Enabling the Internet to Deliver Content-Oriented Services *

André Beck, Markus Hofmann
Bell Laboratories/Lucent Technologies
101 Crawfords Corner Rd.
Holmdel, NJ 07733
{abeck,hofmann}@bell-labs.com

ABSTRACT

Increasing web traffic has led to the deployment of network intermediaries at the edges of the network. In particular caching proxies and content delivery surrogates have been very successful in accelerating web content delivery and reducing the load on origin web servers.

Today, however, users and content providers are demanding faster distribution of web content to end users. Also, many users are looking to their ISPs to provide additional content-oriented services, including filtering, security, personalization, and transformation services. At the same time, ISPs and other service providers are facing increased competition - which drives slimmer margins for basic access and data transport services.

These recent developments suggest utilizing the existing network edge infrastructure as a platform for a new class of intelligent services. These services provide tangible benefits for the end user and incremental revenue opportunity for the service providers.

This article explains a flexible and open architecture to enable network edge intermediaries to host a variety of content-oriented services. Special emphasis is put on the representation and processing of rules leading to the invocation of these services. This article also describes a research prototype implementation of a service-enabled intermediary and several example services.

Keywords

Content delivery and distribution, content-oriented services, iCAP, intelligence at the network edge, rule engine, remote callout, service execution environment, service invocation rules, web caching

1. INTRODUCTION

The Internet as of today is still mostly governed by the "end-to-end" principle [11] which demands that the network itself is to be kept as simple as possible and that all intelligence resides at the end-systems. This principle proved to be very successful and benefical for the evolution of the Internet. Despite its success, we have recently seen more application-specific functionality moving into the network, in particular to the edges of the network. Deployment of network caches and content-aware switches are probably the most widely known examples for this kind of functionality. It helps accelerating the delivery of static web pages by moving content closer to the user. However, margins for such basic delivery services are getting slimmer. Service providers have to take advantage of opportunities to provide new value-added content services for differentiation and additional revenue. Examples of such services include, but are not limited to, content filtering, content adaptation, dynamic and personalized content assembling, ad insertion and virus scanning.

While most of these services could also be installed on and provided at the client machine itself in order to adhere to the end-to-end principle, there are reasons to move them to the network edge. The typical Internet user of today, for example, can be best described as a non-technical consumer who wants to use the latest Internet technology without having to worry about technical matters like software installations or updates. Furthermore, new types of Internet access devices like PDAs and mobile phones may not always have the processing power that is necessary to run the software for providing value-added services. Other Internet appliances may only be capable of running hardcoded software so that software upgrades would not be possible at all.

Rather than developing a service-specific infrastructure from scratch, this article outlines the extension of the existing network edge infrastructure towards a flexible and open platform for a variety of new content services. It makes use of and extends existing intermediary devices, such as caching proxies and content-aware switches, enabling them to perform specific tasks on the application-layer content that is routed through them. Although this might look like a violation of the end-to-end principle, it is somewhat restored by the requirement that one or more of the parties participating in a content transaction must authorize the performed services.

The article is structured as follows: Section 2 describes the overall service platform architecture. In the following section, a rule specification language and the rule engine component of the service platform architecture are discussed in more detail. Section 4 describes a prototype implementation of the proposed service platform and several example services. Section 5 presents the results of performance measurements obtained from experiments with this

1

prototype. Related work is shortly discussed in section 7 and the final section summarizes the results of this work.

2. PLATFORM ARCHITECTURE

The service enabling architecture evolves around a common network intermediary such as a caching proxy by adding new components, in particular a rule engine, a service invocation dispatcher, a local and remote service execution environment, and a message callout client and server. Figure 1 shows these components and their interaction in a service-enabled caching proxy and a remote service execution server. The following sections describe how these components help fulfill the requirements of a network edge service platform.

2.1 Service-Enabled Intermediary

In order to support the execution and installation of value-added service modules, a network intermediary must be augmented by additional components. Service modules may operate on the request/response message stream passing through the intermediary. They may modify or satisfy user requests or modify server responses. One example would be a content adaptation service which adapts requested HTML pages so that they can be viewed with small Internet appliances like PDAs.

It is important to note that more than one service module may operate on any given message stream, although only one service module can modify a message stream at a time. For example, an advertisement insertion service could insert an advertisement banner into a requested web page and a content adaptation service could adapt the same web page to the user's web access device.

