TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Intermediary Message Processing using Open Pluggable Edge Services

Ausgeführt am Institut für

Informationssysteme
Distributed Systems Group
Technische Universität Wien

unter der Anleitung von
Univ.Prof. Mag. Dr. Schahram Dustdar

durch

Martin Thelian
Lederergasse 25-27/2/16
1080 Wien

Wien, 05. März 2007

Unterschrift (Student)

Danksagung

Für die fortwährende moralische Unterstützung und ihren Rückhalt möchte ich meiner Familie und meiner Freundin Sabrina Painhaupt danken.

Ganz besonderer Dank gilt Roland Ramthun für das kritische Gegenlesen, sowie Prof. Schahram Dustdar für die Betreuung dieser Arbeit.

Kurzfassung

Die stetig wachsende Zahl an Internetnutzern und die damit verbundene Zunahme an Traffic im WWW führte im Laufe der Zeit zur Entwicklung von Content-Delivery-Netzwerken (CDN) und dem Einsatz von zusätzlichen Zwischenrechnern an den Netzwerkgrenzen, die es Content-Providern erlaubten ihre Inhalte räumlich näher am Enduser abzulegen, um deren Verfügbarkeit zu erhöhen und die Zugriffszeit zu verbessern.

Mit der Zeit erweiterten die Provider ihre ohnehin bereits vorhandene, nur auf den Transport von Daten ausgerichtete Netzwerkinfrastruktur, um zusätzliche Inhalts-orientierte Dienste wie die Filterung, Konvertierung oder Personalisierung von Inhalten anbieten zu können. Dies erforderte die Aufrüstung von Zwischenrechnern mit zusätzlichen Fähigkeiten zur Ausführung und zum entfernten Aufruf von Services. Welche Maßnahmen zur Aufrüstung der Rechner notwendig sind, wurde in verschiedenen Frameworks beschrieben. Wegen mangelnder Standardisierung hat aber keiner der veröffentlichten Lösungsansätze eine weite Verbreitung gefunden.

Um das Problem der mangelnden Standardisierung zu lösen, wurde die IETF Open-Pluggable-Edge-Services Working-Group (OPES-WG) mit der Zielsetzung gegründet, ein flexibles und offenes Framework für den Einsatz derartiger Application-Level Services zu entwickeln. Zum Entstehungszeitpunkt dieser Diplomarbeit hat die OPES WG zehn RFCs veröffentlicht, welche die Architektur des Frameworks, Anforderungen an Richtlinien- und Authentifizierungsmechanismen, Anwendungsbeispiele und Einsatzszenarien sowie ein Callout-Protokoll zum entfernten Aufruf der OPES-Services beschreiben.

Ziel dieser Arbeit ist die Analyse des OPES-Framework und seiner Komponenten. Weiter wird eine Prototyp-Implementierung des OPES-Frameworks vorgestellt, welche den Empfehlungen der OPES-WG Standards folgt, und unter anderem einen als HTTP-Proxy fungierenden OPES-Prozessor, einen OPES-Callout-Server, eine Implementierung des OPES-Callout-Protokolls (OCP) und des OCP-Profils für HTTP umsetzt. Des Weiteren enthält der Prototyp eine IRML-basierte Rule-Engine zur regelabhängigen Ausführung von Services sowie eine Laufzeitumgebung für Proxylets. Im Anschluss wird eine Fallstudie präsentiert, welche die Funktionsfähigkeit und praktische Nutzbarkeit des Prototyp-Systems demonstriert. Sie enthält Beispiel-Services sowohl für HTTP-, als auch für SOAP-Nachrichten.

Die Fallstudie kommt zu dem Schluss, dass der vorgestellte Prototyp gut geeignet ist um gebräuchliche Mehrwert-Dienste – wie die Transformation oder Generierung von Inhalten – anzubieten. Weiters wird gezeigt, dass sich mit Hilfe von OPES auch Services umsetzen lassen, die normalerweise nur von speziellen SOAP-Intermediary Rechnern angeboten werden können.

Obwohl mit Hilfe der Fallstudie die Funktionsfähigkeit des OPES-Frameworks gezeigt werden konnte, wurden im Zuge der Durchführung dieser Arbeit einige Ungereimtheiten und Probleme in der OCP-Spezifikation und der Zusammenarbeit zwischen OPES und Proxylets entdeckt. Diese Probleme werden am Ende dieser Arbeit detailliert beschrieben.

Abstract

The increasing number of Internet users and the continuous growth in web traffic lead to the development of Content-Delivery-Networks (CDN) and the deployment of network intermediaries on the edge of a network to allow content-providers to move their content closer to the end user.

Over time the providers had the idea to extend their existing network infrastructure classically specialized on pure content-delivery, with the capability to provide additional content-oriented services such as content-transformation, content-filtering- or content-personalization services. Deploying these services required the equipping of network-intermediaries with additional components for service-execution or -invocation, and therefore several frameworks were developed, describing how this can be achieved. But because of a lack of standardization, these approaches have not seen widespread use so far.

To solve these problem, the IETF Open-Pluggable-Edge-Services Working-Group (OPES-WG) was chartered and has spent a lot of work in the development of an architectural framework that is capable to support such application-level service. At the time of writing this thesis, the OPES WG has published ten RFCs describing the framework architecture, policy- and authorization requirements, use-cases and deployment scenarios, and a callout-protocol for the remote invocation of OPES services.

This thesis analyzes the OPES framework and its components. It presents a prototype implementation of the OPES framework following the proposed OPES-WG standards, including an OPES-processor acting as HTTP-proxy, an OPES-callout-server, an implementation of the OPES-callout-protocol (OCP) and the OCP profile for HTTP. Furthermore the prototype implementation contains an IRML rule engine for the rule-based execution of services, and a service-execution-environment for proxylets.

Thereafter a case-study is presented to illustrate the functionality and practical suitability of the prototype system. It includes service-applications performing value-added services on HTTP- as well as SOAP-messages. The case-study points out that the prototype is well suited to perform content-transformation or -generation services on HTTP-messages, but can also be used to perform services acting as an active SOAP-intermediary.

Nevertheless, some inconsistencies in the OCP specification and problems in the combination of OPES with proxylets were detected. These problems are described in detail at the end of this thesis.

Contents

1 Problem Description

1.1 Introduction and Motivation

The Internet has evolved from being a simple data-centric network used to query and publish information, toward a more service-centric network where active intermediaries play an important role in providing new value-added services to the end-user.

This trend can be seen both in the area of Content-Delivery-Networks (CDN) [1] where the capability of network-intermediaries was extended to provide additional content-oriented services such as content-transformation, -filtering or -generation-services, and in the emerging trend of developing Web-service-based applications, and the use of service-intermediaries to provide intermediary-processing of web-service messages.

In the area of content-delivery-networks, a lot of research has been done regarding the development of frameworks describing how service-execution-environments - needed to host these new content-oriented services - can be integrated easily into existing network infrastructure nodes located along the data-path between the users and content-providers. Most of the frameworks suggest the extension of commonly used web-proxies to convert them into service-enabled web-caches so that they are able to provide new content-oriented services to the end-user.

In the area of web-service platforms, the support of intermediaries was one of the major design goals of SOAP [2] and is supported through the SOAP extensibility-model. These SOAP-intermediaries are often used for message-tracing and -auditing, securing of message-exchange or routing of SOAP-messages.

1.2 Problem Definition

Although there are a lot of frameworks available today, which give you definitions how to extend a content-delivery-network with value-added services toward an edge-service-network, which has the capability to provides new value-added services to the end-user, these approaches have not seen widespread use so far.

The main reason for this is the lack of a standard, describing how to deploy services, how to integrate them into the existing data-flow, how to select appropriate services or how to use callout-protocols to execute services remotely.

To solve these problems, the OPES (Open Pluggable Edge Services) Working Group was chartered to define “an architectural framework to authorize, invoke, and trace such application-level services for HTTP” [3].

“In particular, the WG [editor's note: Working Group] has developed a protocol suite for invocation and tracking of OPES services inside the net. The protocol suite includes a generic, application-agnostic protocol core (OCP Core) that is supplemented by profiles specific to the application-layer protocol used between the endpoints. So far, the WG has specified an OCP profile for HTTP, which supports OPES services that operate on HTTP messages.” [3]

In the area of web services, the usage of service intermediaries is already supported by the

SOAP-standard and is supported by commonly used SOAP-servers. But what about intercepting and processing SOAP messages using Open Pluggable Edge Services?

The goal of this thesis is to analyze the IETF OPES framework and protocol-suite, and to design and implement a research-prototype of an OPES-processor, an OPES-callout-server, a service-execution-environment for edge-service, a simple engine for rule-based triggering of edge-services, the OPES callout protocol (OCP) and the OCP profile for HTTP.

The functionality of the OPES-architecture and callout-protocol is demonstrated on the basis of common usage-examples for edge-services, deployed on callout-servers, and designed to operate on HTTP request- and response-messages.

Furthermore, this thesis demonstrates, how edge-services can be used outside of their common area of application (which is in the majority of cases filtering, transformation or personalization of HTML content delivered using HTTP-messages) by implementing example-services operating on web-service request- and response-messages.

1.3 Organization of this thesis

Chapter 2 gives an overview over content-delivery-networks (CDN) and content-oriented-services, an introduction to the IETF Open-Pluggable-Edge-Service (OPES) architectural framework as well as its components, and explains the underlying concepts and technologies used by the prototype implementation presented in chapter 4 of this thesis.
Furthermore, standards and technologies strongly related to the OPES standard such as IRML and Proxylets are introduced.

Chapter 3 describes web-service technologies such as SOAP, which are used by the use-cases presented in chapter 5.

Chapter 4 presents the research-prototype implementation of the OPES-framework and describes its architecture and design.

Chapter 5 presents a case-study to illustrate the functionality of the research-prototype-implementation. It starts with the demonstration of some common use-cases acting on HTTP-request- and -response-messages and proceeds with the presentation of edge-services acting on SOAP-messages.

Chapter 6 gives an evaluation of the work presented in this thesis and lays out future work.

Chapter 7 contains a conclusion and a summary of this thesis.

2 Content-Oriented Services

This chapter gives an introduction into the area of Content-Networks (CN) and content-oriented services.
It starts with a description of Content-Networks (see chapter 2.1) and how they evolved from networks providing simple load-balancing- and caching-services toward edge-service-networks providing additional content-oriented services. After this, the Open-Pluggable-Edge-Services (OPES) framework, its architecture and protocol are described (see chapter 2.2). Furthermore, technologies that are strongly related to Open-Pluggable-Edge-Services such as Proxylets (see chapter 2.3), the Intermediary Rule Markup Language (see chapter 2.4) or the OPES Meta-data Markup Language (see chapter) are described.

2.1 Content Networks

As the Internet grew in terms of traffic, number of pages and domains, new technologies used for content caching- and replication were introduced to spread the load of client requests among multiple servers or proxies.

On the one hand caching-proxies were deployed close to the user to improve performance and availability by serving client requests from the own cache, instead of sending them directly to the original server. On the other hand, server farms were used for load-balancing of requests across multiple servers [4].

These new types of network-entities require an application-layer-routing of request- and response-messages and span a virtual application-layer network, layered on top of the underlying packet-network, that deals with the delivery of content. That's why this overlay network is also called a Content-Network (CN) [5].

Content Delivery Network:

A Content-Delivery-Network[1] (CDN) is a special form of a CN. It tries to push content closer to the user by using mirror servers, allowing content-providers to copy an entire site to multiple locations [6]. The CDN consists of a request-routing-, a content-delivery-, a content-distribution- and an accounting infrastructure. The request-routing infrastructure redirect a client request to one of the available mirror servers, the content-distribution infrastructure is used to move content from the original server to the CDN-mirror-servers, the content-delivery infrastructure delivers copies of the content to the end-users and the accounting-infrastructure tracks and collects data, used to optimize the request-routing and content-distribution [4].

Edge Service Networks:

An Edge-Service-Network[2] is another type of an overlay-network. It is layered on top of a Content-Network (CN) or Content-Delivery-Network (CDN) and extends it with additional functionality to support intermediary processing of content, flowing through the underlying Content-Network. [5]

The “Content-Services” provided by the Edge-Service-Network are hosted on intermediary

[1] sometimes also referred to as Content Distribution Network

[2] sometimes also referred to as Content Service Network (CSN)

nodes of the underlying Content-Network. This service-enabled intermediaries are typically located on the edge of a network. That's why the hosted services are also called edge-services.

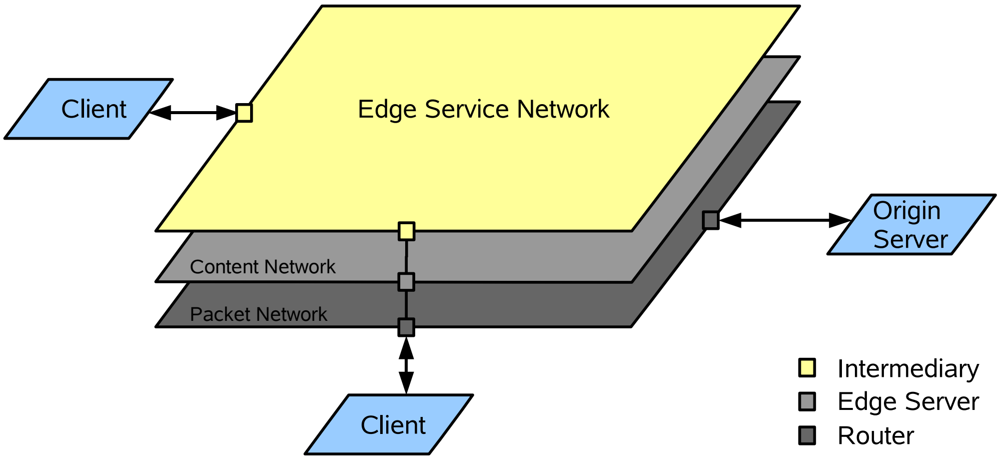


Figure 1: Edge Service Network (based on [5])

An OPES Service-Network is a special form of an Edge-Service-Network and is described in detail in the next chapter.

2.2 Open Pluggable Edge Services

2.2.1 The OPES Working Group

The Open-Pluggable-Edge-Services Working-Group (OPES WG) was chartered by the IETF in December 2000 at the 49th IETF meeting with the initial goal "to develop the protocol and mechanisms for an open content service architecture" [7]. After a long, controversial discussion regarding the risks and security threats that could arise from the deployment of such an intermediary system, and whether the IETF should stay away from such an system [8] or not, the working-group was finally chartered in February 2002 and began to work on the general architecture for a content-service-network, a generic callout-protocol and a specific profile for HTTP-based services. [7].

At the time of writing this thesis, the following set of specifications exists:

- RFC3752: *Open Pluggable Edge Services (OPES) Use Cases and Deployment Scenarios* [9]

  discusses the three different categories of OPES-services (request-modification-, response-modification-, response-generation services) and describes various service deployment scenarios.

- RFC3835: *An Architecture for Open Pluggable Edge Services (OPES)* [10]

  describes the architectural components of the OPES-framework (OPES-entities, OPES-flows, OPES-rules) and discusses security- and privacy considerations.

- RFC3838: *Policy, Authorization, and Enforcement Requirements of the Open Pluggable Edge Services* [11]

  describes the requirements to a policy-architecture, its basic components and functions, and formulates requirements for rule-formats and rule-management.

- RFC3837: *Security Threats and Risks for Open Pluggable Edge Services* [12]

  discovers and analyzes security-threats and risks to OPES-data-flow and OPES-components.

- RFC3914: *Open Pluggable Edge Services (OPES) Treatment of IAB Considerations* [13]

  addresses nine architecture-level considerations formulated by the IETF Internet Architecture Board (IAB) in [8] when the OPES working group was chartered

- RFC3836: *Requirements for Open Pluggable Edge Services (OPES) Callout Protocols* [14]

  formulates functional-, performance- and security-requirements that must be satisfied by an OPES callout protocol (OCP) to support remote execution of OPES-services.

- RFC4037: *Open Pluggable Edge Services (OPES) Callout Protocol (OCP) Core* [15]

  describes the application-independent OPES-callout-protocol, the syntax and semantic of callout-messages, the different types of dataflow and message exchange patterns.

- RFC3897: *Open Pluggable Edge Services (OPES) Entities and End Points Communication* [16]

  contains requirements regarding service-tracing- and -bypass-mechanisms for Open Pluggable Edge Services

- RFC4236: *HTTP Adaptation with Open Pluggable Edge Services (OPES)* [17]

  defines an application-specific profile describing how HTTP messages can be communicated by using the OPES callout protocol (OCP).

- RFC4496: *Open Pluggable Edge Services (OPES) SMTP Use Cases* [18]

  describes various use-cases and deployment-scenarios for using OPES to adapt SMTP-messages.

In addition to the documents listed above, there are a lot of predecessor documents related to Open-Pluggable-Edge-Services, which were published as Internet-Drafts but did not reach the stage of a standards track. However, some of them are of importance for this thesis and are therefore mentioned in the list below.

- *Proxylet Local Execution Environment Java Binding* [19]

  defines a set of Java-interfaces providing an API that should be implemented by proxylet-execution-environments to allow proxylet-developers to develop vendor-independent proxylets.

- *IRML: A Rule Specification Language for Intermediary Services* [20]

  describes an XML markup language used to define policy-rules, which are processed by an intermediary that triggers the execution of OPES-services according to conditions and actions defined in an IRML-rule.

- *Sub-System Extension to IRML* [21]

  describes how IRML could be extended with additional sub-systems providing additional properties and matching rules.

2.2.2 OPES Architecture

The Open-Pluggable-Edge-Service (OPES) architecture as described in [10] consists of three basic elements:

- *OPES entities*
- *OPES flows*
- *OPES rules*

These elements are combined to build a larger system of entities acting as a service-network that provides new content-oriented services to the content-provider or the content-consumer.

The following sections describe this service-network and its components in greater detail.

OPES Entities:

An OPES entity is an application that operates on the application-message data-flow, which is exchanged between a data-consumer- and a data-provider application.

To protect application-message data against unauthorized modifications, the OPES framework follows the one-party-consent model, which was defined by the Internet Architecture Board (IAB) in [8]. In this model, each OPES entity must be authorized by either the data-consumer or the data-provider to process a given application-message.

The set of all OPES entities that are authorized by one of the endpoints is called an OPES system.

OPES System:

An OPES system, as defined in [16], consist of OPES entities that are directly authorized by one of the application-layer endpoints, but could also be formed by induction if the authority agreement allows re-delegation. In this case, authority is delegated from already authorized entities to further entities.

Because OPES entities can act either on behalf of the data-consumer- or of the data-provider-application, not more than two OPES systems can process the same application-message data-flow at the same time.

OPES Service Network:

The entities of an OPES system are layered on top of an existing underlying content-network (see chapter 2.1) and span a virtual application-layer network that provides new application-level services, which are acting on the content exchanged between the data-provider and the data-consumer. That's why this virtual network is called an OPES services-network and the supplied services are called content-oriented services.

Depending on the authoritative domain an authorized entity belongs to, an OPES service-network can be either a delegate overlay-network or a surrogate overlay-network [9].

Delegate-Overlays are OPES-service-networks that have the authority to perform data-services on behalf of the data-consumer. Thus, all entities of the overlay-network logically belong to the authoritative-domain of the data-consumer and are trusted to adhere to the agreed system-policies.

Surrogate-Overlays are OPES-service-networks that have the authority to act on behalf of the data-provider. Therefore, all network entities logically belong to the authoritative-domain of the data-provider and are trusted to adhere to the agreed policies.

Additionally to components directly located in the content-path between the data-provider and -consumer, an OPES entity may also communicate with other logical components such as OPES-Administration-Servers[3] or Remote-Callout-Servers[4], which are members of the same authoritative domain as shown in figure 2 and 3.

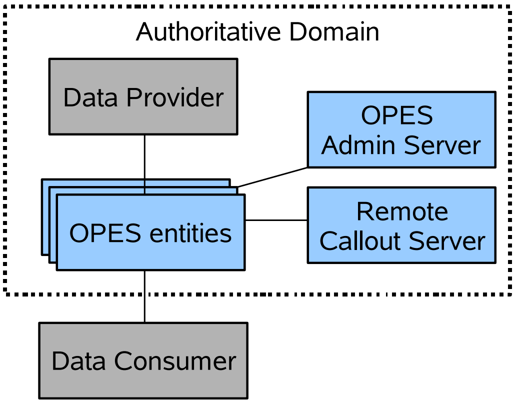


Figure 2: OPES Architecture – Authoritative Domains for Surrogate Overlays (based on [16])

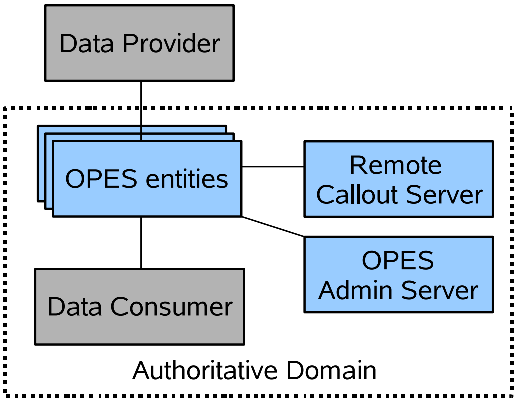


Figure 3: OPES Architecture – Authoritative Domains for Delegate Overlays (based on [16])

Typical services provided by Delegate-Overlays are content-filtering-, content-personalization- or virus-scanning-services. Typical Surrogate-Overlay services are content-transformation or advertisement-insertion services.

OPES Service Application:

The content-oriented services provided by an OPES service-network are implemented by various OPES service-applications, that reside inside OPES processors and analyze and modify application-message data flowing through these.

An OPES-processor is typically located on the edge of a network and is (a part of) an intermediary node of the underlying content-network, for example an application-level proxy (e.g. a HTTP-proxy) or transport-level gateway, and contains one or more OPES entries, whereas an entity can be either a data-dispatcher-application or a service-application.
An OPES-processor must include an OPES data-dispatcher-application and may additionally contain multiple OPES service-applications.

An OPES-processor can either execute OPES-service-applications locally in a local service-execution-environment or may delegate the processing of an application-message to other OPES service-applications that are not hosted locally, but located on one or more OPES callout-servers.

[3] A component that is sometimes used to perform authentication, authorization and accounting (AAA) functions for an OPES networks [5]; cp.: Policy-Decision-Point (PDP) as described in chapter 2.2.3
[4] a component providing OPES-service-applications that can be invoked using a callout protocol

Because both, an OPES-processor and an OPES-Callout-Server, can be used to host and execute OPES-service-applications, they are also called OPES-agents. Thus an OPES-system can be seen as a set of OPES-agents that collaborate to provide content-oriented services of various types. Which types of services are possible is described in chapter 2.2.4.

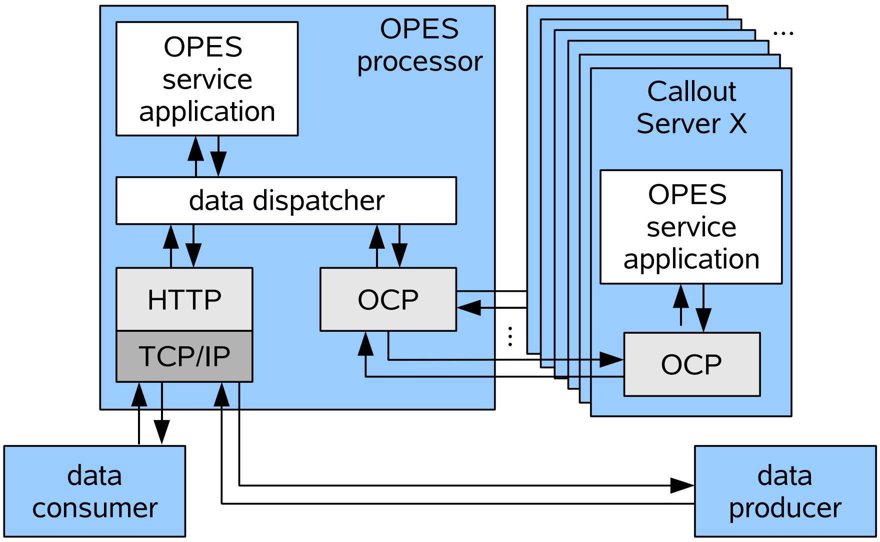


Figure 4: OPES Architecture – Components (based on [10])

If an OPES processor decides to delegate the processing of an application-message to an OPES callout-server, it invokes the desired callout-service using the OPES callout-protocol (OCP) as depicted in figure 5.

The OPES callout-protocol (OCP) itself is application-message independent, but can be extended with additional application-specific profiles, describing how to encapsulate specific application-protocols messages such as HTTP request- or response-messages.
A detailed description of the OPES-callout-protocol is given in chapter 2.3.5 and the OCP-HTTP-profile is described in chapter 2.2.6.

OPES Data-Dispatcher:

An OPES data-dispatcher, in some OPES draft documents also referred to as OPES engine, is another type of OPES entity that resides inside an OPES processor.

An OPES data-dispatcher represents a Policy-Enforcement-Point (PEP) where application-messages, which flow through the OPES processor, are parsed and matched against policy rules that are specified by rulesets.

A policy ruleset consists of OPES rules, whereas each rule contains a set of conditions and corresponding actions. A rule condition specifies the criteria that must be met by an application message to match against the rule, whereas a rule-action specifies the OPES-service-application to execute when the conditions of the OPES-rule are met.

The OPES working-group does not define concrete mechanisms how to configure, evaluate or enforce OPES-rules, but it has published a document describing the requirements that have to be satisfied by a concrete implementation of a policy-architecture [11] (see chapter 2.2.3).

Since an OPES data-dispatchers is mandatory to enforce policies, each OPES processor must include an OPES data-dispatcher, and therefore there must be at least one OPES data-dispatcher presented in an OPES flow. Nevertheless it's also possible to have multiple OPES data-dispatcher present in an OPES-flow.

OPES Flow:

As depicted in figure 5, “an OPES flow is a cooperative undertaking between a data provider application, a data consumer application, zero or more OPES service applications, and one or more data dispatchers” [10].
Thus a single original application-message can sequentially flow through multiple OPES service-applications. With this “chaining” of OPES-application-services, it is possible to combine quite simple service-applications together to perform a more complex service, whereas each OPES application-service involved in application-message-processing can be located on different OPES agents, either on OPES processors or even on OPES Callout-Servers that are connected to OPES processors via OCP.

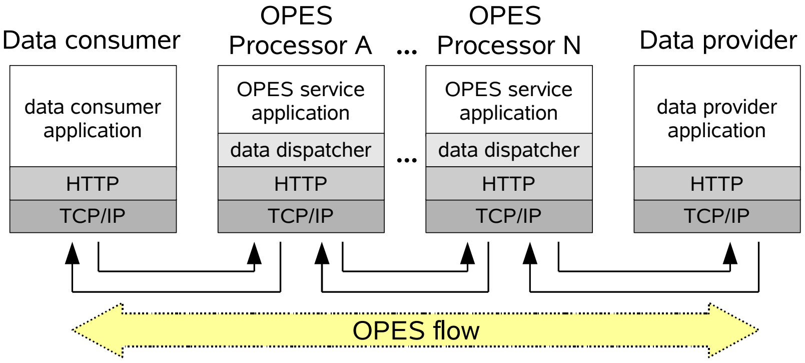


Figure 5: OPES Architecture - OPES flow (based on [10])

To distinguish between multiple OPES processors present in an OPES flow, and to ensure verifiable system-integrity, each OPES processor must be explicitly addressable by the end-user at the IP-layer and must be consented to by either the data-consumer- or data-provider-application. The explicit addressability is e.g. required for service-tracing or to bypass-services (see chapter 2.2.6).

2.2.3 OPES Policy

An OPES service-application analyzes and (possibly) modifies application-messages that are exchanged between a data-provider- and a data-consumer-application, but intercepted by an OPES processor. Which concrete services are applied to a given application-message data-flow depends on the used policy.

In general, policies can have different scopes. On the one hand there are policies describing which services should be called under certain conditions and which parameters to use for the execution of these services. On the other hand there are policies describing the desired behavior of the executed services [11].

The OPES policy-architecture described by Babir, Batuner et al in [11] is limited to the first category of policies and is described in the following section.

OPES Policy Architecture:

The OPES-policy-architecture consists of three main components:

- *Rule Author*
- *Policy Decision Point (PDP)*
- *Policy Enforcement Point (PEP)*

[11] mainly describes the requirements for the decomposition of the policy-architecture into various components and formulates requirements for the interfaces between these components, but does not define concrete mechanisms how to configure, evaluate or enforce policy rules. Thus, it is up to the concrete implementation which format to use for the definition of policy rules or how to evaluate the defined rules, as long as the evaluation of rules defined in a given format "result[s] in the same unambiguous result in all implementations" [10].

The decomposition of the architecture in its components and the interfaces between the components is depicted in figure 6.

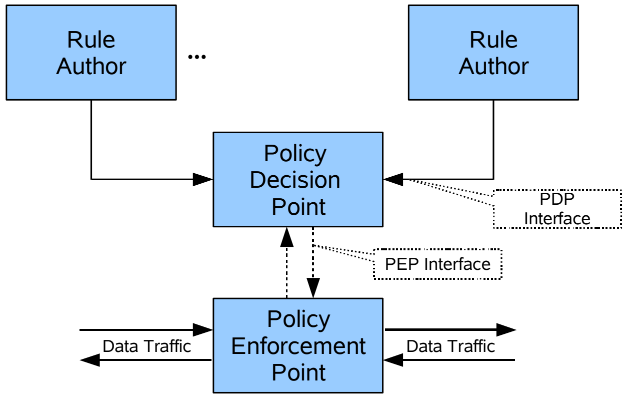


Figure 6: OPES Policy – Components (based on [11])

As shown above the OPES policy-architecture defines separate policy-decision- and -enforcement-points. But this does not imply that these two components must reside on different OPES processors. Nevertheless the separation of the PDP from the PEP, e.g. in the form of a separate OPES-Administration-Server, has the advantage that a single PDP can provide its services to multiple PEPs and that all OPES rules, which are active in an OPES system, can be managed at a central point by the OPES-system administrator.

Rule Author:

The rule-author provides the OPES rules that are used by a Policy-Enforcement-Point to determine the services that should be applied to a given application-message, which is sent from a data-provider to a data-consumer or vice versa.

The rule-author provides the policy-rules in form of a ruleset containing multiple OPES-rules, whereas each OPES-rule defines a set of conditions that must be met by a given application-message to trigger one or more actions, which are defined in the OPES-rule, too.
Because the OPES-working-group has not yet defined a concrete format that should be used for the definition of these OPES-rules, it depends on the Policy-Decision-Point interface in which format the rules must be provided.

An example for a concrete rule-definition-format is the Intermediary Rule Markup Language (IRML) [20] as described in chapter 2.4

The rule-author can be either the data-provider, the data-consumer, or a person who is authorized to provide policy-rules on behalf of one of these two parties. Because of the one-party-consent-model as defined in [8], the rule-author must be a member of either the authoritative-domain of the data-provider or the authoritative-domain of the data-consumer.

The authentication and authorization of the rule-author is performed by the Policy-Decision-Point.

Policy Decision Point:

The Policy-Decision-Point (PDP) is "a logical entity that makes policy decisions for itself or for other network elements that request such decisions" [22].

The PDP is on the one hand used as a policy-compiler that accepts policy-rules, which are supplied by the rule-author in a standardized format via the PDP-interface, syntactically validates them and either sends them via the PEP-interface to the Policy-Enforcement-Point or acts a policy-repository that can be accessed by PEPs using the PEP-interface.

On the other hand the PDP provides services used to authenticate rule-authors and to ensure that the supplied rules are within the scope of the rule-authors authority. For this reason the PDP must be a member of the same administrative-domain as the PEP (see figures 2 and 3).

Policy Enforcement Point:

The Policy-Enforcement-Point (PEP) is "a logical entity that enforces policy decisions" [22].

As defined by the OPES architecture (see chapter 2.2.2), an OPES-data-dispatcher, which is a special OPES entity that resides resides inside an OPES processor, represents a Policy-Enforcement-Point (PEP) that fetches or receives compiled OPES policy-rules via the PEP interface from the PDP and uses these rules to determine the OPES service to perform on the intercepted application-messages.

