the inability of the devices to process the information or services in a proper way. For example, Web pages normally designed for PCs cannot be displayed properly on a PDA without adaptation. Also, a device with limited amount of memory cannot be used to run large monolithic applications (e.g. a full-featured image editor) that consume a large amount of memory resources thus exceeding the client device capability. In order for the client to process them properly, information or services have to be adapted to the client devices, according to their capabilities.

Device-user adaptability. The free associativity between devices and users allows devices of the same capability (e.g. the same model) to be used by different users. The requirements posed by these clients in accessing the information or services might be different. This is due to the different requirements of the users, which are referred to as the user preferences. Adapting to the device-user according to their preferences is, therefore, needed.

User mobility support. User mobility is the ability of a user to move and compute, while being treated as the same user independently of the physical location. In pervasive computing, users are moving all the time, requesting information or services from different locations. Due to this highly mobile nature of the users, there is a need to minimize the disturbances made to the computing experiences of the users, such as maintaining the same environment (e.g. visited Web pages, prepared documents), and providing service continuation for on-going services. All these support for a user at different locations are considered as user mobility support.

Context-awareness. With the ability to compute any place, any time, the execution context of a client changes quite often in pervasive computing environments. The required information, the type and quality of service, etc. are affected by a clients execution context. For example, a detailed road-map when driving in the outback VS a simplified one in the city. Better services can only be provided if information about the clients execution contexts is known.

Among the technical measures to be taken to tackle the problem of integrating services and terminals on the fixed network and the mobile infrastructures, there is a growing demand for network infrastructures that are able to efficiently provide *intelligent* services, such as personalization, localization, adaptation services, at the *edge* of the network, as close as possible to end users, that are, indeed, the ultimate judges of the quality of the services.

5.2 Related work

One of the current research trend in distributed systems is how to extend the traditional client/server computational paradigm to provide *intelligent* and *advanced* services. This computational paradigm introduces new actors within the WWW scene, the intermediaries [BM99b, LA94] that is, software entities that act on the HTTP data flow exchanged between client and server, by allowing content adaptation and other complex functionalities, such as geographical localization, group navigation and awareness for social navigation [BM98a, CMS03], translation services [IBM], adaptive compression and format transcoding [AGL+03, HKO+00, Web06b].

The idea of using an intermediary server to deploy adapted contents is not novel: several popular adaptation systems exist, which include RabbIT [Rab], Muffin [Boy], WebCleaner [Web], WBI [IBM] and Privoxy [Pri]. These systems, whose architecture, model and functionalities are described in detail in Section 5.3, provide functionalities such as compressing text, removing and/or reducing images, removing cookies, killing GIF animations, removing advertisement, Java applets and Javascript code, banners, pop-ups, and finally, protecting privacy and controlling access.

5.3 Programmable intermediary systems on the WWW

In this section we present the main features of existing intermediary frameworks that offer content adaptation capabilities. Furthermore we present SISI, a novel flexible intermediary infrastructure that enables universal access to the Web content. First we give a brief description of each of them, then we compare them all together.

RabbIT. RabbIT [Rab] is a Java-based intermediary adaptation server whose main goal is to speed up users' Web navigation by compressing both

text pages and images, by removing unnecessary parts of HTML pages such as HTML tags (i.e. background images, etc.) and annoying advertisements, banners, pop-ups, etc. Moreover, it caches filtered documents before forwarding them to the clients. RabbIT supports a system-wide profile that is, all users that connect to the Internet through this intermediary will have the same filters applied to all their requests.

RabbIT has a modular architecture that allows an easy definition and implementation of new functionalities. Its programmability allows the implementation of new modules, called *filters*, by using a set of provided APIs. These filters can also be configured at run-time by simply accessing a specific URL. It provides some example of configuration files that can be manipulated to provide new configurations.

Several filters are provided with the distribution and, in particular, an authentication service for access control, a service that allows RabbIT to work as reverse proxy, demo services to filter Web pages (removing background images, ad banners, and blink tags.)

Muffin. Muffin [Boy] is Java-based intermediary adaptation server that provides functionalities to remove cookies, advertisements, to kill GIF animations, to add/remove/modify any HTML tag.

Muffin is easily programmable and configurable: new services, called *filters*, can be implemented through a set of provided APIs and can be added at run-time, by using the provided graphical interface, without restarting the intermediary adaptation server.

Each service is implemented as a Java file. A set of provided APIs and filter interfaces simplify the implementation of new services. Through the GUI, users can also specify in which order services should be invoked.

The distribution includes various filters to de-animate GIF images, remove Javascript and Java for HTML documents, remove images that match specific size conditions, remove/modify colors and background images.

Muffin supports a system-wide profile.

WebCleaner. WebCleaner [Web] is an intermediary adaptation server that provides functionalities to remove advertisements, banners, Flash and Javascript code. It also provides text and image transcoding.

WebCleaner, implemented in C and Phyton, is programmable and highly configurable: new services can be added to the system through a graphical user interface by simply specifying appropriate configuration parameters.

Several filters have been provided with the distribution, and in particular, services for HTML filtering, image size reducing, Web pages blocking, users' authorization, security and virus detection. WebCleaner supports a system-wide profile.

Privoxy Web Proxy. Privoxy [Pri] is an intermediary adaptation server with advanced filtering capabilities to protect privacy, modify Web page content, access control, and to remove advertisements, banners, pop-ups. Privoxy, implemented in C programming language, is programmable and highly configurable. Its services can be customized according to individual user needs. In particular, Privoxy can be configured with various configuration files, accessed through specific URLs, intercepted by the intermediary adaptation server. A set of APIs has been provided to simplify the process of implementing new services.

Different filters are provided with the distribution, such as services for ad-filtering by link and by size, ad-blocking by URL, GIF de-animation.

Privoxy supports a system-wide profile.

Web Based Intermediaries (WBI) [BM98b, BM99b, BM99a] is a Java-based dynamic and programmable intermediary adaptation server, whose main goal is to personalize the Web through an architecture that simplifies the implementation of intermediary applications.

The WBI framework defines a programming model that can be used to implement all form of intermediaries, from simple server functions to complex distributed applications.

A WBI transaction is defined as a complete HTTP request-response and flows through a combination of four types of basic steps, called Megs: the *RequestEditor* and *Editor* Megs modify HTTP requests and responses, respectively; the *Generator* Meg produces the response to an HTTP request; the *Monitor* Meg monitors HTTP requests and response without any modification. A group of Megs is called WBI plugin.

WBI provides a rule-based system to dynamically determine the data path through the various Megs allowing complex boolean expressions. Finally, the WBI GUI provides a convenient way to manage and administer WBI, and to help users to debug the plugins loaded into the intermediary adaptation server.

WBI supports a system-wide profile.

Scalable Intermediary Software Infrastructure (SISI)) This framework has been designed aiming at guaranteeing an efficient and scalable delivery of personalized services. In particular, SISI is able to offer the following functionalities [GMM+05, CGM+05], which we describe in detail in Section 5.4:

- Life-cycle support. The intermediary adaptation server fully supports the deployment and un-deployment of services, by making these tasks automatic and accessible by remote locations. Moreover, support for pausing and restarting services in a simple way from a remote host is also granted.

- Programmability. Programming under existing intermediaries cannot involve the same power, efficiency, expressiveness and generality like developing an open system from scratch. This means that we need a compositional framework and a programming model that allow us to realize a general-purpose programmable environment, whose functionalities are implemented by APIs provided by the intermediary adaptation server.

- Security. SISI is able to allow access to resources in a *trusted* way allowing both authorization and authentication for each user's session. Moreover, services management mechanisms control that users are allowed to perform only the operations for which they have the rights.

- Logging and auditing. SISI provides mechanisms to record security-related events (logging) by producing an audit trail that makes possible the reconstruction and examination (auditing) of a sequence of events. The process of capturing user activity and other events on the system, storing this information and producing system reports is important to understand and recover from security attacks. Logging is also important to provide billing support, since services can be offered with different price models (flat-rate, per-request, per-byte billing options). SISI offers the possibility to manage accounting to users for each service, as specified by the system manager.

- *Per-user* profiling. Many existing intermediary adaptation servers only allow for a system profile, which is applied to all requests, coming from any user. That is, all the requests involve the same adaptation services. SISI, instead, allows each user to define one (or more) personal profiles, in such a way that the requests of each user may involve the application of a specific set of services chosen by the user.

- High scalability. Many advanced adaptation services are computationally onerous and the large majority of existing frameworks does not support more than few units of concurrent requests per second. On the other hand we demonstrate that SISI, with the same hardware facilities than another intermediary adaptation server, nearly triples its scalability.

Table 5.1 sums up the main features of the aforementioned intermediary adaptation servers. As you can see, they all are programmable, even though implemented using completely different programming languages, ranging from

Table 5.1: Main features of some intermediary adaptation servers

Feature	RabbIT	Muffin	Privoxy	WebCleaner	WBI	SISI
Programming language	Java	Java	C	C, Phyton	Java	Perl
Configuration	Web	Web, GUI	Web	Web	GUI	Web, GUI
Per-user profile						X
Host access restriction	X	X	X	X		X
User authentication	X					X
HTTP/1.1 compliance	X	partial	partial	X		X
Gif Deanimation		X	X	X		X
Cookies Removal		X	X		X	X
Java/Javascript removal		X	X	X		X
Header modification/removal		partial	X	X	X	X
Background images removal	X	X			X	X
Images removal		X		X		X
URL blocking	X	X			X	X
Image transcoding	X		X	X	X	X

Java to C or Phyton. This brings the need to master the right programming language in case of creating new services.

From the configuration point of view, they all are easily configurable through a Web interface or with a GUI, without the need to explicitly modify the configuration files. They all support a system wide profile. There is a big difference between SISI an the other intermediary adaptation servers, since they all only allow for a system wide configuration, that is users cannot have their own configuration. One of SISI strengths is a per-user configuration.

Most of them comply in part or fully to the HTTP/1.1 protocol, that is for instance Privoxy does not support requests pipelining. WBI instead does not comply to the HTTP/1.1 specification.

All of them are provided with some already implemented filters, that can also be used as examples when programming new ones. These filters include for instance facilities to remove cookies, de-animate GIF images, add, remove or modify some request/response headers, remove background images and so on. Some SISI filters are described in more detail in Section 5.7.

Other examples of intermediary-based infrastructures include AT&T Mobile Network iMobile [RCCC01] that is, a platform designed to provide per-

sonalized mobile services. It provides a modular architecture that allows an easy definition of new functionalities implemented as building blocks in Java, on top of a programmable framework.

The *extensible Retrieval Annotation and Caching Engine* (eRACE) [DZY01] is a modular, programmable and distributed infrastructure whose main goal is to provide personalized services for a wide range of client devices such as personalization, customization, filtering, aggregation and transformation. Authentication and profiling are managed by a Service Manager integrated into the system. Profiles stored by eRACE (in XML format) include both user preferences and service characteristics.

The MoCA's ProxyFramework [SER+04] allows the deployment of adaptive proxies for context-aware mobile applications. In particular the main components of the MoCA framework include a Monitor (to collect information about device's execution state), a Context Information Service (to receive and process state information), a Discovery Service (to store information about any provided service), a Configuration service (to store and manage configuration information for all mobile devices) and the Location Inference Service (to approximate geographical locations). A set of XML configuration files is provided by the system administrator to manage all environments conditions that the system could support. Through a rule-based approach, the system is able to determine which actions must be taken in order to provide the best service according to the context.

5.4 Scalable Intermediary Software Infrastructure (SISI)

In this section we describe the main characteristics of the SISI architecture. This novel framework is based on top of existing open-source applications, such as Apache Web server [Apa06] and mod_perl [mod].