Rule Engine

Since it may not be desirable to invoke installed service modules for all user requests received by a service-enabled intermediary, there must be a mechanism to trigger the execution of service modules if certain conditions are met.

Alternatively, this decision could be left to the service module. This would require, however, that all installed service modules are invoked for every request/response message passing through the intermediary. Obviously, this approach would introduce a significant delay in requests and responses for which no service modules should be executed. This is especially unacceptable if we assume that only a small fraction of all web transactions require the invocation of a service module.

Therefore, it is necessary for the parties for which service modules may be executed to provide rules that specify the conditions under which an incoming user request should trigger the execution of a service module. The rule engine component must also provide a secure interface through which authorized rule authors or system administrators can install, modify, and delete rules on the caching proxy.

Service Invocation Dispatcher

Once the rule engine has determined that a specific service must be applied to a user request or server response, the appropriate service module must be invoked. The invoked service module must then be provided with an interface to the corresponding request or response message stream
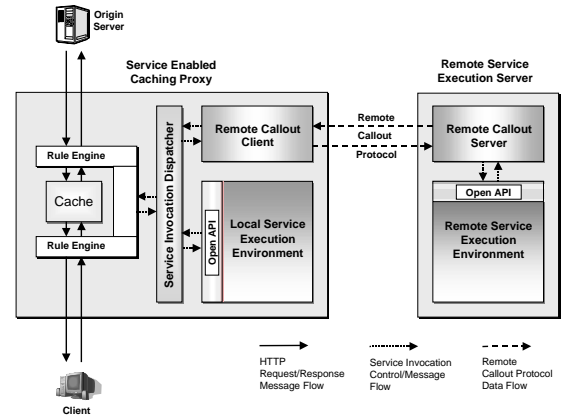

Figure 1: Service Platform Architecture

so that it can modify it. These tasks are accomplished by the service invocation dispatcher.

The rule engine notifies the service invocation dispatcher whenever a service module must be invoked. Service modules, however, may not only be executed on the intermediary itself. In some cases, it makes more sense to use a dedicated service execution server for this task. Therefore, the service invocation dispatcher must differentiate between local service modules and service modules on a remote server.

The service invocation dispatcher also performs simple security checks prior to the execution of a service module in order to prevent unauthorized modifications of messages. A service module provided by cnn.com, for example, should not be executed for a HTTP response from disney.com because a service module by cnn.com is not authorized to modify web objects other than those from cnn.com.

The service invocation dispatcher should also log all successful invocations of service modules in order to provide accounting and billing information to the providers of value-added services so that that they can charge their customers for each successful service invocation.

Local Service Execution Environment

The local service execution environment holds and executes service modules on the caching proxy. It must also be able to monitor and limit the use of resources by service modules during their execution. Service modules should not be able to significantly slow down the normal operation of a service-enabled intermediary. Therefore, resource intensive service types, e.g. virus scanning, may not be appropriate for the local service execution environment. These services should rather be hosted by a remote service execution server (see below).

The service execution environment must also provide a secure interface through which authorized parties such as content providers, access providers, and clients can install, update, and delete service modules.

2.2 Remote Service Execution Server

As mentioned before, the remote service execution server is a separate server dedicated to hosting and executing service modules that need more processing power and system resources than are available on a caching proxy, for example a virus scanning service.

Since the service modules on this remote server need to modify the request/response messages which pass through the intermediary, it is desirable to place the remote service execution server as close to the intermediary as possible.

The communication between the caching proxy and a remote service execution server requires a special protocol which we refer to as "remote callout protocol" since its primary purpose is to efficiently forward request/response messages and information about the service to be executed from the intermediary to the remote service execution server. A service-enabled intermediary may use this protocol to communicate with any number of remote service execution servers on which it wants to execute a service module. This approach also increases the scalability of the proposed service platform. If one service execution server alone cannot handle all service requests, the service platform operator may simply add another service execution server and configure the rules on the caching proxy in a way that balances the load on both service execution servers. This approach requires, however, that remote service execution servers regularly report their current load level to the caching proxy.

The hardware platform of a remote service execution server should be optimized to support the fast execution of service modules. Therefore, it may be desirable to build specialized remote service execution servers which can accelerate the execution of service modules. For example, a Java virtual machine (JVM) built into the hardware of a service execution server could increase the performance of service modules written in Java.

