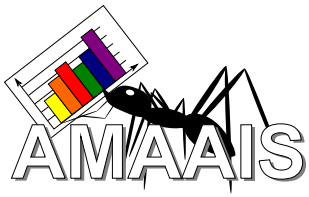

Accounting and Monitoring of AAI Services

Phase 2

Deliverable D2

AMAAIS Phase 2: Architecture Design and Implementation

The AMAAIS Partners

University of Zürich (UZH), Switzerland
SWITCH (SWITCH), Switzerland
ETH Zürich (ETHZ), Switzerland

For more information on this document or the AMAAIS project, please contact:

Martin Waldburger
University of Zürich
Department of Informatics (IFI)
Communication Systems Group (CSG)
Binzmühlestr. 14
CH-8050 Zürich
Switzerland

Phone: +41 44 635 4304
Fax: +41 44 635 6809
E-mail: waldburger@ifi.uzh.ch

Document Control

Title:	AMAAIS Phase 2: Architecture Design and Implementation
Type:	Public
Editor(s):	Guilherme Sperb Machado
E-mail:	machado@ifi.uzh.ch
Author(s):	Guilherme Sperb Machado, Patrik Schnellmann, Matteo Corti, Martin Waldburger, Andrei Vancea, Burkhard Stiller
Doc ID:	D2
Delivery Date:	31.01.2011

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
0.1	2010-01-22	P. Racz	Template created and document structure included.
0.2	2010-02-01	G. Machado	First version of client API and a draft of the accounting protocol included.
0.3	2010-03-31	P. Racz	Architecture and interface description updated. Accounting client API updated. System deployment overview included.
0.4	2010-11-03	G. Machado	Document outline. Provided details in the Architecture and interface description. Accounting client API updated.
0.5	2010-12-20	G. Machado	Many modifications in the document outline. Wrote content in different sections, and included figures.
0.6	2010-12-27	P. Schnellmann	Content in visualization component section.
0.7	2011-01-26	G. Machado	Content in the ASPEAR and Deployment sections.
0.8	2011-01-28	M. Corti	Review, comments, and modifications in the document as a whole.
0.9	2011-01-29	G. Machado	Content in the Collector Guidelines section, and some corrections.
1.0	2011-01-31	G. Machado	Final Version

Legal Notices

Contents

1 Executive Summary

The goal of the AMAAIS (Accounting and Monitoring of AAI Services) project—a collaboration between the Communications System Group (CSG) at UZH, SWITCH, and ETHZ—is to extend the current Authentication and Authorization Infrastructure (AAI) with accounting and monitoring support, enabling inter-domain accounting and the management of the AAI. The AMAAIS project is structured into project phases. Phase 1 [1] was centered at collecting use case scenarios, requirements, and the construction of a high-level architecture. This documents, which describes the results of Phase 2, is focused on the fine design of the architecture, the implementation of the accounting and monitoring architecture, and its deployment.

Phase 2 targets any kind of users (from large educational institutions or private organizations, to small-medium corporations) that already has a Shibboleth-based AAI infrastructure and wants to enable accounting and monitoring. Moreover, this document is mainly interesting for users that want to deploy the AMAAIS project, understanding the technical prerequisites, the limitations, and the underlying technical implementation (*e.g.*, interfaces, possible extensions).

Accordingly, this deliverable contains the following sections. Section 3 describing the AMAAIS architecture and design, Section 4 presenting the common concepts, Section 5 describing the project's deployment, Section 6 that explains how to extend AMAAIS for other service-dependent scenarios, and Section 7 that explicits implementation details.

2 Introduction

A Shibboleth-based Authentication and Authorization Infrastructure (AAI) enables users to access different web resources in a common manner by providing a single-sign-on interface for login. The AAI makes use of the Federated Identity Management concept that allows the use of a single user identity for different services beyond the user's home institution domain. Such characteristic plays an important integration and organizational aspect for institutions: it not only eases the users' access to resources, but also makes the management process more convenient. Pfitzmann et al. [2] provides a comprehensive overview of available federated identity approaches and protocols. Within the scope of the AMAAIS project (Accounting and Monitoring of AAI Services) [3], Shibboleth is used and the considered type of federation includes primarily institutions of higher education in Switzerland, among others. Shibboleth is based on the Security Assertion Markup Language (SAML) [4]. It is open source software and builds on OpenSAML [5], an open source implementation of SAML.

The goal of the AMAAIS project—a collaboration between the Communications System Group (CSG) at UZH, SWITCH, and ETHZ—is to extend the current Shibboleth-based AAI with accounting and monitoring support, enabling inter-domain accounting and the management of the AAI. During 2009, the AMAAIS project ran the Phase 1. The results of AMAAIS Phase 1 are published in the Deliverable 1 [1] which was primarily concerned with service-independent accounting and monitoring (*e.g.*, Service Provider and Identity Provider), as well as with service-dependent scenarios (*e.g.*, SMS and Printing services) and the composition of a high-level architecture to rely on. This document describes the results of Phase 2 completed in 2010.

Phase 2 was mainly focused on the refinement of the architecture and the implementation of the architectural components. Within the architectural scope, one key aspect is the importance of well-defined interfaces between components to enable previously defined requirements on Phase 1 (*e.g.*, extensibility and reliability). The main achievement related to interfaces is the successful development of ASPEAR (AMAAIS SAML-based Protocol to Exchange Accounting Records), which extends messages from SAML (using the consolidated and reliable library OpenSAML) to transfer accounting information between client and servers. The protocol was also designed to be extensible, meaning that other types of messages can be created and easily attached. Within the implementation of components scope, it is plausible to highlight the AMAAIS organization to enable the extension of other service-dependent components. For example, the current implementation enables users to write their own Collector (*e.g.*, a "log parser") in order to account resource usage of a given resource. This is possible due to (1) an API (Application Programming Interface) at the Accounting Client side, and (2) a parsing controller that ease the process of parsing log files—which are the most common method to meter resources. Another implementation key point is the range of configuration parameters that can be adjusted: number of retransmission attempts in case of server's reachability problem, buffer size of accounting records (at the client side), log rotation capabilities, local database (at the client side), rules/policies to forward accounting records to other servers, etc.

Therefore, the main goal of the AMAAIS project Phase 2 deliverable is to explicit the technical implementation details and to show how users can benefit from it, demonstrating

how to deploy or even extend project's components. The document is organized as follows. Section 3 presents the accounting and monitoring architecture with its components. Section 4 shows some concepts that are commonly used by the various components of the architecture. Section 5 explains how to deploy the AMAAIS project. Section 6 explores the AMAAIS extensibility, explaining how developers can build their own Collector component. Section 7 presents the implementation details, *e.g.*, UML diagrams, protocol's sequence messages between components and database tables. Finally, Section 8 gives a brief conclusion about the project as a whole.

3 Architecture Design

3.1 Architecture Overview

Figure 1 shows the AMAAIS architecture with its components and interfaces. The figure includes the components and interfaces required for the accounting of the AAI infrastructure, *i.e.*, Identity Provider (IdP), Service Provider (SP), as well as of the printing service and the SMS service. Further details about the printing and SMS scenarios can be found in [6]. Additional services can be easily integrated into the architecture by defining new service-specific Meter and Collector components. This will be described in more detail in Section 6.

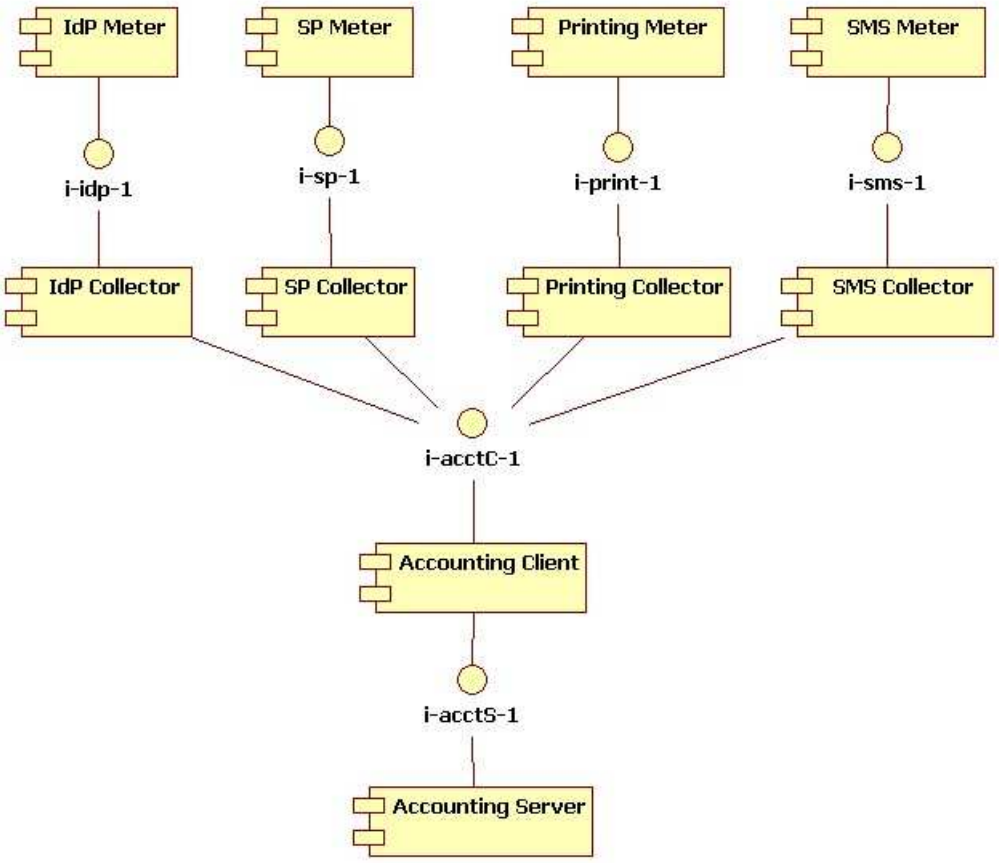

Figure 1: Architecture.

The Meter component is responsible for gathering information about events and service consumption of the AAI and of AAI-enabled services. The Meter component is specific to the AAI component and service for which it collects metering information. Therefore, there are specific Meter components for the IdP (IdP Meter), for the SP (SP Meter), and for the different services (Printing Meter and SMS Meter). The Meter component is usually an integrated part of the system/service that it meters, *e.g.*, in the first version of AMAAIS the IdP and SP Meters are the built-in loggers of Shibboleth (for organizational deployment de-

tails see Section 3.2). Meters generate service-specific metering information (*e.g.*, number of pages printed) which is received and processed by the Collector component specific to the AAI component and service.

