Canali C, Colajanni M, Malandrino D *et al.* A novel intermediary framework for dynamic edge service composition. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(2): 281–297 Mar. 2012. DOI 10.1007/s11390-012-1223-2

A Novel Intermediary Framework for Dynamic Edge Service Composition

Claudia Canali[1], *Member*, *IEEE*, Michele Colajanni[1], *Member*, *IEEE*, Delfina Malandrino[2], Vittorio Scarano[2], *Member*, *ACM*, and Raffaele Spinelli[2]

[1] *Department of Information Engineering, University of Modena and Reggio Emilia, Modena, I-4125, Italy*

[2] *Department of Computer Science, University of Salerno, Fisciano (SA), I-84084, Italy*

E-mail: {claudia.canali, colajanni}@unimore.it; {delmal, vitsca, spinelli}@dia.unisa.it

Received February 25, 2011; revised December 7, 2011.

Abstract Multimedia content, user mobility and heterogeneous client devices require novel systems that are able to support ubiquitous access to the Web resources. In this scenario, solutions that combine flexibility, efficiency and scalability in offering edge services for ubiquitous access are needed. We propose an original intermediary framework, namely Scalable Intermediary Software Infrastructure (SISI), which is able to dynamically compose edge services on the basis of user preferences and device characteristics. The SISI framework exploits a per-user profiling mechanism, where each user can initially set his/her personal preferences through a simple Web interface, and the system is then able to compose at run-time the necessary components. The basic framework can be enriched through new edge services that can be easily implemented through a programming model based on APIs and internal functions. Our experiments demonstrate that flexibility and edge service composition do not affect the system performance. We show that this framework is able to chain multiple edge services and to guarantee stable performance.

Keywords ubiquitous Web, edge service, intermediary framework, service flow, performance

1 Introduction

In the ubiquitous computing epoch, one of the main challenges of Web-based system is to guarantee *anytime/anywhere* access to *any data and information* through *any client device* and access network. The advent of novel Web applications has led to a new way of interaction, has multiplied the heterogeneity of offered contents and has promoted interactive forms of information sharing among users. At the same time, wireless technologies have allowed these interactive Web applications to be accessed through a large variety of mobile devices, such as tablet computers (e.g., iPad), smart phones (e.g., iPhone, NexusOne), car-based systems. These devices are characterized by heterogeneous capabilities in terms of hardware and software properties, thus opening several challenges to the content providers for the delivery and presentation of Web resources. In this scenario, *edge services* are needed to tailor Web contents according to the user preferences and needs and to the capabilities of mobile client devices.

A modern content generation and delivery function is now composed by more edge services that should be applied to obtain the final required version of the content: we define *service flow* the sequence of applications that are able to transform the original content into the modified version delivered to the end user. Most edge services are based on user preferences that are explicitly declared by the user or inferred by the system. This means that the provider's system must support a per-user profiling and the possibility of personal customization. The framework proposed in this paper supports edge services that are able to manage per-user profiles.

Edge services may include a wide range of Web content transformations, such as *content adaptation*, *content filtering* and *privacy protection*, and many others are appearing and will appear in the future. For this reason, flexibility and scalability are mandatory requirements of the provider's system. Content adaptation services involve the transformation of textual and multimedia contents (e.g., text compression, HTML structure manipulations, image removal and/or quality reduction, text-to-speech translation, video-on-demand adaptation, video streams transcoding) to match user preferences, client device capabilities, and available network connections[1–6]. Content filtering services allow to control which Web sites users can access, thus

Regular Paper

making contents accessible only to authorized users and avoiding children to see inappropriate documents (e.g., access control, URL blocking, banners and pop-up filtering). *Privacy protection* services address the issue to protect users from privacy leakage and identity theft during Web navigation (e.g., HTTP request/response header analysis, disabling of dangerous third party script executions, blocking of identifying URLs, Web bugs and other well-known harmful activities[7]).

Edge services can be provided client-side[8], by the content server[9] or by an intermediary framework[6,10-11] after the content generation. In this paper, we consider the last scenario, where, a network intermediary component, interposed between the client device and the content provider, analyzes and transforms the requested content on-the-fly before delivering the result to the end user. A typical service flow includes the following steps: the intermediary framework intercepts the client request, it fetches the contents from the origin server, it applies the edge services that modify the original contents to match the user profile and/or expectations, and it delivers the modified content to the client.

A solution based on an intermediary framework shifts the computational load away from content provider servers, thus simplifying their design. The intermediary-based approach represents the preferable approach to develop Web applications that require transparency, real time processing, easy usage, sharing of access from several Web users, high scalability and performance[10,12].

Specifically, we designed a "close to clients" intermediary framework because this solution presents several advantages. First, an intermediary framework located close to the clients may adapt contents coming from multiple sources. Second, the impact on network traffic of services that considerably increase the size of the delivered content, such as the *Text2Speech* service described in Section 5, is limited because contents have to travel only the last mile to the client. Third, performance may be improved by exploiting caching policies of original and adapted content. It is worth to note that several caching strategies already exist[13-15] that can be easily integrated into our intermediary framework.

On the other hand, an important issue when providing third party managed services is the privacy of users, since there is a potential for privacy violation. However, an intermediary system is often viewed as a *trusted entity* if users configure their browsers to go through it[16].

Several examples of intermediary frameworks for edge services exist, such as RabbIT[17], WebCleaner[18] and Privoxy[19]. All these systems may offer multiple edge services, as we will describe in Subsection 2.2. However, none of them support an efficient and flexible dynamic composition of edge services based on user preferences and device capabilities.

The proposed Scalable Intermediary Software Infrastructure (SISI) is a flexible intermediary infrastructure for ubiquitous Web access that integrates advanced edge services dynamically composed to match user preferences and device capabilities. Unlike other intermediary frameworks providing general services, SISI associates each user to one or more profiles containing personal preferences and device capabilities. In this way, each user can have his/her requests served according to personal settings, independently of the choices of other users. The possibility of managing multiple profiles for each user allows the system to change edge services and related parameters on the basis of current user condition and/or device.

The flexibility is another important feature of SISI. The proposed framework simplifies the integration and configuration of other services and facilitates programmers to create new edge services. Several experiments demonstrate that the dynamic composition of multiple edge services in SISI is not detrimental for performance as in other intermediary frameworks.

The paper is organized as follows. Section 2 discusses the main requirements of intermediary frameworks for edge services and presents some existing solutions. Section 3 describes the SISI architecture and its main components. Section 4 outlines the SISI programmability and flexibility by showing how programmers can easily implement new edge services and integrate them into SISI. Section 5 describes some SISI edge services that are not offered by other intermediary systems. Section 6 presents a qualitative comparison between SISI and existing frameworks for ubiquitous Web access. Section 7, Section 8 and Section 9 introduce the testbed and analyze the performance and the scalability of SISI, respectively. Section 10 concludes the paper with some final remarks.

2 Motivation and Background

Providing advanced edge services for the ubiquitous Web poses several challenges to the underlying systems. In this section, we first discuss the main requirements that an intermediary framework for edge services has to satisfy, then we present some existing frameworks offering edge services for the ubiquitous Web.

2.1 Requirements of Intermediary Frameworks for Edge Services

A fundamental requirement for intermediary

frameworks offering edge services for the ubiquitous Web is the capability to customize the offered services according to the needs of every single user. Since edge services include a highly heterogeneous variety of content transformations, the intermediary framework should support a mechanism for the *dynamic composition* of multiple edge services that will constitute the services flow to be applied to the user requests. Edge services should be dynamically selected, configured and composed depending on the user preferences and device capabilities. As anticipated in Section 1, every user customization is not feasible if the underlying system only supports a system-wide profile that is used for every user request. A per-user profiling should be supported to manage personalized users preferences and device information. This implies that the framework has also to support user authentication and authorization mechanisms, to ensure that each user may perform only the authorized operations. The support of more profiles for each user offers an important enhancement for the content customization, because the edge service flow may be composed by taking into account specific user settings related to his/her current context and used device.

A second important requirement is the *programmability* of the framework, that is the possibility to extend or develop from scratch system functionalities in a quick and easy way. By leveraging a specific execution environment and a programming model, a programmable framework may offer quick prototyping and easy deployment of new customized edge services. The execution environment has to exhibit a modular structure for providing the basic components to develop new edge services, while the programming model should provide APIs and software libraries to simplify the development of new edge services and the enhancement of the existing ones. Developing edge services that can be *plugged-in* very easily into the intermediary framework allows the system to extend its pre-defined behavior without the need of major modifications of the original source code. Finally, by enhancing the separation between the core network infrastructure and the programming model for developing new edge services, programmers can develop services without taking care of the internal details of the underlying system.