Apache is recognized as the world's most popular Web server (HTTP server) [sur04]. It provides a full range of Web server features, including

CGI, SSL, and virtual domains. Apache also supports plug-in modules for extensibility and is reliable and relatively easy to configure. Finally, it is a free software distributed by the Apache Software Foundation, that promotes various open source advanced Web technologies.

mod_perl [mod] represents the perfect marriage of the Perl programming language [per] and the Apache Web server. It brings together the power of these two powerful technologies by providing a programmable framework to build and accelerate dynamic content, providing mechanisms for database integration, allowing a simple customization of new modules that can be directly and easily integrated into Apache, managing the Apache configuration file.

The most important characteristic of the Apache Web server is that it can be extended with new modules, commonly written in C programming language. An apparently easier alternative is to write novel modules in Perl programming language and to integrate them into Apache through mod_perl. This approach extends the behavior of the Apache Web server by taking advantage of the power and flexibility of the Perl language.

Having a persistent Perl interpreter embedded in the Apache Web server allows us to avoid the overhead of starting an external interpreter for any HTTP request which needs to run Perl code. An important feature is the code caching: the modules and scripts are loaded and compiled only once, when the server is first started. Then for the rest of the server's life-cycle the scripts are served from the cache, so the server only has to run the precompiled code.

Each HTTP request is processed in sequential phases, and at each phase different decisions can be taken about the request (processed, rejected, or simply forwarded to the succeeding phase). HTTP requests can be managed by the standard Apache core, or by new and external modules. With mod_perl all phases of the HTTP request cycle can be accessed and controlled, allowing to enhance and personalize the behavior of the Web server. Each HTTP request goes through many phases, and on each phase it is pos-

sible to provide a specialized handler, such as a PerlTransHandler to manipulate the requested URI, the PerlAccessHandler to provide restrictions, the PerlAuthenHandler and PerlAuthzHandler to provide authentication and authorization mechanisms, the PerlHeaderParserHandler to inspect the HTTP requests and perform tasks according to conditions, the PerlResponseHandler to produce HTTP responses.

The main goal of SISI is to realize an innovative framework that aims to facilitate the deployment of adaptation services running on intermediary servers on the Web.

The SISI framework has a modular architecture that allows the implementation of new services from simple buildings blocks, their composition to provide complex functionalities and a simple approach to configure services according to users' needs.

An important feature of the SISI framework concerns the separation between the system core functionalities and new services. The idea is to provide system programmers with a tool to implement services without the need of taking care of the details of the infrastructure that will host such services.

SISI is entirely written in Perl and utilizes mod_perl within Apache to speed up performance. Our framework consists of several modules: each of them acts in a specific phase of the Apache HTTP Request Life-cycle.

To understand how this works, we now briefly describe how request processing works within Apache.

When Apache receives a request, it processes it in a series of phases. For every phase, a standard default handler is supplied by Apache. You can also write your own Perl handlers for each phase; they will override or extend the default behavior. The phases are illustrated in Figure 5.2.

Post-read-request. This phase occurs when the server has read all the incoming request's data and parsed the HTTP header. Usually, this stage is used to perform something that should be done once per request, as early as possible. Modules' authors usually use this phase to initialize per-request data to be used in subsequent phases.

URI translation. In this phase, the requested URI is translated to the name of a physical file or the name of a virtual document that will be created on the fly. Apache performs the translation based on configuration directives such as ScriptAlias. This translation can be completely modified by modules such as mod_rewrite, which register themselves with Apache to be invoked in this phase of the request processing.

Header parsing. During this phase, you can examine and modify the request headers and take a special action if needed, e.g., blocking unwanted agents as early as possible.

Access control. This phase allows the server owner to restrict access to specific resources based on various rules, such as the client's IP address or the day of week.

Authentication. Sometimes there is the need to be sure that a user really is who he claims to be. To verify his identity, the system asks him a question that only he can answer, generally a login name and password.

Authorization. The service might have various restricted areas, and you might want to allow the user to access some of these areas. Once a user has passed the authentication process, it is easy to check whether a specific location can be accessed by that user.

MIME type checking. Apache handles requests for different types of files in different ways. For static HTML files, the content is simply sent directly to the client from the file system. For CGI scripts, the processing is done by mod_cgi, while for mod_perl programs, the processing is done by mod_perl and the appropriate Perl handler. During this phase, Apache actually decides on which method to use, basing its choice on various things such as configuration directives, the file name's extension, or an analysis of its content. When the choice has been made, Apache selects the appropriate content handler, which will be used in the next phase.

Fixup. This phase is provided to allow last-minute adjustments to the environment and the request record before the actual work in the content handler starts.

Response. This is the phase where most of the work happens. First, the handler that generates the response (a content handler) sends a set of HTTP headers to the client. These headers include the Content-type header, which is either picked by the MIME-type-checking phase or provided dynamically by a program. Then the actual content is generated and sent to the client. The content generation might entail reading a simple file (in the case of static files) or performing a complex database query and HTML-ifying the results (in the case of the dynamic content that mod_perl handlers provide). This is where mod_cgi, Apache::Registry, and other content handlers run.

Logging. By default, a single line describing every request is logged into a flat file. Using the configuration directives, you can specify which bits of information should be logged and where. This phase lets you hook custom logging handlers for example, logging into a relational database or sending log information to a dedicated master machine that collects the logs from many different hosts.

Cleanup. At the end of each request, the modules that participated in one or more previous phases are allowed to perform various cleanups, such as ensuring that the resources that were locked but not freed are released (e.g., a process aborted by a user who pressed the Stop button), deleting temporary files, and so on. Each module registers its cleanup code, either in its source code or as a separate configuration entry.

At almost every phase, if there is an error and the request is aborted, Apache returns an error code to the client using the default error handler (or a custom one, if provided).

Figure 5.2 also shows the main components of the SISI framework and when they play their role in the request life-cycle:

- The *ProxyPerl* module, that performs the intermediary's role by intercepting the HTTP requests/responses
- The *Authorization* module, whose main goal is to restrict access to the system and provide a mechanism to distinguish between users
- The *FilterPlugin* module that acts as dispatcher to invoke the services selected by users during a configuration phase
- The *Deploy* module, that allows programmers to easily add new services to the framework without having to know detailed information about the infrastructure that will provide them for users' requests.

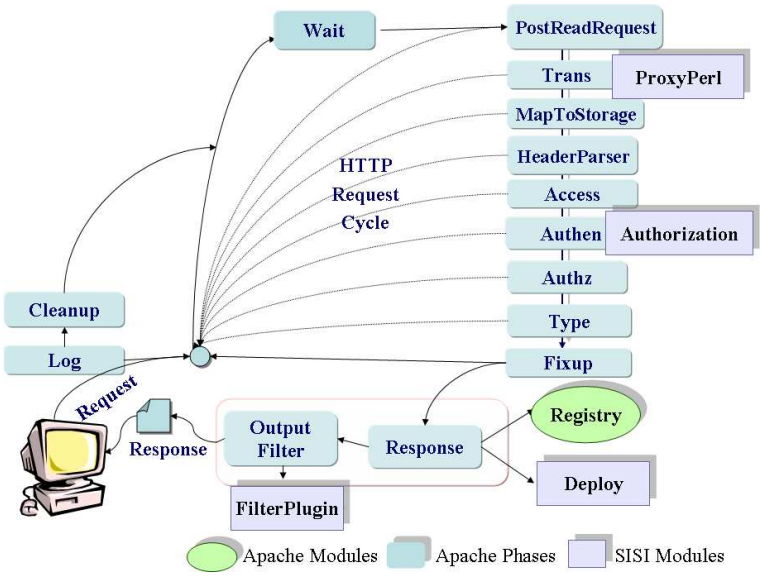

Figure 5.2: Placement of the SISI modules into the Apache request Life-cycle.

ProxyPerl module

The ProxyPerl module acts as the proxy component of the SISI framework. It intercepts HTTP requests and responses, and, if transformations are required, it loads users' configurations and performs the adaptation process.

This module is placed in the Apache *Trans* phase, as shown in Figure 5.2 and is mainly used to manipulate requested URIs. This is also a good place to register new handlers because this module is early involved in the traditional Apache request life-cycle and, then, this is a suitable place to develop handlers that manipulate URIs or HTTP headers that have high priorities. Since the request has not been associated with a particular file name or directory yet, the modules implemented at this phase will be triggered by any HTTP client request.

When the client issues an HTTP request, the ProxyPerl module identifies the user (or, else, provides to initiate a challenge-response authentication), and, then, loads the specific features of the services that the user selected during the configuration. Once the client is identified, the *personalized* navigation begins: the services selected by the user are applied to the HTTP flow of requests/responses.

In this context an important characteristics is the *hot-swap* of services composition i.e. the capability to load and execute different services according to different profiles at run-time with no need to recompile or even simply restart the software infrastructure.

Authorization module

Each service must be able to rely on a module to easily and safely authenticate the user (“*Are you who you say you are?*”) as well as to check the operations that the user is asking to perform (“*Are you allowed to do what you are asking for?*”).

This means that the intermediary adaptation server must verify that the user issuing a request is bearing the necessary authorizations in order, for example, to charge the user for the requested operation. Then, the Authorization service is useful both to restrict access to certain resources or services as well as to distinguish between users. In fact, services that treat users differently, e.g., based on per-user settings, need a mechanism that is able to distinguish among users.

The Apache Authentication phase is called whenever the requested file or directory is password protected. Our *Authentication module*, that acts in this phase, is used to verify user's identification credentials. If the user is authenticated, the handler loads the user's profile and starts the navigation, otherwise it provides a challenge-response authentication mechanism asking for the credentials by using the standard Proxy Authorization HTTP header `Proxy-Authorization` as dictated in [FHBH+99].

Registry module

The Apache Registry handler is the standard Apache module that allows for an efficient running of CGI scripts under mod_perl, by compiling all scripts once at the server start-up and then caching them in main memory.

Our architecture uses the Apache Registry handler to handle user's and services data. In particular, the scripts are part of three different categories: the first one gives information about the available services; in the second one, the administrator can manage user's profiles and, finally, the last category of scripts allows users to modify their own data and profiles.

In order to obtain the services' list, the user must enter his/her credentials (login and password), and access the *Change Profile Section*, after having requested the home Web page of the intermediary adaptation server. At this point, the user can choose which services to activate among all the available services. The page representing the list of services consists of an HTML form, divided in many parts, to set service parameters. In addition, the administrator, after the authentication phase, can perform some tasks such as adding or removing users from the system. After the initialization phase, any user can access the system and, then, perform some operations such as changing the password, creating, modifying or changing his/her profiles.

FilterPlugin module

This module acts as a dispatcher within the SISI architecture. A list of available services is deployed into the Apache Web server and it can be accessed

by the dispatcher in order to activate the services according to the users' preferences (users' profiles). To this end, the *FilterPlugin* module uses an internal parser to dynamically invoke the requested services. From the pool of available services, each service is applied on the HTTP requests/responses only if it was selected from the user issuing the request.

Data flow through the various services and only the activated services take them as input, manipulate them and produce the output data flow.

Modules that implement the services are preloaded in main memory, but only the modules that correspond to requested services will be allocated. Each service is identified by a task-based service name, with a set of parameters that each user can modify according to his/her preferences.

Deploy module

This is an important system functionality that allows programmers to automatically add new services to the framework without necessarily having specific and detailed knowledge about Apache and mod_perl. It consists of an automatic module generation process that implements adaptation services starting from simple Perl files.

If the programmer follows some easy and well defined rules (templates), his/her Perl programs can be used to build the *core* of the service modules to be loaded into Apache. In detail, the *Deploy module* is an Apache module activated via URI. It uses an XML file (.reg) that defines the parameters needed in the dynamic creation and installation of the service.

The deployment is obtained by filling out forms in a HTML page. Information about the service is stored (by a CGI script called Deploy.pl) in a XML file (service.reg) and used by the *Deploy module* (Deploy.pm) to build the service itself. The information sent to the *Deploy module* includes the service name, the module type, service parameters, and so on. The *Deploy module* restarts the Apache Web server and the new service is added to the pool of available handlers, thus the new service can now be invoked by end users.

