RWTH AACHEN UNIVERSITY

COM SYS

Implementation of a OneAPI-REST interface for integrating web services in an IMS-based telecommunication network

Bachelor Thesis

Jens Helge Reelfs

RWTH Aachen University, Germany

Chair for Communication and Distributed Systems

Advisors:

Prof. Dr.-Ing.	Klaus Wehrle
Prof. Dr.	Bernhard Rumpe
Dr. rer. nat.	Dirk Thißen

Registration date: 2nd Dec. 2010

Submission date: 30th Mar. 2011

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, den 30.03.2011

Abstract

Latest development shows that telecommunication providers merge their networks into all-IP-based networks. For this pupose, IMS (IP-Multimedia Subsystem) has been introduced. The architecture allows to deploy application servers providing several telecommunication services ontop of the IMS.

Motivated by the rapidly growing popularity of Web 2.0 applications and the emerging market with value added services on mobile devices, there have been introduced standards to combine the telecommunication and internet domain. The most recent standard is OneAPI which defines some key services in a RESTful design. RESTful Web Services are kept small by directly using the HTTP's protocol mechanisms which makes them very efficient.

This thesis shows a prototype implementation of a generic REST interface, and a gateway module translating between specific OneAPI and REST messages on a JAIN SLEE AS being an event driven platform. To perfect an example of the complete vertical communicationg through this AS, we also present an SMS service implementation.

A brief overview of a performance evaluation is given and the results are discussed.

Acknowledgments

I thank Klaus Wehrle for being my first supervisor and giving me the opportunity to realize my bachelor thesis at the Chair for Communication and Distributed Systems. I thank Bernhard Rumpe for being my second thesis supervisor.

I especially thank Dirk for doing a great job as my advisor and his kind character. I thank him for all the good times when discussing several topics and approaches in relation to the thesis' topic.

I thank all my friends and my brother who willingly proofread this thesis. I also want to thank my housemate Thomas for lending an ear whenever needed.

Last but not least, I thank my parents for giving me as much support as possible during my entire bachelor studies. I also want to thank them for having so much confidence in me.

Contents

1

Introduction

Over the last several years, telecommunication providers have experienced massive changes on their networks and the whole telecommunication market. In the past, providers had been responsible just for transporting data through their networks, but this has changed. Nowadays they turned into service providers in addition to the classical data transportation.
The outdated circuit switched networks have been substituted by packet switched ones. This, in turn, merges to all-IP-based communcation into homogenerous networks which embodies a central access point to the network and enables the providers to supply additional services instead of pure data transportation.
Such services may be e.g. a simple call-redirection or even more complex, a mailbox application. They in general can use any additional available information provided by the network which allows more convenient services (e.g. Location service for mobile phones or Terminal Status).

End-user devices are becoming more powerful and are not restricted to voice communication anymore. With today's mobile technology, internet is available anywhere to nearly all mobile devices. Even the classical telephones turn into more powerful devices which may provide functions e.g. for sending SMS.
There can be observed a current hype about "apps" for mobile devices that can be downloaded from the internet. These apps make use of other internet services very excessively (e.g. an application giving weather information provided by any third party).

Both facts lead to the idea to join both domains, the telecommunication and the web. For interoperability and open access, there have been introduced several standards defining network services. The latest development on this topic is OneAPI. OneAPI defines some elemental services and is designed to expose these to the web. This is done via RESTful Web Services.

The exposure enables third parties (or the telecommunication provider itself) to set up far more sophisticated services which are compositions of these key services. This has a positive effect on three parties. End-users get more convenient and intelligent services provided by any third party which will probably charge its usage. Furthermore the telecommunication provider benefits from these services as well.

1.1 Challenge

The main challenge we address in this thesis is to create an architecture to implement telecommunication services. This architecture is restricted to be deployed ontop of an IMS system as the underlying communication network. The most attention will be given to the external interface defined by the OneAPI specification. It belongs to the familily of the RESTful Web Services which introduce a specialized way on how to access data and modify it. Furthermore a RESTful service predetermines the communication procotol.

This means we are supposed to create a RESTful interface which has to be brought into accordance with the OneAPI service specifications. Thus, we also have to translate generic REST requests into specific OneAPI requests that determine the addressed service and its methods, and of course, the other way round.

To get a better impression of the posposed prototype's performance, it has to be tested at different loads. For this purpose, there is also need to create at least one actual OneAPI service that uses the newly introduced interface.
The tests are expected to spot possible bottlenecks and possibly how to eleminate these and to enhance the whole system.

1.2 Outline

Chapter 2 introduces some basic knowledge about telecummunication networks, services, standards and the platform used in this work.

Chapter 3 presents some approaches related to work described in this thesis.

Our design of the OneAPI REST interface is discussed and presented in Chapter 4.

This leads to the actual implementation described extensively in Chapter 5 introducing an SMS service and details on the different parts of the interface, namely the message translation and Resource Adaptor.

This will be completed with a performance evaluation and some discussion of the results for the proposed implementation in Chapter 6.

In Chapter 7 we give a conclusion and consider future work.

2

Background

At first, we will give some basic knowledge on different involved topics. These are telecommication providers and services, the IMS as a core network and ontop of it an application server for providing services. Some service architectures and interfaces are introduced as well.

2.1 Telecommunication Services

Telecommunication describes the transport of information (e.g. audio, video or data) with the purpose of communication. The classical telecommunication service as such denotes the pure transport of data independently of actual content which is left to the user. Telecommunication Service Providers carry telecommunication networks to provide this service. Some examples are the telephone network or the internet.

One aspect in telecommunication networks is the introduction of Intelligent Networks (IN) which provide value-added services (i.e. telephone number portability or prepaid calling) to conventional telephone and mobile networks owned by the service provider. Nowadays especially when talking about the internet, these networks are very complex and convoluted constructs of different providers. An interesting fact is, that these networks have been converging in the past several years.

2.1.1 Examples

Audio. One of the first introduced and best known telecommunication service is the transport of audio signals or better to say voice. The actual transmission had been done a long time as an analogue signal, but for example with introduction of ISDN in the 1980's, this has been changed to digital transmission up to now in many different ways. One of the most notably sector are mobile phones as an emerging

market. Besides this, youngest development at service providers aims at merging their networks internally to IP-based communication (Voice over IP).

Short Messaging. Another well known example is the Short Messaging Service. This service is not only transport-related, but belongs to the scope of telecommunication services as well. Introduced as a feature of mobile phones, it is now available to most telecommunication devices. It needs some more management by the service provider, since messages have to be stored until their delivery or have to be delivered to clients that are only capable of receiving audio.
In relation to SMS, the next generation Multimedia Messaging Service (MMS) should also be mentioned. It enhances the SMS by the possibility to send multimedia contents such as pictures or videos.

2.1.2 Service Composition and Third Party Access

These basic value-added services become more interesting by composition. Plugging these services together enables the provider to build up more sophisticated services and applications for an end user.
Very simple examples are SMS notifications about missed calls or new messages on the mailbox.

A more complex scenario would be a stock notification application. It observes a certain stock for which a user has subscribed. Whenever it reaches a predefined value, the application tries to set up a call between the user and his broker. If this fails, it is supposed to send a notification by SMS to the user. This service might be charged all-inclusive at subscription time. The main elemental telecommunication services involved here are something for a *third party call*, *short messaging* and *payment*.

With today's powerful mobile devices and internet flatrates, in times of the Web 2.0, there is a big market for infinite different specialized applications using telecommunication services. It is obvious that a single service provider is not able to serve all these demands. Thus, a good solution is to expose interfaces to such elemental key services in the telecommunication domain to third parties (i.e. via the internet) enabling third parties to build up their own high-level specialized services.

2.2 IP Multimedia Subsystem

To merge different networks and to enable providers to build higher level telecommunication services, the the IP Multimedia Subsystem (IMS) has been introduced [7]. It is a collection of different specifications of the 3GPP (3rd Generation Partnership Project). It simplifies and standardises the access to services from several networks. IMS is not meant to standardise applications as such, but getting easy common access to a central control unit and merge different network types.

It can be splitted into the transport or media layer, the session control layer and the application layer. The application layer can be used to set up services like those provided by INs. We will not go in too much detail on this topic, thus figure 2.1 depicts a very simplified overview of the IMS architecture.

Media/User Layer. IMS internally uses IP based communication, thus, an IP-Backbone as a central unit connects all convergent networks. Access to the IMS is mosty gained via direct use of IP (i.e. PDAs, SIP phones, mobile phones). Anyway, other technologies such as Digital Subscriber Line (DSL), cable modems, mobile access via GSM or GPRS as well as wireless access (WLAN) are supported. Technologies like cable switched networks (e.g. plain old telephone system) are supported via specialized gateways.

Session and Control Layer. To manage all internal (IP-based) communication, the Session Initiation Protocol (SIP) is employed. This control layer architecture is motivated by and partly derived from classical GSM. It generally consists of the Home Subscriber Server (HSS) and the Call Session Control Functions (CSCFs).
The HSS is a central database that holds all user and subscription related information. It is responsible for user authentication, authorization and provides current information about a user's IP address and location.
The CSCF servers can be seen as a network or infrastructure to provide several signaling and management functions. This can be seen analogously to routing and switching in the OSI reference model. The most important are a central (per registration session persistent) proxy through which a user is reachable; a component that is responsible for forwarding messages from remote IMS systems; and a unit providing several management functions such as user registration.

Application/Service Layer. The IMS architecture also arranges application servers. For communication, SIP is used as well. These ASs can provide any type of value-added service or service compositions as mentioned in 2.1.1 and 2.1.2. Such an AS may either be owned by the provider or a third party. Furthermore ASs can be cascaded by exposing network capabilities or higher-level services to other applications.

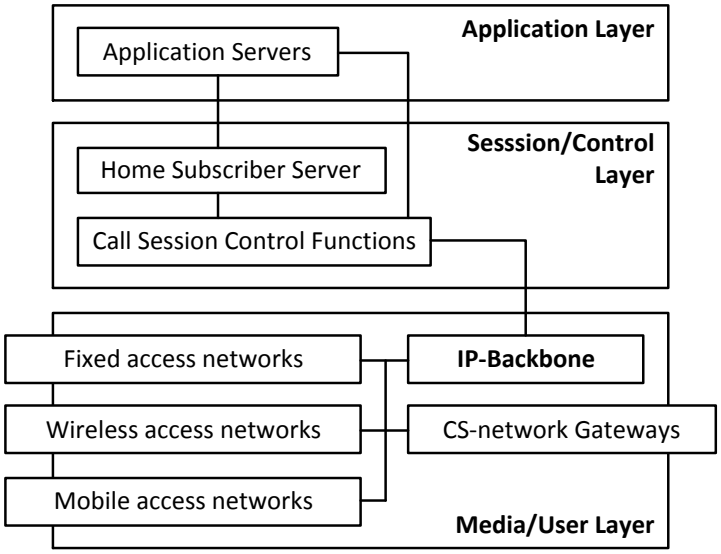

Figure 2.1 Simplified IMS architecture. It can be subdivided into three different Layers. Application servers and services are located in the Application Layer. Signaling and user management are done on the Session and Control Layer. The Media Layer combines different access networks into an all-IP-based network.

2.3 Web Services

To provide any service, there is need for an interface and a communication platform. One widespread approach to these concerns are Web Services. First of all they are application components which are mostly supposed to communicate machine to machine. Thus, they are accessable via well defined interfaces and protocols. The main goal of them are to assure interoperability by using HTTP together with XML as the typical platform. They are either used to create new reusable application components (e.g. currency conversion or weather reports) or to connect existing software. So they can provide a platform to link different applications independently of their origin. The W3C defines Web Services generically as follows:

Definition 2.1. A Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML based messages exchanged via internet-based protocols.

A typical Web Service architecture is defined by a service provider, consumer and a directory service. Actual services can be discovered by a consumer via the directory service. This is done by the client looking up a service at the directory server which responds a Web Service Description Languare (WSDL) file defining the service's functions and access interface. With this information about the service, the client then is able to send a request encapsulated in a Simple Object Access Protocol (SOAP) envelope to the service provider. SOAP uses HTTP for transport. These SOAP messages can contain any XML content representing an actual request. After receiving such a request, the server sends the corresponding response according to the WSDL definition also encoded in XML and encapsulated into a SOAP message.