Similar to the Meters, there are Collector components for the IdP (IdP Collector), for the SP (SP Collector), and for the different services (Printing Collector and SMS Collector). There can be different implementations of the Collectors depending on the Meters, *e.g.*, a Collector can be a log file parser if the Meter is a text logger. Collectors process the data received from the Meters and make them available for the Accounting Client in the form of accounting records. The Accounting Client sends these records to the Accounting Server using the ASPEAR accounting protocol (cf. Section 7.1), which specifies a common interface towards the Accounting Server. The Accounting Client provides a common interface to the different Collectors and it is the same for the IdP, SP, and any service. The Accounting Server is a central component of the architecture and is responsible for receiving accounting records from clients and store them in a local database. Additionally, an Accounting Server can forward records to another Accounting Server, while some of the attributes in the records might be filtered or changed depending on the accounting policy between the two servers/domains.

3.2 System Deployment—Organizational Recommendation

A possible deployment of the architecture components is shown in Figure 2. There are different organizational ways of deploying AMAAIS, because *e.g.*, some of the components might run on the same physical host or different database servers can be used for the accounting database. The project was designed to be aware of IT architectural variations and different necessities of organizing its physical resources.

As illustrated in Figure 2, the Accounting Server component is deployed on a separate physical machine and it uses a PostgreSQL database server for the accounting database (other database servers are also possible as detailed in Section 7.5). The Accounting Server runs as a Java Servlet web application in a Java-Webserver, like Apache Tomcat. In the example, the IdP, SP, Printing, and SMS components are running on separated physical machines. However, some of them might be deployed on the same physical machine or also in different virtual machines.

In AMAAIS v1.0 the IdP Meter is the standard logger of the Shibboleth IdP implementation, and the SP Meter is the standard logger of the Shibboleth SP implementation (shibd daemon). The Printing Meter is specific to the printing solution (VPP) deployed at ETHZ [7] and it is the logger of the print server (VPP logger). The SMS Meter is also specific to the SMS service at ETHZ [8], but in this case it is not a logger, but a database that stores all usage from the SMS web gateway in a structured manner. In AMAAIS v1.0 the different Collectors (IdP, SP, Printing, SMS) are integrated with the Accounting Client components and they run as a single process (daemon) on each machine where a Meter is deployed. More than one Collector might be integrated with the Accounting Client in a single daemon if required, *e.g.*, if there are several services running on a machine, the Collectors of each of these services can run in a single daemon.

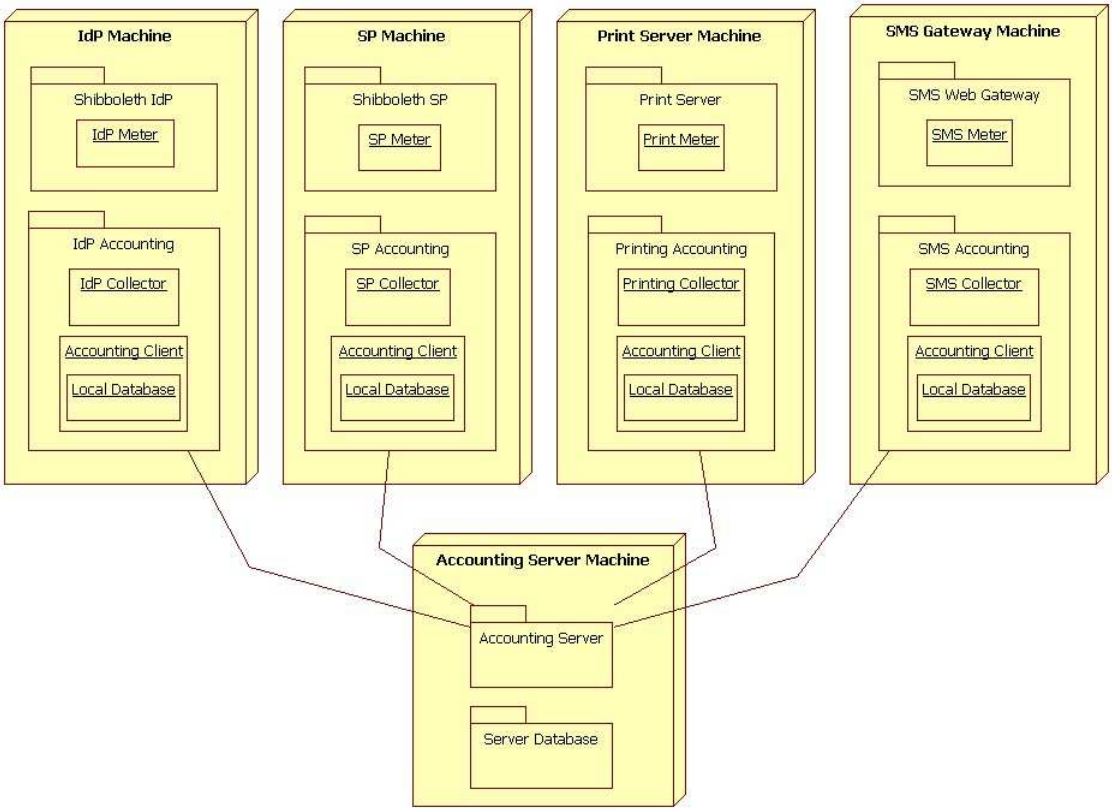

Figure 2: System deployment.

3.3 Interfaces

Table 1 lists and summarizes all interfaces of the AMAAIS architecture, while the following sections give an overview on each interface separately.

The interfaces between Meter and Collector components are specific to the metered service and can be implemented in different ways, *e.g.*, based on log files, an API, or socket communication, which can also differ between service. In the AMAAIS system v1.0 the interface between Meters and Collectors is based on log files for most of components, *i.e.*, the IdP and SP Collector uses the standard Shibboleth log files, and the Printing Collector uses the log file of the print server. Using log files enables the integration of accounting into an existing AAI environment without requiring any changes in the Shibboleth implementation and installation. In a later release this interface might be changed if required, without modifying the Accounting Client and Accounting Server part. However, as presented before, the SMS Collector establish a database connection to fetch the data related to the usage of the SMS web interface (Shibboleth). In this case, the meter itself is an application that populates a database each time the resource is used.

The interface between Collectors and the Accounting Client is an API in AMAAIS v1.0. This can be adapted to another type of interface, *e.g.*, socket-based communication, if future services requires that. Since the Collectors hide any service-specific interfaces from the Accounting Client, the architecture can integrate any future service without modifying

Table 1: Interfaces of AMAAIS v1.0.

Interface ID	Component providing the interface	Component using the interface	Description	Type/Protocol
i-idp-1	IdP Collector	IdP Meter	To send IdP-related metering data to the collector.	File based, using Shibboleth IdP log files
i-sp-1	SP Collector	SP Meter	To send SP-related metering data to the collector.	File based, using Shibboleth SP log files
i-print-1	Printing Collector	Printing Meter	To send metering data about print jobs to the collector.	File based, using log files of the print server
i-sms-1	SMS Collector	SMS Meter	To send SMS-related metering data to the collector.	Database based, using log information of the SMS web interface
i-acctc-1	Accounting Client	Collectors	To create and terminate accounting sessions, to create accounting records, and to send accounting records to the server.	Java API
i-accts-1	Accounting Server	Accounting Client and Server	To send/forward accounting records to a server. Both the client and the server can use this interface to send accounting records to a server.	Based on ASPEAR

the core components of the architecture which are the Accounting Client and Server. The interface between the Accounting Client and Server is based on the ASPEAR protocol (cf. Section 7.1) that determines a common interface used for the AAI components and for all services to communicate with the Accounting Server. The ASPEAR protocol is also used between Accounting Servers. For further details about the interactions over these interfaces during the accounting process see also Section 3.4.

3.3.1 Interface i-idp-1

The i-idp-1 interface is specified between the IdP Meter and the IdP Collector and it is used to send metering data related to IdP events, *e.g.*, authentication events and attribute release events, to the IdP Collector. In AMAAIS v1.0 this interface is based on the Shibboleth IdP log files idp-access.log and idp-audit.log. The standard log level is needed by AMAAIS v1.0. Details about the Shibboleth IdP logging features can be found in [6] and [9].

To use the standard Shibboleth log files has the advantage that the accounting functionality can be deployed without changing the Shibboleth implementation and installation. This is a desired feature especially for the integration of accounting in a trial phase. However, using the standard log files has some drawbacks as well, since some accounting-relevant information might not be available in current log files, some loggers might require more verbose log levels (debug), and the parsing of large log files determines additional overhead. Therefore, in a later release the Shibboleth logging feature might be adjusted to

specifically consider accounting-relevant log messages and to define a separate logger and log file for accounting purposes. This would allow the separation of the accounting functionality and the generic Shibboleth logging, the logging for accounting could be configured separately, and the accounting-related log messages could be better customized to the accounting needs (*i.e.*, to log information relevant for accounting which is not available in current log file or only available in higher log levels, resulting in larger log files and parsing overhead). This observation also applies for the i-sp-1 and i-ds-1 interfaces.

3.3.2 Interface i-sp-1

The i-sp-1 interface is specified between the SP Meter and the SP Collector and it is used to send metering data related to SP events, *e.g.*, authorization events, to the SP Collector. In AMAAIS v1.0 this interface is based on the Shibboleth SP log files native.log, shibd.log, and transaction.log. The standard log level is needed by AMAAIS v1.0. Details about the Shibboleth SP logging features can be found in [6].

3.3.3 Interface i-print-1

The i-print-1 interface is specified between the Printing Meter and the Printing Collector and it is used to send metering data related to print jobs, *e.g.*, number of pages printed and printer type, to the Printing Collector. In AMAAIS v1.0 this interface is specific to the printing solution (VPP) deployed at ETHZ [7] and it is based on the log files of the print server (VPP logger). In a later release additional interfaces can be added (*i.e.*, i-print-2) that support additional, different printing solutions.

3.3.4 Interface i-sms-1

The i-sms-1 interface is specified between the SMS Meter and the SMS Collector and it is used to send metering data related to the SMS service, *e.g.*, number of SMS sent, to the SMS Collector. In AMAAIS v1.0 this interface is specific to the SMS Gateway deployed at ETHZ [8] and it is based on a database that meters the SMS web interface.

3.3.5 Interface i-acctc-1

The i-acctc-1 interface is specified between the Collectors (IdP, SP, and services) and the Accounting Client and it is a common interface for all type of Collectors to create and terminate accounting sessions and to send accounting records to the Accounting Server. In AMAAIS v1.0 this interface is based on an Java API. The details of the API are described in Section 7.3.1. In a later release additional interfaces can be added (*i.e.*, i-acctc-2) to adapt the interface to another type, *e.g.*, to a socket-based communication if the Collector is implemented in another language than Java. This will allow to integrate any future services or extensions into the architecture.

