SEE: A Service Execution Environment for Edge Services

Vikrant Mastoli, Valmik Desai, and Weisong Shi

Mobile and Internet System Group
Department of Computer Science
Wayne State University
{*vikrant,valmik,weisong*}*@wayne.edu*

Abstract

The increasing mismatch between the low-bandwidth, resource characteristics of wireless mobile devices and the high-bandwidth expectations of many content-rich services drives the demand for deploying content-oriented services along the data path between the end users and the content servers. In this paper, we argue that the idea of extending existing caching proxies to support these services is promising. This suggests extending the proxy caches for more than just their original intended purpose, that is the creation of an execution environment within them, which allows the execution of services locally and remotely.

In this paper we describe the design, implementation and evaluation of a Service Execution Environment (SEE) in the context of the CONCA proxy cache. We also compare the performance of Simple Object Access Protocol (SOAP) and Internet Content Adaptation Protocol (ICAP) by using them as call-out protocols between SEE and the service providers.

1 Introduction

The role of the Internet has undergone a transition from simply being a data repository to one providing access to a plethora of sophisticated content-oriented network-accessible services. These services could include access control to block inappropriate sites, virus scanning, anonymization services to protect privacy, language translation, addition of region-specific information, image resizing and image filtering to reduce the quality of images and thereby shorten download time. Additionally, these services are increasingly being accessed by mobile consumers using end devices such as PDAs, Pocket/Handheld PCs, cellular phones and two-way pagers. The combination of these two trends holds out the possibility of providing a user with seamless, ubiquitous access to a service irrespective of the user's end device and location.

Although several infrastructures have been proposed [5, 6, 7, 15] to do this, there has not been a widespread use of them due to the concerns about their deployability, performance, and scalability. Central to each of these concerns are the questions such as where to deploy these services, how to integrate them within the existing data flows, and who would select the services. Most of the previous works neglect these questions and focus on the infrastructure supporting these services. Our argument that the idea of extending existing caching proxies to support these services is promising comes from the fact that this not only allows the end-user to enable services that allow personalization, guarantee privacy and security for all communication but also at the same time, it presents a business opportunity for ISP's and content provider's to provide these value-added services to their clients. Also in the infrastructure the service selection should be separated from the execution of services, and most importantly, it should be able to use different protocols to execute the services remotely. Based on these arguments, we designed, implemented, and evaluated a Java-based service execution environment to support service execution both locally and remotely. The major contributions of this paper include:

- A novel design of a Service Execution Environment (SEE), having three unique features: (1) a secure web interface for service registration, (2) separation of service selection and rule generation, and (3) support for multiple call out protocols.

- Comparison of the call-out protocols: Internet Content Adaptation Protocol (ICAP) [8] and Simple Object Access Protocol (SOAP) [13] from the perspective of their performance, codeability, and scalability.

- To the best of our knowledge, our work is the first public experimental platform which supports multiple protocol bindings.

The rest of the paper is structured as follows: Section 2 explains CONCA cache. Section 3 provides the objectives

of SEE and a description of the design of SEE. Section 4 describes the implementation of SEE. Section 5 presents the results of the performance evaluation. Related Work and concluding remarks are listed in Section 6 and Section 7 respectively.

2 CONCA Proxy Cache

CONCA is a proposed edge architecture for the efficient caching and delivery of dynamic and personalized content to users who access this content by using diverse devices and connection technologies [10]. CONCA attempts to exploit reuse at the granularity of individual objects making up a document, improving user experience by combining caching, prefetching, and transcoding operations as appropriate.

To achieve its goals, CONCA relies on additional information from both servers and users. All content supplied by servers in CONCA architecture is assumed to be associated with a "document template" which can be expressed by formatting languages such as XSL-FO [14] or Edge Side Includes (ESI) [12]. Given this information, CONCA node can efficiently cache dynamic and personalized content by storing quasi-static document templates and using the sharable objects among multiple users. Moreover, based on the preference information provided by users, a CONCA cache node delivers the same content to different users in a variety of formats using transcoding and reformating. The work on SEE is a part of the ongoing work on CONCA project.