2.3.1 Representational State Transfer

Web Services are very powerful, but we have seen that they introduce much overhead by encapsulating the actual data into SOAP envelopes.
To minimize network communication while maximizing scalability and independence of implementations, the Representational State Transfer (REST) has been introcued by R. Fielding [8]. It is an architectual style meant for distributed media systems and has been developed in parallel to HTTP/1.1. The communication participants must be at least a client and a server whereas proxies or gateways can be involved. A client sends requests to certain resources and the server gives an appropriate response.

REST is defined by four different interface constraints. These are manipulations of a resource via representations, identification of them, self-descriptive messages and hypermedia for transport.

Resources in general are an abstraction of any "thing", that should be exposed. This can be a document, a service (e.g. "today's weather in Aachen") or a nonvirtual object like a person. A client does not access resources directly, but uses representations of it which can be seen as a mapping.
These resources are addressed via their unique *resource identifier*. The identifier should be self-explaining for a more natural understanding.

Data element	*RESTful Web Service*
resource identifier	URI
control data	HTTP method or response code
resource metadata	source-link, alternates
representation metadata	media-type, last-modified time
representation	XML document, JPEG image

Table 2.1 Elements of a REST message and corresponding examples for RESTful Web Services

To perform actions on a specific resource, *representations* of this resource are used. Such representations consist of some *metadata* (e.g. last modified date, media type), can contain metadata for integrity purposes, and an actual *representation* of the resource as a byte-sequence. Additionally there is control data included defining the requested action on the resource; or, in case of a response, its semantics.
Another key aspect of REST is its statelessness. In fact all requests contain all information neccessary to perform the requested action independently of any previous request. This leads to several key benefits. No processing entity needs to store state related information resulting in lower physical resource usage and higher scalability. Requests can be processed in parallel giving better performance. Furthermore requests with its answers can be seen isolated enabling simple caching.

Example. The largest and best known example for REST-based communication is the World Wide Web. The communication here is done via pure HTTP where the requested URI (Universal Resource Identifier) defines the resource, the method and response code denote the control data, additional headers provide metadata and the actual resource representation is contained in the message body.

Note: This communication is mostly stateless as defined at REST, except cookies and some security issues at user authentication (see [14] for more detail).

2.3.2 RESTful Web Services

A RESTful Web Service is typically a service implemented using HTTP in a REST style. The resources are identified and addressed by a URI. The requested action is determined by the HTTP-method. The responses make uses of the typical HTTP response codes. An overview of the methods and response codes can be found at figure 2.2. The general structure of a message in this context is found at table 2.1. More detail on the different possible actions is given in the following listing:

- The GET method is used to request information about a certain resource. The response will contain the acutal representation, its metadata and optionally resource metadata. Such a request does not include any representation of a resource.
- PUT requests an assertion of a resource. The server tries to change (or even create) the resource to match the containing representation. Note that such a request typically should contain specified values instad of relative information (e.g. set the value to 10 instead of increment the value by 5).

- In contrast to PUT, the POST method requests to create a new resource out of an existing one. This relation might be a child relation in a datastructure-view. The main difference between POST and PUT is, that with PUT, the client needs to be able to compute the new resource's URI.
- DELETE is used to remove a certain resource.

These web services are not always stricty stateless as defined by the REST architecture. A server might store some application related state information such as authentication credentials, but if, every request has to contain this information.
The resource will be represented in a common media type like XML or JSON.

Note: RESTful Web Services do not use a directory-server as it might be the case at classical Web Services.

Method	*Description*	*Resp. code*	*Description*
GET	retrieve data from a certain	200	OK
	resource	201	Created
POST	create a new resource	204	No Content
PUT	update an existing resource	400	Bad Request
DELETE	remove a resource	404	Not found
		503	Service unavailable

Table 2.2 HTTP methods and selected responsecodes used at RESTful Web Services. The HTTP protocols functionalities are directly utilized.

2.3.3 Classic Web Services versus RESTful Web Services

We have introduced the classical and RESTful Web Services. Both are used to set up reusable application components based on HTTP for communication. They provide interoperability independently of the used platform or vendor.

The main difference is, that in RESTful architectures, all information is packed into the actual HTTP message. They make direct use of commonly known functions of the HTTP protocol. This includes the control data (i.e. at requests, the method information; at responses, the response code), the identifier via the URI as well as possibly a resource representation in the message body. This means, that already the first line of a HTTP request can determine what a client wants to do (e.g. "`GET /document.txt HTTP/1.1`").

In contrast to this, the classical Web Service approach uses HTTP just for transport purposes. The actual data is ecapsulated into an envelope (e.g. SOAP) which is put into the HTTP message body. The request or generally method information are kept inside this envelope. In addition to this, the format of these messages introduce complete new vocabulary defined in the service related WSDL file [14].

Conclusion. Summing the facts up, classical Web Services present a very powerful interface with extended possibilities by defining completely new individual messages with its own vocabulary, but this results in overhead in two ways. The messages need more network bandwidth by the protocol overhead because of the encapsulation of the actual data into an envelope. Moreover the parsing of the messages becomes computationally more expensive.
A compromise between power and resource usage are the RESTful Web Services. They use the HTTP protocol functionalities directly and put the actual data into the HTTP message body without any additional envelope. As a result, the REST messages are smaller compared to a comparable SOAP message, which results in less bandwidth usage. Keeping the different possible functions very small (e.g. we only can use the typical HTTP methods at requests) causes less expressiveness.
Nevertheless, the outcome of this is normally that we do not need as much cpu-time as it is the case at classical Web Services.

2.4 Standards

As telecommunication providers nowadays are very interested in serving value-added services to their customers and there is even need to expose elemental key services to third parties, there has been a couple of standards developed over the past ten years.
We will now introduce the most important ones defining interfaces for different telecommunication services. They guarantee interoperability between varying vendors and platforms, as well as enabling third parties to use the defined services particularly to create more sophisticated applications. Togehter with IMS, they define the foundation to the general concept of opening the telecommunication world towards the web.

2.4.1 Parlay/OSA and Parlay X

The Parlay OSA specifications are one of the first published standards to define interfaces to telephone network services at all, but on a very low level. The aim is to give developers an open common interface for network features independently of the specific underlying network. It has been defined by a consortium of several industry vendors called Parlay Group. It was founded in 1998 and ended about 2007. There was hope that IT developers rather than telephony experts can build services.

A more abstract approach to this is the Parlay X specification [1] introduced in 2003, that opens the telecommication world to the internet via Web Services. The Parlay X API set is much simpler (and hence, less powerful), but had impact on the developer community to grow. Most important services defined by Parlay X can be found at table 2.3. The work on Parlay X has ended in 2007 with its version 3. However, it has been taken over by the Open Mobile Alliance (OMA). Latest progress has resulted in the second version of RESTful (see RESTful Web Services at 2.3.2) bindings for some of the parts of Parlay X.

Part	*Name*	*Description*
1	Common	definition of common data objects
2	Third Pary Call	third parties initiate a call between two participants
3	Call Notification	notifications for calls and their handling
4	Short Messaging	send and receive short messages (SMS)
5	Multimedia Messaging	send and receive multimedia messages (MMS)
6	Payment	simple reservation and charging of amounts/volumes
8	Terminal Status	get status of a terminal/user
9	Terminal Location	get geographic location of a terminal or group
10	Call Handling	setup rules for call handling
14	Presence	get presence information of a terminal

Table 2.3 Some selected services defined by Parlay X

2.4.2 OneAPI

Having published the RESTful bindings for some Parlay X parts, OneAPI defines a subset of these aiming to give open access to network capabilities and information via the web, regardless of the operator more easily. OneAPI is group founded by the GSM Alliance (GSMA) in cooperation with the OMA. This collaboration has revisited the Parlay X specifications and the associated RESTful bindings and published a subset as OneAPI. The idea is to create a far more easy API for web developers in a RESTful architecture resulting in RESTful Web Services for some parts of Parlay X.

2.4.2.1 Version 1.0

After a testing period of a pre-version, the Canadian Plot (v0.91), the group released the first specification. This contains interfaces for

- Short Messaging: Send and receive textual short messages
- Multimedia Messaging: Send and receive multimedia messages
- Payment: Charge and reserve amounts
- Location: Get the geographical location of a terminal or group

The mappings from the Parlay X RESTful bindings to OneAPI can be found at [10] whereas the actual RESTful interface is defined in [12]. Note that OneAPI version 1 also includes the classical Parlay X SOAP APIs for the mentioned services.

2.4.2.2 Version 2.0

In addition to the services of version 1, the second version released in february 2011 introduces the following new APIs:

- Voice Call Control: Create a call between multiple users, notify about call-events
- Data Connection Profile: Retrieve information of the end users connection type and roaming status
- Device Capability: Retrieve information of a device

These are in based on the Restful Parlay X bindings at version 2. OneAPI version 2.0 correlates to the newer version of the bindings as it defines a subset of these as well. There are currently no additional SOAP bindings available for the newly introduced services.

2.5 JAIN Service Logic Execution Environment

The JAIN Service Logic Execution Environment (JAIN SLEE or SLEE) [16] defines an architecture meant for communication applications. It specifies a component model for building structured reusable object-oriented logical blocks. These can be composed to set up more sophisticated services. In addition, the specification defines several standard facilities such as a timer and tracer. The latest release is version 1.1.

In particular it can be used ontop of an IMS system to enable providing services as defined by the mentioned standards (e.g. Parlay X and OneAPI) or even more complex ones.

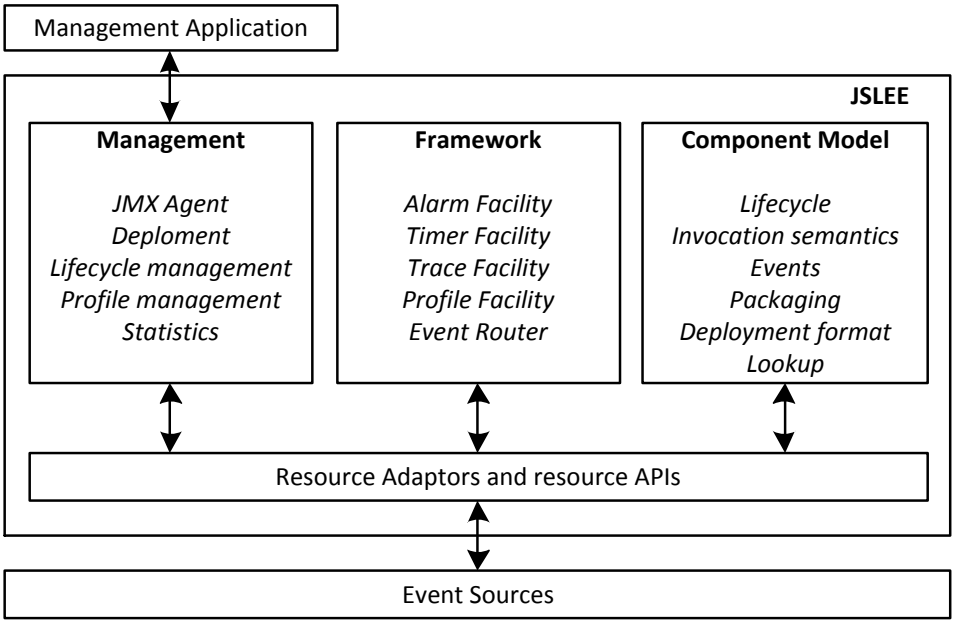

Figure 2.2 JAIN SLEE architecture overview. It can be subdivided into four cornerstones: the Management (e.g. deployment and lifecycle management), the Framework (e.g. timer facility and event router), the Component Model (e.g. defining SBBs) and the Resource Adaptors (interface to external resources).

A SLEE is an application server (AS) which acts as a container for the software components. The architecture is optimized and designed for asynchronous or event driven applications (EDA). These applications receive their requests in the form of events. Typically there is no active thread for execution, but there are handler methods defined for different events that will be executed. These handler methods are allowed to make use of resources and emit new events.
The SLEE architecture can be subdivided into three major internal subsystems seen at figure 2.2 taken from [9]. For communication with external components, there are Resource Adaptors to be set up, that act as an interface between the SLEE and these external resources.