Remote Callout Client/Server

The remote callout client is a component on the caching proxy which talks to the remote callout server component on the service execution server. They communicate through the remote callout protocol in order to exchange request/response messages that may be modified by service modules on the remote service execution server. The remote callout client also sends information about the type of the forwarded message (request/response) and the expected behavior of the service module which is to be executed on the remote service execution server.

Remote Service Execution Environment

The remote service execution environment is similar to its local pendant on the intermediary with the exception that in this execution environment the restrictions on the available resources are not as strict. It is still desirable, though, to enforce the usage of resources so that the available resources are shared equally among running service modules.

3. SERVICE ACTIVATION RULES

3.1 Rule Engine Requirements

Performance is of paramount importance in an intermediary like a caching proxy whose main purpose is to accelerate the web access of its users. Even though a service-enabled intermediary offers additional value-added services to its users, this should not affect the performance of the normal operation of the intermediary, at least not significantly. It is not acceptable, for instance, that users who do not want to use these additional services are penalized for using a service-enabled intermediary instead of a regular intermediary.

The rule engine component of a service-enabled intermediary ensures that service modules are invoked only for those user requests or server responses that match certain rules. These rules are provided by the entities for which services can be executed. In the case of HTTP, these include at least content providers, access providers, and clients. Access providers are ISPs, enterprises, CDN service providers, and other organizations operating network edge devices like caching proxies, switches, and surrogates.

The rule engine component of a network edge service platform should meet the following requirements:

- It should be optimized for performance so that user requests for which no service modules are executed are not slowed down.

- It should accept rules in a standardized format from different external rule authors. This is necessary so that the parties for which service modules can be executed have a means to control when service modules are executed.

- Rules should allow rule authors to specify rule conditions on a very fine grained level, for instance on the message property level. This ensures that service modules are not invoked in cases where this is not absolutely necessary. For example, a virus scanning service should only be invoked for web objects which can possibly contain a virus.

- The rule engine should execute services according to the intended service execution order of rule authors but also be able to find a "sensible" service execution order among rules from different rule authors.

- The rule engine should take the dynamic message modifications of service modules into consideration. For example, a content adaptation service may under certain circumstances convert images from the JPEG into the GIF graphics format. A subsequent service, however, may not be able to process GIF images and does therefore not have to be executed any more.

- The rule engine should allow for service modules to be executed at different points within the round-trip message flow. This may be necessary, for instance, for services that only need to operate on messages that are served from the origin server and not from cache.

3.2 A Rule Specification Language

In order to simplify and standardize the exchange of rules between rule authors and service platforms, rules should be specified in a standardized rule specification language. [9] describes an XML rule specification language

for this purpose. In a caching proxy rules can be processed up to four times for each web transaction so that service modules can modify messages at different points within the message flow. The following section describes these four rule processing and service execution points.

Rule Processing Points

Figure 2 shows the typical HTTP data flow between a client, a caching proxy, and an origin server. The four processing points (1-4) represent locations in the round trip message flow where rules can be processed and service modules can be executed. Note that the message flow may skip points 2 and 3 after point 1 if the requested object can be served from cache.
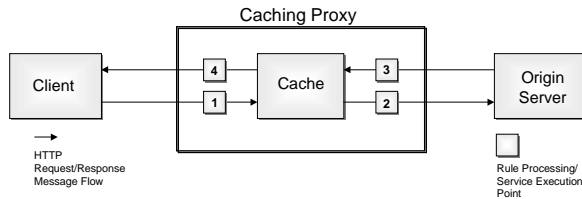

Figure 2: Rule Processing/Service Execution Points

Point 1 - Client Request A request from a client has been received. A possible cache lookup has not yet occurred.

Point 2 - Proxy Request The requested web object cannot be served from the cache and the origin server is about to be contacted for the HTTP resource.

Point 3 - Origin Server Response The response from the origin server has been received. It has not yet been stored in the cache.

Point 4 - Proxy Response The response from the cache or the origin server is about to be sent back to the client.

Depending on the service type, rules may be processed and services may be executed at any of the four points outlined in figure 2. A virus scanning service for instance should be executed at point 3 in figure 2 in order to scan all web objects for viruses before they can be stored in the cache. A URL-based request filtering service on the other hand should be executed at point 1 and an ad insertion service will probably be executed at point 4.