3 Service Execution Environment

3.1 Objectives

Our service execution environment is designed with the following features in mind: secure, scalable, high-performance and ease-of-use. We demonstrate these requirements by describing how existing architectures [3, 11] do not prove to be as efficient as intended. First, Open Pluggable Edge Service (OPES) [11] framework requires both content providers and content consumers to use IRML [4] to specify the rules for service execution on some content. Although it provides a standard interface, it is unrealistic to ask content providers or end users to write such sort of complex rules to employ some personalized services. Furthermore, because of the prevalence of web services, it is impossible to ask service providers to support only one protocol.

Our approach to address the above problems depends on the following four components: (1) a secured web interface for service registration, (2) a simple Web interface that allows the authorized parties to select the services they desire as well as choose the providers for the selected services, (3) supporting multiple protocol bindings to make the execution environment more flexible, (4) providing a feedback-based content integrity mechanism to allow both clients and content providers to check the correctness of content after applying the content-oriented services. In this paper we discuss the former three points. Figure 1 shows the logical design of the service execution environment.

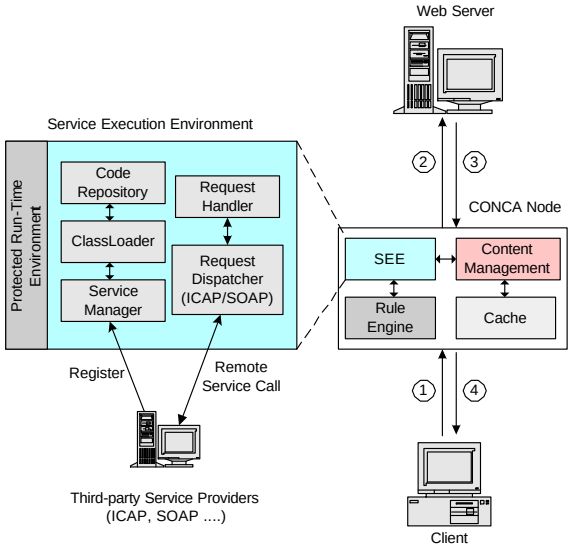

Figure 1. The architecture of service execution environment.

3.2 User Interface

The most important feature of our design is its simplicity. We create a web login for user's that would authenticate them and then present a webpage outlining the current services available and the names of all service providers. We ensure that this webpage accurately reflects the most recent list of active services by creating it dynamically when the user logs in. The user can specify the rules for a service on the web interface by using regular expressions.

3.3 Service Manager

To handle different services with differing parameters it is important that the task of managing services be assigned to a dedicated module. This is done by the *Service Manager* shown in Figure 1. It is used to handle the details of all the services that are currently deployed. It ensures that the back end always accurately reflects the services that are currently active. The *ServiceRegistry* handles the registration of services by the service provider through Java RMI. The entire service registration is done securely using SSL.

4 Implementation

Our prototype implementation of SEE is based on the architecture in Figure 1. We have deployed three services on the proposed architecture and used a basic proxy server that has no caching capability. We support two protocols that serve as call-out protocols, ICAP and SOAP. The proxy server and the architectural modules have been implemented in the Java programming language. A detail explanation of the implementation of the modules in SEE can be found in

Image Resizer	Provider_A,`www.a.com`, ICAP, Remote
	Provider_B, `localhost:8080`, SOAP, Local, ImageResizer
	Provider_C, `www.c.net`, ICAP, Remote
Virus Scanning	Provider_A, `www.a.com`, ICAP, Remote, arguments(optional)
	Provider_C, `www.c.net`, ICAP, Remote
	Provider_D, `www.d.com`, SOAP, Remote, checkandremove()

Table 1. An example structure of service management in SEE.

the technical report version of this paper [9].

We implemented the ICAP and SOAP clients in Java and placed them in the *Request Handler* module of SEE. The ICAP client creates ICAP requests for one of the two modes: REQMOD (Request Modication) and RESPMOD (Response Modification) and sends it to the ICAP server for processing. We implemented an ICAP server in Java for each service. The SOAP client has been implemented by using the Apache AXIS Java API [1]. We used Apache Axis implementation of SOAP as our SOAP server and placed the Java files of the services into the Axis webapp directory, giving them the *.jws* extension.