- **Management** provides functions for deployment, lifecycle management, profile management as well as statistics. These functions can be accessed internally as well as externally via Java Management Extensons (JMX).

- The **Framework** provides several standard functions to be used in applications. The most important ones are the following:
  - Alarm facility: This facility is used to notify management applications about certain events. These may be unexpected states (e.g. a database connection failure).
  - Timer facility: With the timer facility it is possible to trigger periodic or delayed events. As soon as the timer expires, a new event will be thrown by the SLEE on which an application can react.
  - Trace facility: This facility provides trace functionality. The trace messages can be observed and analized by external applications. The number of genereted messages depends on the set log level.
  - Profile facility: The profile facility allows to store application relevant information inside the SLEE. It builds up user-predefined tables and enables the usage of static SQL queries. The information can then be managed easily.
- A **Component Model** provides infrastructure for checking and deploying the actual application's service logic implemented by Service Building Blocks (SBBs). It is responsible for building the runtime environment where whole SBB trees are instantiated in containers. It performs the setup of the event handlers and is responsible for lifecycle related management jobs as well as the actual event routing and delivery.

2.5.1 Resource Adaptors

Resources represent external systems to the SLEE. Some examples are network devices, protocol stacks and databases. These can have Java APIs which are internally represented by events. SLEE applications are able to interact with such resources via Resource Adaptors (RAs) building an interface to these. The RAs define all needed functionalities to adapt a resource for the SLEE's needs. A RA consists of the following main concepts:

- Resource Adaptor type. The type declares common characteristics for a set of RAs. It defines all interfaces that a RA implements. It also defines event types for events that can fired into the system.
- Resource Adaptor. This is the actual resource adaptor implementation of one or more RA types. (This means it is allowed to have multiple implementations for the same type.) It also includes all needed Java classes that represent the actual resource to adapt.
- Resource Adaptor entity. An entity is an instance of a RA. There are multiple instances allowed, for example it is possible to set up multiple instances of the same particular RA on different IP addresses or ports.
- Resource Adaptor object. The RA object is the instantiated RA class that is used by the SLEE to interact with the RA entities.

- SLEE endpoint. To interact with the SLEE, the endpoint declares an interface for a RA to create new acitvities and fire new events into the system. It is implemented by the SLEE.

2.5.2 Service Building Blocks and Services

Service Building Blocks (SBBs) each define logical blocks providing a set of specialized functions.

Components are an abstract declaration of an SBB including what events will be received and fired. Thus, it defines which methods are implemented that can be invoked by the SLEE. Furthermore it defines which libraries are used. Components can be linked together into an **entity**-tree consisting of several logical blocks. Each component belongs only to one entity at a specific point in time.

An entity tree with its root node typically defines a complete service. It is important to point out that the entities define a logical relationship whereas objects are assigned to an SBB entity dynamically. At runtime, they have a lifecycle and thus, they have to cache stored information for each entity they are used in.

This means, that services can be created out of different small blocks or modules each doing a specialized job.

In order to cumminicate with any external device, SBBs use Resource Adaptors. For this purpose, one has to bind a RA to the SBB and implement methods to handle the desired events emitted by this RA.

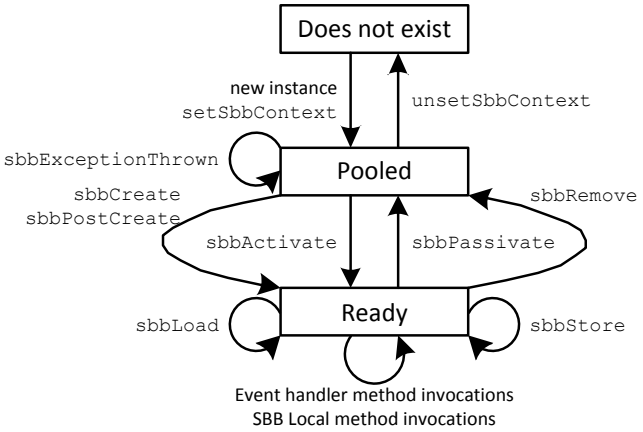

Figure 2.3 SBB object lifecycle. SBBs are pooled inside the SLEE. This saves instantiating time because of reusing them, but introduces some management overhead. If there is no available instance in the pool (*Pooled*), a new SBB has to be instantiated. These new instances or other pooled SBBs get aquired to be used in an entity of a service (*Ready*).

Lifecycle. SBBs instances are managed by the SLEE in a pool. New one are only instantiated by the SLEE when needed. Such instances are called **Objects**. Whenever an object is created, it changes into the *Pooled* state. *Pooled* objects can change their status to *Ready* by activating or creating. They get *Pooled* again by passivating or removing when they are not needed anymore. At *Ready* state, methods can be invoked either by receiving an event or by another linked SBB. Here it is also

possible to load and store information. The corresponding state machine can be found at figure 2.3.

2.5.3 Events

Events represent transition changes or occurrences of significance and are defined by an event type and an event object. They can be fired from inside or outside the SLEE. Event producers can be either inside, SBB entities, the SLEE facilities, the SLEE itself; or RAs entities (outside). Events convey any infomation encapsulated in the object itself.

Corresponding event consumers are SBB entities only. For this reason they implement handler methods for the different event types.
Such an event object is an instance of an event class. The event type is defined by deployment descriptors being independent of an actual class. When firing an event, the application has to give information about its type by which the SLEE determines how to route this specific event, i.e. which SBBs will receive it and which handler methods to invoke.

2.5.4 Acitivities and Activity Contexts

The SLEE specification introduces activities representing an event bus for each ongoing process/action. Any event must be fired on an activity which acts like a communication channel. The related important terms are:

- Activity. An activity is a logical entity defining a process or stream of one or more events. For example it can represent a communication channel inside the SLEE towards a resource (i.e. an incoming TCP connection at a RA). State changes to this resource typically result in an event that gives related information.
- Activity Object. An activity object is the java representation of an activity. They are created either by a RA entity or a SLEE facility. The actual object defines an interface for interaction with underlying activity.
- Activity Context. An AC represents an acitvity by encapsulating the activity object. It is a logical entity as well that enables the application to send and receive events on an activity. It also holds and gives access shareable attributes between different SBBs.
- Activity Context Interface object. Each AC is accessed by an SBB trough an interface object. This interface may be the generic `ActivityContextInterface` or a specific SBB local Actvitivy Interface which allows to define shareable attributes to use between different SBBs on the same context.

SBB entities use an `ActivityContextInterface` object to gain access to the actual Activity Context. There can be many of these interface objects each used at different

entities. Every interface object has a particular Acitivity Context where it belongs to being identified at the resource domain by the Activity Handle. Each Activity is represented by an activity object being identified in the SLEE and Resource domain by such a handle. These relations are shown in figure 2.4.

Besides the activities created by RAs or facilities, the SLEE also provides Nullactivities that are used as a private event bus for communication between different SBB entities.

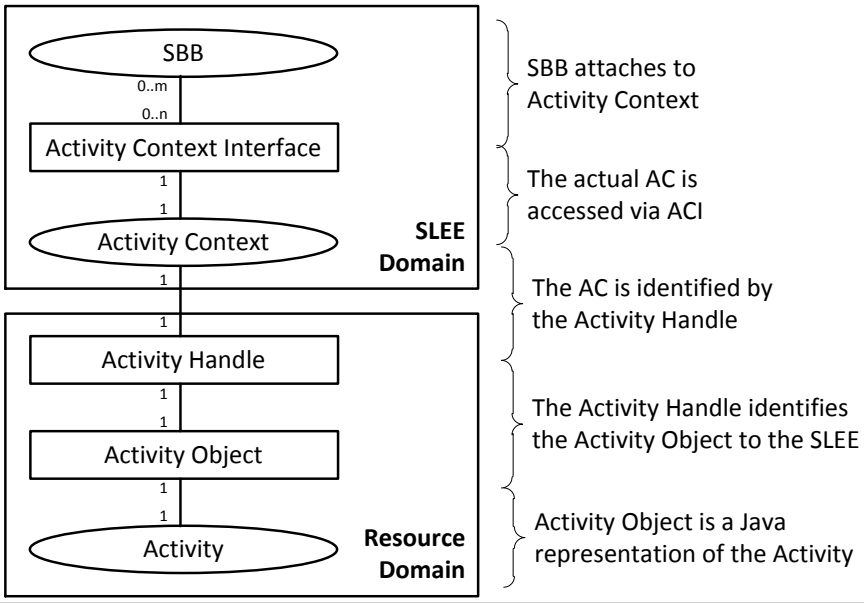

Figure 2.4 Activity Context and the connection to Activity Objects (ellipses denote logical entities, rectangulars actual java objects)

2.5.5 Profile Tables and Container Managed Persistance Fields

It is often important to store information about an ongoing activity. Since the SBB objects are pooled instances being reused at different activities, it is not possible to store such information directly inside this object. For this reason, the SLEE architecture provides two different methods to store persistent information. These are the Container Managed Persistance (CMP) fields and the Profile Tables.

CMP fields are a set of variables defined in the SBB component. These fields are virtual fields only that can be accessed via a getter and setter method which are implemented by the SLEE. Their possible values are restricted to Java primitives, Java serializable types and SLEE specific interfaces (e.g. Activity Context Interfaces).

Profile Tables can be used to store more sophisticated related data. In contrast to the CMP fields, a profile is a set of related variables. These are grouped and managed in profile tables. Access to the profile tables is gained via the Profile Facility that can be accessed either by SBBs or RAs. It is possible to define static SQL queries in order to get specific entries of interest.

2.6 Summary

Now we have introduced background information for our work. We have seen what a telecommunication service is and some examples. Another fact is, that telecommunication networks have been converging the past several years. A good solution to this process and enabling a provider to create value-added services, is the IMS as an interconnection backbone with control functionality.

The raised standards of the Parlay family and the newest development, namely OneAPI, define elemental services and their interface. OneAPI specifies elemental services (e.g. Short Messaging, Terminal Location, Payment) which are exposed to the web as RESTful Web Services. The RESTful approach is simple with low protocol overhead by using the HTTP protocol's functionality directly to determine the requested action (e.g. GET, POST, DELETE) or the kind of response (e.g. 200 OK, 404 Not found). The actual message payload is diretly packaged into the message body without any additional container. Furthermore, the typical encoding format for the messages is XML or JSON which are commonly used, simple and well known.

These services are typically facilitated at application servers which are integrated in the IMS architecture. For this purpose, especially in the telecommication domain, the JAIN SLEE AS can be used. It is a specialized event driven platform and thus, is supposed to handle many of events/processes quickly and in parallel.
For communication in general, the SLEE uses events. They denote any occurence of significance. They can be received and handled by SBBs, that are single reusable software components. SBBs can be set up into trees building a complete service. To communicate with external resources (e.g. a database), the SLEE uses RAs. They translate external bevahior into events that can be used inside the SLEE and enable SBBs to send some specified data towards the resource (e.g. an SBB requests a database RA to handle an SQL query - the database's response is fired as an event). Each ongoing process in the SLEE has an underlying activity held either by a RA or the SLEE itself. These activities can be seen as communication channels inside the SLEE, an event-bus respectively, where events are fired on (i.e. put onto the bus). Activities are accessed by SBBs via a context interface. To trigger a new service instance, SBBs can declare events as initial (e.g. a service declares a request-event as initial, but not the response-event).

This means we have our IP based IMS network with its control functions. This architecture allows a JAIN SLEE AS set ontop communicating towards this network via SIP. Various telecommunication services can be implemented on this AS and exposed towards the web as RESTful Web Services.

3

Related Work

3.1 Parlay X and SOAP

The idea using an event driven platform for telecommunication services is for example introduced at [5]. The dissertation focuses more an context-awareness, which means using network information about a user to provide extended services. Related to this idea and the JSLEE as the used platform, there has been work on implementing parts of the Parlay OSA [15] and Parlay X [9] specification. It turned out, that the Parlay OSA interface is powerful, but does not allow non-telecommunication-experts to create services easily. With the Parlay X interface using the underlying Parlay OSA functions, we see a lot of non-neccessary overhead.