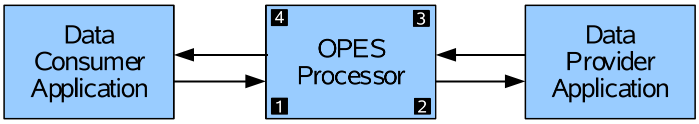


Figure 7: OPES Policy – Processing Execution Points (based on [11])

As defined in [11], every PDP and PEP must support four commonly used processing-points. These processing-points are depicted in figure 7 and listed below:

P1: *Data-Consumer Request handling role*

P2: *OPES-Processor Request handling role*

P3: *Data-Provider Response handling role*

P4: *OPES-Processor Response handling role*

In each processing-point, application-messages are evaluated by the data-dispatcher against the compiled OPES-rules defined for that point.

In P1, request-messages received from the data-consumer are processed. In P2, request-messages are processed before they are forwarded to the data-provider. In P3, response-messages received from the data-provider are processed. And in P4, response-messages are processed before they are returned to the data-consumer.

If a given application-message fulfills the criteria defined by an OPES-rule, the action(s) defined in the matched rule is performed on the application-message. The execution of an action implies the execution of an OPES service-application that possibly modifies the application-message properties. For this reason the data-dispatcher must re-evaluate messages, just after the service-application was terminated, to determine if further actions must be applied and thus further service-applications must be executed.

The services that can be specified as rule-actions are not restricted to service-applications located on the local OPES-processor, but can be even remote callout-services that are hosted on an OPES-callout-servers and are invoked using the OPES-callout-protocol (see chapter 2.2.5).

An OPES-rule may also define a chain of services that must be performed on a given application-message, whereas the involved services are executed sequential in the order in which they are specified in the ruleset.

2.2.4 OPES Service Execution

An OPES system consists of a set of OPES agents that cooperatively provide application-level-services to the application-endpoints. The provided services can be classified into the following two types:

- *Serviced performed on requests*
- *Services performed on responses*

Depending on the respective processing-point (see figure 8) of an OPES data-dispatcher, either a request-modification-service or a response-modification-service can be performed.

Service Activation Points:

As described in chapter 2.2.3, an OPES data-dispatcher (which acts as a PDP) must support four commonly used processing-points.

OPES policy-rules written for the service-activation-points one (P1) and two (P2) can only trigger the execution of services that operate on request-messages, whereas policy-rules for the service-activation-points three (P3) or four (P4) can only trigger services that operate on response-messages.

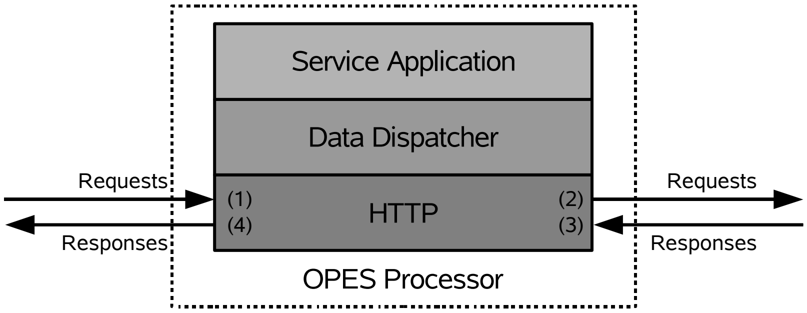


Figure 8: OPES Service Execution – Service Activation Points (based on [16])

Because the OPES-policy-framework [11] does not define a concrete format to formulate policy-rules and does not specify how policy-rules should be evaluated, an OPES-data-dispatcher implementation is not restricted to exclusive evaluation of request-messages in `P1` and `P2` and response-messages in `P3` and `P4`, but could also support the definition of rules that evaluate request-messages in `P3` and `P4`. In this case, it would be possible to define a rule that triggers an action if and only if both the request-message sent by the client and the response-message returned by the remote-server fulfill a given set of criteria.

Additionally to the distinction between services acting on request- and response-messages, services operating on requests can be divided into services modifying request-messages and services generating response-messages.

Figure 9, gives an overview about all possible types of OPES-services:

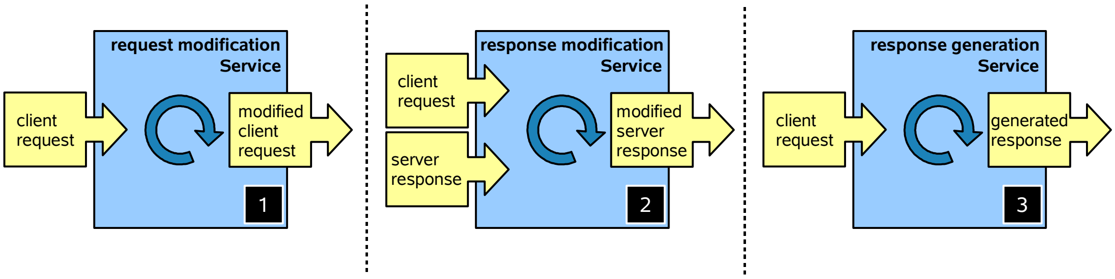


Figure 9: OPES Service Execution – Service Types

Request Modification Services:

A request-modification service is executed or invoked when a request-message is either received (`P1`) or is about to be sent out (`P2`) by an OPES processor, and the message additionally fulfills all criteria defined by an OPES-rule that requires the execution of the service as an action.

Once the OPES-service-application execution is triggered, the service application starts to analyze and process the request-message. Depending on the function of the service, the request-message either remains unmodified or is modified in some way. Alternatively, a request-modification-service can also generate a new response-message as result of the request-message-processing. The (possibly) modified or generated message is then returned to the OPES processor that evaluates the returned message against further OPES rules to determine if additional services need to be executed.

If no further action needs to be triggered, the message is passed to the next service-execution-point[5] or is forwarded to the remote server[6].

In case that the service has generated a response-message, the message is evaluated against policy-rules defined for P4 and, in case that no further actions need to be performed, is sent back to the client.

Examples for service-applications that do not modify request-messages are logging-, or accounting-services. Examples for services that modify request-messages are request-anonymization- or -redirection-services. Examples for response generation services are content-caching- or request-blocking-services.

[5] a message that was evaluated in P1 must also be evaluated in P2

[6] if the message was already evaluated against the policy-rules specified for P2

Response Modification Services:

Much similar to a request-modification-service, a response-modification-service is executed or invoked if a response-message is either received (P3) or is about to be sent out (P4) by an OPES processor, and if the intercepted message (or messages, if both the request-message and the corresponding response-message needs to be evaluated) additionally matches against an OPES-rule that specifies the execution of the service as an action.

Once the OPES-service-application execution is triggered, the service-application starts to analyze and process the response-message (and depending on the implementation also the original request-message) and decides if to modify the response-message or not.
The (un)modified response-message is then returned to the OPES processor that evaluates the returned message against further OPES rules to determine if additional services need to be executed.
If no further action needs to be triggered the message is passed to the next service-execution-point[7] or is forwarded to the client.

Examples for services that do not modify response-messages are logging- or accounting-services. Examples for services that modify response-messages are content-transformation services, such as translation- or advertisement-insertion-services, or e.g. virus-scanning-services that return an error-page instead of the original-response in case a virus was detected.

Service Execution Environments:

OPES service-applications are executed in the service-execution-environments of OPES agents, whereas an OPES agent can be either an OPES processor or an OPES callout-server.

The OPES working group does not define how a service-execution-environment for OPES-service-applications could look like but specifies some service-binding requirements that must be fulfilled by a concrete OPES-processor implementation.

First, it must be possible for a rule-author to specify the service that should be invoked in a location independent manner. Thus the OPES processor needs to be able to do some kind of service lookup to determine whether the specified service can be executed locally or needs to be invoked remotely and which callout-server to use.

Second, the OPES processor should support the maintenance of state information via environment variables. These environment variables should on the one hand be usable by policy-rules, e.g. to define conditions that are only evaluated to true if a given environment variable has a specific value. On the other hand these environment-variables should be accessible by a service-application during service-execution, e.g. to control the runtime-behavior of the service-application via configuration parameters.

Third, the environment-variables should not only be used to control the behavior of a service from outside, but also the other way around, i.e. the service should be allowed to set an environment variable whose value is then used during rule-evaluation, e.g. to determine the group membership of an user. Thus instead of calling the service each time the user sends a request to a protected website, the service is only executed once, authenticates the user and adds his identifier to a special environment-variable that is thereafter used by the rule-engine to determine the group membership of the user.

[7] a message that was evaluated in P3 must also be evaluated in P4

Fourth, the environment-variables should also be used to allow the maintenance of user sessions or to exchange any information between services.

Finally, it must be possible for a rule-author to define action-parameters that are passed as run-time parameters to the OPES-service, once the service execution is triggered.

Chaining of Services:

An OPES processor could either execute an OPES service-application locally or could decide to invoke a remote callout-service. In the later-mentioned case, the service is hosted on call-out-servers, which communicate with an OPES-processor using the OPES callout-protocol (see chapter 2.2.5).

As described in chapter 2.2.3, an OPES ruleset may define a chain of services that need to be performed on a given application-message, whereas it does not matter whether the service is located on the local OPES-processor or hosted on an OPES-callout-server. This type of service-chaining is also referred to as chaining along the content-path.

Beside the chaining of services along the content-path, it's also possible to chain services along the callout-path. In this case, callout-servers are organized in a series where each callout-server executes a different service that operates on the data stream and then forwards the data to the next callout-server in the list.

2.2.5 Callout Protocol

The content-oriented services provided by an OPES service-network are realized by service-applications that are either located on an OPES processor or on an OPES callout-server.

OPES service-applications located on OPES-callout-servers are invoked remotely using a callout-protocol.

Requirements for the Callout Protocol:

The functional-requirements that must be fulfilled by a callout-protocol for the OPES architecture are described in [14] and summarized below.

First, the callout-protocol must be reliable and must ensure to apply congestion-avoidance (according to [23]), e.g. by using a reliable and congestion-controlled transport-layer protocol such as TCP [24].

Second, the callout-protocol must support the establishment of a callout-connection, which is a logical connection between an OPES processor and an OPES callout-server at the application-layer and therefore multiple callout-connections may be multiplexed on a single transport-layer-connection.

Additionally the callout-protocol must allow OPES processors and OPES callout-servers to establish multiple OPES connections concurrently, even to different endpoints. Thus an OPES-processor is allowed to simultaneously invoke callout-services hosted on different callout-servers, and an OPES callout-server is allowed to simultaneously execute services on behalf of different OPES processors.

The callout-protocol must support the negotiation of capabilities and connection-specific parameters, whereas this negotiation process is done on a per-callout-connection base and

therefore different “features” can be negotiated for different callout-connections.
A callout-connection is always established by an OPES-processor and can be terminated by both endpoints.

Third, the callout-protocol must allow the establishment of callout-transactions and the association of them to a callout-connection, whereas a single transaction must not be multiplexed over multiple callout-connections.
A callout-transaction consists of a callout-request and a corresponding callout-response, whereas both can consist of multiple callout-protocol messages containing application-messages or application-message-parts, which need to be exchanged between an OPES-processor and an OPES-callout-server. For instance an OPES-service may receive an application-message or -message-part via a callout-request and may either modify the message(-part) or generate a new message(-part) that is then sent back via a callout-response to the OPES-processor.
A callout-transaction is always initiated by the OPES-processor but can be terminated by both endpoints, whereas a premature termination of transactions must also be supported.

Fourth, the callout-protocol must support asynchronous message exchange, so that the callout-request processing can be done separately from the callout-response generation and transmission. Thus, the callout-protocol must allow multiple outstanding requests and must allow a callout-server to start sending a callout-response back, even if it has not received all callout-protocol-messages that belong to a callout-request.
Additionally the callout-protocol must support message-segmentation. An OPES-processor or an OPES-callout-servers must be able to split a single application-message into multiple fragments in a way that the receiver is able to re-assemble the fragmented message. Furthermore an OPES-processor has to establish a callout-transaction and has to start sending message-fragments before it has received the entire original application-message. And an OPES-callout-server must be able to process and return message-fragments while the OPES-processor is still sending message-fragments to the callout-server.
All these capabilities described above are essential to support services acting on huge application-messages or services that can transform audio- and video streams on the fly.

Fifth, the callout-protocol must support the exchange of metadata between the endpoints. For instance, an OPES-processor must be able to specify an ordered list of services that should be performed on the transmitted application-message.
Additionally a callout-server must be able to indicate that a callout-response is cache-able, and an OPES-processor must be able to indicate that it has kept a local copy of a transmitted application-message(-part).

Sixth, the callout-protocol must provide mechanisms to keep a connection alive and to detect failures of the other endpoint even if no transaction is active. Additionally the protocol should be NAT[8]-friendly so that NAT-devices[9] located on the callout-path do not compromise the communication between the OPES-processor and OPES-callout-server.

Finally, the callout-protocol should be application-protocol-agnostic and should not make assumptions neither on the application-protocol that is used between the application-layer endpoints, nor on the characteristics of the protocol. But the callout-protocol must be at least compatible with HTTP [25].

[8] Network Address Translation (see RFC 1631)

[9] usually router- or gateway-devices located at the network boundary

Which protocol to use as callout protocol:

As described in the previous section, there are various functional-requirements that need to be fulfilled by an OPES callout-protocol.
Although at a first glance the Internet Content Adaptation Protocol (ICAP, [26]) seems to be a suitable candidate for an OPES callout-protocol, evaluations have shown that it does not fulfill all defined requirements:
“ICAP is incomplete with respect to support for multiple application protocols, does not have keep alive messages and it lacks some security requirements.” [27]

Furthermore there were some considerations to use SOAP ([2]) as an OPES callout-protocol as suggested by Rahman et al. in [28], but because of concerns regarding “complexity, bandwidth, delay, size of the transported information” [29] and the ability to handle large opaque data [30], the OPES working group decided to design a new protocol that is capable to fulfill all formulated requirements, especially to handle opaque data efficiently and to support features such as data-preservation and premature-termination.

OPES Callout Protocol (OCP):

The OPES callout-protocol (OCP) is application agnostic, but can be extended by additional application-specific profiles, describing how to encapsulate and transport specific application-protocol messages such as HTTP-request- or -response-messages over OCP.
At the moment the working-group has just defined an OCP profile for HTTP (see chapter 2.2.6), but more profiles, e.g. a profile for SMTP, are planned.

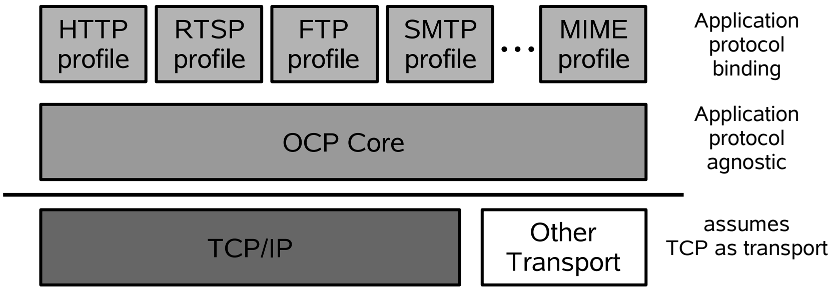


Figure 10: OCP - Building Blocks (based on [31])

The following sections provide a detailed description of the used message-formats and the overall operation of the OCP-Core as defined in [15].

OCP Message Format:

An OCP message is a “basic unit of communication between an OPES processor and a callout server. The message is a sequence of octets formatted according to syntax rules” [15].

The syntax of an OCP-message is defined by the following Augmented Backus-Naur Form (ABNF, [32]):

```
message = name [SP anonym-parameters]
          [CRLF named-parameters CRLF]
          [CRLF payload CRLF]
          ";" CRLF
```

```
anonym-parameters = value *(SP value)              ; space-separated
named-parameters  = named-value *(CRLF named-value) ; CRLF-separated
list-items        = value *("," value)              ; comma-separated

payload = data

named-value = name ":" SP value

value     = structure / list / atom
structure = "{" [anonym-parameters] [CRLF named-parameters CRLF] "}"
list      = "(" [ list-items ] ")"
atom      = bare-value / quoted-value

name = ALPHA *safe-OCTET
bare-value = 1*safe-OCTET
quoted-value = DQUOTE data DQUOTE
data = size ":" *OCTET                  ; exactly size octets

safe-OCTET = ALPHA / DIGIT / "-" / "_"
size = dec-number                       ; 0-2147483647
dec-number = 1*DIGIT                    ; no leading zeros or signs
```

Figure 11: OCP - message syntax [15]

All protocol elements are case sensitive and are using UTF-8 as default encoding.

Each OCP message-type must have an unique name. To avoid conflicts, the names of OCP core-messages are short in size and message defined by protocol-extensions must avoid short names.

Each message can have zero or more anonymous-parameters, zero or more named-parameters and if necessary a payload which contains the encapsulated application-message-data to transfer.
Anonymous-parameters are nameless and are identified by their position in the parameter-list, contrary to named-parameters that have a dedicated name and are not position-dependent.

The declaration of all used OCP-core-message-types and parameter-types is done using PETDM[10], which is a “formal declaration mnemonic for protocol element types” and is used to “extend base types [author's note: atom, list, structure, message constructs] semantics” [15].

The defined OCP-core parameter-types are used as:

- *URI[11]-types* to identify features and services
- *UNI[12]-types* to define unique service-group IDs or OCP-transaction IDs
- *decimal-types* used to define size-, offset- and percent-values
- *boolean-types* used e.g. to signal pending-negotiation-offers
- *structure-types* to report processing-results or to declare supported features of services
- *list-types* used to define lists of features or services.

The defined OCP-core message-types are used to ...

- start and terminate OCP-connections (CS, CE)
- create and destroy service-groups (SGC, SGD)
- start and terminate OCP-transactions (TS, TE)
- negotiate features (NO, NR)

[10] Protocol Element Type Declaration Mnemonic
[11] Unique Resource Identifiers
[12] Unique Numeric Identifiers

- signal start or end of application-message-data transfer (AMS, AME)
- send application-message-data (DUM)
- pause or resume sending of data (DWP, DPM, DWM)
- premature termination of data transfer (DWSR, DWSS, DSS)
- use data-preservation (DUY, DPI)
- exchange progress information (PQ, PA, PR)
- query and report abilities (AQ, AA)

Further message-types or parameter-types may be defined by OCP-core extensions, such as the OCP profile for HTTP (see chapter 2.2.6).

The various types of OCP-core messages are used in four different phases of an OCP callout-connection:

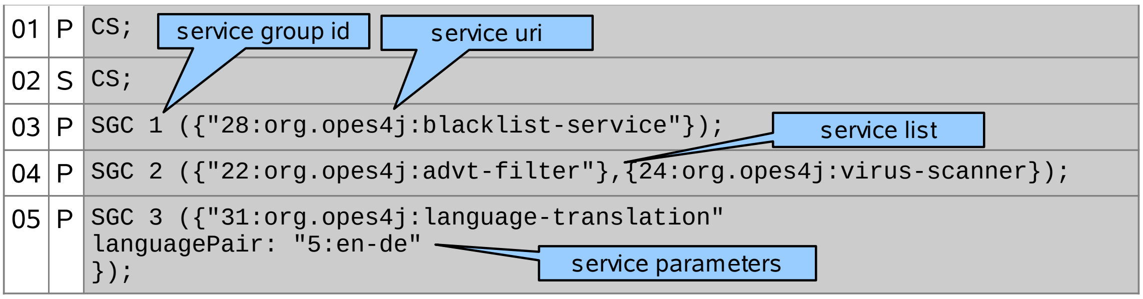
- *Connection Establishment*
- *Negotiation*
- *Data Exchange*
- *Connection Termination*

The different phases are described in the next sections.

Connection Establishment:

In this phase, a transport-layer connection between an OPES-processor (P) and an OPES-callout-server (S) is established and Connection-Start (CS) messages are exchanged between the two OCP-agents to ensure that both ends are talking OCP.

In case that the logical OCP-connection can be established successfully, the OPES-processor can send a Service-Group-Create message (SGC) to create a new service-group.
Each service-group has an unique numeric ID (sg-id) and specifies a list of services that should be executed by the callout-server on all application-message(-part)s received over the given OCP-connection (or transaction; see later). Which services belong to a given service-group is defined using the service-URIs.



01	P	CS;
02	S	CS;
03	P	SGC 1 ({"28:org.opes4j:blacklist-service"});
04	P	SGC 2 ({"22:org.opes4j:advt-filter"},{24:org.opes4j:virus-scanner});
05	P	SGC 3 ({"31:org.opes4j:language-translation" languagePair: "5:en-de" });

Figure 12: OCP – Connection Establishment

The service-parameter used by the SGC message is of type structure (which can consist of multiple anonymous- and named parameters; see figure 11 for details) and therefore can also be used to pass service-related parameters to the callout-server as shown in figure 12, line five. However, the specification does not define how these additional parameters should be interpreted and handled by the callout-server.

In case that the OCP connection can not be established successfully, or if the callout-server doesn't offer some of the services referenced by a service-group, the connection must be closed using a Connection-End (CE) message that indicates the error.

Feature Negotiation Phase:

The negotiation phase is entered the first time immediately after the creation of all needed service-groups has finished, but can be entered multiple times during the lifetime of an OCP connection.

The negotiation-phase is used to negotiate features that should be used either for the current OCP-connection (features with connection scope) or for a given service-group created in the scope of the current connection (features with service-group scope).
Due to the fact that the OCP-core protocol is application-agnostic, two OCP agents at least have to negotiate the OCP profile that should be used to exchange application-message(-part)s between them.

A feature negotiation-phase can be initiated by both endpoints, starting with the first Negotiation-Offer (NO) message that is send, and ending with the first Negotiation-Response (NR) message that is send or received.
During the negotiation-phase, an OPES agent is only allowed to send messages to query or report capabilities[13] or to query or report progress-status information[14]. But in case that an agent needs to send further NO or NR that belong to the same negotiation-phase (e.g. if the negotiation-phase was initiated by an OPES-processor, but the callout-server itself needs to send a NO message to offer additional features), it has to signal this by sending a special "Offer-Pending" parameter with the negotiation-messages.

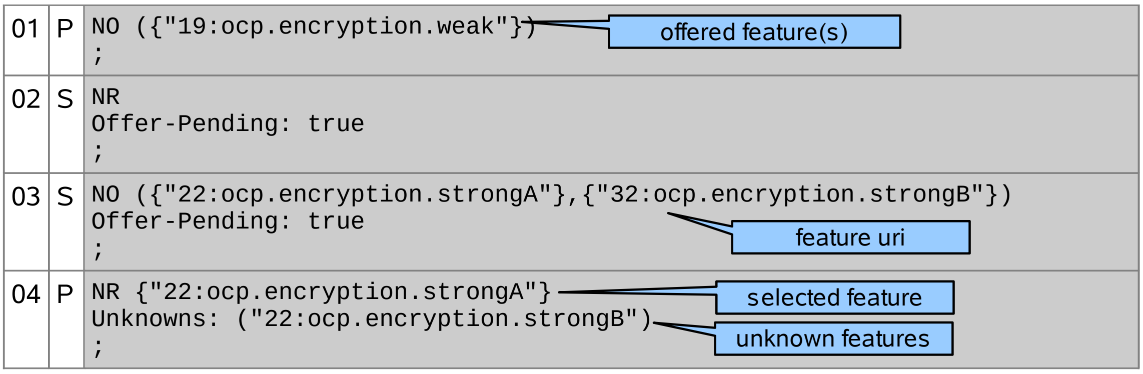
Because an OCP feature can have connection- or service-group-scope, a NO message can optionally include the identifier of the service-group, for which the feature-negotiation should be done. If the service-group-ID is omitted, the negotiated feature has connection-scope.

The receiver of the NO message inspects the message to determine if he is capable to support the offered features, whereas each feature is identified using an unique feature-URI.
Furthermore, the receiver must ensure that the offered features do not conflict with already negotiated features, which are either active for the same service-group or for the whole connection. For example, it should not be possible to negotiate a different application-profile (e.g. HTTP) for a specific service-group in case that another application-profile (e.g. SMTP) with connection-scope was already negotiated.

To complete a negotiation-phase, the receiver of the NO message returns a NR message specifying the feature it has selected[15], which must be one of the offered features.
Unknown- or rejected-features can be reported using special "Rejects" and "Unknowns" parameters.

To completely reject an offer, the selected-feature list must be omitted, otherwise the negotiation has finished successfully and the selected feature becomes active immediately.

[13] Ability-Query- (AQ) and Ability-Answer- (AA) messages
[14] Progress-Query- (PQ), Progress-Answer (PA) or Progress-Report (PR) messages
[15] at most one feature can be selected out of the offered set of features



01	P	`NO ({"19:ocp.encryption.weak"})`
		`;`

02	S	`NR`
		`Offer-Pending: true`
		`;`

03	S	`NO ({"22:ocp.encryption.strongA"},{"32:ocp.encryption.strongB"})`
		`Offer-Pending: true`
		`;`

04	P	`NR {"22:ocp.encryption.strongA"}`
		`Unknowns: ("22:ocp.encryption.strongB")`
		`;`

Figure 13: OCP – Feature Negotiation (based on [15])

In the example shown above, the OPES-processor (P) and the OPES-callout-server (S) are trying to negotiate which encryption to use for the current connection.
The callout-server rejects the weak encryption offered by the OPES-processor and itself offers two strong encryption methods. Encryption method B is unknown to the processor, therefore it selects encryption method A. The selected feature immediately takes affect after the last `NR` message was sent.

Data Exchange Phase:

Once the initial negotiation-phase has finished, and at least the application-profile was negotiated, the OCP connection is ready for use and the OPES-processor can start to invoke callout-services.

The transmission of application-message-data requires the establishment of an OPES transaction, which is associated with a single OCP-connection[16], and represents a sequence of OCP messages related to the processing of usually[17] a single application-message.
The sequence of transaction-related messages (i.e. messages with transaction scope) sent from the OPES processor to the callout-server is called the original-dataflow, whereas the sequence of transaction related messages returned by the callout-server is called adapted-dataflow.

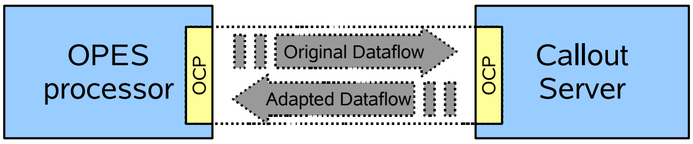


Figure 14: OCP - Original and Adapted Dataflow (based on [15])

An OPES processor indicates the start of a new callout-transaction by sending a Transaction-Start (`TS`) message containing two anonymous-parameters: the first is the transaction-ID (an `UNI` called `xid`) of the transaction, the second specifies the id of the service-group that should be applied to the original-dataflow.
The ID of a transaction is used during the data-exchange phase by other OCP-core-messages to indicate belonging to a specific transaction. This is necessary, because multiple transactions may be active simultaneously on the same OCP-connection.

In a next step, the OPES processor announces the start of the original application-message-

[16] a single transaction must not be multiplexed over multiple callout-connections
[17] a transaction is usually associated with a single original-message and a single adapted-message, but OCP-core extensions may require e.g. to return multiple adapted-messages as result of original-message-processing

dataflow by sending an Application-Message-Start (AMS) message. This AMS message contains just one anonymous-parameter specifying the transaction-ID of the application-message.

Thereafter the OPES processor starts sending the actual application-message data by using Data-Use-Mine (DUM) messages. A DUM-message encapsulates a chunk of application-message-data in its payload and contains two anonymous-parameters: the first specifies the transaction-ID, the second is an offset specifying the position of the encapsulated data-chunk within the original application-message-stream. Thus for each DUM-message this offset value is increased by the chunk-length of the predecessor DUM-message (if any), whereas gaps in the transmitted data are not allowed but the transmission of empty payloads is possible.
Further named-parameters of the DUM-message are used by optional extensions such as the data-preservation feature, which is as described later in this chapter.

The callout-server sends back an AMS-message (with the same transaction-ID as the received AMS-message) to announce the start of the adapted application-message-dataflow and then applies those services to the received data-chunks that were assigned to the current transaction via the service-group-ID parameter of the TS message.
Thereafter the callout-server starts sending the adapted application-message-data - produced by the callout-service(s) - back to the OPES processor using DUM-messages as well. Once more, the DUM-message contains two anonymous-parameters: the first parameter specifies the transaction-ID, the second parameter specifies the offset of the data-chunk within the (in this case) adapted application-message data-stream.

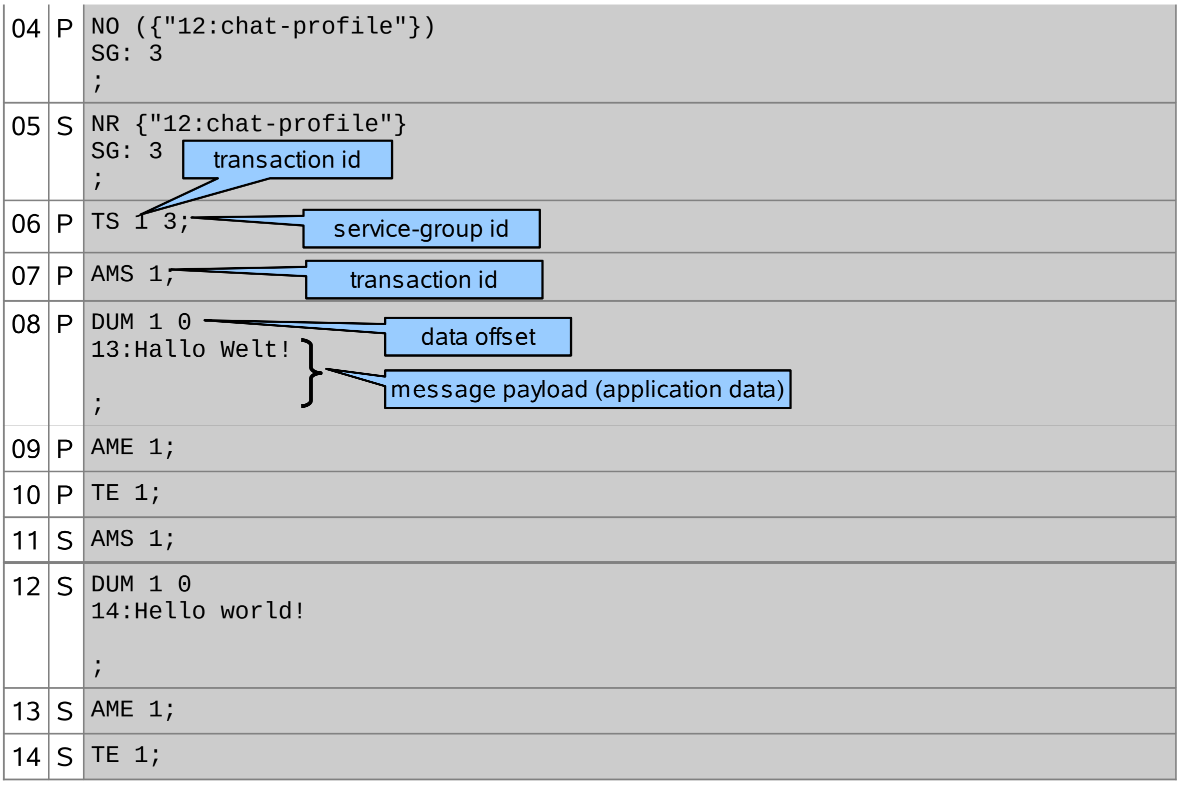
An OCP agent that has finished the transmission of the dataflow (or for other reasons needs to stop processing of application-message-data, e.g. because of an error), immediately[18] indicates this by sending an Application-Message-End (AME) message.
An AME-message contains one mandatory and one optional anonymous parameter. The mandatory parameter specifies the transaction-ID whereas the optional parameter is of type “result” and is used to report the status of the message-transmission.
If the result parameter is omitted, the application-message transmission was successful. Otherwise the result parameter contains an error-code and a textual description of the detected error.
The receiver of an AME containing an error-code must immediately free all status information associated with the erroneous application-message-transmission.

Once an OCP agent has sent all transaction-related messages, it must terminate the transaction by sending a Transaction-End (TE) message. Similar to an AME message, a TE message contains one mandatory and one optional parameter. The first parameter specifies the transaction-ID, the second optional parameter can be used to report the status of the transaction.

