University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Implementation of a Single Funding Point Charging Mechanism (SFP-CM)

Cyrill Pedol
Zurich, Switzerland
Student ID: 07-728-207

Supervisor: Christos Tsiaras, Andri Lareida
Date of Submission: 30.11.2012

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Organizations like universities often belong to a federated system, where members of an organization are given the possibility of using a service by authenticating via an Authentication and Authorization Infrastructure (AAI). However, when a member of organization A would like to use a service at organization B, the procedure of charging that member is not a trivial process. The solution proposed in this work is the so called Single Funding Point Charging Mechanism (SFP-CM).

This paper represents an approach of how to implement the SFP-CM. The main goal is to describe an overall architecture in a top-down manner by starting with rather abstract components and getting more and more granular. Thus, the general idea of the implementation will be presented as well as a technical introduction to the prototype implementation that has been developed in order to proof the concepts.

As is shown in this work, the proposed mechanism is generally realizable as it has been designed. In some cases implementation specific changes have been necessary in order to avoid unnecessary difficulties.

This work proposes some improvements to the Single Funding Point Charging Architecture (SFP-CA) proposed at [15] and also discusses points that need some further research. However the implementation presented here still remains a prototype that is not suited for practical use. Thus, an industrial solution is part of future work.

Zusammenfassung

Organisationen wie Universitäten gehören oft einem Netzwerk aus Institutionen an, wobei diese ihren Mitgliedern durch eine Authentifizierungs- und Authorisierungs-Infrastruktur (AAI) Zugang zu Services gewähren. Dabei soll gewährleistet werden, dass Mitglieder einer Organisation A einen Service der Organisation B benutzen können, womit das Verrechnen zu einem nicht trivialen Prozess wird. Eine vorgeschlagene Lösung dieses Problems ist der sogenannte Single Funding Point Charging Mechanism (SFP-CM).

Diese Arbeit soll einen möglichen Weg einer Implementierung dieses SFP-CM aufzeigen. Das Hauptziel ist die Beschreibung einer gesamthaften Architektur, welche dem top-down Schema folgen soll. Angefangen bei den eher abstrakten Komponenten soll die Beschreibung immer detaillierter werden. Damit wird die generelle Idee erklärt, aber auch auf die technischen Gegebenheiten der Prototyp-Implementierung eingegangen. Der Prototyp stellt also ein "proof-of-concept" dar.

Die Arbeit zeiget, dass der SFP-CM grundsätzlich umsetzbar ist, auch wenn in gewissen Punkten implementierungsspezifische Verbesserungsvorschläge gemacht werden mussten. Anpassungen des Designs verfolgten oftmals das Ziel, die Programmierung an sich um unnötige Schwierigkeiten zu erleichtern.

In dieser Arbeit wurden Verbesserungsvorschläge für die in [15] vorgeschlagene Single Funding Point Charging Architecture (SFP-CA) gezeigt, und es wurde auch auf Punkte hingewiesen, die weiterer Forschung bedürfen. Die Implementierung bleibt aber nach wie vor ein Prototyp und ist nicht für den Gebrauch in der Praxis vorgesehen. In diesem Sinne ist eine industrielle Version als Teil künftiger Projekte zu sehen.

Acknowledgments

I would like to thank Prof. Dr. Burkhard Stiller and the Communication Systems Group at the University of Zurich because they gave me the possibility of working on that thesis.

I especially thank my supervisors of this project, Christos Tsiaras and Andri Lareida for the valuable guidance and advice. Without their competent support this project would not have been possible.

Contents

Abstract **i**

Zusammenfassung **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of Work 1

1.3 Thesis Outline 2

2 Related Work **3**

3 Preconditions **5**

3.1 Preconditions 5

4 Software Environment **7**

4.1 Distributed Character of the SFP-CM 7

5 Architecture Overview **9**

5.1 Charging Manager 10

5.2 Charging Manager & Remoting 10

5.3 Charging Rate Manager 11

5.4 Service Provider Manager 12

5.5 Account Balance Manager 12

6 Communication between Components **13**

6.1 Registration 14

6.2 Exchange Formats 14

6.3 Charging Mechanism 14

6.4 Events vs. Sessions 14

6.5 Session Management 16

6.6 Credits 18

6.7 Discounts & Constrains 19

6.8 Charging Objects 21

6.9 Complete Charging Process 22

7 Security **25**

7.1 Security Considerations 25

8 Implementation **27**

8.1 Project Architecture 27

8.2 Modules 28

8.3 Spring Contexts 29

8.4 Persistence 31

8.5 Building the Artifacts 34

8.6 Usage 34

8.7 GUI 36

9 Evaluation **37**

10 Summary and Conclusions **39**

Bibliography **39**

Abbreviations **43**

Glossary **45**

List of Figures **45**

List of Listings **47**

A Installation Guidelines **51**

B Contents of the CD **53**

Chapter 1

Introduction

1.1 Motivation

Organizations like universities often belong to a federated system, where members of an organization are given the possibility of using a service by authenticating via an Authentication and Authorization Infrastructure (AAI). However, when a member of organization A would like to use a service at organization B, the procedure of charging that member is not a trivial process. The solution proposed in this work is the so called Single Funding Point Charging Mechanism (SFP-CM) and is built on the Single Funding Point Charging Architecture (SFP-CA), which is described in [15]. As its name implies, this architecture provides the possibility of charging services in federated systems using a single funding point, instead of multiple funding points.

1.2 Description of Work

Based on the idea provided in [15] and with the help of its main author Christos Tsiaras, this paper aims at describing the corresponding software specification. The elaboration of the specification will follow a top-down approach, hence starting with the abstract system components and selecting more fine grained components later on. In addition, various third-party libraries will be examined. This is because the specification tries to reuse as much external libraries as possible, in order to reduce the amount of redundant work and having a more stable groundwork when using well-proven libraries. The ideas and concepts that evolved from this work have been applied to a prototype implementation. Hence, the prototype primarily represents a proof of concept, proving that the core functionality is implementable and how it is realizable.

1.3 Thesis Outline

The paper is structured as follows: After mentioning certain related work, some preconditions the software has to fulfill will be clarified. This is followed by a chapter that describes the general environment as well as some problems that might have to be faced when trying to embed the software in an existing environment. The next chapter moves away from the environment and explains the rough component-architecture of the software itself. Using this component-architecture, the sixth and the seventh chapter shall present how the communication between the components and the charging of requests work in more detail, but also how security is handled. After that, the core concepts have been discussed and the actual implementation will be explained from a rather technical point of view. Next, a short user guide is presented, explaining how to build the prototype and how to use it. Finally, the last two chapters try to evaluate and conclude what has been reached as part of this work and how the results concern future work.

Chapter 2

Related Work

The charging of services based on the Internet Protocol (IP) is a task that a lot of service providers already accomplish. Charging architectures have established especially in the domain of the telecommunication market and the mobile communication market. Different solutions for either prepaid or postpaid options are in use, whereas prepaid mainly emerged from the mobile communication [7]. Known payment schemes of commercial service providers are for example wire transfer, ACH, SET or credit card payment. These solutions however cannot be used as part of the core federation structure and in consequence deter service provider to use it. Additionally, these schemes are often limited to serve the needs of mobile network operators only [8].

The SWITCH Federation [13] initiated a project called AMAAIS (Accounting and Monitoring of AAI Services) [1], which is currently subject to research at the University of Zurich. The project is being developed and primarily aims at monitoring the use of resources of organizations that are for example part of the SWITCH Federation. Whereas the mentioned project focuses on monitoring facilities, this project extends AMAAIS by charging functionality. In contrast to solutions mentioned above, the SFP-CM is universal and independent of the AAI system

Chapter 3

Preconditions

Some technical decisions have already been taken. Thus, the prototype implementation will have to follow these preconditions. The next section presents those preconditions in more detail and also explains why they have been chosen.

3.1 Preconditions

The list below summarizes the given technical decisions:

- The chosen programming language is Java [5].
- The base framework is represented by Spring [12].
- The software can assume a reliable channel and is not responsible for that.
- There must be a way to store and retrieve data (uVFA, oVFA, STM, ...).
- The system must be capable of handling multiple requests in parallel.

By having Java as underlying programming language, the resulting software is platform independent. Thus, the software will run on every machine that has a supported operating system installed, which is a big advantage. Additionally, Java has an extremely large community, which makes it easier to find reusable software components as it would be with other programming languages.

The Spring Platform is written in Java and has been chosen because of various features that significantly facilitate the development. On the one hand, it provides an Inversion-of-Control (IOC) container that allows for Dependency Injection (DI). The DI makes the system more flexible and improves the integration of different systems or functionality, because dependencies can easily be replaced. On the other hand, Spring provides a great foundation of ready to use components, thus supporting the development.

Another precondition is the assumption of a reliable channel. This is because reliability (and security) issues do not necessarily concern the SFP-CM. It is more important to focus on the core functionality of the SFP-CM and that is why such responsibilities are delegated to lower layers.

Additionally, there must be some kind of data store. The SFP-CA describes different data entities that the system must be capable of managing.

The system must not be a single user system. Hence, it must be able to handle multiple users, and also multiple requests per user. This leads to a multi-user and multi-threaded system.

Given these facts, the next chapter shall identify the software environment.

Chapter 4

Software Environment

The environment of a software influences various properties and has a substantial effect on the software architecture eventually. Thus, an important question is what the environment of the software looks like and how it can be embedded in it. The next section roughly identifies how the SFP-CM fits in to an IT environment.

4.1 Distributed Character of the SFP-CM

The SFP-CM is a system that spans multiple institutions or organizations. These organizations are most likely geographically apart, which means that the software has to be network enabled. The SFP-CM is designed to be a decentralized system, thus it does not have a centralized server that all the organizations might be connected to. Decentralized systems are less prone to breakdowns caused by malicious attacks or random events, since there is no single point of failure. In consequence, the software implementation will represent a distributed system. Each organization has an autonomous server infrastructure, however the end users of the software think they are dealing with a single system [14]. Hence, the users does not experience any difference while they use diverse services across different organizations.