The i-acctc-1 interface provides in general the functions listed below (for details see Section 7.3.1). These functions have to be supported by any possible future interface of the

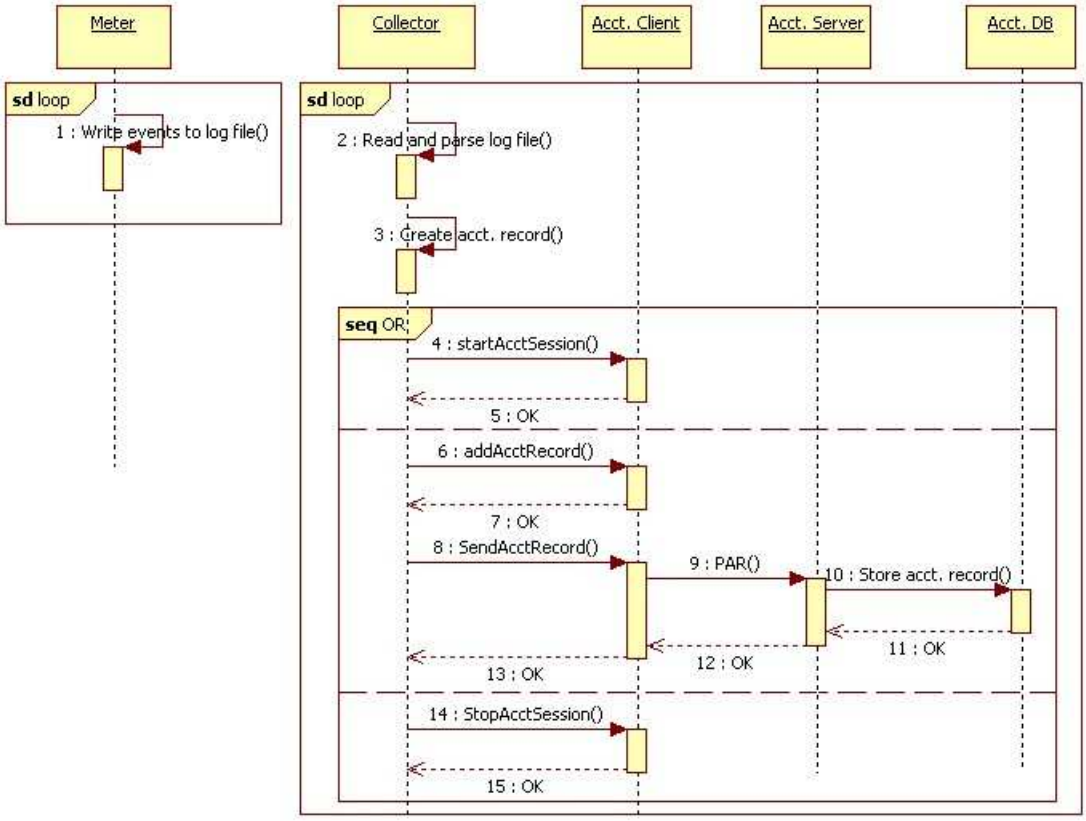

Figure 3: Accounting process

Accounting Client, irrespective of the communication type (*e.g.*, API, socket) of the new interface.

- Create new accounting sessions.
- Terminate existing accounting sessions.
- Assign accounting records to existing accounting sessions.

3.3.6 Interface i-accts-1

The i-accts-1 interface is specified between the Accounting Client and the Accounting Server as well as between Accounting Servers (within the same domain or in different domains). This interface is used to send accounting records to the Accounting Server and it is based on the ASPEAR protocol (see Section 7.1). Accounting records can be sent either by Accounting Clients or by Accounting Servers (servers can forward accounting records to other servers depending on their accounting policy). The interface is a common interface used to transfer accounting records of any type of service. Thus, the interface/protocol can accommodate the accounting for any future service without need to be changed.

3.4 Accounting Process

Figure 3 shows the general steps of the accounting process. Since the Meters are built-in loggers of different components, they run separately from the other components and continuously write log messages into their corresponding log files. These log files are read and parsed by the corresponding Collectors and in case there is an event relevant for accounting (*e.g.*, a print job is finished) the Collector creates a new accounting record that contains all relevant information of the event (*e.g.*, the user who printed and the number of pages printed). If there is no existing accounting session associated with the service, the Collector creates a new accounting session otherwise the Collector assigns the new accounting record to the existing session and sends the record via the Accounting Client to the Accounting Server. The record is sent using the ASPEAR protocol to the Server, where it is stored in the accounting database. If the service's metering is finished and all related accounting records have been sent to the Server, the Collector terminates the accounting session. It is important to note something implicit in the figure: in AMAAIS v1.0 the Accounting Client holds a local database to store accounting information if the server cannot be reached in the moment. More information about the local database configuration settings can be found in Section 5.3.3.

4 Common Accounting Information Model

Before explaining the deployment and implementation details related to the AMAAIS project, it is important to explicit some common concepts among architectural components. The AMAAIS Common Accounting Information Model is strongly based on concepts from OGF recommendations, as well as based on RFCs (Request for Comments) 2866 [10] and 3588 [11].

Figure 4 depicts the model. The top-level class is called “Accounting Session”, which represents a period of time devoted to perform a certain activity. A session can handle multiple activities, or none. For example, a session can be represented by a period of time in which a given user used a Web service. During his Web session, many activities can be performed, and therefore should be monitored and accounted properly. The session definition is meant to be a generic concept to group many accounted activities.

The aforementioned “activities” that an “Accounting Session” can handle are represented by the “Accounting Record” class. Following the previous example, a user begins to interact with a Web service in order to watch a video. During his session—besides the video streaming—another activity can occur as the user could comment the video with some text. Both activities are monitored by the system and translated to Accounting Records. To have a more concrete vision about records, in the scope of this example, the system can generate an Accounting Record related to the attribute “comment” (usually represented by an unique identifier), carrying the value “54”—which can be the number of characters written by the user. Moreover, the system can generate multiple Accounting Records related to the bandwidth usage to stream the video. All these Accounting Records should be linked to an Accounting Session in order to associate a service to it (“serviceID”) and give timing dimensions (“startTime” and “endTime”). The Accounting Session should also have an “accounting client” identifier, which is the Accounting Client daemon that generated the session. The Accounting Record must contain a “timestamp” which is the exact time that the record was generated. Since multiple records can be associated to the same session, a “recordNumber” becomes necessary.

Following the previous example, the attribute “comment” is represented by the “AcctAttribute” class. In fact, an Accounting Attribute should be represented with a specific type, depending on each attribute’s data value type. In the case of the attribute “comment”, the data type should be an integer (number of characters written by a given user). Since the class “AcctAttribute” is just an interface, the concrete class to be instantiated is the “AcctAttrString”. Note that an Accounting Record can also have an “AcctAttrGrouped” class that represents an Accounting Attribute, but containing many other attributes from the same nature. A typical example is a grouped attribute related to “memory” with sub-attributes as “RAM-Memory”, “Virtual-Memory”, etc.

It is important to note that sessions can also represent events. For example, if the Meter of a printing service just monitors events like “a job was printed using colors, on a A3 page and the duplex option”, the Accounting Session will have the same “startTime” and “endTime”. If it is available, the Meter can provide the time when the printer started printing and the moment when the job was finished. In the case of well-defined start and end times, the metered entry (in the Meter component) characterizes as being a session, having some activities during such period related to a given service. However, when this kind of

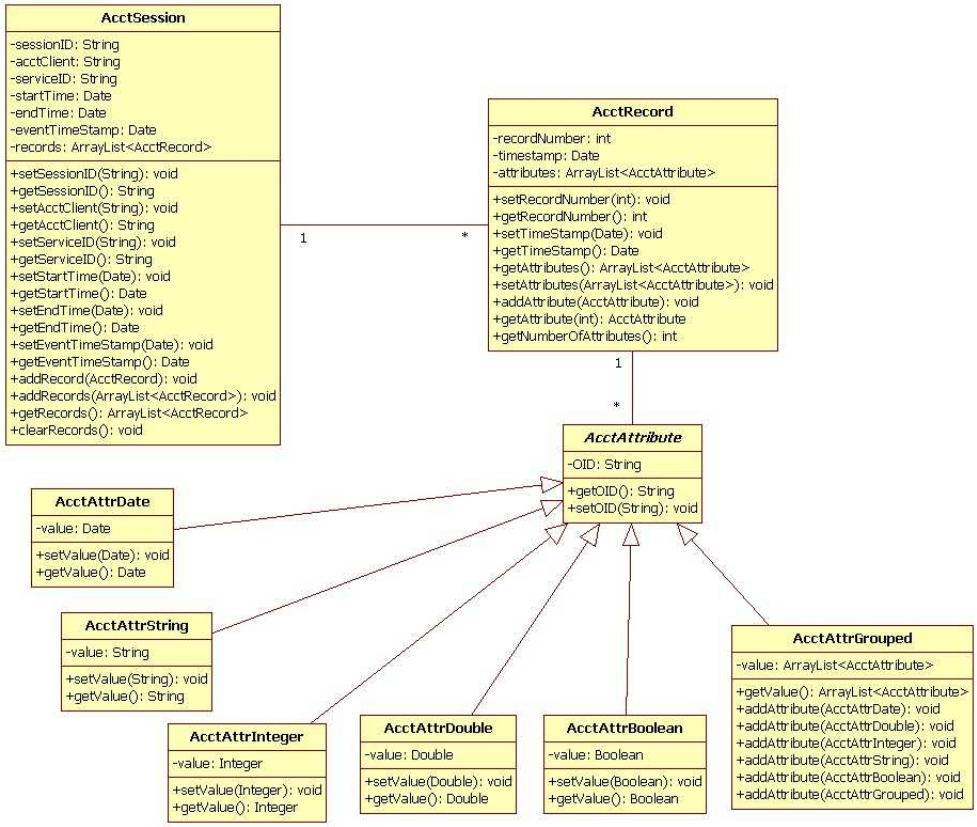

Figure 4: Common Accounting Information Model for AMAAIS components

information is not available, being represented just as a “file that was printed” at a given time (*i.e.*, a high granularity of information), the session concept should be adapted to represent such event.

Better described on Section 7, the AMAAIS implementation is strongly based on this model. Moreover, developers that aim to build service-dependent accounting solutions—by developing AMAAIS Collectors—should follow to the definitions presented in this section.

5 AMAAIS Deployment

This section covers how to deploy the implemented prototype. The steps on how to install, execute, and configure will be based on a Linux environment. Windows is supported but Linux was chosen since it is the reference implementation and tests with Windows will occur in a later stage only.

5.1 Prerequisites

At the moment of the deliverable conclusion, the prototype was only tested under Linux. It is therefore strongly recommended that the system runs under Linux (at least version 2.6.30).