01	P	`CS;`
02	S	`CS;`

03	P	`SGC 3 ({"31:org.opes4j:language-translation"`
		`languagePair: "5:de-en"`
		`});`

[18] to avoid unnecessary delays e.g. if an agent relies on the occurrence of this end-of-message event



04	P	NO ({"12:chat-profile"}) SG: 3 ;
05	S	NR {"12:chat-profile"} SG: 3 ;
06	P	TS 1 3;
07	P	AMS 1;
08	P	DUM 1 0 13:Hallo Welt! ;
09	P	AME 1;
10	P	TE 1;
11	S	AMS 1;
12	S	DUM 1 0 14:Hello world! ;
13	S	AME 1;
14	S	TE 1;

Figure 15: OCP – Data Exchange Phase

Figure 15 shows a simple message-flow example using a fictive profile to transfer newline-separated chat-messages over OCP to a translation-callout-service that translates the messages and returns the translated text back to the OPES-processor.

Dataflow Control:

During the data-exchange phase (between the the TS- and the corresponding TE-message) both agents can control the dataflow by using special messages to challenge the sender to pause data transmission and to resume data transmission afterwards.

The Want-Data-Paused (DWP) message can be used by OCP agents to request a pause in sending data. The DWP message contains two anonymous-parameters: The first parameter specifies the transaction-ID, the second is an offset-value that indicates at which offset within the original- or adapted- dataflow, the data-transmission should be paused.

The receiver of the DWP-message confirms the send-pause by returning a Paused-My-Data (DPM) message to the sender and thereafter does not send any further data until it is directed to resume. A DPM-message contains just one anonymous-parameter specifying the transaction-ID of the message.

Once the data-transmission is paused, the OCP agent, which has requested the send-pause, can resume data-transmission by sending a Want-More-Data (DWM) message to the other endpoint. The DWM message contains a mandatory transaction-ID parameter and may also contain an additional anonymous “Size-request” parameter, indicating the minimum chunk-size that should be used by successive DUM-messages. Thus, a DWM message can not only be used to resume data-transfer but also to regulate the data-chunk size to use for data-transfer by using the desribed “Size-request” parameter.

Data Preservation Optimization:

Data-Preservation is one of the optional optimization features provided by OCP. It helps to reduce the round-trip time of the application-message-data by using a cache located on the OPES Processor. This cache stores a single[19] continuous chunk of the original-dataflow, which can be referenced by callout-services using a Data-Use-Yours (`DUY`) message.

A `DUY` message contains three anonymous-parameters: the first one specifies the transaction-ID, the second specifies the offset of the referenced data within the original-dataflow and the third parameter specifies the size of the data-chunk that should be read from the OPES-processor cache and forwarded to the data-provider or consumer. The referenced chunk of data is handled as if it was received from the callout-server via a `DUM` message.

One precondition for the use the data-preservation feature is, that the callout-server knows which data-chunk the OPES-processor has stored in its data-preservation-cache. This information is transported from the processor to the callout-server via the "`Kept`" parameter of a `DUM` message. "`Kept`" is a named-parameter containing two fields: the first field specifies the offset of the first byte of the cached data within the original-dataflow whereas the second field specifies the size of the cached data. For simplicity, segmentation of the cached data is not allowed.

Once the OPES processor has sent the "`Kept`" parameter, it must keep a copy of the offered data-chunk until either the callout-server sends a Data-Preservation-Interest (`DPI`) message to inform the OPES-processor that parts of the preserved data will not be referred anymore, or the transaction is terminated. Nevertheless, an OPES-processor is always allowed to increase the size of the cached data.

In theory, data-preservation is quite powerful but in practice it is not usable because of a syntactical error in the definition of the "`Kept`" parameter (see also chapter 6.1.1).

Premature Dataflow Termination:

A second optional optimization feature offered by OCP is premature dataflow termination. This feature can be used in case that a callout-service only adapts a small portion of the original-content or does not modify the original-content at all, but e.g. logs parts of it. In this case a callout-service does not need to receive the whole original-application-message. But without the possibility to prematurely terminate the transaction, the service must continue sending unmodified original-data back to the OPES-processor, otherwise data will be lost.

If both OCP-endpoints support premature-dataflow-termination, a callout-server can request the *termination of original-dataflow* by sending a Want-Stop-Receiving-Data (`DWSR`) message, which signals that the sender does not want to receive any further data. The processor confirms the termination of the original-dataflow by sending an `AME`-message with a special status-code indicating "partial success". Thereafter the processor continues to receive the adapted-dataflow from the callout-server and forwards the data to the data-provider or data-consumer. I.e. a callout-service has the possibility to continue sending of an adapted-dataflow after the OPES-processor has terminated the original-dataflow.

In case that a callout-service wants to *terminate the adapted-dataflow*, it can request this by sending a Want-Stop-Sending-Data (`DWSS`) message to the OPES processor. If the processor

[19] to avoid the need for a complex cache management, only a single continuous portion of data is stored in the cache

permits and confirms this with a Stop-Sending-Data (DSS) message, the callout-service is allowed to terminate the adapted-dataflow by sending an AME-message with a special status-code indicating “partial success”. Thereafter the OPES-processor just forwards the remaining portion of the original-dataflow directly to the data-provider or -consumer.

Finally if a callout-service wants to completely *get out of the loop* and therefore both, the original-dataflow and the adapted-dataflow should be terminated, it has to terminate the adapted-dataflow first. Thereafter the original-dataflow can be terminated. The order of termination is very important because otherwise the processor would not know at which offset to start forwarding the original-dataflow to the data-provider or -consumer. This is necessary because only the callout-service knowns the exact mapping between the adapted- and the original-dataflow.

Progress Status Information

Further OCP-core messages are defined to support the exchange of progress-information between the OCP endpoints.

The Progress-Query (PQ) message can be sent by an agent (optionally with a transaction-ID) to query the state of its conversational partner. If an agent receives a PQ message, it must respond with a Progress-Answer (PA) message, which (if a transaction-ID was specified) includes the total amount of data that has been sent or received by the agent for the given transaction so far. Thus, PQ can be used to determine the processing-speed of an agent, for debugging purposes or to just keep an idle connection alive.

An agent can also use Progress-Report (PR) messages, to send progress-status-reports unsolicited to the other endpoint, e.g. for logging- or debugging purpose.

Connection Termination

If an OCP-agent decides that a previously created service-group is not needed anymore, it reports this by sending a Service-Group-Destroyed (SGD) message to the other endpoint. The SGD message contains one anonymous-parameter specifying the service-group-ID of the group that should be destroyed. The receiver thereafter frees all state-information associated with the specified service-group.

Finally an OCP connection can be terminated by sending a Connection-End (CE) message to the other endpoint. Much similar to an AME or TE messages, a CE message may contain a “result” parameter that indicates the overall status of the connection. If no error has occurred the result parameter can be omitted, otherwise the parameter contains an error-code and a textual description of the error.

Normally, an OCP-connection is only terminated if it is not needed anymore. But in case of an error, which could not be reported using AME- or TE-messages (e.g. if an erroneous message is not in the scope of the application-message-transmission or transaction), the OCP-connection must be terminated using a CE message including an error-code.

2.2.6 OCP HTTP profile

The Opes-Callout-Protocol (OCP) can be extended with application-specific profiles describing how the callout-protocol can be used to transport specific application-messages between an OPES-processor and an OPES-callout-server.

In [17], Rousskov and Stecher describe an application-specific profile for OCP that defines, how HTTP-messages can be transported using OCP and how HTTP-message adaption can be done using callout-services.
In this chapter, we give a detailed description of this OCP HTTP-profile.

Application Message Parts:

As specified in [25], a HTTP-message consists of multiple well-known parts, which are namely the start-line, zero or more header fields, a separator line, the message body and zero or more trailer fields. In case of HTTP request messages, the start-line is called the request-line whereas for HTTP response messages it is called the status-line. The block of header fields is usually referred to as HTTP message "headers" and the trailer fields block simply called the HTTP message "trailer".

Due to the fact that an OPES callout server in not always interested in receiving all parts of a HTTP message, the OCP HTTP profile [17] defines a set of application message parts to allow OPES processors to split HTTP-messages in their parts, so that these parts can be transfered and interpreted separately.

A detailed list of all defined application message parts is shown in the table below.

Message part name	***Description***
`request-header`	This part consists of the HTTP request line (containing the request method, request URL and HTTP version to use), the request headers and the empty separator line.
`request-body`	This part is the message payload containing application data transferred from the client to the server in case of a HTTP POST or PUT request. Other requests such as GET or HEAD requests do not carry a payload.
`request-trailer`	This part requires the use of the "chunked" transfer encoding and contains the trailer fields of the request.
`response-header`	This part consists of the HTTP status line (containing the status code and text returned by the server and the HTTP version used by the server), the response headers and the empty separator line.
`response-body`	This part represents the message payload containing application data delivered by the server to the client as result of the request-message processing.
`response-trailer`	This part can be used to carry response trailer fields, if the message is sent using the chunked transfer encoding.

Table 1: OCP HTTP Profile - Application Message Parts

As show above, the application-message parts defined by the OCP HTTP-profile correspond with the HTTP-message parts defined in [25]. Thus it's e.g. possible to transfer the header block of a HTTP request to a callout-server whereas the request body is not transferred but could be cached by the OPES processor in an optional data-preservation cache, so that the callout-server could refer to it.

Application Profile Features:

The OCP extension for HTTP defines two separate profiles, one for HTTP-request-message processing and one for HTTP-response-message processing. Both profiles have an unique feature identifier (an URI) and a list of application message parts that belong to the profile, whereas the specification differs between "original" and "adapted" message parts.

Which parts are accessible in which profile is shown in the table below, whereas auxiliary parts are marked with the abbreviation "aux".

	HTTP request profile	***HTTP response profile***

Profile ID	`http://www.iana.org/↵`
	`assignments/opes/ocp/↵`
	`http/request`
		`http://www.iana.org/↵`
		`assignments/opes/ocp/↵`
		`http/response`

Original request parts	`request-header`
	`request-body`
	`request-trailer`
		`request-header (aux)`
		`request-body (aux)`
		`request-trailer (aux)`

Original response parts	-	`response-header`
		`response-body`
		`response-trailer`

Adapted request parts	`request-header,`
	`request-body,`
	`request-trailer`
		-

Adapted[20] response parts	`response-header`
	`response-body`
	`response-trailer`
		`response-header`
		`response-body`
		`response-trailer`

Table 2: OCP HTTP Profile – Profile Parts

As show in table 2, the specification distinguishes between original request-message parts that are auxiliary and therefore are only transferred by the OPES processor if explicitly negotiated in the feature negotiation phase, and mandatory parts that must be transferred if present.

Once all parts that should be sent are negotiated, an OCP agent must send all application-message parts in the order predefined by the corresponding application-profile (see table 2), whereas an OPES processor must not send parts that are not listed as "original" and a callout server must not send parts that are not listed as "adapted". In case of missing parts, e.g. if a request message has no request-body- or -trailer-part, the OCP agent has to skip the absent part.

Which application-message parts are sent back by an OPES callout server, as the result of original message processing, depends on the type of the executed callout-service. Request-modification services can either modify the original request-message and send back adapted request-message-parts, or can generate response-messages and send back response-message-parts.

[20] or generated parts as in case of a request modification service that has the capability to generates response messages

This “short-circuiting” e.g. can be used by blacklist services that prevent an user from accessing a web-site, or by redirection services to redirect the user to a different web-site. In case of response modification services only adapted response-messages-parts can be sent back to the OPES processor.

Feature Negotiation:

The OCP HTTP-profile extends the semantics of various OPES-core-protocol messages and message-parameter-types.

One of the extended parameter-types it the `Feature-`type[21], which is used on the one hand by a Negotiation-Offer (`NO`) message to offer the available features, and on the other hand by a Negotiation-Response (`NR`) message to identify the feature that was selected.
This `Feature-`type is extended by the HTTP-profile with additional named-parameters such as the `Aux-Parts` parameter, which is used to negotiate the auxiliary message-parts that should be delivered by the OPES processor to the OPES callout-server.
More specific, the OPES processor uses this parameter to advertise the availability of auxiliary message-parts and the OPES callout-server responds with a subset of parts it has selected. Once negotiated, the processor must send all negotiated auxiliary parts unless the part is present in the original application message.

Another additional parameter that can be used within the `Feature` parameter is the `Content-Encodings` parameter. This parameter can be used by OCP agents to specify a list of content-encodings that are preferred to over other encodings, whereas encodings that are listed first have higher priority. Due to the fact that this parameter only specifies “preferred” encodings, an OPES service must not be bypassed even if the currently used content-encoding was not listed in the `Content-Encodings` list. If an OPES callout server does not support a specific encoding, it must terminate the transaction.

01	P	`NO ({"54:http://www.iana.org/assignments/opes/ocp/http/response"`
		`Aux-Parts: (request-header,request-body,request-trailer)`
		`})`
		`SG: 1`
		`;`

02	S	`NR {"54:http://www.iana.org/assignments/opes/ocp/http/response"`
		`Aux-Parts: (request-header)`
		`Content-Encodings: (gzip,compress)`
		`}`
		`SG: 1`
		`;`

Figure 16: OCP HTTP Profile – Feature Negotiation

Other extensions to the `Feature` parameters are used to communicate the willingness of an OCP agent to use the OCP data preservation feature. But because of syntactical errors in the definition of the `DUM-`message `Kept-`parameter, these parameters are not operational yet and therefore are not described in detail here (see also chapter 6.1.1).

[21] a structure type that originally just contains the URI of an offered- or selected feature

Application Message Start (AMS) Message:

The OCP Application-Message-Start (AMS) message is extended with a new `AM-EL` parameter. This parameter is optional and can be used by OCP agents to specify the size of the response-body of the HTTP-message, in case that the OCP agent knows the exact length. If the length is unknown, the agent must not send the parameter.
This parameter is introduced because the HTTP profile defines that an OCP agent must not trust the HTTP `Content-Length` header, to avoid problems with callout services that modify the message length but erroneously do not set the content-length properly.

Data Use Mine (DUM) Message:

The OCP Data-Use-Mine (DUM) message is extended with a mandatory `AM-Part` parameter that specifies which application-message-part is transported as payload of the current `DUM`-message. The parameter value is one of the application-message parts listed in table 1.
A single `DUM` message could only contain application-message-data of a single application-message-part, but a single application-message-part can be split among multiple `DUM`-messages with the same `AM-Part` value.

Message Encoding:

As described above, an OCP agent must not be skipped in case that the currently used content-encoding was not listed in the content-encoding preference list. It is assumed that an OCP agent is capable to handle all commonly used content-encoding formats, which are gzip, compress and deflate. If for some reason a given content-encoding format is not supported, the OCP transaction must be terminated by the agent.

Beside the content-encoding, HTTP agents may use transfer-encodings to transport HTTP messages to the next hop. In contrast to content-encodings, the HTTP-profile does not specify an explicit transfer-encoding negotiation and therefore regulates that an agent must not send transfer-encoded message bodies. Therefore, an OCP processor has to remove all applied transfer-encodings and the corresponding transfer-encoding HTTP-header before the application-message is forwarded to the callout-server.

Service Tracing Facility:

In [16] Barbir documents some requirements that need to be fulfilled by OPES-entities to support tracing functionality, as described in [10].

OPES systems must provide tracing information to allow an application-endpoint to detect the presence of OPES-systems in the message-flow. Furthermore the provided tracing information should be sufficient for a system administrator to interpret the content of a trace, to identify the participating OPES entities and to identify the applied adaption-services.
Because the tracing format depends on the application-protocol that is adapted by an OPES service, the corresponding OCP profile has to define the concrete tracing format that should be used.

The OCP HTTP-profile specifies the usage of two additional HTTP-headers to carry tracing information. These headers are the `OPES-System` header and the `OPES-Via` header as shown in the example below.

```
HTTP/1.1 200 OK
Date: Thu, 18 Sep 2003 06:25:24 GMT
Last-Modified: Wed, 17 Sep 2003 18:24:25 GMT
Content-type: application/octet-stream
OPES-System: http://www.cdn.example.com/opes?session=ac79a749f56
OPES-Via: http://www.cdn.example.com/opes?session=ac79a749f56,
   http://www.srvcs-4u.example.com/cat/?sid=123,
   http://www.srvcs-4u.example.com/cat/?sid=124,
   http://www.srvcs-4u.example.com/cat/?sid=125 ; mode=A
```

Figure 17: OCP HTTP Profile – Tracing Headers (based on [17])

An OPES System that is involved in an OPES data-flow must add its tracing entry to the `OPES-System` header and must append an identical entry to the `OPES-Via` header, if the header is already present in the original message.
OCP agents or even OPES services could also append their entries to the `OPES-Via` header.

Each tracing entry consists of at least an unique ID in form of an URI and may also contain additional parameters (e.g. to describe the operation mode of a service).
Further tracing entities can be placed into the message trailer as long that they are not mandatory and the corresponding HTTP-message is transported using chunked transfer-encoding.

System Bypass facility:

Another requirement documented in [16] is the possibility to bypass an OPES System or a single OPES entity.

An OPES system must not prevent a data-consumer from accessing a non-OPES version of a resource. But if no non-OPES version is available without a specific OPES-Service, the bypass-request for that service needs to be ignored. It also depends on the defined system-policy, whether the endpoint is allowed to request a non-OPES version of a content. For instance the end-user may not be allowed to bypass a virus-scanning- or authentication-service.

A bypass request can be seen as an instruction containing identifiers to address those OPES agents and OPES services that should be bypassed. In the previous section we have seen that an OPES system has to add its identifier and an OCP agent or service should append its identifiers to special OPES tracing headers located in the application-message. The identifiers listed in these tracing headers can then be used in combination with a special `OPES-Bypass` HTTP-header defined by the OCP HTTP profile, to address the OPES agents or OPES services that should be bypassed.

```
GET / HTTP/1.0
Host: theHostName
Accept-Encoding: gzip
OPES-Bypass: http://www.srvcs-4u.example.com/cat/?sid=123;
   http://www.srvcs-4u.example.com/cat/?sid=124
```

Figure 18: OCP HTTP Profile – Bypass Header

As shown above, the OPES-Bypass can be added to a HTTP-request- or -response-message to instruct the OPES system to bypass the listed agents or services, whereas it is also possible to bypass all available OPES agents and OPES services by using the asterisk “`*`” character as parameter value.

HTTP Message Adaption Example:

The following figure shows an example for an OCP-message-flow between an OPES-processor (P) and an OPES-callout-server (S). The callout-service that is applied to the application-messages is a blacklist-service that blocks HTTP-requests to given URLs and generates an access-denied response-message (HTTP-status code 403) that is sent back to the originator of the request.

01	P	CS;
02	S	CS;
03	P	SGC 1 ({"22:org.opes4j:BlacklistService"});
04	P	NO ({"53:http://www.iana.org/assignments/opes/ocp/http/request"}) SG: 1 ;
05	S	NR {"53:http://www.iana.org/assignments/opes/ocp/http/request"} SG: 1 ;
06	P	TS 1 1;
07	P	AMS 1 AM-EL: 0 ;
08	P	DUM 1 0 AM-Part: request-header 56:GET http://www.test.at/ HTTP/1.1 Host: www.test.at ;
09	P	AME 1;
10	P	TE 1;
11	S	AMS 1 AM_EL: 39 ;
12	S	DUM 1 0 AM-Part: response-header 108:HTTP/1.1 403 Denied Date: Thu, 25 Jan 2007 13:07:31 GMT Content-Length: 39 Content-Type: text/plain ;
13		DUM 1 80 AM-Part: response-body 39:You are not allowed to access this URL. ;
14	S	TE 1;

Figure 19: OCP HTTP Profile – Message Flow Example

The conversation between the OCP processor and callout-server starts with establishing a new OCP connection (lines 01-02).

After the connection is established, the OCP processor creates a new service-group by specifying the identifier of the callout-services that should be bundled together to a service-group (line 03). This group specifies those services that should be performed on consecutively received original-application-messages.

Next the negotiation-phase is entered and the OCP features that should be used, are negotiated (lines 04-05). Once the OCP processor and callout-server have clarified that the HTTP-request profile should be used, the data-transfer-phase is started (lines 06-14).

In the data-transfer-phase the OCP processor transmits the original-request-message-parts to the callout-server. Because our example HTTP-request is a GET-request, no request-body is available and therefore only a single `DUM`-message, which encapsulates the `request-header` part, is transferred.

Next the callout-service inspects the received HTTP-header-part and generates a HTTP-response-message, whereas each response-message-part (the `response-header` and `response-body`) must be transmitted in a separate `DUM`-message.
Finally the OCP transaction is terminated by the callout-server (line 14) to indicate that no more transaction-related-messages will be sent.

2.3 Proxylet

The term “proxylet” is used in various contexts in the literature.

SUN MICROSYSTEMS uses the term “proxylet” in a description of the Sun Java System Portal Server and describes a proxylet as “a Java applet that sets itself as a proxy server on the client machine” by modifying “the proxy settings in the Proxy Auto Config (PAC) file on the client machine” [33].

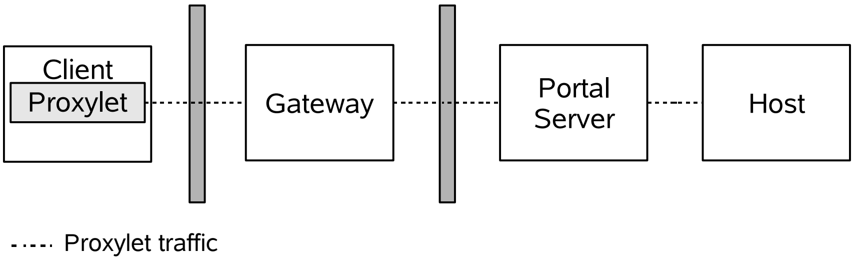


Figure 20: Sun Portal Server – Proxylet [34]

This kind of proxylets is not used to modify content, but to redirect requests via a secure-channel to a special gateway node, if the domain or subdomain of the request URL is listed in the gateway profile. Otherwise the request is directed to the Internet.

Other projects such as the ALAN[22] system [35] are using the term “proxylet” in the context of active networks.
In [35] and [36] FRY, GHOSH and MACLARTY are introducing proxylets as control-modules implementing an interface much similar to an applet running in a web-browser.
A proxylet consists of a collection of class files bundled together in a single Java jar file, can be identified using an URL reference, downloaded from a proxylet server onto a dynamic-proxy-server (DPS), and passed to the execution-environment for proxylets (EEP) of the DPS.

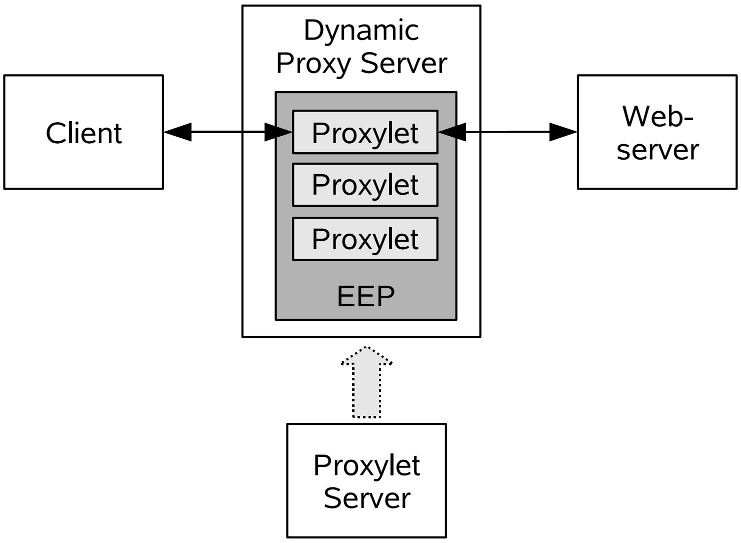


Figure 21: ALAN – Proxylet (based on [35],[36])

The EEP acts as a proxy-device and uses a Java virtual-machine to execute proxylets that perform value-added services on application-layer messages.

[22] Application Layer Active Networking

The example services introduced by the ALAN project are audio transcoding, content compression or application level routing services.

A quite similar approach can be found in the area of Open Pluggable Edge Services (see chapter 2.2).

Although the OPES Working-Group was mainly chartered to specify an architectural framework for edge services (see chapter 2.2.2) and to define a callout-protocol for the remote execution of callout-services (see chapter 2.2.5), there are some predecessor documents pointing out how a service-execution-environment for edge-services could be realized using “proxylets” (e.g. [37], [5], [38], [39]).

In these documents, a proxylet is described as “a piece of code that runs on the (local) OPES device [an OPES processor or OPES callout-server] and provides a service on the transit requests or responses.” [39]. The proxylet can be either pre-installed on the OPES-device or can be downloaded from an OPES-admin-server. The execution of the proxylet is triggered if all conditions of a rule are met. A rule consists of various patterns that are evaluated by a rule processor against application-message properties and will either cause the rule to match or fail.

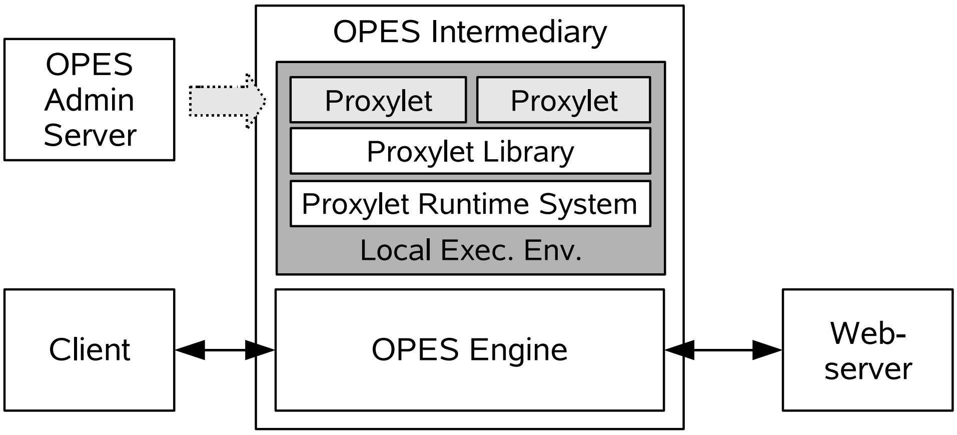


Figure 22: OPES – Proxylets (based on [5])

To avoid writing separate rule parsers and proxylets for each type of intermediary that is used as OPES device (e.g. Caching proxies, gateways, application level router, ...) or even for each OPES-device vendor, Yang and Hofmann recommended in [39] to use standard formats for the definition of rules and a standard API that can be used by proxylet-developers to implement device-independent proxylets.

Yang and Hofmann suggested the use of the Intermediary Rule Markup Language (IRML) as language for rule specification. Additionally in [19] Walker introduced a *Proxylet Local Execution Environment Java Binding* defining a set of Java-interfaces allowing a proxylet-service to manipulate the application-message-protocol-headers and to access and modify the message-payload using Java `Input/OutputStreams`.

The following section is focused on this proxylet-execution-environment and describes the proposed interfaces and architecture details.

2.3.1 Proxylet Execution Environment Java Binding

As suggested by Yang and Hoffman in [39], vendors of proxylet-execution-environments should provide a standardized API to allow proxylet-developers to implement OPES-device - and vendor-independent proxylets. In [19], Walker has specified a set of Java interfaces defining such an API.

This chapter contains a detailed description of these interfaces and the supposed proxylet-execution-environment architecture.

Proxylets:

As defined by Walker: “Proxylets are small pieces of code that execute on an intermediary at the request of an authorising party.” [19]

The authorizing party may be either the content-provider or the end-user. Therefore a proxylet must not be executed on behalf of an Internet service-provider or a transparent-proxy.

Walker recommends to use the Intermediary Rule Markup Language (IRML) [20] to define configuration rules that explicitly authorize a service to adapt the content flowing from the data consumer to the data provider or vice versa.

Proxylet Engine:

Because proxylets are service-applications that can be downloaded and deployed dynamically at runtime, each proxylet-execution-environment must have a dedicated Proxylet-Engine that is responsible for the management and control of proxylets during their runtime life-cycle. Moreover, the proxylet-engine is used to encapsulate proxylets in secure sandboxes to protect the underlying server against malicious code and misoperation.

The interaction between an encapsulated proxylet and its environment is done via well defined interfaces, specifying the functions a proxylet can use to alter request- and response-messages, to access the proxylet-configuration-data, to manage user-related data using sessions, and to communicate with the underlying proxylet-engine.

Furthermore the proxylet-engine provides mechanisms to notify a proxylet about events that occur outside the proxylet-sandbox, during the execution of the proxylet. These events vary from events caused by the disconnection of clients and servers to events related to the underlying server such as system-shutdown events.

Proxylet Life-Cycle:

As mentioned above, the life-cycle of a proxylet is controlled by the proxylet engine.

A proxylet life-cycle can be divided into the following stages:

- Instantiation
- Initialization
- Service Execution phase(s)
- Destruction
- Undeployment

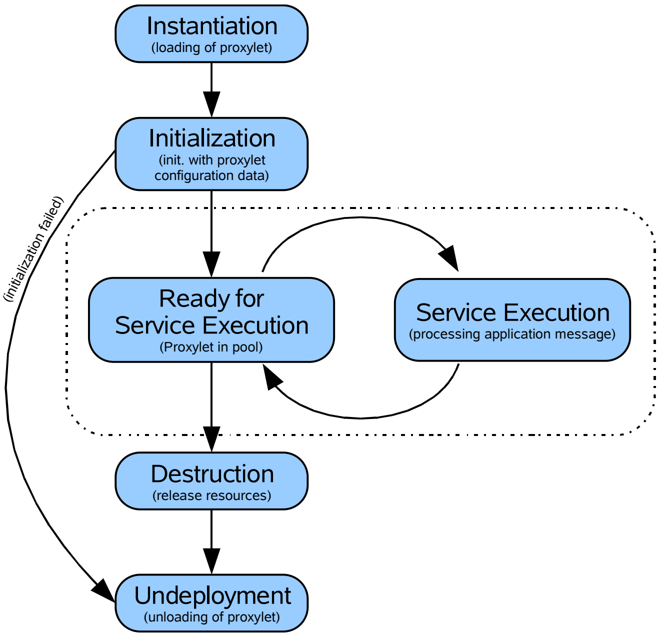


Figure 23: Proxylet - Life-Cycle

In the following section a detailed description of the various proxylet life-cycle stages is given.

Proxylet Instantiation:

In this stage, the proxylet is deployed by the proxylet-engine. To do this, the proxylet-engine needs to have sufficient information such as the location of the proxylet-code or the name of the proxylet Java-classes to use, so that the proxylet-code can be loaded into the proxylet-engine. [19] does not define how such a "deployment descriptor" could look like, but refers to [40], which contains a description of an XML markup language called OMML[23] that can be used for that purpose.

After the proxylet Java-class is loaded, the engine takes a look onto the interface(s) implemented by the proxylet-class to decide, how many instances of the proxylet are needed at runtime. If the proxylet class implements the `SingleThreadedProxylet` interface, then only one thread is allowed to pass through a single proxylet instance at a time and therefore multiple instances need to be loaded. Otherwise a single instance is sufficient to handle all requests related to the specific proxylet.

Proxylet Initialization:

After the instantiation of the proxylet has finished, the initialization stage is entered.

In this stage the proxylet-configuration-data, which is usually provided by the deployment-descriptor and contains all configuration-data needed by the proxylet to run, is loaded and passed to the instantiated proxylet-instance in form of a Java `ProxyletConfig` object. This class provides various methods to loop through the list of available initialization-parameters and to fetch the value of each parameter.
Furthermore this class provides access to the `ProxyletContext`, which is a Java object,

[23] Opes Meta Data Markup Language [40]

that can be used by the proxylet to communicate with the underlying proxylet-engine. This context object provides access to proxylet-engine version information, vendor-specific server-attributes and allows the proxylet to log messages into the server-log or to register itself as listener for server-specific events (e.g. server shutdown events).

Proxylet Service Execution:

Once a new service call was received, the proxylet-engine either fetches an already initialized proxylet-instance from a pool of initialized and ready-to-use proxylets, or has to instantiate and initialize a new proxylet-instance. The fetched or newly instantiated proxylet then enters the service-execution stage via a call to its request- or response modification function.