5 Performance Evaluation

5.1 Environment Setup

The performance evaluation was done by setting up an emulated environment consisting of three machines, which includes a web server, on a Ultra-Sparc2 200MHz with 512MB memory; The SEE, on a Pentium-4 (2.2GHz) desktop with 512MB memory; The service providers, on a different Pentium-4 (2.2GHz) desktop with 512MB memory. All these machines are in a local area network connected with a 100Mbps switched Ethernet. This is to avoid the effects of the external traffic and network congestion. Also to avoid the effects like local caching of a Web browser and the Internet traffic, all requests to SEE were generated by a client program. The client program creates requests for a fixed web page, present in the web server and consisting of one html file (30KBytes), and five images (34KBytes each). Each value is the mean of 20 runs.

Figure 2 lists the detailed timeline of a request and reply between a client, SEE, origin web server and service provider. Based on this figure, the overhead that we are interested in are defined as follows:

$T(\text{Response}) = T_{16} - T_1$
$T(\text{Rule Engine}) = (T_{15} - T_{14}) + (T_5 - T_4) + (T_3 - T_2)$
$T(\text{Origin Server}) = T_4 - T_3$
$T(\text{ICAP/SOAP Client}) = (T_7 - T_6) + (T_{13} - T_{12})$
$T(\text{ICAP/SOAP Server}) = (T_9 - T_8) + (T_{11} - T_{10})$
$T(\text{Service}) = T_{10} - T_9$
$T(\text{Network}) = (T_8 - T_7) + (T_{12} - T_{11})$

5.2 Three Services

We used three services, *Image Resizer (IR)*, *Language Translation (LT)* and *Virus Scanning (VS)*. The *IR* service processes the image bytes and reduces the image size by scaling its height and width. It has been implemented in Java. It is aimed at the users having limited bandwidth and to shorten the download time for large JPEG images. The *LT* service has been implemented by a wrapper, which uses the language translation service provided by [2]. The *VS* service has been implemented using the virus scanning service available on `www.openantivirus.com`. The reason we choose these three services is the diversity of their input/output ratio. The input of *LT* is a URL but its response is almost of the same size as that of the original content, the input of *VS* is the original content but its response is either 'yes' or 'no', and the input of *IR* is the original image but its response is a scaled form of the original image. The execution times obtained for these three services on the fixed web page are IR: 345ms, VS (text): 177ms, VS (image): 133ms and LT: 7858ms.

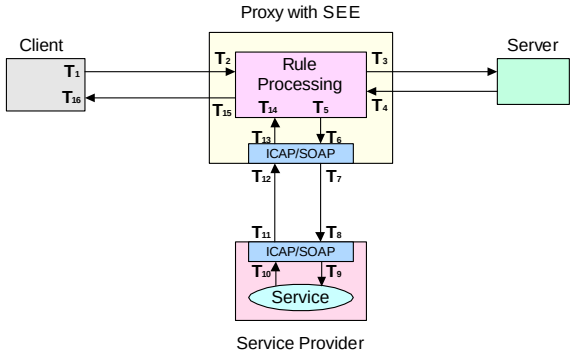

Figure 2. The detailed timeline of a request and reply.

5.3 Overhead of SEE

We first evaluated the overhead of SEE. Figure 3 compares the overhead of SEE in five different cases. The readings are independent of the call-out protocol being used to send the request to the services. In the figure, *SEE-Disabled* represents the case when SEE is not enabled, *SEE-0* represents the case when there are no services in the user's properties file, i.e., the properties file of the user is empty, *SEE-IR* represents the case when IR service is invoked on the request, *SEE-VS* represents the case when VS is invoked on the request, and *SEE-IR-VS* represents the case when IR and VS services are invoked on the request. In comparison to the

overhead of *SEE-Disabled*, the overhead of *SEE-0* is more by 136%. We ascribe this to the inherent overhead of rule processing (which includes reading a file from disk), and should be optimized in the future. From the same figure, we can see that the average processing overhead of each rule is around 30ms.

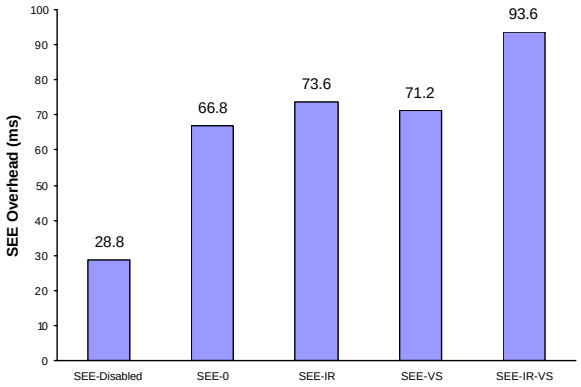

Figure 3. The execution overhead of service execution environment.