The AMAAIS system was tested using different distributions (Red Hat Enterprise edition, Ubuntu, Gentoo), it is recommended that the machines that will host Collectors/Accounting Clients and Accounting Servers run in a minimal configuration. It means that, since AMAAIS daemons are constantly running and requiring CPU to parse reasonably amount of data, it is recommended that just the necessary packages are installed in the operational system.

Except those essential packages that comes with typical Linux distributions, the following Java libraries are strictly required to run the AMAAIS system as a whole:

```
http-client-4.0.1.jar
http-core-4.0.1.jar
jdom.jar
jnotify-0.93.jar
servlet-api.jar
xalan-2.7.1.jar
xercesImpl-2.7.1.jar
xml-apis-2.9.1.jar
resolver-2.9.1.jar
serializer-2.9.1.jar
bcprov-ext-jdk15-1.40.jar
commons-codec-1.3.jar
commons-collections-3.1.jar
commons-httpclient-3.1.jar
commons-lang-2.1.jar
jargs-1.0.jar
jcip-annotations-1.0.jar
jcl-over-slf4j-1.5.5.jar
joda-time-1.5.2.jar
log4j-over-slf4j-1.5.5.jar
not-yet-commons-ssl-0.3.9.jar
opensaml-2.2.3.jar
openws-1.2.2.jar
slf4j-api-1.5.6.jar
```

```
slf4j-jdk14-1.5.6.jar
slf4j-nop-1.5.6.jar
velocity-1.5.jar
xmlsec-1.4.2.jar
xmltooling-1.2.0.jar
postgresql-9.0-801.jdbc3.jar
db4o-7.12.156.14667-all-java5.jar
log4j-1.2.16.jar
hsqldb.jar
slf4j-api-1.5.8.jar
slf4j-log4j12-1.5.10.jar
jta-1.1.jar
javassist-3.9.0.GA.jar
hibernate3.jar
hibernate-tools.jar
freemarker.jar
dom4j-1.6.1.jar
commons-logging-1.1.1.jar
commons-collections-3.1.jar
c3p0-0.9.1.jar
antlr-2.7.6.jar
```

Besides these Java libraries, the Accounting Server machine must have Apache Tomcat up and running as the Accounting Server runs as a servlet (the Accounting Server should run on any Java Servlet Container bu was tested on Tomcat only). Since the Server is a Webservice the Tomcat running port should not be blocked by a firewall.

5.2 Obtaining AMAAIS

The AMAAIS software can be obtained at http://amaais.switch.ch, at the Download section and is released under the Apache2 license.

5.3 Installation and Configuration

Basically, the AMAAIS software package is one piece of software that can be separately installed depending on how the organizational system deployment is organized. In the following subsections we describe the unpacking, installation and configuration with a typical example (see Section 3.2).

5.3.1 Unpacking

The user should unpack the software using the following command:

```
tar -xvzf amaais-v1.2.tar.gz
```

Once executed, this command will result in three distinct directories:

```
amaais-client-SP
amaais-client-IdP
amaais-server
```

The “amaais-client-SP” directory contains all the files related to Collectors and the Accounting Client for the Service Provider, the “amaais-client-IdP” directory contains all the files for the Identity Provider and “amaais-server” contains the required files to deploy the Accounting Server.

5.3.2 Executing

The SP and IdP Accounting Client and Collector can be executed as it follows:

```
java -jar amaais-client-SP.jar
java -jar amaais-client-IdP.jar
```

Note that the Accounting Client prints a lot of information to the standard output (which is, most of cases, the terminal). To avoid this, it is recommended to redirect the output to a temporary file. Moreover, before executing the JAR files for the first time it is recommended to tune the configuration settings as explained in Section 5.3.3.

The Accounting Server is packaged in the “amaais-server.war” file in the “amaais-server” directory and has to be deployed using Tomcat. Note that all the libraries distributed under the server directory should be placed at the appropriate Tomcat folder. Consult the Apache Tomcat website[1] for further information.

5.3.3 Configuration Files

The AMAAIS software has three configuration files that is recommended to be properly tuned depending on users’ needs. Actually, due to the distribution of the service-independent part (IdP and SP), the AMAAIS system should be configured in 2 files: amaais-client-SP.conf and amaais-client-IdP.conf.

Since each Collector has its own Accounting Client, the configuration file for the client daemon will contain Collector information as well. Therefore, the amaais-client-IdP.conf looks like as the configuration on Listing 1.

Listing 1: amaais-client-IdP.conf

```
1 acctclient.identity = acctclient.ethz.ch
2 acctclient.server =
3                 http://acct-server.ethz.ch:8080/amaais/acctserver
4 acctclient.localdb = /path/to/local.db
5 acctclient.buffersize = 1
6 acctclient.retransmissions = 2
7
```

[1] http://tomcat.apache.org

```
collector.keypath = /path/to/keys
collector.idp.logfile.idp-access.name = idp-access.log
collector.idp.logfile.idp-audit.name = idp-audit.log
collector.idp.logfile.idp-process.name = idp-process.log
collector.idp.logfile.idp-access.path =
                collectors/aai/test/testlogfiles/idp/
collector.idp.logfile.idp-audit.path =
                collectors/aai/test/testlogfiles/idp/
collector.idp.logfile.idp-process.path =
                collectors/aai/test/testlogfiles/idp/
collector.idp.logfile.idp-access.rotation.pattern =
                idp-access-%yyyy-%mm-%dd.log
collector.idp.logfile.idp-audit.rotation.pattern =
                idp-audit-%yyyy-%mm-%dd.log
collector.idp.logfile.idp-process.rotation.pattern =
                idp-process-%yyyy-%mm-%dd.log
collector.idp.logfile.idp-access.rotation.mode = archival_static
collector.idp.logfile.idp-audit.rotation.mode = archival_static
collector.idp.logfile.idp-process.rotation.mode = archival_static
```

The first element of the configuration parameters represent AMAAIS that the settings will be set to. In the example above, there is configuration related to the Accounting Client ("acctclient") and Collector ("collector").

For the client ("acctclient") it is necessary to set:

- the identity of the Accounting Client ("acctclient.identity"), which is an identifier (URL) used to globally identify the given Accounting Client.
- the Accounting Server that the client will send the Accounting data to ("acctclient.server"). Note that users can specify more than one Accounting Server using commas. Therefore, each time that the Accounting Client has something to send, it will be sent to all the specified Servers. This mechanism can be used as a backup mechanism as well, since multiple "mirrored" Accounting Servers can be set. However, this approach can have a high impact on the used bandwidth.
- the local database file ("acctclient.localdb"), which stores some Accounting data will be stored in case of server's unavailability or any other unexpected error. The size of the database should not grow significantly (only in case of low server availability).
- the Accounting Client's buffer size ("acctclient.buffersize"), which is the maximum buffer size (in number of records) until the Accounting Client sends the Accounting data to the configured Servers. This can be optimized/tunned depending on how reliable users want its Accounting System. If the value is set to "1" (low value), for example, it means that every time that the Accounting Client has a new Accounting Session/Record to be sent, it will not wait for big period of time until the data gets persisted. On the other hand, if the value is set to more than "10", the Accounting Client may wait longer to send the Accounting data to the Server and gets the positive answer that it was persisted.

- the number of retransmissions retries (“acctclient.retransmissions”), which is the number of times that the Accounting Client will try until giving up and store the Accounting data in the local database. Note that it is not possible to configure/tune the interval set between retransmissions, as well as how long an Accounting Record will remain at the local database.

For the Collector (“colletor”) it is necessary to set:

- Key path (collector.keypath) is the directory where the “ParsingLogController” tool stores the files that controls how much of data was sent/forwarded to the Accounting Client.
- Rotation Mode (collector.*.logfilename.*.rotation.mode), can be set as: “archival_static”, meaning that log files rotate by date, in a static manner (example: test.log to test_20101227.log), or “archival_dynamic”, meaning that files rotate given an index, and keep rotating by a date/file size (example: test.log to test.log.1 and test.log.1 to test.log.2, etc).
- Logfilename (collector.*.logfilename): specify the absolute path of the log file. Note that a collector can have more than one log filename, if it has to parse more than one source. In the case above, the IdP collector has “idp-access”, “idp-audit”, and “idp-process” to parse.
- Rotation Pattern (collector.*.logfilename.*.rotation.pattern): specify what is the pattern followed by the log rotation. Standardized values to be used: %yyyy = year, in four decimal digits, %mm = month, in two decimal digits, %dd = day, in two decimal digits, %i = index, usually used in the “archival_dynamic” rotation mode, which %i represents one decimal digit.

At the Accounting Server side it is possible to configure the forwarding policies: a set of rules that the Accounting Server will evaluate to release Accounting Attributes to other Accounting Servers. An example of such configuration file is showed on Listing 2. The configuration file is XML-based and is called “attribute-filter-policy.xml”.

Listing 2: attribute-filter-policy.xml

```
1  <AttributeFilterPolicy id="forwardToAcctServerExample">
2
3      <!-- Policy requirement rule that indicates this policy
4      is only used to forward attributes to the following
5      Accounting Server:
6      http://acctclient.ethz.ch/amaais/acctserver -->
7
8      <PolicyRequirementRule
9        xsi:type="basic:AttributeForwardingRule"
10       value="http://acctclient.ethz.ch/amaais/acctserver"/>
11
12     <!-- Attribute rule for the email attribute -->
13     <AttributeRule attributeID="email">
14         <!-- Permit value rule that releases any value. -->
```

```
15          <PermitValueRule xsi:type="basic:ANY" />
16      </AttributeRule>
17
18      <AttributeRule attributeID="userName">
19          <!-- Permit value rule that releases any value. -->
20          <PermitValueRule xsi:type="basic:ANY" />
21      </AttributeRule>
22
23  </AttributeFilterPolicy>
```

The XML element “AttributeFilterPolicy” specifies one policy. The element must contain the “PolicyRequirementRule” element which type should always be “basic:AttributeForwardingRule”. It means that it is an Attribute Forwarding Rule, and therefore the Server can evaluate if such configuration file has the purpose that was originally designed to. The “AttributeRule” element can appear multiple times, depending on how many attributes it is desired to release/forward. Therefore, the attribute “attributeID” that defines the name of such attribute to be released/forwared. Due to the sake of simplicity in this document, it was not used the attribute’s URN. However, the “attributeID” should be globally unique.

At the Accounting Server side, there is one configuration file which should be adjusted, and that is out of the AMAAIS package distribution: the Hibernate configuration file. Essential parameters like the database host, database port, username and password should be set. Please, refer to the Hibernate documentation website[2] to properly understand each available option. Another fundamental piece of software that should be configured and tested beforehand is the PostgreSQL, which is the database software used in the scope of AMAAIS. Since just the JDBC driver for PostgreSQL is distributed with AMAAIS, the tuning of the database should be made by the user, consulting the documentation website[3].