A final, but not least, important requirement is related to the *efficiency* of the framework in offering advanced edge services. The high computational expensiveness of most edge services[20-21] represents a great challenge for the underlying system, that has to provide users with acceptable response time even in case of high load. Moreover, since a single user request usually requires a service flow including multiple edge services, the intermediary framework should be efficient in composing edge services and avoid performance degradation. It is important to note that, even if efficiency is an essential requirement for systems offering edge services, not much attention has been devoted to this aspect during the design phase of existing frameworks, as we will discuss in Section 6.

2.2 State-of-the Art of Intermediary Systems

We now present an overview of three frameworks offering edge services for the ubiquitous Web, RabbIT, Privoxy, and WebCleaner, with a description of their main features. It is worthy to note that the frameworks that we consider are based on Open Source software, and they are written in different programming and scripting languages, in order to provide a comprehensive comparison with the SISI proposal.

RabbIT Web Proxy. RabbIT[17] is a Java-based intermediary server that aims to speed up Web navigation by compressing text pages and images, by removing unnecessary parts of HTML pages, such as HTML tags, background images, advertisements, banners, pop-ups, by caching filtered documents before forwarding them to the clients. RabbIT supports one system-wide profile, that is, all users that connect to the Internet through this intermediary will have the same services applied to their requests, differently from SISI that allows the definition of multiple profiles according to different needs. RabbIT has a modular architecture that facilitates the integration of new modules through a set of provided APIs, but their development is not a trivial task because RabbIT does not provide any tool for modules deployment and configuration. To enable new modules, the main configuration file of RabbIT must be accessed and modified accordingly. In summary, RabbIT has mainly been conceived as a tool to compress Web resources and remove possibly annoying elements from Web pages.

Privoxy Web Proxy. Privoxy[19] is an intermediary system with advanced filtering capabilities to protect privacy, to modify Web contents, to control accesses, and to remove advertisements, banners, pop-ups. Privoxy is implemented in C language and new services can be added to the system and configured through proper configuration files, that are accessed through specific URLs. The configuration files must be edited according to the user needs because there is not a simple user interface, and a correct modification with no error requires experience. Few filters are provided with the present distribution for ad-filtering by link and by size, for ad-blocking by URL, and for GIF De-animation. Moreover, implementing novel services for Privoxy is not a trivial task, since programmers can

not rely on a modular architecture and are not provided with simple building blocks. Similarly to the other frameworks, Privoxy allows the definition of system-wide profiles, with no possibility for user differentiation. The system administrator can choose among many profiles, but only one of them may be loaded at run-time and will be used for the requests coming from every user. It is important to highlight that Privoxy is not compliant with the HTTP protocol version 1.1.

WebCleaner. WebCleaner[18] is an intermediary system implemented in C and Phyton languages. WebCleaner offers filtering capabilities (e.g., removal of advertisements, banners, Flash and Java-script code), possibilities of text and image transcoding, services of Web pages blocking and virus detection. Moreover, WebCleaner allows only one system-wide profile. Even if it is not upgraded since 2006, we decided to analyze its performance for the following reasons: 1) it provides functionalities comparable with all other systems we selected; 2) it is an Open Source software and licensed under the GPL; 3) it is implemented in Python with a parser written in C to parse and analyze Web pages (i.e., it gives us the possibility to analyze edge servers implemented in another programming language); 4) it is configurable and new services can be implemented to enhance the pre-defined behavior. However, this framework does not offer any support for a quick prototyping of new edge services and for an ease implementation of new system functionalities. WebCleaner simply employs a configuration process to customize services according to the specific context. New services can be added through a Web interface by specifying some configuration parameters.

3 Scalable Intermediary Software Infrastructure (SISI)

In this Section we first present the architecture of the SISI framework and its components, then we describe how SISI allows dynamic edge service composition to satisfy heterogeneous user needs and device constraints in a ubiquitous Web environment, as shown in Fig.1.

3.1 SISI Architecture

From the architectural point of view, SISI consists of many modules that allow to intercept client requests, apply the required edge services, and deliver the adapted response to the client. The SISI modules have been implemented in the Perl language as *Apache* modules[22]. There are multiple reasons that motivate this implementation choice. The Apache Web server is reliable and highly popular, it supports plug-in modules for extensibility and is relatively easy to configure[23]. These characteristics allow programmers to quickly and easily implement new modules in the Perl language and integrate them into Apache by taking advantage of the power and the flexibility of the standard *mod_perl* Apache module[24]. Furthermore, it

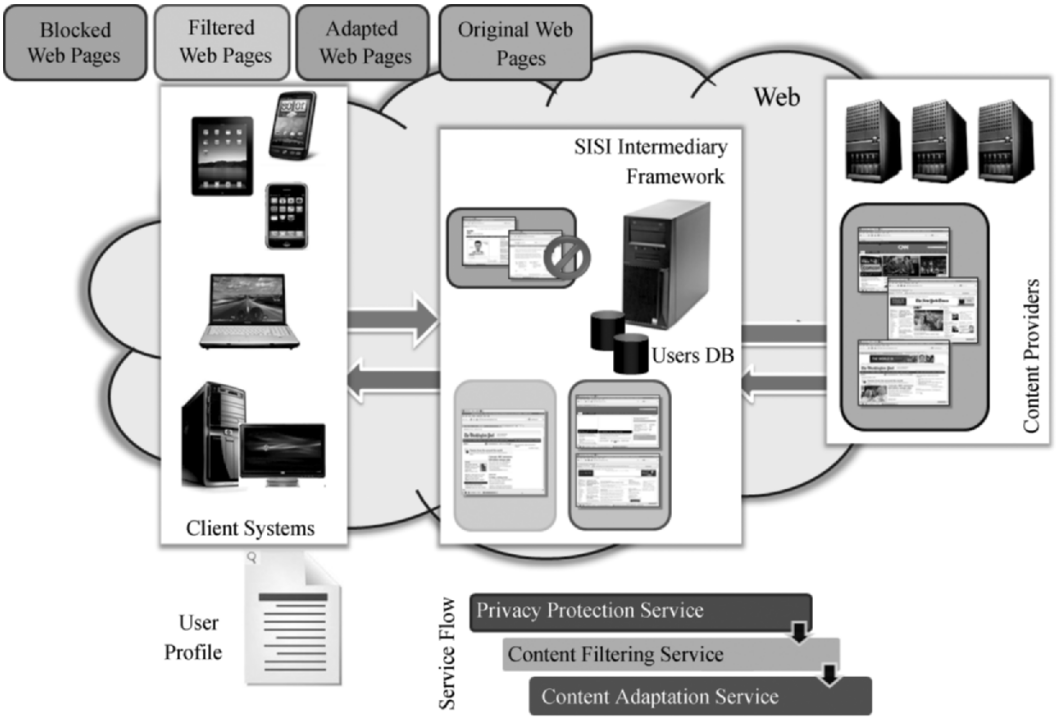

Fig.1. SISI intermediary framework.

is important to note that having a persistent Perl interpreter embedded into the Apache Web server avoids the overhead of activating an external interpreter for any HTTP request that requires to run Perl code.

Fig.2 shows the SISI internal structure, where we can distinguish three categories of modules:

- *CORE Modules*. They include the main building blocks of the SISI framework. The *Proxy module*, the *Dispatching module* and the *Authorization module* are devoted to intercept and apply some manipulations at the early stage of the request processing, while the *Deployment module* is used to enhance the SISI framework with new implemented services. These modules are presented next in this Section.
- *SERVICE Modules*. They include all details related to the edge services implementation. These modules are presented in Section 5.
- *SYSTEM Modules*. They are at the basis of the SISI programming model and are presented in Subsection 4.1.

In the Apache programming environment, each HTTP request is processed in sequential phases, and different decisions can be taken about the request at each phase. This provides the possibility to access and control all phases of the HTTP request life-cycle, thus allowing the programmer to enhance and personalize the behavior of the intermediary server. For each phase a specialized handler may be provided to manipulate the requested URI. In addition, multiple handlers can be chained together to contribute to the processing of a request.

In the following we describe the SISI CORE modules outlining their tasks and their relationships with other modules. They offer all the main internal functionalities and are activated in different phases of the Apache HTTP request life-cycle.

3.1.1 SISI CORE Modules

SISI CORE modules includes the *Proxy module*, the *Authorization module*, the *Dispatching module* and the *Deployment module*.