5.5 Integrating SISI with new functionalities

SISI programmability is a crucial characteristics since it allows an easy implementation and assembling of adaptation services that enhance the quality and the perception of user's navigation. In particular, easily programmable intermediary adaptation servers can be a consistent aid in coping with the rich and dynamic nature of the Web [MS06], and tackling the challenges that come from the ever growing need for advanced services from a mature audience of users as well as the huge corpus of information that is daily accessed by anyone and anywhere in the world.

Often the introduction of new services into existing software infrastructure is an expensive and time-consuming process. To simplify this task, SISI exhibits a flexible and extensible environment in which a compositional framework is used to provide the basic components to develop new services, and a programming model is used to assemble and configure services to enhance the set of pre-defined functionalities.

5.5.1 SISI services

SISI provides a modular architecture that allows an easy definition of new functionalities implemented as building blocks in Perl. These building blocks, that derive from SISI superclasses, are packaged into *Plugins* and produce transformations on the information stream as it flows through them.

Moreover, they can be combined in order to provide complex functionalities (i.e. a translation service followed by a compression service). Thus, multiple Plugins can be composed into SISI advanced adaptation services, and their composition is based on preferences specified by end users.

Developing a SISI service consists of the following basic phases:

A - Writing the service module. The programmer can choose among two different possibilities: either implementing the service, from scratch, by using mod_perl and by following the rules dictated by Apache module

programming, or writing a simple Perl script by using, then, the SISI Deploy module to load it into Apache;

B - Registering the service. The module is added to the Apache pool of modules and registered in the SISI FilterPlugin module;

C - Writing the service configuration files. To configure the service, it is necessary to write simple HTML and XML files.

An example of how a SISI service is implemented follows.

As previously stated, the implementation undergoes three phases:

A - Writing the service module. To show how to implement a SISI service we refer to a simple example of a `RemoveLink` Plugin, which searches for all the links embedded in a Web page and substitutes them with plain text, whose color can be chosen by the user. The source code of the service is at the end of this subsection.

Line 1 defines the Apache module name, a Plugin class that extends the HTTPGenerator class (line 13), that is part of the SISI core APIs.

The `new()` method (lines 14-22) is invoked by the FilterPlugin module when the user adds the service to his/her personal profile. The method that encompasses service functionalities, is the `handler()` method (lines 23-52), invoked when the service is scheduled by the SISI Parser module. One of the main goals of the method is to instantiate and register the Generator in the pool of Apache modules (line 27); then, it copies the HTML content from the `$f Apache2::Filter` object to a `$content` buffer (lines 38-46) that is then passed to as argument (line 58) to the method `runCore()` that implements the semantics of the service. In lines 48 and 59 the `handler` method logs information into the Apache error log file including Package name, the input stream and the output stream of the service. Log information can be visualized by the Visual Monitor, described later in Section 5.6.1 and shown in Figure 5.5.

The `runCore()` method is called the *service method* i.e., the only part that is heavily dependent on the semantics of the service; it uses SISI HTMLParser to place all the links in the HTML page into a TagIterator (i.e. the Perl class that models HTML tags in SISI) (lines 68-69). Each Tag is, then, replaced with a new Tag that is set to the content of the old Tag (i.e. the anchor text) with the requested color.

The return value of the service method represents the modified HTML content and is written into the Apache response and returned to the client. The compositional nature of SISI programming model makes possible, however, to invoke successive services if the FilterPlugin module matches another filter by allowing easy services chaining.

It should be easily noticed that the only parts that are really dependent on the specific service are:

- the module name (i.e. line 11 in the example above);
- the content types (MIME types) which the service is applied on (i.e. line 33);
- the parameters of the service method (i.e. line 58);
- the definition of the service method (i.e. method `runCore()` on lines 64-81).

B - Registering the service. In order to register the service the Apache main configuration file (i.e. `httpd.conf`) is modified by adding the `PerlModule Apache::ServiceName` directive; the FilterPlugin module is changed by inserting the following lines in order to add the new service into the pool of Apache modules:

```
1  if(LibProfiles::SISI_getvaluemod($user, 'RemoveLink', 'activate') eq "on") {
2    my $refRL = MyApache::RemoveLink->new($f,"RemoveLink","true",\$content);
3    HttpPlugin::AddFilter($refRL);
4    undef($refRL);
5  }
```

C - Writing the service configuration files. To configure the service, it is necessary to write simple HTML and XML files. The HTML file shows to users, during the configuration phase, information about the service's functionalities and the service's parameters. The service parameters are used to set Perl variables that are used by the service method and can be configured for each user. In the example above, the parameter is the color that is defined by the `$list` variable (lines 52-56) and passed to the `runCore` method (line 58). The XML file contains information about the default parameter values of each implemented service (also shown to users during the configuration phase).

```
1  package MyApache::RemoveLink;
2   use Apache::Parser ();
3   use Apache2::Filter (); #$f
4   use Apache2::RequestRec (); #$r
5   use HttpGenerator ();
6   use LibProfiles;
7   use Tracing;
8   use Apache2::Const -compile => qw(OK DECLINED);
9   use Tag;
10  use TagIterator; use HTMLParser;
11  use constant MODNAME =>'RemoveLink';
12
13 @ISA=qw(HttpGenerator); sub new {
14      my $first=shift;
15      my $f=$_[1];
16      my $mod_name = $_[2];
17      my $classe = ref($first) || $first;
18      my $this = $classe->SUPER::new($f, MODNAME,"true", $_[4]);
19      bless $this, $classe;
20      return $this;
21 } sub handler {
22      my $f = shift;
23      my $stream = shift;
24      my $r = $f->r;
25      my $parser = Apache::Parser->new(MODNAME);
```

```
26      my $hostname = $r->hostname;
27      my $f_name = $r->filename;
28      my $uri = $r->uri;
29      my $user = $r->user;
30
31      unless ($r->content_type =~ m!text/html!i) {
32          $log->info('skipping request to ', $uri, ' (not an HTML document)');
33          return Apache2::Const::DECLINED;
34      }
35
36      my $content = $$stream;
37      if(!($content)) {
38          my $buffer;
39          while(1){
40              my $len=$f->read($buffer, 4092);
41              last if(($f->seen_eos)||($len==0));
42              $content = "$content"."$buffer";
43           }
44      }
45
46      Tracing::tracing_msg($r, __PACKAGE__, "InputStream", $content);
47      if($content){
48           my $param_list = $parser->getAllParamsName();
49           my @servParameters = split(/,/, $param_list);
50           my $elem=undef;
51           foreach $elem (@servParameters){
52              my $list.= $parser->getParams($user, $elem);
53              my $list.=",";
54          }
55          chomp($list);
56          $$stream = runCore ($r, $content, $list);
57          Tracing::tracing_msg($r, __PACKAGE__, "OutputStream", $$stream);
58          undef $content;
59      }
60  1;
61
62  sub runCore {
63      # the parameters are: the request, the input HTML stream
```

```
64      # and the color used for the text that substitutes the link
65      my ($r, $buffer, $color) = @_;
66      my $h_parser = HTMLParser->new(2, $buffer);
67      my $linkIter = TagIterator->new();
68      $linkIter = $h_parser->searchAllLinks();
69      my $tag;
70      while($linkIter->hasNext()) {
71          $tag = $linkIter->nextTag();
72          my $cnt = $tag->getContent();
73          my $replaceTag = Tag->new();
74          $color = "red" unless($color);
75          $replaceTag->initTagByString("<font color=".$color.">$cnt<\/font>");
76          $h_parser->replaceTag($tag, $replaceTag);
77      }
78      return $h_parser->getContent();
79  }
```

5.5.2 Easy SISI Services Deployment

The process of deploying a SISI service is partially automated since it can be performed by simply uploading a Perl fragment (i.e. the service method) plus few additional info by using an HTML form.

Automated deployment allows a quick and effective life-cycle management of the services, since a service can be developed off-line, as a traditional Perl program, accessing locally stored HTML files that act as test-bed for the filtering that is required, and, when ready, the Perl program can be simply deployed on the intermediary adaptation server.

In practice, the deployment supports the programmer (partially) in phase A and (completely) in phases B and C of the service development. In fact, once the programmer supplies, through the forms in the Deploy.html file (as shown in Figure 5.3), the module name, the content types the service is applied onto (as a regular expression), the definition of the method with its parameters, and the service parameters (to be used by the method), the deployment module Deploy.pl provides the necessary actions to add the new

service into SISI.

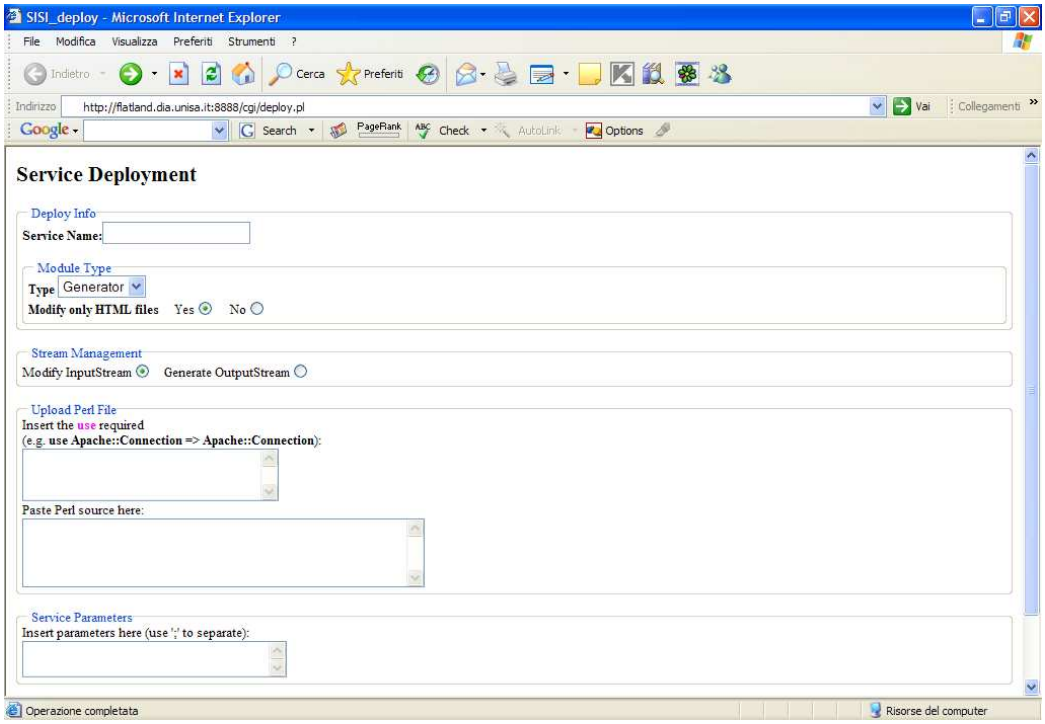

Figure 5.3: The SISI Deployment: the HTML forms used by the programmer to provide information about the new service to load into Apache

In Figure 5.4 we show all the steps performed by the SISI Deploy.pm module, and in particular, the creation of the Apache module (Service.pm) that encapsulates the semantic of the service (i.e. the `RemoveLink.pm` module described in Section 5.5.1), the update of the Apache `httpd.conf` configuration file with the directive `LoadModule` <`MyApache::ServiceName`>, the update of the SISI FilterPlugin.pm module in order to add the new service into the pool of Apache modules, and finally, the creation of the HTML file that shows information about the service's functionalities and the service's parameters.

After the Apache server has been (automatically) restarted, the added service is available for dynamic invocation by the FilterPlugin Module according to user's profile and service's constraints.