The request- and response-modification functions of a proxylet are called repeatedly during the life-cycle of a proxylet-instance, until the proxylet-instance is not needed anymore and therefore is destructed.

During the service-execution, the proxylet-instance has access to the received application-message via special message-objects providing functions to access the received application-message-data.

Application-message meta-data can be read and modified directly by using special functions provided by the message object, whereas the application-message-payload can be read and written using dedicated Java `Input`- and `OutputStream`s as depicted in the figure below.

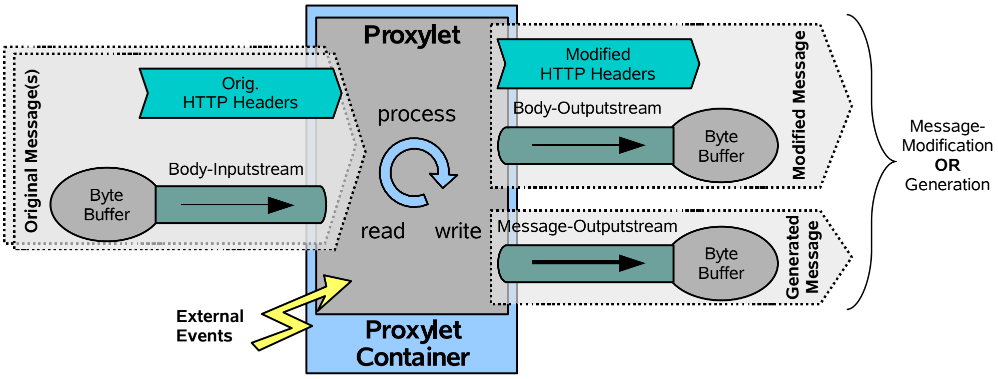


Figure 24: Proxylet – Service Execution

As shown in figure 24, a proxylet that is applied to an original-application-message reads and inspects the received application-message-data and then either writes out a modified version of the received message or even generates a new application-message.

More specific, a request-modification-proxylet takes the original request-message as input-parameter and either generates a modified request-message or even generates a new response-message as result of the request-message processing.

A response-modification-proxylet takes the original response-message as input-parameter and processes it to generate a modified response-message.

Request Modification Proxylet:

If a proxylet is used to act as a request-modification service, its `modRequest` function is called and the two Java objects, the `ProxyletRequest` and the `ProxyletSession`, are passed to the proxylet as input-parameters.

ProxyletRequest is a Java object, represents a concrete request-message and provides various functions, allowing the proxylet to access application-message meta-data and payload.

For each application-protocol supported by the OPES device, the proxylet-engine has to provide separate Java-classes, which are inherited from ProxyletRequest and provide protocol-specific methods to access and manipulate the application-message-data.

For the processing of HTTP-request messages, WALKER defines the HTTPProxyletRequest class, which provides sufficient methods to read and modify the HTTP-request-line and to get, set or remove HTTP-headers, whereas special methods for the processing of HTTP cookie are available. The request-message-body can be read using an InputStream, whereas the modified version of the request-message-body can be written out using an OutputStream.

For proxylets, that need to generate a new response-message as a result of a request-message processing (e.g. for content-filtering- or blacklist-services), the HTTPProxyletRequest object provides an additional OutputStream, allowing proxylets to write out the newly generated response message as a byte-stream.

The ProxyletSession object allows proxylets to maintain sessions. These sessions can be used to store data (in the form of ProxyletSessionData objects) related to either the client or the server. The data stored in the session can then be accessed during the consecutive processing of application-messages received from the same client or server.

At the end of the request-message-processing, the request-modification function returns a ProxyletStatus object as output parameters, which reports the overall message-processing status, indicated by status/error codes and corresponding status texts.

Depending on this status returned by the proxylet, the underlying proxylet-engine decides if the generated- or modified application-message should be delivered to the remote-server (in case of a request-modification), to the client (in case of a response-generation), or if an error-message must be sent back to the client.

If the proxylet returns without modifying the original request-message and without generating a response-message, and if no error was reported, the original request-message is forwarded to the remote-server.

Response Modification Service:

If a proxylet should act as a response-modification service, then its modResponse function is called and three parameters are passed to the proxylet.
The first parameter is the ProxyletRequest object representing the original client-request-message that was already delivered to the remote server. The second parameter is the ProxyletResponse object, which represents the response-message that was returned by the remote server. The third parameter is a ProxyletSession object used for session management.

The ProxyletRequest object was already described in the context of a request-modification-service. But there are some differences if using this object in the context of a response-modification-service: It is neither possible to write meta-data or to modify the message-body, nor to use the request-object to generate a response-message. On the one hand, it makes no sense to modify a request message that was already sent to the remote server. On the other hand the proxylet is not allowed to generate a response message but has to use the response-object to modify the original response.

The ProxyletResponse object represents an original response-message and provides various functions to access the application-metadata and payload. As already described for request-messages, an OPES device that supports multiple protocols has to provide a separate Java-class for each supported protocol. For the processing of HTTP-response messages a special HTTPProxyletResponse object is provided, that offers functions to manipulate HTTP-headers and allows the proxylet to read and write the message body using dedicated Java Input- and OutputStreams.

The ProxyletSession is used in the same way as described in the context of a request-modification service and provides functions for user- or server-related session management.

At the end of the response modification, the function returns a ProxyletStatus object indicating the overall message-processing status. The proxylet-engine then uses the status-code to decide whether the modified response-message or an error-message should to be sent to the end-user.

If the proxylet returns without modifying the original response-message and if no error was reported, the original response-message is delivered to the client.

Proxy(let) Events:

As mentioned above, some well defined events can occur outside a proxylet-instance during service-execution. If a proxylet-instance needs to get informed about the occurrence of one of these asynchronous external events, it needs to register itself as an event-listener.

If a proxylet needs to listen for global events[24], it has to implement the ProxyEvent-Listener interface, whereas proxylets listening for proxylet-related events[25] have to implement the ProxyletEventListener interface.

The (un)registration as an event-listener for global-events can be done using special functions provided by the ProxyletContext object, whereas the (un)registration for proxylet-related events can be done using functions provided by the ProxyletRequest object.

Once the proxylet has registered itself as event-listener, the proxylet-engine informs the proxylet about the occurrence of new events using one of the event-listener functions that are defined by the listener-interface and must be implemented by each proxylet. For instance, there are dedicated functions to report the shutdown of the proxylet engine or the disconnection of clients or servers. Events that are not pre-defined in [19] can be reported using the proxyEvent function, which is a general-purpose method accepting objects of the type Event. This Event-class provides functions to determine the request- or response-message the event is associated with, and can be used by proxylet-engines to inherit custom event classes from it.

At the end of the service-execution phase, the proxylet uses the unregister function provided by the ProxyletContext or ProxyletRequest object to remove itself from the event-listener queue.

[24] events that are not related to a specific proxylet-instance, e.g. a server-shutdown event

[25] e.g. the disconnection of the client or server

Proxylet Chains:

As described above, once a new original request- or response-message is received, the proxylet-engine instantiates and initializes a new proxylet-instance and passes the received message to it. Once the service processing has finished, the proxylet-instance returns a status-code indicating if the service execution was successful and therefore the modified (or generated) message can be send to the ultimate receiver.

Additionally to this common scenario, its also possible to chain multiple proxylets together. In this case, the modified or generated message returned by a proxylet-instance is not sent back to the ultimate-receiver, but is passed to the next proxylet in the chain and so forth. Thus, with chaining it is possible combine simple proxylets together, so that they provide a new, more complex service.

Once a message has successfully passed the last proxylet in the chain, it is sent back to the ultimate-receiver. But if one of the proxylets reports an error, the processing is aborted and an error message is sent back.

Proxylet Destruction:

After the proxylet has finished service execution, it is stored by the proxylet engine into an internal pool, so that it could be re-used to process further requests.

If a proxylet-instance is not needed anymore, e.g. if the number of pooled proxylets should be reduced or if a server-shutdown is in progress, the proxylet enters the destruction-phase. In this phase, the proxylet-engine calls the `destroy` function of the proxylet. This allows the proxylet-instance to free resources that were used during service-execution, e.g. opened files or connections to a databases. Thereafter the proxylet will be finalized and garbage collected.

Proxylet Undeployment:

A proxylet is handled as un-deployed, if all its instances have been destructed. The reason for this could be because of a server-shutdown or if e.g. the administrator for some reason decides that a specific service should not be provided anymore.

Session Management:

If the proxylet-engine calls the modify-request- or -response function of a proxylet-instance, it passes a `ProxyletSession` object as input-parameter to the function.

The `ProxyletSession` object can be used by the proxylet to manage sessions. For this purpose it provides functions to read objects from the session using the objects-name or to store objects in the session, whereas for each object its name, value and storage-state must be specified. With the storage state, the proxylet can declare whether the data should be only accessible by the proxylet itself, or if it can also be accessed by other proxylets. Furthermore an object can be flagged as persistent, which signals that the object should be transferred to a persistent storage area.

[19] does not define how sessions should be maintained by a proxylet, but suggests to use cookies or URL-rewriting to maintain the session-state at the intermediary.

2.4 Intermediary Rule Markup Language (IRML)

The Intermediary Rule Markup Language (IRML) was proposed by Beck and Hofman in [20] and is an XML-markup-language, which can be used to express policy-rules needed by intermediaries to trigger the execution of edge-services according to the conditions and actions defined in the policy-rules.

Because intermediary-services should only be executed on behalf of either the data-provider or data-consumer, each IRML-rule must be authorized by one of the endpoints. Thus, IRML differentiates between rule-authors and authorizing parties.

The root element of a rule-file is the `rulemodule` that contains information about the `author` of the rule-module and a single `ruleset` element containing the actual IRML-rules. The overall structure of the `rulemodule` is depicted in figure 25.

```
<rulemodule xmlns="http://www.rfc-editor.org/rfc/rfcxxxx.txt">
  <author type="self">
     <name>Martin Thelian</name>
     <contact>xxx@yyy.zz</contact>
     <id>XXXX</id>
  </author>
  <ruleset>
     <!-- rules -->
  </ruleset>
</rulemodule>
```

Figure 25: IRML - Rule Module

The rule author can be either the data-provider or -consumer itself (type is `self`) or a delegated person (type is `delegate`) whereas an author is uniquely identified with his `id`.

A `ruleset` element must contain an `authorized-by` element containing information about the authorizing party, a `protocol` element specifying the application-protocol for which the `ruleset` was specified, and one or more `rule` elements. An example is shown in the figure below.

```
<ruleset>
  <authorized-by class="data-consumer" type="individual">
     <name>Martin Thelian</name>
     <contact>xxx@yyy.zz</contact>
     <id>XXXX</id>
  </authorized-by>
  <protocol>HTTP</protocol>

  <!-- rules -->
</ruleset>
```

Figure 26: IRML - Rule Set

The authorizing party can be the `data-provider` or `data-consumer` and furthermore can be either of type `individual` or a `group` of individuals. The `group` type is especially useful in cases where a group of providers or consumers has identical rules.

The application-level-protocol, to which the rules of the ruleset should be applied to, is specified by the `protocol` element, which is typically HTTP but is not restricted to it.

An example for an IRML-rule is shown in figure 27. The depicted rule triggers a service, used to load a given web-page from the Google cache, if the original server reports a HTTP error by returning the 404 error code.

```
<rule processing-point="3">
   <property name="response-code" context="res-msg" matches="404">
      <property name="request-method" context="req-msg" matches="GET">
         <property name="request-path" context="req-msg"
            matches=".*(/|\.html)$">
            <execute>
               <service name="404 Cache view service">
                  <uri>opes://pinky:8071/org.opes4j:ViewCacheService</uri>
                  <parameter name="requestURL" type="dynamic">
                     <variable name="request-uri" context="req-msg"/>
                  </parameter>
                  <parameter name="googleKey" type="static">
                     <value>12345</value>
                  </parameter>
               </service>
            </execute>
         </property>
      </property>
   </property>
</rule>
```

Figure 27: IRML - Rule

Each `rule` has a dedicated rule `processing-point` for which it is provided. The processing-point can be one of the four common processing-points supported by a PDP[26] or PEP[27], as described in chapter 2.2.3.
In the example shown in figure 27, the IRML-rule is only evaluated for response-messages received from the data-provider (`P3`).

The conditions of an IRML-rule are specified using `property` elements. Conjunctions are expressed by nested `property` elements whereas disjunctions are expressed by sequences of `property` elements.

Each `property` element has a `name`, a `context` and a `matches` attribute. The `name` refers to a property within a given context, whose value should be evaluated. The `matches` attribute specifies a regular-expression-pattern that is matched by the rule-engine against the given property-value to determine if the condition can be evaluated to true.

The `context` attribute of a property-element can have one of the following values: `req-msg`, `res-msg`, `system` or `service`. The values `req-msg` and `res-msg` can be used to refer to a request- or response-message-property[28], whereas intermediary-system- or service-application-properties are referenced using the values `system` or `service`.
Which one of the two message-property contexts can be used by a rule depends on the processing-point of the rule. Rules defined for processing-point one or two can only reference request-message-properties, whereas rules active for processing-point three or four can reference both, request- and response-message-properties.
Which property-names can be used by a rule depends on the application-layer-protocol for which the ruleset is provided, and is specified in a separate IRML extension provided for each protocol supported by a rule-engine.

If the rule-engine evaluates an IRML-rule to true, in case all conditions are met, the action specified by the rule needs to be executed. In IRML an action is defined using `execute` elements, which contain one or more `service` elements.

[26] Policy Decision Point
[27] Policy Enforcement Point
[28] e.g. the headers or the request- or response-line, in case of HTTP

Each service element must include an uri element containing an absolute URI, which uniquely identifies the service-application that should be executed. The identifier must be mapped at runtime by the intermediary device to a specific service-instance, which can be either located on the intermediary or on a remote callout server.
Additionally to the uri element, the service element may contain one or more parameter elements specifying service-parameters that should be passed to the service-application when triggering the service.
A parameter can by either defined as static or dynamic. Static parameters have a constant-value, whereas the value of dynamic parameter is evaluated at runtime by querying a given property, specified by its name and context (similar to the reference of properties by the property elements).

IRML also supports unconditional rules. For this, a rule element just contains an execute element without any surrounding property element.

As described above, a property- or variable-element uses name- and context-attributes to reference specific message-, system- or service-properties. Additionally to this two attributes, an optional sub-system attribute can be used in combination with a context value set to "system" to support the evaluation of properties defined by optional IRML-extensions.

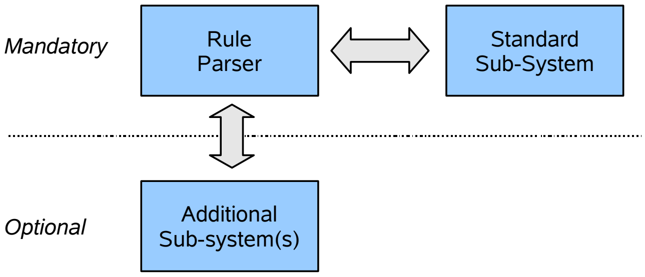


Figure 28: IRML – Subsystems (based on [21])

The default value for the sub-system attribute is "standard". This standard-subsystem only defines two properties: the system-date and the client-ip.
Further properties may be provided by optional subsystems defined in IRML-extensions [21].

An optional subsystems can introduce new properties that are evaluated by the subsystem-implementation at runtime. Thus subsystems are free to re-define the method that is used for property-evaluation. For instance, a subsystem may decide to evaluate conditions using arithmetic operations instead of regular-expressions used by the standard subsystem.

Additional subsystems are always optional. This means that a rule-engine should treat a condition as not matched, if it does not support the specified subsystem.

```
<rule processing-point="4">
  <property name="allocated-bandwidth" matches="low" sub-system="QoS">
    <execute>
      <service>
        <uri>callout.service.transform</uri>
        <parameter name="mode" type="dynamic">
          <variable name="allocated-bandwidth" context="QoS"/>
        </parameter>
      </service>
    </execute>
  </property>
</rule>
```

Figure 29: IRML – QoS Subsystem

Currently there exists an IRML-subsystem for Quality of Service [41], which can be used to define rules that evaluate QoS parameters, e.g. to trigger the transformation of a website if a client with a very limited bandwidth tries to access a specific web-site. An example for a rule using the QoS subsystem is depicted in figure 29.

3 Web Services

The Web Services Architecture Working Group at the W3C[1] defines a Web Service as follows: “A Web service is a software system identified by a URI [RFC 2396], whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.” [42]

I.e. one of the main benefits of the Web Service Architecture (WSA) is the decoupling of service-interfaces from the concrete service-implementation. Due to the fact that XML is used for service-interface-definition and -publishing, as well as for service invocation, the WSA is an “interoperability architecture”[43], which ensures high cross-language and cross-platform interoperability. Thus, two systems neither need to be run on the same platform nor need to be written in the same programming language, to be able to cooperate.

Unlike Content-Delivery- or Edge-Service-Networks as described in chapter 2, which are content-oriented, the Web-Service Architecture (WSA) is service-oriented.

There are three standards, based on XML, that are commonly used to define, discover, and invoke Web Services [43]:

- **SOAP[2]:** used to exchange messages between a Web-Service provider and a Web-Service requester.
- **Web Service Description Language (WSDL):** used to describe the interface of a Web Service.
- **Universal Description Discovery and Integration (UDDI):** used to publish and find Web Service descriptions

The remaining part of this chapter is focused on SOAP.

3.1 SOAP

SOAP [2] is a lightweight protocol to exchange XML-based messages between a Web-Service provider and a Web-Service requester, whereas a message may pass various intermediary devices, located along the message-path, until it reaches the ultimate receiver.

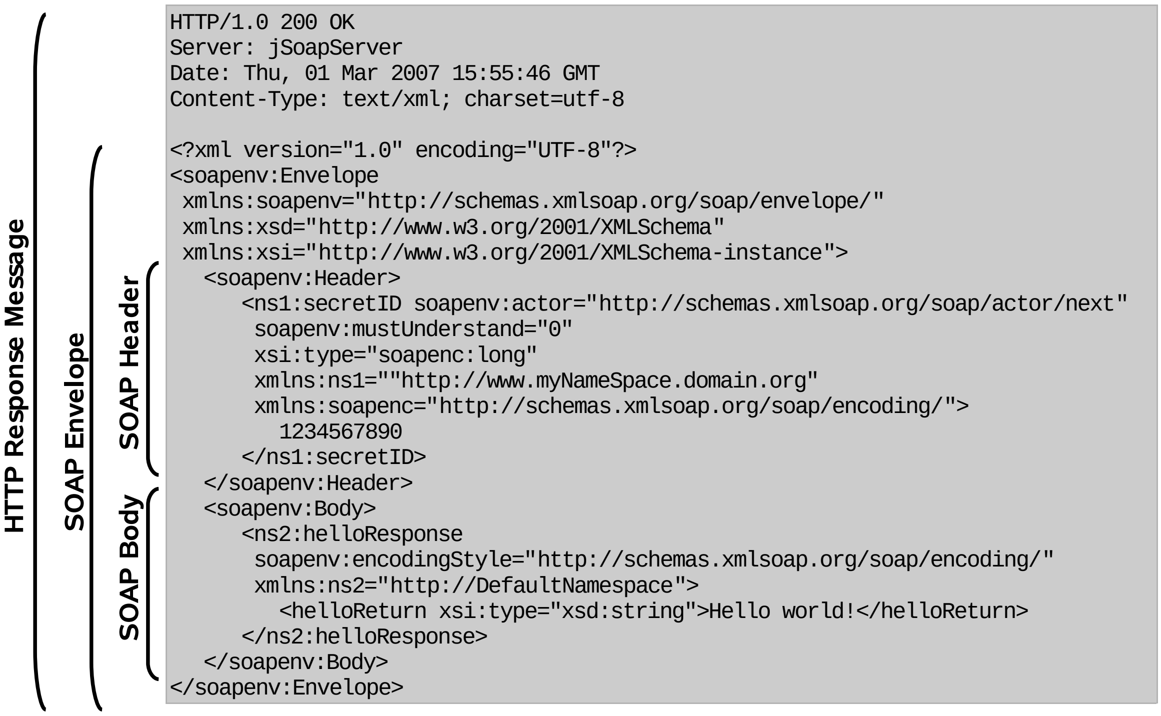
SOAP itself is not a transport protocol, but uses various underlying protocols for message transmission. How a concrete underlying protocol can be used to exchange SOAP messages between SOAP-nodes is defined in a protocol-binding specification. The most commonly used protocol binding is the SOAP HTTP Binding, but other binding specifications exist, e.g. a binding for SMTP.

Figure 30 shows an example of a SOAP-message that is transported using HTTP. The transported HTTP-body has the MIME type `text/xml`[3], and contains the SOAP-envelope, which is the outermost element of a SOAP-message. The envelope contains an optional SOAP-header, which in turn contains a collection of SOAP-header-blocks and a mandatory SOAP-body element.

[1] World Wide Web Consortium (W3C)

[2] formerly known as Simple Object Access Protocol, but as of SOAP version 1.2 the acronym is not spelled out anymore

[3] resp. `application/soap+xml` for SOAP version 1.2



```
HTTP/1.0 200 OK
Server: jSoapServer
Date: Thu, 01 Mar 2007 15:55:46 GMT
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Header>
      <ns1:secretID soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
       soapenv:mustUnderstand="0"
       xsi:type="soapenc:long"
       xmlns:ns1=""http://www.myNameSpace.domain.org"
       xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
         1234567890
      </ns1:secretID>
   </soapenv:Header>
   <soapenv:Body>
      <ns2:helloResponse
       soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:ns2="http://DefaultNamespace">
         <helloReturn xsi:type="xsd:string">Hello world!</helloReturn>
      </ns2:helloResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 30: SOAP – HTTP Binding

The body of a SOAP-envelope is always targeted at the ultimate receiver, whereas a header-block may be targeted at any intermediary node located on the message-path.

Which node is the target of a header-block is specified using the `role`[4] attribute. Currently the specification defines three roles: “none”, “next” and “ultimateReceiver” [44]. But using application-specific roles is also possible. The “none” role means that no intermediary node should process the message, whereas “next” references the next intermediary node on the path[5].

Another important header attribute is the `mustUnderstand` attribute. If a node receives a SOAP-message containing a header-block targeted at itself and with a `mustUnderstand` attribute set to “`true`”, the node must process the header-block. If it is unable to do this, it has to report the error to the originator of the SOAP-message. This is done by sending back a SOAP-message containing a SOAP-fault element, which carries information about the error.

The body element of a SOAP-message carries the actual application data that should be exchanged between the initial sender and the ultimate receiver.
If SOAP is used to perform a remote-procedure-call (RPC), the request-message-body contains the name of the procedure to be called together with all parameters that should be passed to the service. The response message for a RPC contains the return value of the executed procedure as shown in figure 30.
However SOAP is not restricted to RPC, but can be used to exchange any XML-based content in its body.

[4] this attribute was previously called `actor` and was renamed in SOAP version 1.2

[5] which could also be the ultimate receiver

4 OPES Research Prototype

One of the goals of this thesis is to design and implement a research-prototype of an OPES-framework following the standards proposed by the IETF OPES-Working-group, including an OPES processor, an OPES callout-server, an implementation of the OPES callout-protocol (OCP) and the OCP profile for HTTP.

Chapter 4.1 describes the overall architecture of the research prototype, chapter 4.2 defines some functional and non-functional requirements that should be fulfilled by the prototype implementation and chapter 4.3 describes the implementation details.

4.1 Architectural Components

The research prototype can be divided into the following parts:

- OPES processor
  - HTTP Proxy
  - OPES Data Dispatcher
  - OPES callout protocol agent
- OPES Callout Server
  - OPES callout protocol agent
  - Service Execution Environment
  - OPES Services

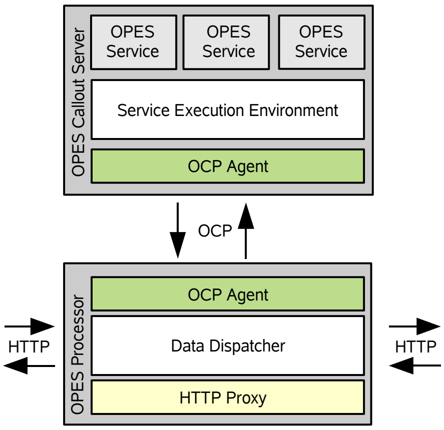


Figure 31: Prototype - Architecture

Because the architecture of the prototype is closely related to the architecture of the OPES-framework as described in chapter 2.2.2, the following chapters only provide a short summary of the architectural components of the research prototype.

4.1.1 OPES processor

The OPES processor is located on the content-path, between a content-provider and a content-consumer and acts as application-level proxy. It consists of three logical components: a HTTP proxy, an OPES Data-Dispatcher and an OPES callout-protocol-agent.

HTTP Proxy:

The OPES processor contains a not-caching HTTP-proxy, which intercepts HTTP-request- and response-messages, exchanged between HTTP-clients and HTTP-servers.
The intercepted HTTP-messages are parsed to extract application-message-properties, such as the HTTP-headers or the HTTP-status- or request-line, that are needed by the OPES data-dispatcher to determine whether an OPES-service should be performed on the intercepted application-message.
If no service must be applied to an intercepted application-message, the HTTP-proxy forwards the unmodified application-message-data to the receiver. Otherwise the data-dis-

patcher invokes the desired OPES-service and passes the received application-message-data to it. Depending on the service-type[1], the OPES-service either returns an adapted application-message which is delivered to the receiver, or a completely new generated response-message which is sent back to the sender.

OPES data dispatcher:

The data-dispatcher loads policy-rules from the file-system and uses a rule-parser to parse and compile them. A rule can be provided for one of four common processing-points[2].

If an application-message is received by the HTTP-proxy, the data-dispatcher determines if policy-rules are defined for the given processing point. Thereafter the found rules are evaluated against the application-message-properties received from the HTTP-proxy.

If a policy-rule is evaluated to true, which is the case if all conditions specified in the rule are met, the OPES-services which were defined as actions in the matched rule must be invoked by the data-dispatcher. If no rule matches, no action has to be taken. Because the data-dispatcher executes services based on policy-rules, it is also referred to as policy-enforcement-point[3].

If no service needs to be applied to a given application-message, the data-dispatcher signals the HTTP-proxy to just forward the original application-message unmodified to the receiver.

But in case that an OPES-service needs to be performed, the data-dispatcher invokes the desired service-application(s) using OCP. Therefore the data-dispatcher uses the OCP-agent, which is also located on the OPES-processor, to send an OCP-request to a callout-server, which is hosting the desired services. The callout-server starts the specified services and hands over the received original-application-message data to them. Thereafter the adapted application-message-data is sent as OCP-response back to the OPES-processor, where it is received by the OCP-agent, handed back to the data-dispatcher, which in turn passes the message to the HTTP-proxy to send it out to the receiver.

OPES callout protocol agent:

This component implements the OPES callout-protocol (OCP) and is used by the OPES data-dispatcher to invoke remote services that should be applied to the application-messages, intercepted by the HTTP-proxy.

If the data-dispatcher instructs the OCP-agent to invoke a remote service, the agent opens an OCP-connection to the desired callout-server, negotiates with the callout-server to use the OCP HTTP profile, creates a service-group to signal the callout-server which service(s) to apply to the application message, opens a new OCP-transaction, and finally starts sending the application-message data to the callout server. In parallel, the OCP-agents starts to receive adapted-application-message data from the callout-server and passes it via the data-dispatcher to the HTTP-proxy, which forwards it to the receiver.

Once both endpoints have finished data transmission, the OCP-agent closes the transaction and signals the callout-server to destroy the previously created service group.

[1] request modification service, response modification service, response generation service (see chatper 2.2.4)

[2] see the definition of the rule processing points in chapter 2.2.3

[3] PEP ... Policy Enforcement Point (see chapter 2.3.3)

4.1.2 OPES Callout Server

The callout-server is used to hosts callout-services that can be invoked by OPES-processors via the OPES callout protocol. The callout-server consists of two components: an OPES callout-protocol agent and a service-execution-environment for callout-services.

OPES callout protocol agent:

Similar to the OCP-agent located in an OPES-processor, this OCP-agent implements the OPES callout-protocol (OCP). But unlike the OCP-agent of an OPES-processor, this OCP-agent does not establish OCP-connections by itself, but listens for incoming OCP-connection.

Once an OPES-processor established a connection to the callout-server, the OCP-agent negotiates with the processor to use the OCP HTTP profile and creates a new service-group, including the services requested by the processor. Thereafter, the received original-application-message data is passed to the desired OPES-services, which are executed in sandboxes of the service-execution-environment and perform value-added services on the original-application-message data. The adapted application-message produced by the services is then sent back by the OCP-agent via the OCP-connection to the OPES-processor.
Once both endpoints have finished data transmission, the transaction is closed, the service-applications are destructed, and the previously created service group is destroyed.

Service Execution Environment:

The service-execution-environment is used to manage and control OPES-services during their runtime life-cycle, and to encapsulate service-applications in a sandbox providing well defined interfaces that allow them to interact with their environment.

After the OCP-agent has detected the start of a new application-message, it instructs the service-execution-environment to instantiate and initialize a new service-application instance, which then receives the original-application-message-data for processing. The service-application analyzes the application-message and then either generates a modified version of the application message or - in case that the original message was a request message – may generate a new response-message as result of request-message processing.
The newly generated or modified application message is then either passed to the next OPES-service in the group (as requested by the OPES-processor when creating the service-group), or is sent back by the OCP-agent to the OPES-processor.

4.2 Requirements

This chapter formulates some additional requirements, beside those formulated by the OPES-WG itself, which should be fulfilled by our prototype implementation.

4.2.1 Performance:

One important issue for an intermediary system is, how it affects the delivery-latency of application-message-packages flowing through the system.

Thus the speed of an OPES-system should be close to line-speed, at least for application-messages not processed by OPES-services but just forwarded to the receiver. And the latency added to application-messages, because of the transmission of application-message(-part)s to a remote callout-server and the execution of edge-services, should be kept to a minimum.

To accomplish this goal, the reference-implementation should use the following techniques:

Keep connections alive:

The prototype-implementation should keep established connections alive and should try to reuse them as often as possible.

The OPES-processor should use the corresponding HTTP-connection headers to indicate that the server should consider the HTTP-connections as persistent, after the current request is complete. To allow persistent HTTP-connections, the OPES-processor itself either has to set the content length header of a HTTP-messages accordingly, or has to use chunked transfer-encoding for the transmission of the HTTP-message-body. If both is not possible, the OPES-processor has to indicate the application-message end by closing the HTTP-connection, but this should be avoided.

The OPES processor and the OPES callout-server should use dedicated keep-alive messages to ensure that OCP-connections are kept alive, as suggested by the OPES working-group.

Use connection pools:

The prototype-implementation should pool persistent connections to allow multiple worker-threads to share a set of connections, established to remote servers.

The OPES-processor should maintain a connection-pool for HTTP-connections, whereas a HTTP-connection typically has a timeout and the maximum number of pooled connections for each web-server is restricted (according to [25]).
If a worker-thread needs to send a request to a web-server on behalf of a HTTP-client, the worker has to determine whether a connection to this web-server is available in the pool or if a new connection must be established.
Once the worker has finished using the HTTP-connection, it should not close and release the connection but should store it to the connection-pool so that it can be re-used by other worker-threads to send further requests to the same web-server.

Additionally, the OPES-processor should maintain a connection pool for OCP-callout-connections, but in opposite to HTTP-connections, OCP-connections do not time-out because they send dedicated keep-alive messages. The maximum number of pooled OCP-connections

should be restricted to avoid that the callout-server runs out of connection handlers.
If a worker thread needs to invoke a callout-service, it has to determine whether a connection to the desired callout-server is available in the pool or if it has to wait until a connection is free. Alternatively the OPES-processor could decide to multiplex multiple OCP-transactions over a single OCP-connection.
Once the worker has finished the callout-transaction, it should not close the OCP-connection, but should store it to the connection-pool for re-use.

Use thread pools:

The prototype-implementation should pool worker-threads to avoid the overhead of creating a new thread, every time a new request needs to be handled.

The OPES-processor should maintain a pool of HTTP worker-threads. If a new HTTP-request arrives at the HTTP-proxy, the main-thread needs to determine whether an idle worker is available in the pool or if the request must be blocked until a worker-thread becomes free. Once a free worker is available, the main-thread fetches it from the pool and passes the request to it for further processing. Then the main-thread continues to listen for new requests.
Once the worker-thread has finished processing of the request message, it returns itself back into the pool and blocks until it is re-used.

The OPES-callout-server should also maintain a pool of worker-threads, whereas in this case each worker is responsible for a single OCP connection. Thus, if a new TCP connection is established to the callout-server, the main-thread fetches a new worker-thread from the pool which then listens for incoming OCP messages.
Once the OCP connection is closed, the worker returns itself back into pool for later re-usage.

Use service pools:

The prototype-implementation should pool service-applications to avoid the overhead of instantiating and initializing a new service-application instance, every time the execution of a services is triggered.

The OPES-callout-server should maintain a pool of service-application instances.
If a new OCP-request arrives, the callout-server determines whether an instance of the desired service-application is available in the pool, or a new service-application instance must be instantiated and initialized.
Once the service execution has finished, the service-application instance should be returned into the pool for later re-use.

Use data compression:

The prototype-implementation should use data compression to increase throughput.

On the one hand, the OPES-processor should indicate to HTTP-servers that it supports receiving response-bodies that are compressed using various HTTP content-encodings, and should itself use HTTP content-encoding to deliver the response-body compressedly to a HTTP-client, in case the client supports this.

But on the other hand, the OPES-processor should not send content-encoded message-bodies via OCP to callout-servers. Sending content-encoded message-bodies is not prohibited by the specification but would prevent a callout-service from using data-preservation- or premature-termination features.

Use message segmentation:

The prototype-implementation should avoid to store and forward application-protocol messages.

The OPES-processor should use the rule-engine to determine which service to trigger for a given message, once the HTTP-message headers were received, and should start to forward the application-message-parts via OCP to the desired callout-server even before the whole message-body was received. Thus, the OPES-processor needs to split the HTTP-message body into multiple chunks and needs to send these chunks continuously to the callout-server, using separate OCP-protocol messages for each chunk.

The callout-server should be capable to pass the received message-body chunks to the invoked callout-services, and to start sending back the adapted application-message to the OPES-processor, even before the whole message-body was received and processed by the callout-services. Thus the callout-server needs to split the adapted HTTP-message parts into chunks and needs to send these chunks continuously to the OPES-processor using separate OCP-protocol messages for each chunk.

Finally, callout-services should be capable to start processing of application-messages even if they have not received the whole message so far. Thus it is desired that callout-services process application-messages as streams.

Use message buffers and pipelines:

The prototype-implementation should use byte buffers to reduce the number of low-level I/O operations needed to receive and send data.

The OCP protocol stack should use incoming- and outgoing message-buffers to avoid blocking of service-applications because of OCP I/O operations.

The OPES-processor should support HTTP-message pipelining. I.e. a HTTP-client should be able to send further requests to the processor even if the processor has not finished sending back the response message of a previous request.

4.2.2 Flexibility

The prototype-implementation should not be restricted to any special use-case scenario.

The OPES-processor, the OPES-callout-server and the service-execution-environment should be flexible enough to support the processing of even unusual application-message data-flows, such as HTTP messages with huge bodies, messages containing multiple parts, or even audio or video streams.

Additionally the prototype should not require any special HTTP-version and should not prevent clients from using HTTP as plain transport-protocol for other protocol, e.g. SOAP.

Even though the prototype only provides support for processing of HTTP-messages, the various system components should be kept as application-agnostic as possible, to reduce the efforts needed to extend the system to support additional application-layer-protocols.

4.2.3 Modularity

The prototype should follow a modular approach.

It should be possible to easily change or replace the various system-components and -modules (e.g. to replace the rule-engine of the prototype with an other engine that uses a different language for rule-definition) without the need to modify the whole system.

Additionally, it should be possible to re-use components (at least the OCP-core protocol stack, but maybe also the service-execution-environment or the rule-engine) in other projects. Thus the prototype-implementation should define suitable interfaces that allow external programmers to integrate the components in their projects.

4.2.4 Software

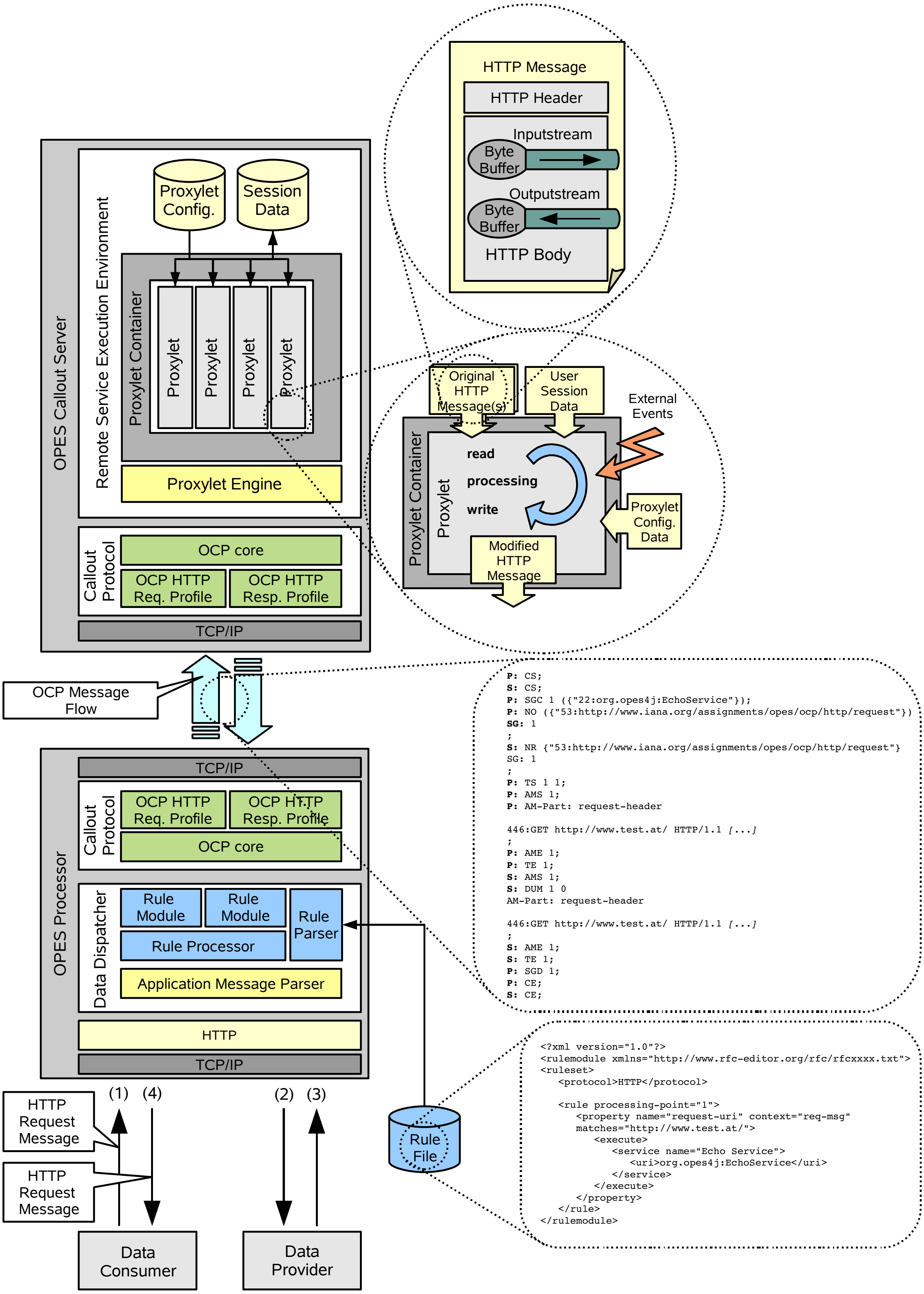
The prototype-implementation should not have extraordinary requirements regarding the operating system or system software needed to run the prototype and should be as platform independent as possible. Thus, the various system components should be implemented using the Java programming language ([45]).

Both, the OPES-processor and the OPES callout-server, should not require to be deployed in an application-server, but should be implemented as standalone server-applications. This should make it easier to integrate them into existing network environments.

Additionally, the prototype-implementation should not try to re-invent the wheel but should use existing open-source softwares, and should combine and extend them, to get the desired functionality. For instance, there are well suited open-source libraries to maintain thread- and object-pools, to parse XML configuration files, to log debugging-messages or to test the system using test-cases.

4.2.5 Hardware

The system should not require any special hardware to run.