We decided to give up on these ideas and create the new OneAPI interface more or less from scratch, i.e. eleminate the Parlay OSA layer. Thus, there exist only small artifacts that are helpful or can be reused.

Parlay X Library. In [9] there is a Parlay X library introduced. It models the function calls of the different services defined by the Parlay X specification. The original library did not represent these funtion calls as events inside the SLEE. The information was delivered each in an encapsulating SOAP event. Thus, it was not possible to distinguish between different function calls, but we only have a generic event on which a service can react.
To eleminate this shortcoming, the library has later been restructured into two seperate parts. On the one hand, we have the actual function calls with its corresponding responses acting as events for the SLEE. On the other hand, we have several common datasctructures and datatypes that are commonly used within these events. This will be discussed in more detail at chapter 4 and section 5.2.

SOAP Resource Adaptor. There has also a generic SOAP Resource Adaptor been developed. It is able to handle simple HTTP connections and parsing a SOAP-message out of it. Incoming SOAP-messages are fired into the SLEE, whereas it is

also possible to send outgoing messages either via an existing activity or by generating a new activity at outgoing requests. It also includes some conveniece methods like a SOAP-factory that is used to set up new SOAP events.

3.2 GSMA OneAPI implementation

In february 2011, the OMA released a project that implements OneAPI version 1.0 (SMS, MMS, Location and Payment) [11]. This implementation contains a server and several different client parts.

Server. The server part is written in Java to be deployed as Servlets in a normal JEE Application Server. There is a common Servlet that provides methods for some parsing purposes, request validation, error handling and to send responses. Each service has own Servlets (inheriting from the common Servlet) each implementing an actual service-method. An incoming request gets validated and a predefined dummy-response will be given back. For parsing the messages, the Jackson JSON library [18] is used.
Note: This implementation does not provide any further processing, but only the actual REST interface.

Client. The client implementations provide methods for invoking the OneAPI services offered by the server. Till now, the project contains several client implementations written in Python, Ruby, Java and PHP. All of these implementations provide methods for the different OneAPI service-funtions. The requests and responses are each modeled as objects which include the logic to process them (i.e. message setup, validation and error handling).

3.3 seekda!

The seekda GmbH provides several commercial web applications. One interesting service is their Web Services search engine [3]. The engine crawls the internet for Web Services and their WSDL files respectively, like google does for example with web sites. They claim to have the largest collection of Web Services known. As the crawler has already been set up in 2006, they provide some rude statistics on publicly available services. The number of crawled Web Services has rapidly been increasing in about 2007. After reaching more than 25,000, this level has slightly grown over the past few years to currently nearly 29,000 Web Services by nearly 7,000 different providers.

The services in the scope of seekda are classical Web Services only. Nevertheless, their statistic shows an enormous growth and popularity of this technique. Reasons for this might be the handiness and interoperability. Because the RESTful Web Services can be considered as way simpler, their popularity will probably have been growing as well.

In relation to these numbers, it seems that Web Services in general are nowadays the new widely accepted de facto standard to provide any kind of service to third parties via the internet. This definitaly argues for exposing telecommunication services as Web Services. At this point, OneAPI comes in. It defines interfaces for the telecommunication services as RESTful Web Services.

4

Design

The challenging task is to create an architecture to implement telecommunication services. For this purpose, we use a SLEE application server which is associated to an IMS. This architecture should be simple, easy to understand, extendable, high-perfomant and of course doing its job.

With concern to these constraints, there are some key questions to be answered:

- How to build the interface from the internals of the application server to the external network
- How to implement service calls and answers
- How and where to distinguish between different types of messages or services
- What addressing scheme to use
- How and where to decide what content-format to use

The interface design from the external network into the application server and the other way round is given by the SLEE, this will be a Resource Adaptor. As RESTful services use HTTP's addressing and its body as the transport layer for payload, there is need to have a Resource Adaptor that is capable to process these messages. This adaptor requires to set up TCP connections for new outgoing messages and to listen for incoming connections for new incoming requests. It is a decision to not make this resource adaptor generic HTTP (these more or less already exist), but encapsulate these messages into new REST messages. The reason herefore is, that the purpose is not being a generic fully-implemented hypertext client and server typically listening on port 80, but understanding these special messages. Another reason is to distinguish interally between simple hypertext and RESTful services in the future.

With RESTful services, we always have a request-response combination like function calls (the response may be void though). To model this on the event-driven platform, we use for each function call two events - the request and its corresponding response. These can be either incoming (initiated from an external user) or outgoing (initiated by the application server, i.e. one of its services). As a consequence, we introduce two types of events for the resource adaptor - a request and a response.
Each call takes place in an own activity which starts with the initial request and is terminated with the response. To distinguish between incoming and outgoing requests and responses, these activities are split accordingly.

A key question is how to distinguish between different types of messages and their corresponding service. Ealier work has proposed using a generic event that encapsulates a certain message. The services that are interested in only few specific types of encapsulated messages have to react on each generic event looking into it to determine if it is of any further interest. This works fine for only few services, but as soon as this number increases, this leads to a considerable overhead.
To eleminate this obvious shortcoming, we have to add information (i.e. a specific type of an event) at some point. This can either be done directly by the RA giving it the responsibility to know and recognize all specific event types, or we use a translation unit, that takes the generic events as a single instance and fires the corresponding specific event furhter into the system (and the other way round). The second idea sounds much more reasonable, since a generically held RA will be more beneficial in the future and it will not be overloaded with the translation logic.

For this pupose, we intodruce a gateway module. Its main task is to receive REST requests from the resource adaptor, decide on the encapsulated HTTP data (preferebly on the header information) what kind of specific message is included and translate it into a new specific event to give it further into the system. To ensure that not all services receive all incoming REST messages (with which they maybe cannot deal at all), we use a specific event-pair (request and response) for each method of each service. It also has to translate responses from the system into REST responses respectively and hand these over to the resource adaptor.
The other way is to translate requests from internal into REST requests and give these to the resource adaptor. Here it has to save the information of the activity that is started by this action and wait for the answer to give it back into the system.
As OneAPI must implement at least XML and JSON as payload data format, it has to be able to handle both. For convenience, the payload translation is dealt by a seperate library which may be used at clients as well.

Addressing and content format is only reasonable for outgoing requests. The service is determined by the type of the specific event. Incoming requests define by themselve in what data format they are encoded and the target address for further processing is defined in the payload and not of interest here.
Answers to these requests will be send on the same activity, which means back to the sender. The data format for these are predefined either chosen by a default value or defined by the previous related request (which has to be remembered by the gateway).
More interesting are outgoing requests. Here we use the SLEE capability for defining addresses for events. Messages supposed to be send, must include an address-URI starting either with XML or JSON as these define here uniquily to be send via the

REST interface. The host part of the URI determines where to send the request, that means where the resource adaptor should send the actual HTTP request. The answer to this will come back on the activity started by the invocation.

The actual application logic is implemented in SBBs. At [9], it turned out using different SBB entities communicating via an internal private communication bus (i.e. Nullactivites) compromises better reusability for the different modules, but adds heavy overhead by creating these activities. Thus, the SMS service is packed into one single entity.

For external communication, this SBB uses the new REST-RA amd the SIP-RA, that is already provided by the used mobicents [21] platform.
The state and service related information will be kept in CMP fields and Profile Tables.

5

Implementation

The main task was to build the OneAPI interface on the SLEE architecture. As discussed before the key elements are the gateway module and the Resource Adaptor. To represent the different function calls respectively the messages for the services, we also introduce a library for these data structures. To have at leat one service for reasonable tests and to show the whole vertical communication, we also implemented the Short Messaging Service. This section follows logically the communication upwards through the system. A first overview can be found at figure 5.1.

5.1 Short Messaging Service

The Short Messaging Service is implemented in the SLEE as one single SBB. It has two different communication partners, on the one hand we have the mobicents SIP resource adaptor which is the bottom interface towards the IMS; On the other hand we have the gateway.

All data of this service is kept in two different tables realized by Profile Tables. One keeps the information of the actual SMS (to send, sent or received) and the other one keeps information about notifications. Using the SLEE's internal feature to store data is easy and provides fast access, since these tables are build up by an internal database and are kept in memory.

As the SLEE dictates event-based behaviour of the components, we use a state machine within each SBB-instance to keep track of the current status. For each activity that is either started by the SIP-RA or comes from the REST-adaptor, the SBB instance stores its state which then may be changed if a certain event occurs.

SMS-messages are represented towards the IMS as SIP messages with a `MESSAGE`-header. All of these are supposed to be routed to our application server, since the IMS cannot decide wether this message is of any interest for further use or not. This means our SMS service has to provide basic functionality to forward these messages

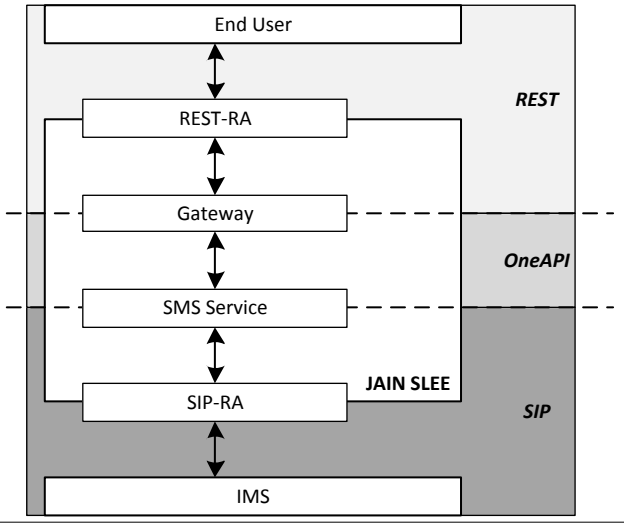

Figure 5.1 This diagram shows the whole vertical communication through the application server. The communucation with the IMS is done via the SIP-Resource Adaptor which is listening for SIP messages and firing corresponding events into the SLEE. It is also able to send SIP messages towards the IMS. The actual services make use of this RA to communicate downwards. Upwards we now use a new Event-library representing the OneAPI methods. These Events are received and generated by the new Gateway which then translates these messages and gives them further to the REST-RA. The REST-RA is responsible for the actual HTTP communication with the end user.

towards their destination. This means incoming messages might be sent back into the IMS - with or without being processed any further.

Besides this basic requirement, our service can be split into four categories of interfaces and accordingliy responsibilities: Send, Notification, Notification Manager and Receive.

Send. The send interface provides two functionalities. First, we can send short messages towards a list of SIP-recipients (Note: the recipients are restricted to SIP-recipients since the IMS only provides this interface; other addresses need to be mapped). The sending-request may provide a notification callback to get notified if the message is or cannot be delivered. The response contains a unique message-id. Another function is to ask for the delivery status.