Since the SFP-CM is meant to be a distributed system there are though various challenges the system has to face:

- Transparency
- Heterogeneity
- Failure Handling
- Openness
- Scalability
- Security

Throughout the remainder of this document, some of these challenges are explained in more detail. Figure 4.1 illustrates a scenario of the mentioned distributed system. There are three organizations (A,B and C). Each organization has its own server infrastructure running the SFP-CM implementation. There might be other instances running on it, such as a database management system. The SFP-CM instances can establish a bidirectional connection to every other instance. This connection should preferably be built on top of a secure channel, but this is not part of this specification. In other words, the software must be able to create connections to other instances and accept connections from other instances at the same time. Hence, it must be both a client and a server.

Such a system could be embedded within an Authentication and Authorization Infrastructure (AAI), such as Shibboleth [10]. However, this specification defines an AAI independent approach, meaning that the resulting software is autonomous and provides simulators for components like identity providers or service providers.

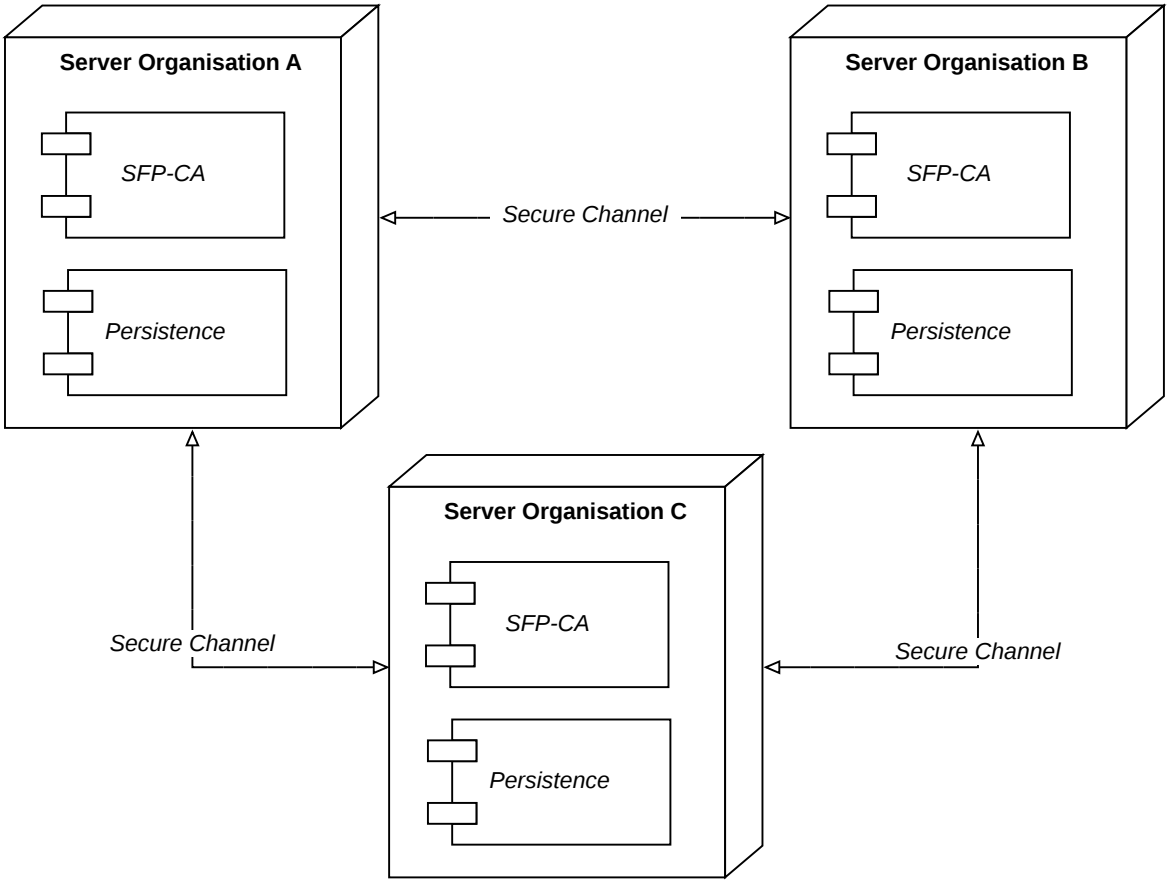

Figure 4.1: Possible Distributed System Layout

Chapter 5

Architecture Overview

The main components have already been described in [15]. However, the mentioned paper only provides rather abstract descriptions. Figure 5.1 shows the component in a more refined way. It represents the SFP-CM within a single organization. As it can be seen, the architecture consists of various different managers, which are an important part of the whole architecture. Four basic managers are provided by the SFP-CM:

- Charging Manager (CM)
- Charging Rate Manager (CRM)
- Service Provider Manager (SPM)
- Account Balance Manager (ABM)

All these managers can be seen as self-contained instances that run in separate Java Virtual Machines (JVM) or even on different physical machines. Together, these manager or instances span a network that provides the mechanism to charge a single funding point. Thus, in order to have a fully functional network, at least one instance of each manager must be running. In the following, the basics of all types of managers will be explained, starting with the Charging Manager.

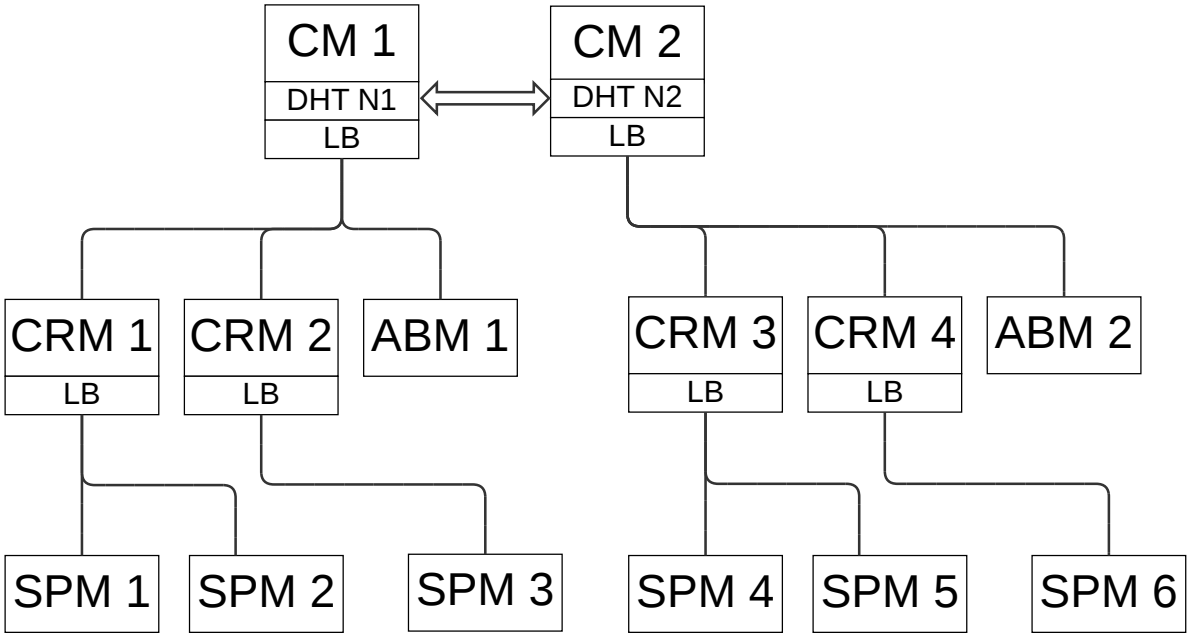

Figure 5.1: Possible Component Architecture of the SFP-CM. DHT NX stands for the node X of the DHT network. LB stands for Load Balancer.

5.1 Charging Manager

The Charging Manager is the first entry point for a user and its main task is to initiate and managing charging requests. Multiple Charging Managers may exist within an organization, spanning a P2P network. Therefore, a Chord DHT mechanism will be used. Instead of programming the DHT from scratch, it is intended to use an already existing implementation. A specific Charging Manager is determined to be the primary one. Typically, this will be the first one created. Those that will be created later on can just join the existing network. Every Charging Manager node is responsible for a specific set of users. When a user tries to initiate a charging request, the DHT is requested with the user id and the corresponding Charging Manager is returned eventually. In more detail this means, that the DHT stores a key value pair consisting of the user id and the IP/host address of the Charging Manager. The returned Charging Manager is then responsible for further processing of the charging request.

In order to make it possible to initiate charging requests, the Charging Manager must provide a list of services that are available to the specific user. Using this list, a user can either start a service (initiate the charging request) or, if a request is already running, interrupt a service.

5.2 Charging Manager & Remoting

The situation is further exacerbated if the user does not belong to the same organization as the requested service. One difference is that instead of having both the user data

(identity provider) and the service data within the same organization, the user data is now located at a foreign organization. In [15], the remote user sequence diagram proposes a simple request forwarding, meaning that the same request is passed to a remote Charging Manager. This works fine as long as its about the user data. However in order to retrieve the service data (STM), the diagram shows an interaction call from the remote Charging Manager to the local Charging Rate Manager. This is probably not optimal, since it requires the local Charging Rate Managers to be exposed across organization boundaries and additionally, both ends must be able to establish a connection, which makes it a bidirectional connection. Both factors can be problematic when it comes to security and failure handling issues. Hence, this paper proposes a slightly different approach. Instead of forwarding everything to the remote Charging Manager, the service related data (STM) is still processed on the local side. However, this requires additional changes: Since the remote system does not know anything about the STM that is currently used, most charging calculation will have to take place on the local Charging Manager. But this has the advantage that no bidirectional communication is needed. The calculated charges are just pushed to the remote system. The forwarding itself is again very simple. A local Charging Manager just does the mentioned calculations and forwards the request to the responsible Charging Manager (might be remote), which in turns acts as a proxy that calls the Account Balance Manager. In short, instead of requesting the local STM on the remote side, the remote Service Usage Constraints and Limits (SUCL) is requested on the local side. As a result, the request can be processed within a unidirectional flow, leading to a less invasive and transparent way. Bidirectional calls should be avoided if possible, because it makes it harder to control and is more error-prone eventually. Thus, the call from the remote Charging Manager to the local Charging Rate Manager should be avoided. Instead, the needed data could be pushed alongside the request.