The Proxy module intercepts all HTTP requests and, as the first step, identifies the user or initiates a challenge-response authentication cooperating with the Authorization module (that verifies that the user issuing a request is bearing the necessary authorizations), then it loads the current user profile, fetches the original resources and forwards them to the following Apache phase. It must be emphasized that we have used this mechanism to authenticate users to both restrict access to resources and to enhance the trustiness of the SISI intermediary system. In this phase intervenes the Dispatching module that acts as a dispatcher within the SISI architecture for the available edge services, which can be customized according to the user preferences. All modules that implement the edge services are preloaded in the main memory, but only the modules corresponding to the services requested by the user are actually added to the transaction. Its main goal is, therefore, to

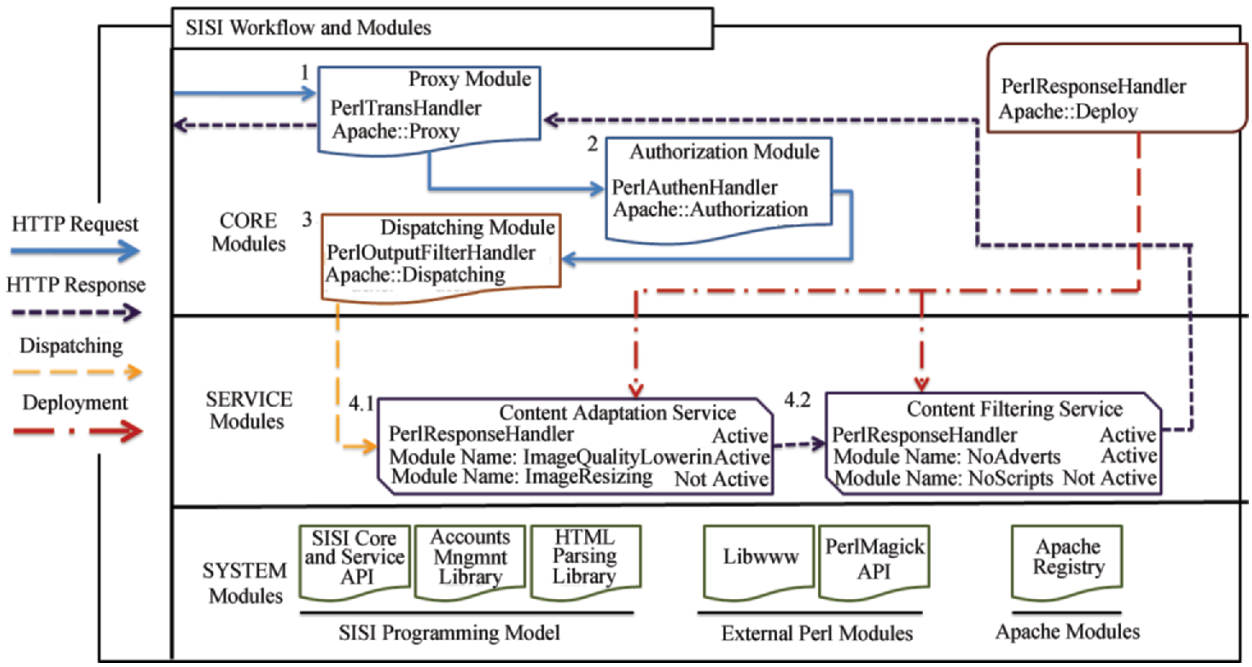

Fig.2. SISI modules and Workflow. SERVICE modules include the Content Adaptation and the Content Filtering edge services. For these services we show the modules involved in the benchmarking process described in Section 8.

compose edge services in a complex service flow according to user preferences. Finally, the Deployment module allows the system administrator to automatically add new services to the framework with no need of a detailed knowledge of the underlying software infrastructure.

3.2 SISI User Profile Management

The support for user and device profiling is an important requirement in the context of the ubiquitous Web because of the heterogeneity of the devices and offered services. Users need multiple profiles according to their status or preferences, and different configurations that should be easily modified.

As discussed in Section 2, user authentication is a fundamental requirement to allow user profile management to fully support the dynamic composition of edge services and their customization. While systems described in Subsection 2.2 do not provide profile management features lacking in differentiate the offered services depending on the user identity, SISI provides full support for both creation and management of user profiles.

Specifically, when a new user is added to the system by the administrator, a default profile is automatically generated and the user can modify it the first time he/she logs into the system. This default profile is filled with default values for each content service. A user may create a new profile, modify or delete an existing profile through simple forms. Since mobile devices are characterized by limited capabilities (e.g., display, processing power and connectivity) users connected through a wireless device may want only black/white images or resized images to save bandwidth. To this aim, they can activate the Content Adaptation edge service by enabling the the Color2Black&White or ImageResizing modules. If these users are also worried about objects that may represent a risk for their privacy, then they can activate the Privacy Protection edge service by composing it with the previous Content Adaptation edge service.

A user may also have more profiles that correspond to different devices. To this purpose, the user can simply activate the profile suitable to the device and connection that was previously stored. Through a user-friendly interface, each user may change the service parameters according to his/her personal preferences and authorizations. The services activation is manual (i.e., done by users) and the system has only to load the current profile associated with the user and apply the services in the order specified by him/her during the initial service configuration phase. A future work envisions the design of an automatic service activation, inferred by the system, with checks about consistency and correctness.

Of course the management of per-user profiles adds some overhead for the system. Specifically, the use of SISI is subject to one registration step and an authentication for each user session. This causes an initial overhead for each HTTP client request that is managed by SISI. This overhead will be quantified in the experimental results in Section 8.

4 SISI Programmability and Flexibility

In this section we describe the main strengths of the SISI framework, that is programmability, extensibility and flexibility.

4.1 SISI Programming Model

SISI allows programmers to design services starting from simple building blocks, implemented in Perl language, that can be further enhanced by means of a supported programming model. This programming model includes facilities that can be used both to implement new services and to enhance the system core functionalities. Specifically, it is composed of the following SYSTEM modules:

- *SISI Service APIs.* They allow programmers to start from templates to develop new (Apache) modules (i.e., handlers) and include APIs for managing and accessing the phases of the Apache request life-cycle.
- *SISI Core APIs.* They deal with all the operations that are carried out without programmers intervention, such as source content fetching or HTTP request/response headers manipulation.
- *SISI Accounts Mngmt Library.* This library includes functions for the system administrator to create/modify/delete user accounts and for end users to customize edge services and define profiles according to their own needs. This library also includes a specific function to select the correct (i.e., active) profile for a user among these available profiles.
- *SISI HTML Parsing Library.* This 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, images, scripts, in Web pages.

4.2 SISI Service Implementation Details

SISI programmability, as previously described, is a crucial characteristic since it allows an easy implementation of novel edge services that enhance the quality and the perception of user navigation. Easily programmable frameworks for edge services are necessary to cope with the rich and dynamic nature of the

Web[25], and to address the challenges coming from the ever growing demand for complex and differentiated services.

Often the introduction of new edge services into an existing software infrastructure is an expensive and time consuming process. To simplify this task, SISI exhibits an extensible and flexible environment where a compositional framework may provide programmers with the basic components to quickly and easily develop new services. The provided services exhibit a minimal complexity from the programming point of view and a smaller number of code lines if compared with filters provided by other systems, typically Java-based. For example, if we focus the attention on the part of the HTML page that needs to be filtered, functions like blink, background or advertisement filtering require about 50 lines of Java code in RabbIT[17], against the 20 lines of Perl code thanks to the use of regular expressions in SISI.

To implement a new edge service the programmer can choose among two possibilities: either implementing the service from scratch, by using the SISI programming model, mod_perl APIs and following the rules of the Apache module programming, or (this is where SISI programmability comes in handy) writing a simple Perl script and then using the SISI Deployment module to load it into the Apache software.

When services are added to the Apache pool of available modules (by means of the directives specified in the Apache main configuration file), and the corresponding configuration files filled out with the needed parameters, they can be invoked according to a user request.

4.3 Service Deployment Details

Through the deployment module, SISI offers an important system function that allows the system administrator to automatically add new services to the framework with no necessity of a detailed knowledge about Apache and mod_perl. It consists of an automatic module generation process that implements edge services starting from simple Perl code fragments.

If the programmers follow some easy and well defined rules (templates), Perl programs can be used to build the *core* of the service to be loaded into Apache. The Deployment module, activated through a specific internal URI, uses an XML file to define the parameters needed in the dynamic creation and installation of the new edge services. The deployment is obtained by filling out forms in an HTML page and provided information (i.e., service name, service parameters, and so on) is stored in an XML file and used by the Deployment module to build the service itself.

Automated deployment allows a quick and effective life-cycle management of the service, 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 required filtering. When ready, the Perl program can be simply deployed on the intermediary-based system.