Each request will be stored in the internal table and answered with the message-id (see 1 at figure 5.2). The initial delivery status is *MESSAGE WAITING*. Afterwards there is a new Nullactivity started for each recipient of this particular request and the sending procedure is triggered on this (see 2 at figure 5.2). This means the settling of the message is done and the request-activity will be ended. The actual sending is then done on the new Nullactivity. New SBB instances then try to deliver the message (see 3 at figure 5.2). If this ends with success, the delivery-status entry in the table is updated to *DELIVERED TO TERMINAL* for this single recipient (see 5 at figure 5.2). Otherwise this procedure will be retried after a certain time (see 4 at figure 5.2). The time period while the service retries to deliver a message is bounded by the Retention time. If a message could not be delivered wihtin this time, the

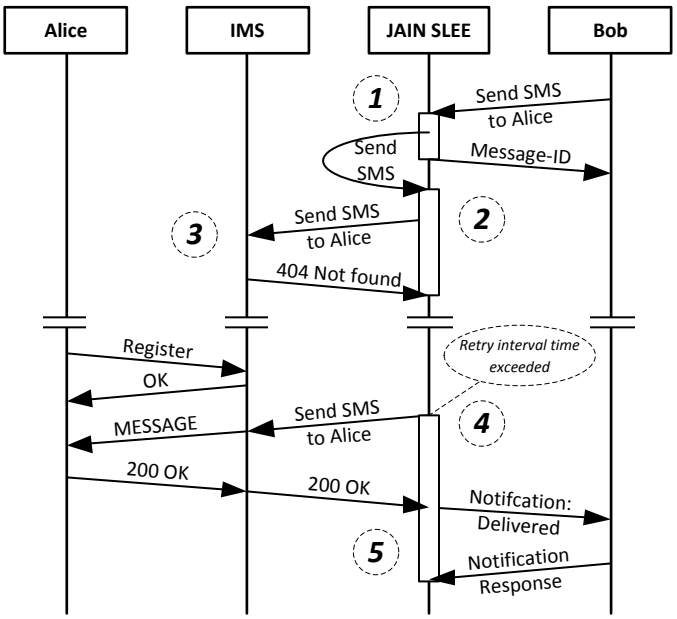

Figure 5.2 First a new SMS is sent to Alice by Bob. The message is stored by the service and an answer with a unique message-id is given. The service then starts a new activity and tries to deliver the SMS towards the IMS. Alice is currently not registered and the message cannot be delivered yet. After a retry interval, the delivery succeeds and the end user is notified about this event.

acitivity will be ended and the delivery status is set to *DELIVERY IMPOSSIBLE*. As soon as the delivery status is changed, the service checks wether to notify the sender about this event (see 5 at figure 5.2).

The request for delivery status by a sender returns a list with the current delivery status stored in the internal database.

Notification and Notification Manager are closely coupled. The notification manager provides two types of subscriptions: message delivery and message notification. The message delivery subscription overrides any given (or not given) callback for the notification for sending sms via the send interface. This means at each time the delivery status changes at the process of sending a message, the service checks for delivery-subscriptions and if there are any, it uses these to provide actively the new information.

The message notification subscription determines wether the application server stores an incoming message. This subscription is uniquely defined by a list of recipients and may have a specific criteria (this is an alphanumeric sequence with which the message body is supposed to start with) (see 1 at figure 5.3). All incoming messages are checked against these subscriptions. Is there a matching one, this message will be stored and answered to be successfully *DELIVERED TO NETWORK* in OneAPI terms or in SIP terms with the response code 200, OK. If the subscription provides a callback, a notification about this new message will be sent immediately (see 2 at figure 5.3).

Besides setting up new subscriptions, this interface also provides functions to delete subscriptions identified by their unique id.

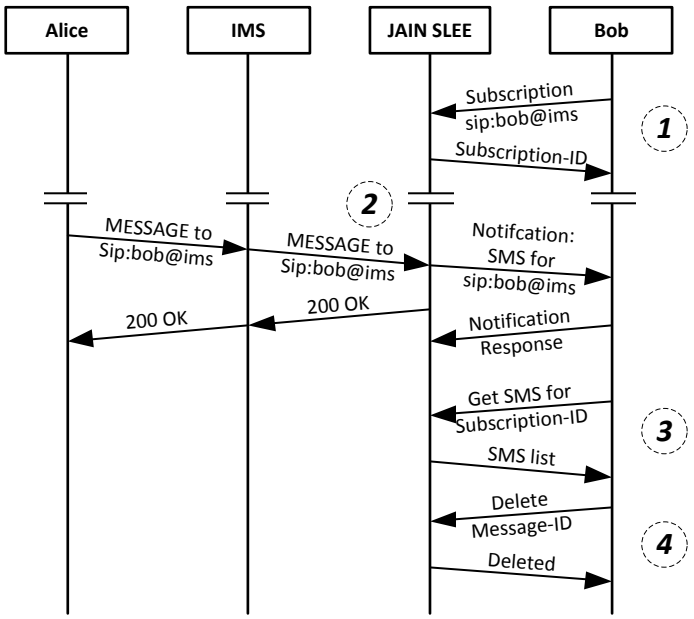

Figure 5.3 Bob subscribes for the address sip:bob@ims.test. Afterwards Alice sends a new SMS to this address towards the IMS which forwards it to the application server. The service looks up machting subscriptions. As Bob has subscribed to the address, Alice gets a success message. Defined by the subscription, Bob gets a notification and confirms it. Then Bob requests all new messages. Afterwards Bob deletes the new one.

Receive. To receive SMS, it is mandatory to set up a message notification subscribtion first (see 1 at figure 5.3). Without any subscription, the service will not store any messages. On each incoming message, our service checks against the subscriptions if it has to store the message. These stored messages can then be queried within the receive interface (see 3 at figure 5.3).

Messages are deleted with their corresponding subscription- and message-id (see 4 at figure 5.3); Note: Against the Parlay X specification, OneAPI no longer defines automatic deletion of messages after having received them.

5.2 OneAPI Library and Events

The different services provide several function calls. These are modeled as events inside the SLEE in a request-response manner. These events are represented by objects. The parameters or return values are described as member variables. The required parameters are defined in their constructors, the optional ones can be adjusted via seperate set-methods.

Many events have some data objects like `NotificationFormat` or `CallbackReference` in common. These do not need to be declared as events as they are just used as data objects inside of the events.

The existing restructured Parlay X library has proven of value. It presents a good structure with one package per service and sub-packages for the different interfaces

(as defined in the Parlay X specification). But not all OneAPI function calls can directly be mapped on related Parlay X methods.
One example where Parlax X is different to OneAPI is the Parlay X method `GetReceivedSms` against OneAPI `GetInboundSms` at the Short Messaging Service. Both enable the user to retrieve received messages to a certain subscription, but the Parlay X specification requires to delete the received messages after having retrieved it, whereas at OneAPI one needs to delete these messages explicitly. Note: There are some more differences i.e. at the Payment service, where OneAPI uses more information than ParlayX in general. Another fact is, that OneAPI drops some of the methods, that Parlay X provides (i.e. enabling the user to schedule SMS).
Because of these shortcomings, we decided to create a complete new library matching the OneAPI specification. The structure is mostly taken from the old Parlay X library, but it is fitted to OneAPI.

To keep it more general, we in fact introduce two libraries, one representing the actual method calls and common data objects of the different OneAPI methods and another providing interfaces to these to be used as events inside the SLEE. An example is shown at Figure 5.4, here we see the `GetInboundSms` event which is an actual implementation of the interface of this OneAPI method call.
This also corresponds to our need to transform the data with JAXB and the JSON library. Because of the restrictions of the application server, these frameworks cannot make use of event objects directly.

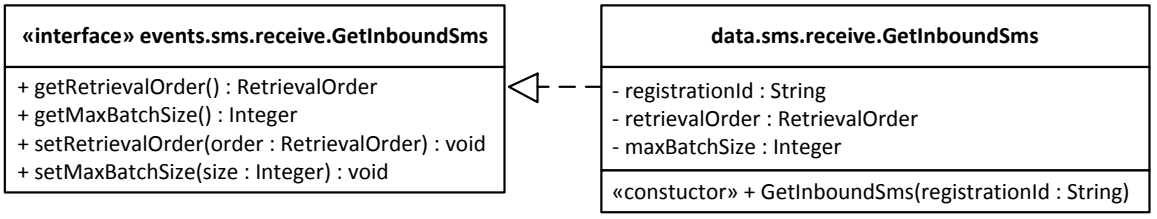

Figure 5.4 The SLEE uses interfaces as event types. The actual event object is an implementation of these.

5.3 Gateway

The main task to build the REST interface is to translate the internal specific service events to REST messages and the other way round, to give the REST messages a semantic. This is done by a gateway. It has two different directions. On the one hand, it listens for new requests coming from the services destined to be send to an external user (e.g. an SMS delivery notification) and plays back the response to the service. On the other hand, it listens for new incoming REST requests from an external user which is the most usual case and returns the answer. These requests have to be translated into specific service request events to fire them into the system.

A **REST request** is an actual HTTP request. The method of the HTTP header defines the action. The HTTP request URI together with the requested hostname defines the resource in a natural way. This is for example for a certain inbound message at the SMS service `http://example.com/1/smsmessaging/inbound/registrations/{registrationId}/messages/{messageId}`. Typically the POST and PUT methods need to provide more information. This is done by encoding this additional information as XML or JSON into the message body. For convenience we provide such an event with an encapsulated HTTP request.

A **REST response** is modeled as a seperate event as well. Such a response is encapsulating an HTTP response like at request. Responses give information if the requested action failed or succeeded. For this purpose, there are typical HTTP status codes used.
The succeeding responses OK and Created typically carry back the requested content in their message body. This can include additional information. OneAPI Exceptions like `ServiceException` or `PolicyException` get the statuscode 400 Bad Request with detailed failure information.

5.3.1 Formats

The OneAPI REST interface requires to support at least two different types of payload data formats. These are XML and JSON.

```xml
<?xml version="1.0" encoding="
   UTF-8" ?>
<Animals>
   <dog>
      <name attr="1234">Rufus<
         /name>
      <breed>labrador</breed>
   </dog>
   <dog>
      <name>Marty</name>
      <breed>whippet</breed>
   </dog>
   <dog/>
   <cat name="Matilda"/>
   <a/>
</Animals>
```

```json
{"Animals": {
   "a": null,
   "cat": {"name": "Matilda"},
   "dog": [
      { "breed": "labrador",
        "name": { "$t": "Rufus",
                  "attr": "1234"}
      },
      { "breed": "whippet",
        "a": null,
        "name": "Marty"
      },
      null
   ]
 }
}
```

Figure 5.5 An example XML file (left column) and the same revisited in JSON data format (right column).

5.3.1.1 XML

The Extensible Markup Language defined by the W3C [4] is nowadays a well known common standard markup language for exchanging data. The structure is easy to understand and quite simple. An XML document starts with a declaration providing

some information about the document itself. Typically such a document consists of different (nested) elements consisting each of a start and end delimiter (tag) in a tree-like structure. Each element can additionally have attributes and a body with some content (including more elements).
A short example document is found at Figure 5.5.

5.3.1.2 JSON

The IETF has published RFC 4627 [17] to the lightweight data exchange format JSON which is a subset of the Javascript language defined by the ECMA [17]. A JSON document consists of a number of key:value pairs (objects) or lists (arrays) of values. A value is either a string, float-number, object, array or a constant (true, false or null). JSON documents in general are ordered alphebetically by their key. The endcoding of JSON content is supposed to be Unicode.
One benefit of this format compared to XML is the less overhead. To make the idea clear, the revisited XML example is shown in the JSON format at Figure 5.5. Note: There are 204 characters for XML against 164 characters for JSON (without unneccessary whitespaces).

5.3.2 Message Translation

The gateway contains a seperate unit to translate REST messages into specific events and the other way round.
At translation we have to distinguish between different cases. These are incoming REST requests from an external user with the corresponding response and outgoing requests from a service towards an end user with its response.

For **incoming requests** we have to determine the specific event type out of the HTTP header information and the message body. This is primarily done via the request's HTTP method and URI. If this is not sufficient, we have to look at the payload as well. In most cases, the combination of URI and method uniquely identifies the type. To build such a specific request in this case, all required information is gathered by tokenizing the URI and converting the message body (if it has one) into a java object.

Responses to these incoming requests can be translated into a REST request with ease. The specific event by itself defines what HTTP response code to use. Responses typically contain a `Location` header field to give information about the requested resource back to the end user. This must be build up carefully out of the information of the specific response event. Moreover a response normally contains a representation of the requested resource as well, i.e. the request's message body. Our design of the OneAPI library defines that the originating request is encapsulated in the specific response. With this information, the original message body can be translated according to the specified `Accept-Type` header field of the previous request.

Outgoing requets can be translated into a REST request in the same way as the responses to incoming requests. The method and URI are defined by the specific request. The format and destination of this request is determined by the address which is supposed to be given by the service. The SLEE enables to set this address for each event. According to this information, the message body will be encoded properly as requested.

With responses to these outgoing request, we have to face the problem, that some REST responses do not provide enough information to distinguish between different specific events. These are for example 204 No-Content responses. To set up the correct answer, we have to store the context, i.e. the corresponding originating request. With the knowledge about the previous request, we are able to give the right specific response.