We can imagine that in the future there will be a need to have more processing points (at a finer granularity) than the ones mentioned above. It is also important to note that processing points are device-specific. A content-aware switch, for instance, has different processing points than a caching proxy.

Intermediary Rule Markup Language

The Intermediary Rule Markup Language (IRML) [9] allows rule authors to specify rules for network edge services in a standard format. It is important to create a standard rule format that will be supported by vendors of service-enabled caching proxies/surrogates so that rules can be distributed to different service platforms owned by different access providers in the same standard format.

The Intermediary Rule Markup Language also facilitates the exchange and discussion of network edge service rules between and within groups of rule authors.

IRML is an application of XML. Thus, its syntax is governed by the rules of the XML syntax as defined in [2], and its grammar is specified by a DTD, or Document Type Definition.

Valid and well-formed IRML documents consist of one or more rule modules. Each rule module contains a set of rules and information about the rule module provider. Rule modules are provided by a content provider, an access provider, or by a client (although usually indirectly through an access provider). In the future, however, rule modules may also be provided by other parties. On the content provider side, for example, one could differentiate between content hosters and content owners.

The rules contained in rule modules each consist of a number of conditions and a number of consequent actions that must be executed if the conditions are met. The conditions within a rule refer to message properties in the request or response of a given web transaction. They are met if the property value matches the pattern specified in the condition.

Order of Service Execution

The order in which service modules on the caching proxy are executed may influence the final result of a web transaction. For example, an ad insertion service executed against the result of a web page translation service may produce a different result than a reverse execution order.

A natural processing order for rule modules would be one that reflects the message flow from the user via the intermediary to the origin server and back. According to this order, up to three rule modules would have to be processed by the rule engine of an intermediary per transaction. For incoming requests at points 1 and 2 in figure 2, rule modules would be processed in the order: client rule module, access provider rule module, content provider rule module.

For outgoing responses at points 3 and 4, rule modules would be process in the opposite order: content provider rule module, access provider rule module, client rule module.

Within a single rule module, the caching proxy must process and execute all rules and actions in the order they are specified in the rule module (both within "property" and "rule" elements). If the rule processor determines that an action must be executed, it must do so before continuing the rule matching process since service modules may modify message property values. This may influence the result of subsequent pattern matches.

3.3 Rule Engine Architecture

Figure 3 illustrates the architecture of a complex rule engine that can process rules specified in IRML. It consists of a message dispatcher and parser component, a message context, an XML parser, a rule module optimizer and validator, a rule module repository, and a rule module processor.
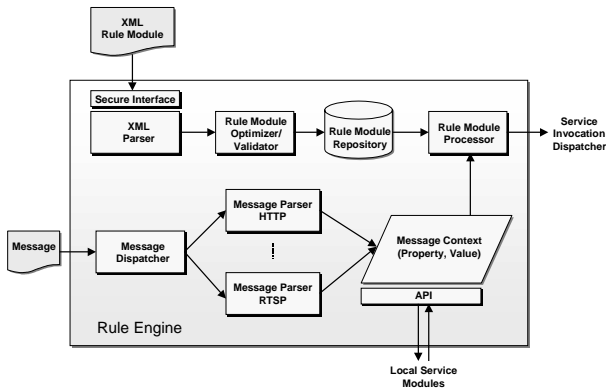
Figure 3: Rule Engine Architecture

XML Parser

IRML rule modules are XML documents which have to be distributed to the network edge service platforms on which they should be processed. The XML parser component in the rule engine is invoked whenever a new rule module is received. It parses the received IRML rule module and creates an internal tree representation of the XML elements and values. It then forwards this data structure to the rule module optimizer.

Figure 3 also shows a secure interface on the XML parser component. Although not a part of the rule engine itself, this interface must be provided to rule authors so that they can distribute their rule modules to network edge service platforms.

Rule Module Optimizer/Validator

The rule module optimizer and validator component of the rule engine pre-processes rule modules once they have been parsed by the XML parser. This component can detect inconsistencies in rule modules, for instance references to non-existing service modules. It also optimizes the internal representation of rule modules so that they can be processed more efficiently. The goal should be to minimize the number of pattern matches when rule modules are processed. Pattern matches, especially if regular expressions are involved, can be very expensive operations. The rule module optimizer and validator is invoked once for each new rule module and forwards the result of its operation to the rule module repository.