```
P: CS;
S: CS;
P: SGC 1 ({"22:org.opes4j:EchoService"});
P: NO ({"53:http://www.iana.org/assignments/opes/ocp/http/request"})
SG: 1
;
S: NR {"53:http://www.iana.org/assignments/opes/ocp/http/request"}
SG: 1
;
P: TS 1 1;
P: AMS 1;
P: AM-Part: request-header

446:GET http://www.test.at/ HTTP/1.1 [...]
;
P: AME 1;
P: TE 1;
S: AMS 1;
S: DUM 1 0
AM-Part: request-header

446:GET http://www.test.at/ HTTP/1.1 [...]
;
S: AME 1;
S: TE 1;
P: SGD 1;
P: CE;
S: CE;
```

```
<?xml version="1.0"?>
<rulemodule xmlns="http://www.rfc-editor.org/rfc/rfcxxxx.txt">
<ruleset>
   <protocol>HTTP</protocol>

   <rule processing-point="1">
      <property name="request-uri" context="req-msg"
      matches="http://www.test.at/">
         <execute>
            <service name="Echo Service">
               <uri>org.opes4j:EchoService</uri>
            </service>
         </execute>
      </property>
   </rule>
</rulemodule>
```

Figure 32: Prototype – Components

4.3 Design and Implementation

This chapter is focused on the prototype-implementation, which is based on the architecture described in chapter 4.1. For reasons of space we only outline some interesting implementation-details here.

The architectural components are implemented in the Java programming language and are using various open-source libraries, providing basic functionalities to maintain threads-, connections- and Java-objects pools or to parse XML formatted configuration files.

For this chapter it is assumed, that the reader has some basic knowledge about the Java programming language and is familiar with frequently used Java-classes such as `InputStreams`, `OutputStreams` or Java `Threads`.

4.3.1 TCP/IP Server

As defined in chapter 4.2, both, the OPES-processor and the OPES-callout-server, should not require to be deployed in an application-server, but should be implemented as standalone server-applications, to make it easier to integrate them into existing network infrastructure.

Both servers require the use of TCP as transport protocol (both, HTTP and OCP are layered on top of TCP), need to be able to accept multiple concurrent connections from an infinite number of clients and therefore have to be implemented as multi-threaded server applications. Because it makes no sense to re-invent the wheel, we decided to use the open source library QuickServer [46] as basis of our server-applications and have extended it to fulfill our needs.

QuickServer is a server framework, implemented in Java, which can be used to develop multi-client, multi-threaded TCP servers. It can read its configuration from an XML file, uses thread-pools to increase performance, provides support for remote server administration, and has a clear design, which separates the server from the protocol- and application-logic.

Figure 33 shows the basic architecture of the QuickServer library.

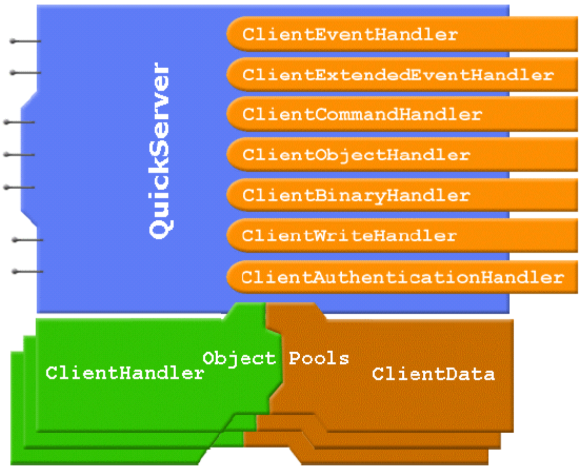


Figure 33: Prototype - QuickServer Basic Architecture ([47])

If a client connects to the server, a worker thread is picked from the pool and instructed to process the client request. How a client-request is processed, is defined in a special request-handler class, which implements the QuickServer `ClientHandler` interface and contains the actual application-logic.

In case of our prototype-implementation, the OPES-processor provides a custom handler class that acts as a HTTP-proxy, whereas the OPES-callout-server provides a custom handler that acts as an OCP-agent.

Furthermore our research-prototype extends the `QuickServer` class (the main class of the QuickServer framework) with additional functionality needed to initialize all needed subcomponents of the OPES-processor, respectively the callout-server.

4.3.2 OPES processor

The OPES-processor is based on the QuickServer server framework and is implemented as a standalone server-application.

Server startup:

On server startup, the server reads its XML configuration-file, initializes all needed subcomponents and thereafter opens a TCP/IP socket to listen for incoming client-connections.

When a client connects, the main-thread of the server fetches a new worker-thread from a pool of idle workers, hands over the client-connection to the worker and thereafter continues to listen for other incoming connections.

The server worker in turn starts to listen for incoming client-requests and calls a “handle-request” function of the request-handler class, once it detects a new request.
After the client-request was processed by the handler-class, the worker continues to listen for new client requests.

Request Handler:

The request-handler class implements the QuickServer `ClientHandler` interface and acts as special HTTP-proxy, which invokes callout services to perform value-added services on request- and response-messages, exchanged between the clients and remote web-servers. Which service is triggered for which message depends on a set of policy-rules loaded into a rule-engine.

Because services can be applied to both, the client-request- and the server-response-message, and because a web-server might be capable to start sending back the response-message, even if it has not finished receiving the request-message[4], the request-handler uses

$$(N+1)+(M+1)$$

proxy-worker-threads for message-data transmission between the OPES-processor, the HTTP-client, the HTTP-server and multiple remote OPES-callout-servers, whereas N is the number of callout-services applied to the request-message of the client and M is the number of callout-services applied to the response-message of the server, assuming that each callout-service is located on a different callout-server.

If the last service in a chain of request-modification-services does not modify the request-message but generates a response-message then only

$$(N+1)$$

proxy-worker threads are needed for message transfer.

[4] for example an audio transcoding service located on a web-server, which is capable to transcode an audio stream received from a client on the fly and uses chunked transfer encoding for data transmission.

Figure 34 depicts how many worker threads are needed in the case that a single service is applied to each, the request- and the response-message.

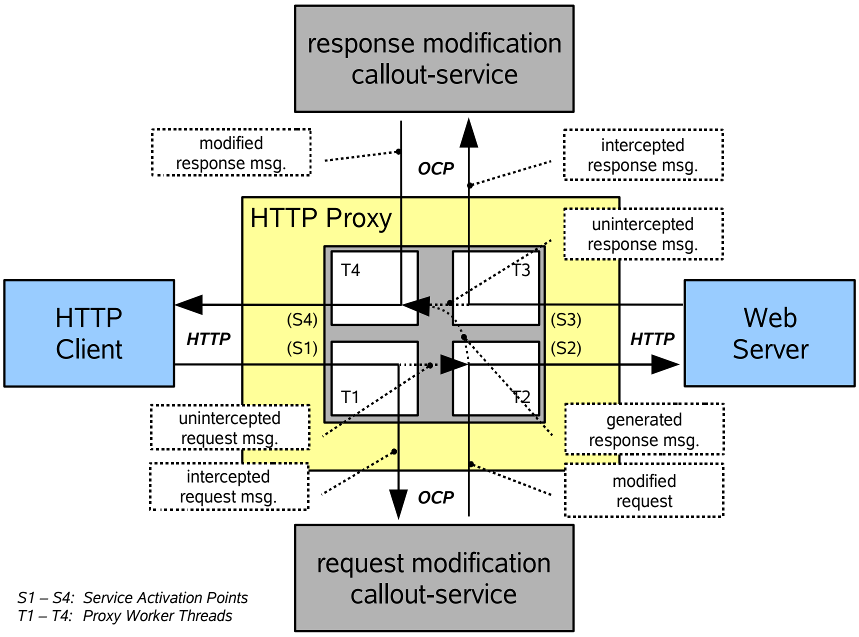

response modification callout-service
modified response msg.
OCP
intercepted response msg.
HTTP Proxy
unintercepted response msg.
T4
T3
HTTP Client
HTTP
(S4)
(S1)
(S3)
(S2)
HTTP
Web Server
T1
T2
unintercepted request msg.
generated response msg.
intercepted request msg.
OCP
modified request
request modification callout-service
S1 – S4: Service Activation Points
T1 – T4: Proxy Worker Threads

Figure 34: Prototype - Proxy Threads

In the figure shown above, the proxy-worker-thread `T1` receives the original client-request via HTTP and forwards it via OCP to a callout-server. The proxy-worker `T2` receives the adapted request-message via OCP and forwards it via HTTP to the desired web-server. The proxy-worker `T3` receives the web-servers response-message via HTTP and forwards it to another callout-server via OCP. Finally, the proxy-worker `T4` receives the adapted response-message via OCP and delivers it via HTTP to the client.

The coordination between the request-handler and its worker-threads is done using a handler-internal message-queue. If one of the worker-threads finishes message transmission, detects an error, or requests a policy decision, it inserts a message into the request-handlers message-queue and if required waits for a response.

Proxy Worker `T1`:

Once the “handle-request” function of the request-handler is called to process a new client-request, the handler starts a new proxy-worker-thread `T1`, passes an `InputStream` object to it, and instructs the worker to start reading the request-message of the client. Then the request-handler blocks on its message-queue until it receives a message from `T1`.

The newly started worker thread reads the head of the HTTP-request (i.e. the request-line and the HTTP-header block) from the stream, parses the data and inserts a new message into the message-queue of the handler to request a policy decision for service-activation-point one (`S1`). The enqueued notification-message is used on the one hand to pass the parsed application-message-properties to the request-handler and on the other hand as synchronization object to allow the worker-thread to block on the message-object until the handler returns the result of the policy-decision.

The request-handler passes the extracted message-properties to the rule-engine, which evaluates policy-rules against them to determine whether an OPES-service should be applied to the request-message, or the message should be left unmodified.

In both cases, the request-handler passes a Java `OutputStream` back to the worker-thread. The worker uses this stream to first write out the previously parsed HTTP-head and thereafter copies the remaining parts of the request (the HTTP-body and trailer parts) from the `InputStream` to the `OutputStream`.

If the rule-engine came to the decision that no service should be performed on the request-message, the returned `OutputStream` is used to send the request-message data via a HTTP-connection to the web-server that was addressed by the client.

In case a callout-service needs to be executed, the `OutputStream` is used to send the request-data via an OCP-connection to a remote callout-sever. The `OutputStream` transparently sends an `AMS`-message[5] to the callout server to signal the start of a new application-message, splits the data written to it into chunks, encapsulates each chunk into a separate OCP `DUM`-message[6], and sends the `DUM`-messages via the OCP-connection to the callout-server. After the worker has finished writing to the stream, it calls its close method. This causes the stream to send an `AME`-message[7] to the callout-server to indicate the end of the application-message. Thereafter the worker-thread can close the OCP-translation.

The HTTP- or OCP-connection that is used by the `OutputStream` to send the data to its destination, is established transparently by the request-handler. Thus, the worker-thread doesn't need to know details about the connection and in case of OCP doesn't need to know which OCP-features or -profiles to negotiate or which service-groups to create. I.e. all negotiation- and initialization-work is done transparently by the request-handler.

Proxy Worker `T2`:

If the proxy-worker thread `T1` requests a policy-decision for service-activation-point one, the request-handler contacts its rule-engine to determine whether a callout-service should be applied to the application-message.

If no service needs to be performed, the request-handler simply establishes a connection to the remote web-server and passes a Java Output`Stream` to `T1`. The worker thread then uses the stream to send the client-request to the web-server.

But if a callout-service needs to be executed, the handler connects to the callout-server, which hosts the desired service, establishes and initializes the OCP-connection, and sends a `SGC`-message[8] to create a new service-group, containing the identifiers of the callout-services that should be invoked. Thereafter it sends a `TS-`message[9] to indicate the start of a new OCP-transaction and specifies that the service-group should be applied to all application-messages that are consecutively sent over the transaction.
Finally, the handler creates a special Java `OutputStream` for OCP and passes it to `T1`. The worker thread then uses the stream to send the request-message-data to the desired callout-server.

[5] AMS ... OCP Application-Message-Start message

[6] DUM ... OCP Data-Use-Mine message

[7] AME ... OCP Application-Message-End message

[8] SGC ... OCP Service-Group-Create message

[9] TS ... OCP Transaction-Start message

In parallel, the request-handler starts a new proxy-worker-thread T2, creates a special Java InputStream for OCP and passes the stream to T2, which then uses it to receive all incoming OCP-messages that belong to the created OCP-transaction and to extract the adapted application-message data generated by the callout-service(s).

Once T2 has finished receiving the header of the adapted HTTP-message, it parses the header-data, inserts a new message into the request-handlers message-queue to request a policy-decision for service-activation-point two (S2), and thereafter blocks on the message-object until the handler returns a response.

If the rule-engine comes to the decision that an additional callout-service needs to be applied to the request-message (in this case the adapted request-message), the request-handler initializes another proxy-worker thread of type T1 and instructs it to send the adapted-message to another callout-server. In this case, the InputStream that is passed to the new worker-thread is a pipe, which is filled by T2 with adapted-message-data and is read out by the new worker-thread.

If no further callout-services need to be applied to the request-message, the request-handler establishes a HTTP-connection to the remote web-server and passes an OutputStream to T2, which uses the stream to deliver the adapted request-message to the web-server.

Proxy Worker T3:

If the proxy-worker-thread T2 requests a policy-decision for service-activation-point two (S2), the request-handler uses the rule-engine to determine, whether further callout-services need to be invoked or if the request can be delivered to the remote web-server.

If no services needs to be executed, the request-handler starts a new proxy-worker-thread T3 and passes an InputStream object to it, which belongs to a HTTP-connection established by T2 to the remote web-server. T3 then uses the InputStream to read the server response.

T3 works much similar to T1 except that it operates on response- instead of request-messages. T3 receives and parses the HTTP-response-message returned by the web-server, requests a policy-decision for service-activation-point three (S3), and thereafter blocks on the message-object to wait for a response. The request-handler either returns an OutputStream to send the response-message via OCP to a callout-server (in the same way as already described for T1), or an OutputStream that belongs to the client-connection and can be used to deliver the unmodified response-message to the client.

Proxy Worker T4:

If the proxy-worker-thread T3 requests a policy-decision for service-activation-point three (S3), and the rule-engine comes to the decision that a callout-service needs to be applied to the response-message, T3 gets a special OutputStream to send the response-message via OCP to the desired callout-server.

In parallel, the request-handler starts a new proxy-worker-thread T4 and instructs it to receive the OCP-response generated by the callout-server via a special Java InputStream for OCP. T4 receives the adapted response-message, parses it, sends a policy-decision-request for service-activation-point four (S4) to the request-handler and waits for the response of the handler.

If the rule-engine decides that another callout-service needs to be invoked, the request-handler initializes another proxy-worker-thread of type `T3` and instructs it to send the adapted response-message to another callout-server, whereas the `InputStream` passed to the new worker works as a pipe that is filled with data by `T4`.

If no further callout-service should be invoked, `T4` receives an `OutputStream` that belongs to the client-connection and uses the stream to deliver the adapted response-message to the client.

HTTP connections:

As described above, the request-handler transparently establishes and initializes HTTP-connections to remote web-servers and passes the corresponding input- and output-streams to proxy-worker threads, which use these streams to send request-message-data to or receive response-message-data from the remote web-servers.

To avoid the overhead of establishing a new HTTP-connection, each time a new request needs to be sent to a web-server, the request-handler uses a HTTP-connection pool to maintain persistent HTTP-connections.

Because the implementation of a HTTP-connection pool is quite complex, we decided to use the Apache Jakarta Commons HttpClient [48] Java library, which provides a connection-manager class that keeps a pool of HTTP-connections. These connections are then shared by multiple threads, whereas “ping” messages are used by the manager to ensure that idle connections hold in the pool are kept alive as long as possible.

HttpClient is not based on the Java `URLConnection` objects, but provides a custom `HttpConnection` object, which has the capability to detect a stale connection and ensures that the connection is kept open, even if the input- or output-stream of the connection is closed. After the `releaseConnection` method of the `HttpConnection` is called, it is returned into the pool and can be reused by other threads to transmit further requests to the remote web-server.

OCP connection:

OCP-connections are also established and initialized transparently by the request-handler.

If one of the proxy-worker-threads requests a policy-decision for one of the four service-activation-points, the request-handler calls a function of the rule-engine to determine whether a callout-service needs to be invoked. If this is the case, the request-handler determines the callout-server on which the service is located and thereafter needs to establish a new callout-transaction to the remote callout-server.

To avoid the overhead of establishing and initializing a new OCP-connection each time a callout-service needs to be invoked, the request-handler maintains a pool of ready to use OCP-connections.

If an idle connection to the desired callout-server is available in the OCP-connection-pool, the request-handler neither needs to establish a new connection nor needs to negotiate the OCP-features and -profiles to use for the new connection. Thus, it just can create a new service-group indicating the callout-services to invoked, and passes service-specific parameters to the callout-server. Thereafter a new OCP-transaction can be established that uses the newly created service-group for application-message processing.

Once the OCP-transaction is established, one proxy-worker-thread is used to write the original application-message-data to a special Java `OutputStream` for OCP, which transparently sends an `AMS`-message and the application-message-data encapsulated in multiple `DUM`-messages via the OCP-connection to the callout-server.

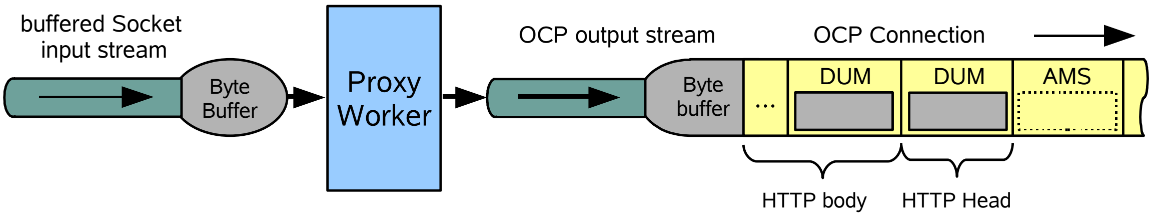


Figure 35: Prototype – OCP output stream

In parallel a second worker-thread uses a special `InputStream` for OCP to receive the callout-server OCP-response-messages and extract the adapted application-message-data from the DUM-message payload. Thereafter the data is forwarded using HTTP to the receiver.

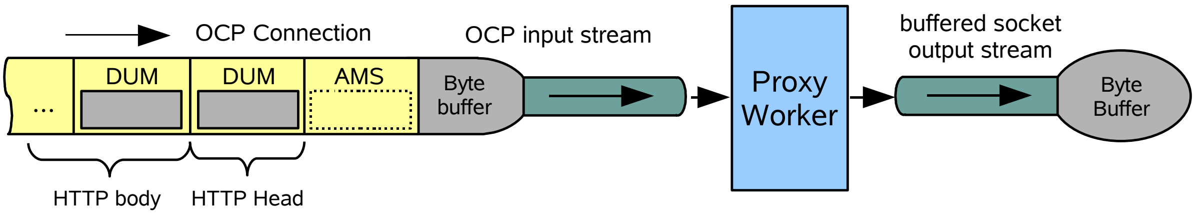


Figure 36: Prototype - OCP input stream

After both worker-threads have finished their work and have closed the OCP-transaction, the request-handler is notified to destroy the previously created service-group and to put the OCP-connection object back into the pool.

The pooled connections are automatically kept alive by using dedicated keep-alive messages (in our prototype we used `PR` messages[10] for this), which are sent transparently by the OCP stack. Thus, the request-handler does not need to take care of this.

4.3.3 Rule Engine

As described in chapter 2.2.2, each OPES-processor must contain an OPES-data-dispatcher application, which acts as a policy-enforcement-point and parses and matches application-messages against policy-rules.

Policy Decisions:

In our prototype implementation, the data-dispatcher functionality is compounded cooperatively by multiple component parts. For instance, the proxy-worker-threads are responsible to parse the application-messages, whereas the request-handler of the OPES-processor uses a rule-engine to evaluate policy-rules against the extracted application-message-properties.

[10] OCP Progress-Report Messages (see chapter 2.2.5)

For this evaluation, the rule-engine takes a set of message-properties as input and additionally requires to know the service-activation-point for which a policy-decision was requested.

The rule-engine internally loops through a list of rules provided for the specified activation-point[11] and checks for each rule, if all criteria defined in the rule conditions are met by the given application-message (resp. by the extracted message-property values).
Once a rule evaluates to true, the engine returns information about the action, defined in the body of the matched rule, back to the response-handler. This information contains at least the identifiers of the remote callout-service(s) to execute and may contain additional environment-variables or message-properties that should be passed as parameters to the service-application.

In the next step, the request-handler determines on which remote callout-server the desired services are hosted, bundles services that are located on the same server together to a service group, opens a connection to the callout-server (or fetches an idle connection from the connection-pool) and sends an OCP `SGC`-message to the callout-server to indicate which services should be invoked. If service-specific parameters are defined in the policy rule, they are transferred as part of the `SGC`-message.

Rule Language format:

The OPES working group was initially chartered with the sub-goal to “define a rule language to control selection and invocation of services by an OPES processor” [49].
Even though some drafts were published supposing rule-language candidates such as the Message-Processing-Language “P” [50] or the Streaming-Rules-Language “Hopalong” [51], the working group didn't reach a consensus on the direction they should move forward. In the end, the work item of defining a rule-language was removed from the charter of the working group [52].

For this reason and because of recommendations found in [19] and [39], we decided to use the Intermediary Rule Markup Language (IRML) as rule language for our research prototype.

As described in chapter 2.4, IRML is an XML markup language used to define policy rules that are loaded and evaluated by an OPES device. Based on these rules the OPES device decides, which action (if any) should be applied to an application message.

Rule file parsing:

Figure 37 shows an example IRML rule. Please note that for greater clearness, some mandatory IRML tags such as the `author` or `authorized-by` tags were cut out of the example shown below.

```
<?xml version="1.0"?>
<rulemodule xmlns="http://www.rfc-editor.org/rfc/rfcxxxx.txt">
<ruleset>
  <protocol>HTTP</protocol>

  <rule processing-point="1">
    <property name="request-uri" context="req-msg"
      matches="http://theli:8090/UserToken">
      <execute>
        <service name="SOAP WS-Security UserToken Insertion Service">
```

[11] this point is often referred to as a rule-processing-point

```
          <uri>opes://callout:8071/org.opes4j:UserToken</uri>
        </service>
      </execute>
    </property>
  </rule>
</ruleset>
</rulemodule>
```

Figure 37: Prototype – Policy rule example

For the step of loading an IRML rule, we decided to use the Apache Jakarta Commons Digester [53] library published by the Apache Software Foundation. Commons Digester is a Java library that can be used to read and parse any kind of XML file, and allows to specify an XML-Tag to Java-object-mapping via a parser-rule file.

The following figure shows how a digester-rule, needed to parse the example IRML-rule shown in figure 37, looks like[12]. It must be noted that this is not the whole parser-rule-file provided by the prototype-implementation, but just those parts needed to parse the example IRML rule.