The actual payload can be encoded in the two mentioned formats. For transformation the following methods are used.

XML. The translation from and to XML is done via the Java Architecture for XML Binding (JAXB). JAXB allows to give annotations directy in a Java class defining information about its corresponding XML representation. With this framework Java objects can be marshaled into XML and XML content unmarshaled into a java object with ease. One has to create a new single context giving information which classes are processed. Afterwards marshallers and unmarshallers can be derived from this context. For each process one un-/marshaller has to be instantiated as they cannot be used at the same time by more than one thread.

Compared to other methods (e.g. Simple API for XML (SAX)) for this translation, JAXB is more comfortable and faster by the cost of greater memory consumption (see [13] for more information).

JSON. For the translation between java objects and JSON, we use the JSON-lib [20]. It provides as simple methods as JAXB for translation. One shortcoming is, that JSON-lib does not allow to define the information via annotations but uses the member-variables and their names of a java class directly. This restrcition does not matter in our case as we have defined our OneAPI library as that.

To translate from a Java object to JSON, one has just to call a static serialization method. The other way round needs some more infomation, namely a config providing information which class to deserialize.

5.3.3 Outgoing Requests

A brief overview of the behaviour at outgoing requests can be found at figure 5.6. Outgoing requests are triggered by the internal services. The gateway handles service specific *OneAPIRequest* events. These events are declared as initial at the SLEE, such that each occurence triggers a new SBB entity to handle it. The SBB contains a method `onOneAPIRequest` for each of these specific request events which delegates this event directly to a `sendRestRequest` procedure.
This procedure is responsible for several steps:

1. Attach to the activity on which the new *OneAPIRequest* event arrived and store the context in order to receive all further events on this process.

2. Store the originating event to determine what kind of specific answer to give later on. This has to be done, because many REST responses just contain a 204 No-Content response code which will be ambiguous otherwise.

3. Create a new *RestRequest* by using the content translator. Important here is, that both, the target address and format are defined by the event's context. The SLEE provides a method for defining a target address for each fired event and the service is supposed to give this information. The content translator takes these arguments and encodes the message body accordingly. Thus, the destination is defined by the REST request respectively the encapsulated HTTP request itself. This means the resource adaptor does not need any further information except the actual REST request.

4. Invoke sending the request by the resource adaptor. This will return a new *OutgoingRestActivity* on which we have to attach and which is stored as well to keep track of all collaborating activities (one to the service and one to the external user).

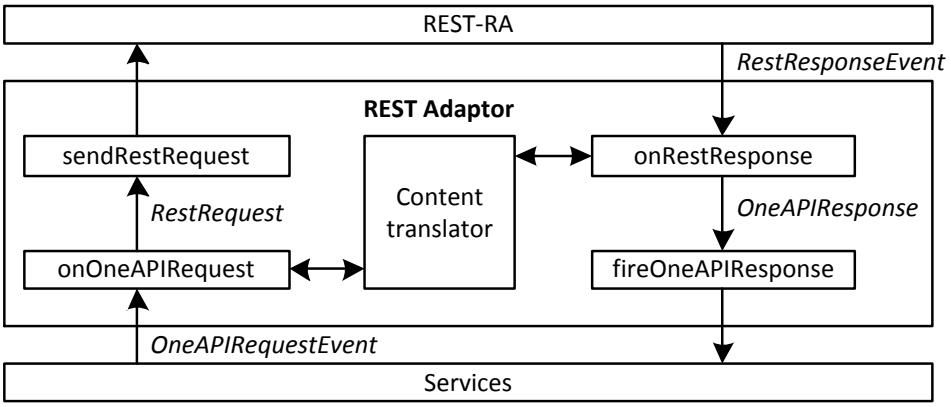

Figure 5.6 A service triggers an outgoing request by firing a specific *OneAPIRequest* event. The gateway translates this to REST and invokes the sending at the resource adaptor which will then deliver it to the end user. The answer will be received by a new *RestResponse* event thrown by the RA after having received the message from the end user. This is translated into a specific *OneAPIResponse* event and fired back into the system.

Now the gateway has stored the activity towards the internal service and another to the external user as well as the originating event. It is awaiting a reply by the end user. As soon as this is received by the resource adaptor, the RA fires a new *RestResponse* event on the corresponding activity. This invokes the handler-method `onRestResponse` of the gateway which is responsible for translating this response into the appropriate specific answer and giving it back into the system. If the *RestResponse* is such a mentioned No-Content answer, the handler looks up the stored prior event and generates the correct specific response. Otherwise it lets the content translator generate the specific response. Determined by the type of the event, the adaptor then fires the *OneAPIResponse* on the activity towards the service. Having done these steps, it detaches from all activities.
Note: The outgoing activity started by the RA will be terminated automatically by

the SLEE invoking a termination method of the RA, since there are no more entities interested in it.

5.3.4 Incoming Requests

Incoming requets are triggered by an external user. A brief overview of this procedure can be found at figure 5.7. As soon as the reource adaptor gets a new connection and receives a new request, it throws a new *RestRequest* event. This type is defined as initial at the SLEE and a new instance of the gateway will receive the event and the method `onRestRequest` is invoked. This request can be handled a bit easier. Here is no need to save the originating event since their type is unique - it is more important to determine this type. That task is done by the content translator. Now the adaptor looks up what firing method to use by the type of the specific *OneAPIRequest* and invokes it on the same activity on which the RestRequest event came in.

After attaching to this activity it now waits for the corresponding *OneAPIResponse* event from the responsible service. With the arrival of this specific answer, the handle method for this delegates the event to the `sendRestResponse` procedure which again uses the content translator to transform this into a *RestResponse*. Finally it demands the resource adaptor to send the response back to the end user. After that it detaches from the activity since there is no further job to be done.

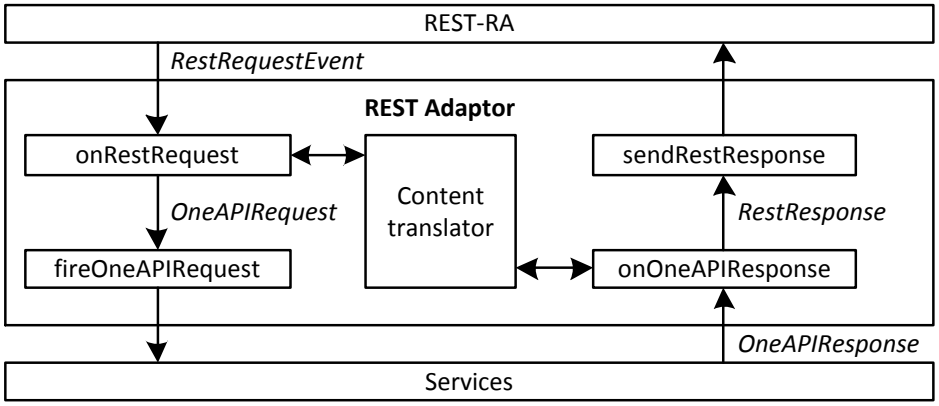

Figure 5.7 After receiving a new *RestRequest*, the resource adaptor throws this as a new event into the system. The gateway translates this into a specific *OneAPIRequest* event and gives it into the system. The responsible service gives back a specific *OneAPIResponse* event which is translated into a *RestResponse*. This response is then sent by the RA back to the end user.

5.4 REST Resource Adaptor

The REST Resource Adaptor is the main interface between the external network and the inside of the application server. It can be split into four packages. A library containing a generic representation of HTTP messages; Event definitions for the REST messages: `RestRequest` and `RestResponse` each encapsulating a HTTP request or response; The interfaces of the RA; and the actual implementation.

Besides a model of HTTP requests and responses with a factory class, the library also contains a lightweight message parser.
The events provide convenience methods for getting important information of the underlying HTTP message. At requests, this information are the method and request-URI; at responses, these are the status code and the corresponding reason phrase. (The actual implementation is left to the package of the RA implementation, since this is the only instance that is supposed to be able to generate these events.)
The adaptor starts up a new threadpool enabling it to listen for new incoming TCP connections. As soon as an external user starts a new socket connection, a listening thread is attached to it. The HTTP parser is then used to set up the new HTTP request. Now an `IncomingRestActivity` is started and a the new `RestRequest` is fired on this activity for being further processed by the systeme, i.e. the gateway. The connection is held open until the `sendResponse` method provided by the activity is invoked and the answer is sent.

A provider interface enables any service of the system to send new requests to an external user. An `OutgoingRestActivity` is created each time a new request from inside of the SLEE is triggered.

To check wehter we have a reasonable REST message is left up to our gateway that translates these into specific events.

6

Evaluation

To show different performance aspects of our interface implementation on the SLEE platform, we have run several tests. The results are expected to show a general overview how the system behaves at different loads.
There are three scenarios covered by the tests. They are chosen to evaluate different parts of our implementation. These are the complete send SMS process triggered by a third party (this means send SMS invocation, delivering towards the IMS and giving the appropriate response) to get an impression of the performance under normale usage including the communication with the IMS; only the send SMS invocation with its corresponding response, but without actual delivering, to evaluate just the interface; and the notification subscription process as this is another common use-case where the IMS is not involved.

6.1 Tools

OneAPI traffic generator. In order to represent a third party that invokes the different services, we introduce a simple traffic generator, that creates unique requests and sends them at a predefined rate (requests per second) towards the SLEE for a specified time. It then awaits the responses. At each sent request, it measures the time until the response is received.

A core instance invokes periodically new client threads within a predefined time period. Such a client manages a OneAPI request invocation, i.e. sending a request and receiving the answer. The core uses the `ThreadPoolExecutor` Java class to manage the client threads for gaining better performance. This threadpool is bound to 300 core threads which should normally be enough. The client thread saves a timestamp at invocation time. When it is run by the executor, it first checks wether a timeout of 60 seconds has already been exceeded. If there is time left, it opens a new socket to send a request and afterwards awaiting the answer. The socket here gets a timeout such that it will shut the socket down exactly 60 seconds after the

stored invocation time. The value of 60 seconds is chosen as a tradeoff between a possibly too low level and unacceptable response times at most telecommunication services.
The current time before sending a request as well as the time of the answer are traced. This enables computing the round trip time (RTT) of this specific OneAPI method call which is given back to the core.
Generally if a response does not arrive within 60 seconds or even does not get send within this time, the request will be dircarded and is considered to be lost. This information is given to the core as well.
At runtime all results of the client threads will be stored in a simple array. After terminating the whole test, the resulting list will be stored encoded in XML.

Receive SMS. For receiving SMS from the invoked send SMS method, we use a simple SIP softphone called Twinkle [6], that is capabale to register at the IMS and to receive and send messages via SIP.

6.2 Setup

For performance evaluation of our REST interfcae, there has been set up a testbed environment at the Chair of Communication and Distributed Systems of RWTH Aachen University. An overview can be found at figure 6.1.

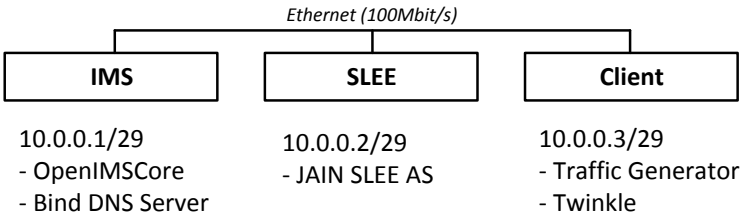

Figure 6.1 The performance evalutation testbed environment. It consists of an IMS server also being responsible for DNS, a SLEE Application Server and a client for generating and receiving traffic.

The test environment consists of two Personal Computers (PCs) each running Ubuntu Maverick (10.10) Server Edition Linux operating system at kernel 2.6.34 and a notebook running Ubuntu Lucid Lynx (10.04) Desktop Edition at kernel 2.6.32. The PCs both have Intel Core 2 Duo 2.4 GHz processors and 2 GB of RAM. The notebook has an Intel Core 2 Duo at 1.8 GHz with 4 GB of RAM. They are connected via a 100Mbit/s network switch.