Rule Module Repository

The rule module repository holds all current rule modules in their optimized internal representation. The rule modules can be accessed by the rule processor when they need to be processed and they are added to the repository by the rule module optimizer. A new version of an existing rule module replaces an older version when it is added to the repository. The rule module owner, name, and version information are used in order to manage these rule module updates.

Message Dispatcher/Parser

A service-enabled network device is not limited to HTTP applications. It may also support other message-based protocols, for example RTSP [12]. The rule engine may therefore require more than one message parser for each supported protocol.

The message dispatcher is invoked once for each incoming or outgoing message. The message dispatcher directs the message to the approriate message parser. For example, an incoming HTTP request is forwarded to the HTTP message parser while an outgoing RTSP message is forwarded to the RTSP message parser. The protocol-specific message parser extracts name/value pairs from the headers of the processed message. The property names and values are then stored in the message context. This procedure ensures an important performance requirement; each message is parsed only once in a single pass. It is necessary to parse all message headers because it is not known in advance which property values will be modified by service modules.

Message Context

The name/value pairs of the request and response messages of a specific web transaction are stored in the message context. When the rule engine is invoked for the first time (at processing point 1), the message context contains only the request properties because a response message has not yet been received by the service platform at this point in the message roundtrip flow.

The message context offers an interface to other components whereby message property values can be read and modified. This interface is used by service modules which are executed in the local service execution environment and by the rule processor component (see below) in order to access message property values. The message context also contains the additional properties defined in IRML which cannot be directly mapped to message headers, for instance a user ID.

Rule Module Processor

The rule module processor is the core component of the complex rule engine. The rule processor decides which service modules are invoked at which points in the roundtrip message flow of a web transaction. Depending on whether a request is served from cache or the origin server, the rule processor is invoked at two or four of the processing points described in section 3.2.

When the rule processor is invoked, it retrieves appropriate rule modules from the rule module repository and processes them in the correct order. Rules within rule modules are matched in the order in which they are specified. Whenever a rule is matched, the rule processing stops and the service invocation dispatcher is called in order to execute the specified service module.

The rule processing continues after the execution of the service module which may have modified message property values. In the case of local service modules, these modifications are made directly in the message context through the message context API. The local service execution environment provides an interface to these functions. In the case of remote service modules, however, the

modified message properties are contained in the remote service execution server response. Therefore, the remote callout client on the service platform parses the remote callout response and updates the message context after each remote callout transaction.

4. PROTOTYPE IMPLEMENTATION

Our prototype implementation of the proposed generic service platform follows the architecture presented in the previous sections, but only the core functionality is supported. The service platform prototype consists of a service-enabled proxy with no caching capabilities and a remote service execution server. iCAP [4] is used as the remote callout protocol. Our prototype currently supports an older version of iCAP, but we are currently updating the prototype to support the latest iCAP version. The service platform does not have a local service execution environment. All service modules are hosted by the remote service execution server.

Our service platform is implemented on top of an existing non-caching proxy server for the FreeBSD operating system. The open-source proxy server is written in C. The non-caching proxy server was chosen for its simple design which facilitated the implementation of the service platform. Most of the proxy code could be reused, but had to be modified in various parts. Because some of our service prototypes require the caching of web objects, we simulate a cache by storing web objects on a local web server. Our prototype implementation contains a simple rule engine and supports the remote execution of service modules. These components are explained further in the following sections.

4.1 Rule Engine Implementation

Our protoype implementation has a simpler rule engine than the one described in section 3. The rule engine in our prototype maintains a list of rules which are limited to the following properties: index, status, client IP address, request URL, service module name, and iCAP mode. Table 1 shows a list of five example rules. The index property controls the order in which the rules are processed and the status property indicates whether a rule is active or inactive. Only active rules are processed by the rule engine.

The client IP address specifies for which user(s) a particular rule must be processed. This field may also contain a wildcard if a rule should be processed for all users. The request URL field contains the URL that must match the URL in a user request in order to trigger the execution of a service. The request URL property can also be specified in a pattern which may contain wildcards in each component of the URL: the protocol, the domain name/port number, and the request path. An example using all three wildcards is given in rule number 1 in table 1.