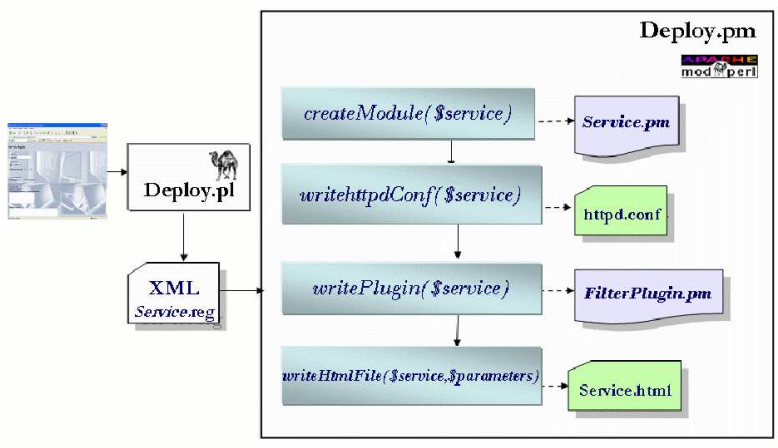

Figure 5.4: The SISI Deployment

5.5.3 HTML/HTTP Parsing Library

Dealing effectively with HTTP requests and responses requires a certain amount of support to the programmer. SISI provides utilities APIs and libraries to simplify the implementation of new services.

The HTML/HTTP Parsing Library, implemented in Perl, realizes an efficient parsing of HTML documents. Several methods have been provided to programmers, such as methods to search for links (HTML `<A>` tags), images, scripts in a Web page.

The SISI HTMLParser is a simple but highly efficient parser for HTML documents. The goal was to obtain a simple design and a quick and effective stream-oriented filter for HTML. The two main cases handled by the parser are the extraction and the transformation of the HTML stream.

The extraction allows us to take very useful information from an HTML document, such as:

- text extraction, to be used, for example, for text search engine databases;
- link extraction, to crawl through Web pages or harvest email addresses;
- resource extraction, like collecting images or sounds embedded in HTML documents;

- link checking, ensuring links are valid;
- site monitoring, checking for significant page differences i.e. beyond a simplistic `diff`.

The transformation includes all processing where the input and the output are HTML documents. Some examples are:

- URL rewriting, modifying many or all links on a page;
- site capture, moving content from the Web to local disk;
- censorship, removing offending words and phrases from Web pages;
- HTML cleanup, correcting erroneous or non-standard pages (e.g. pages that do not adhere to the W3C recommendations for users with disabilities [WWW05]);
- ad removal, by eliminating URLs that reference advertising;

The SISI HTMLParser is easy to use. The two most important methods are: `SearchGeneralTag` searches for a specific tag in the input page and returns a `TagIterator` containing all the Tag objects found; `ReplaceTag` replaces a tag with another tag (both passed as arguments). The two arguments passed in `SearchGeneralTag` represent the `NAME` attribute and a regular expression as `VALUE` attribute: in this case, the search is restricted to all tags where the `NAME` attribute has a value matching the specified regular expression. In this way, the same method can be used to search for links, images or scripts (or anything else) in the input page. In the example provided in Section 5.5.1, the HTMLParser was used to remove all the links and replace them with the corresponding anchor text (lines 58-69).

5.6 SISI Management and Configuration

The management and configuration capabilities of an environment for advanced adaptation services are a crucial factor to quickly respond to the

"everything changes" paradigm of the World Wide Web. SISI was designed as a programmable intermediary adaptation server where both management (from the system manager) and personal configuration (from each user) are provided by easy-to-use and, yet, powerful tools.

5.6.1 SISI Visual Monitor

SISI Visual Monitor (SISI VM) is the Graphical User Interface implemented to simplify the management and the debugging of the SISI components. The goal is to relieve the system administrator and programmers from learning or remembering complex and tedious commands during the administration phase and the debugging of new deployed modules.

The GUI requires Perl/Tk, which is used to write truly portable cross-platform GUI applications (they work similarly across Win32, Macintosh, Linux, and even the AS/400). The GUI also requires the MiniXML Perl module that provides an easy, object-oriented interface for manipulating XML documents and their elements without including external modules. The GUI is not automatically loaded at the start-up of the SISI framework but used only if needed.

The main features of the SISI VM are: execution control, activity tracing and, finally, the management of users and services configurations.

The debugging and the monitoring of the functionalities provided by the SISI GUI concerns three different management aspects: system, user, and logs management. SISI VM is organized into three sections according to the activities to monitor. In particular the first section of the GUI shows information about the system that runs SISI and its configurations, the second one shows information about user configurations, and finally the third section shows system logging information (Apache logs).

Information collected and displayed by the GUI are organized in menus, as shown in Figure 5.5. For example, in the File Menu of the GUI the administrator can monitor the resources of the system running the intermediary framework with actions like the following:

- Open Configuration File: to read/modify/save configuration files.
- Network status: to display network configuration and activities.
- Memory status: to display system memory utilization (physical memory utilization, system swap space utilization and physical memory currently devoted to system buffers).
- Restart: reload and re-initialization of the services.

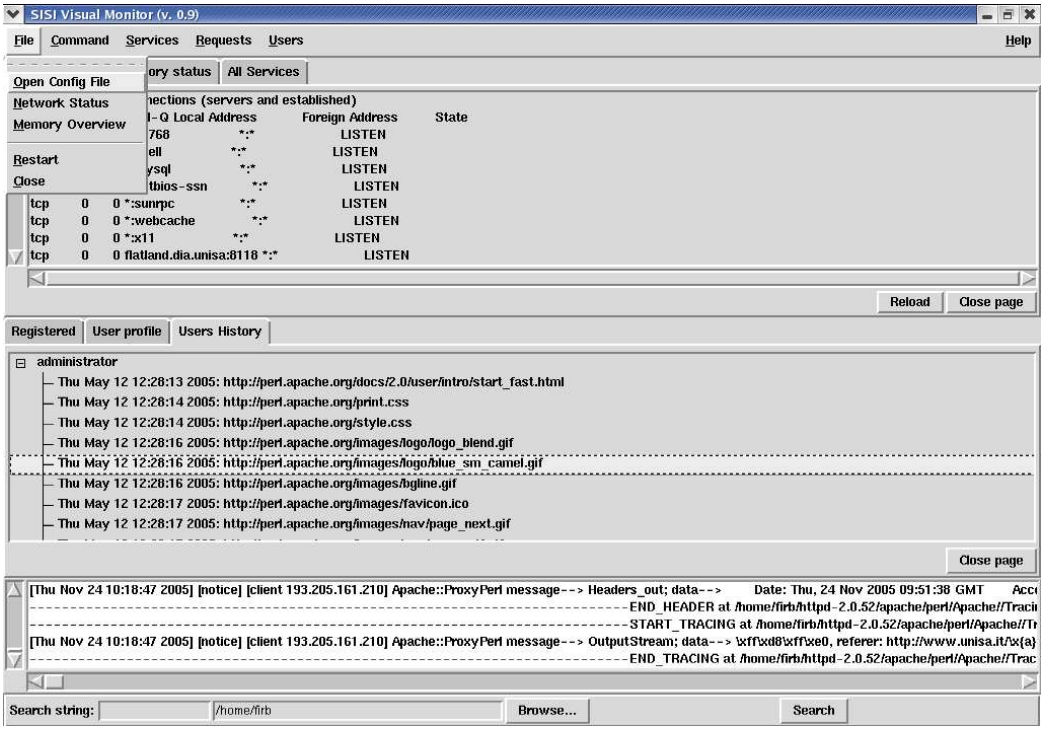

Figure 5.5: The SISI Visual monitor with the three areas shown: system, user, logs

The Command menu allows the displaying of different type of messages (Apache log messages), such as error, warning or no messages at all.

By enabling the RequestTracing from the Request Menu it is possible to monitor each HTTP request/response that is served by SISI. In particular, for any HTTP request/response, it is possible to obtain information about the

services involved in the transaction, their order (which one is being executed first, which one last), the values of the I/O data stream, the fields of HTTP request and response headers, and so on. In this way, programmers can have the full awareness of what happened into the system and test the correct execution and applicability of the services selected during the configuration phase.

In the Service Menu, services can be enabled, disabled, registered and unregistered. Registering a service means loading into Apache the module that encapsulates the service's functionality. The crucial requirement is how to inform the SISI FilterPlugin module that a new service is available and has to be added into the pool of Apache modules. The SISI GUI simplifies this task by allowing the uploading of the XML file that provides information about the service to register into the Apache pool handlers. Sometimes it can be useful to disable a specific service (by removing it from the pool of available services) in order to avoid its scheduling from the FilterPlugin module and, thus, excluding it for a specific user transaction. Finally, from the Service Menu it is possible to achieve information about the service itself such as the service's authors and other valuable information.

The User Menu allows the system administrator to trace users navigation to inspect and derive, for example, their behaviors by collecting information about users' preferences, histories, last requests, and so on.

5.6.2 Users' profiles configuration

User and device profiling is a very important characteristic because of the heterogeneity of the devices and services used by the users of the mobile Web. Users need multiple profiles according to their status or preferences, and, then, different configurations that must be easily switched to as well as modified. Of course, configuration must be performed by the user itself.

Within the SISI framework the administrator manages users' accounts, by adding or removing users from the system, resolving incorrect situations (for example, forgotten passwords), providing information about the activated

service for a given user. When a new user is added to the system, a default profile is automatically generated and the user can modify it the first time s/he logs into the system to choose his/her preferences.

Our approach to manage user profiles is to explicit ask the user what s/he needs and use this information with a rule-based approach to personalize the content. In particular, users have to fill-out forms to create new profiles and to modify or delete the existing ones, as shown in Figure 5.6.

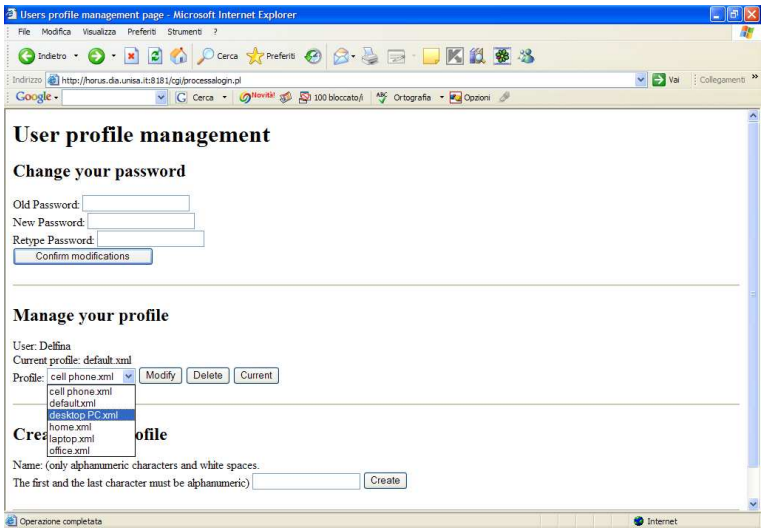

Figure 5.6: Users's Profile Management

Furthermore, because of the different client capabilities (mainly display, processing power and connectivity capabilities) of the devices used to access the Web, users must be allowed to specify, in their device's profiles, the correct and needed parameters both for devices and services. For example, when a user connects with a PDA s/he could want to be displayed only black and white images or not given images at all in order to save bandwidth. To this purpose the user has to activate the corresponding profile previously saved into the system, as shown in Figure 5.6.

SISI user-friendly configuration of services is an important feature since in this phase the users can easily provide information about the required services and personalize their navigation on the Web. In fact, by allowing

services' configuration, SISI is able to affect the adaptation of a given delivery context, and to change accordingly the user experience.

5.7 “Out-of-the-box” SISI Services

The deployment and management of many advanced mechanisms require a full support for the implementation of new services. Quick prototyping and manageability represent crucial requirements to assure that programmers can quickly respond to mutations of data format, content or standards that are so common in the present Web.