[2] Hibernate Documentation available at: http://www.hibernate.org/docs.
[3] PostgreSQL Documentation available at: http://www.postgresql.org/docs.

6 Collector Guidelines

This section provides a practical guide on how to develop service-dependent Collectors. Throughout the section it will be used the example of a printing service to better illustrate how different services are attached to the AMAAIS system.

It is important to note that even if this section is directed and described using an example, developers should not be retained to printing. Other use cases like SMS accounting, network bandwidth, etc., can be also considered.

Another essential point to mention is the use of the Common Accounting Information Model presented on Section 4. Developers should use the accounting concepts described in the AMAAIS information model (*e.g.*, Accounting Session, Accounting Records) in order to successfully benefit from extending the Collector component to service-dependent scenarios.

6.1 Extending the Collector class

The Collector class is a Java interface that just provide some methods to be implemented. Based on the printing example, the following code is a skeleton on how the Collector class may be extended.

Listing 3: Example on how to extend the Collector class

```
1  public class PrintingCollector implements Collector {
2
3          private static Logger logger =
4                  Logger.getLogger(PrintingCollector.class.getName());
5          private static final long PRINTING_COLLECTOR_PERIOD = 600000;
6
7          private boolean isInitialized;
8          private VPPLogParser parser = null;
9          private Timer        timer = null;
10
11         public int init(AccountingClient ac) {
12                 logger.debug("Initializing the printing collector");
13                 isInitialized = false;
14                 parser = new VPPLogParser(ac);
15                 timer  = new Timer(true);
16                 isInitialized = true;
17                 return RET_OK;
18         }
19
20         public int start() {
21                 logger.debug("Starting the printing collector");
22                 if(isInitialized == false) {
23                         logger.error("Not initialized");
24                         return RET_ERROR_START;
25                 }
```

```
26                  timer.scheduleAtFixedRate(parser, 0,
27                          PRINTING_COLLECTOR_PERIOD);
28                  return RET_OK;
29          }
30
31          public void stop() {
32                  logger.debug("Stopping the printing collector");
33                  if(timer != null) timer.cancel();
34          }
35  }
```

Before describing the Listing 3, it is important to mention that:

- The Printing Collector example, which will be used throughout the current and following sections, is composed by a parser;
- The parser looks to a specific file, grabs the metered information and composes Accounting Sessions and Accounting Records accordingly to its needs;
- The parser is a totally independent piece of software that can be implemented in several ways. This section does not aim to exhaust nor explain specific details of a parser implementation. The focus is to explicit how to extend a Collector, and once the metered data is available (*e.g.*, number of pages, colored, etc.), explain how to organize such data – according to the Common Accounting Information Model of AMAAIS –, and how to interface with the Accounting Client API.

Observing the Listing 3 at line 3, there is a logger declaration for information and debug purposes. This is not an essential part of extending a Collector, but it is highly recommended due to best practices for accounting components. Lines 5-9 are specific declarations for the Printing Collector implementation. The variable “isInitialized” controls whether the daemon is up and running or not. The variable “VPPLogParser” is an object which represents the parser that grabs the printing information. Such parser will be better explained on Sections 6.2 and 6.3. The variable “Time” is a variable that schedules the interval that the parser will be executed. The static variable “PRINTING_COLLECTOR_PERIOD” is actually the interval rate, in milliseconds, that the parser is executed. This value can vary and should be tuned. However, in the specific case of parsing print server logs, we can assume that some minimal delay can be accepted in the accounting process.

The Listing 3 at lines 11-35 contains the implemented methods from the Collector interface class. It is important to note that within the “init()” method, line 14, the parser is instantiated. Other specific variables (depending on the service needs) can be instantiated here. At the method “start()”, line 26, the parser is scheduled and started. Note that scheduling a parser is not strictly necessary. The whole implementation of the Collector can be made using one single class, in the same thread. However, it is recommended due to processing load and failures to start a separated thread to handle different jobs (*e.g.*, as parsing files or consulting a database). The method “stop()” basically stops the “timer” execution, which consequently stops the printing parser.

6.2 Using the ParsingLogController Util

Since Collectors are components that in many cases rely on processes like parsing to account data, a tool called “ParsingLogController” was created. This tool has the goal to turn the parsing process more transparent and reliable to its users. Basically, it has the following interfaces and functionalities:

- Provides a method “read()” that reads one specified file and returns the current line (string);
- Provides a method “signalSent()” which signals the ParsingLogController that data until a certain point was parsed and forwarded to the Accounting Client component;
- Once instantiated, the ParsingLogController creates files in order to store some information like (a) bytes which were just parsed until the moment and (b) bytes that were parsed and sent through the Accounting Client API;
- During its initialization, the ParsingLogController looks to these files and checks its consistency by checking if everything that was parsed was also sent to the Accounting Client. If the ParsingLogController identifies that something was not sent, a problem may be occurred and the parsing process starts from where it stopped before (*i.e.*, right after from where the “signalSent()” was called).

The Listing 4 provides an example on how the ParsingLogController should be used.

Listing 4: Example on how to use the ParsingLogController

```
1  public void run() {
2          ParsingLogController logController =
3                  new ParsingLogController("print", "vpp");
4          String currentLine;
5          while ((currentLine = logController.read()) != null) {
6                  ...
7          }
8          logController.signalSent();
9          ...
10 }
```

Observing the Listing 4, it is important to highlight the “ParsingLogController” class instantiation. The constructor method requires to set two strings which are (1) the service identifier and (2) the log identifier. The service identifier should be the same used in the amaais-client.conf configuration file. In this case, just as an example, the key used to configure the name of the “vpp” log file which belongs to the “print” service should be: “collector.*print*.logfile.*vpp*.name = vpp_accounting-2010-05.log”.

The Listing 4 at line 5 presents a “while()” construction that reads data from the instantiated “ParsingLogController” class. Since the method “read()” returns some data or null (when there is no data), some condition checking should be made. After the data returned is treated and organized as accounting classes following the Common Accounting Information Model (this process is better explained on Section 6.3), the “signalSent()” should

be called. Note that the “signalSent()” can be called in any point or any time during the “ParsingLogController” object’s lifetime. It will depend on how the parser is implemented.

It is important to note that at the time of the document conclusion, the implementation of ParsingLogController tool was stable in the functionalities listed in this section. However, a desired functionality that was not tested but partially implemented is the “rotation transparency”. Such functionality enables a more transparent interface if log files that are being currently parsed get suddenly rotated by the application/system.

6.3 Generating Accounting Sessions and Records

An essential step on building a service-dependent Collector is to generate Accounting Sessions and Records based on the AMAAIS Common Accounting Information Model described on Section 4.

The Listing 5 presents a incomplete piece of code that highlights how Accounting Sessions and Records can be constructed using AMAAIS common concepts.

Listing 5: Example on how to construct Accounting Sessions and/or Records based on Listing 4

```
public void run() {
        ParsingLogController logController =
                new ParsingLogController("print", "vpp");
        String currentLine;
        while ((currentLine = logController.read()) != null) {
                // Creates an Accounting Session
                int sessionRef = client.startAcctSession("VPP");
                // Splitting the 'currentLine' based on the
                // 'VPP_LOG_FILE_DELIMITER' static value that
                // represents a log file delimiter
                String[] fields = currentLine
                        .split(VPP_LOG_FILE_DELIMITER);
                // Creating a Record
                AcctRecord record = new AcctRecord();
                // Filling the mandatory record attributes
                Date t = new Date();
                record.setTimestamp(t);
                AcctAttribute attr;
                // Parsing number of pages, represented by
                // 'FIELD_POS_PAGES' static value
                try {
                        int pages = Integer
                                .parseInt(fields[FIELD_POS_PAGES]);
                        attr = AcctAttributeFactory
                                .getAcctAttribute("pages");
                        ((AcctAttrInteger) attr).setValue(pages);
                        record.addAttribute(attr);
                } catch (...) {
                        ...
```

```
30                      }
31                      ...
32              }
33              logController.signalSent();
34              ...
35      }
```

Observing the Listing 5 at line 7 it is described how an Accounting Session is created. The method “startAcctSession()” requires an identifier that is the log ID (usually, the same as the used on “ParsingLogController” constructor). Lines 14-17 describes the instantiation of an Accounting Record and the mandatory attribute that should be fulfilled (timestamp). Lines 18-27 describes how an Accounting Attribute is instantiated using the example of matching the number of pages (type integer). Note that at line 24 it is used an Attribute Factory which grabs the type of such attribute. Even if not strictly necessary, the AMAAIS project assumes the use of an Attribute Dictionary that should be globally common among Collectors, Accounting Clients, and Accounting Servers. Moreover, the missing parts of the code represented by ellipsis may contain specific parts of the implementation – as mentioned before, the Collector developer should understand its necessities and adapt it to the AMAAIS extensibility.

6.4 Notes on the Integration with the Accounting Client

The integration of a Collector and its Accounting Client is made through an API as mentioned before. However, there are some points that should be taken into consideration when developing a service-dependent Collector:

- The Accounting Client is totally responsible on sending/forwarding Accounting Sessions/Records to the Server;
- The Collector component cannot control when the data will be sent to Servers, but it controls if, for example, the parsed data was “given” to the Accounting Client;
- The users can tune the configuration file amaais-client.conf to set what is the buffer size at the Accounting Client side. This option may impact directly on how often the Accounting Client sends Accounting Sessions/Records to the Accounting Server.

6.5 Building a Daemon

Figure 5 presents the class diagram of the daemons developed in the scope of AMAAIS.

Even if daemons are not only related to Collectors, Collector’s developers have the responsibility to build the proper daemon integrating (a) the developed Collector component and (b) the Accounting Client. Therefore, the Listing 6 presents a complete example on how a daemon should look like in the scope of the printing example.