5.3 Charging Rate Manager

Likewise the Charging Managers, there might exist multiple Charging Rate Managers. This redundancy helps to overcome potential performance problems as well as to improve failure handling. An interposed load balancer, which is integrated in a Charging Manager, is responsible for dispatching request objects coming from a Charging Manager to an idle or adequate Charging Rate Manager. A Charging Rate Manager is typically running as an independent instance, so that the Load Balancer has to address it over the network. In order to provide a fair balancing, the Load Balancer implements a scheduling algorithm, such as round robin. When a proper Charging Rate Manager has been selected, the request has to be passed to the corresponding Service Provider Manager. Each Charging Rate Manager manages a map of Service Provider Managers. From time to time, the Charging Rate Managers synchronize this map with each other.

5.4 Service Provider Manager

The Service Provider Manager is responsible for returning an appropriate Service Tariff Map of a given service. It does so by managing so called STM providers. For each service an STM provider exists. The provider has some prior knowledge about how to calculate the STM. The calculation can be influenced by passing appropriate service parameters alongside the charging requests. An exemplary parameter for a voice-over-IP service could be the phone number. Not only must the Service Provider Manager return the appropriate STM, but also a list of managed services. This list might be requested by a Charging Manager.

5.5 Account Balance Manager

The Charging Rate Manager and the Service Provider Manager are used to get the service related data. The Account Balance Manager on the other hand is responsible for managing the user related data. It has to provide the Charging Attributes, containing some user account data and also the SUCL. Additionally it must be capable of charging the User Virtual Funds Account(uVFA) and the Organization Virtual Funds Account(oVFA).

Chapter 6

Communication between Components

Since it has already been shown that the entire system consists of multiple nodes, the next step is to determine how they communicate with each other. Each manager typically runs in its own JVM, possibly even located on a separate server. Thus, the communication between the managers is taking place over the network. Since this communication steps generally follow a synchronous request-response pattern, there is no need for asynchronous communication styles like messages queues. The system should be able to invoke remote procedures in a simple and flexible way. This is why this paper proposes a REST communication style. All managers expose their functionality as RESTful services. Hence, a manager has to be deployed in an HTTP-server/Servlet-Container eventually. This allows the managers to communicate with each other by sending HTTP-requests and returning XML-data. The advantages of using REST for communication with HTTP servers are the following:

- The system is scalable [3]
- Easy integration with other components [3]
- Independence of components regarding deployment [3]
- The HTTP server handles the multi-threading / multi-requesting
- Flexible return format (XML, HTML, binary, ...)
- A request object (URI-format) is also a REST-call at the same time.
- individual components can be tested very easy, i.e. by using a browser.
- The REST architecture imposes a stateless interaction. Whereas the handling of session-based requests might be slightly more complicated, in the end it though facilitates the development process, since the management of distributed sessions (multiple nodes) can be quite difficult.
- Easy communication debugging, for example by using Wireshark [16]

6.1 Registration

The communication however requires that the manager know each other, at least alongside the communication flow. Whereas this could be achieved by a static configuration on startup of a manager, a registration mechanism is proposed in this section. Every instance that another instance must know of, has to register itself to the latter when it starts. For example, an instance of a Charging Rate Manager has to register to the corresponding Charging Manager. Registration in that case just means pushing the own host or IP-address to the other Manager via a specific communication protocol, which is REST in this paper. This approach has the advantage that the whole network can be extended dynamically without configuring the complete deployment architecture in advance. Listing 6.1 contains an exemplary registration object.

Listing 6.1: Registration object

```
<registration>
    <host>hostname/IP-adress</host>
    <port>8090</port>
</registration>
```

6.2 Exchange Formats

As mentioned before, the managers basically just exchange XML-data. XML might not be the most efficient data representation but it is easy to read for human beings and this makes it easier to debug. Furthermore, XML is well-supported when it comes to marshalling and unmarshalling of objects. This makes it easy to work with, because the programmer can just use objects and the communication layer transparently translates it into XML or back.

6.3 Charging Mechanism

The charging functionality is the most important of this system, but also quite a delicate one. The SFP-CM supports two different types of charging: Session-based and Event-based. Independent of the type of charging, all manager types are involved in the charging procedure. Whereas the Charging Rate Manager and the Service Provider Manager are responsible for providing a proper Service Tariff Map, the main actions are taking place between the Charging Manager and the Account Balance Manager. In the following charging types and the role of the managers are going to be explained in more detail.

6.4 Events vs. Sessions

Whereas event-based charging requests typically require a single charging operation, session-based requests need to be charged frequently, corresponding to the respective

STM. Which type the STM is intended for, is given by the STM structure itself, which is returned by the Service Provider Manager (REST):

Listing 6.2: Service Tariff Map

```
<stm>
    <startup>2</startup>
    <termination>0</termination>
    <event>0</event>
    <rate>
        <value>2</value>
        <sec>2</sec>
    </rate>
    <minBalance>22</minBalance>
</stm>
```

Since event-based services do not have startup or termination costs, the only information is basically contained within the "event" element. This indicates the price of the service. All other elements are needed for session-based services:

startup Costs to start the service.

termination Costs to stop the service.

rate The amount of monetary units the user will be charged at a fix rate, according to the sec value (seconds).

minbalance A value used to keep the charging of session service under control.

Thus, if the Service Provider Manager returns an STM, the Charging Manager primarily just needs to check whether there is a "rate" element. If yes, the corresponding other elements must also be there. If the "rate" element is not there, it can just go for the "event"-element The type of request (session or event) is indicated by the STM structure. If there is a "rate" element, the Charging Manager has to deal with a session based request. Otherwise, if there is an "event" element, it is an event-based request. If both elements occur in an STM it represents an invalid state. In this case the system should throw something like an "AmbiguousServiceTypeException".

Another difference is, that event-based services do not have startup costs or termination costs. Despite their different nature, events and sessions can be treated in a similar way: An event-based service might be considered the same as a session-bases service, whereas the former just has singular frequency (executed once) and additional costs of zero. Thus, all requests are treated as sessions.

Since session types might be running for an indefinite period of time, this session must be managed somewhere. The next section specifies how and where this session is managed.

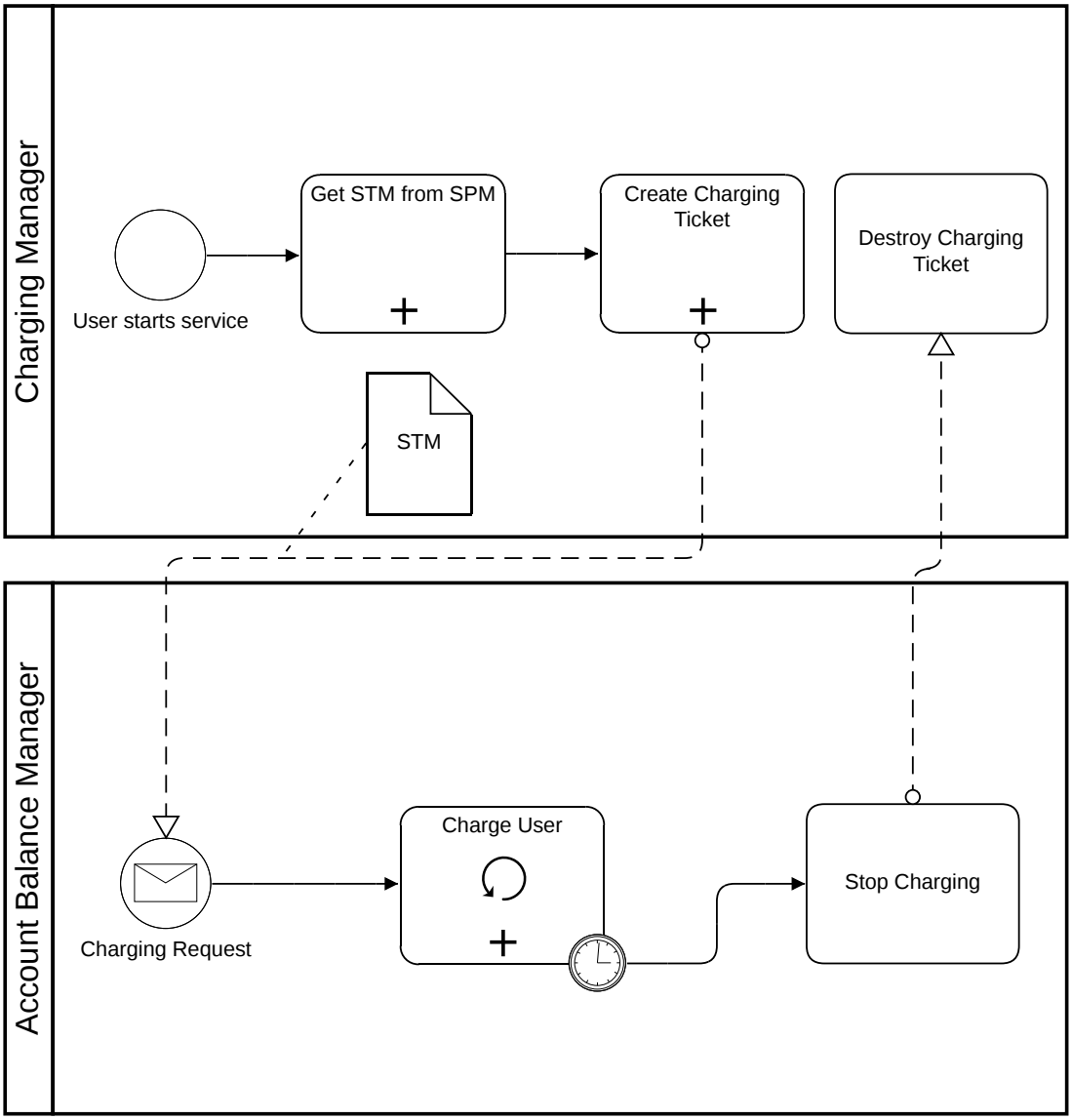

Figure 6.1: Unfavorable Charging Procedure

6.5 Session Management

Since maintaining a distributed session means maintaining the session state multiple times, it provides a broader attack surface and faces extended difficulties regarding consistency. In consequence this makes it more error-prone. This paper uses an approach that holds the session on one side without the knowing of the counterpart. The question whether it should be placed at the Charging Manager or at the Account Balance Manager might be raised. In diagram 6.1 is shown that the session is managed by the Account Balance Manager. The diagram shows a simple push of the STM to the Account Balance Manager, which in turn starts the charging session (executed at fixed rate). It becomes problematic when a session needs to be canceled. This can be the case if the current session is violating a constraint or if the user decided to do so. Due to the nature of the SFP-CA, the Account Balance Manager must not directly contact the Service Provider Manager to interrupt a