One of the PCs (IMS) is used for the the HSS and CSCFs of the IMS core. The running implementation is the open source OpenIMSCore published by the Fraunhofer FOKUS institute [2]. In addition to the IMS part, it is also responsible for a local Domain Name System (DNS) server which serves the domain *open-ims.test*. The second PC (SLEE) acts as the SLEE AS. We use the open source Mobicents JAIN SLEE platform [21] at version 2.3.0 which is based on a JBOSS AS [19] at version 5.1.0. The used Java implementation is the OpenJDK [22] at version 6 providing a Java Runtime Environment at version 1.6.0.
The notebook (Client) is in charge of traffic generation with our tool and running the softphone for receiving messages.

6.3 Results

At first initial tests, we found out that the SLEE is varying the processing times at very short testings periods. There seems to be an initial transient phase, thus all tests have been running for 10 minutes. A clean new system without any influences of previous tests was guaranteed by rebooting the whole system each time. To have a good comparable basis for the different testruns, each test scenario has been running 5 times. All presented results here are averages over these 5 testruns. Generally we observe, that the testruns for a certain scenario reveal very similar results.
The console tracer-level has been set to *ERROR*, since we are not interested in any information supposed for debugging as this would probably have a negative impact on the whole system performance.

Having build the traffic generator in Java turned out as a poor idea, at least in the way we did. A big problem is the time resolution. With Java, there is no easy way to get an accurate precision beyond milliseconds. There exists the possibility to use nanoseconds, but seperate tests have shown, that this is not useful at all.
The resolution of milliseconds is no problem while dealing with less than 100 incovations per second as it is the case at SMS delivery. However, the resolution becomes a serious problem at an increasing number of invocations per second beyond this treshold. As a matter of fact, we in particular were not able to set the rate very accurate before the test. Nevertheless, it can be calculated afterwards. It might be possible to get generate other rates by changing the test execution time, but we did not evaluate on this idea any further, since it changes the test conditions.

Another problem occurrs at heavy load. We have noticed that the SLEE gets socket errors mostly at nearly the end of a testrun. These failures result automatically in losses in our statistics as a new socket connection might get refused.
We assume that more excatly these exceptions rise when the client closes connections because of an exceeded timeout. The error becomes visible at the RA when it tries to open a new server socket. This fails with an exception triggered by the OS saying that there are too many files opened.
Further analyzing has shown, that many sockets stay in the *CLOSE_WAIT* state which means, that the socket has been closed, but there is still an acknowledgement missing or any thread is still referencing to this socket. The connection stays in this state as long as the application is running. As a matter of this, these sockets remain blocked and the maximum number of sockets exceeds at some point.
In general, this problem often occurs at server applications opening and closing many sockets very quickly like it is the fact here. The most common reason is simply a fault in the application. Thus, we assume that in some cases, a socket is not properly closed by the resource adaptor. As this failure only occurs at heavy load where the client gets timeouts.
An attempt to get the sockets cleanly closed, is to set a client socket to remain in the *TIME_WAIT* state for a short time period after closing. Another try was to set a lower socket timout at the server. Furthermore, as a possible workaround, we tried to adjust the systems socket limit which is under linux typically set to 1024 (`ulimit -n`). Neither changed anything and this serious bug has not been resolved while working on this thesis.

Either way, we proceded testing our implementation, since we only have to face the mentioned problem at very heavy load.
Remember: This failure is automatically counted as a loss.

6.3.1 Sending SMS

There necessity to test at least the whole vertical communication through the system. This goal is achieved by invoking a request to send an SMS. This use case is quite simple. It includes a third party, that sends the actual request; the OneAPI interface translating the request into a specific event to be received by the SMS service. This service sends back a response and handles the SIP message delivery towards the IMS keeping track of the delivery status.
This setup does not scope the delivering time towards the client which receives the messages from the IMS.

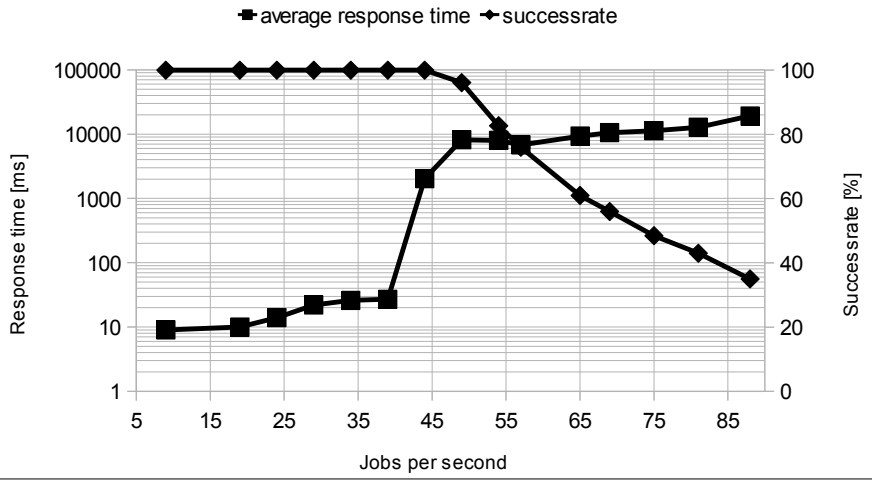

Figure 6.2 Result of send SMS and delivery. This diagram shows the result of the different testruns at a certain rate of requests per second. The left y-axis denotes the average response time of the system in milliseconds, the right y-axis denotes the success rate of the requests in percent. The system can considered as stable at loads up to 40 invocations per second.

A first impression of the test results for sending an SMS can be found at figure 6.2. The system shows a quite stable behavior up to 39 requests per second keeping the response time below 50ms. It is still capable to deal with 44 requests per second, but the response time increases massively to slightly below 2 seconds.
We can observe first losses at a threshold of 49 requests per second where at the same time, the response time raises to over 8 seconds. Increasing the load leads to a slightly increasing response time, whereas the absolute number of successful requests decreases. Even at a high load of 75 requests per second, the system still reacts to our requests in an acceptable time of about 11 seconds, or at 88 jobs per second in a time below 20 seconds which is still fairly acceptable for sending an SMS as it is no time critical service.

Load	*Avg.*	Q_3	*Stddev.*	*Success*	*Abs. succ.*
34	28 *ms*	10 *ms*	154	100.00 %	20,449
39	42 *ms*	14 *ms*	213	100.00 %	23,699
44	1,742 *ms*	115 *ms*	4,507	100.00 %	26,902
49	8,174 *ms*	9,402 *ms*	14,678	96.13 %	28,435
54	9,387 *ms*	13,602 *ms*	15,212	82.69 %	27,012
57	6,786 *ms*	12,890 *ms*	12,806	75.71 %	26,261
75	11,292 *ms*	16,581 *ms*	14,898	48.44 %	21,836
88	19,274 *ms*	30,234 *ms*	18,988	34.77 %	18,435

Table 6.1 Detailed results of the send SMS and delivery tests. At higher loads, the response time gets higher and much requests get lost.

Another interesting fact is, that the upper quartile (Q_3) of the response time at load 44 remains at about 115ms which is far from the average. So, at some point, the system obviously cannot handle all requests in time and they are getting queued. Either the system serves such a queued request or the client terminates it after its 60 second timeout. This corresponds to the fact that the response time increases heavily at loads over 50 and, simultaneaously, the number of lost requests increases.

Note: All messages are stored in a Profile Table which is cleared before each testrun. Messages exceeding a predefined age are deleted by periodic maintenance of the short messaging service, but as the number of messages increases, the lookup for a certain message becomes more costly which has to be done when updating the delivery status. This might slow down the service at running for longer times. The periodic maintenancing as such needs cpu time as well while running.

6.3.2 SMS invocation

After testing the whole SMS sending process, we are now interested only in the invocation of sending an SMS. This test is meant to get an impression of the system's behavior when we just stress the actual REST interface instead of giving it additional responsibility of sending the SMS.
For this purpose, the SBB providing the service has been modified to receive a request, store it internally and give the related response as usual, but not to start a new activity for the actual sending of a SIP message towards the IMS. Thus, the service's business is solely to store a message and give a response back.

An overview of the average response times of the SLEE for a certain number of invocations per second is depicted at figure 6.3.
We observe, that the system manages up to 158 requests per second without a prominent increse of the average response time or any losses. This behavior changes at a load of 189 requests per second. Here the response time raises to over 4 seconds and very few requests get a timeout. Taking a closer look on the upper quartile reveals, that most messages are processed in below 16ms. We have seen this case at the first test where the SMS have been delivered as well, as the upper quartile shows a very low value against the average. This means most messages are successfully processed, but some get queued and it takes relatively long before these get processed.

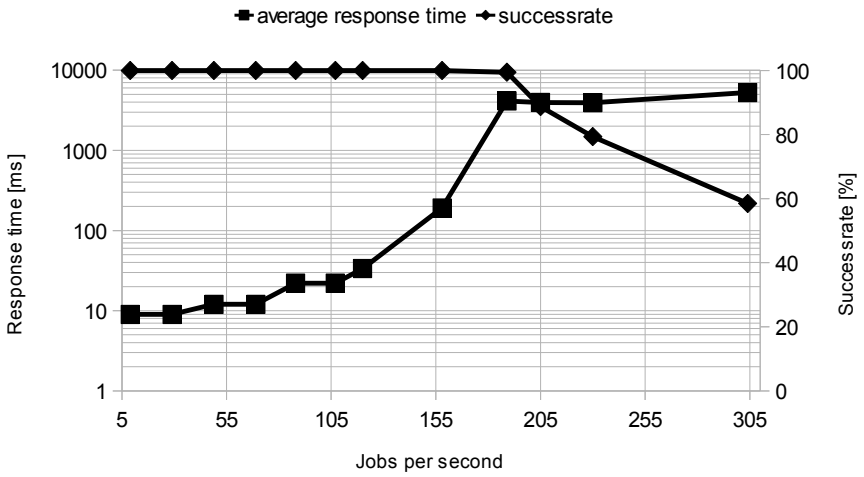

Figure 6.3 Test results of the send SMS invocation tests. This diagram shows the result of the different testruns at a certain rate of requests per second. The left y-axis denotes the average response time of the system in milliseconds, the right y-axis denotes the success rate of the requests in percent. The system remains stable at managing up to 155 requests per second. At further increasing of the load, more requests get lost, but the response time for the successful requests just increases slowly.

Load	*Avg.*	Q_3	*Stddev.*	*Success*	*Abs. succ.*
107	22 *ms*	6 *ms*	155	100.00 %	64,490
158	191 *ms*	5 *ms*	834	100.00 %	95,083
189	4,115 *ms*	16 *ms*	11,407	99.66 %	113,043
209	3,966 *ms*	771 *ms*	10,650	87.71 %	110,367
230	3,953 *ms*	174 *ms*	10,753	79.01 %	109,291
304	5,262 *ms*	7,147 *ms*	10,733	58.49 %	106,831

Table 6.2 Detailed results of the send SMS invocation tests when the system becomes unstable. As the average processing time and standard deviation do not change intensely, the system only handles a roughly fixed number of requests.

Higher loads at over 209 requests per second result in nearly the same amount of about 11,000 successfully processed messages. The average that time as well as the standard deviation do not change considerable. The upper quartiles show, that most of these successing messages have response times below 1 second. This value increases significantly to over 7 seconds at a load of 304 which acceptable for sending an SMS.

The oberservations seem to proof our assumption about the networking part of the implementation. Whenever we do not recognize any losses or in other words timeouts, the system reacts and answers very fast; there are response times below 200ms on average as well as an upper quartile of 5ms at load 158. The average processing time heavily increases as soon as there emerge losses, but note that the upper quartile remains relatively low (load 209).
We know, that some sockets (for whatever reason) are not closed properly and stay in the blocked *CLOSE_WAIT* state. This leads to a smaller amount of possible open sockets on the whole OS. Regardless of this, the RA accepts new socket connections as long as possible which again may lead to more blocked sockets. We suppose, that as a matter of this, there are no more or few additional connections possible at some point. Thus, further requests get rejected and are counted as a loss.
We have randomly examined the number of such blocked sockets while running such a critical test. The amount of sockets remaining in the *CLOSE_WAIT* state is very high (e.g. 500, 900) which also proofs our assumption.