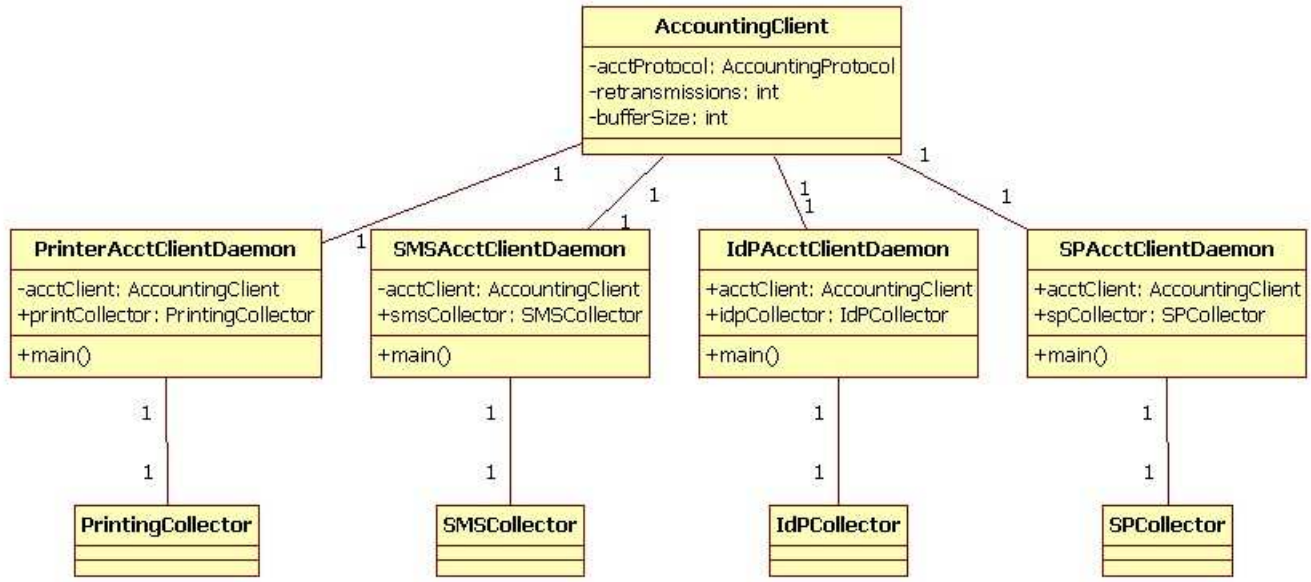

Figure 5: Daemons Class Diagram

Listing 6: Complete example of the Printing Daemon

```
1  public class PrintAcctClientDaemon {
2          private static Logger logger =
3                  Logger.getLogger(PrintAcctClientDaemon.class.getName());
4          private static AccountingClient acctClient = null;
5          private static PrintingCollector printingCollector = null;
6          // Shutdown hook that is called when the daemon is terminated.
7          private static class DaemonShutdownHook extends Thread {
8                  public void run() {
9                          logger.debug("Shutting down the printing
10                                 accounting client daemon");
11                         // Terminate all running components
12                         if(printingCollector != null)
13                                 printingCollector.stop();
14                         if(acctClient != null) acctClient.stop();
15                         logger.info("Printing accounting
16                                 client daemon terminated");
17                 }
18         }
19         public static void main(String[] args) {
20                 BasicConfigurator.configure();
21                 logger.info("Starting the printing
22                         accounting client daemon");
23                 // Register the shutdown hook
24                 DaemonShutdownHook sdHook = new DaemonShutdownHook();
25                 Runtime.getRuntime().addShutdownHook(sdHook);
26                 // Initialize the config util
27                 ConfigUtil config = ConfigUtil.getInstance();
```

```
28                  if(!config.init("amaais-client.conf")) {
29                          logger.error("Error initializing
30                                  the config util");
31                          System.exit(Accounting.RET_ERROR_INITIALIZE);
32                  }
33                  // Initialize the accounting client
34                  acctClient = new AccountingClient();
35                  if(acctClient.init() != Accounting.RET_OK) {
36                          logger.error("Error initializing the
37                                  accounting client");
38                          System.exit(Accounting.RET_ERROR_INITIALIZE);
39                  }
40                  // Start the accounting client
41                  if(acctClient.start() != Accounting.RET_OK) {
42                          logger.error("Error starting the
43                                  accounting client");
44                          System.exit(Accounting.RET_ERROR_START);
45                  }
46                  // Initialize the printing collector
47                  printingCollector = new PrintingCollector();
48                  if(printingCollector.init(acctClient) !=
49                                  Accounting.RET_OK) {
50                          logger.error("Error initializing the
51                                  printing collector");
52                          acctClient.stop();
53                          System.exit(Accounting.RET_ERROR_INITIALIZE);
54                  }
55                  // Start the printing collector
56                  if(printingCollector.start() != Accounting.RET_OK) {
57                          logger.error("Error starting the
58                                  printing collector");
59                          acctClient.stop();
60                          System.exit(Accounting.RET_ERROR_START);
61                  }
62                  while(true) {
63                          try {
64                                  Thread.sleep(...);
65                          } catch (...) {
66                                  ...
67                          }
68                  }
69          }
70  }
```

Observing the Listing 6 it worth to highlight that the main method basically invokes (with “init()” and “start()”) the Collector component and the Accounting Client component. In the “while()” construction the “sleep()” call is not strictly necessary, being just an specific implementation detail of the parsing component.

7 Implementation Documentation

In the following sections we present technical information related to the AMAAIS implementation. The description is not exhaustive but describes the majority of the functionalities and how they were meant to work in the scope of each AMAAIS component.

7.1 AMAAIS SAML-based Protocol for Exchanging Accounting Records (ASPEAR)

The main goal of the protocol is to exchange accounting records. The ASPEAR protocol is based on the SAML specification [4], meaning that message types, attributes, and encapsulation are followed by the SAML standard.

ASPEAR has three characteristics: flexibility, extensibility, and adaptability. It means that the protocol should handle future extensions without redesign and that it should be able to adapt in any context that involves the exchange of accounting records.

7.1.1 ASPEAR as a SAML Protocol Extension

The ASPEAR protocol is based on the SAML 2.0 protocol following all recommendations by the SAML Core document [12] which describes how a Request/Response should be constructed and which attributes (*e.g.*, version, issueInstant, issuer) it must contain. Therefore ASPEAR implements a new profile, with the namespace “urn:mace:switch.ch:doc:accounting:profiles:1.0” and with the prefix “accp” (acronym for ACCounting Protocol).

Basically, the ASPEAR profile differs from how an SAML Assertion message is exchanged. Among the protocols defined by the SAML Core document [12]—especially the “Assertion Query and Request Protocol”—an SAML Assertion with a Request message is used to query a certain kind of information. Applied to the purpose to exchange accounting records, the Client/Server should push records to the other end. In this case an Assertion can be used to inform (as an Accounting Request) which attributes were accounted and their values.

Another difference is that the Response—inside the accounting profile scope—is just used to inform if the SAML Assertion was received/persisted accordingly or not.

7.1.2 Protocol Messages

The ASPEAR protocol is a so-called Request-Response protocol: we therefore define two message types: “Accounting Request” and “Accounting Response”. Each of these messages are described below.

7.1.2.1 Accounting Request

The accounting request is represented by the XML element “accp:AccountingRequest”. The message is formed as follows:

- a header, represented by the XML element “accp:AccountingRequest”, which encapsulates all the other elements
- inside the “accp:AccountingRequest” element, it is mandatory—as described in the SAML Core Request/Response—to include at least one SAML Assertion
- The SAML Assertion must contain: (a) one SAML Issuer, which is the Accounting Client URL (identifier) that generated such metered information (*e.g.*, https://printqueue-accountingclient.ethz.ch), (b) one SAML Subject, which is the SP URL (*e.g.*, https://printqueue.ethz.ch/shibboleth), and (c) one SAML Attribute Statement, which contains all the Attributes that should be transmitted.

The Listing 7 shows an example of the Accounting Request message.

Listing 7: Accounting Response message

```
<accp:AccountingRequest
  xmlns:accp="urn:mace:switch.ch:doc:accounting:profiles:1.0"
  AccountingRequestType="PushAssertionRequest"
  ID="_21f025ca846ca0a4124c1b36d4905821"
  IssueInstant="2010-10-18T15:20:15.600Z"
  Version="2.0">

  <saml:Assertion
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ID="_089ed0c014a534281228eb2f2c94bd87"
    IssueInstant="2010-10-18T15:20:15.484Z"
    Version="2.0">

    <saml:Issuer>
      https://print-accountingclient.ethz.ch
    </saml:Issuer>
    <saml:Subject>
      <saml:NameID
        Format="SPNameQualifier"
      >https://printqueue.ethz.ch/shibboleth</saml:NameID>
    </saml:Subject>
    <saml:AttributeStatement>
      <saml:Attribute
        FriendlyName="recordTimestamp"
        Name=".2.16.756.1.2.7.1.1.11"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue
          xsi:type="accp:date">
          Mon Oct 18 17:20:11 CEST 2010
```

```
31          </saml:AttributeValue>
32        </saml:Attribute>
33
34        ...
35
36      </saml:AttributeStatement>
37    </saml:Assertion>
38  </accp:AccountingRequest>
```

It should be observed that the “accp:AccountingRequest” element has the attribute “AccountingRequestType”. This attribute was created due to the necessity to specify which is the type of the request. In the current implementation of the Accounting Server and the ASPEAR protocol, the only “AccountingRequestType” available is the “PushAssertionRequest”. It means that the client is generating an Accounting Request message to push Assertions containing Accounting Sessions/Records.

7.1.2.2 Accounting Response

The Accounting Response is represented by the XML element “accp:AccountingResponse”. The message is formed as follows:

- a header, represented by the XML element “accp:AccountingResponse”, which encapsulates all other elements
- the “accp:AccountingResponse” element which must contain one SAML Status element
- The SAML Status contains one SAML StatusCode element which expresses a standardized code to describe how the Accounting Request was processed
- The SAML Status can contain (not mandatory) a SAML StatusMessage which textually describes the SAML StatusCode element. For example, if the SAML StatusCode express an error code, the SAML StatusMessage can contain a text with the error description.

Considering the information above, Listing 8 shows a brief example on how the Accounting Response message is composed.

Listing 8: Accounting Response message

```
1  <accp:AccountingResponse
2    xmlns:accp="urn:mace:switch.ch:doc:accounting:profiles:1.0"
3    ID="_b5164b875b42e51a38878a09d526c785"
4    IssueInstant="2010-10-18T15:20:15.787Z"
5    Version="2.0">
6
7    <samlp:Status
8      xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
9      <samlp:StatusCode
```

```
10          Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
11       <samlp:StatusMessage>Persisted  Successfully</samlp:StatusMessage>
12    </samlp:Status>
13
14 </accp:AccountingResponse>
```

It is important to mention that the definition of the SAML Status element structure was not modified (as described in the SAML Core document [12]). Therefore, even if the ASPEAR implementation does not bring any other SAML element within the SAML Status, the ASPEAR core can be extended to allow other kinds of SAML structures.

7.1.3 Protocol Class Diagram

Figure 6 depicts the common classes used to build an ASPEAR message. Theses common classes should be used by the protocol implementation and any other piece of software that uses ASPEAR.