service. In consequence the Account Balance Manager needs to call the Charging Manager which takes action on it. As displayed in the figure, this would require a interruption request initiated by the Account Balance Manager. Alternatively (not shown in the figure), a manual cancellation of the charging session would require an interruption request initiated by the Charging Manager. The result of this is a bidirectional control flow, possibly across network boundaries, and might lead to serious problems. A very unfavorable scenario would be the failure of the network link between the Charging Manager and the Account Balance Manager. The manual interruption request had no chance to reach the Account Balance Manager, charging would go on and the user would loose money more and more. On the other hand, if a constraint has been violated and the Account Balance Manager intends to interrupt the service, the Account Balance Manager is forced to stop charging because of the constraint, but its request might not reach the Charging Manager, which leads to service usage without charging. Of course, network failures can be kept under control, but there are still timeouts that might negatively influence the "real-time" charging behaviour.

The better choice is to put the session management in the Charging Manager. As shown in Figure 6.2 the Account Balance Manager in this case is only responsible for charging a specific amount it received and returning an appropriate charging result. As a result, an interruption is no longer a self initiated request by the Account Balance Manager, but rather the response in a simple request/response interaction. Instead of letting the Account Balance Manager do the interruption, the control is moved to the Charging Manager. If the Charging Manager receives a negative feedback as part of the response, it knows that an interruption must be triggered. In case of network problems, if the Account Balance Manager is not reachable, or if no response is received, the Charging Manager can again initiate an interruption. The session should be managed at the Charging Manager side due to a more stable and easier handling of interruptions.

Compared to diagram 6.1, the second diagram 6.2 requires more charging requests. The reason is that whenever a further rate is due for payment, the Charging Manager needs to produce a charging request which is then sent over the network. This is opposed to the first diagram, where the Account Balance Manager recognizes due payments by itself (it manages the session). Having more request is not advantageous, since it highly stresses the network, which in turn provokes session interruption due to communication breakdowns. Additionally, the second approach burdens the Charging Manager with more work. The latter problem can easily be solved by scaling up the Charging Mangers. The former problem however can be bypassed by introducing a credit based system. The next section explains the credit system in more detail.

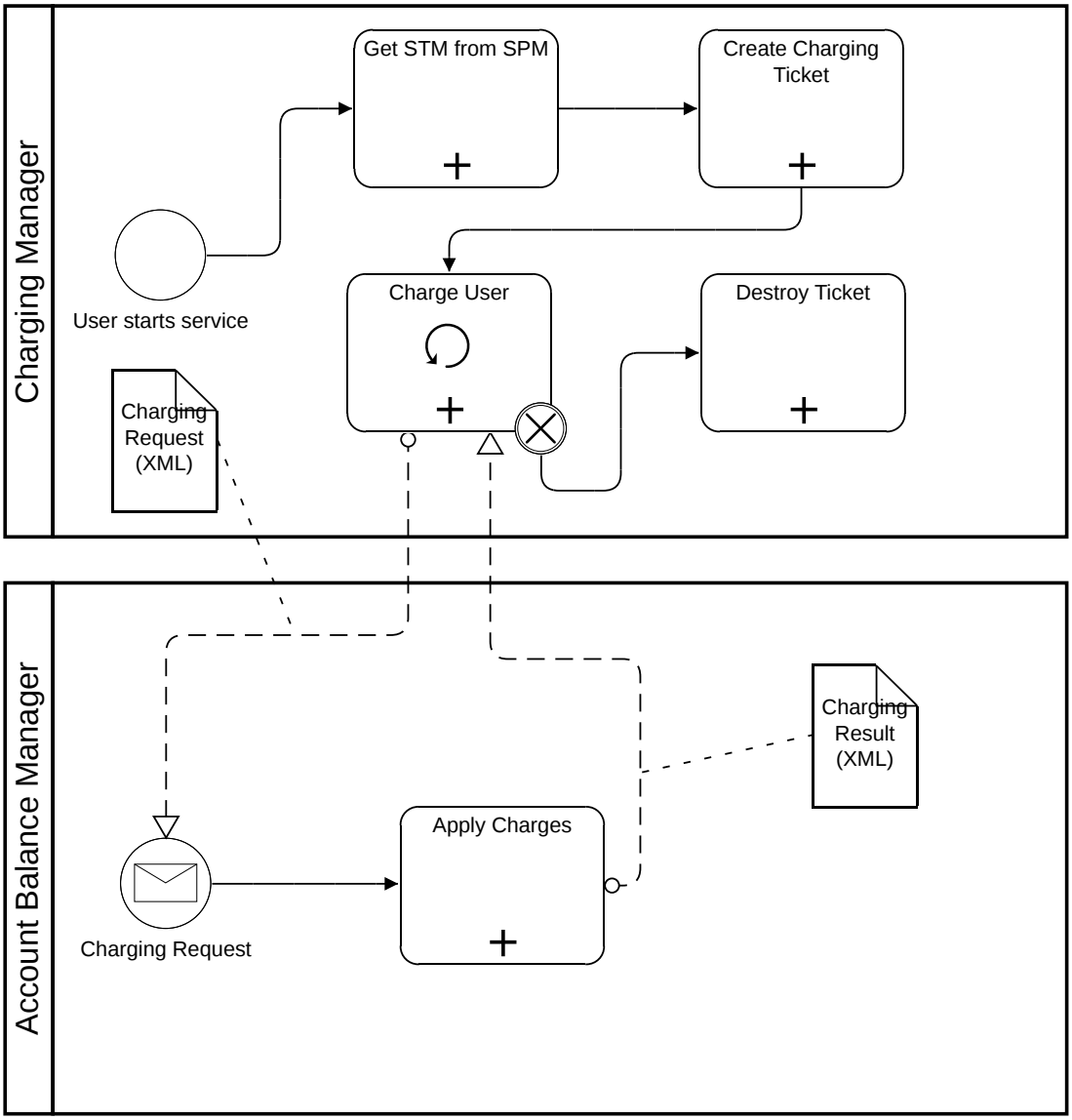

Figure 6.2: Better Charging Procedure

6.6 Credits

As mentioned before, credits are a way to reduce the network traffic and the failure rate at the same time. Instead of contacting the Account Balance Manager for every single rate that has to be paid, the Charging Manager reserves some credits in advance. These credits can then be consumed locally, as long it lasts. If it runs out of credits, the Account Balance Manager has to be contacted again.

Listing 6.3: Check if new credits are needed

```
private boolean needsNewCredits() {
    double limit = 0;
    limit += chargingValue;
    limit += startupValue;
    limit += terminationValue;
```

```
            return this.credits < limit;
        }
```

Listing 6.3 shows how the system detect if it runs out of credits. As it can be seen, additional costs like startup costs or termination costs must also be taken care of. Whether they are deducted at the beginning of a session or at the end does not make a difference. In this example, the additional costs are charged at the end, this is why the method is always reserving this costs in the limit value, so that it can be guaranteed that the user has enough credits to pay it. Another question is, how much credits should be bought during a request. This can vary individually, but a good approach is to use the minbalance value. Thus, if the credits value reaches zero, the credits are recharged with the minbalance value. However, if recharging is not possible or the user wants to abort the service, there might be some credits left. In this case, the remaining credits are refunded to the user, imposing another request. A credit based system reduces the network traffic and is less prone to session failures due to communication breakdowns.

Beside the costs of a service itself, the charging can also be influenced by the SUCL, which every user has. On the one hand, the SUCL contains information about discounts, on the other hand, it also specifies various constraints that must not be violated during a charging procedure. The next section shall discuss that more granular.