Providing new functionalities into the SISI intermediary system is a simple task, since the programmer only needs to develop the new service, by exploiting the provided set of APIs without taking care of the detail of the framework that hosts such service.

We already implemented a variety of services, that can be taken as examples by a programmer who wants to create new services. Some of them with their main goals and features are reported below.

The GIFDeanimate Service. This service parses a Graphical Interchange Format (GIF) image into its component parts and, by eliminating the animation effect, it can avoid unnecessary usage of CPU on limited resource terminals. The GIF-Deanimate service writes out a de-animated version of the original image, showing only its last frame.

The LinkRelationship Service. The HTML `LINK` element can be used to improve the Web site accessibility and, at the same time, to ensure a better support for search engines (e.g. semantic information).

As defined in W3C guidelines[1], content developers should use the `LINK` element and link types to describe document navigation mechanisms and organization. Some user agents may synthesize navigation tools or allow

[1] http://www.w3.org/WAI/

ordered printing of a set of documents based on such markup. The `LINK` element may also be used to designate alternative documents. Browsers should load the alternative page automatically based on the user's browser type and preferences. For example, content developers can produce a different content for browsers that support "braille" rendering.

Our service adds a toolbar containing the `LINK` attributes on top of each HTML page, as shown in Figure 5.7. It can also be useful to make HTML pages more accessible and their reading by screen readers, also to improve the Web navigation on devices with limited capabilities.

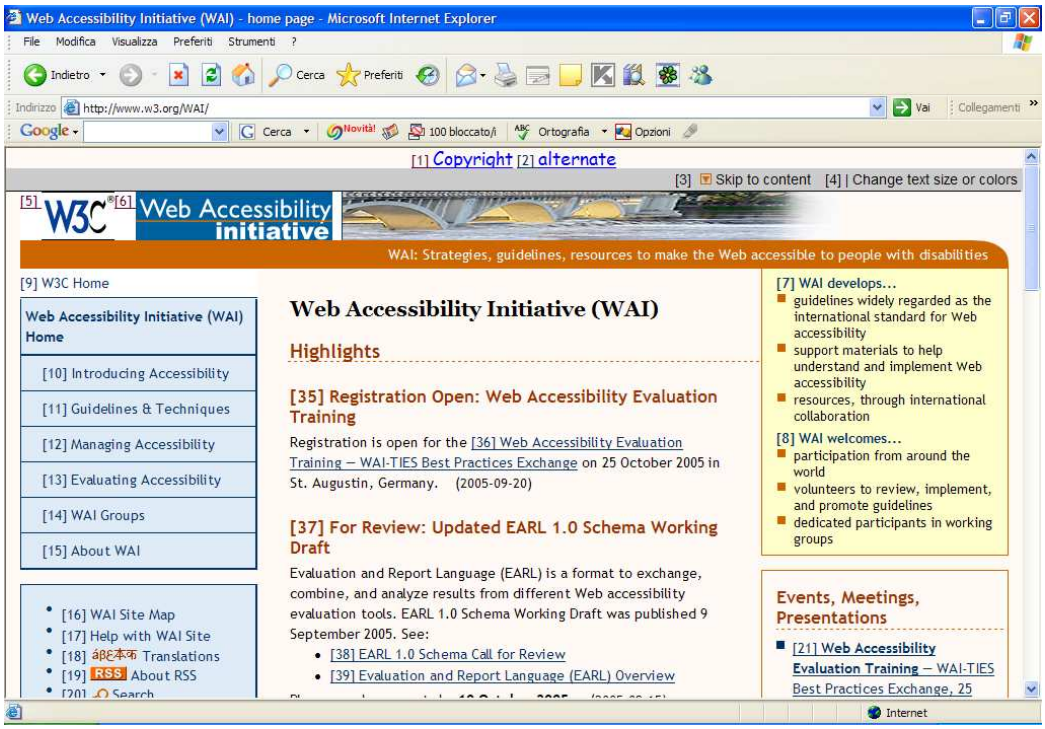

Figure 5.7: The LinkAccessKey and Link relationship services

The LinkAccessKey Service. The W3C Guideline 9 *Design for device-independence* [W3C00], in section Keyboard access, states the need to use access keys HTML elements to allow users with some disabilities to browse the Web.

As a rule, content developers should always ensure that users may interact with a page with input devices other than a mouse. Unfortunately, this is not very common and we provide a solution by developing a service that allows both motor and visual disabilities users to browse the Web without limitations.

This service adds to any link embedded in a Web page a numeric *Access Key* in such a way to make it accessible through a simple combination of keyboard keys ALT+*Access Key*+Return. Pressing the access key assigned to an element gives focus to the element, and the corresponding action will be executed. In particular by pressing the access key, the browser will follow the corresponding destination link. Moreover, the numeric value of the access key is also added into the HTML source of the Web page so it can be easily read by any screen reader. As the LinkRelationship service, also the LinkAccessKey service could be useful to improve the Web navigation on devices with limited display capabilities.

The FilterImages Service. Pervasive devices suffer from limited bandwidth and constraints on device capabilities (small screens, low depth colors, or even a black and white screen).

The FilterImages service performs a set of transformations or distillation on images. These transformations are completely user-configurable and include:

- scaling of source images down to reasonable dimensions for the client display
- reducing image quality
- transcoding into gray-scale color
- replacement of images with a link and a textual description

The FilterImages service has been implemented by using the well-known PerlMagick [Per05] library, that is an object-oriented Perl interface to ImageMagick [Ima05], a free software that allows to create, edit, and compose

images with different formats. In particular, images can be cropped, rotated and combined, colors can be changed, different effects can be applied, by adding for example, text, lines, curves on the manipulated images.

Here is a simple `imgDownGrade` service (i.e. a functionality provided by the FilterImages service), which downgrades the image resolution to a new one, chosen by the user:

```
sub imgDownGrade {
    # the parameters are: the request, the image to be processed and
    # the downgrade percentage
    my ($r, $data, $dg) = @_;
    $dg = "15" unless(defined $dg);
    my($image, $d);
    $d = $$data;

    if(defined($d)){
        $image = Image::Magick->new();
        my $x = $image->BlobToImage($d);
        $x =~ /(\d+)/;
        undef $d;
        $image->Set(quality=>$dg);
        $$data = $image->ImageToBlob();
        undef $image;
    }
}
```

Because distillation can be performed in real time, it eliminates the need for content providers to maintain multiple intermediate-quality representations of a document. Distillation, thus, allows bandwidth to be managed in a way that exploits the client's strengths and limitations [FB96]. Still, a cached copy of adapted resources in the intermediary adaptation server could speed up performance, so we are planning to study the pros and cons of integrating a caching mechanism directly into SISI.

The ImageRemoval Service. Images embedded in Web pages represent an obstacle for the natural navigation of users with visual disabilities since

images cannot be read by a screen reader. With the aim at facilitating their navigation, we implement a service that removes images, replacing them with their textual description. The service intercepts the *“img”* tags from the HTML page and replaces them with a comment that represents the content of the HTML *“alt”* attribute or the image file name.

The TrafficLights Service. The goal of the TrafficLights service is to test the availability of the links of Web pages navigated by end users, in order to detect the broken ones and then avoiding the users for surfing them. It is a perfectly suited service to users who want to avoid overcrowded links that dynamically change the quality of user experience of the Web. A similar (but more complex) service was among the first ones to be provided by IBM's Web Based Intermediaries [IBM]. For each link embedded in a Web page, the TrafficLights service sends a TCP `SYN` packet to the remote host, then immediately returns. If the remote host replies with TCP `ACK` within a specified timeout, the remote host is considered reachable. According to the estimated responses times, it adds an intuitive colored image at each link. A green color image states that the link is up and quickly reachable (time less than 200ms), a yellow color image states a medium connection (within 200 and 500ms) and, finally, a red color image states that the link connection is very low or the link is broken or unreachable.

The HtmlClean Service. The main goal of this service (based on the `HTML::Clean` Perl library) is to clean HTML pages from useless or redundant code. It is very useful for Web pages built with programs that silently add internal code. Other functionalities include removing white spaces, useless (i.e. non-standard) META elements, HTML comments, replace tags with equivalent shorter tags. By using the provided methods, the service is able to reduce the size of Web pages thus speeding up the download.

The BlockList and the AnnojanceFilter Services. The main goal of these services is to get rid of particularly annoying abuse during the naviga-

tion on the Web. In particular, the BlockList service provides a simple way to block a Web browser from viewing sites that are not on a list of approved sites. This service searches for all the links embedded in a Web page and substitutes the unapproved ones with plain text.

The AnnojanceFilter service provides functionalities for removing advertisement, banners, pop-ups in JavaScript and HTML, JavaScript code, for disabling unsolicited pop-up windows, etc. During service's configuration users can choose the functionality to enable by providing parameters if required.

Device Independence. The W3C draft on Content Selection [DIW05] specifies a syntax and a processing model for general purpose content selection or filtering. Selection involves conditional processing of XML information according to the results of the evaluation of expressions. Using this mechanism some parts of the information can be selected while other not delivered, automatically adapting the original content according to particular accessibility rules.

The main goal of the Device Independence Working Group (DIWG) is to realize a markup language that supports the creation of Web sites and presentations suitable for a wide variety of devices with a wide variety of characteristics. To this end, the main work is based on the development of a device independent profile for XHTML.

In the SISI Content Selection service (SISI CS service) we are implementing the specifications of the draft [GMMS05]. We have currently implemented the conditional expressions and the conditional expressions that return values, that can be further used to check if a particular piece of content is to be included for processing in the Web page delivered to end users. The remaining part of the specification is currently under testing or being developed.

If the end user aims to use the content selection service, s/he has to configure service's parameters in order to meet the capabilities of the accessing terminal device. If the used device is a mobile one with a small display, the

user can accordingly set height and width parameters, specifying the number of pixels that the device is able to support.

The CS service intercepts user's request, reads the user's profile and the service user-defined parameters (in this example, height and width), retrieves the Web page from the Web server (the original Web pages augmented with the DISelect markup expressions), performs some computation and, finally, delivers only the content that satisfies the specified conditional expressions and rules. Computations include the parsing of the HTML Web page to pull out the `DISelect` tags, invocation of the functions according the matched expressions to validate the values of the corresponding variables.

5.8 Performance Evaluation

The SISI framework is implemented on top of the Apache Web server software. The prototype is extensively tested to verify its scalability and to compare its performance against other intermediary adaptation servers.

A first set of experiments focuses on the response times the user gets when connecting to an intermediary adaptation server, instead of directly connecting to the origin Web server.

A second set of experiments aims at comparing SISI against Muffin [Boy]. It is worth noting that SISI supports user authentication, while Muffin does not, hence we can anticipate that the performance comparison is rather unfair in favor of Muffin.

A third set of experiments aims at verifying the SISI scalability by applying an increasing number of concurrent services.

In all the experiments, the request rate is referred to a whole Web page, including the HTML base container and its embedded objects.

5.8.1 Workload Model and Testbed

We set up a testbed platform consisting of three nodes connected through a switched fast Ethernet LAN (Figure 5.8). This choice allows us to avoid

possible non predictable network effects and is considered the fairest way to compare the considered frameworks for content adaptation.

The client node runs *httperf* [MJ], which is a tool to generate various HTTP workload models. A second node runs the intermediary adaptation server. Finally, a third node runs an Apache Web server and a MySQL database. The origin resources are a replica of a real Web site. The basic workload is created on the basis of the request logs to this site. Moreover, in our tests the same trace is replayed at different request rates until reaching the maximum capacity of the intermediary adaptation server. Requests are referred to a mix of content types consisting of images (49%), HTML documents (27%), others (24%). HTML resources typically contain embedded objects ranging from a minimum of 0 to a maximum of 25, with a mean value of 10. Embedded objects are images (GIFs and JPEGs), cascading style sheets (CSS) or Flash animations (SWF). Animated GIF images are about 6% of the whole workload.