The name of the service module that must be executed if a rule matches is specified in the field "service module". The "iCAP mode", finally, tells the rule engine which of the two iCAP modes must be used for the specified service module. The iCAP mode field may also contain the value "negative". In this case, the rule is to be interpreted as a negative rule meaning that the specified actions must not be executed if the rule matches. Since negative rules always overrule positive rules, they allow for the exclusion of certain values when they are used in combination with wildcards in a corresponding positive rule. Example rule 4, for instance, excludes the user whose client IP address is 155.145.123.23 from the corresponding positive rule number 3.

For each web transaction the rule engine processes all rules and compiles a list of actions which is then passed to the service invocation dispatcher to execute triggered service modules in the specified order and iCAP modes.

Rule Con£guration

The rule sets on a caching proxy effectively control which services are executed for which HTTP transactions. A service-enabled caching proxy must therefore provide an open rule configuration interface to entities for which services may be executed. In our prototype implementation, we had implemented a web interface for this purpose. It allows authorized users to add, modify, and delete rules. It is also possible to enable or disable rules and to change the order of rules. We use basic HTTP authorization as defined in [5, 6] to restrict the access to this web interface.

4.2 iCAP Implementation

Our iCAP client implementation is integrated into the non-caching proxy server on top of which our service platform is built. The proxy server is augmented by an iCAP client which can compose, receive, and parse iCAP messages. The iCAP client uses existing, slightly adapted functionality of the proxy server, for instance in order to forward user requests to an iCAP server within an iCAP request.

The iCAP server is also implemented in C and on the FreeBSD operating system. An existing simple HTTP server provides the skeleton for a remote service execution server. The HTTP parser of the web server is modified so that it can parse the additional iCAP-specific headers and behave in accordance with the iCAP specification.

Service Execution Interface

In the existing prototype, service modules on the implemented remote service execution interface are invoked through a CGI interface on the iCAP server. We are currently in the process of adding other standard web server interfaces to the iCAP server such as FastCGI or the Java Servlet API which offer better performance than the CGI interface.

4.3 Service Prototype Implementations

We have implemented demo prototypes of four different example services in order to test our service platform prototype. All service prototypes have been implemented as CGI programs for the remote service execution server and are described in the following sections.

Language Translation Service

The translation service translates web pages to the user's preferred natural language with the help of a freely available language translation program (Altavista's BabelFish). The user's preferred language is derived from the "Accept-Language" header [6]. The translation service translates

Index	Status	Client IP	Request URL	Service	iCAP Mode
1	Enabled	*	*://*/*	reqfiltering	Resp.Mod.
2	Enabled	*	http://yahoo.de/*	adinsertion	Resp.Mod.
3	Enabled	*	http://*/*	translation	Resp.Mod.
4	Enabled	155.145.123.23	http://*/*	translation	Negative

Table 1: Example Rules

a web page if there is a mismatch between the user's preferred language and the document language.

Background Translation Service

The background translation service is similar to the language translation service with the exception that it returns the original web page to the user and prepares a translation for this page while the user views the original page. After having viewed the original page the user can decide whether he wants to see a translation for this page or not. He can request the translation by clicking on a link which the background translation service added to the original page. The translation can then be served to the user very quickly (possibly even from the intermediary's cache if the background translation service can pro-actively cache the translation).

Web Access Control Service

This service restricts web access by analyzing requested web content. All HTTP response messages are forwarded to the web access control service if it is enabled. Once invoked the service module scans the message for a list of words. If the service module finds a "forbidden word" in the requested web page, it returns an HTML error message to the iCAP client instead of the original web page. The proxy then forwards this error message to the user. Otherwise, the original HTTP message is returned to the iCAP client and from there to the user.

Personalization Service

The personalization service reduces the web access latency for personalized web pages. It satisfies certain user requests by assembling personalized news pages on the network edge rather than on the origin server. In this prototype, the user can customize a news page through a web interface in which he can select the news categories he is interested in. These settings are then saved in a cookie variable on the user's machine. Whenever this user requests his personalized news page, the value of the previously set cookie variable is sent along with his request. Thus, the personalization service module can obtain this cookie variable from the user request and assemble the news page accordingly.

5. PERFORMANCE EVALUATION

5.1 Rule Engine Performance

With this performance experiment we measure the additional delay that is introduced by the rule engine for requests that do not trigger any services. We therefore measure the performance penalty a user has to pay for using a service-enabled intermediary instead of a standard intermediary.