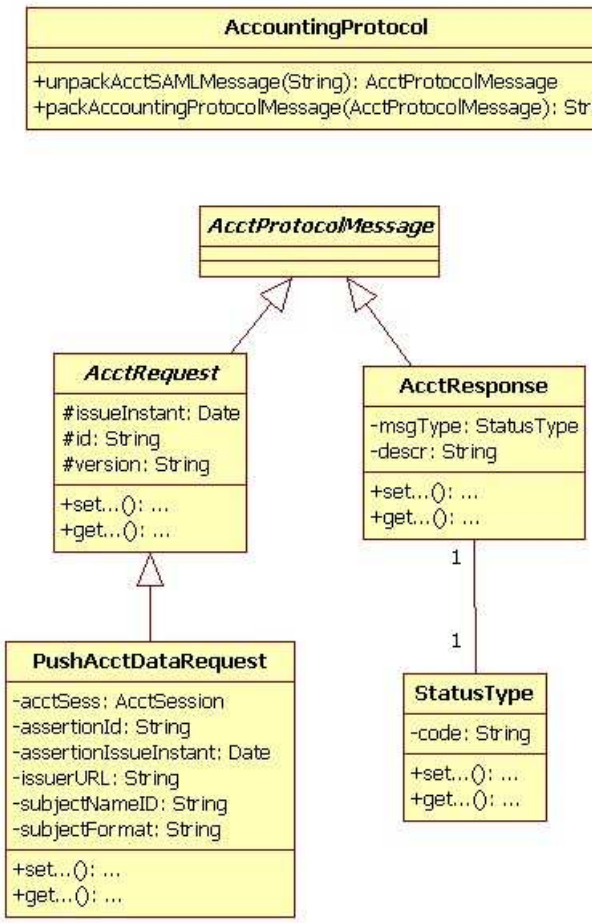

Figure 6: ASPEAR implementation class diagram

The AccountingRequest class abstractly defines a generalization of possible requests. The concrete class that can be instantiated is the “PushAcctDataRequest”, which represents a specific type of Request message. On the other hand, the AccountingResponse is a concrete class since it is mandatory to generate responses containing some specific information (*e.g.*, StatusType).

7.1.4 Protocol Sequence Diagram

Figure 7 depicts ASPEAR messages exchanged between components.

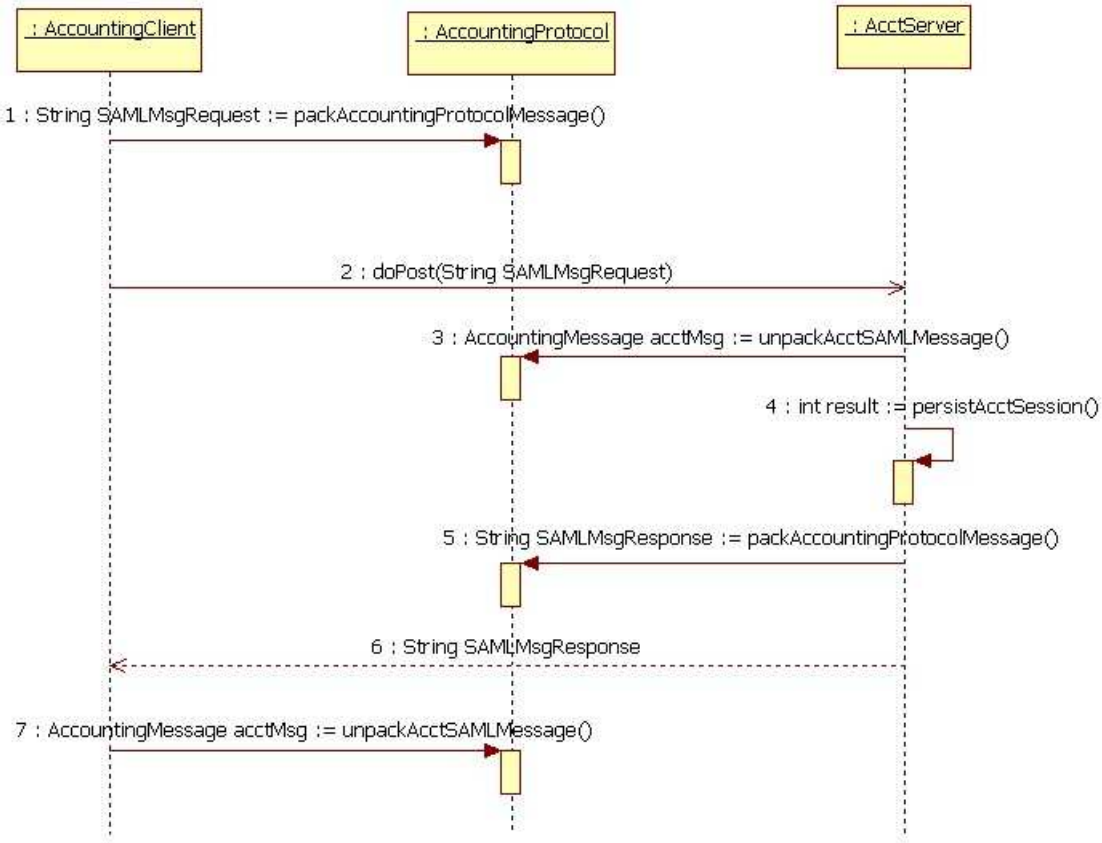

Figure 7: ASPEAR messages exchanged between implementation components

Assuming that the Accounting Client has a “AcctProtocolMessage” object with all the necessary information, it calls the API of the Accounting Protocol (ASPEAR) to marshal it. The result is a SAML-based message (in other words, ASPEAR message) ready to be sent. The Accounting Client sends the ASPEAR message through HTTP (POST method) to the Accounting Server. Once the message is received, after checking its validity, the Accounting Server unmarshals it using the ASPEAR implementation API. Once the Server gets the object reflecting the Accounting Session and its Accounting Records, it persists it in the Accounting Database. Just after the attempt to persist the accounting information, it generates an Response message with the result (success or failure, error).

In order to generate the response, the Server packs an “AcctProtocolMessage” object that reflects the AccountingResponse type and sends the generated string (to the Accounting Client, in this case). In a last step the Accounting Client unpacks the response message and checks the result to evaluate which action should be taken as the next sending iteration (*e.g.*, a retry).

7.1.5 Interface / API

The ASPEAR protocol has two public methods:

`public String packAcctProtocolMessage(AcctProtocolMessage acctMsg)`
Pack all information contained in the AcctProtocolMessage object into a SAML string according to the type of the AcctProtocolMessage.

`public AcctProtocolMessage unpackAcctSAMLMessage(String acctMsg)`
Unpack a SAML string into the AcctProtocolMessage object. Basically, the method parses the string and populates the object accordingly (depending on what kind of message was received).

7.2 Collector

The Collector component is mainly responsible to observe metered data (from Meter component) and turn it in accounting information. Moreover, the Collector is responsible to use the Accounting Client API to push Accounting data to the Server.

7.2.1 Interfaces

The Collector provides an interface to communicate actively with other system’s components. In fact, the Collector interface class has some methods that should be implemented by classes that “implements” the Collector object (*e.g.*, IdPCollector, SPCollector). The method start(), stop(), and init() are mandatory to be implemented since the junction of Collectors and Accounting Clients form system’s daemons.

7.2.2 Component Architecture

Figure 8 depicts the Collector class diagram with some examples of service-independent and service-dependent use cases.

The Collector is an abstract class that defines a common interface to subsequent classes that extends it. Basically, as described in Section 6, anyone can develop a Collector to handle accounting in service-dependent scenarios by extending the Collector abstract/interface class.

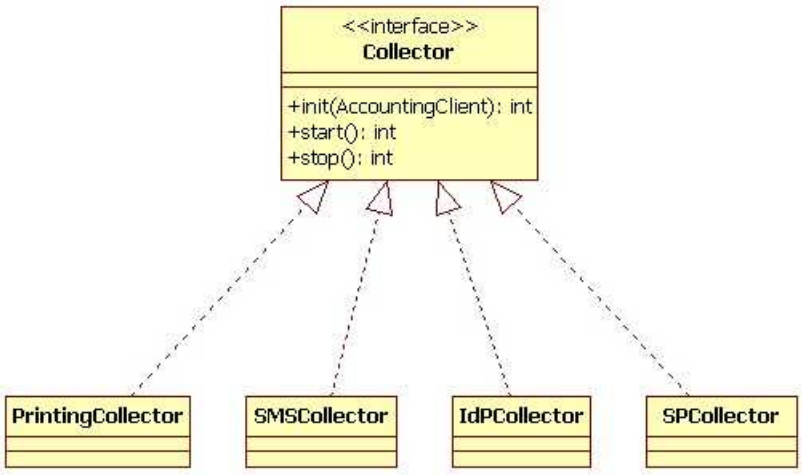

Figure 8: Collector Class Diagram

7.3 Accounting Client

The Accounting Client component is responsible to send Accounting information. This component has a well-defined API to interface with, for example, the Collector. The Accounting Client is not a daemon itself: it works together with the Collector component, which basically receives information generated by, *e.g.*, a parser (Collector), and organizes such accounting information to be sent to a given Accounting Server. It uses ASPEAR to pack and unpack accounting messages.

7.3.1 Interfaces

The Accounting Client API defines the following methods:

`public int startAcctSession()`
Creates a new accounting session. After creating a session, accounting records can be assigned to this new accounting session. The method returns the local-scope reference of the accounting session or an error code.

`public int stopAcctSession(int sessionRef)`
Terminates an existing accounting session. If there are any pending accounting records, these are sent to the accounting server. If no accounting record is specified it returns an error. After termination no more accounting records can assigned to the specified accounting session. `sessionRef` specifies the local-scope reference of the accounting session (that is returned by the startAcctSession). The method returns an error code.

`public int addAcctRecord(int sessionRef, AcctRecord record, boolean isLast)`
Assigns an accounting record to an accounting session. Multiple records can be assigned to a session before sending them or every record can be sent immediately. The last record of a session has to be specified using the `isLast` parameter. `sessionRef` specifies the local-scope reference of the accounting session; `record` is the accounting record to

be assigned to this session; `isLast` is to be set to true if this record is the last record in the session and to false otherwise. The method returns the local-scope reference of the accounting record (the record number) or an error code.

7.3.2 Component Architecture

Figure 9 depicts the Accounting Client class diagram.

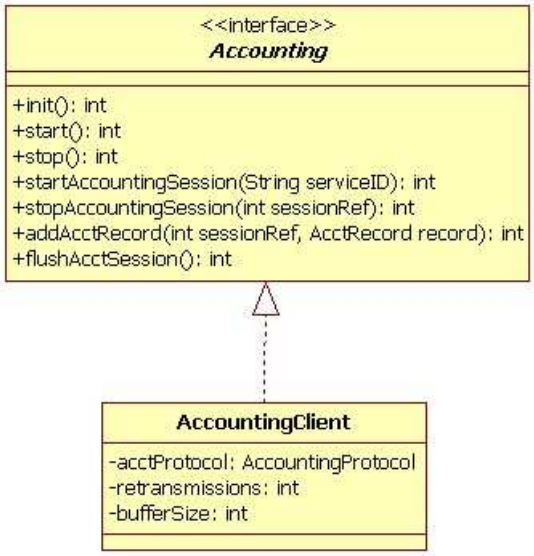

Figure 9: Accounting Client Class Diagram

7.4 Accounting Server

The *Accounting Server*, which is implemented as a Java-based servlet, receives SAML requests over HTTP which contain accounting sessions that have to be stored in the database. This component uses the AMAAIS SAML-based protocol component for parsing the received SAML requests, and the *Accounting Database* component for storing the account sessions.