5 Out-of-the-Box SISI Edge Services

Providing new edge services into the SISI intermediary framework is a simple task, since the programmer may take advantage of the SISI programming model and does not need to take care of the details of the underlying system. We implemented a variety of services that can be taken as examples by programmers who wants to create new ones. In particular, the implemented services encompass three different service areas:

- Content adaptation for ubiquitous and mobile computing;
- Content filtering and privacy protection;
- Web accessibility.

Before describing some of these services, we have to emphasize that SISI is the only framework that offers all these functionalities. Content adaptation (i.e., image transcoding) is also provided by RabbIT and WebCleaner (privoxy does not provide any service to manipulate Web images), services for privacy protection are provided by Privoxy, while filtering capabilities are provided by all the analyzed systems but with different complexity.

5.1 Content Adaptation

For this service area we provided services that allow client systems to access Web content regardless of device capabilities, network connectivity, location, user preferences and evolving needs. The content adaptation edge service includes modules that allow to adapt Web pages according to the capability of requesting client systems (i.e., *Image removal*, *ImageQualityLowering* for quality downgrade and *ImageResizing*).

Another interesting example for this service area is the content selection service, that address the challenge of creating and presenting Web sites in a form that is suitable for a wide variety of devices with different characteristics[26]. It is based on the W3C draft on content selection[27] that 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. Through this mechanism, some parts of the information can be selected while other not delivered, automatically adapting the original content according to particular accessibility rules.

5.2 Content Filtering and Privacy Protection

Internet Web sites deliver to the end users heterogeneous contents, that differ in size, format, message and so on. Different technologies are required to verify the appropriateness of these contents. For example, some contents may contain material that is not suitable for children, who increasingly access the Web at an early age. Content filtering mechanisms allow to get control on Web site accesses, ensuring that contents are accessible only by authorized users, and avoiding children to see inappropriate documents. For this category we have implemented the ParentalControl and NoAdverts modules. The ParentalControl provides functionalities for cookie analysis, text analysis (i.e., pornographic content, violence- or hate-oriented content), image (pornography) analysis, and so on. This service also provides functionalities to block requests for destination sites that are included in blacklists since their represent unwanted content. The NoAdverts module removes advertisements and other annoying Internet junk (pop-ups, js, etc.).

We also implemented the privacy protection edge service, that provides functionalities for disabling cookies (for all or for third-party servers), disabling or filtering out script execution, filtering all third-party objects (this technique can eliminate all object retrievals that could be used by third-party servers to aggregate information about a user's page retrieval), filtering requests with specified URLs, header filtering, filtering objects from top aggregation servers, removing invisible Web bugs.

5.3 Web Accessibility

For this service area we provided services that, taking into account the rules suggested by the W3C①, improve the accessibility of Web pages by enhancing the navigation of users with visual, motor or cognitive disabilities.

An example is the LinkRelationship module[28] that adds a toolbar containing the `LINK` attributes on the top of each HTML page. The objective is twofold: make HTML pages more accessible by screen readers and allow content developers to produce alternative contents, for example, for browsers that support "braille" rendering. Another example is the LinkAccessKey module[28] that 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. The goal is to make URLs accessible from users with motor disabilities. Another interesting application concerns the improvement of Web navigation for color blind users. To this aim, the ColorBlind module modifies background and foreground colors in Web pages and re-colors embedded images (also animated GIF images) in order to make more recognizable the red/green contrast for dichromatic users[29].

Finally, we implemented the Text2Speech module to make Web content accessible to people with visual disabilities. However, it could also help the comprehension of documents that are written in foreign languages, since a reader that is partially familiar with the spoken language and not with its written form can be supported in getting, at least, the meaning of the information contained in the document.

To implement the Text2Speech module we used eSpeak[30], a compact Open Source software speech synthesizer for English and other languages and for Linux and Windows platforms. The Text2Speech module first parses the original HTML document to eliminate useless characters, then, the resulted document is parsed to obtain a text version of the HTML document (by using an External Perl Library); finally, the resulted text document is manipulated by the eSpeak synthesizer to reproduce the corresponding spoken version. Once the synthesizer outputs the WAVE file, by using the "lame" application[31], the WAVE file is translated into a (more compact) MP3 file that is incorporated into the HTML page through the `EMBED` HTML tag and, finally, the requested document is returned to the client. When the output reaches the browser, the result will be a normal page HTML with contemporaneous reading of the content.

6 Qualitative Evaluation

In this Section we present a qualitative comparison between SISI and the existing frameworks for ubiquitous Web edge services presented in Subsection 2.2. The comparison focuses on the capability of dynamically composing the edge service flow, the programmability and the efficiency of the considered frameworks. In Table 1 we summarize the main features of each described framework.

One of the main differences between SISI and the other intermediary frameworks is that SISI allows a per-user configuration, with authentication and authorization mechanisms, while other systems just support a system-wide profile configuration. This means that in all other frameworks each user cannot have his/her personal configuration, but each request will undergo to the same edge service flow. On the other hand, SISI not only supports a per-user profiling, but also allows

① http://www.w3.org/WAI/

Table 1. Main Features of Intermediary Frameworks

Feature	RabbIT	Privoxy	WebCleaner	SISI
Programming Language	Java	C	C, Phyton	Perl
Configuration Interface	Web[a)]	Web[b)]	Web[c)]	Web[c)], GUI
Configuration	Very complex	Very complex	Complex	Simple
Installation/Management	Simple	Complex	Complex	Simple
Per-user Profile	–	–	–	√
User Authentication	√	–	–	√
Performance Evaluation[d)]	Partial	–	–	√
HTTP/1.1 Compliance	√	No pipelining	√	√
Supported Platform	Linux/Windows	Linux/Windows	Linux/Mac OS X	Linux/Windows

Note: [a)]: The Web interface only shows information about system status and managed connections.
[b)]: The Web interface allows to modify the configuration file from a text box instead from the command line. It does not offer more powerful tools.
[c)]: The Web interface allows both service activation and customization.
[d)]: This refers to any performance study before this work.

each user to have more personal profiles, then to choose the active profile depending on current conditions, such as the used device. In this way, the service flow may be different for requests coming from the same users but through different devices, allowing a customization that can better satisfy the changing user needs. The intermediary framework supports the dynamic composition of the edge service flow thanks to its modular structure that allows to select at run-time the required services and to apply them to the request.

As regards the framework programmability, the extension of existing edge services and the development of new ones are possible for all the considered frameworks. However, some frameworks (RabbIT and SISI) provide programmers with exhaustive APIs and libraries that allow easy and quick prototyping of new edge services, without the need of knowing the details of the underlying infrastructure. On the other hand, other frameworks (Privoxy and WebCleaner) are not programmer friendly and require much higher efforts to obtain similar results. It is worthy to note that all the considered frameworks offer a basic set of edge services, as shown in Table 2. In SISI we implemented additional edge services, as described in Section 5. As we can see from Table 1, all frameworks are configurable through a Web interface or a GUI; however, it is worth to note that the configuration is very easy for SISI, while it is complex for RabbIT and WebCleaner, and error-prone for Privoxy, since for the last system, the configuration process may also require to edit text configuration files, involving not trivial tasks for standard users.

It is worth to note that a common drawback of existing frameworks is the trade-off between programmability and performance, because frameworks are often programmable but not efficient, or programmer unfriendly but efficient. Moreover, often not much attention has been devoted to the performance of the existing systems in offering edge services. For example, RabbIT has been tested only to prove the compliance to HTTP/1.1; WebCleaner and Privoxy intermediary systems come with no information about their performance. Efficiency is essential for the use of an intermediary framework in a real Web context, because many advanced edge services are computationally expensive and the composition of multiple edge services may lead to significant performance degradation in serving requests. SISI is designed to support the composition of multiple edge services without affecting system performance. An important characteristic of SISI is the hot-swap of service composition, that is its capability to load and execute different edge services according to different user profiles at run-time without the need of recompiling any part of the software framework. While the majority of existing frameworks are able to support a little number of concurrent client requests, SISI may achieve an efficient edge service composition, as we will demonstrate in Section 8. We have to recall that performance may be further improved by introducing the caching functionality, that can be easily and quickly integrated in the proposed framework. However, we did not consider this functionality in our performance evaluation to carry out a fair comparison with the alternative frameworks we have analyzed.

Table 2. Basic Set of Offered Edge Services

Functionality	RabbIT	Privoxy	WebCleaner	SISI
GIF De-Animation	–	√	√	√
Cookies Analysis/Removal	–	√	–	√
Header Analysis/Removal	–	√	√	√
Content Filtering	√	√	√	√
URL Blocking	√	–	–	√
Image Transcoding	√	–	√	√

7 Performance Study

The SISI framework was extensively tested to verify that its programmability and extensibility have not been achieved by damaging the system performance. To

this aim, we evaluate the performance of SISI and other intermediary frameworks described in Subsection 2.2 under reproducible workloads, that we especially designed for evaluation purpose. Our choice has been dictated by the lack of standard benchmarks that support and stress service composition, therefore, we have decided to create a realistic workload according to the most popular studies in this field.