6.7 Discounts & Constrains

By using discounts and constraints the charging procedure can be regulated more fine grained. Both are properties are part of the SUCL, which might look like listing 6.4:

Listing 6.4: Service Usage Constraints List

```
<sucl>
    <service>all</service>
    <maxUnit>
    <session>700</session>
    <event>100</event>
    <period>
        <days>2</days>
        <limit>200</limit>
        <used>0</used>
        <start></start>
    </period>
    </maxUnit>
    <discount>
    <domain>
        <name>home</name>
    </domain>
    <tos>all</tos>
    <percent>50</percent>
```

```
    </discount>
    <negative>
    <allowed>true</allowed>
    <domain>
        <name>home</name>
        <amount>-200</amount>
    </domain>
    </negative>
</sucl>
```

The SUCL contains 3 different parts for constraints and 1 part for discounts:

service Specifies the service permission constraint, which is a list of services that the user is allowed to use. There is one predefined constant called "all", that just represents a wildcard. In order to specify single services, a comma separated list containing the service IDs must be provided.

maxUnit Specifies the maximal amount constraint. There are different situations that can cause a violation of this constraint. A user either exceeds the limit of the amount that is allowed per session / event, or per period. In order to check the session limit, the responsible instance must keep track of how much has been spent so far. Similar to the session limit, it must be kept track of the amount spent so far in order to check the period limit. But since this might span multiple sessions or/and events, the status must be stored in the "used" element. The system must also record the start of the respective period, by writing into the start field.

discount Specifies the discount for a domain. The system only supports one discount per user, which is typically set to its home institution. Likewise the service permission constraint, the "tos" element tells the system which services the discount has to be applied for. The "percent" value declares the actual discount in percent: 0 would mean no discount, 100 would mean free service.

negative Specifies the negativity constraint. This is where "postpaid" and "prepaid" users are distinguished. A "prepaid" user should typically not be allowed to have a negative account balance. A "postpaid" user on the other hand is allowed to go into negative values. In this case the negative limit can be specified. Again here, the system only supports one domain to specify it for, even though this can be easily extended.

The next question is where and when to apply the discounts and the constraints. One possibility could be to apply and check both at the Account Balance Manager. Technically this would work, but it brings on some delicate problems. As mentioned earlier, the Account Balance Manager is session-independent, and simply applies a charge. Thus, the Account Balance Manager would have to fetch the SUCL and calculate the discount for every request. The problem is that all changes of the SUCL would immediately be

applied. This has to be avoided by all means, because a user must be guaranteed, that the costs are always the same during the complete session. This is why the discounts should be calculated in advance at the Charging Manager upon creation of a charging ticket. The discounts should be calculated and applied during the beginning of a ticket creation at the Charging Manager. As a result, a session is not affected by SUCL changes and the user is guaranteed the same costs during a session. Opposed to the discounts, the constraints should better be aware of changes in a SUCL. Variations in constraints do not put a user nor an organization to expense, they just cause a service to stop sooner or later. However there are various situations where it might be necessary to immediately populate SUCL changes, on future sessions but also on running sessions. For example, an unexpected maintenance of the system might possibly require all users to get out of the system. By just denying access to all service an easy shutdown of the system could be achieved. Whereas this behaviour could be brought about by the Charging Manager as well as the Account Balance Manager, constraint checks need access to the account balance of a user and this is nothing the Charging Manager should care about. Constraints should be checked by the Account Balance Manager in order to have faster reaction times. Additionally it requires access to the current account balance, which should only be visible to the Account Balance Manager. Constrains violations are then simply returned as part of the result object.

6.8 Charging Objects

As mentioned before, the interaction of charging between the Charging Manager and the Account Balance Manager consists of a request object and a result object that is sent back. Listing 6.5 shows an example of such a request object.

Listing 6.5: Charging Request Object

```
<chargingRequest>
  <user>Cyrill Pedol</user>
  <amount>-17.0</amount>
  <requester>UZH</requester>
  <receiver>UZH</receiver>
  <totalCharges>5.0</totalCharges>
  <event>false</event>
</chargingRequest>
```

user The user that is going to be charged.

amount The amount to be charged. Negative amounts can be used to refund units to the user.

requester The organization the request has been initiated at.

receiver The organization that is responsible of charging the user.

totalCharges The accumulated value of all charges performed within a session.

event Indicates whether the requests originates from a event or session type service.

The "request" and "receiver" elements are used to detected whether it is a remote charging procedure and to apply constraints that are domain specific. The "totalCharges" element and the "event" element are needed to apply constraints that are service type specific.

The result object just contains a state message, indicating whether it could successfully charge the user or if a problem occurred. Listing 6.6 illustrates how a result object could look like.

Listing 6.6: Charging Result Object

```
<chargingResult>
  <state>OK</state>
</chargingResult>
```

6.9 Complete Charging Process

Even though the process in figure 6.3 is quite simplified, it still show the main steps of the charging procedure described before. The diagram does not explicitly mention updates to the oVFA. This is because it must be thought of a subprocess to the "Charge User" task as well as to the "Apply Charges" task. Opposed to changes in the uVFA, updating the oVFA must be performed twice, once for each organization involved in a remote charging procedure. Again here, it can be seen that the chosen process is advantageous. Having a Charging Manager that manages the session and an Account Balance Manager that does the further processing, makes it easy to integrate oVFA charging. Thus, whatever charges the Charging Manager is pushing to the Account Balance Manager can be added to the local oVFA at the same time. On the other hand, whatever the Account Balance Manager receives can be subtracted from the remote(from the Charging Manager's point of view) oVFA.

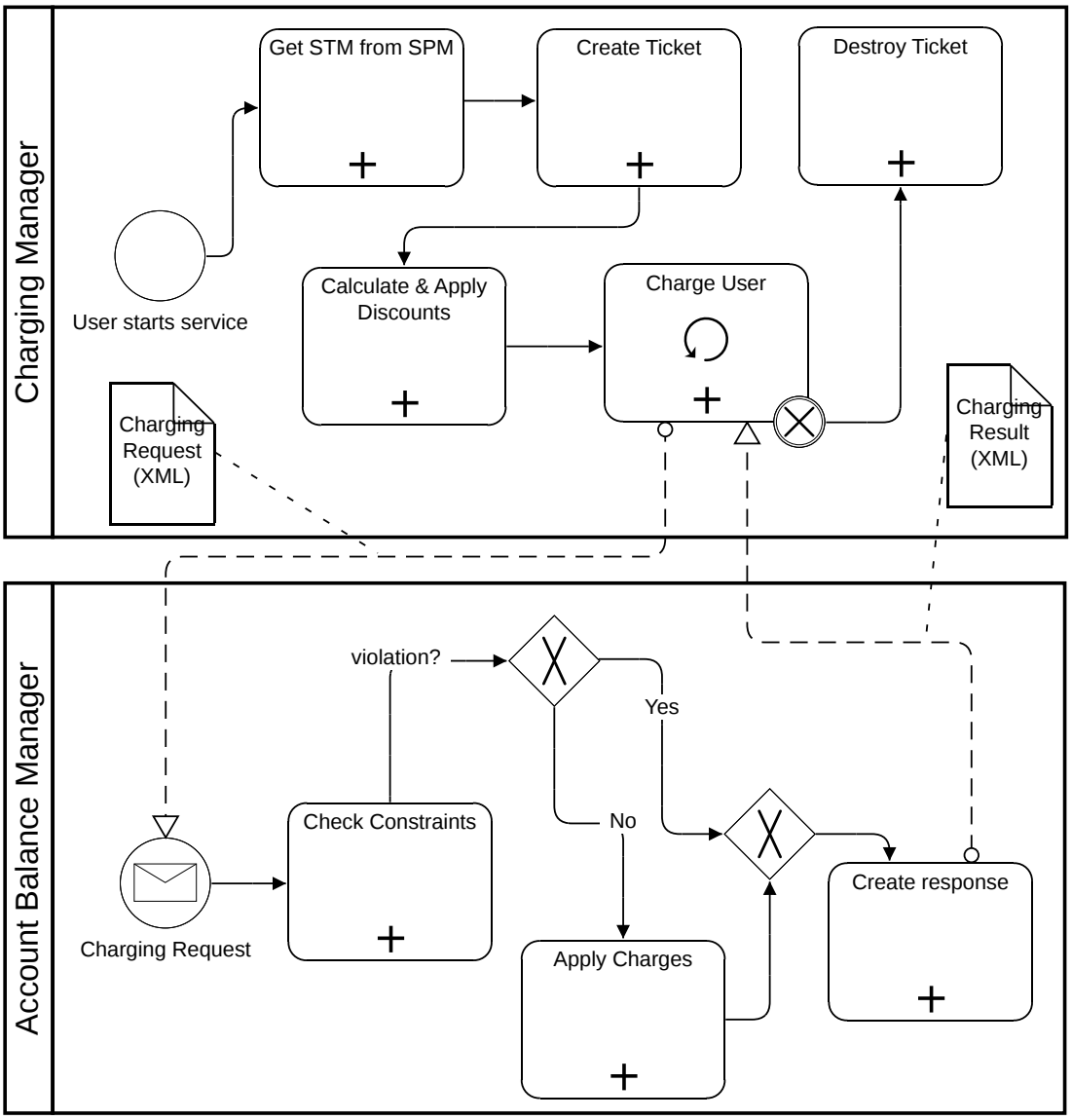

Figure 6.3: Complete Charging Procedure

Chapter 7

Security

When it comes to charging people or exchanging delicate data, security becomes an important part more and more. This work assumes a certain level of security provided by the infrastructure of the organizations and the use of well known and secure protocols and libraries. In order to be able to operate the SFP-CM securely, a full and well-conceived security concept needs to be elaborated in the future. The findings of this work can though be used to provide ideas for a broader security concept. The following section presents some considerations regarding a security concept.

7.1 Security Considerations

What can be claimed is that providing a secure data exchange is relatively easy, since the chosen communication style (REST/HTTP) already allows that by using HTTPS. Other security issues such as access control need further concepts. The prototype implemented as part of this work integrates the Spring [12] Security module, which is used to define Authentication Managers and provide a "login"-mechanism eventually. But this only takes into account "user-to-system" access control and ignores "system-to-system" access control. In other words, a system component like a Charging Manager is capable of recognizing a request of an authenticated user, but it becomes more difficult if the requester is another system component, that might possibly have been spontaneously added to the system. So the problem to find out, how this can even be distinguished, and if it could, what security rules should be applied. One possible solution could be to create security rules that either check whether a user is logged in or whether a (HTTP) call is coming from a predefined IP-range. Another solution might be to have some kind of an extended Single-Sign-On system that would require every component in the system to check against a Single-Sign-On server for each request. By whatever means this problem is going to be solved, it requires a more sophisticated concept has needs to be proven for practical purposes, which again is not part of this work.