The test environment consists of our service-enabled proxy implementation with the simple rule engine design as described in section 4. There are 10 rules configured, but none of them fire a service for our test request which retrieves a 1.4 kbyte web page from a co-located local web server in order to rule out any external effects on the measurements. In order to obtain comparative data we have also installed a copy of our prototype implementation on the same server, but disabled the rule engine in that version.

The performance metrics in this experiment are:

1. the time from the beginning of the web transaction until a connection to the proxy is established (Connect)

2. the time from the beginning of the transaction until the first bit of the server response is received by the client (First Response)

3. the time it takes to complete the web transaction (Complete)

For the first test run in this experiment we configure the service-enabled proxy in our browser and request the sample web page. For the second test run we configure the modified service proxy in which the rule engine is disabled. We then request the same sample web page. The results shown in table 2 are average values from 50 identical requests with each proxy version. The variance between the measured values was very low.

The results show that the "connect" value is almost identical with both proxies which is not surprising because the rule engine has not even been invoked when this value is measured. The "first response" and "complete" values both show an additional delay of approximately only 0.5ms for the proxy where the rule engine is enabled. These results demonstrate that the additional delay introduced by the processing time of the simple rule engine is neglectable considering that average web transactions take hundreds of milliseconds.

It must be emphasized, however, that this value can increase when the rule engine has to process hundreds of rules instead of only ten rules. A more sophisticated rule engine like the complex rule engine described in section 3 also may have to match more message properties per rule than just the request URL and the client IP address and may be invoked up to four times for each web transaction. The additional delay of a more complex rule engine may be offset, however, by a more efficient implementation.

	Connect	First Response	Complete
Standard Proxy	1.511 ms	16.167 ms	18.450 ms
Service-Enabled Proxy	1.519 ms	16.652 ms	18.907 ms

Table 2: Rule Engine Delay

Services Enabled	Connect	First Response	Complete
None	5.810 ms	21.407 ms	27.846 ms
Background Translation	5.854 ms	137.506 ms	144.178 ms
Web Access Control	5.828 ms	134.777 ms	141.311 ms
Both	5.833 ms	246.275 ms	253.214 ms
No Proxy	3.698 ms	6.261 ms	10.725 ms

Table 3: Service Introduced Delay

6. SERVICE INTRODUCED DELAY

This performance experiment examines the additional delay which is introduced by value-added services like the background translation and the web access control service. These services do not reduce the access latency of web requests but modify or filter server responses as they pass through the service-enabled intermediary.

The test environment consists of our service-enabled proxy server and a co-located iCAP server on which the background translation and web access control service modules are installed. The client is located in the same LAN in order to reduce the influence of any network congestions. In each test run, the client is configured to use the proxy server and requests a very small web page (391 bytes) from a co-located web server. We want to measure the additional delay introduced by the background translation and web access control service modules.

In the first test run, all rules are disabled and the client receives the original web page. In the second test run, the background translation service is enabled and adds a translation image link to the web page before it is sent to the client. It is important to note that the background translation service does not translate the requested web page or wait for its translation. Instead, the service only makes a small modification to the original web page by inserting a new HTML link.

The web access control service is enabled in the third test run and scans the web page for illegal content. Both service modules are enabled in the fourth test run which means the original page is first modified by the background translation service and then scanned by the web access control service. The last test run was conducted with no proxy configured in order to obtain some reference data. The results shown in table 3 are average values from 50 identical requests in each test run. The performance metrics are the same as in the previous experiment. The variance between the measured values was low.

The results show that the "connect" value is comparable for all test runs with the exception of the last test run where no proxy was used. The lower value for this test run indicates that the web server can handle incoming requests faster than the service-enabled proxy. This is understandable because we have used a pre-forking web server for these tests while the proxy server on top of which we implemented our service platform spawns a new process for each incoming request.

The additional delay introduced by the background translation and web access control services is for both services approximately 110ms. This value can be determined if we subtract the "first response" or "complete" value of the first test run where no services are triggered from the corresponding values of the second or third test run. The remainder equals the service delay introduced by the execution of each service since all three test runs were conducted under otherwise equal conditions. An additional access delay of 110ms is quite significant even though average web transactions last hundreds of milliseconds. However, the poor performance of the service executions can be explained partly by the limitations in the used prototype: The implemented service modules use the CGI interface which means that a new process is forked for each service invocation. Also, the implemented older iCAP version did not yet have support for chunked encoding and persistent connections between iCAP client and server.