We realize two different workload models: 1) a real workload, for benchmarks that aim to evaluate the systems behavior under realistic traffic conditions, and 2) an intensive workload, for benchmarks that reproduce intensive traffic conditions, such as flash crowd events. Moreover, we have realized three different class of working tests, where different edge services and request rates are applied to the real and intensive workload models, in order to analyze and evaluate the performance of the analyzed intermediary frameworks when multiple services are composed and applied to the user requests.

In the rest of this section we describe the real and intensive workload models, then the testbed architecture used to evaluate the performance of the considered intermediary-based systems under a wide range of conditions. In the next section, we will describe the Working Tests and the corresponding experimental results.

7.1 Workload Models

We rely on past studies on Web workload characterization[20,32-33] to create a realistic Web benchmark that reproduces a "real life" workload. In terms of workload composition, the Real workload consists of: 60% of images (25% JPEG and 35% GIF) and 30% of HTML documents.

The real workload is modelled through the following parameters: 1) the file size distribution, modeled through a hybrid distribution where the body follows a Lognormal distribution with mean (μ) $\sim$ 7 000 and standard deviation (σ) $\sim$ 11 000, and the tail follows a Pareto distribution with (α) $\sim$ 1.3[32,34]; 2) the resource popularity, modeled through the Zipf-like distribution with parameter α equal to 0.75[32,35]; 3) the temporal locality, that refers to the time-based correlation in document access behavior and is modeled through a Least Recently Used Stack Model with Stack size equal to 1 000[32]. A summary of our workload model and related parameters are shown in Tables 3$\sim$4.

The intensive workload model aims to stress the systems in order to estimate the maximum load sustainable by each intermediary framework. Specifically, we designed two workloads, namely "Picture" and "Annoyance", with different sets of Web resources. The basic idea of the intensive workload is to put much more stress on the transcoding process in the case of the "Picture" workload (expressly made by only images) and on the filtering process for the "Annoyance" one (expressly made by only HTML pages with a lot of unwanted content).

Table 3. Real Workload Model Distributions

Category	Distribution	Formulas
Resource Size (tail)	Pareto	$\alpha k^{\alpha} x^{-\alpha-1}$
Resource Size (body)	Lognormal	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$
Resource Popularity	Zipf-like	$P(r) = kr^{-\alpha}$
Temporal Locality	LRU Stack	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$

Table 4. Real Workload Model Parameters

Parameter	Value
Mean of the Lognormal Distribution (μ)	7 000
StdDev of Lognormal Distribution (σ)	11 000
Pareto Tail Index	1.3
Zipf Parameter	0.75
LRU Stack Size	1 000

The "Picture" workload has been built based on the results of the studies on image workload characterization in [21, 36]. The workload consists entirely of images, 50% GIF and 50% JPEG images. The 80% of the GIF images are smaller than 6 KB, while the 20% are larger than 6 KB, including a 15% of animated GIF. Of the JPEG images, the 40% is larger than 6 KB. The "Annoyance" workload is composed of Web pages from English-language sites chosen across various categories from Alexa's most popular sites[37]. This set included pages from each of 13 categories: arts, business, computers, games, health, home, news, political, recreation, reference, regional, science, and shopping. We have analyzed off-line these pages in order to choose those with a high percentage of unwanted content.

Once defined the workload model, the next step is to define the performance indexes to evaluate the considered systems. We choose to analyze the *response time*, that is, the time elapsed between sending the request and obtaining the last byte of the corresponding response.

Once defined the performance indexes, we must choose some metrics to get the values that are representative for such indexes. Common statistical metrics are mean, standard deviation, X-percentile and finally cumulative distribution functions. We consider the Cumulative Distribution Function (CDF) of the response time as the main performance metric because it is more significant than average values in a system characterized by heavy-tailed distributions.

We use httperf[38] as synthetic workload generator. Httperf provides a flexible facility for generating

various HTTP workloads and collecting several metric measurements, i.e., connection time, latency time, request and reply rate, throughput, etc. We modified the httperf tool to collect information about the response time of each HTTP request issued in the request stream. To analytically synthesize our Web workload models, we also used WebTraff[39]. This tool is able to capture the salient characteristics of Web workloads, such as Zipf-like document popularity, heavy-tailed file size distribution and temporal locality.

7.2 Testbed Architecture

We set up a testbed architecture consisting of three nodes connected through a switched fast Ethernet LAN (see Fig.3). We consider this setting as the fairest way to compare the considered frameworks for providing edge services because it avoids possible non predictable WAN network effects.

The client node runs httperf[38] to submit the HTTP request streams, the second node runs the intermediary servers, and, finally, the third node runs a (Apache) Web server where we preloaded the files of all our workloads.

All nodes are equipped with a Dual Core Pentium 4 at 3.40 GHz, 1 GB Memory RAM, running Fedora Core 11 Linux with Kernel 2.6.29 (as shown in Fig.3). We should mention that SISI does not require special hardware because it may run on off-the-shelves PCs.

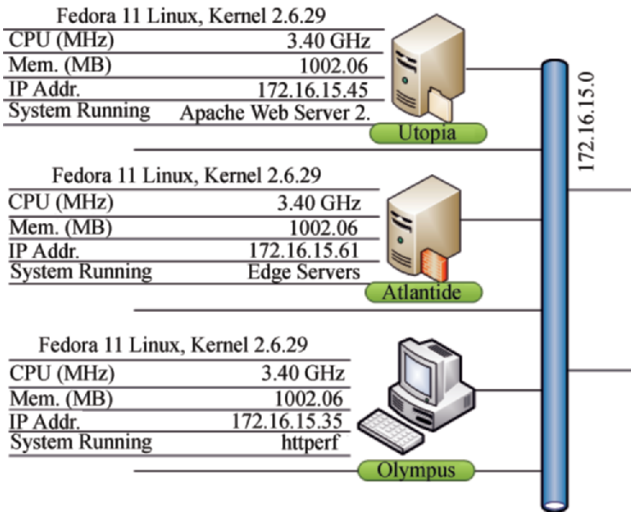

Fig.3. Experimental testbed: the client (i.e., olympus) requests resources (hosted by utopia) through an intermediary-based system (i.e., atlantide).

8 Experimental Results

In this section we describe the experiments we performed to evaluate the performance of the considered intermediary frameworks and the overhead introduced by the SISI user profile management. It is worthy to note that we do not modify any service that was already implemented in the systems, since our goal is to test the intermediary frameworks without taking any (even simple) modification.

We organize the experiments in the following categories:

- Performance Comparison Tests (PCT);
- Stress Tests (ST);
- User Profiles Tests (UPT).

To provide a deeper insight on the performance evaluation of the intermediary frameworks, we carried out a preliminary experiment to measure the response time achieved when requests are issued directly to the Web server without any intermediary intervention in the HTTP flow. Specifically, using the Real workload model we configured the client running on the olympus node to request Web pages directly to the origin Web server running on the utopia node. The result of this experiment is shown in Fig.4 (i.e., mean response time of 1.4 ms).

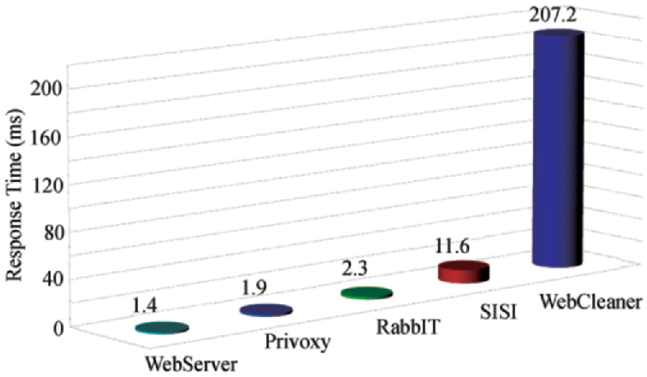

Fig.4. Mean response time (ms) for the NoService Working Test. The first column (i.e., WebServer) refers to the case when no intermediary framework is considered and HTTP requests are sent directly to the Web server.

8.1 Performance Comparison Test (PCT)

The PCT test aims to evaluate the intermediary frameworks performance in realistic traffic conditions, under the Real workload model. To compare the intermediary frameworks performance we have analyzed the services provided by each of them (see Table 2), then we have selected the services that exhibit "comparable" complexity and computational requirements.

The PCT test can be further classified in three different working tests:

- *NoService* working test;
- *OneService* working test;
- *MoreServices* working test.