The next chapter descends one level and tries to explain how the groundwork of the mentioned prototype looks like, how it can be built and how it can be used.

Chapter 8

Implementation

This chapter describes the prototype implementation. Opposed to the previous chapters, this should examine the SFP-CM from a rather technical point of view. The next section aims at describing the basic architecture of the project.

8.1 Project Architecture

The project is designed according to a Maven [2] multi-module layout. Apache Maven is a project management and comprehension tool, which can be used to support the building process as well as the dependency resolution [2]. Figure 8.1 shows all the modules that the project includes and it also illustrates how they are connected to each other. As it can be seen there is a module for each manager, but also some additional modules. In the following a short description for the modules is given:

sfpca-parent This is the parent module. All other modules are children of this one and thus inherit the module settings.

sfpca-core The core module implements functionality that can be identified as main functionality every other module must have too. This is why the manager modules must have a dependency on this one.

managers All the modules that represent a specific manager basically need to have a dependency on the sfpca-core module. Moreover, they implement the functionality that the respective manager must provide, such as RESTful services.

sfpca-aggregator This module is responsible for providing "Runner" classes and for the building process. It has a dependency on all the manager modules and is thus able to produce the final generated artifact.

The next section will explain how a module looks inside in more detail.

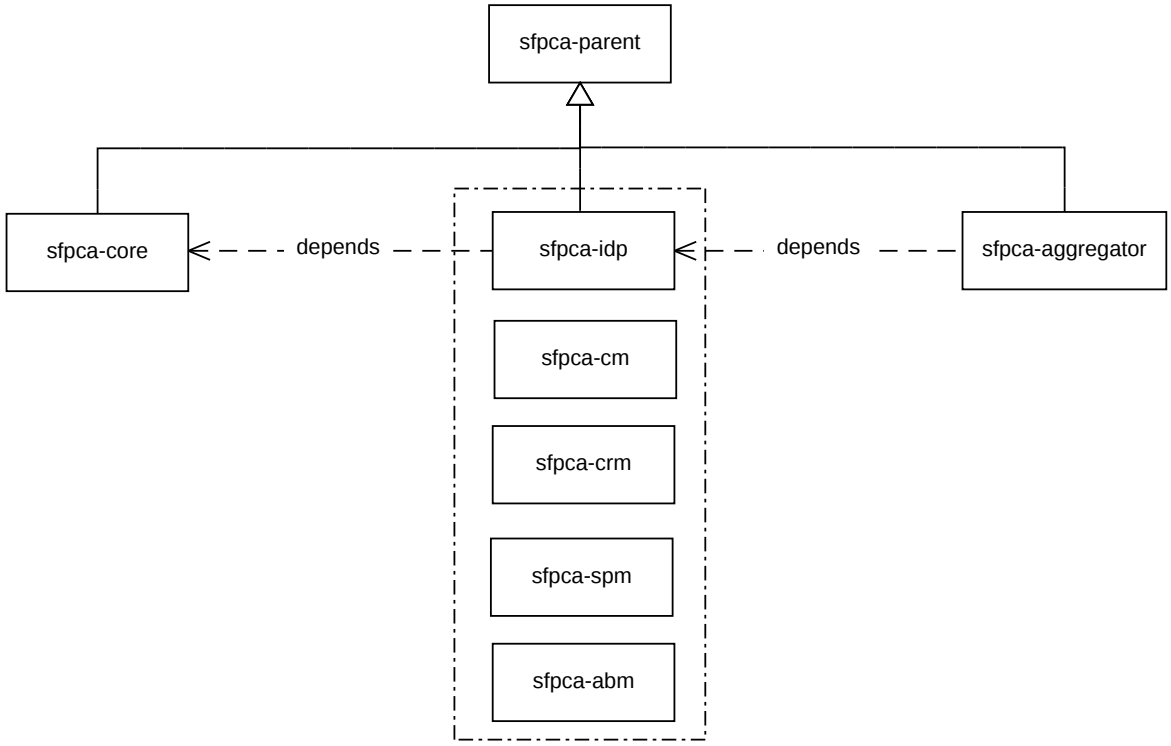

Figure 8.1: Maven Architecture

8.2 Modules

The modules themselves embody the default maven package structure. Hence, each module is built upon the following 2 source folders (leaving out tests).

- src/main/java
- src/main/resources

As their names imply, the first folder contains the Java source files, whereas the second folder contains other resources, mainly XML configuration files and resources for the front-end. The Java folder has a similar package layout for all the modules. By having a look into it, the following packages occur throughout the modules:

dto This package contains the Data Transfer Objects. This is basically just a collection of serializable object that can be exchanged between different instances.

dao A package that contains classes that allow for accessing the actual underlying persistence storage. These classes provide methods that encapsulate the domain specific data access logic.

entities This package contains the data entities. A manager uses these entities to work with, but these entities are not meant to be sent over the network.

Therefore, the properties of an entity must be copied to a corresponding data transfer object.

services The service package implements the actual module logic.

util This is where utility classes should be placed in.

web The web package contains RESTful services. Service methods are mapped onto an URL and either return XML or, in case of web browsers, HTML.

Likewise the java sources, the files in the resources folder have a common arrangement:

repository This is where "persistent" data can be placed. For the sake of convenience, the implementation stores data within XML files. This simplifies the project setup.

config Some modules need to provide some configuration files. Again, this configuration files are written in XML.

spring-module This folder contains the spring configuration files for a specific module. The value "module" should be replaced with the module name, such as "CM" or "SPM".

ui-module This is where the user interface related files are located. Typically, this contains the Freemarker [4] templates and the static resources such as images or CSS files that are all used to build the presentation layer.

8.3 Spring Contexts

As mentioned earlier, Spring [12] is an essential part of the implementation. It serves as basic platform for the prototype and thus, it is worth discussing the spring context configuration in more detail. Figure 8.2 shows how Spring is used, or rather, which Spring contexts exist and what they are responsible for. As it can be seen 3 Spring contexts exist, whereas they all are subject to an inheritance hierarchy. As its name implies, the "Root Context" is the topmost Spring context in the hierarchy and is configured to manage the core Spring beans that should be globally available (such as CommandLineService). The root context file is part of the core module and is automatically located and read when the application starts. There is only one root context file. The child context of it is called "Application Context". This context file is used to configure a Spring context for each module individually. The web related configuration must be placed within the "Web Context". The web context contains bean declarations and configuration items that are needed by Spring sub-modules such as Spring MVC or Spring Security. For the sake of clarity Spring Security is defined in a separate file, but it is still part of the web context.

The special thing about the application context file is that every module has it, but only one is bootstrapped. When a user is about to start the application, the user first has to declare which type of instance should be booted. For example, if someone would

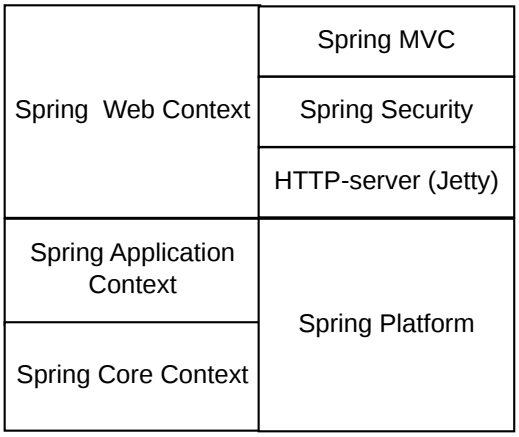

Figure 8.2: Spring Usage

like to start an instance of the Charging Manager by using the command line, a specific parameter would indicate the type, in that case "CM". On startup, the application notices that an instance of a Charging Manager is desired, so it only loads the application context defined in the Charging Manager module. Thus, the root context file is always loaded, the application context file and the web context file only if the specific module has been selected. Figure 8.3 illustrates the interactions that are made when the application is going to start.

1. A class called "ServerRunner" contains the main method. First it looks at the command line arguments that are passed by the user and parses them accordingly.
2. The ServerRunner starts the root context.
3. Now that a Spring Container (root) is running, some managed beans can already be accessed. The parsed options are passed to the "CommandLineService".
4. The ServerRunner starts the application context.
5. Now that another Spring Container (application) is running, the module specific managed beans can be accessed. Every module must have exactly one service implementing the so called IInstanceTypeService. This service provides methods for the module lifecycle. The start method of this service is called.
6. The ServerRunner starts the web context.
7. Now that the Web Container is running, the actual HTTP server (embedded Jetty) can be started by giving it the context.