```
<?xml version="1.0"?>
<digester-rules>
  <pattern value="rulemodule">
    <pattern value="ruleset">
      <object-create-rule classname="org.opes4j.irml.Ruleset" />
      <set-next-rule methodname="setRuleset" />

      <call-method-rule pattern="protocol" methodname="setProtocol"
      paramcount="0" />
      <pattern value="rule">
        <object-create-rule classname="org.opes4j.irml.Rule" />

        <set-properties-rule>
          <alias attr-name="processing-point" prop-name="processingPoint"/>
        </set-properties-rule>
        <set-next-rule methodname="addRule" />
      </pattern>
    </pattern>
  </pattern>

  <pattern value="*/property">
    <object-create-rule classname="org.opes4j.irml.Property" />
    <set-properties-rule>
      <alias attr-name="name" prop-name="name" />
      <alias attr-name="context" prop-name="context" />
      <alias attr-name="matches" prop-name="matches" />
    </set-properties-rule>
    <set-next-rule methodname="addProperty" />
  </pattern>

  <pattern value="*/execute">
    <object-create-rule classname="org.opes4j.irml.Execute" />
    <set-next-rule methodname="setExecute" />
  </pattern>

  <pattern value="*/service">
    <object-create-rule classname="org.opes4j.irml.Service" />
      <set-properties-rule>
      <alias attr-name="name" prop-name="name" />
      <alias attr-name="failure" prop-name="failure" />
    </set-properties-rule>
    <call-method-rule pattern="uri" methodname="setUri" paramcount="0" />
    <set-next-rule methodname="setService" />
  </pattern>
</digester-rules>
```

Figure 38: Prototype – Commons Digester rule-file.

[12] You are right, we are using a rule-file to specify how the parser should parse our IRML rule-file.

Much similar to other rule-languages, a digester rule consists of conditions and corresponding actions. For digester a condition is specified using a `pattern` tag, which is used to match XML tags based on their name and position in the XML tree. But it's also possible to use wild-cards, e.g. to match a given XML-element, independently from its location in the XML tree. For instance, the pattern "`*/property`" shown in the above example, matches all IRML-`property` elements, even for elements nested in other elements.

Once a pattern match is found, all actions associated with the pattern are performed in the order they are specified. For instance, an action can be the creation of a new Java object, setting the attributes of an object or adding a child- to a parent object (e.g. to establish parent-child relationships). An example for the various actions is shown in the figure 39.

```
<pattern value="*/property">
  <object-create-rule classname="org.opes4j.irml.Property" />
  <set-properties-rule>
    <alias attr-name="name" prop-name="name" />
    <alias attr-name="context" prop-name="context" />
    <alias attr-name="matches" prop-name="matches" />
  </set-properties-rule>
  <set-next-rule methodname="addProperty" />
</pattern>
```

Figure 39: Prototype – Commons Digester pattern example

As shown in the example above, if a `property` element is found in the IRML file, a new Java class of type `org.opes4j.irml.Property` is created. Thereafter various fields of the object are set, if the corresponding XML attributes are present in the `property` element. Finally the newly created `property` object is added to the list of property-elements in its parent Java object.

4.3.4 Callout Server

The OPES-callout-server is also based on the QuickServer server framework and implements a standalone server-application accessible via OCP.

Server startup:

On server startup, the XML configuration file is loaded, all needed subcomponents, such as an OCP-extension-manager and a service-execution-environment for OPES-services, are loaded and the server is bound to a TCP/IP socket to listen for incoming OCP-connections.

If a remote OCP-agent connects to the server, the main-thread fetches a new worker-thread from pool, which is then responsible to handle the connection using a special connection-handler class.
After the OCP-connection was closed by either endpoints, the worker is returned to pool for later reuse.

Connection Handler:

The OCP-connection-handler class implements the QuickServer `ClientHandler` interface. If a new incoming OCP-connection is accepted, an `OcpConnection` object is created and passed to the handler class.

Once started, the handler uses methods of the `OcpConnection` class to listen for incoming

OCP-messages. These messages are returned by the OcpConnection object in the form of Java Message objects (resp. objects inherited from it). Each message-type needs to be handled differently.

Connection meta-data and information about established service-groups, active transactions or parameters used for service-execution are stored by the handler in an OcpConnectionContext object.

Negotiation Phase :

If a SGC message is received, the connection-handler contacts the service-execution-environment to determine whether the requested services are known and if so, creates a new OcpServiceGroup object, representing the chain of services that should be applied to subsequently received application-messages, and stores the object in the OcpConnectionContext.
In case that at least one of the requested services is not known, the error is reported, the connection is closed and the handler function is terminated.

If a NO-message is received, the handler inspects the list of suggested OCP-features contained in the message and uses the OCP-extension-manager to determine which one of the suggested features is supported. The selected feature is then acknowledged by sending back a NR-message to the OCP-processor.
If none of the suggested features is supported, an empty NR-message is sent back to reject the offer.

For each supported feature, a separate class is provided that implements the Extension interface. Currently, the prototype provides two classes, the HttpRequestExtension supporting the OCP HTTP-request profile and the HttpResponseExtension supporting the HTTP-response profile.

Once a feature is selected, the corresponding Extension class is created and initialized by the extension-manager, stored in the OcpConnectionContext, and can subsequently be used to process incoming OCP-messages that are in the scope of the negotiated feature.
Each extension-class must implement the processMessage function, which takes a received Message object and the OcpConnection- and the OcpConnectionContext-objects as input-parameters.
The OcpConnection is needed by extensions to send response messages back to the OPES-processor, whereas the OcpConnectionContext is used to read out metadata, such as service-specific parameters to pass to an invoked OPES-service.

Data Exchange Phase:

If a TS-message is received, the connection-handler creates a new OcpTransaction object and stores it in the OcpConnectionContext. Additionally a reference to the previously created OcpServiceGroup is assigned to an OcpTransaction field, to remember which service-group should be applied to messages within transaction-scope.

If an AMS-message is received, the message is passed to the previously created Extension-class, which thereafter fetches the OcpTransaction, to which the received AMS-message belongs to, from the connection-context, starts a new service-execution-thread[13] and passes the OcpTransaction object to it.

The service-execution-thread uses the service-execution-environment to create and initialize a new callout-service of the type specified in the service-group active for the transaction. Thereafter it creates various input- and output-streams that are needed by the service-applications to read the original application-message-data and write out the adapted- or generated application-message.

In case of a request-modification service, an InputStream and two OutputStreams are created. The InputStream is used to read the request-message, the first OutputStreams is used to write out the adapted request-message, and the second OutputStream is used to write out a generated response-message.

In case of a response-modification service, two InputStreams and a single Output-Stream are created. The two InputStreams are used to read the request- resp. the response-message, and the OutputStream is used to write out the adapted response-message.

In any case, the created InputStreams are pipes filled by the extension-class with original application-message-data and read out by the service-application. The OutputStreams are special streams for OCP. Thus the data written out by the service-application is transparently encapsulated by the streams into DUM-messages and transferred over the OCP-connection to the OPES-processor.

As a final step, the service-execution-thread starts the callout-service and waits until the service execution has finished.

In the meantime received DUM-messages are passed to the Extension class, which extracts the application-message-data from the message-payloads and writes it into the corresponding piped InputStreams.

If an AME-message is received, the Extension-class closes the piped InputStreams to signal the service-application-that the end of the application-message is reached. Thereafter it waits until the service-execution-thread terminates and sends an AME-message back to the OPES-processor, containing the status code returned by the service-execution-thread.

Finally a TE-message is sent to the OPES-processor to indicate the end of the transaction.

Connection Termination:

If a CE-message is received by the connection-handler, but service-applications are still running, the corresponding service-execution-threads are instructed to terminate service-execution. Thereafter the connection is closed and the handler function returns.

4.3.5 Service Execution Environment

The OPES-callout-server contains an execution-environment for callout-services.

The Service Execution environment implemented by the research prototype is based on the concept of proxylets, as described in chapter 2.3, and implements the proxylet-execution-environment API as specified by Walker in [19] and described in chapter 2.3.1.

[13] if multiple services must be applied to an application-message, multiple threads must be started

Proxylet Execution Environment:

Although Walker distinguishes between callout services located on remote servers, and locally hosted services, the prototype-implementation does not distinguish between these two categories of services.

Even though the implemented proxylet-execution-environment is currently only used to execute services on the callout server, it could be also used as an execution environment to execute services locally on an intermediary device.

The main difference between these two purposes is that remote-proxylet may require additional functions, allowing them to take advantage of special OPES callout-protocol features such as data-preservation or premature-dataflow-termination. But our research prototype does not provide such functions.

Proxylets:

As described in the previous chapter, a new proxylet is started each time a new `AMS`-message is received. The service-execution-thread then creates piped `InputStreams` that are filled with incoming application-message data and special `OutputStreams` to write out either the adapted- or the generated application-message.

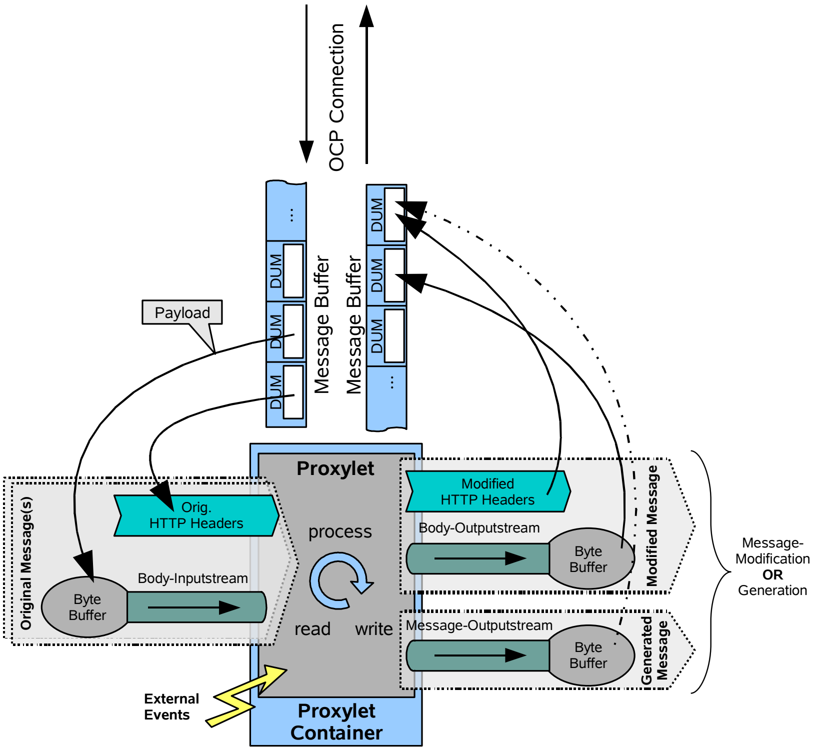


Figure 40: Prototype - Proxylet

Figure 40 shows, how a proxylet interacts with its environment.

A proxylet can access the `Input`- and `OutputStreams` via the `getPayloadInputStream`, respectively the `getPayloadOutputStream` function of the `HTTPProxyletRequest`- or `HTTPProxyletResponse` object. Additionally the stream to write out a generated response-message can be fetched using the `getResponseOutputStream` function of the `HTTPProxyletRequest` object.

Thus, there is no need to adapt existing proxylets so that they are able run in the prototype proxylet-execution-environment. They can simply be used as is.

Proxylet structure:

The following figure shows the basic structure of a proxylet-service.

```
import org.ietf.opes.proxylet.ProxyletException;
import org.ietf.opes.proxylet.ProxyletSession;
import org.ietf.opes.proxylet.ProxyletStatus;
import org.ietf.opes.proxylet.http.HTTPProxyletRequest;
import org.ietf.opes.proxylet.http.HTTPProxyletResponse;
import org.opes4j.services.AbstractService;

public class myOpesService extends AbstractService {
   public ProxyletStatus init(ProxyletConfig config) throws ProxyletException{
      // initialize the proxylet [...]
      return new ProxyletStatusImpl(ProxyletStatus.PROXYLET_OK,"OK");
   }
   public ProxyletStatus modRequest(
         HTTPProxyletRequest request,
         ProxyletSession session
   ) throws ProxyletException {
      // reading message metadata (e.g. the used content encoding)
      String contentEncoding = response.getHeader("Content-Encoding");
      [...]
      // getting message body input and output streams
      InputStream bodyIn = request.getPayloadInputStream();
      OutputStream bodyOut = request.getPayloadOutputStream();
      // process original request body and modify it
      int c;
      byte[] buffer = new byte[512];
      while (!this.aborted && (c = input.read(buffer)) > 0) {
         // do something with the read data
         [...]
         // write the adapted data out
         output.write(buffer,0,c);
      }

      // return the overall processing status (OK in this example)
      return new ProxyletStatusImpl(ProxyletStatus.PROXYLET_OK,"OK");
   }
   public ProxyletStatus modResponse(
         HTTPProxyletRequest request,
         HTTPProxyletResponse response,
         ProxyletSession session
   ) throws ProxyletException {
      [...]
   }
}
```

Figure 41: Prototype - Proxylet class structure

A proxylet can simply access and modify application-message meta-data, such as HTTP-headers but also parts of the request- or status-line, by using corresponding `get`-, `set`-, `add`, `remove`-`Header` functions provided by the HTTP request- and response-objects.

However, the HTTP-headers can no longer be modified once the output-stream was requested via the `getPayloadOutputStream` method, because the first call to this function causes the message-object to send an `AMS`-message and thereafter to send a `DUM`-message containing the HTTP-Head via the current OCP-connection to the OPES-processor.

We have chosen this approach because otherwise a store and forward approach must be used, but this would not be feasible for the processing of application-messages with huge bodies or even audio- or video-streams.

Changes to the proxylet API:

For our prototype-implementation we had to changed some of the Java interfaces defined by Walker in [19].

Walker seems to assume that a response-modification service always has access to the `HTTPProxyletRequest` object. But this not true for OPES callout-services, because the OCP HTTP-response profile defines that the request-message parts are auxiliary parts. Thus they are not sent per default, but only if explicitly negotiated.

This causes problems, because the `HTTPProxyletRequest` object is on the one hand used to register and de-register the proxylet as proxylet-event-listener, and on the other hand to set- and get- attribute values assigned to the proxylet-instance, e.g. service-specific parameters defined by a rule-author in an IRML-rule. But if no request-data is available, no `HTTPProxyletRequest` object can be created by the proxylet-engine and therefore the proxylet could not call the corresponding object-functions.

The solution for this problem was quite simple. We just decided to move the problematic functions to the `ProxyletContext` object.

4.3.6 Callout Protocol Implementation

In the previous chapters we have described at a high level how the various framework components exchange application-messages using OCP.

Now we will provide an example how our OCP implementation can be used at the OCP-message level to exchange data using OCP.

OCP implementation usage example (message level):

The following example shows how the message-flow example presented in figure 15 (chapter 2.2.5) could be realized on the OPES-processor side by using the OCP-protocol implementation provided by our research-prototype.

The example uses a fictive profile to transfer chat-messages over OCP to a translation-service.

```
// open a connection to a callout server
OcpConnection connection = new OcpConnection();
connection.establishCalloutConnection("callout",8071);

// send the mandatory CS message
connection.sendMessage(new CS());

// create a service-group
ServiceList serviceList = new ServiceList();
Service translation = new Service("22:org.opes4j:language-translation");
translation.addNamedParameter("languagePair","de-en");
serviceList.addService(translation);

// transmit the SGC message, service-group-ID=3
connection.sendMessage(new SGC(3,serviceList));

// create OCP feature-list
Feature offeredFeature = new Feature("chat-profile");
FeatureList fl = new FeatureList();
fl.addFeature(offeredFeature);

// send negotiation: service-group-ID=3, offer-pending=false
connection.sendMessage(new NO(features,3,false));

// receive negotiation answer
NR msg = (NR) connection.readMessage();

// get selected feature and test if everything is ok
Feature selected = msg.getFeature();
if((selected == null) ||
   (!selected .getFeatureUri().equals(offeredFeature.getFeatureUri()))) {
  // the requested feature is not supported, close connection and return
  connection.close();
  return;
}

// init a new transaction, transaction-ID=1, service-group-ID=3
connection.sendMessage(new TS(1,3));

// signal start of application-message, transaction-ID=1
connection.sendMessage(new AMS(1));

// send data, transaction-ID=1, offset=0
DUM data = new DUM(1,0);
data.setPayLoad("Hallo Welt!\r\n".getBytes("UTF-8"));
connection.sendMessage(data);

// signal end of application-message,transaction-ID=1
connection.sendMessage(new AME(1));

// close transaction,transaction-ID=1
connection.sendMessage(new TE(1));

// read adapted message
Message msg = null;
while(true) {
  // read the next message from the connection
  msg = (Message) connection.readMessage();

  if (msg instanceof DUM) {
     // print out the translated message
     System.err.println(((DUM)msg).getPayLoad());
  } else if (msg instanceof CE || msg instanceof TE) {
     // connection or transaction terminated
     return;
  }
}
```

Figure 42: Prototype – OCP implementation usage example (message level)

OCP implementation usage example (stream level):

Next we would like to demonstrate how our OCP implementation can be used at a higher abstraction level. Instead of using the various OCP-message objects as shown in figure 42, a special OCP `OutputStream` is used to send HTTP-messages to a callout-server.

```
// a java object representing the OCP transaction to use
// it is assumed that the transaction was initialized elsewhere
OcpTransaction transaction;

// create an OutputStream for the OCP HTTP profile
// and pass the transaction to use to it
HttpOutputStream httpOutput = new HttpOutputStream(transaction);

// our example HTTP response message
String httpMsg =
   "HTTP/1.1 200 OK\r\n" +
   "Date: Thu, 16 Nov 2006 14:47:27 GMT\r\n" +
   "Content-Length: 39\r\n" +
   "Content-Type: text/html\r\n" +
   "\r\n" +
   "<html><body><h1>Test</h1></body></html>";

   // set the AM-EL parameter (optional)
   httpOutput.setContentLength(39);

   // write out the data
   httpOutput.write(httpMsg.getBytes("UTF-8"));

   // signal end of application-message
   httpOutput.close();
```

Figure 43: Prototype – OCP implementation usage example (stream level)

The code shown above generates the following OCP-message-flow on the OCP-connection:

01	P	AMS 1 AM-EL: 39 ;
02	P	DUM 1 0 AM-Part: response-header 142:HTTP/1.1 200 OK Date: Thu, 16 Nov 2006 14:47:27 GMT Accept-Ranges: bytes Content-Length: 39 Content-Type: text/html Connection: close ;
03	P	DUM 1 142 AM-Part: response-body 39:<html><body><h1>Test</h1></body></html> ;
04	P	AME 1;

Figure 44: Prototype – OCP implementation usage example output (stream level)

As shown above, the OCP `HttpOutputStream` automatically determines the type of the HTTP-message, i.e. whether it is a request- or response-message, and splits the HTTP-message accordingly into application-message-parts and if necessary into multiple data-chunks.

5 Case Study

In this chapter we present a case-study to illustrate the functionality of the prototype-implementation and its components by the use of example services.

The provided examples can be divided into OPES-services acting on HTTP-request- and -response-messages, OPES-services acting on SOAP-request- and response-messages, and OPES-services demonstrating how to bring content-oriented- and Web-Service together.

Even though the OPES-architecture is not restricted to a specific protocol, our prototype-implementation currently only supports HTTP. Therefore all presented examples, including those operating on SOAP-messages, are using HTTP. I.e. the SOAP-related OPES-services require SOAP-clients and servers to use the SOAP-HTTP-Binding for SOAP-message exchange.

For each introduced example we provide a short task description. Thereafter we give a short summary on the used Java libraries (if any) and techniques and describe the IRML rule used to trigger the example service. Finally the adapted application-message produced by the OPES-service is described.

5.1 Use cases for HTTP messages processing

5.1.1 Cache View Service

The first example we would like to present is a response-modification-service.

The service is invoked in case that a remote web-server returns a HTTP-response-message containing a `404` status code. This status code indicates that the server has not found a document matching the Request-URI. Most web-servers are configured to return a standard error-message in the response body that is usually not very informative, but just contains a default error message and therefore does not help the user to determine whether he has mistyped the URL or the requested resource was removed.

In this situation the example services can help. It intercepts the original request message, and tries to fetch a cached copy of the requested resource either from the Wayback Machine or the Google cache.

The Wayback Machine [54] is an Internet library provided by the non-profit organization “Internet Archive”, which archives versions of websites to prevent them from disappearing completely, when the website goes offline. An archived copy can be easily accessed via HTTP using a special URL specifying the desired web-page and either a version-date or an asterisk (to access the most recent copy). For example the URL

```
http://web.archive.org/web/19961106224842/http://www.infosys.tuwien.ac.at/
```

can be used to access a version of the Information System Institute of the Technical University of Vienna homepage, taken at November, 1996.

The Google cache contains copies taken from web-sites crawled by the Google web-crawlers. These copies can be either accessed by using a special “cached” link, displayed next to a search results displayed on the Google search page, or by using the Google SOAP Search API [55].

This OPES-service first tries to fetch an archived copy from the Wayback Machine using HTTP. If no copy was found the service tries to fetch a copy via SOAP call from the Google cache.

If a cached copy of the requested resource is available, the OPES-service replaces the original error-message with the cached copy. To avoid to confuse the user and to ensure that he recognizes the `404` error, the error-code remains untouched and a banner is inserted on top of the web-page indicating the circumstance.

Used Tools and Libraries:

The OPES-service uses two additional Java libraries. First, the Apache Commons HttpClient [48] to send HTTP requests to the Wayback Machine and second a Java library provided by Google that makes it easy to use the Google SOAP Search API.

IRML Rule:

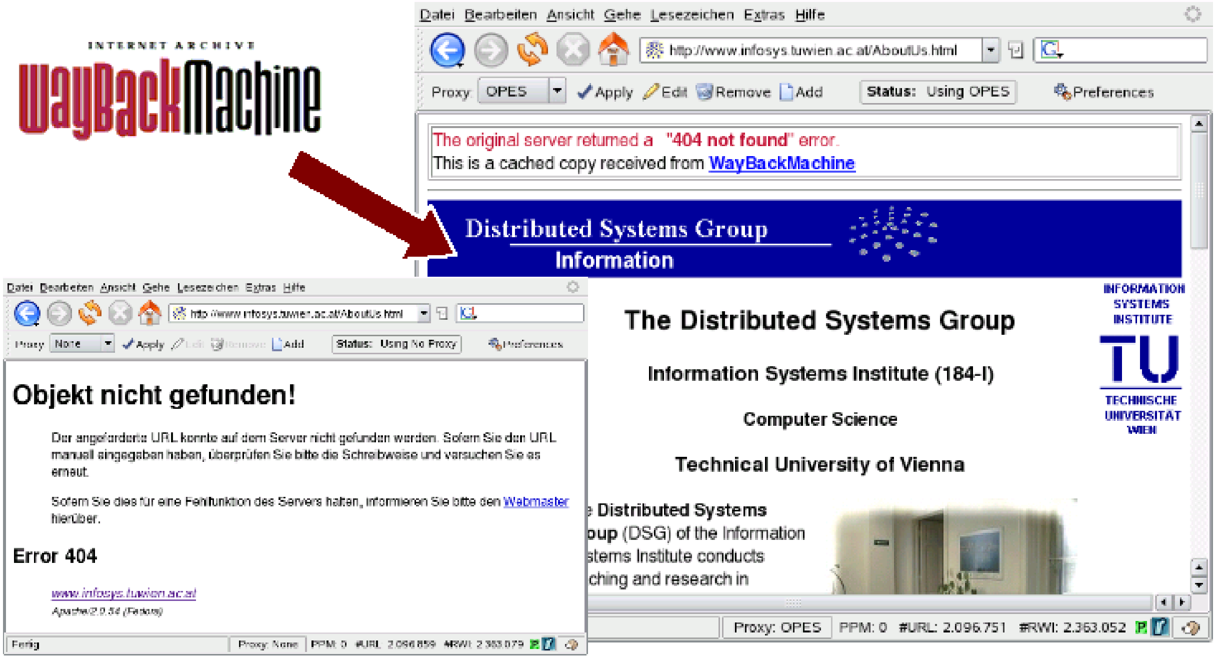
The following figure shows the IRML rule that is used to trigger the service.

```
<rule processing-point="3">
   <property name="response-code" context="res-msg" matches="404">
      <property name="request-method" context="req-msg" matches="GET">
         <property name="request-path" context="req-msg" matches=".*(/|\.html)$">
            <execute>
               <service name="Cache view service">
                  <uri>opes://callout:8071/org.opes4j:ViewCacheService</uri>
                  <parameter name="requestURL" type="dynamic">
                     <variable name="request-uri" context="req-msg"/>
                  </parameter>
                  <parameter name="googleKey" type="static">
                     <value>xyzabc</value>
                  </parameter>
               </service>
            </execute>
         </property>
      </property>
   </property>
```

Figure 45: Cache View Service – IRML rule

As shown above, the callout-service is triggered for HTTP GET-requests to HTML-documents[1], if and only if the web-server returns a 404 error code in its response. Once the rule matches, two parameters are passed to the OPES-service. The first one specifies the URL of the requested resource. We have used this approach to avoid transferring the whole request-message to the callout service (see auxiliary parts in the OCP HTTP response profile). The second parameter specifies a special key that is needed to gain access to the Google SOAP API.

Screenshot:



INTERNET ARCHIVE
WayBackMachine
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://www.infosys.tuwien.ac.at/AboutUs.html
Proxy OPES
Apply Edit Remove Add
Status: Using OPES
Preferences
Proxy None
Status: Using No Proxy
Fertig
Proxy: None PPM: 0 #URL: 2.096.859 #RWI: 2.363.079
Proxy: OPES PPM: 0 #URL: 2.096.751 #RWI: 2.363.052
INFORMATION SYSTEMS INSTITUTE
TU
TECHNISCHE UNIVERSITAT WIEN

The original server returned a "404 not found" error.
This is a cached copy received from WayBackMachine

Distributed Systems Group
Information

The Distributed Systems Group

Information Systems Institute (184-I)

Computer Science

Technical University of Vienna

e Distributed Systems
oup (DSG) of the Information
stems Institute conducts
ching and research in

Objekt nicht gefunden!

Der angeforderte URL konnte auf dem Server nicht gefunden werden. Sofern Sie den URL manuell eingegeben haben, überprüfen Sie bitte die Schreibweise und versuchen Sie es erneut.

Sofern Sie dies für eine Fehlfunktion des Servers halten, informieren Sie bitte den Webmaster hierüber.

Error 404

www.infosys.tuwien.ac.at
Apache/2.0.54 (Fedora)

Figure 46: Cache View Service – Screenshot

On the left side of figure 46, the original error-messagethat is returned by a web-server if the requested document could not be found is shown. The right side displays the cached copy as loaded by the OPES-service from the Wayback Machine archive.

[1] because only HTML documents are archived by the two caches

5.1.2 News Extraction Service

This service is an example for a response-modification-service that transforms content from one to another format.

The service is triggered if the user opens the index-page of the website of the Information System Institute of the Technical University of Vienna in a feed-reader[2].
It intercepts the response-message returned by the web-server, converts the contained HTML document into an XHTML[3] document and uses XSLT[4] [56] to extract news information from the content and to generate a RSS[5]-feed containing the extracted information. Thereafter the HTTP `Content-Type` header is changed accordingly and the RSS feed is delivered instead of the original content back to the feed-reader.

The advantage of this services is, that the user doesn't need to frequently access the web-page in his browser to see if news were published, but can use his feed-reader to monitor the page for news.

Used Tools and Libraries:

This OPES-services uses two additional Java libraries.

First, the Java library jTidy [57], which is basically a HTML syntax checker and pretty printer, but can also be used to cleanup malformed HTML documents or to convert HTML document into XHTML format, what is exactly the purpose for which jTidy is used by this example service. An usage example for the library is shown in figure A.1.

Second, the Apache Xalan [58] Java library is used. It provides an XSLT processor that can be used to transform XML formatted documents into other formats such as text, HTML or other XML based formats. This OPES-service uses Xalan to transform an XHTML document into a RSS-feed based on the instructions contained in an XSLT stylesheet. This stylesheet is depicted in figure A.2.

IRML Rule:

The following figure shows the IRML rule that is used to trigger the service.

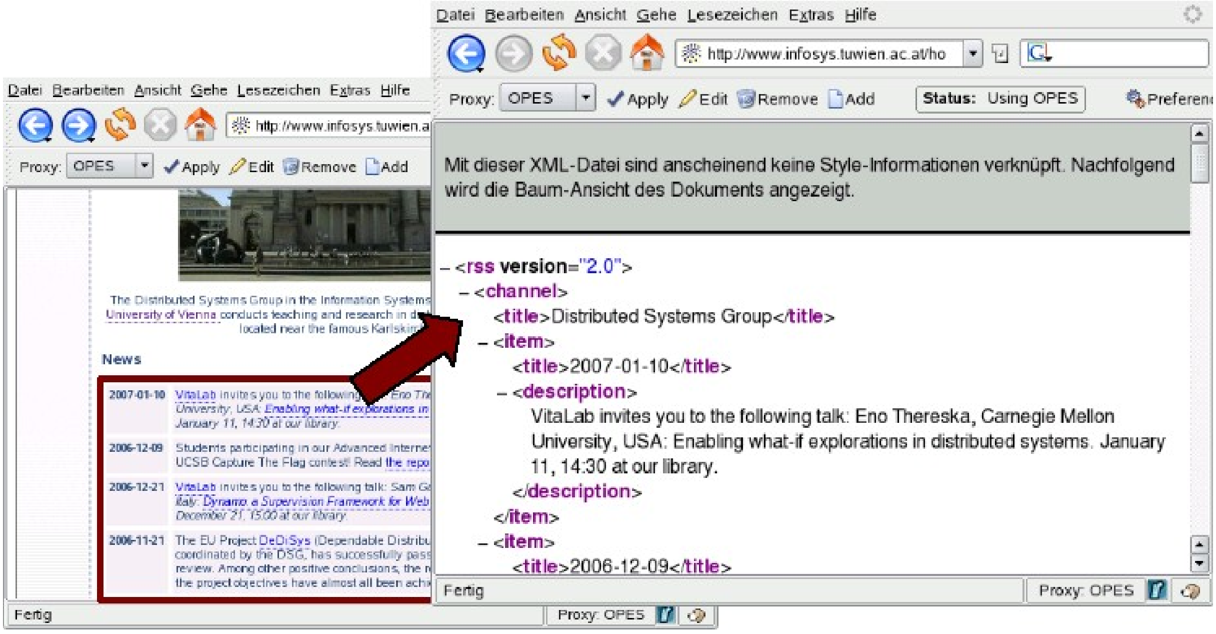
```
<rule processing-point="3">
   <property name="request-host" context="req-msg"
    matches="www.infosys.tuwien.ac.at">
      <property name="request-path" context="req-msg" matches="/home/home.page">
         <property name="User-Agent" context="req-msg" matches="Akregator.*">
            <execute>
               <service name="InfoSys News Feed">
                  <uri>opes://callout:8071/org.opes4j:InfoSysNewsFeed</uri>
               </service>
            </execute>
         </property>
      </property>
   </property>
</rule>
```

Figure 47: News Extraction Service – IRML rule

[2] a program to read RSS-, Atom- and other online news feeds
[3] eXtensible HyperText Markup Language: a redefinition of HTML in XML.
[4] eXtensible Stylesheet Language Transformations: a language to transform XML documents
[5] Really Simple Syndication:

As shown above, the OPES-service is only triggered if the user requests the URL `http://www.infosys.tuwien.ac.at/home/home.page` and if the HTTP `User-Agent` header starts with the String “`Akregator`”, which represents the name of the used feed-reader.

Screenshot:



Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://www.infosys.tuwien.a
Proxy: OPES
Apply
Edit
Remove
Add
Fertig
Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://www.infosys.tuwien.ac.at/ho
Proxy: OPES
Apply
Edit
Remove
Add
Status: Using OPES
Preferen
Fertig
Proxy: OPES

The Distributed Systems Group in the Information Systems
University of Vienna conducts teaching and research in d
located near the famous Karlskir

News

2007-01-10	VitaLab invites you to the following … Eno Th University, USA: Enabling what-if explorations in January 11, 14:30 at our library.
2006-12-09	Students participating in our Advanced Interne UCSB Capture The Flag contest! Read the repo
2006-12-21	VitaLab invites you to the following talk: Sam G Italy: Dynamo: a Supervision Framework for Web December 21, 15:00 at our library.
2006-11-21	The EU Project DeDiSys (Dependable Distribu coordinated by the DSG, has successfully pass review. Among other positive conclusions, the r the project objectives have almost all been achi

Mit dieser XML-Datei sind anscheinend keine Style-Informationen verknüpft. Nachfolgend wird die Baum-Ansicht des Dokuments angezeigt.