NoService Working Test. We carried out this working test as preliminary experiment to measure the overhead introduced by the presence of the intermediary framework in the path between the client and the origin Web server. To this aim, we configured httperf to request Web pages through the intermediary frameworks when no service is applied on the HTTP request stream. We disabled, on each intermediary framework, all services and specific internal core functionalities (e.g., caching).

As we can see from Fig.4, Privoxy and RabbIT exhibit similar mean response time, while SISI shows a response time that is greater than one order of magnitude of the two previous intermediary frameworks. This is due to the fact that SISI is implemented as part of the Apache architecture, and therefore, it adds an overhead due to the memory usage of all Apache and mod_perl modules loaded at Web server startup. Finally, WebCleaner shows a response time that is two orders of magnitude higher than Privoxy and RabbIT response time. A possible explanation for this result is that WebCleaner uses an HTML SAX parser invoked for every HTTP request.

OneService Working Test. In the OneService Working test we have compared the performance of the intermediary frameworks when only one service is applied on the HTTP request stream. For this tests we choose the content filtering and the transcoding edge services. The functionality of the content filtering edge service is to remove all advertisements from the requested Web resources (it corresponds to the SISI NoAdverts module of the content filtering edge service shown in Fig.2). The *Transcoding* edge service aims to reduce the quality of images in order to save bandwidth (it corresponds to the SISI ImageQualityLowering module of the *Transcoding* edge service shown in Fig.2).

From Fig.5, which shows the cumulative distributions of response time of the considered systems for the content filtering edge service, we can see that RabbIT shows the best results. However, it is important to note that the provided content filtering edge services are not comparable in terms of computational load: by inspecting the Java source code of the RabbIT ad-filtering service, we found that RabbIT uses a simple regular expression to look for the string *ad*, while other systems rely on complex regular expressions to provide advanced filtering capabilities that remove ads and other unwanted Web junk. From the results, we see that also Privoxy outperforms SISI, with a 90-percentile of 9 ms against 18.8 ms. The good performance of Privoxy may be due to its structure, that is a single C module, while in other frameworks, such as SISI, an overhead is introduced because each HTTP request has to go through multiple modules to be served. However, we should consider that for a fair comparison in this test we configured httperf to use HTTP version 1.0 since Privoxy does not support the request pipelining (HTTP version 1.1). This may explain the worse performance of SISI, which does not fully exploit its capabilities. WebCleaner achieves the worst performance, however, it shows a minimal increment (i.e., 2.29%) with respect to the NoService Working test, while Privoxy shows the biggest increment (i.e., 272%) in terms of user response time.

Fig.6 shows the cumulative distributions of the response time of the considered systems for the Transcoding edge service. This experiment does not take into consideration the performance of Privoxy since it does not provide any service for image transformation. As we can observe, SISI achieves a 90-percentile of the response time lower than 20 ms, while the same percentile is achieved by RabbIT in approximately 35 ms. For this type of service SISI performs better than RabbIT, which starts to experience an increasing overhead since

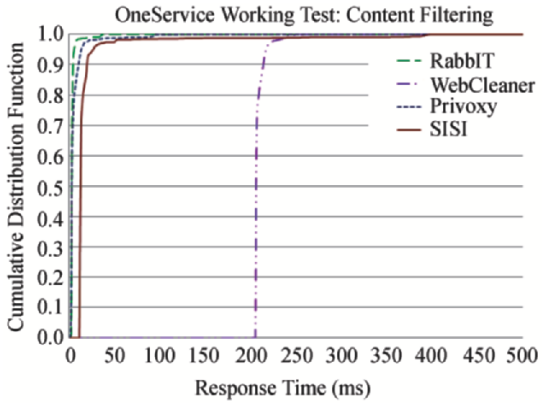

Fig.5. Cumulative distributions of response time for the OneService Working Test for the real workload model and the content filtering edge service.

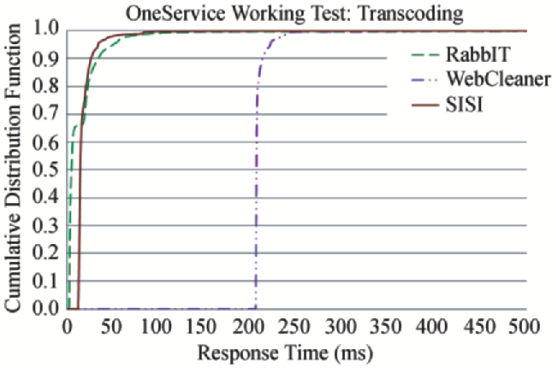

Fig.6. Cumulative distributions of response time for the Transcoding edge service.

the complexity (in terms of computational load) of the service tested. The step in the curve of Rabbit depends on the size of the transcoded images.

MoreService Working Test. This test aims to evaluate the intermediary frameworks performance when a chain of edge services is applied to the HTTP request stream. For this working test we use the Content filtering and Transcoding edge services considered for the OneService Working Test, which in this case are applied one after the other to satisfy the client request.

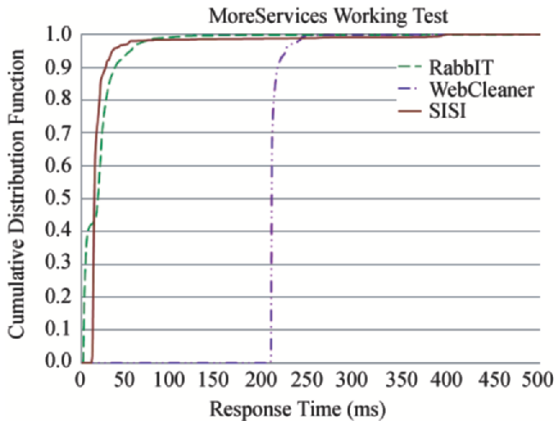

Fig.7. Cumulative distributions of response time for the MoreServices Working Test for the real workload model.

As we can see from Fig.7, WebCleaner shows once again the worst results even if with a minimal increase compared with previous test. On the contrary, RabbIT shows a significant performance degradation, while SISI shows a minimum increase compared to the previous test. SISI appears to be efficient in chaining edge services, because it adds a small overhead for additional services.

From Table 5, which shows the 90-percentile for both OneService and MoreServices Working Tests, we can see that RabbIT shows an overhead that is higher than that experienced by SISI and WebCleaner. WebCleaner response time remains high, even if the overhead obtained by increasing the number of services is minimal. The same applies for SISI, which in comparison with RabbIT achieves better results.

Table 5. 90-Percentile for the OneService [Content Filtering], OneService [Transcoding] and MoreServices Working Tests

Framework	90-percentile (ms)		
	OneService		MoreServices
	Content Filtering	Transcoding	
Privoxy	9.04	–	–
RabbIT	2.97	32.22	35.72
SISI	18.87	24.84	25.27
WebCleaner	212.11	211.05	213.11

8.2 Stress Test (ST)

With this class of tests we want to evaluate the performance of intermediary frameworks under intensive traffic conditions. For this category of tests we used the Intensive workloads defined in Subsection 7.1 expressly created to stress the edge services. Therefore, the "Picture" workload will be used to stress the Transcoding edge service and the "Annoyance" workload to stress the content filtering edge service.

As for the PCT category also for the ST category we organized tests in two different Working Tests:

- LowQualityPictures working Test;
- NoAnnoyance Working Test.

LowQualityPictures Working Test. In this working test we have compared the performance of SISI and RabbIT by analyzing the response time when the Transcoding edge service is applied on the request stream. Since Privoxy does not provide any transcoding services, we do not consider it in this test.

As we can see from Table 6 (i.e., Intensive Workload, column on the right), SISI achieves better results with respect to RabbIT. The same applies when we carried out the same experiment by using the Real Workload model (Table 6, column on the left). In both experiments SISI outperforms RabbIT, but the most interesting result is that SISI shows the lowest increase in terms of response time when passing from a realistic workload to stressful conditions. Finally, for this Working Test, SISI was able to sustain a load of 77.2 req/s (request per second) against 44.7 req/s of RabbIT.

Table 6. 90-Percentile for SISI and RabbIT When the Transcoding Edge Service is Applied to Both the Real and the Intensive Workload (i.e., "Picture" Workload Model)

Intermediary Framework	90-Percentile (ms)	
	Real Workload	Intensive Workload
SISI	24.84	144.02
RabbIT	32.22	769.44

NoAnnoyance Working Test. In this working test we compared the performance of SISI and Privoxy. The tested edge service is the content filtering edge service.