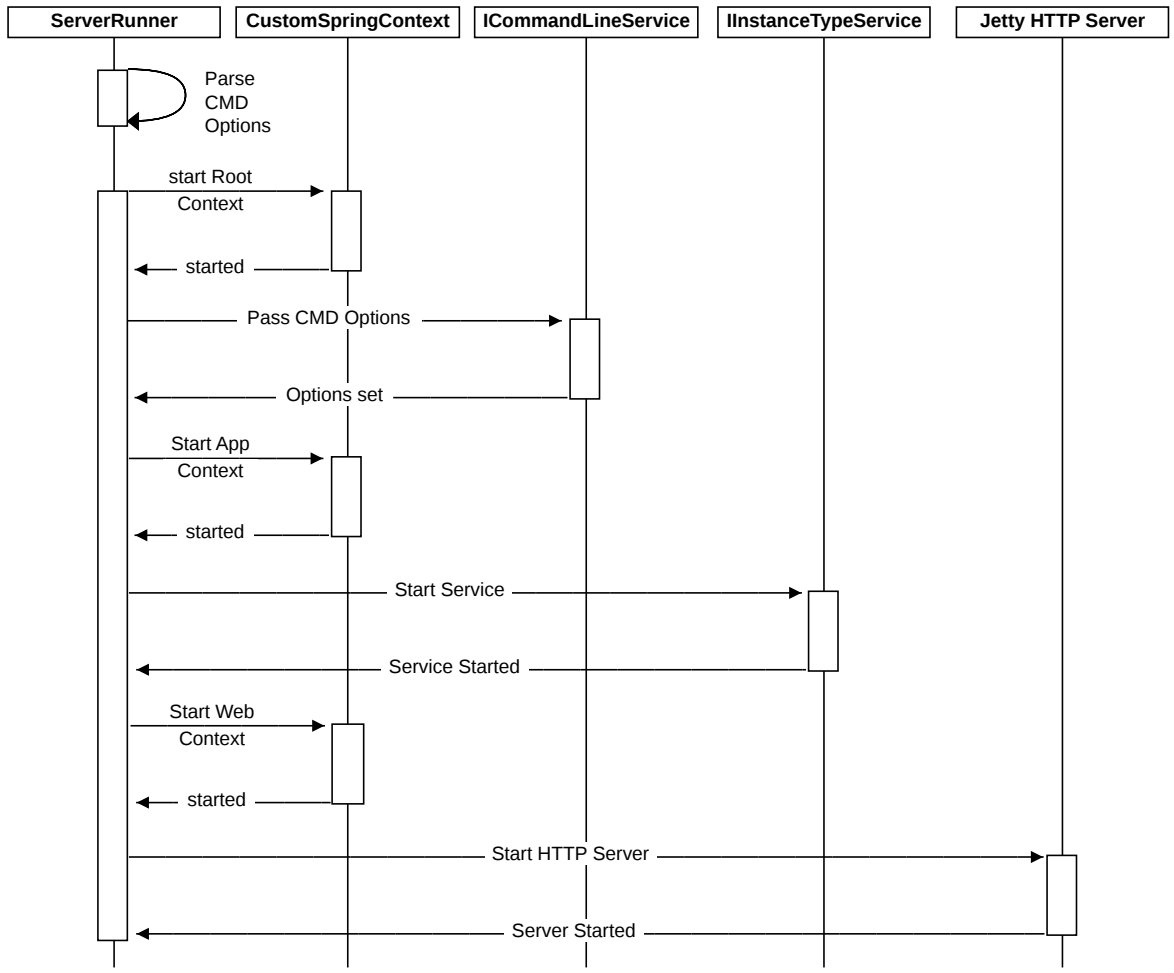

Figure 8.3: Startup Interactions

8.4 Persistence

Instead of using a relational database management system, this prototype limits itself to an in-memory database. Once the application has been started it performs an initial seed by unmarshalling XML files located in the repository folder. The application then just works with the mapped data entities created out of that. Writing operations are though possible but only affect the entity the data is written to. Thus, real persistence is not provided. However, if persistence would be needed, it would be easy to adapt the data access layer, since it has been designed technology independent (as far as possible). The seeding is done by the data access object, which takes the XML data file and populates it to the entities by using the Simple Framework [11].

XML data files are only needed by the identity provider module and the Service Provider Manager module. The former uses it to manage all the user data, such as name, balance and SUCL. The latter needs information about the services. The following listing 8.1 shows an example of the user repository file.

Listing 8.1: User Repository XML

```
<userRepository>
```

```
    <users>
        <user>
            <uid>000001</uid>
            <username>Cyrill Pedol</username>
            <password>tester</password>
            <organization>UZH</organization>
            <accountBalance>400</accountBalance>
            <chargingMethod>PREPAID</chargingMethod>
            <sucl>
            ...
            </sucl>
        </user>
    </users>
</userRepository>
```

As it can be seen, this XML file simply contains a list of users that have certain properties:

uid A unique ID of the user.

username The name of the user. Also the login name.

password The password needed to login.

organization The user's home organization.

accountBalance The current account balance of the user. This value is going to change during a charging procedure, but, as mentioned before, changes are only applied in-memory to the respective user entity.

chargingMethod Contains an enumeration value, that indicates whether the user has a "PREPAID" or a "POSTPAID" account. Technically, this field is redundant because it could be derived by the SUCL. However for the sake of convenience the information is placed again here.

SUCL This is the user's SUCL. The contents of this element have already been explained in a previous chapter.

The second file describes the services and an excerpt of the data structure is shown below:

Listing 8.2: Service Repository XML

```
<serviceRepository>
    <services>
        <service>
            <id>voip1</id>
            <name>Voice-over-IP</name>
            <description>Voip Service</description>
            <category>VOIP</category>
            <type>SESSION</type>
```

```
            <serviceParameters>
                <serviceParameter>
                    <id>pn</id>
                    <name>phone number</name>
                    <type>text</type>
                    <description>the number to call</description>
                </serviceParameter>
            </serviceParameters>
            <stmList>
                <!-- national calls -->
                <stm>
             ...
                </stm>
                <!-- international calls -->
                <stm>
             ...
                </stm>
            </stmList>
        </service>
    </services>
</serviceRepository>
```

Likewise the user repository, the service repository embodies a list of services. In this case, just one service is defined. The following properties are used to describe a service:

id A unique ID of the service.

name The name of the service.

description A description of the service.

category Each service is assigned to a specific category. This aims at handling groups of services when STM providers must work on services.

type This field indicates the type of the service. This value can either be "SESSION" or "EVENT".

servicePa-rameters As described earlier in this document, services can be parameterized. Listing the parameters here primarily helps when displaying the services, because the parameters can be generically visualized.

stmList This is list that contains the service tariff maps for this specific service. The Service Tariff Maps itself look like it has been explained in previous chapter. However, as it can be seen, multiple STM can be defined for a service. In case of a voice-over-IP service this list might contain one for a national call and another for an international call.

Whenever the application has been started it has the values defined in these 2 files.

8.5 Building the Artifacts

Since the application uses Maven [2] the process of building the final artifacts is relatively easy. Instead of generating a .war file for each deployable manager, only one .jar file is created. This makes it easier to have a quick start when using the application and it also facilitates the development because it must not be deployed to a servlet container for each single code change. So instead of running .war file within a servlet container, a servlet container is running within the .jar file (embedded jetty [6]). The mentioned .jar file, named sfpca.jar, can be found in the target folder of the "sfpca-aggregator" module. In order to build the jar the maven distribution is needed. Then, after opening the command line and switching to the sfpca-parent directory, the following command can be used to build the project.

- mvn package

There is one maven dependency that is not in the central maven repositories. This dependency is used to construct the DTH network and is called OpenChord [9]. So this dependency must possibly be installed manually or retrieved by a different maven repository. When installing it manually, the following parameters should be used:

Listing 8.3: Maven Parameters for OpenChord

```
<dependency>
      <groupId>net.sourceforge</groupId>
      <artifactId>open-chord</artifactId>
      <version>1.0.5</version>
</dependency>
```

If maven complains about the java versions, the following command can be used to solve the problem:

- mvn -Dmaven.compiler.source=1.6 -Dmaven.compiler.target=1.6 package

If everything worked fine, the sfpca.jar file should appear in the target folder. The next section explains how this jar file must be parameterized in order to start the instances appropriately.

8.6 Usage

The whole application should be delivered as a Java archive (i.e. sfpca.jar). Using the command line interface, it should be possible to create an instance of each type of managers. By specifying a port value, the application starts an embedded HTTP server for the given instance type.

-i Creates a new instance of a manager. Possible values are CM, CRM, SPM, ABM or IDP.

-p Specifies the port which should be used for the respective instance.

-h Specifies the target host. In case of a Charging Manager the value represents the load balancer the Charging Manager is connected to. In case of a Charging Rate Manager it represents the load balancer the Charging Rate Manager must register to. In case of an Service Provider Manager, it represents the corresponding Charging Rate Manager the Service Provider Manager must register to.

-b Specifies the host address of the DHT primary node. This option must be provided if a Charging Manager instance is started that should extend the DHT and should not start a new DHT.

-c This parameter specifies the path to a configuration file. This configuration file contains information that is shared between all instances of a given organization.

An example of the configuration file specified by the "-c" option is depicted in the following listing 8.4:

Listing 8.4: Application Settings

```
<sfpca>
    <domain>UZH</domain>
    <idp>[address]</idp>
    <partners>
        <partner>
            <domain>ETH</domain>
            <idp>[address]</idp>
            <primaryCM>[address]</primaryCM>
        </partner>
    </partners>
</sfpca>
```

The listing above describes that all instances which have this configuration belong to the domain called UZH. Additionally all instances have a parter domain, that can be used for remote charging calls. In the following a few examples of how to use the generated artifact are listed:

1. java -jar sfpca.jar -i IDP -p 8120 -c config.xml
2. java -jar sfpca.jar -i CM -p 8080 -c config.xml
3. java -jar sfpca.jar -i CRM -p 8090 -c config.xml -h localhost:8080
4. java -jar sfpca.jar -i SPM -p 8100 -c config.xml -h localhost:8090

5. java -jar sfpca.jar -i ABM -p 8110 -c config.xml -h localhost:8080

An examination of the examples show that the commands always start another instance type. The first one starts an identity provider, the second one a Charging Manager, and so forth. Using the "-p" option a port is specified for each instance, which means that the HTTP server is running on that port. Hence, the given port number just specifies the HTTP server, but not the port numbers of the DHT nodes. The latter number are generated automatically by just incrementing the specified port number. The "-c" option specifies the configuration file for settings that concern all instances. The last three commands additionally have the -h option. This option tells the respective instance where it has to register to. The instance in the last command for example, which is an Account Balance Manager, must register to the Charging Rate Manager eventually. Thus, it can be seen that the oder of the commands matters. Instances that must register to another one can only do that if the other instance actually exists. In case it does not exist retrials will be performed every second. A special thing about the Charging Manager is that it start a DHT network in the background.