```
– <rss version="2.0">
  – <channel>
      <title>Distributed Systems Group</title>
    – <item>
        <title>2007-01-10</title>
      – <description>
          VitaLab invites you to the following talk: Eno Thereska, Carnegie Mellon
          University, USA: Enabling what-if explorations in distributed systems. January
          11, 14:30 at our library.
        </description>
      </item>
    – <item>
        <title>2006-12-09</title>
```

Figure 48: News Extraction Service – Screenshot

The left side of figure 48 shows how the original web-page looks like if it is viewed in a web-browser. The right side shows how the generated RSS-feed is displayed in a web-browser, if the browsers User-Agent information is set to “`Akregator`” to cause the IRML rule to match.

5.1.3 Library Query Service

This example service is a response-modification-service. It is triggered if the user views the product-information page of a book provided in the Amazon online book-shop.
The service intercepts the response-message returned by the Amazon web-server, queries the online catalog of the library of the Technical University of Vienna, and displays an information-box on top of the production-information page indicating whether the given book is also available in the university library or not.

Used Tools and Libraries:

The following libraries are used by the example-service.

First, the Apache Commons HttpClient [48] library is used to send HTTP requests to the online catalog of the university library and to receive the HTTP response. Second, the Java library jTidy [57] is used to convert the catalog response into XHTML format. Third, the Java library Apache Xalan [58] is used to extract the books availability information from the XHTML document using an Xpath[6] [59] expression.

IRML Rule:

The following figure shows the IRML rule that is use to trigger the service.

```
<rule processing-point="3">
  <property name="request-method" context="req-msg" matches="GET">
    <property name="request-host" context="req-msg" matches=".*amazon.*">
      <property name="request-path" context="req-msg" matches="/[^/]+/dp/.*">
        <execute>
          <service name="Request Deny Service">
            <uri>opes://callout:8071/org.opes4j:AmazonInsertion</uri>
            <parameter name="requestURL" type="dynamic">
              <variable name="request-uri" context="req-msg"/>
            </parameter>
          </service>
        </execute>
      </property>
    </property>
  </property>
</rule>
```

Figure 49: Library Query Service – IRML rule

As shown above, the OPES-service is only triggered for HTTP GET-requests whose request-URL follows the following schema:

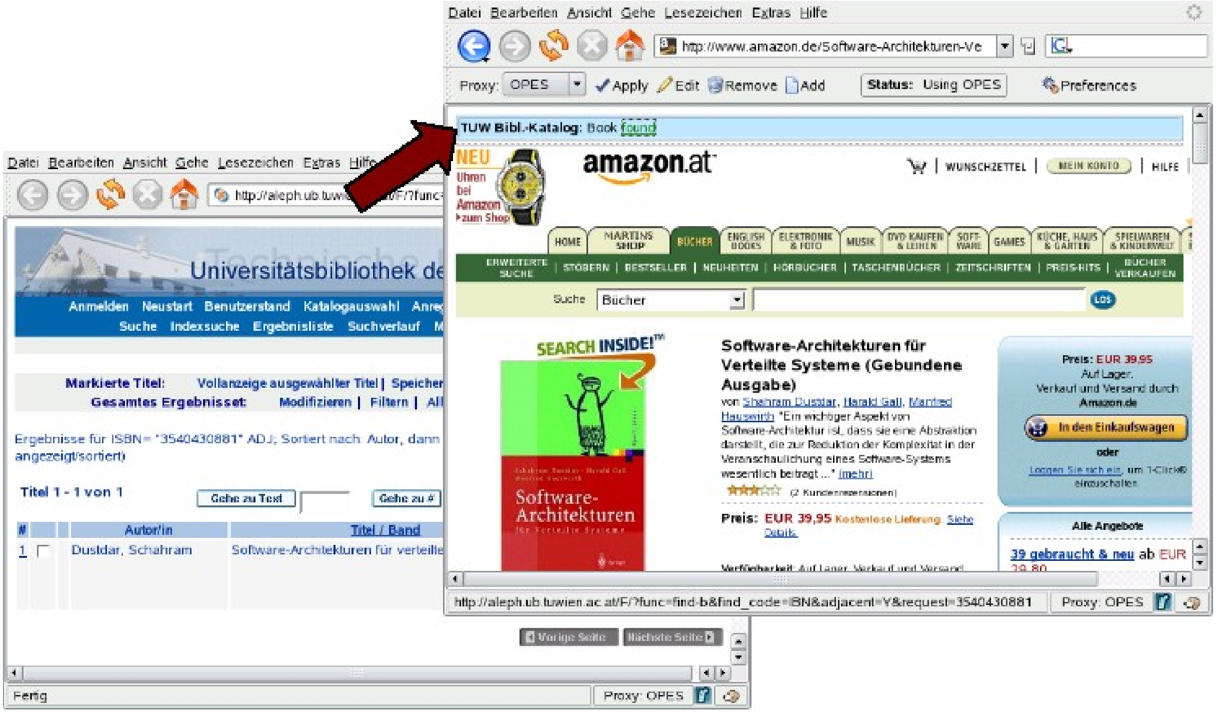
```
http://www.amazon.de/<book-name>/dp/<book-ISBN>/[...]
```

The service is only invoked for those URLs because it requires the ISBN of the displayed book to query the online catalog of the university library.

The matched request-URL is passed as service-parameter to the callout-service to avoid sending the whole request-message as separate application-message part to the callout-service.

[6] XML Path Language, “a language for addressing parts of an XML document” [59]

Screenshot:



http://www.amazon.de/Software-Architekturen-Ve

Proxy: OPES

Status: Using OPES

TUW Bibl.-Katalog: Book found

http://aleph.ub.tuwien.ac.at/F/?func=find-b&find_code=ISBN&adjacent=Y&request=3540430881

Ergebnisse für ISBN= "3540430881" ADJ; Sortiert nach: Autor, dann angezeigt/sortiert)

Titel 1 - 1 von 1

#		Autor/in	Titel / Band
1	□	Dustdar, Schahram	Software-Architekturen für verteilte

Software-Architekturen für Verteilte Systeme (Gebundene Ausgabe)

von Shahram Dustdar, Harald Gall, Manfred Hauswirth "Ein wichtiger Aspekt von Software-Architektur ist, dass sie eine Abstraktion darstellt, die zur Reduktion der Komplexität in der Veranschaulichung eines Software-Systems wesentlich beitragt ..." (mehr)

(2 Kundenrezensionen)

Preis: EUR 39,95 Kostenlose Lieferung. Siehe Details.

Preis: EUR 39,95 Auf Lager. Verkauf und Versand durch Amazon.de

Figure 50: Library Query Service – Screenshot

The right side of figure 50 shows how the adapted version of the product-information page for a book available in the Amazon book-store looks like. The information-box on top of the page provides a link that can be used to view the search results page returned by the online catalog of the library, as shown on the left side of figure 50.

5.2 Use cases for SOAP message processing

In this chapter we demonstrate, how edge-services can be used outside of their common area of application. The use-cases presented in the previous chapter were centered on HTML-content.

In this chapter we present OPES-services operating on SOAP-messages exchanged between SOAP-clients and servers, whereas HTTP is used as underlying transport protocol for SOAP.

Each OPES-services presented in this chapter can be seen as some kind of active SOAP-intermediary as described in [44], which does additional processing on a SOAP-message, before it is forwarded to the next SOAP node in the SOAP-message-path.

5.2.1 Test Environment

One difference between the OPES-services presented in this chapter to those presented in chapter 5.1 is, that the previously introduced services were tested on “real life” HTTP request- and response-messages sent to or received from real web-servers available on the Internet.

For the services we present in this chapter, except the Google-Key-Insertion-Service, we had to setup a test environment containing a SOAP-server on which we have deployed our own SOAP services, needed to test the functionality of the OPES-services.

We decided to use the Java library jSoapServer [60] as basis of our SOAP-server. jSoapServer is itself based on Apache Axis [61] and implements a multi-threaded SOAP server that can be easily integrated into existing Java applications, to extend them with a SOAP API or used as stand-alone server-application.

For our test-environment, we have bundled jSoapServer with the Apache WSS4J [62] library. WSS4J is an implementation of the OASIS WS-Security standard and is used by our example-services to insert security tokens into SOAP messages or to sign them.

Additionally we have created custom deployment-files for our SOAP-services. The used deployment files can be found in the appendix.

5.2.2 Google Key Insertion service

This OPES-service is an example for a request-modification services. It intercepts a SOAP-message that is sent from a SOAP-client to the Google SOAP Search API [55] and inserts the Google license key for the given proxy user into the corresponding XML element of the SOAP-request-message body. Thereafter the adapted message is forwarded to the recipient.

Used Libraries:

The OPES-service uses a Java XML parser to read the SOAP-request-message as XML document. Furthermore it uses Apache Xalan and an Xpath expression to find the XML element whose value should be replaced with the Google license key.

IRML Rule:

Figure 51 shows the IRML rule used to trigger the OPES-service.

```
<rule processing-point="1">
  <property name="request-uri" context="req-msg"
   matches="http://api.google.com/search/.*">
    <execute>
      <service name="GoogleKeyInsertion">
        <uri>opes://callout:8071/org.opes4j:GoogleKeyInsertion</uri>
      </service>
    </execute>
  </property>
</rule>
```

Figure 51: Google Key Insertion Service – IRML Rule

Adapted Request Message:

Figure 52 shows how the adapted SOAP message body looks like. The inserted key is marked dark-gray.

```
<SOAP-ENV:Body>
  <ns1:doGoogleSearch
   xmlns:ns1="urn:GoogleSearch"
   SOAPENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <key xsi:type="xsd:string">012345678901234567890123456789</key>
    <q xsi:type="xsd:string">test</q>
    <start xsi:type="xsd:int">0</start>
    <maxResults xsi:type="xsd:int">15</maxResults>
    <filter xsi:type="xsd:boolean">true</filter>
    <restrict xsi:type="xsd:string" />
    <safeSearch xsi:type="xsd:boolean">true</safeSearch>
    <lr xsi:type="xsd:string" />
    <ie xsi:type="xsd:string">latin1</ie>
    <oe xsi:type="xsd:string">latin1</oe>
  </ns1:doGoogleSearch>
</SOAP-ENV:Body>
```

Figure 52: Google Key Insertion Service – Adapted SOAP request body

The OPES-services uses the XPath expression shown in figure 53 to locate the XML key element whose value then is replaced with the Google key of the given proxy user.

```
//Body/doGoogleSearch/key/text()
```

Figure 53: Google Key Insertion Service – XPath expression

5.2.3 SOAP Session Mapping Service

This service is an example for a request- and response-modification service. It is triggered the first time when a SOAP-client sends a SOAP-request-message to a specific Web-Service endpoint and the second time when the contacted SOAP-server sends the SOAP-response-message back to the client. The purpose of the OPES-service is to convert a HTTP-cookie-based SOAP-session into a SOAP-header-based session.

The OPES-service intercepts the SOAP-request-message of the client, extracts the session ID contained in the HTTP `Cookie`-header, inserts a new `sessionID` SOAP-header-block into the request-message envelope and stores the session ID in it. Thereafter the HTTP `Cookie`-header is removed and the adapted message is delivered to the SOAP-server. When the SOAP-server returns its response, the SOAP-message is intercepted, the session ID is extracted from the `SessionID` SOAP-header-block and copied into a HTTP `Set-Cookie`-header. Thereafter the SOAP-header-block is removed and the message is delivered to the SOAP-client.

The SOAP-service, which we have used for testing, was deployed on our adapted jSoapServer using the deployment-file as shown in figure A.3. The service-class needs to be deployed with “`Session`” scope. Additionally a special `SimpleSessionHandler` class, provided by Apache Axis, is required for session maintenance.

Although the use of sessions in a Web Application is nothing new, there is no standardised session maintenance mechanisms in SOAP [63]. However, many SOAP-implementations support the use of HTTP-Cookies for session maintainance. This OPES-service is a good example how an intermediary OPES-service can increase the interoperability between different SOAP implementations. It allows SOAP-clients that are only capable to manage SOAP-sessions via HTTP-Cookies to contact a SOAP-server that requires the usage of SOAP-header based sessions.

IRML Rule:

The following figure shows the IRML rules used to trigger the service.

```
<rule processing-point="1">
  <property name="request-uri" context="req-msg" matches="http://soapServer:8090/Session">
    <property name="Cookie" context="req-msg" matches="JSESSIONID.*">
      <execute>
        <service name="SOAP Session Mapping Service">
          <uri>opes://callout:8071/org.opes4j:SessionMapper</uri>
        </service>
      </execute>
    </property>
  </property>
</rule>

<rule processing-point="3">
  <property name="request-uri" context="req-msg" matches="http://soapServer:8090/Session">
    <property name="Cookie" context="req-msg" matches="JSESSIONID.*">
      <execute>
        <service name="SOAP Session Mapping Service">
          <uri>opes://callout:8071/org.opes4j:SessionMapper</uri>
        </service>
      </execute>
    </property>
  </property>
</rule>
```

Figure 54: SOAP Session Mapping Service – IRML Rule

Unlike the other OPES example-services, this services requires two IRML rules, one for service-execution-point one and another one for service-execution-point three. I.e. the first rule triggers the service for incoming client-request-messages to the given Web-Service endpoint and the second for the response-message generated by the server as result of the request-message processing.

Original Request Message:

Figure 55 shows the SOAP-request-message as intercepted from the client. The HTTP-`Cookie` header, which contains the session ID, is marked dark-gray.

```
POST http://soapServer:8090/Session HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: soapServer:8090
SOAPAction: ""
Cookie: JSESSIONID=-805620543070357845

<?xml version="1.0" encoding="UTF-8"?>
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
     <ns1:test
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://DefaultNamespace"/>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 55: SOAP Session Mapping Service – Original SOAP request message

Adapted Request Message:

Figure 56 shows the SOAP-request-message after it was modified by the OPES-service. The inserted SOAP-header block is marked dark-gray.

```
POST http://soapServer:8090/Session HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: soapServer:8090
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
     <ns1:sessionID
      soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
      soapenv:mustUnderstand="0" xsi:type="soapenc:long"
      xmlns:ns1="http://xml.apache.org/axis/session"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        -805620543070357845
     </ns1:sessionID>
  </soapenv:Header>
  <soapenv:Body>
     <ns1:test
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://DefaultNamespace"/>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 56: SOAP Session Mapping Service – Adapted SOAP request message

Original Response Message:

Figure 57 shows the original SOAP-response-message returned by the server as result of the SOAP-request-message processing. The SOAP-header-block, which is used by the SOAP-server to maintain the SOAP session, is marked dark-gray.

```
HTTP/1.0 200 OK
Server: jWssSoapServer
Date: Thu, 01 Mar 2007 15:55:46 GMT
X-Powered-By: jSoapServer 0.2.9
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
     <ns1:sessionID soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
      soapenv:mustUnderstand="0" xsi:type="soapenc:long"
      xmlns:ns1="http://xml.apache.org/axis/session"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        -805620543070357845
     </ns1:sessionID>
  </soapenv:Header>
  <soapenv:Body>
     <ns2:testResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns2="http://DefaultNamespace">
        <testReturn xsi:type="xsd:string">test 2</testReturn>
     </ns2:testResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 57: SOAP Session Mapping Service – Original SOAP response message

Adapted Response Message:

Figure 58 shows the modified SOAP-response-message as generated by the OPES-service. The inserted HTTP-Header is marked dark-gray.

```
HTTP/1.0 200 OK
Server: jWssSoapServer
Date: Thu, 01 Mar 2007 15:55:46 GMT
X-Powered-By: jSoapServer 0.2.9
Content-Type: text/xml; charset=utf-8
Set-Cookie: JSESSIONID=-805620543070357845;Path=/Session

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
     <ns2:testResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns2="http://DefaultNamespace">
        <testReturn xsi:type="xsd:string">test 2</testReturn>
     </ns2:testResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 58: SOAP Session Mapping Service – Adapted SOAP response message

5.2.4 WSS UserToken Insertion service

This OPES-service is a request modification service. It is triggered when a SOAP-clients sends a SOAP-request-message to a specific Web-Service endpoint. It intercepts the SOAP-request-message and inserts an `UserToken` header, as defined in the OASIS Web Service Security (WSS) specification, into the SOAP header block. Thereafter the adapted message is forwarded to the remote SOAP-server.

Which `UserToken` header is inserted depends on the HTTP `Authorization-`header located in the original request-message. The OPES-services uses this header to determine which `UserToken` to use. I.e. a mapping is done between a HTTP-header-based to a WSS-based user-authentication.

The SOAP-service, which we have used for testing, is deployed on our adapted jSoapServer using the deployment-file shown in figure A.4.
The validation of the inserted SOAP `UserToken-`header is done by a special Apache Axis `handler` class provided by the WSS4j project. The `handler` executes a callback method provided by our SOAP-service-class to compare the received with the expected user-name and password pair. If the authentication was successful, the actual SOAP-service is invoked, otherwise a SOAP-fault is returned to the SOAP-client.

The advantage of this OPES-service is, that it enables old SOAP-clients, which do not support the WSS standard, to interact with SOAP-services that require various WSS-headers to be set.

Used Libraries:

The OPES-service uses Apache Axis [61] to convert the received SOAP envelope into an XML document. Thereafter the WSS4j library is used to insert the security headers and to serialize the modified XML-document to the java `OutputStream`.

IRML Rule:

Figure 59 shows the IRML rule that is used to trigger the OPES service.

```
<rule processing-point="1">
  <property name="request-uri" context="req-msg"
   matches="http://soapServer:8090/UserToken">
    <property name="Authorization" context="req-msg" matches=".*">
       <execute>
          <service name="SOAP WS-Security UserToken Insertion Service">
             <uri>opes://callout:8071/org.opes4j:UserToken</uri>
          </service>
       </execute>
    </property>
  </property>
</rule>
```

Figure 59: WSS UserToken Insertion Service – IRML Rule

The OPES-services is only triggered if the SOAP-request is sent to a specific Web-Service endpoint (in the above case to our test SOAP-service) and the SOAP-client has set the HTTP `Authorization` header properly.

Adapted Request Message:

Figure 60 shows the adapted SOAP-request-message returned by the OPES-service. The dark-gray section shows the WSS-Header that was inserted by the OPES-service.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <SOAP-ENV:Header>
    <wsse:Security
      SOAP-ENV:mustUnderstand="true"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/←
                 oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <wsse:UsernameToken wsu:Id="UsernameToken-18122243"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/←
                  oasis-200401-wss-wssecurity-utility-1.0.xsd">
        <wsse:Username>
          userName
        </wsse:Username>
        <wsse:Password
          Type="http://docs.oasis-open.org/wss/2004/01/←
               oasis-200401-wss-username-token-profile-1.0#PasswordText">
          userPwd
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    <ns1:test SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:ns1="http://DefaultNamespace"/>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 60: WSS UserToken Insertion Service – Adapted SOAP request message

5.2.5 WSS Signature service

This service is a request-modification service that is triggered if a SOAP-call is sent to a specific Web-Service endpoint. It intercepts the original SOAP-request-message, extracts user-information from the HTTP `Authorization` header, uses this information to load the users certificate from a Java keystore[7], and signs the SOAP-request-message body with the certificate according to the OASIS Web Service Security standard. Thereafter the adapted SOAP-request is forwarded to the SOAP-server.

Used Libraries:

Similar to the UserToken-service described in the previous chapter, this service uses Apache Axis [61] to convert the received SOAP envelope into an XML document. Thereafter the WSS4j library is used to sign the SOAP-body and to insert related security headers (e.g. containing information about the used signature method, the signature value itself, information about the certificate, etc.) into the SOAP-header. Thereafter the XML document is canonicalized in a way that the serialization does not break the signature[8].

IRML Rule:

Figure 61 shows the IRML rule that is used to trigger the OPES service.

```
<rule processing-point="1">
  <property name="request-uri" context="req-msg"
   matches="http://soapServer:8090/Signature">
     <property name="Authorization" context="req-msg" matches=".*">
        <execute>
          <service name="SOAP WS-Security Signature Service">
             <uri>opes://callout:8071/org.opes4j:SignatureService</uri>
          </service>
        </execute>
     </property>
  </property>
</rule>
```

Figure 61: WSS Signature Service – IRML Rule

The OPES-services is only triggered if the SOAP-request is sent to a specific endpoint and if the request-message contains a HTTP `Authorization` header, which is needed to determine the certificate to use for signing the message.

Adapted Request Message:

Figure 62 shows the adapted SOAP-request-message returned by the OPES-service. The dark-gray lines shows those WSS-related parts that are inserted by the OPES-service.

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Header>
  <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/←
   oasis-200401-wss-wssecurity-secext-1.0.xsd"
   soapenv:mustUnderstand="1">
     <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
      Id="Signature-7640118">
```

[7] a special password-protected database that holds keys and certificates

[8] see the W3C Recommendation “Canonical XML” [64] for details

```
      <ds:SignedInfo>
        <ds:CanonicalizationMethod
         Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
        </ds:CanonicalizationMethod>
        <ds:SignatureMethod
         Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1">
        </ds:SignatureMethod>
        <ds:Reference URI="#id-5585368">
          <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
            </ds:Transform>
          </ds:Transforms>
          <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
          </ds:DigestMethod>
          <ds:DigestValue>e+AIGj4KA/vx9GwstEAsrvhgIKo=</ds:DigestValue>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>
        MvxA/HqEWNcI4ytnGIPmDASDUov7kYnJyZnO5H5PPOf7wYLvECliNYV59←
        SPOM1Jt67ESWQDvxTnwEfGZW/xuGnr7+tAMRAdH1nFHJDuK0HdxTNRmcU←
        NeVtSdvV7+vfZMXBM8oIohOAYMClU8RoMn7+fCEJ+GKtIVj7XllrItFR0=
      </ds:SignatureValue>
      <ds:KeyInfo Id="KeyId-22619584">
        <wsse:SecurityTokenReference xmlns:wsu="http://docs.oasis-open.org/←
         wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
         wsu:Id="STRId-6037166">
          <ds:X509Data>
            <ds:X509IssuerSerial>
              <ds:X509IssuerName>
                CN=Testclient 01,OU=wss signature test,O=opes4j,←
                L=vienna,ST=vienna,C=AT
              </ds:X509IssuerName>
              <ds:X509SerialNumber>1165584748</ds:X509SerialNumber>
            </ds:X509IssuerSerial>
          </ds:X509Data>
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
    </ds:Signature>
  </wsse:Security>
</soapenv:Header>
<soapenv:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/←
              oasis-200401-wss-wssecurity-utility-1.0.xsd"
              wsu:Id="id-5585368">
  <ns1:test xmlns:ns1="http://DefaultNamespace"
   soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  </ns1:test>
</soapenv:Body>
</soapenv:Envelope>
```

Figure 62: WSS Signature Service – Adapted SOAP request message

It must be mentioned that figure 62 depicts a pretty-printed, human-readable form of the message returned by the OPES-service. But in practice the XML is canonicalized in a way that the serialization does not break the signature (see [64] for details).

5.3 A combination of Content-Services and Web-Services

5.3.1 Bookmark Lookup Service

The example presented in this chapter differs from those in the previous chapters as it combines a content-oriented service with a Web-Service to provide a value-add to the content for the user.

If an user uses Google to search for a given keyword, an OPES-service is triggered when Google sends back a HTTP-response-message, which contains the search results. The service intercepts the response-message and adapts the contained HTML-document.
First a HTML `script` tag is inserted into the HTML-header of document that points to a special Javascript file located on a web-server. Furthermore a Javascript function-call is added to the `onload` event of the HTML-body. Thereafter the modified HTML document is forwarded to the user.

Once the user's browser has received the modified document, the `onload` event is triggered and the inserted Javascript function is called. The Javascript function loops through the search results returned by Google to determine whether the user has bookmarked some of the displayed search-result links in the YaCy bookmarking system.

YaCy [65] is basically a distributed web search engine, but also provides various communication tools, such as a build-in Blog and Wiki as well as an integrated bookmarking system, to the peer owners. Many of the provided functionalities are accessible via a SOAP API.

To test whether a given link is bookmarked, the Javascript invokes a specific Web-Service-function provided by the YaCy SOAP-API and passes the link as parameter to it. If the Web-Service response indicates that the link is bookmarked, the Javascript displayes a list of tags below the link in the browser, containing all tags that were assigned by the user at bookmarking time.

IRML Rule:

The following figure shows the IRML rule that is used to trigger the OPES-service.

```
<rule processing-point="1">
  <property name="request-uri" context="req-msg"
   matches="http://www.google.at/search\?.*&?q=.*">
     <property name="User-Agent" context="req-msg" matches="Firefox/.*">
       <execute>
          <service name="Bookmark Lookup Service">
             <uri>opes://callout:8071/org.opes4j:BookmarkService</uri>
          </service>
       </execute>
     </property>
  </property>
</rule>
```

Figure 63: Bookmark Lookup Service – IRML Rule

The Javascript code described above only works with the Firefox web-browser, therefore the IRML rule shown above is only triggered if the HTTP User-Agent header starts with the String “`Firefox`”.

Adapted Application Message:

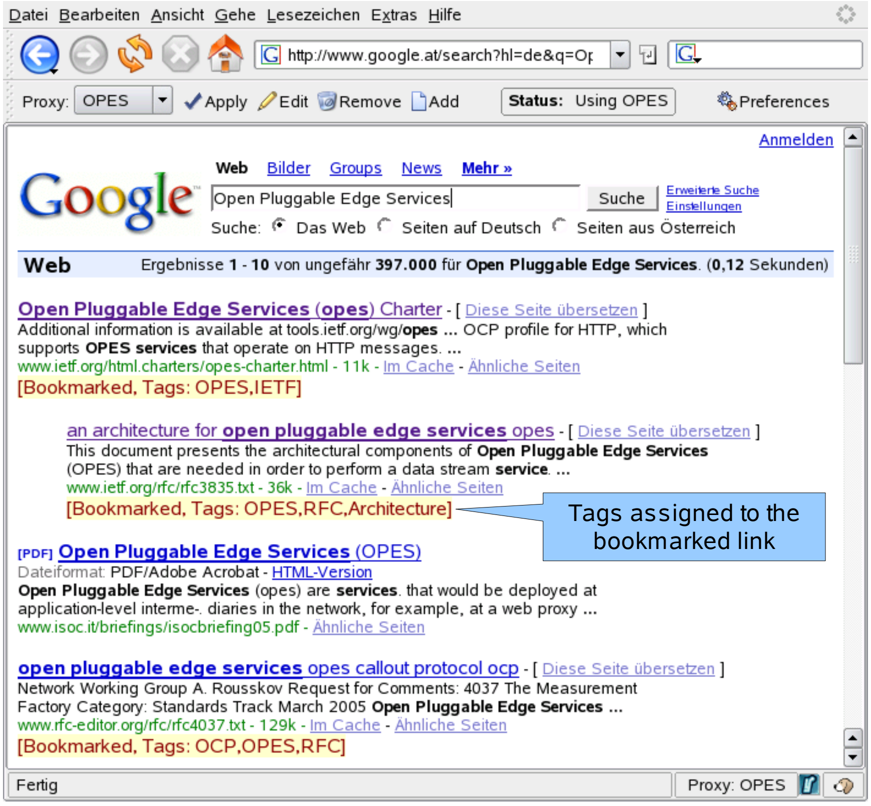
Figure 64 shows the modified HTML document as it generated by the OPES callout-service. How the referenced Javascript file looks like is depicted in figure A.6.

```
<html>
   <head>
      <meta http-equiv="content-type" content="text/html; charset=UTF-8">
      <title>Open Pluggable Edge Services – Google-Suche</title>
      <script type="text/javascript" src="http://server/bookmarks.js"></script>
[...]
</head><body onload="bookmarkTest('userAuth');[...]
```

Figure 64: Bookmark Lookup Service – Adapted Application Message

Screenshot:

Figure 65 shows how the Javascript function modifies the Google search-result page, if a given link was exposed as bookmarked.


Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe
http://www.google.at/search?hl=de&q=Op
Proxy: OPES
Apply
Edit
Remove
Add
Status: Using OPES
Preferences
Anmelden
Web Bilder Groups News Mehr »
Google
Open Pluggable Edge Services
Suche
Erweiterte Suche
Einstellungen
Suche: ● Das Web ○ Seiten auf Deutsch ○ Seiten aus Österreich
Tags assigned to the bookmarked link
Fertig
Proxy: OPES

Web Ergebnisse **1** - **10** von ungefähr **397.000** für **Open Pluggable Edge Services**. (**0,12** Sekunden)

Open Pluggable Edge Services (opes) Charter - [Diese Seite übersetzen]
Additional information is available at tools.ietf.org/wg/**opes** ... OCP profile for HTTP, which supports **OPES services** that operate on HTTP messages. ...
www.ietf.org/html.charters/opes-charter.html - 11k - Im Cache - Ähnliche Seiten
[Bookmarked, Tags: OPES,IETF]

an architecture for **open pluggable edge services** opes - [Diese Seite übersetzen]
This document presents the architectural components of **Open Pluggable Edge Services** (OPES) that are needed in order to perform a data stream **service**. ...
www.ietf.org/rfc/rfc3835.txt - 36k - Im Cache - Ähnliche Seiten
[Bookmarked, Tags: OPES,RFC,Architecture]

[PDF] **Open Pluggable Edge Services** (OPES)
Dateiformat: PDF/Adobe Acrobat - HTML-Version
Open Pluggable Edge Services (opes) are **services**. that would be deployed at application-level interme-. diaries in the network, for example, at a web proxy ...
www.isoc.it/briefings/isocbriefing05.pdf - Ähnliche Seiten

open pluggable edge services opes callout protocol ocp - [Diese Seite übersetzen]
Network Working Group A. Rousskov Request for Comments: 4037 The Measurement Factory Category: Standards Track March 2005 **Open Pluggable Edge Services** ...
www.rfc-editor.org/rfc/rfc4037.txt - 129k - Im Cache - Ähnliche Seiten
[Bookmarked, Tags: OCP,OPES,RFC]

Figure 65: Bookmark Lookup Service - Screenshot

6 Evaluation and further work

6.1 Evaluation

In the previous chapter we have successfully demonstrated the functionality of the prototype-implementation by presenting various example services.

In this chapter we outline problems that arose during the implementation of the prototype and inconsistencies we have found in the used standards and technologies.

6.1.1 OPES Callout Protocol

In this section we outline problems and contradictions we have detected regarding the OCP-core-protocol- and OCP-HTTP-profile specification.

Kept Parameter:

As mentioned in chapter 2.2.5 and 2.2.6, and as reported by the author of this thesis on the OPES WG mailing-list [66], there is a syntactical error in the definition of the "`Kept`" parameter of an OCP `DUM`-message. The PETDM[1] definition of the `DUM`-message is depicted below.

```
DUM: extends message with {
  xid my-offset;
  [As-is: org-offset];
  [Kept: org-offset org-size ];
  [Modp: modp];
} and payload;
```

Figure 66: Evaluation – DUM message definition (based on [15])

As shown above the "`Kept`" parameter is defined as a named parameter that consists of two fields. However, this definition doesn't follow the syntactical rules for a valid message format. As defined by the ABNF shown in the figure 11, the value of a named parameter can be a `struct`, `list` or `atom`. But the value of the "`Kept`" parameter is none of these three types.

This is especially a problem because as defined in the OCP-core specification: "Messages violating formatting rules are, by definition, invalid" [15]. And furthermore: "Unless explicitly allowed to do otherwise, an OCP agent MUST terminate the transaction if it receives an invalid message with transaction scope" [15].

Thus, if the OPES-processor utilizes the "`Kept`" parameter as defined in 66, but the callout-server does not, the transaction is inevitably closed with an error-message. Thus if the transmission of the "`Kept`" parameter can not be disabled on the OPES-processor, the two OCP-agents could never cooperate successfully.

A solution for this problem would be to represent the value of the "`Kept`" parameter as `list` or `struct` containing two anonymous parameters.

TE / CE Messages:

Another problem in the specification reported by the author of this thesis on the OPES-WG

[1] Protocol Element Type Declaration Mnemonic, which is used by the OCP specification for a formal declaration of protocol element types

mailing-list [67] is the ambiguous definition of the OCP TE-message.

As defined in the OCP-core specification: “A callout server MUST maintain the state [author's note: of the OCP-transaction] until it receives a message indicating the end of the transaction or until it terminates the transaction itself” [15]. Furthermore: “A transaction ends with the first Transaction End (TE) message sent or received, explicit or implied” [15]. Finally, “the recipient must terminate the transaction when the xid parameter [author's note: the transaction-ID] in a Data Use Mine (DUM) message refers to an unknown or already terminated OCP transaction” [15].

Thus, if both OCP-agents follow this rule and e.g. a callout-service is performed that just appends data at the end of the original application-message, the callout-server will never have the chance to deliver the entire adapted application-message back to the OCP-processor, due to the fact that the processor freed all state information associated with the transaction, after it has sent the TE-message.