The fourth test run in which both services are executed one after the other clearly shows that the service introduced delay has doubled as well. This becomes obvious if the "first response" and "complete" values of the first run are subtracted from the corresponding values in the fourth test run. The result, ca. 220ms, equals twice the introduced delay from test runs 2 and 3. These numbers suggest that a mechanism that allows for the execution of multiple service modules during a single iCAP transaction may decrease the service introduced delay. In the current design, two service executions on the same iCAP server require two separate iCAP transactions even when they both operate in the same iCAP mode like the background translation and web access control service.

7. RELATED WORK

The idea of intelligent networks is not new. In the past, there have been different approaches to intelligent networks. A lot of research has been done on "active networks" [14, 13]. The active networks approach, however, is not restricted to the edges of the network and also operates on lower levels of the protocol stack (i.e. on a packet-

by-packet basis). Active networks have not yet received widespread support.

Many different applications for network edge intermediaries, particularly for (caching) proxies, have already been suggested and explored in past publications. For example, in [3], Deng and Chi present a dynamic active proxy for local web advertisement insertion and its benefits. The concept of content transcoding or transformation performed on caching proxies has been discussed for a long time. More recently, the focus in content transformation has shifted to the requirements of mobile web access. In [8] and in [1], for example, the authors present a transcoding proxy for mobile web browsing.

The just recently started IETF activity Open Pluggable Edge Services (OPES) attempts to create a standardized network edge service platform. The OPES initiative has incorporated several Internet Drafts. Among them are the Internet Draft that defines the Intermediary Rule Markup Language (IRML) [9] and an Internet Draft which presents service examples [10]. Another Internet Draft named "Extensible Proxy Services Framework" [15] describes a network edge service framework similar to the one explained in this article but from a rather high-level perspective. OPES is also considering to develop the current version of iCAP [4] into a standardized remote callout protocol.

With the exception of the IETF activity OPES, related work in this area appears to be focused on specific applications for network edge intermediaries. Some service ideas like ad insertion or content transformation have been explored and also applied before, but the notion of a standardized full-fledged, multi-purpose service platform at the network edge is new.

8. CONCLUSION

In this article, we have presented the benefits of value-added, intelligent services operating on intermediaries at the edges of the Internet. We have presented a generic service platform architecture with special emphasis on the rule engine component. Our prototype implementation of a simple service platform and four example services proved the general concept proposed in this article and also demonstrated that a variety of different service modules can be implemented on top of a generic service platform with very little efforts. Our performance measurements have shown potentials for future design and implementation improvements, but also confirmed that intermediary services and the additional components of a service-enabled intermediary (in particular the rule engine) do not necessarily introduce a significant additional delay to web transactions passing through such a service-enabled intermediary.

This article has also shown the required complexity of a generic service platform. Open issues not addressed in this article will therefore be subject of future work. Within the rule engine component, for example, this includes the rule module distribution process and the resolution of rule conflicts.

9. REFERENCES

[1] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul. An active transcoding proxy to support mobile web access. *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, 1998.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, W3C, Oct. 2000.

[3] J. Deng and C. Chi. Local Web Advertisement Through Dynamic Active Proxy. In *ICME00*, page TP11, 2000.

[4] J. Elson et al. ICAP, the Internet Content Adaptation Protocol. Internet Draft, Internet Engineering Task Force, Jan. 2000.

[5] R. Fielding, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.0. Request for Comments 1945, Internet Engineering Task Force, May 1996.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Request for Comments 2616, Internet Engineering Task Force, June 1999.

[7] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[8] R. Han, P. Bhagwat, and R. L. et al. Dynamic adaptation in an image transcoding proxy for mobile WWW browsing. *IEEE Personal Communication*, 5(6):8–17, Dec. 1998.

[9] M. Hofmann and A. Beck. IRML: A Rule Specification Language for Intermediary Services. Internet Draft, Internet Engineering Task Force, Mar. 2001.

[10] M. Hofmann, A. Beck, and M. Condry. Example Services for Network Edge Proxies. Internet Draft, Internet Engineering Task Force, Nov. 2000.

[11] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems 2*, pages 277–288, November 1984.

[12] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (RTSP), request for comments 2326, Apr. 1998.

[13] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[14] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), Apr. 1996.

[15] G. Tomlinson et al. Extensible Proxy Services Framework. Internet Draft, Internet Engineering Task Force, July 2000.