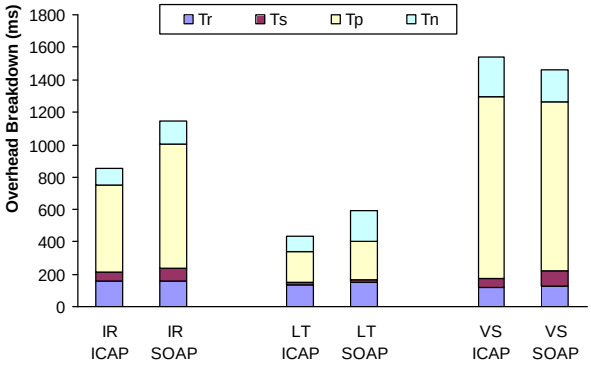

Figure 4. Breakdown of overhead of ICAP (left) and SOAP (right).

5.4 Overhead of ICAP and SOAP Protocols

The bars of each group (based on the three services) in Figure 4 show the breakdown of the overhead of processing and communication, where T_r is the processing overhead of the Rule Engine, T_s is the request and reply time between the proxy cache and origin server, T_p represents the ICAP/SOAP Client overhead, and T_n is the ICAP/SOAP server overhead. We found that the overhead of the Rule Engine is independent of the call-out protocol and the service, which reflects that the design of SEE is stable. However we found a large overhead due to the ICAP/SOAP client and also a large variation in their overhead for different services. On comparing the overheads of ICAP and SOAP clients, we found that in most cases ICAP client outperforms the SOAP client.

Table 5.3 provides the user-perceived latency when single and combination of services are invoked over ICAP and SOAP. The latency is defined as the difference between T_{16} and T_1 in Figure 2. *SEE-0* depicts the case when the properties file of the user is empty. We compared the latencies obtained for each service and combination of services with the *SEE-0* case, and made the following observations: (1) third party services may not always be good for us; (2) if the service is running far away from the data flow, the performance may become 7 times slower. Here our experiment supports the view that to obtain a good performance the service running on a remote machine should be as near as possible to the service execution environment.

After comparing the latencies obtained for ICAP and SOAP we found that there is not much difference in their performance but deploying a service over SOAP is more easier than doing it over ICAP, this is true especially in case of the legacy services. Based on the performance of ICAP and SOAP we believe that a service should be invoked remotely only if its a proprietary of someone or computing intensive.

5.5 Scalability Analysis

Our last concern of the service execution environment is its scalability. We used a multithreaded client in Java, to emulate 1, 2, 4 and 8 clients and send requests to the SEE. The results of our evaluation are shown in Figure 5. Our observations from the figure are: (1) the user-perceived latency is dominated by the service implementation, and on it being executed locally or remotely (as can be seen from the graph for LT); (2) the overhead of SEE is almost independent of the number of clients.

6 Related Work and Discussion

Our work was motivated by two related research works: Open Pluggable Edge Services (OPES) [11] and distributed content adaptation [6, 15].

In [11], IETF's Open Pluggable Edge Service working group proposes an environment to provide value-added services to the end-users, which motivates our work. Our work is an implementation of a service execution environment and the performance evaluation of ICAP and SOAP. Beck and Hofmann in [4] proposed a rule specification language for intermediary services. But in this paper we argue that the service execution environment should provide a simple web interface that allows authorized parties to select the services they desire based on the name of the services. In [3], Beck et al proposed a service execution environment prototype in which the rule engine processes all the rules for each web transaction, which is different from our implementation.

There is a large amount of prior work on content adaptation architectures, which allow the construction of *network-aware access paths* from application-specific component

Protocol	SEE-0	IR	VS (html)	VS (image)	IR+VS	LT	LT + VS
ICAP	387.7	990.2	665.6	772.2	1362.9	8180.8	8321.6
SOAP	387.8	1220.1	942.2	1046.6	1311.2	11062.2	7802.4

Table 2. Benefit of ICAP and SOAP protocol (in milliseconds).