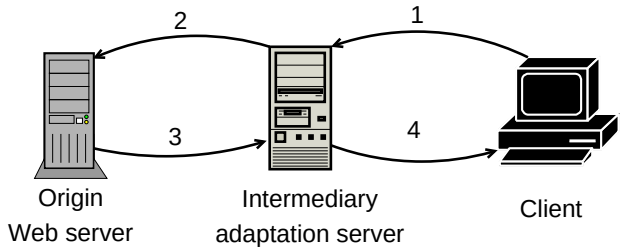

Figure 5.8: The experimental testbed: the client requests resources through an intermediary adaptation server

The first test about the overhead of the intermediary node is done on a cluster consisting of three nodes all equipped with a hyper-threaded Intel Xeon@2.4GHz and 1GB RAM, running Debian GNU Linux with kernel 2.6.13.

We carry out the other experiments on three nodes with different capacity. The origin Web server is hosted on a node equipped with a Pentium 4@1.8GHz and 512MB RAM, running Gentoo Linux with kernel 2.6.11.

The intermediary adaptation server runs on a node equipped with a Pentium 4@2.4GHz and 1GB RAM, running Fedora Core 3 Linux with kernel 2.6.9. The *httperf* load generator runs on a node equipped with a Pentium 4@1.8GHz and 1.5GB RAM, running Gentoo Linux with kernel 2.6.11.

5.8.2 Overhead of the intermediary adaptation server

In this section we evaluate how much the user response time is affected by an intermediary adaptation server in the path between the client and the origin Web server.

We configured *httperf* to request 5000 HTML pages with the related embedded objects directly to the origin Web server and then to request the same resources through the intermediary adaptation server. To the purpose of evaluating the impact of the intermediary adaptation server overhead to the user response time, the intermediary adaptation server forwards the requested resources from the origin Web server to the users, without applying any content adaptation to the resources. Hence, this latter scenario evidences just the cost related to the three way handshaking to open the connection between the intermediary adaptation server and the origin Web server and the resource transfer time.

There is a main difference between SISI and the other intermediary adaptation servers, since SISI performs user authentication and has to load a specific profile for each user, instead of loading a system-wide profile. As shown in Figure 5.9, the client requests a resource to the SISI framework (step 1), SISI asks the client to authenticate (step 2), the client sends username and password (step 3), SISI loads the profile for the user issuing the request (step 4) and understands no adaptation services have to be applied to the request. SISI requests the resource to the origin Web server (step 5). The origin Web server delivers the requested resource to the SISI framework (step 6) and SISI delivers the requested resource to the client (step 7).

Figure 5.10 shows the 90-percentile of the page response time in milliseconds for various intermediary adaptation servers and the origin Web server

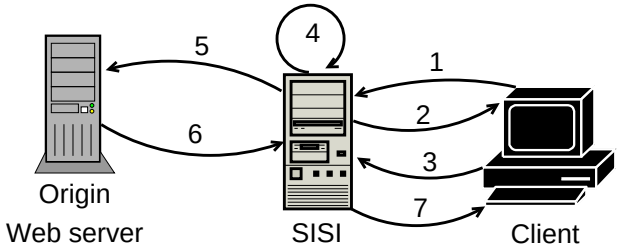

Figure 5.9: The client requesting a resource through the SISI framework

when resources are requested at a slow rate of 10 pages per second.

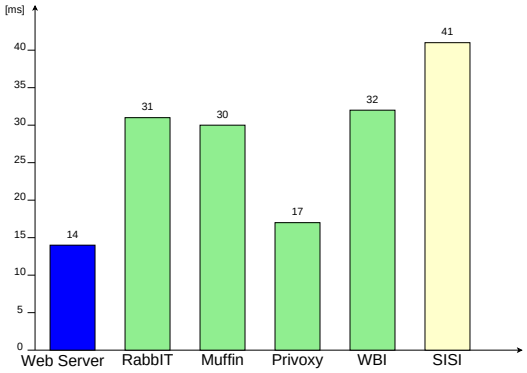

Figure 5.10: 90-percentile of the page response time

Although SISI adds a major overhead, we should consider that this framework provides additional services such as user authentication and per-user profile loading.

5.8.3 Performance Comparison

In this section we compare Muffin and SISI performance by considering Muffin Painter service and SISI RemoveBackground service. Both services parse the HTML content of a Web page, search for the tags that specify background colors and images and substitute them according to the user preferences.

We set up *httperf* to request 3000 HTML pages and the related embedded objects at varying request rates. Figure 5.11 and Table 5.2 show the average response time when Muffin or SISI are contacted as intermediary adaptation

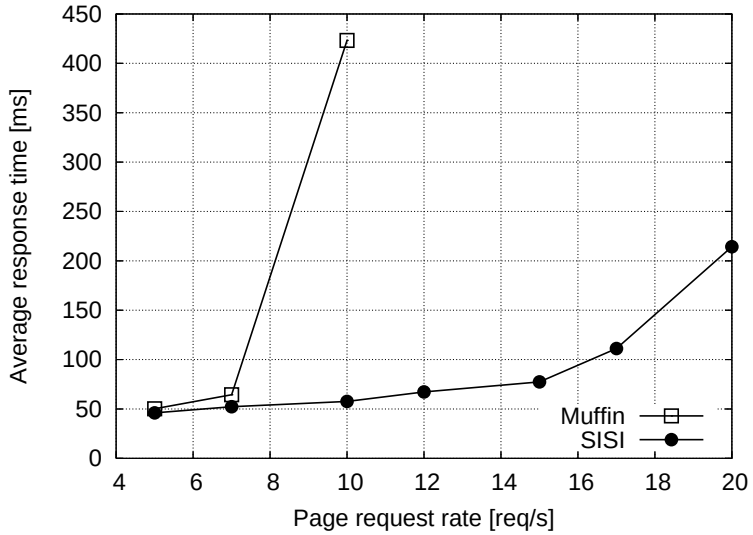

Figure 5.11: Average response time for Muffin and SISI when removing background colors and images

Table 5.2: Average response time [ms] as a function of the page request rate

	Page Request Rate							
	5	**7**	**10**	**12**	**15**	**17**	**20**	**22**
Muffin	50.2	64.4	423.2	x	x	x	x	x
SISI	46	52.3	57.6	67.2	77.4	111.2	214.3	x

servers, respectively. An "x" sign in Table 5.2 (as well as in the following) means that the intermediary is overloaded.

SISI always shows better response times than Muffin, for instance at a rate of 7 pages per second SISI is nearly 20% faster than Muffin. Furthermore, Muffin shows poor performance: it can sustain a rate of 7, while SISI up to 20 pages per seconds, nearly tripling the sustainable load. This shows that in some cases SISI can even outperform other intermediaries.

5.8.4 SISI Scalability

In this section we evaluate the SISI scalability by choosing the RemoveBackground and the GIFDeanimate services. The RemoveBackground service parses the HTML content of a Web page, searches for the tags that specify background colors and images and substitutes them according to the user preferences, while the GIFDeanimate service parses a GIF image into its component parts. The GIFDeanimate service produces a de-animated version of the original image, by showing only its last frame. Its main goal is to save bandwidth in the delivery of Web pages with a lot of animated embedded images.

We set up *httperf* to request 3000 HTML pages with the related embedded objects at varying request rates. The first two rows of Table 5.3 show the average response time when the RemoveBackground and the GIFDeanimate services are applied to the requests, respectively.

With the RemoveBackground service SISI shows a limited increase in the user response time up to a rate of 20 pages per second, while with the GIFDeanimate service such increase is linear up to a rate of 17 pages per second. The fact that the system gets overloaded earlier is a sign that GIFDeanimate is heavier than the RemoveBackground service, as we could expect, since it has to parse images instead of text.

We also want to test the SISI scalability when more services are combined and applied to the same request. To this purpose, we set up SISI to apply both RemoveBackground and GIFDeanimate services to each request.

httperf is set to issue 3000 HTML requests at varying rates. The last row of Table 5.3 shows the average response time when both the GIFDeanimate and the RemoveBackground services are applied to the requests. The time necessary to apply more than one service is far less then the sum of the response times obtained when the single services are applied to the requests. This happens because services can be easily and quickly chained one after the other: the output of the first applied service becomes the input of the

Table 5.3: SISI: Average response time [ms] with RemoveBackground and GIFDeanimate services

	Page Request Rate						
	5	**7**	**10**	**12**	**15**	**17**	**20**
RemoveBackground	46	52.3	57.6	67.2	77.4	111.2	214.3
GIFDeanimate	54.7	57.7	63.4	68	78.4	124.9	x
Both services	57	59	64.7	70	89.9	154.9	x

second service one and so on. Many operations related to each request, such as user authentication, profile loading, origin Web page fetching only occur one time, independently of how many services are applied to the request. The results on the last row of Table 5.3 show that SISI is very effective in chaining services, in fact it adds a very small overhead that is negligible with respect to the time spent to apply the services. We can notice that the response times with both services applied to the requests are similar to those of the GIFDeanimate service alone.

Chapter 6

Conclusions

In this thesis we propose efficient architectures and mechanisms to support a high quality usage of Web-based services. Indeed, present users want to download content in an efficient way, and want content suiting their personal preferences and matching their device capabilities.

We separate in three logical levels the infrastructure that is able to provide such services and provide innovative contributions for each level.

6.1 Results

Let us summarize the main research contributions.

- We propose a novel approach to the design of Web systems that take into account performance constraints, even for unpredictable request frequencies. It is conceptually valid for every multi-level Web-based service, although we focus on dynamic Web sites because this category is widely diffused and it presents interesting design challenges. Furthermore, we propose an innovative methodology for the performance evaluation and the bottleneck analysis. We demonstrate that, if a coarse grain view of the system performance may be useful for the bottleneck localization, it is necessary to pass to a fine grain analysis to understand the motivations and remove the problems. We also

notice the importance of using percentiles and cumulative distribution functions instead of average values, because Web-based services are characterized by complex sub-system interactions, burst arrivals and heavy-tailed workload models.

- We propose the first thorough comparison of existing algorithms for efficient Web content delivery through distributed Web caching systems. We evaluate the scalability of hierarchical and flat cooperation architectures, with the goal of identifying the limits of these architectures and cooperation schemes. Moreover, we propose innovative architectures, algorithms and protocols for cooperative Web caching and delivery. We provide a thorough performance evaluation of the novel two-tier protocols, demonstrating that they are a scalable solution for Web delivery and limit the variance in response time, thus improving the stability of performance results.

- We present a novel framework, namely Scalable Intermediary Software Infrastructure (SISI), that aims at facilitating the deployment of adaptation services of Web content. SISI is characterized by a modular architecture that allows an easy definition of new functionalities implemented as building blocks in Perl language. The prototype is extensively tested. Our experiments demonstrate that SISI is a viable and scalable solution to deploy adaptation services on the WWW. Indeed, despite its flexibility, SISI does not penalize the users with long response times.

6.2 Future directions

The core of the future research is dealing with the adaptation level of the Web-based infrastructure. The main goal is to improve scalability to support a very high number of concurrent requests. To this purpose, we are planning to distribute the adaptation process among different nodes in a clustered

environment, in order to support a very high number of concurrent requests. We are considering how to replicate the intermediary adaptation servers and propose new policies to achieve a good load sharing.

We also aim to improve the adaptation process, by adding some caching mechanisms to the intermediary adaptation servers. Caching user profiles and adapted resources should reduce the server load, as well as the user perceived latency.

Finally, we are studying a way to integrate SISI with an existing middleware for context-awareness [GMMR06]. Users will only have to specify which services have to be applied to their requests. The majority of information regarding the technical specification of the client device and context-related information, for instance the user geographical location, will be directly sent by the context-awareness middleware, without user intervention.