8.7 GUI

Given the fact that the components are deployed within an HTTP-container, a suitable solution for displaying information in a graphical user interface would certainly be a browser. Additionally, due to the REST nature of the system, a browser can be a useful tool to check the RESTful services. Thus, if an instance has been started successfully, it can be accessed through a web interface. The most important one is the interface of the Charging Manager. It provides features to get a list of services, to start a service and even interrupt it. Some actions might require a user to login. The following example could represent an URL to access the web interface of a Charging Manager:

- `http(s)://localhost:8080`

Given the fact that the Charging Manager has been started on port 8080, this would display the web interface of the Charging Manager. In order to access different interfaces, the URL has to be changed accordingly, in this example by alternating the port.

Chapter 9

Evaluation

This chapter aims at giving an overview of what this work covers in respect to the features that the SFP-CA should provide as it has been designed in [15]. The following list summarized what has been discussed in the previous chapters and shows all the points that could have been successfully implemented and are part of the prototype eventually:

- Event-based services
- Session-based services
- Interruption Service Trigger
- Handling of simultaneous requests
- Inter-domain charging (remote) with oVFA
- Chord DHT
- Multiple CRMs, SPMs and CM
- SUCL awareness: Discounts & Limits & Method (Prepaid, Postpaid)
- STM awareness: Multiple STMs & Parameterization

Despite the fact that the list above covers most requirements, there are still some open issues. These open issues are either just partially solved or have not been approached as part of this work. The list below points out the open issues:

- Security is only partially integrated. Spring security provides a login mechanism, however security is not looked at in its entirety.
- No direct AAI integration is provided (such as Shibboleth).
- Synchronization between managers is not implemented (such as synchronization between CRMs)

The first two open issues could not have been implemented completely because it would have been too much for this work. The security issue has already been explained in previous chapters. The last point has been left out because it is rather seen as a general programming problem than a core feature of the SFP-CA and it is thus set aside for further implementations.

Chapter 10

Summary and Conclusions

The primary goal of this work was to elaborate a proof-of-concept based on the ideas of [15]. By defining a rough specification and implementing a software prototype, it is shown, that the concept is doing what it promises. Since the mentioned ideas in [15] were rather abstract and did not always provide all necessary information needed for a implementation, in this work certain propositions that aimed at extending the concept by that what was missing have been made, except for one point: The interactions for remote charging as they have been shown in the paper [15] slightly differ from how it has been proposed in this work. However, the theory is generally reconcilable with this practical work and the concepts could eventually be realized as specified.

The prototype implementation itself focuses on providing the main functionality and is not focused to features such as error handling, security or stability. Due to lack of time the scaling test was limited. However for every of the use cases the application accomplished all necessary tasks successfully. Given the fact that the application already provides mechanisms to handle that by itself, scaling up should though not be a big problem. It can be assumed that this should be a smooth operation, since the architecture follows a REST/HTTP design. Thus, an AAI must only be capable of processing HTTP requests, or, in other words, it must expose REST endpoints and return a proper XML format. Finally, the security part is an open part as well. The concept in [15] does not clarify or comment on how security should actually be realized, nor is it subject to this work. Apparently, exchanged data can easily be secured by using HTTPS for communication. However managing access right for individual instances is an issue that requires a more sophisticated solution and this solution needs to be elaborated first.

This work as a proof of concept software is not a ready to install application. Several modifications and adjustments should be done by the developer in order to adopt the system to the technology used by each organization. The SFP-CM must be thought of an experimental piece of software that helps providing information about how to implement the SFP-CA in one possible way.

Bibliography

[1] AMAAIS Project. Accounting and Monitoring of AAI Services, URL: http://www.csg.uzh.ch/research/amaais, Visited in November 2012.

[2] Apache Maven Project, URL: http://maven.apache.org/, Visited in November 2012.

[3] T. Bayer, "REST Web Services - Eine Einführung," 2002, URL: http://www.oio.de/public/opensource/t3n_nr8_rest_webservices.pdf, Visited in November 2012.

[4] Freemarker Project, URL: http://freemarker.sourceforge.net/, Visited in November 2012.

[5] Java Platform, URL: http://www.oracle.com/us/technologies/java/index.html, Visited in August 2012.

[6] Jetty Project, URL: http://jetty.codehaus.org/jetty/, Visited in August 2012.

[7] P. Kurtansky and B. Stiller, "State of the art prepaid charging for ip services," in *Wired/Wireless Internet Communications*, ser. Lecture Notes in Computer Science, T. Braun, G. Carle, S. Fahmy, and Y. Koucheryavy, Eds. Springer Berlin Heidelberg, 2006, vol. 3970, pp. 143–154, URL: http://dx.doi.org/10.1007/11750390_13, Visited in November 2012.

[8] D. Lutz and B. Stiller, "Combining identity federation with payment: The saml-based payment protocol," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, april 2010, pp. 495 –502.

[9] Open Chord Project, URL: http://open-chord.sourceforge.net/, Visited in August 2012.

[10] Shibboleth Project, URL: http://shibboleth.net/, Visited in November 2012.

[11] Simpleframework Project, URL: http://www.simpleframework.org/, Visited in November 2012.

[12] Spring Platform, URL: http://www.springsource.org/, Visited in August 2012.

[13] SWITCH Federation, URL: http://www.switch.ch/, Visited in November 2012.

[14] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[15] C. Tsiaras, M. Waldburger, G. Machado, A. Vancea, and B. Stiller, "The design of a single funding point charging architecture," in *Information and Communication Technologies*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7479, pp. 148–160, URL: http://dx.doi.org/10.1007/978-3-642-32808-4_14, Visited in November 2012.

[16] Wireshark Project, URL: http://www.wireshark.org/, Visited in November 2012.

Abbreviations

AAI	Authentication and Authorization Infrastructure
SFP-CA	Single Funding Point Charging Architecture
SFP-CM	Single Funding Point Charging Mechanism
CM	Charging Manager
CRM	Charging Rate Manager
SPM	Service Provider Manager
ABM	Account Balance Manager
IDP	Identity Provider Manager
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
SUCL	Service Usage Constraints and Limits
STM	Service Tariff Map
CSS	Cascading Style Sheets
DTO	Data Transfer Object
DAO	Data Access Object
DHT	Distributed Hash Table
uVFA	User Virtual Funds Account
oVFA	Organization Virtual Funds Account

Glossary

Charging Request A request that is created when a user starts a service.

Service Provider An institution that gives access to certain service facilities such as Internet telephony or printing.

Service Interruption The interruption of a service comprises the canceling of charging and the notification of a Service Provider Manager.

Distributed Hash Table A hash table that is capable of storing data items on various different nodes, whereas items represent a key-value pair.

P2P Network A distributed system architecture which may consists of multiple nodes that all represent a logically equivalent part and each part is assigned an own share of the complete data set[14].

List of Figures

4.1 Possible Distributed System Layout 8

5.1 Possible Component Architecture of the SFP-CM. DHT NX stands for the node X of the DHT network. LB stands for Load Balancer. 10

6.1 Unfavorable Charging Procedure 16

6.2 Better Charging Procedure 18

6.3 Complete Charging Procedure 23

8.1 Maven Architecture 28

8.2 Spring Usage 30

8.3 Startup Interactions 31

Listings

6.1 Registration object 14
6.2 Service Tariff Map 15
6.3 Check if new credits are needed 18
6.4 Service Usage Constraints List 19
6.5 Charging Request Object 21
6.6 Charging Result Object 22
8.1 User Repository XML 31
8.2 Service Repository XML 32
8.3 Maven Parameters for OpenChord 34
8.4 Application Settings 35

Appendix A

Installation Guidelines

This guideline explains how to build and run the prototype implementation that can be found on the attached CD. Since this prototype is going to set up a small distributed system, a fictional scenario must be assumed:

- The distributed system consists of 2 organizations (UZH and ETH)
- The system uses the default users defined.
- Both organization have one each manager running.

Additionally there is a set of prerequisites.

- Java
- Apache Maven
- Bash
- Browser

The guideline has been tested on a system that had the following tools installed:

- Oracle JDK 1.7.0
- Apache Maven 3.0.4
- Linux Bash (64 Bit Linux 3.2.0-30)
- Chrome 20

First 2 nodes for the distributed system must be chose. One node will represent "UZH", the other "ETH". These nodes can either be on separate physical machines or also within a virtual machine. It is important that both are part of the same network, or in other words, that they can connect each other. If the nodes have been chosen the LAN addresses of both nodes should be noted for setting up the environment. During the startup of the application, they try to get their own LAN address, thus, you should make sure that the file /etc/hosts contains the LAN address instead of the loop-back address:

- It has to be checked /etc/hosts contains the LAN address instead of the local address (for both nodes).

If that has been done, the application is ready to be build. The following steps will produce the artifact and have to be applied twice, once for each node:

1. Extract the file sources.zip to any directory (~/sources is assumed).
2. Open the command line and change the directory to ~/sources
3. In the command line enter:
   "mvn -s settings.xml -Dmaven.compiler.source=1.6 -Dmaven.compiler.target=1.6 package"

If everything worked fine, there should be a file called sfpca.jar in the folder

- ~/sources/sfpca-aggregator/target/

If that file exists, the build has been successful. The applications can now be configured:

1. Open the file ~/sources/sfpca-aggregator/launch/config.xml
2. Replace the IP addresses according to your 2 nodes, the port numbers can be the same.

After configuring the the 2 nodes, they can be started. Do the following steps on each node respectively:

1. In the command line go to:
   ~/sources/sfpca-aggregator/launch
2. Enter the command:
   ./start.sh config.xml

This should boot the respective instances, and they can be accessed eventually:

- Open your browser and enter the URL:
  `http://localhost:8080`

The instances can now be used.

Appendix B

Contents of the CD

The following list contains the files that can be found on the attached CD:

Zusfsg.pdf The abstract in german.

Abstract.pdf The abstract in english.

thesis.pdf The thesis as pdf.

thesis.ps The thesis as PostScript.

thesis.zip The LaTeXsource files of the thesis.

sources.zip The source files of the prototype implementation.

presentation.ppt The PowerPoint file of the presentation.