A solution for this problem would be to define that an OCP-agent must not free associated state until both OCP-agents have sent their TE-messages. Only if a sent or received TE-message reports an error, the OCP-agent should be allowed to immediately free all associated state information.

But if the usage of the TE-messages is redefined as described above, it should be reconsidered whether the usage of the CE-message should also be redefined in that way.

Transfer Encoding:

Although the OCP profile for HTTP defines a separate Application-Message-Part for the transmission of request- and response-trailers (see table 1), the specification also defines that “an OCP agent MUST NOT send transfer-encoded application message bodies” [17].

Let us assume the following scenario: An OPES-processor receives a server-response using chunked transfer-encoding. The message is decoded by the processor and forwarded to the callout-server, which applies a callout-service to it. The service modifies the response-trailer part and the server sends all application-message-parts back to the processor.

The problem that could arise is, that the HTTP-client does not support receiving of transfer-encoded messages and therefore is not capable to receive HTTP trailer-fields[2]. But what happens with the modified response-trailer-part in this case? Should it be just thrown away? The specification does not clarify this.

The OPES-processor could not just copy the trailer fields into the HTTP-header, because the adapted trailer-part is received not until the processor at least has started to deliver the response-body to the client. Of course a store-and-forward-approach could be used for message transmission, but this would contradict the idea of operating on data-flows and moreover would not be a sufficient solution for the processing of large application-messages. Furthermore the processor does not even know if it will receive a trailer part - maybe the callout-service has just thrown away the trailer - and maybe would wait for nothing.

[2] because trailer-fields requires the use of the chunked transfer-encoding

Content Encoding:

Although the OCP profile for HTTP [17] defines that a callout-server should be capable to handle content-encoded message-bodies, it does not define how the usage of content-encodings affects OCP-features such as premature-dataflow-termination or data-preservation.

Let us assume a callout-service that takes the compressed (resp. the content-encoded) body and unzips it to insert content at a specific position. After the content was inserted the service would like to use the premature-dataflow-termination feature to get out of the loop. But how does the service know the exact mapping between the uncompressed data-stream used by it and the compressed data-stream used by the OPES-processor? Does such a mapping exist at all, and for all commonly used content-encodings?

Furthermore neither the OCP-core- nor the HTTP-profile-specification defines that the data-preservation-cache is application-protocol-dependent. Thus it can be assumed that the original-application-message-data is just stored as is in the cache. But in this case, the same problem arises as described for the premature-dataflow-termination feature.

A possible solution could be do decode the content before it is stored into the preservation-cache and forwarded to the callout-server. But then the cache could not be implemented application-protocol-agnostic. Additionally the processor would need to continue decoding the original-data-stream, even if the service got out of the loop, because of the mapping problem described above. In the worst case the processor would need to decode and thereafter immediately to re-encode the content, if the client has requested to receive the content using content-encoding.

Service Group Creation:

As defined in the OCP-core specification [15] service-group have connection scope. Thus the easiest way to manage service-groups would be to create them once the OCP-connection was established and to destroy them immediately before the connection is shutdown.

But as our case-study has shown, many service-applications need additional parameters to perform there service. These parameters could be related to the current user or even to the current application-message. And because the OCP specification has not defined ways to pass parameters to services, except by specifying them during service-group creation, the OPES-processor needs to misuse `SGC` message for parameter passing purpose.

Additionally the OPES-processor does not know any details about the callout-services and therefore could not know which parameters are needed by a given service or which scope these parameters have (i.e. user-specific, message-specific, etc. parameters). Thus the OPES-processor can not determine whether an existing service-group needs to be re-created but with differing parameter values.

Furthermore a data-dispatcher may support policy-rules that decide at runtime, which services to applied to a given application-message. In this case, the composition of a service-group is not predefined but determined at runtime.

For these reasons, `SGC`-messages and corresponding `SGD`-messages must be sent very frequently, in the worst case for each application-message to process. Furthermore if the negotiated features (e.g. the HTTP request- and response-profile) do not have connection-scope, but are assigned to specific service-groups[3], the negotiation must be repeated each time the service-group is re-created. Alternatively a separate OCP-connection can be used for each used used profile, but then the OCP-processor needs to store which profile is assigned to which connection.

The high frequency of group-create and -destroy operations may cause further problems. As defined in the specification the callout-server must immediately terminate the connection if it can not create a service-group, e.g. because a service-identifier is unknown. Thus only a single configuration-mistake in a policy-rule-file, e.g. the mistyping of a service-identifier, could cause the whole connection to shutdown. And this would also cause unexpected termination of currently active transactions.

Service Parameters:

The missing definition how parameters can be sent to a callout-services causes further problems.

As defined in [11], information passing between services should be possible. But because of a missing standardization how this should be done, it would be hard to bring together components provided by different vendors, especially if parameter-passing is required between callout-services hosted on different callout-servers.

Furthermore there is currently no way to specify the data-types of these parameters. But if an used callout-service is independent of the callout-protocol (e.g. a service taken over from an other service-execution environment), there should be a possibility to describe how a mapping between the parameter-type delivered by the OPES-processor and the parameter-type required by the service could be done.

Chaining of Services:

A OPES-callout-server supports the “chaining” of OPES-service-applications using service-groups. I.e. multiple services can be bundled together to a service-group and perform their services sequentially on the same application-message.

But the specification does not describe how the chaining of services affects the possibility to use the OCP data-preservation- or premature-termination features. But it does not answer the question what happens if a service located in the middle of the chain would like to get out of the application-message-processing loop[4]? Respectively, what happens if the service requests the OPES-processor to use data from its preservation-cache, but a preceding service just required to adapt this portion of data?

Multiple Application Messages:

As specified in the OCP-core-specification “a transaction is associated with a single original and a single adapted application message. OCP Core extensions may extend transaction scope to more application messages” [15]. But because the HTTP-profile does not extend transactions in that way, a callout-service operating on HTTP-messages is currently not allowed to generate multiple HTTP-messages as callout-response.

We think this would be a quite useful extension. For instance, a virus-scanning services could generate `100`-Continue HTTP-messages to avoid a timeout in the HTTP-connection to the

[3] e.g. if different OCP profiles should be used on the same connection. This is only possible if the profiles are negotiated per service-group.

[4] i.e. both, the original-dataflow and the adapted-dataflow should be terminated (see chapter 2.2.5)

client. Or a callout-service could duplicates request-message to send them transparently to two web-servers e.g. for failure safety reason.

Protocol Converter:

Currently a negotiated OCP profile is used for both, the original-dataflow and the adapted-dataflow. Although this is a sufficient solution for most scenarios, it also prohibits the creation of callout-services acting as some kind of protocol converter or gateway.
For example think of an OPES-service acting as a SOAP-intermediary which accepts incoming SOAP-messages using HTTP as transport-protocol and converts them into SOAP-messages using the SOAP SMTP-Transport-Binding.

6.1.2 Proxylet API

In this chapter we describe the problems we have encountered while trying to combine OPES with the proxylet API described in chapter 2.3.1.

Response Modification Service:

As mentioned in chapter 4.3, the proxylet specification seems to assume that a response-modification service always gets the HTTP-request-message as input-parameter in addition to the mandatory HTTP-response-message. But because the OCP HTTP-response profile specifies that all HTTP-request-message parts are auxiliary and are only transferred if negotiated, this may not be the case.

The problem is that a missing request-object would prohibit the proxylet to use functions provided by the object. The request-object does not only provide request-message-specific functions but also provides functions to register the proxylet as event listener or to fetch proxylet-specific parameters (e.g. those specified by a rule-author in the rule-file). Thus a missing request-object could cause unexpected problems.

In our prototype implementation we have solved the problem by moving the problematic functions to the `ProxyletContext` object, which is always accessible no matter which application-messages are available.

Application Message Parts:

As mentioned above, the OCP HTTP-response profile differentiates between mandatory and auxiliary application-message parts. As mentioned earlier, problems could arise if the whole request-message is missing. Additionally it is also possible that some but not all parts of the request-message are transferred to the callout-server. In this case we have the problem that a proxylet is unable to determine which parts were omitted.

A solution for this problem would be to add information into the deployment-description-file of a proxylet, indicating which auxiliary parts the proxylet requires. Additionally the proxylet API should be extended with additional functions allowing a proxylet to query which application-message-parts are available and which are not.

Another problem of the proxylet API is its missing support for HTTP-trailers. Currently the API does not define appropriate functions to get, set, add or remove trailer fields, thus the API should be extended with them.

Proxylet Status Code:

As defined in [19], a proxylet must return a `ProxyletStatus` object containing a status-code that indicates the overall message-processing status. Many of the available status-codes are strongly related to HTTP status-codes. For instance if a proxylet returns the status-code `PROXYLET_INTERNAL_SERVER_ERROR`, the error should be reported to the user via a corresponding HTTP-message containing the `500` status-code.

The problem with these status-codes is, that they are reported at the end of service-execution. But the consideration of them would require that a store and forward approach is used, because otherwise the HTTP-header was already sent out, when the error is detected.

But a store-and-forward-approach is undesired for an OPES-system, because it significantly increases the message-delivery latency. Therefore we have decided to not rely on these proxylet status-codes. In our prototype-implementation, these status-codes are only taken into consideration if the proxylet has not written out any application-message-data so far[5].

OPES Features:

The proxylet API was originally intended to be used on intermediary devices for the local execution of proxylet services. Thus it was assumed that the entire original application-message-data is “flowing” through a proxylet, and a proxylet only prematurely terminates the processing of a message if an error occurs.

But for our prototype implementation we have decided not to distinguish between proxylets that are executed in a local- or a remote proxylet-execution-environment. I.e. a single proxylet should be executable in both locations and should not need to take care of whether it is executed locally or remotely.

However if sending data over a callout-protocol, everything that helps to reduce the message-data round-trip time is desirable. Thus proxylets acting as callout-services should have the possibility to use OCP features such as data-preservation or premature-dataflow-termination.

[5] as described in chapter 4.3, if a proxylet requests the `OutputStream` via the `getPayloadOutputStream` method, this causes the stream to transparently send an AMS-message and thereafter to send a DUM-message containing the HTTP-Head via the OCP-connection to the OPES-processor.

6.2 Further work

Proxylet Execution Environment:

As mentioned in the previous chapter, the proxylet-execution-environment – which was implemented by the research-prototype – should be extended with additional functionality to allow remotely executed proxylets to use OCP-features such as data-preservation or premature-dataflow-termination. Furthermore response-modification proxylets need to have the possibility to indicate which auxiliary parts of a request-message they would like to receive.

IRML Subsystem for SOAP:

As described in chapter 2.4, IRML can be extended with additional subsystems. This subsystems can introduce new properties that are evaluated by the subsystem-implementation at runtime, and could re-define the method that is used to test the matching of a property value to a rule condition.

An interesting idea would be to design and implement an IRML-subsystem for SOAP. The subsystem could allow to use Xpath-expressions as property-names and regular-expressions as matching conditions. An example is shown below.

```
<property name=”//Header/sessionID/text()” matches=”-805620543070357845”
  sub-system=”SOAP”>
  <!-- actions to trigger -->
</property>
```

The example IRML-rule could be used to trigger the execution of an OPES-services, if an intercepted SOAP-message contains a `sessionID` SOAP-header-block with a given value.

OCP Profile for SOAP:

The OCP-protocol-core can be extended with additional profiles, describing how to transport specific application-protocol-messages using OCP.

In combination with an IRML subsystem for SOAP, it would be nice to have an OCP profile for SOAP. Similar to the OCP profile for HTTP, the SOAP profile could split a SOAP-message into application-message-parts, such as the SOAP header, the SOAP body and multiple SOAP attachment parts.

This would have the advantage that a callout-server, which is not interested in receiving all parts of a SOAP message, could define precisely which parts it is interested in. Lets assume a callout-service that just operates on SOAP headers and therefore is not interested in receiving the SOAP body, or a service that just operates on SOAP attachments and therefore does not require to receive the SOAP envelope.

Apache Axis Transport for OCP:

The Apache Axis library, which we have used as SOAP-engine for our SOAP-related callout-services, contains a transport framework that makes it easy to plug in new transport handlers used to transport a SOAP-message via a specific transport-protocol. Currently handlers exist for HTTP, SMTP and JMS. In combination with an OCP profile for SOAP, it would be interesting to have an Apache Axis transport handler for OCP.

7 Summary and Conclusion

In the course of this thesis, a prototype implementation of the Open-Pluggable-Edge-Service (OPES) framework was developed, which follows the proposed standards of the IETF OPES working-group (OPES WG). The prototype system consists of an OPES-processor, an OPES-data-dispatcher, an OPES-callout-server, a service-execution-environment, an implementation of the OPES-callout-protocol (OCP) and the OCP profile for HTTP.

The OPES-WG neither standardized a rule-language for the definition of policy-rules, nor an execution-environment for the execution of edge-services. Instead some requirements were formulated that need to be fulfilled by a concrete implementation of the policy-architecture and the service-execution-environment. We decided to use the Intermediary-Rule-Markup-Language (IRML) as language for the definition of policy-rules and the concept of proxylets to build up a service-execution-environment for OPES-services on our prototype callout-server.

Based on the research-prototype, a case study was conducted to demonstrate the functionality of the prototype-implementation, to test the practical suitability of the OPES-framework and -protocol, and to see whether edge-services can be used outside of their common area of application for the intermediate processing of SOAP-request and -response messages.

As the case-study pointed out, the prototype is well suited to perform common edge-services such as content-transformation- or -generation services on HTTP-messages, exchanged between a HTTP-client and -server. Even more, it was shown that it is quite easy to build OPES-services acting as an active SOAP-intermediary. Furthermore the case-study demonstrates that the combination of OPES with IRML and proxylets fits together quite well, even though an extension of the used proxylet API would be required to allow proxylets to benefit from various OCP-optimization-features such as data-preservation or premature-dataflow-termination.

The main advantage of the OPES-framework and protocol-suite is its universality and flexibility and the possibility to extend the generic callout-protocol-core with special application-protocol profiles. The standardization of the callout-protocol enables the integration and re-use of existing services provided by different vendors, without the need to adapt neither the intermediaries nor the service-applications.

Nevertheless we have also pointed out some inconsistencies in the specification of the OPES callout-protocol, which affect the interoperability between OCP-agents and could even prevent OPES-processors and callout-servers of different vendors to cooperate successfully. Furthermore there are some unresolved issues regarding the usage of OCP-optimization-features in combination with service-chaining or HTTP transfer- and content-encodings, or how parameter-passing could be done between an OPES-processor and callout-services.

The OPES working-group needs to perform further standardization work to correct the detected inconsistencies and should perform interoperability-tests on existing prototypes to gain experience regarding the practical suitability of the OPES-framework and protocol.

At the moment no public available OPES-framework implementation exists, therefore providers have no motivation to switch from their currently used specific solutions to the OPES framework. Therefore more prototyping is required to demonstrate the benefits of the OPES-architecture to convince providers to adopt their systems.

A Appendix

A.1 Case Study

Cache View Service:

The following figure shows how the Java library jTidy [57] can be used to covert a HTML document into XHTML format.

```
// create a new jTidy instance
Tidy tidy = new Tidy();

// tell jTidy to return XHTML, to correct mistakes in the HTML code and to
// suppress warnings
tidy.setXmlOut(true);
tidy.setMakeClean(true);
tidy.setQuiet(true);
tidy.setShowWarnings(false);
tidy.setCharEncoding(Configuration.UTF8);

ByteArrayInputStream bais = null;
bais = new ByteArrayInputStream(htmlString.getBytes("UTF-8"));

// convert HTML to XHTML
Document theDom = tidy.parseDOM(bais, null);

// blose buffer
bais.close();
```

Figure A.1: Cache View Service - HTML to XHTML conversion using jTidy

News Extraction Service:

The following figure shows the XSLT stylesheet used by the News-Extraction-Service to convert the source XHTML document into a RSS feed.

```
<?xml version="1.0" encoding="utf-8" ?>
<?xml-stylesheet type='text/xsl' href='/rss.xsl' version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:dc="http://purl.org/dc/elements/1.1/" version="1.0">
  <xsl:template match='/html'>
     <rss version="2.0">
       <channel>
          <title><xsl:value-of select='head/title' /></title>
          <xsl:apply-templates select='body/table[2]/tbody/tr[2]/td/table/tr' />
       </channel>
     </rss>
  </xsl:template>

  <xsl:template match='tr'>
     <item>
       <title><xsl:value-of select='td[1]/b' /></title>
       <description><xsl:value-of select='td[2]' /></description>
     </item>
  </xsl:template>
</xsl:stylesheet>
```

Figure A.2: News Extraction Service – XSLT Stylesheet

WS Session Mapping Service:

```
<deployment  xmlns="http://xml.apache.org/axis/wsdd/"
             xmlns:java="http://xml.apache.org/axis/wsdd/providers/java" >
  <service name="@serviceName@" provider="java:RPC">
     <parameter name="typeMappingVersion" value="1.1"/>
     <parameter name="className" value="@serviceClassName@" />
     <parameter name="allowedMethods" value="*" />

     <!-- set the class scope to session -->
     <parameter name="scope" value="Session"/>

     <!-- define the session handlers for SOAP header based sessions -->
     <requestFlow>
        <handler type="java:org.apache.axis.handlers.SimpleSessionHandler"/>
     </requestFlow>
     <responseFlow>
        <handler type="java:org.apache.axis.handlers.SimpleSessionHandler"/>
     </responseFlow>
  </service>
</deployment>
```

Figure A.3: WS Session Mapping Service – SOAP Service Deployment File

WSS User Token Insertion Service:

```
<deployment  xmlns="http://xml.apache.org/axis/wsdd/"
             xmlns:java="http://xml.apache.org/axis/wsdd/providers/java" >
  <service name="@serviceName@" provider="java:RPC">
     <!-- default jSoapServer deployment template -->
     <parameter name="typeMappingVersion" value="1.1"/>
     <parameter name="scope" value="Request"/>
     <parameter name="className" value="@serviceClassName@" />
     <parameter name="allowedMethods" value="*" />

     <!-- special handler for WSS -->
     <requestFlow>
        <handler type="java:org.apache.ws.axis.security.WSDoAllReceiver">
           <parameter name="passwordCallbackClass" value="PasswordCallback"/>
           <parameter name="action" value="UsernameToken"/>
        </handler>
     </requestFlow>
  </service>
</deployment>
```

Figure A.4: WSS User Token Insertion Service – SOAP Service Deployment File

WSS Signature Service:

```
<deployment  xmlns="http://xml.apache.org/axis/wsdd/"
             xmlns:java="http://xml.apache.org/axis/wsdd/providers/java" >
  <service name="@serviceName@" provider="java:RPC">
     <parameter name="typeMappingVersion" value="1.1"/>
     <parameter name="scope" value="Request"/>
     <parameter name="className" value="@serviceClassName@" />
     <parameter name="allowedMethods" value="*" />

     <!-- special handler for WSS -->
     <requestFlow>
        <handler type="java:org.apache.ws.axis.security.WSDoAllReceiver">
           <parameter name="action" value="Signature"/>
           <parameter name="signaturePropFile" value="Signature.properties" />
        </handler>
     </requestFlow>
  </service>
</deployment>
```

Figure A.5: WSS Signature Service - SOAP Service Deployment File

Bookmark Lookup Service:

```
function linkTest(userAuth) {
   var a = document.getElementsByTagName("a");

   // loop through all search results
   for(var x=0;x<a.length;x++) {
      if (a[x].className == 'l') {
         // exec SOAP call
         var tags = isListed(userAuth, a[x].href);

         // if tags were found ...
         if (tags != null) {
            // set style
            var divStyle = document.createAttribute("style");
            divStyle.nodeValue = "color:darkred; background-color:#FFFFCC;";

            var div = document.createElement("span");
            div.appendChild(document.createTextNode("[Bookmarked, Tags: "
            + tags +"]"));
            div.setAttributeNode(divStyle);
            a[x].parentNode.parentNode.appendChild(div);
         }
      }
   }
}

function isListed(userAuth, url) {
   netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserRead");

   // specify SOAP service to invoke
   var serviceURI = "http://yacy:18091/soap/bookmarks";
   var method = "bookmarkIsKnown";

   // generating soap header paramaters (needed for user authentication)
   var h = new Array();
   h[0] = new SOAPHeaderBlock(userAuth,"Authorization",
         “http://http.anomic.de/header");

   // parameters to pass to the service
   var p  = new Array();
   p[0] = new SOAPParameter(url,'url');

   // initialize SOAP call
   var soapCall = new SOAPCall();
   soapCall.transportURI = serviceURI;
   soapCall.encode(0, method, serviceURI,0,null,p.length,p);

   // invoke SOAP service
   var response = soapCall.invoke();
   var responsArray = new Array();
   responsArray = response.getParameters(false,{});

   // get the result, which is the tag list associated to the bookmark
   return responsArray[0].value;
}
```

Figure A.6: Bookmark Lookup Service – javaScript

List of Figures

References

[1] B. Molina, V. Ruiz, et al. *A Closer Look at a Content Delivery Network Implementation,* Electrotechnical Conference, 2004. MELECON 2004, Proceedings of the 12th IEEE Mediterranean,2,pages 685-688, May 2004

[2] M. Gudgin, M. Hadley et al. *SOAP Version 1.2 Part 1: Messaging Framework,* http://www.w3.org/TR/soap12-part1/, June 2003

[3] OPES Working Group. *Open Pluggable Edge Services (opes) Charter,* http://www.ietf.org/html.charters/opes-charter.html, August, 16, 2006

[4] M. Day, B. Cain, et al. *A Model for Content Internetworking (CDI),* http://bgp.potaroo.net/ietf/all-ids/draft-day-cdnp-model-09.txt, November 2001

[5] G. Tomlinson, R. Chen, M. Hofmann. *A Model for Open Pluggable Edge Services,* http://standards.nortelnetworks.com/opes/non-wg-doc/draft-tomlinson-opes-model-00.txt, July 2001

[6] F. Douglis, M. F. Kaashoek. *Scalable Internet Services,* IEEE Internet Computing,5,4,pages 36-37, August 2001

[7] M. Hofmann, L. R. Beaumont. *Open Pluggable Edge Services: An Architecture for Networked Content Services,* IEEE Internet Computing,11,1,pages 67-73, February 2007

[8] S. Floyd, L. Daigle. *IAB Architectural and Policy Considerations for Open Pluggable Edge Services,* http://www.ietf.org/rfc/rfc3238.txt, January 2002

[9] A. Barbir, E. Burger, R. Chen, et al. *Open Pluggable Edge Services (OPES) Use Cases and Deployment Scenarios,* http://www.ietf.org/rfc/rfc3752.txt, April 2004

[10] A. Barbir, R. Penno, R. Chen, et al. *An Architecture for Open Pluggable Edge Services (OPES),* http://www.ietf.org/rfc/rfc3835.txt, August 2004

[11] A. Barbir, O. Batuner, A. Beck, et al. *Policy, Authorization, and Enforcement Requirements of the Open Pluggable Edge Services (OPES),* http://www.ietf.org/rfc/rfc3838.txt, August 2004

[12] A. Barbir, O. Batuner, B. Srinivas, et al. *Security Threats and Risks for Open Pluggable Edge Services (OPES),* http://www.ietf.org/rfc/rfc3837.txt, August 2004

[13] A. Barbir, A. Rousskov. *Open Pluggable Edge Services (OPES) Treatment of IAB Considerations,* http://www.ietf.org/rfc/rfc3914.txt, October 2004

[14] A. Beck, M. Hofmann, H. Orman, et al. *Requirements for Open Pluggable Edge Services (OPES) Callout Protocols,* http://www.ietf.org/rfc/rfc3836.txt, August 2004

[15] A. Rousskov. *Open Pluggable Edge Services (OPES) Callout Protocol (OCP) Core,* http://www.ietf.org/rfc/rfc4037.txt, March 2005

[16] A. Barbir. *Open Pluggable Edge Services (OPES) Entities and End Points Communication,* http://www.ietf.org/rfc/rfc3897.txt, September 2004

[17] A. Rousskov, M. Stecher. *HTTP Adaptation with Open Pluggable Edge Services (OPES),* http://www.ietf.org/rfc/rfc4236.txt, November 2005

[18] M. Stecher, A. Barbir. *Open Pluggable Edge Services (OPES) SMTP Use Cases,* http://www.ietf.org/rfc/rfc4496.txt, May 2006

[19] A. Walker. *Proxylet Local Execution Environment Java Binding V0.1,* http://standards.nortelnetworks.com/opes/non-wg-doc/draft-walker-opes-proxylet-java-binding-01.txt, August 2001

[20] A. Beck, M. Hofmann. *IRML: A Rule Specification Language for Intermediary Services,* http://tools.ietf.org/html/draft-beck-opes-irml-03, June 24, 2003

[21] C. W. Ng, P. Y. Tan, H. Cheng. *Sub-System Extension to IRML,* http://xml.coverpages.org/draft-ng-opes-irmlsubsys-00.txt, July 2001

[22] A. Westerinen, J. Schnizlein, et al. *Terminology for Policy-Based Management,* http://www.ietf.org/rfc/rfc3198.txt, November 2001

[23] S. Floyd. *Congestion Control Principles,* http://www.ietf.org/rfc/rfc2914.txt, September 2000

[24] J. Postel. *Transmission Control Protocol,* http://www.ietf.org/rfc/rfc793.txt, September 1981

[25] R. Fielding. *Hypertext Transfer Protocol -- HTTP/1.1,* http://www.ietf.org/rfc/rfc2616.txt, June 1999

[26] J. Elson, A. Cerpa. *Internet Content Adaptation Protocol (ICAP),* http://www.ietf.org/rfc/rfc3507.txt, April 2003

[27] M. Stecher. *Evaluating the ICAP protocol regarding the OPES callout protocol requirements,* http://www.martin-stecher.de/opes/draft-stecher-opes-icap-eval-00.txt, June 2002

[28] R. Rezaur, R. R. Menon. *Enabling OPES to Use Web Service for Callout Service,* http://www3.ietf.org/proceedings/01dec/I-D/draft-rrahman-web-service-00.txt, May 2002

[29] J. Morfin. *RE: AW: Using XML in OCP transport, email to ietf-openproxy@imc.org,* http://www.mhonarc.org/archive/html/ietf-openproxy/2003-05/msg00038.html, May 7, 2003

[30] A. Rousskov. *RE: SOAP and OCP, email to ietf-openproxy@imc.org,* http://www.mhonarc.org/archive/html/ietf-openproxy/2003-04/msg00196.html, April 21, 2003

[31] M. Stecher, A. Rousskovs. *OCP: OPES callout protocoll, Proceedings of 57th IETF, Vienna,* http://www3.ietf.org/proceedings/03jul/slides/opes-1.pdf, July 2003

[32] D. Crocker, Ed., P. Overell. *Augmented BNF for Syntax Specifications: ABNF,* http://www.ietf.org/rfc/rfc2234.txt, November 1997

[33] Sun Microsystems, Inc. *Sun Java System Portal Server Administration Guide - Proxylet and Rewriter,* http://docs.sun.com/source/817-7693/3-rewriter.html, 2004

[34] Sun Microsystems, Inc. *Sun Java System Portal Server Administration Guide - Creating Your Portal Design,* http://docs.sun.com/source/817-7697/5-design.html, 2004

[35] M. Fry, A. Ghosh. *Application Layer Active networking,* Computer Networks, 31, 7, pages 655-667, 1999

[36] M. Fry, A. Ghosh, G. MacLarty. *An infrastructure for application level active networking* Computer Networks, 36, 1, pages 5-20, 2001

[37] A. Barbir, N. Bennett, R. Penno, et al. *A Framework for Service Personalization,* http://standards.nortelnetworks.com/opes/non-wg-doc/draft-barbir-opes-fsp-00.txt, November 2001

[38] G. Tomlinson, H. Orman, M. Condry, et al. *Extensible Proxy Services Framework,* http://standards.nortelnetworks.com/opes/non-wg-doc/draft-tomlinson-epsfw-00.txt, July 2000

[39] Lily Yang, Markus Hofmann. *OPES Architecture for Rule Processing and Service Execution,* http://quimby.gnus.org/internet-drafts/draft-yang-opes-rule-processing-service-execution-00.txt, 2001

[40] C. Maciocco, M. Hofmann. *OMML: OPES Meta-data Markup Language,* http://xml.coverpages.org/draft-maciocco-opes-omml-00.txt, August 2001

[41] C. W. Ng, P. Y. Tan, H. Cheng. *Quality of Service Extension to IRML,* http://standards.nortelnetworks.com/opes/non-wg-doc/draft-ng-opes-irmlqos-00.txt, July 2001

[42] D. Austin, A. Barbir, et al. *Web Services Architecture Requirements,* http://www.w3.org/TR/wsa-reqs/, February 11, 2004

[43] D. Booth, H. Haas, et al. *Web Services Architecture,* http://www.w3.org/TR/ws-arch/, February 11, 2004

[44] N. Mitra. *SOAP Version 1.2 Part 0: Primer,* http://www.w3.org/TR/soap12-part0/, June 24, 2003

[45] Sun Microsystems, Inc. *Java Technology,* http://java.sun.com/, 2007

[46] Akshathkumar Shetty. *QuickServer: Java library for creating robust, multi-threaded, multi-client TCP servers,* http://www.quickserver.org/, 2006

[47] Akshathkumar Shetty. *QuickServer 1.4.6 - Basic Architecture,* http://www.quickserver.org/docs/architecture.pdf, 2005

[48] The Apache Software Foundation. *Jakarta Commons HttpClient,* http://jakarta.apache.org/commons/httpclient/, 2006

[49] OPES Working Group. *Open Pluggable Edge Services (opes) Charter,* http://www.ietf.org/html.charters/opes-charter.html, August, 16, 2006

[50] A. Beck, A. Rousskov. *P: Message Processing Language,* http://tools.ietf.org/html/draft-ietf-opes-rules-p-02, October 2003

[51] H. K. Orman. *Hopalong: A Streaming Rules Language,* http://www.purplestreak.com/ietf-opes/draft-ietf-opes-rules-language-hopalong-00.txt, August 2005

[52] M. Hofmann. *Re: Rules Language, email to ietf-openproxy@imc.org,* http://www.mhonarc.org/archive/html/ietf-openproxy/2005-10/msg00010.html, October 19, 2005

[53] The Apache Software Foundation. *Jakarta Commons Digester,* http://jakarta.apache.org/commons/digester/, 2006

[54] Internet Archive, *Wayback Machine,* http://www.archive.org/about/about.php, 2007

[55] Google. *Google SOAP Search API,* http://code.google.com/apis/soapsearch/, 2007

[56] J. Clark. *XSL Transformations (XSLT) Version 1.0,* http://www.w3.org/TR/xslt, November 16, 1999

[57] Andy Quick, et.al. *jTidy* http://jtidy.sourceforge.net/, 2006

[58] The Apache Software Foundation. *The Apache Xalan Project,* http://xalan.apache.org/, 2006

[59] A. Berglund, S. Boag et al. *XML Path Language (XPath) 2.0,* http://www.w3.org/TR/xpath20/, January 2007

[60] Martin Thelian. *jSoapServer,* http://jSoapServer.sourceforge.net/, 2006

[61] The Apache Software Foundation. *Web Service - Axis,* http://ws.apache.org/axis/, 2006

[62] The Apache Software Foundation. *Apache WSS4J - WSS4J Documentation,* http://ws.apache.org/wss4j/, 2007

[63] M. Jakesch. *Vergleich von SOAP Kommunikationsplattformen,* October 14, 2004

[64] J. Boyer. *Canonical XML Version 1.0,* http://www.w3.org/TR/2001/REC-xml-c14n-20010315, March 15, 2001

[65] M. Christen, et al. *YaCy - P2P Web Search Engine,* http://www.yacy.net/yacy/, 2007

[66] M. Thelian. *Callout Protocol Core - Kept parameter, email to ietf-openproxy@imc.org,* http://www.mhonarc.org/archive/html/ietf-openproxy/2006-04/msg00000.html, April 2006

[67] M. Thelian. *OCP - Transaction End (TE), email to ietf-openproxy@imc.org,* http://www.mhonarc.org/archive/html/ietf-openproxy/2007-02/msg00005.html, February 2007