Appendix A

List of Acronyms

API = Application Program Interface
AS = Autonomous System
ASP = Active Server Pages
CARP = Cache Array Routing Protocol
CD = Cache Digests
CDN = Content Delivery Network
CGI = Common Gateway Interface
CMP = Cache to Master Protocol
CPU = Central Processing Unit
CSS = Cascading Style Sheet
DBMS = Data Base Management System
DIWG = Device Independence Working Group
DNS = Domain Name System
EJB = Enterprise JavaBeans
FCFS = First Come First Served
GIF = Graphical Interchange Format
GUI = Graphical User Interface
GNU = GNU's Not Unix
HR = Hit Rate
HTML = HyperText Markup Language

HTTP = HyperText Transfer Protocol
I/O = Input/Output
ICP = Internet Cache Protocol
IP = Internet Protocol
ISP = Internet Service Provider
J2EE = Java 2 platform Enterprise Edition
JPEG = Joint Photographic Experts Group
JSP = JavaServer Pages
LAN = Local Area Network
MIME = Multipurpose Internet Mail Extensions
NLANR = National Laboratory for Applied Network Research
OS = Operating System
PC = Personal Computer
PDA = Personal Digital Assistant
QoS = Quality of Service
RAM = Random Access Memory
RPM = Rounds Per Minute
SISI = Scalable Intermediary Software Infrastructure
SLA = Service Level Agreement
SSL = Secure Socket Layer
SWF = ShockWave Format
TCP = Transmission Control Protocol
UMTS = Universal Mobile Telecommunications System
URI = Uniform Resource Indicator
URL = Uniform Resource Locator
W3C = World Wide Web Consortium
WAN = Wide Area Network
WBI = Web Based Intermediaries
WiFi = Wireless Fidelity
WRR = Weighted Round Robin
WWW = World Wide Web

XHTML = Extensible HyperText Markup Language

XML = Extensible Markup Language

XSL = Extensible Stylesheet Language

Bibliography

[ABCdO96] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizimg Reference Locality in the WWW. In *Proc. of the International Conference on Parallel and Distributed Information Systems (PDIS'96).*, pages 92–103, December 1996.

[ACC+02] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *Proc. of the IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Nov 2002.

[ACLM04] Mauro Andreolini, Michele Colajanni, Riccardo Lancellotti, and Francesca Mazzoni. Fine grain performance evaluation of e-commerce sites. *SIGMETRICS Performance Evaluation Review*, 32(3):14–23, 2004.

[ACM02] Mauro Andreolini, Michele Colajanni, and Ruggero Morselli. Performance study of dispatching algorithms in multi-tier Web architectures. *ACM Sigmetrics Performance Evlaution Review*, 30(2):10–20, 2002.

[ACM06] Mauro Andreolini, Valeria Cardellini, and Francesca Mazzoni. Evaluating QoS-aware policies for Web-based services. *SIMPRA (Simulation Practice and Theory Journal)*, 2006. to appear.

[ACN03] Mauro Andreolini, Michele Colajanni, and Marcello Nuccio. Scalability of content-aware server switches for cluster-based Web information systems. In *Proc. of the* 12^{th} *Interational World Wide Web Conference (WWW2003)*, Budapest, HU, May 2003.

[AGL+03] S. Ardon, P. Gunningberg, B. LandFeldt, M. Portmann Y. Ismailov, and A. Seneviratne. MARCH: a distributed content adaptation architecture. *International Journal of Communication Systems, Special Issue: Wireless Access to the Global Internet: Mobile Radio Networks and Satellite Systems.*, 16(1):97–115, 2003.

[AKR01] M.F. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large scale Web-based shopping system. *IEEE/ACM Trans. on Internet Technology*, 1(1):44–69, September 2001.

[AMW+03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of* 19^{th} *ACM Symposium on Operating Systems Principles (SOSP03)*, October 2003.

[Apa06] Apache Web Server, 2006. – http://httpd.apache.org/.

[ATT02] ATT. AT&T, 2002. – http://www.att.com.

[AW97] Martin F. Arlitt and Carey L. Williamson. Internet Web servers: workload characterization and performance implications. *IEEE/ACM Trans. Netw.*, 5(5):631–645, 1997.

[BM98a] R. Barrett and P. P. Maglio. Adaptive Communities and Web Places. In *Proceedings of* 2^{th} *Workshop on Adaptive Hypertext and Hypermedia, HYPERTEXT 98.*, Pittsburgh (USA), 1998. ACM Press.

[BM98b] R. Barrett and P. P. Maglio. Intermediaries: New Places for Producing and Manipulating Web Content. *Computer Networks and ISDN Systems*, 30(4):509–518, 1998.

[BM99a] R. Barrett and P. P. Maglio. WebPlaces: Adding people to the Web. In *Poster Proc. of the* 8^{th} *International World Wide Web Conference*, Toronto (Canada), 1999. ACM Press.

[BM99b] R. Barrett and Paul P. Maglio. Intermediaries: An approach to manipulating information streams. *IBM Systems Journal*, 38(4):629–641, 1999.

[Boy] M. Boyns. "Muffin - a filtering proxy server for the World Wide Web". http://muffin.doit.com.

[Car01] Valeria Cardellini. *Scalable Web server systems*. PhD thesis, University of Rome "Tor Vergata", Mar. 2001.

[CB97] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Trans. on Networking*, 5(6):835–846, December 1997.

[CCCY02] 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(2):263–311, 2002.

[CCE+03] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *Proc. of the ACM/IFIP/USENIX Int'l Middleware Conference (MIDDLEWARE2003)*, June 2003.

[CGM+05] Michele Colajanni, Raffaella Grieco, Delfina Malandrino, Francesca Mazzoni, and Vittorio Scarano. A scalable framework for the support of advanced edge services. In *Proc. of the 2005 International Conference on High Performance Computing and Communications (HPCC-05)*, pages 1033–1042, September 2005.

[Chi00] Willy Chiu. Design pages for performance. *IBM HVWS*, 2000.

[CM02] X. Chen and P. Mohapatra. Performance evaluation of service differentiating Internet servers. *IEEE Trans. on Computers*, 51(11):1368–1375, November 2002.

[CMS03] M. G. Calabrò, D. Malandrino, and V. Scarano. Group Recording of Web Navigation. In *Proc. of the* 14^{th} *conference on Hypertext and Hypermedia (HYPERTEXT'03)*. ACM Press, August 2003.

[CSD+95] A. Chankhunthod, M. Schwartz, P. Danzig, K. Worrell, and C. Neerdaels. A Hierarchical Internet Object Cache. In *Proc of Usenix Annual Technical Conference*, 1995.

[Dan97] P. Danzig. NetCache Architecture and Deployment. In *Proc. of* 3^{rd} *W3 Cache Workshop*, Feb. 1997.

[Dav01] B. D. Davison. A Web Caching Primer. *IEEE Internet Computing*, 5(4):38–45, Jul./Aug. 2001.

[DB206] DB2 Universal Database, 2006. – http://www-306.ibm.com/software/data/db2/.

[DIW05] DIWG. W3C Working Draft: Content Selection for Device Independence (DISelect) 1.0, 2005. http://www.w3.org/TR/2005/WD-cselection-20050502/.

[DMB01] Ronald C. Dodge, Daniel A. Menascé, and Daniel Barbará. Testing E-commerce Site Scalability with TPC-W. In *Proc. of 2001 Computer Measurement Group Conference*, Dec 2001.

[DR01] Sandra G. Dykes and Kay A. Robbins. A Viability Analysis of Cooperative Proxy Caching. In *Proc. IEEE Infocom*, pages 1205–1214, Apr. 2001.

[DR02] S. G. Dykes and K. A. Robbins. Limitations and Benefits of Cooperative Proxy Caching. *IEEE Journal on Selected Areas in Communication*, 20(7), Sept. 2002.

[DZY01] M. Dikaiakos and D. Zeinalipour-Yiazti. Distributed Middleware Infrastructure for Personalized Services. Technical Report TR-2001-4, University of Cyprus, December 2001.

[ENTZ04] 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 Int'l Conference on World Wide Web (WWW2004)*, May 2004.

[FB96] Armando Fox and Eric A. Brewer. Reducing WWW latency and bandwidth requirements by real-time distillation. In *Proceedings of the* 5^{th} *international World Wide Web conference on Computer networks and ISDN systems*, pages 1445–1456, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[FCAB98] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proc. of SIGCOMM '98*, 1998.

[FHBH+99] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Peach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication, June 1999. RFC 2617.

[GCR98] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Proc. of Workshop on Internet Server Performance (WISP'98)*, 1998.

[GMM+05] Raffaella Grieco, Delfina Malandrino, Francesca Mazzoni, Vittorio Scarano, and Francesco Varriale. An Intermediary Software Infrastructure for Edge Services. In *Proc. of the* 1^{st} *Int. Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI'05) in conjunction with the* 25^{th} *International Conference on Distributed Computing Systems (ICDCS'05)*, pages 259–265, June 2005.

[GMMR06] Raffaella Grieco, Delfina Malandrino, Francesca Mazzoni, and Daniele Riboni. Context-aware Provision of Advanced Internet Services. In *Proc. of the* 4^{th} *Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2006)*, Pisa, Italy, Mar. 2006.

[GMMS05] Raffaella Grieco, Delfina Malandrino, Francesca Mazzoni, and Vittorio Scarano. Mobile Web Services via Programmable Proxies. In *Proc. of the IFIP TC8 Working Conference on Mobile Information Systems - 2005 (MOBIS)*, Leeds, UK, December 2005.

[God04] Sébastien Godard. Syssyat: System performance tools for Linux OS, 2004. – http://perso.wanadoo.fr/sebastien.godard/.

[GRC97] S. Gadde, M. Rabinovich, and J. Chase. An Approach to Building Large Internet Caches. In *Proc.* 6^{th} *Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May 1997.

[Hem04] Stephen Hemminger. netem: Network Emulator, 2004. – http://developer.osdl.org/shemminger/netem/.

[HKO+00] M. Hori, G. Kondoh, K. Ono, S. Hirose, and S. Singhal. Annotation-Based Web Content Transcoding. In *Proceedings of the* 9^{th} *International World Wide Web Conference*, Amsterdam (The Netherland), 2000. ACM Press.

[HNY01] Yiming Hu, Ashwini Nanda, and Qing Yang. Measurement, Analysis and Performance Improvement of the Apache Web Server. *International Journal of Computers and Their Applications*, 8(4), Dec. 2001.

[HY00] Xubin He and Qing Yang. Performance Evaluation of Distributed Web Server Architectures under E-Commerce Workloads. In *Proc. of the 1st Int'l Conference on Internet Computing (IC'2000)*, Jun 2000.

[IBM] Web Based Intermediaries (WBI). http://www.almaden.ibm.com/cs/wbi/.

[IBM02] IBM. IBM WebSphere Transcoding Publisher, 2002.

[IKLR03] Arun Iyengar, Richard King, Heiko Ludwig, and Isabelle Rouvellou. Performance and Service Level Considerations for Distributed Web Applications. In *In Proc. of the 7^{th} World Multiconference on Systems, Cybernetics, and Informatics (SCI 2003)*, Jul 2003.

[Ima05] ImageMagick 6.2.5, 2005. http://www.imagemagick.org/script/index.php.

[IRC95] IRCache. IRCache Project, 1995. – http://www.ircache.net.

[Jac95] Van Jacobson. How to kill the Internet, Aug. 1995. Talk in SIGCOMM '95 Middleware Workshop.

[JKB03] Kai S. Juse, S. Kounev, and A. Buchmann. PetStore-WS: Measuring the Performance Implications of Web Services. In *Proc. of the 29th Int'l Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2003*, December 2003.

[KO00] F. Silva K. Obraczka. Looking at network latency for server proximity. In *Proc. of the IEEE Globecom 2000*, Dec. 2000.

[KWZ01] B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proc. Internet Measurement Workshop*. ACM SIGCOMM, 2001.

[LA94] A. Luotonen and K. Altis. World-Wide Web proxies. *Computer Networks and ISDN Systems*, 27(2):147–154, 1994.