As we can see from the results in Table 7, in this test SISI outperforms Privoxy, that for the first time shows worse performance. It is important to highlight that Privoxy starts to get worse results when the applied service exhibits a greater complexity. We argue that when the cost of the filtering process increases, Privoxy is not able to sustain too many concurrent requests. To show that we compared the results of this test with the results of the experiment (described in Subsection 8.1 and outlined in Table 7, Real Workload column) carried out by applying the same service but by using the

real workload model (that does not put more stress on the filtering process since the different nature, in terms of type of Web resources, of the working set). We observe that the experienced overhead is high for both the intermediary frameworks. However, the most interesting result is that under the Intensive workload Privoxy shows a 90-percentile of the response time that is higher of 50% with respect to SISI, while under the real workload the 90-percentile of Privoxy was 66% lower than that of SISI. This means that the Privoxy framework is overwhelmed under an intensive traffic. Finally, for this working test, SISI was able to sustain a load of 92.6 req/s against 60 req/s of Privoxy.

Table 7. 90-Percentile for SISI and Privoxy When the Content Filtering Edge Service is Applied to Both the Real and the Intensive Workloads

Intermediary Framework	90-Percentile (ms)	
	Real Workload	Intensive Workload
SISI	15.52	7 238.90
Privoxy	5.43	10 807.94

8.3 User Profiles Tests

An important difference between SISI and the other intermediary frameworks is that SISI performs user authentication and then has to load a specific profile for each user, instead of just considering a system-wide profile. It is, then, important to measure the overhead introduced by the SISI profile management.

We compare the results of the OneService Working Test described in Subsection 8.1, where authentication was disabled, with the results obtained by applying the same service with enabled authentication. The end result of this experiment is that the overhead introduced by the authentication process is negligible, since it corresponds to a small increment of the 8% on the median response time. This result emphasizes that the overhead due to the management of user profiles is not critical, in terms of performance, for intermediary frameworks that provide advanced edge services.

8.4 Final Considerations

In this subsection we summarize the results obtained in all experiments providing some considerations about the analyzed intermediary frameworks.

WebCleaner does not provide any support for quick and easy development of new edge services. Moreover, this framework shows the worst results compared to all other intermediary framework for all the performed experiments. This is basically due to the high overhead introduced by the system even when no services are required and the requests just flow through the intermediary framework without any intervention.

The main advantages of Privoxy are robustness and reliability, while the main drawbacks are that its architecture is neither modular nor programmable. Implementing new edge services is not a trivial task, as well as the configuration of the intermediary framework itself. In terms of performance, Privoxy shows very good performance under realistic traffic conditions thanks to its structure. However, its performance significantly degrades under intensive traffic conditions. Furthermore, Privoxy is not fully compatible with HTTP 1.1 specifications.

RabbIT has a modular and programmable architecture, new services can be added and configured through configuration files. This framework achieves good performance only when tested with a Content filtering edge service under realistic traffic conditions; however, this result is due to the specific implementation of the Content filtering service, that has a very low computational cost. On the other hand, the RabbIT performance considerably degrades when a chain of multiple services is applied on HTTP request streams as well as under intensive traffic conditions.

The proposed SISI framework is easy to install, to configure and to extend with new edge services thanks to its programmability and flexibility. It is the only framework being able to provide complex functionalities other than filtering, compression and image manipulation as provided by others. It is also able to provide mechanisms to record security related events (logging) by producing an audit trail that makes possible the reconstruction and examination of a sequence of events. SISI shows the best performance in case of intensive traffic conditions with respect to all other frameworks and is very efficient in chaining multiple edge services. Finally, SISI is the only framework that allows each user to define one or more personal profiles in according to the capabilities of heterogeneous client devices and user preferences/needs. We proved that the SISI user profile management is not critical in terms of performance when edge services are applied on HTTP transactions, both under realistic and intensive traffic conditions.

9 Scalability Study

In this Section we describe the scalability experiments we have performed to show that increasing the number of offered services does not impact negatively on user experience and that services composition does not affect the overall system performance.

Before describing in detail our scalability test, we have to present two important considerations. First, since we are offering edge services rather than Web services we cannot test millions of them. Second, edge services, as described in Section 1, involve (text and image)

content transformations and for that reason, their invocations is not always idempotent. In particular the latter consideration was fundamental in the design of our scalability experiments.

To accomplish all experiments we have used the same setting used for the performance study. Specifically, we have run httperf (on the olympus client node) sending a request for a specific resource (hosted by the utopia server node) to the SISI framework (on the atlantide intermediary node) by applying i service invocations, where i varies between 1 and 1 024 (choice needed, as anticipated before, since we are testing idempotent services). The size of the requested resource is nearly 400 KB (i.e., a heavy resource) while the applied service is a text-based service, that is the LinkAccessKey (i.e., LAK) described in Section 5.

In general, the overall response time experienced by end users is the sum of three different time values: a constant time due to the network latency (between the client and the intermediary in the request phase and between the server and the intermediary in the response phase), the overhead of the infrastructure (time required to load all Apache and SISI modules) and, finally, the time required for service processing (strongly dependent on the service implementation).

Each service also loads specific modules (categorized as SYSTEM modules and described in Section 4) to provide specific functionalities (as an example all image-based services load the PerlMagick library to manipulate images[40], and in general, all implemented services load the SISI SYSTEM modules to manage user profiling and HTML parsing).

Our goal is to calculate the overhead achieved when comparing i composed instances of the same service against a single instance. We have calculated and compared two values: 1) the time required to process a single instance of the *LAK* service that we denote with $T_S(1)$ and 2) the time required to apply i instances of the *LAK* service, denoted by $T_S(i)$, and applied iteratively on SISI. These values correspond to the infrastructure and service time processing only, since we calculated and isolated the network latency time. We plot the scaled performance index (SPI) defined by:

$$SPI(i) = \frac{T_S(i)}{i * T_S(1)}, \quad \forall i \in [1..1\,024]. \tag{1}$$

As shown in Fig.8, the time of iteratively executing i instances of the same service within SISI is very close to the time needed to execute i copies of the service separately (once eliminated the network overhead). The small advantage in terms of performances of $T_S(i)$ can be explained: each service loads modules needed for user profiling, image or text manipulations, HTML parsing and so on. Obviously, for service composition experiments, this modules loading is performed only one time and shared for all the different instances of the applied service, involving, therefore, better response time.

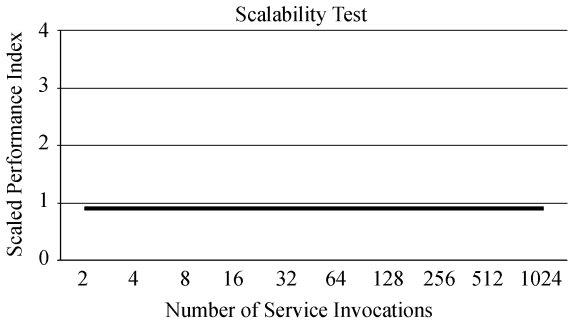

Fig.8. Scaled performance index for 1 024 services invocations.

In summary, we can conclude that the SISI framework is able to provide and compose multiple edge services and to guarantee stable performance and scalability, without affecting users experience during their navigation on the Web.

10 Conclusions

In this paper we present a new framework, namely Scalable Intermediary Software Infrastructure (SISI), that aims at facilitating the deployment of advanced edge services and their dynamic composition to meet users preferences and needs.

It is able to support dynamic edge service composition by creating at run-time complex applications starting from a set of available edge services; service configuration is possible through a user-friendly interface.

SISI differs from existing intermediary frameworks because it is able to offer per-user profiles and user authentication. Users can define one or more personal profiles, in such a way the requests of each user may involve the application of specific edge services, that can be dynamically composed at run-time. A future work will focus on how to make automatic service activation, allowing services to be automatically invoked according to users' needs, preferences and also contexts. SISI also fills the gap between programmability and efficiency that are pursued separately by existing frameworks. SISI leverages on a modular architecture that allows an easy definition of new functionalities for a wide range of application fields.

An important direction is about the distribution of SISI on several machines to achieve better performance. The idea is to implement a dispatching component, inside SISI, which invokes remote services, implemented by exploiting the virtual host mechanism of Apache.

The SISI framework was extensively tested to verify that the capability of dynamically composing edge services and the framework flexibility do not affect the performance and the scalability of the offered services. Our experiments demonstrate that SISI is able to efficiently compose edge services to allow complex functionalities for a wide range of application fields, and therefore, it represents a viable and efficient solution to deploy advanced edge services for the ubiquitous Web.

Finally, SISI is an open source project, it can be downloaded at the address http://isis.dia.unisa.it/projects/SISI/Downloads/SISI_EdgeServer.zip, and will be placed on Sourceforge at the address http://sourceforge.net/projects/sisi-edgeserver/ for future community-driven development.