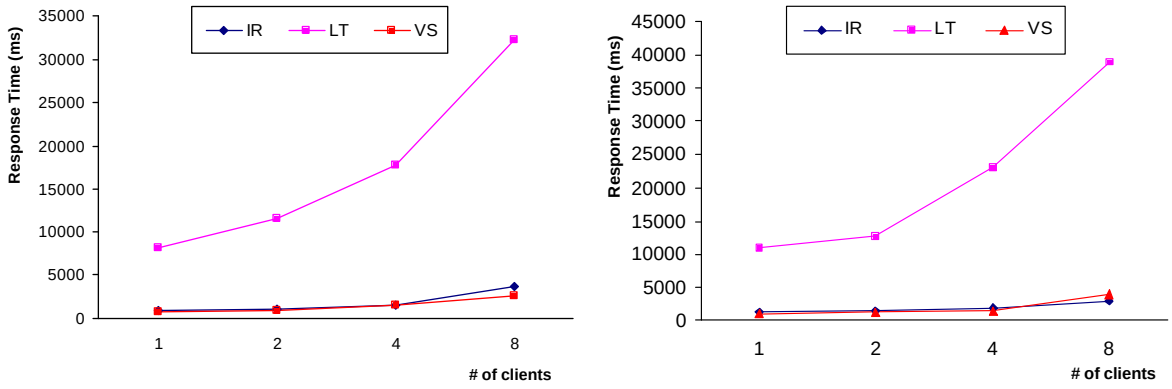

Figure 5. Scalability analysis of SEE: (a) ICAP, (b) SOAP.

and the adaptation functionalities are conducted in multiple places along the data path between content providers and clients. In this paper, we believe that computing intensive or proprietary Internet services should be executed in specific places. Therefore, our work focus on the execution environment that support different remote call-out protocols, which complements the previous work on content adaptation.

7 Conclusions and Future Work

This paper proposes a novel design and implementation of a service execution environment, which distinguishes itself from other work by its ability to support multiple protocol bindings and the ease of rule specification. Further it allows secure registration of services, as well as provides a simple web interface to allow authorized clients to configure preferences. After comparing the latencies obtained for ICAP and SOAP we find that there is not much difference in their performance but deploying a service over SOAP is more easier than doing it over ICAP, this is true especially in case of the legacy services.

Our future work includes integrating SEE into the CONCA proxy cache, optimizing the execution of multiple services within one execution environment, providing support for distributed service composition among multiple SEEs. The code of SEE will be public available soon at `http://mist.cs.wayne.edu`.

References

[1] Apache AXIS Group. Axis, `http://ws.apache.org/axis`.

[2] Babel Fish Translation, `http://www.freetranslation.com`.

[3] A. Beck and M. Hofmann. Enabling the internet to deliver content-oriented services. *Proc. of the WCW'01*, June 2001.

[4] A. Beck and M. Hofmann. IRML: A rule specification language for intermediary services, work in progress, Nov. 2001.

[5] A. Fox and et al. Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Prespectives. *IEEE Personal Communication*, Aug. 1998.

[6] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. *Proc. of the USITS'01*, pp. 135-146, Mar. 2001.

[7] S. D. Gribble and et al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Journal of Computer Networks* 35(4), Mar. 2001.

[8] ICAP Protocol Group. ICAP: the internet content adaptation protocol, work in progress, Feb. 2001.

[9] V. Mastoli, V. Desai, and W. Shi. SEE: a service execution environment for edge services. Tech. Rep. CS-MIST-TR-2003-002, Department of Computer Science, Wayne State University, Feb. 2003.

[10] W. Shi and V. Karamcheti. CONCA: An architecture for consistent nomadic content access. *Workshop on Cache, Coherence, and Consistency (WC3'01)*, June 2001.

[11] G. Tomlinson, R. Chen, and M. Hofmann. A model for open pluggable edge services, work in progress, Nov. 2001.

[12] M. Tsimelzon, B. Weihl, and L. Jacobs. ESI language sepcification 1.0, 2000, `http://www.esi.org`.

[13] W3C Consortium. Simple object access protocol (SOAP) 1.1, 2000, `http://www.w3.org/TR/SOAP/`.

[14] W3C XSL Working Group, `http://www.w3.org/Style/XSL/`.

[15] M. Yavis and et al. Conductor: Distributed Adaptation for complex Networks. *Proc. of the HotOS VII*, Mar. 1999.