[LASV02] K. W. Lee, K. Amiri, S. Sahu, and C. Venkatramani. On the Sensitivity of Cooperative Caching Performance to Workload and Network Characteristics. In *Proc. of ACM Sigmetrics*, Jun. 2002.

[LCC02] Riccardo Lancellotti, Michele Colajanni, and Bruno Ciciani. A Scalable Architecture for Cooperative Web Caching. In *Proceedings of Web Engineering Workshop, Networking2002*, May 2002.

[LCC03] Riccardo Lancellotti, Michele Colajanni, and Bruno Ciciani. A Distributed Cooperation Schemes for Document Lookup in Geographically Dispersed Cache Servers. In *Proc. of IEEE International Symposium on Network Computing and Applications (NCA2003)*, Cambridge, MA, USA, Apr. 2003.

[Lev04] John Levon. Oprofile - A system profiler for Linux, 2004. – http://oprofile.sourceforge.net/.

[LMC03] Riccardo Lancellotti, Francesca Mazzoni, and Michele Colajanni. Scalability of Cooperative Algorithms for Distributed Architectures of Proxy Servers. In *Proc. of the International Conference WWW/Internet 2003 (ICWI'2003)*, pages 458–466, November 2003.

[LMC05] Riccardo Lancellotti, Francesca Mazzoni, and Michele Colajanni. Hybrid cooperative schemes for scalable and stable performance of Web content delivery. *Computer Networks*, 49(4):492–511, Nov. 2005.

[MBD01] Daniel A. Menascé, Daniel Barbará, and Ronald Dodge. Preserving QoS of e-commerce sites through self-tuning: a performance model approach. In *Proc. of the 3rd ACM conference on Electronic Commerce*. ACM Press, 2001.

[McM99] Patrick McManus. A passive system for server selection within mirrored resource environments using as path length heuristics, Apr. 1999. – http://proximate.appliedtheory.com/.

[Mic06a] Microsoft Internet Information Services, 2006. – http://www.microsoft.com/windowsserver2003/iis/evaluation/default.mspx.

[Mic06b] Microsoft SQL Server, 2006. – http://www.microsoft.com/sql/default.mspx.

[MJ] D. Mosberger and T. Jin. httperf, A Tool for Measuring Web Server Performance.

[mod] mod_perl. http://www.perl.apache.org.

[MS06] Delfina Malandrino and Vittorio Scarano. Tackling Web Dynamics by Programmable Proxies. *Computer Networks*, 50(14), October 2006.

[MyS04] MySQL database server, 2004. – http://www.mysql.com/.

[net05] netcraft, 2005. – http://www.netcraft.com/survey/archive.html.

[Nie94] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann, 1994.

[NLA02] NLANR. National Laboratory for Applied Network Research, 2002. – http://www.nlanr.net.

[NRSA01] Erich M. Nahum, Marcel-Catalin Rosu, Srinivasan Seshan, and Jussara Almeida. The effects of wide-area conditions on WWW server performance. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 257–267, 2001.

[Ora06a] Oracle, 2006. – http://www.oracle.com/.

[Ora06b] The Oracle Portal, 2006. – http://www.oracle.com/technology/products/ias/portal/index.html.

[PCL06] Marco Emilio Poleggi, Emiliano Casalicchio, and Riccardo Lancellotti. A Simulation Framework for Cluster-based Web services. *SIMPRA (Simulation Practice and Theory Journal)*, 2006. to appear.

[per] The Perl Programming Language. http://www.perl.com.

[Per05] PerlMagick 6.22, 2005. http://www.imagemagick.org/script/perlmagick.php.

[PH97] D. Povey and J. Harrison. A Distributed Internet Cache. In *Proc. 20th Australian Computer Science Conference*, Sydney, Australia, Feb. 1997.

[PHP04] PHP scripting language, 2004. – http://www.php.net/.

[pos05] postgreSQL database server, 2005. – http://www.postgresql.org.

[PP01] A. Papoulis and S. U. Pillai. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, 2001.

[Pri] Privoxy Web Proxy. http://www.privoxy.org/.

[Rab] RabbIT proxy. http://rabbit-proxy.sourceforge.net/.

[RCCC01] C. Rao, Y. Chen, Di-Fa Chang, and Ming-Feng Chen. iMobile: A Proxy-Based Platform for Mobile Services. In *Proceedings of the 1st ACM Workshop on Wireless Mobile Internet (WMI 2001).* ACM Press, 2001.

[RCG98] M. Rabinovich, J. Chase, and S. Gadde. Not All Hits Are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network. In *Proc. of the 3rd International WWW Caching Workshop*, Jun. 1998.

[RSB99] P. Rodriguez, C. Spanner, and E. Biersack. Web caching architectures: hierarchical and distributed caching. In *Proc. of Web Caching Workshop (WCW'99)*, 1999.

[RSB01] P. Rodriguez, C. Spanner, and E. Biersack. Analysis of Web caching architectures: hierarchical and distributed caching. *IEEE/ACM Transactions on Networking*, Aug. 2001.

[RW98] A. Rousskov and D. Wessels. Cache Digests. *Computer Networks and ISDN Systems*, 30(22-23), Nov. 1998.

[RW00] A. Russkov and D. Wessels. Web Polygraph, 2000. – http://www.web-polygraph.org.

[SCCQ02] A. Santoro, B. Ciciani, M. Colajanni, and F. Quaglia. Two-Tier Cooperation: A Scalable Protocol for Web Cache Sharing. In *Proc. of IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, Feb. 2002.

[SER+04] Vagner Sacramento, Markus Endler, Hana K. Rubinsztejn, Luciana S. Lima, Kleder Gonalves, Fernando N. Nascimento, and Giulliano A. Bueno. MoCA: A Middleware for Developing Collaborative Applications for Mobile Users. *IEEE Distributed Systems Online*, 5(10), October 2004.

[SPE05] SPEC Web, 2005. – http://www.spec.org/web2005/.

[Squ] Squid Internet Object Cache. `http://www.squid-cache.org`.

[Sun06] The Sun Java System Web Server, 2006. – http://www.sun.com/software/products/web_srvr/.

[sur04] December 2004 Web Server Survey, 2004. http://news.netcraft.com/archives/web_server_survey.html.

[SWCK02] W. Shi, R. Wright, E. Collins, and V Karamcheti. Workload characterization of a personalized Web site and its implications for dynamic content caching. In *Proc. of the* 13^{th} *Int'l Conf. on Web content Caching and Distribution (WCW2002)*, August 2002.

[TDVK99] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: design considerations for distributed caching on the Internet. In *Proc.* 19^{th} *International Concerence on Distributed Computing Systems*. IEEE, Jun. 1999.

[tom05] The Tomcat Servlet Engine, 2005. – http://jakarta.apache.org/tomcat/.

[TPC04] TPC-W transactional web e-commerce benchmark, 2004. – http://www.tpc.org/tpcw/.

[W3C00] Techniques for Web Content Accessibility Guidelines 1.0, 2000. http://www.w3.org/TR/WCAG10-HTML-TECHS/.

[WB01] C. Williamson and M. Busari. On the Sensitivity of Web Proxy Cache Performance to Workload Characteristics. In *Proc. of IEEE Infocom*, Apr. 2001.

[WC97a] D. Wessels and K. Claffy. Application of Internet Cache Protocol (ICP), version 2, Sep. 1997. RFC 2187.

[WC97b] D. Wessels and K. Claffy. ICP and the Squid Web Cache, Aug. 1997.

[WC97c] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2, Sep. 1997. RFC 2186.

[Web] Webcleaner Filter Proxy. http://webcleaner.sourceforge.net/.

[Web06a] The BEA Weblogic Server, 2006. – http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/server/.

[Web06b] IBM Websphere Transcoding Publisher, 2006. http://www-3.ibm.com/software/webservers/transcoding.

[Wes01] Duane Wessels. *Web Caching*. O'Reilly, 2001.

[Wes02] D. Wessels. *Squid Programmers Guide*, 2002.

[WVS+99] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative Web proxy caching. In *Proc. Symposium on Operting System Principles*, 1999.

[WWW05] Web Content Accessibility Guidelines 2.0, W3C w3c working draft, 30 June 2005. http://www.w3.org/TR/WCAG20/.

[XBC02] Haiyong Xie, Laxami Bhuyan, and Yeim-Kuan Chang. Benchmarking Web Server Architectures: A Simulation Study on Micro Performance. In *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-02), with HPCA-8*, Feb 2002.

[Zeu06] The Zeus Web server, 2006. – http://www.zeus.com/.

[Zip99] G. K. Zipf. *Human behaviour and the Principle of Least Effort.* Addison Wesley, 1999.

List of Tables

2.1 Composition of the workload models. 34
2.2 Hardware resource utilizations 40

3.1 Cache size per node [% of working set] for query-based cooperation schemes 60
3.2 Cache size per node [% of working set] for CD protocol 62
3.3 Global Hit Rate of Cache Digest as a function of the frequency of client requests 63

4.1 Cache hit rates and overheads for Workload 2 83
4.2 Cache hit rates and overheads for Workload 1 85
4.3 90 percentile of response times in seconds 90

5.1 Main features of some intermediary adaptation servers 106
5.2 Average response time [ms] as a function of the page request rate 139
5.3 SISI: Average response time [ms] with RemoveBackground and GIFDeanimate services 141

List of Figures

1.1 A logical view of a modern Web-based infrastructure for the support of efficient services 2

2.1 The generation level of the Web-based infrastructure 9
2.2 A typical LAN-based system for content generation 11
2.3 Temporal diagram of a request for a Web object to a Web-based system 27
2.4 An example of PHP-based architecture of a Web-based system 29
2.5 Architecture of the testbed for the experiments 35
2.6 Throughput of the Web-based system 36
2.7 Response time of the Web-based system 37
2.8 90-percentile of the contributions to the system response time at the node level 38
2.9 CPU utilization of the back-end node (240 clients) 39
2.10 CPU utilization of the back-end node (360 clients) 40
2.11 CPU utilization percentages at the function level (database process) 41
2.12 Cumulative distribution functions of the back-end response time before and after the system tuning operation 43
2.13 Cumulative distributions of the response time T_r and the network time T_{net} (WAN scenario) 44
2.14 Cumulative distributions of the components of T_{sys} (no WAN scenario) 45

2.15 Cumulative distribution function of the components of T_{sys} (WAN scenario) 45
2.16 Number of utilized sockets by the back-end server in the WAN and no-WAN scenarios 47

3.1 The delivery level of the Web-based infrastructure 51

4.1 Example of a two-tier architecture: groups and representative subsets 66
4.2 Configuration example of a system using InterQ-IntraS cooperation 70
4.3 InterQ-IntraS cooperation. A client requests resources to a non master server 71
4.4 InterQ-IntraS cooperation. A client requests resources to a master server 71
4.5 InterS-IntraQ architecture 74
4.6 InterS-IntraQ cooperation protocol: answer to an HTTP query from a client. 75
4.7 Comparison between the InterS-IntraQ protocol and its variant 77
4.8 Sensitivity of traffic overhead for cooperation to cache capacity 87
4.9 Response time (light network load) 89
4.10 Response time (heavy network load) 90

5.1 The adaptation level of the Web-based infrastructure 98
5.2 Placement of the SISI modules into the Apache request Lifecycle. 112
5.3 The SISI Deployment: the HTML forms used by the programmer to provide information about the new service to load into Apache 122
5.4 The SISI Deployment 123
5.5 The SISI Visual monitor with the three areas shown: system, user, logs 126
5.6 Users's Profile Management 128

5.7 The LinkAccessKey and Link relationship services 130
5.8 The experimental testbed: the client requests resources through an intermediary adaptation server 136
5.9 The client requesting a resource through the SISI framework . 138
5.10 90-percentile of the page response time 138
5.11 Average response time for Muffin and SISI when removing background colors and images 139