Acknowledgment We would like to thank Anna Pizzuti for her collaboration during the experimental process.

References

[1] Bellavista P, Corradi A, Stefanelli C. Application-level QoS control for video-on-demand. *IEEE Internet Computing*, 2003, 7(6): 16-24.

[2] Colajanni M, Lancellotti R, Yu P S. Web Content Delivery, Tang X, Xu J, Chanson S (eds.), Springer USA, 2005, pp.285-304.

[3] El-Khatib K, Bochmann G V, El Saddik A. A QoS-based framework for distributed content adaptation. In *Proc. the 1st International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks*, Washington, DC, USA, Oct. 2004, pp.308-312.

[4] He J, Gao T, Hao W *et al.* A flexible content adaptation system using a rule-based approach. *IEEE Transactions on Knowledge and Data Engineering*, 2007, 19(1): 127-140.

[5] Jang M, Kim J H, Sohn J C. Web content adaptation and transcoding based on CC/PP and semantic templates. In *Proc. the 12th International World Wide Web Conference WWW* (*Posters*), Budapest, Hungary, May 20-24, 2003.

[6] Wijnants M, Monsieurs P, Quax P, Lamotte W. Exploiting proxy-based transcoding to increase the user quality of experience in networked applications. In *Proc. the 1st International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications*, Orlando, FL, USA, June 15, 2005, pp.73-80.

[7] Krishnamurthy B, Malandrino D, Wills C E. Measuring privacy loss and the impact of privacy protection in web browsing. In *Proc. the 3rd Symposium on Usable Privacy and Security* (*SOUPS 2007*), Pittsburgh, PA, USA, July 18-20, 2007, pp.52-63.

[8] Adblock plus. http://adblockplus.org/.

[9] Hoskins J. Exploring IBM accelerators for websphere portal, IBM White Paper, 2009.

[10] Canali C, Colajanni M, Lancellotti R. A Two-level distributed architecture for the support of content adaptation and delivery services. *Cluster Computing*, 2010, 13(1): 1-17.

[11] Hsiao J L, Hung H P, Chen H S. Versatile Transcoding proxy for Internet content adaptation. *IEEE Transactions on Multimedia*, 2008, 10(4): 646-658.

[12] Saddik A E. Performance measurements of Web services-based applications. *IEEE Transactions on Instrumentation and Measurement*, 2006, 55(5): 1599-1605.

[13] Küngas P, Dumas M. Configurable SOAP proxy cache for data provisioning web services. In *Proc. the 2011 ACM Symposium on Applied Computing* (*SAC 2011*), Taiwan, China, May 21-24, 2011, pp.1614-1621.

[14] Waleed A, Shamsuddin M S, Ismail A S. A survey of Web caching and prefetching. *International Journal of Advances in Soft Computing and Its Applications*, 2011, 3(1): 19-24.

[15] Kumar C, Norris J B. A new approach for a proxy-level Web caching mechanism. *Decision Support Systems*, 2008, 46(1): 52-60.

[16] Krishnamurthy B, Rexford J. Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement. Addison Wesley, 2001.

[17] RabbIT proxy. http://khelekore.org/rabbit/.

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

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

[20] Canali C, Colajanni M, Lancellotti R. Performance impact of future mobile-Web based services on the server infrastructure. *IEEE Internet Computing*, 2009, 13(2): 60-68.

[21] Chandra S. Content adaptation and transcoding. In *Practical Handbook of Internet Computing*, Singh M P (eds.), Chapman Hall & CRC Press, 2004.

[22] The Apache Software Foundation. http://www.apache.org.

[23] Web server survey. http://news.netcraft.com/archives/web_server_survey.html.

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

[25] Malandrino D, Scarano V. Tackling Web dynamics by programmable proxies. *Computer Networks*, 2006, 50(10): 1564-1580.

[26] Grieco R, Malandrino D, Mazzoni F, Scarano V. Mobile Web services via programmable proxies. In *Proc. the IFIP TC8 Working Conference on Mobile Information Systems* (*MOBIS 2005*), Leeds, UK, December 2005, pp.139-146.

[27] W3C working draft: Content selection for device independence (DISelect) 1.0. http://www.w3.org/TR/cselection/.

[28] Erra U, Iaccarino G, Malandrino D, Scarano V. Personalizable edge services for Web accessibility. *Universal Access in the Information Society*, 2007, 6(3): 285-306.

[29] Iaccarino G, Malandrino D, Percio M D, Scarano V. Efficient edge-services for colorblind users. In *Proc. the 15th International Conference on World Wide Web* (*WWW 2006 Posters*), Edinburgh, Scotland, May 23-25, 2006, pp.919-920.

[30] eSpeak text to speech. http://espeak.sourceforge.net/.

[31] The Lame Project. http://lame.sourceforge.net/.

[32] Williams C W A, Arlitt M, Barker K. In *Web Content Delivery*, Tang X, Xu J, Chanson S (eds.), Springer USA, 2005, pp.3-21.

[33] Faber A M, Gupta M, Viecco C H. Revisiting Web server workload invariants in the context of scientific Web sites. In *Proc. the 2006 ACM/IEEE Conference on Supercomputing*, Tampa, Florida, USA, Nov. 2006, Article 110.

[34] Bent L, Rabinovich M, Voelker G M, Xiao Z. Characterization of a large Web site population with implications for content delivery. In *Proc. the 13th International Conference on World Wide Web*, New York, NY, USA, May 17-20, 2004, pp.522-533.

[35] Yamakami T. A zipf-like distribution of popularity and hits in the mobile web pages with short life time. In *Proc. the 7th International Conference on Parallel and Distributed Computing, Applications and Technologies* (*PDCAT 2006*), Taiwan, China, December 2006, pp.240-243.

[36] Hu J, Bagga A. Categorizing images in Web documents. *IEEE Multimedia*, 2004, 11(1): 22-30.

[37] Alexa. http://www.alexa.com/.

[38] Mosberger D, Jin T. httperf: A tool for measuring Web server performance. *Performance Evaluation Review,* 1998, 26(3): 31-37.

[39] Markatchev N, Williamson C. WebTraff: A GUI for Web proxy cache workload modeling and analysis. In *Proc. the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, Washington, DC, USA, 2002, pp.356-363.

[40] PerlMagick API 6.70. http://www.imagemagick.org/script/perl-magick.php.

Claudia Canali is a researcher at the University of Modena and Reggio Emilia since 2008. She got the degree with highest honor in computer engineering from the same university in 2002, and the Ph.D. degree in computer engineering from the University of Parma in March 2006. During the Ph.D. she spent eight months as visiting researcher at the AT&T Research Labs in Florham Park, New Jersey. Her research interests include content adaptation and delivery, distributed architectures for Internet-based services, and wireless systems for mobile Web access. Home page: http://weblab.ing.unimo.it/people/canali.

Michele Colajanni is a full professor in computer engineering at the University of Modena and Reggio Emilia since 2000. He received the Master's degree in computer science from the University of Pisa, and the Ph.D. degree in computer engineering from the University of Roma in 1992. He manages the Interdepartment Research Center on Security and Safety (CRIS), and he is the director of the postgraduate master course in information security: Technology and Law. His research interests include performance and prediction models, information security, management of large scale systems. Home page: http://weblab.ing.unimo.it/people/colajanni.

Delfina Malandrino received the degree with highest honor in computer science from the University of Salerno (Italy) in 2000 and the Ph.D. degree in computer science from the University of Salerno in 2004. From October to December 2006 she visited the AT&T Research Labs in Florham Park, New Jersey, USA, working with Balachander Krishnamurthy in the field of online privacy protection. From November 2007 she is an assistant professor at the Department of Information of the University of Salerno. Her research activities mainly focus on the following research areas: distributed systems, adaptive and collaborative systems, information visualization systems, social networking, Internet traffic measurement and benchmarking, privacy protection. Home page: http://www.dia.unisa.it/professori/delmal.

Vittorio Scarano received the degree in computer science from the University of Salerno (Italy) in 1990 and he received the Ph.D. degree in applied mathematics and computer science from the University of Naples (Italy) in 1995. Since 2001 he is an associate professor at the University of Salerno. His research focuses on multimedia and distributed systems on the World Wide Web, covering several aspects from a theoretical perspective (P2P systems and architectures) to applications (intermediaries, cooperative systems and multimedia). Recently, his research interests are also devoted to information visualization and interactive virtual environments. Home page: http://www.dia.unisa.it/professori/vitsca.

Raffaele Spinelli received the degree with highest honor in computer science from the University of Salerno (Italy) in 2010. He is currently a second year Ph.D. candidate in computer science at the University of Salerno. His research focuses on Internet traffic measurement and benchmarking.