Apart from this, the actual interface seems to be quite efficient which can be observed at the cases (up to nearly 189 requests per second) where no losses occur.

6.3.3 Notification subscription

Another typical use case will be the subscription for certain SMS. This use case is very similar to the send SMS invocation, but is slightly more involved. Triggered by an imcoming request, the service checks the Profile Table that holds all subscriptions against the specified criteria as they are not allowed to overlap. If no overlapping is found, it stores the subscription. Afterwards it gives an appropriate response (success or failure).

A first impression of the test restult gives the diagram at figure 6.4. The system is able to manage loads up to 69 requests per second where it keeps the average response time below 22ms. It can be considered as stable at higher loads up so 88 requests per second as there are no losses, but the response time increaes to nearly 5s. This value is still very acceptable for subscribtions to receive certain SMS as they are usually set up for longer terms.

Heavier loads result in a higher number of lost requests. As this test is very similar to the send SMS invocation test, it can be observed that it reveals the same behavior. The only difference shows up at the threshold where the system becomes unstable. A detailed view on the measurement results gives table 6.3.
An answer to the lower threshold is, that the responsibilities at a subscription request are more involved. Especially in comparison to the invocation test where the service is solely responsible to give a response, this service has to check for overlapping subscriptions in the Profile Table before returning a response which takes longer.

The fact that higher loads roughly only causes the number of lost requests to grow, has already been discussed at the send SMS invocation test.

Again, we observe fast reaction times in the cases where we do not recognize any losses up to 69 requests per second. This does not directly proof that the inferface is quite fast as the load is much lower here compared to the invocation test, but it also does not contradict.

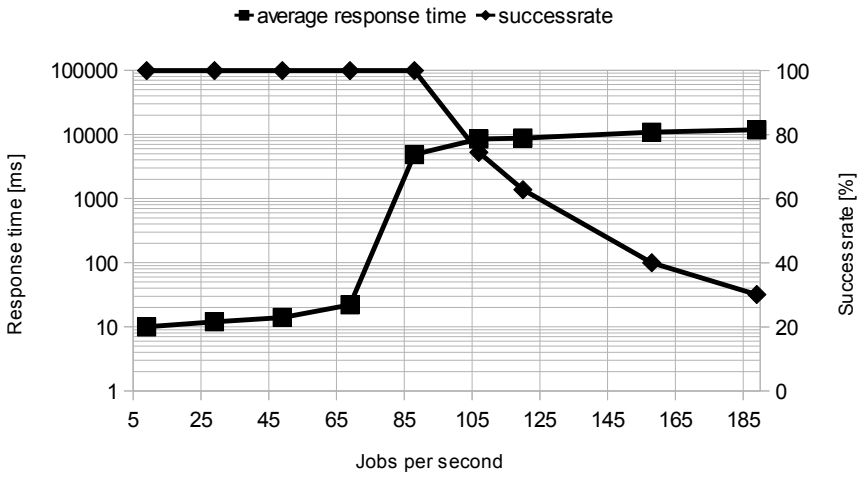

Figure 6.4 Test results of the subscription test. This diagram shows the result of the different testruns at a certain rate of requests per second. The left y-axis denotes the average response time of the system in milliseconds, the right y-axis denotes the success rate of the requests in percent. The system manages up to 85 requests per second without any losses. At higher loads, more requests get lost, but the response time for the successful request remains stable

Load	*Avg.*	Q_3	*Stddev.*	*Success*	*Abs. succ.*
29	12 *ms*	12 *ms*	28.02	100.00 %	17,966
69	22 *ms*	13 *ms*	124.23	100.00 %	41,882
88	4,861 *ms*	2,830 *ms*	10,070,80	100.00 %	53,066
107	8,500 *ms*	10,246 *ms*	15,030.53	74.49 %	47,935
120	8,768 *ms*	11,397 *ms*	15,202.18	62.83 %	45,255
158	10,648 *ms*	15,153 *ms*	16,950.58	40.02 %	34,260

Table 6.3 Detailed results of the subscription tests when the system becomes unstable. The system is able to hanlde up to 88 requests per second without losses whereas the response time remains low. As already seen at the send SMS invocation test, the standard deviation and response time remain on the same level, but more requests get lost.

7

Conclusion

We have introduced information and latest development about telecommunication services and networks. The proccess of merging these networks together to all-IP-based communication via the IMS architecture is a convenient way to administrate users even of external IMS networks. The design with the control layer has already proofed its benefits at GSM.
The architecture also allows to integrate application servers with ease. They enable a provider to apply several new services built ontop of the actual network. This shows up another benefit of the IP-based network, since the single language for signaling is SIP. SIP is a well known protocol and an accpeted standard for these purposes.

To combine telecommunication services and the web, we have introduced Web Services in general as well as Web Services in a RESTful design, namely RESTful Web Services. The most important standards have been presented which are Parlay X and OneAPI. Parlay X is deprecated and OneAPI is being in further development and the newest standard for telecommunication services. It is defines some elemental telecommunication services as a RESTful Web Services.
The current OneAPI version 2.0 only provides the services SMS, MMS, Location, Payment, Voice Call Control, Data Connection Profile and Device Capability, but version 3.0 will extend this palette.

For integrating OneAPI into the IMS, we use a JAIN SLEE application server. This event driven platform is meant to be used at telecommunication applications, since it claims to be able to handle many events in parallel very fast.
The SLEE platform provides a model to create single small application blocks (SBBs) that can be plugged togehter into a complete service. It also affords the possibility to integrate custom external resources via Resource Adaptors.

Our OneAPI REST interface has been built for such a SLEE AS. This includes a new Resource Adaptor for managing REST requests and responses which are in fact HTTP requests or responses, respectively. This RA acts as an interface between the external world and the inside of the SLEE.
Function calls and their return value are represented as events in the SLEE. To

distinguish between different OneAPI function calls, we do not use a generic OneAPI-request and response, but have created a whole library containing for each method of each service a request and a response class.
Thus, generic REST messages from and towards the RA have to be translated into these specific events. This task is done by the gateway module which handles the whole translation.

To show the complete communication from an end user to the IMS, we have also implemented the Short Messaging Service. This service uses a SIP-RA to communicate with the IMS. For cummunicating with an external user through the gateway and via the REST interface, it just has to receive and fire the OneAPI specific events for the SMS service.

The performance evaluation has revealed some serious problems. On the one hand, the approach of our traffic generator does not seem to be the best idea because of problems with the time resolution beyond milliseconds. This resulted in some shortcomings of our testruns as we were not able to set all desired request rates. Nevertheless, it worked and the results are still very useful.
On the other hand a problem with Resource Adaptor's networking part could not be resolved while this thesis duration. As soon as requests get lost because the traffic generator does not get an answer before a timeout, the client closes the corresponding socket. We observed that at some point the server seem to not recognize the closed connection correctly and the server socket remains in a blocked state as long as the application (i.e. the SLEE itself) is running. This leads to reaching the operating system's maximum number of sockets where most of the used slots are blocked. Thus, the server is not able to open new connections anymore and subsequent request attempts by a client may get refused.
Setting a lower socket timeout at the RA or increasing the maximum number of filepointers on the OS did not help to eleminate this problem. Another approach setting the client to stay a short period in a wait state when a socket is closed, did not help either. This issue is still unsolved and will need further investigation.

The evaluation results especially for the send SMS invocation seem to proof our assumption on this problem. The performance of the interface turns out as being quite good with low response times as long as the server does not produce these blocked sockets. The current bottleneck is definitily located at the mentioned problem.
We further assume, that the server will be capable of handling higher loads when this bug is resolved. This assumption is made due to the fact, that at both, the invocation and subscription test, the absolute number of successful requests remains on the same level at some point. The same applies for the response time and standard deviation which are still very acceptable for the tested service parts.

Future Work. First of all, the mentioned socket bug has to be resolved and as a consequence, maybe the networking part of the RA has to be totally redesigned. After getting rid of this problem, the performance tests should be rerun. Here we will have to face the time resolution problem of our traffic generator. It has to be reconsidered as the number of possible successful requests will probably increase without the socket bug. There will be the need to set different request rates per second in a precise way to determine thresholds of the system behavior more exactly.

Our RA has some more shortcomings. The HTTP implementation is very simple, which means e.g. chunking or keepalive connections are not supported right now. The RA also does not implement HTTPS which may be desired for security reasons. Security and authentication in general should be considered in the future, since we ignored these aspects till now.

We have introduced an implementation of the Short Messaging Service. Currently this service is packed completely into one single SBB as earlier work on the SLEE platform ([9], [15]) pretend that splitting responsibilities results in massive internal communication overhead by creating new NullActivities. We currently use such NullAcitvities at the SMS service and results show that they do not seem to slow down the system in a very considerable way. There should be spent some time on this open question.
Depending on the answer to this question, the implementation possibly should be splitted up into several reusable blocks as the SLEE proposes such a design.

The whole interface becomes more interesting having more services implemented. Building new services will gradually become easier with an increasing number of such logical applicaton blocks. A higher number of miscellaneous key-services will lead to the ability to create far more sophisticated service compositions.

Bibliography

[1] *Parlay X Web Services (ETSI Standard 202 504)*, May 2008.

[2] Open IMS Core project website, March 2011. `http://www.openimscore.org/`.

[3] The seekda! Web Service search engine, March 2011. `http://webservices.seekda.com`.

[4] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. *Extensible Markup Language (XML) 1.1*, 2nd ed. World Wide Web Consotium, http://www.w3.org, August 2006. `http://www.w3.org/TR/2006/REC-xml11-20060816/`, March 2010.

[5] Carlin, J. M. E. *Context-Aware Service Provisioning in Converging Networks*, 1st ed. Shaker Verlag GmbH, November 2010.

[6] De Boer, M. The Twinkle SIP softphone project website, March 2011. `http://www.twinklephone.com`.

[7] Ericsson. *Introduction to IMS*, 2007. White Paper.

[8] Fielding, R. T., and Taylor, R. N. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology 2*, 2 (May 2002), 115–150.

[9] Herpertz, R. A Platform for Exposing Telecommunication Capabilities in Service-Oriented Environments. Master's thesis, RWTH Aachen University, June 2010.

[10] Open Mobile Alliance. *OneAPI Profile of ParlayREST Web Services*, August 2010. v1.0.

[11] Open Mobile Alliance. GSMA-OneAPI project website, March 2011. `http://github.com/OneAPI/GSMA-OneAPI`.

[12] Open Mobile Alliance. *RESTful bindings for Parlay X Web Services*, January 2011.

[13] Ort, E., and Mehta, B. Java Architecture for XML Binding (JAXB). *Oracle Technology Network* (March 2003). `http://java.sun.com/developer/technicalArticles/WebServices/jaxb`, March 2010.

[14] Richardson, L., and Ruby, S. *RESTful Web Services*, 1st ed. O'Reilly Media, Inc., May 2007.

[15] Rugina, A. A Platform for Event-Based Provisioning of NGN Services. Master's thesis, RWTH Aachen University, March 2010.

[16] Sun Microsystems and Open Cloud. *JAIN SLEE (JSLEE) 1.1 Specification*, final release ed. Sun Microsystems Inc., 4150 Network Circle, Santa Clara, CA 95054 USA and Open Cloud, 140 Cambridge Science Park, Milton Road, Cambridge, CB4 0GF United Kingdom, 2008.

[17] The Ecma working group. *ECMAScript Language Specification*, 5th ed. Ecma International, Rue de Rhône 114, CH-1205 Geneva, December 2009. `http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf`.

[18] The Jackson community. Jackson High-performance JSON processor website, March 2011. `http://jackson.codehaus.org/`.

[19] The JBoss community. The JBOSS Application Server project website, March 2011. `http://www.jboss.org/jbossas`.

[20] The JSON-lib community. The JSON-lib project website, March 2011. `http://json-lib.sourceforge.net`.

[21] The Mobicents community. The Mobicents JAIN SLEE project website, March 2011. `http://www.mobicents.org/slee/intro.html`.

[22] The OpenJDK community. The OpenJDK project website, March 2011. `http://openjdk.java.net/projects/jdk6`.