7.4.1 Interfaces

The Accounting Server defines the following methods:

`doPost(HttpServletRequest req, HttpServletResponse resp)`
Handles the execution of the HTTP post request. Parses the received SAML-based request and stores accounting session using the *Accounting Database* component.

7.4.2 Component Architecture

Figure 10 illustrates the class diagram of the *Accounting Server* component. The *AccountingServer* class implements the *HttpServlet* interface.

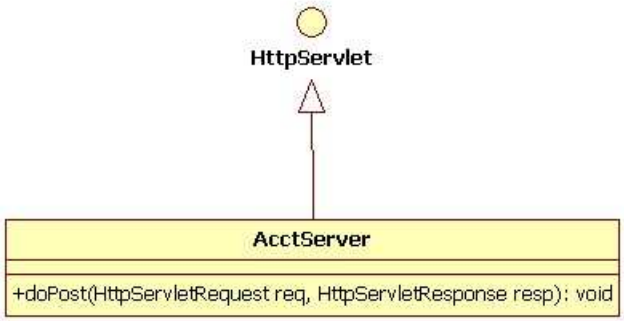

Figure 10: Accounting Server Class Diagram

7.4.3 Component Behavior

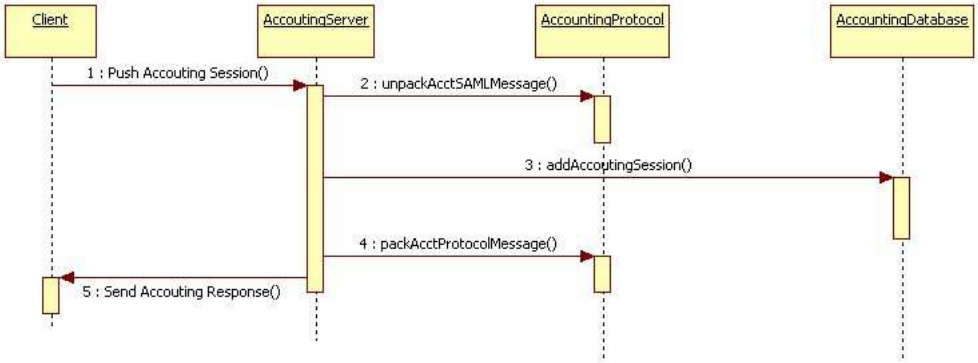

Figure 11: Accounting Server Sequence Diagram

Figure 11 illustrates how the Accounting Server interacts with the other AMAAIS components. Client sends an accounting session to the server (1), in order to be stored. The Accounting Server unpacks the SAML-based message using the accounting protocol (2). The received session is stored using the accounting database component (3). The response is then packed (4) using the protocol and sent to client (5).

7.5 Accounting Database

The accounting database component handles the storage of accounting sessions. Its implementation uses the Hibernate Java persistence framework, for mapping between the

Java-based objects oriented model and the relational back-end. Accounting sessions are stored in a PostgreSQL database.

7.5.1 Interfaces

The Accounting Database defines the following methods:

`public void addAccountingSession(AcctSession acctSession)`
Stores the specified accounting session in database. If the session already exists, the new accounting records will be added to the existing session.

`public AcctSession getAcctSession(String sessionID)`
Returns the accounting session and associated accounting records identified by the session ID.

`public List<AcctSession> getAcctSessions(Date from, Date to)`
Returns the list of accounting sessions and associated accounting records that have their start time in the specified time interval.

`public List<AcctSession> getAcctSessions(Date from, Date to, String service)`
Returns the list of accounting sessions and associated accounting records that have their start time in the specified time interval and are related to the specified service.

`public void deleteAcctSession(String sessionID)`
Deletes an accounting session and associated accounting records from the database.

7.5.2 Component Architecture

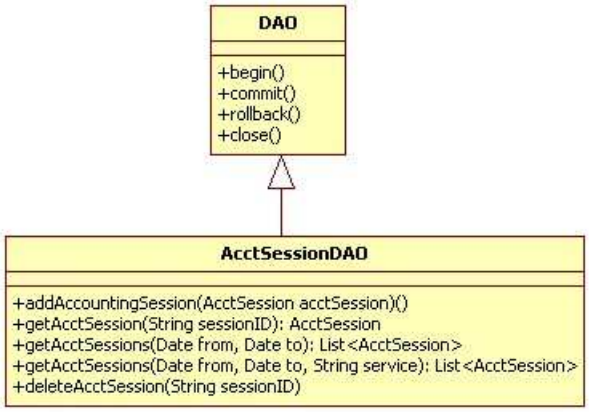

Figure 12: Accounting Database Class Diagram

The UML class diagram of the database component is presented in Figure 12. The abstract class *DAO* implements the basic transactional functionalities (*begin*, *commit*, *rollback*), while its subclass *AcctSessionDAO* contains specific methods for storing accouting sessions.

7.5.3 Data Model

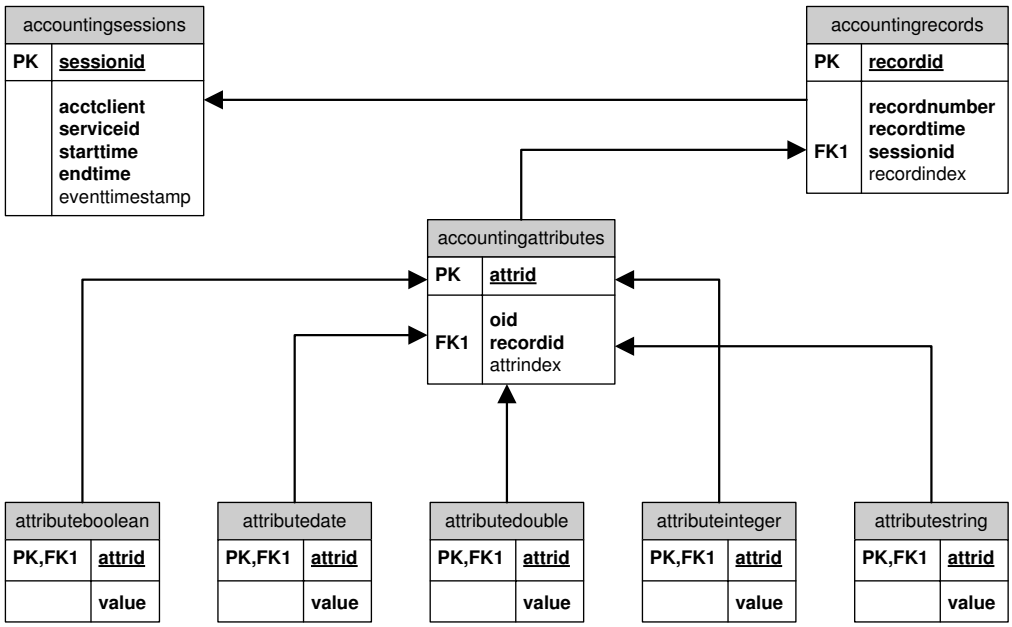

Figure 13: AMAAIS Data Model

Figure 13 illustrates the AMAAIS data model used for storing accounting sessions into the PostgreSQL database. Accounting sessions are stored in the *accountingsessions* table. The table *accountingrecords* stores the accounting records. There is a one to many relationship between *accountingsessions* and *accountingrecords*. Attributes are stored in the tables *accountingattributes* tables, while values of different types are kept in *attributeboolean*, *attributedate*, *attributedouble*, *attributeinteger* and *attributestring* tables.

7.6 Visualization Component

AMAAIS collects large sets of data. To make these piles of data easily consumable for the user, the visualization component helps her to get a quick overview and understanding of the data. A reporting tool provides the user with different types of charts to visualize the data and also export the data in common file formats (*e.g.*, for further processing). Concerning the monitoring aspect of AMAAIS, the charts also allow the viewer to easily identify anomalies in the data which otherwise would be hard to find in the raw data.

7.6.1 Evaluation of the visualization software

For the visualization of the collected data on the accounting server, the software has to fulfill the following criteria:

- provide a web interface to show reports

- support common export formats for report (PDF, CSV, Excel)
- provide an interface to an SQL database
- run on multiple operating systems (*i.e.*, Windows, Linux, Mac OS)
- provide a user-friendly interface to design reports or adapt existing reports
- provide good documentation about installation and usage
- provide a large user base to ensure future maintenance and development of the software

The following software products were evaluated:

- Business Intelligence and Reporting Tools (BIRT): `http://eclipse.org/birt/`
- Pentaho Reporting: `http://reporting.pentaho.org/`
- JasperReports: `http://jasperforge.org/projects/jasperreports`

All of these products are open source. They are available as a free community edition as well as versions with commercial support. For the AMAAIS project, BIRT is the software that best matches the criteria listed above. The biggest differences between BIRT and the other products are the user-friendly interface and the sound documentation. An advantage of JasperReports and Pentaho Reporting is that they are parts of business intelligence (BI) suites that offer the BI typical extract transform load (ETL) functionality. BIRT does not provide that, but as the ETL functionality is not needed in AMAAIS, that is irrelevant.

7.6.2 Eclipse BIRT

BIRT consists of two main components: a visual report designer (Eclipse plugin) for creating BIRT Reports and the BIRT viewer, a component for generating reports. The report designer uses a charting engine that can be used as a standalone component to integrate charts into Java applications. For Windows, a standalone application to design reports is available, the RCP Report Designer. It uses the Eclipse Rich Client Platform (RCP) technology. The information given below refers to the current BIRT version 2.6.1.

7.6.2.1 Functionality

Supported platforms Platform independent (requires Java JDK 1.5 or newer)

Data sources JDBC, XML, CSV, web services, POJO (plain old java objects)/Java Beans

Report export formats HTML, PDF, Postscript, RTF, CSV, Word Doc, XLS, PPT

Chart formats PNG, JPG, BMP, SVG

Chart types Bar, line, area, pie, meter, scatter plot, stock, bubble, difference, gantt, tube, cone, pyramid, radar

Further functionality Preview of report in report designer, multiple data sources per report, drill down analysis in report, internationalization (currency, character set, ...), sub-reports, parametrized reports

7.6.3 Installation and Configuration

7.6.3.1 Report Designer

The Report Designer for BIRT is available as an Eclipse plugin or as a standalone application. The installation instructions are given on the BIRT web pages: `http://www.eclipse.org/birt/phoenix/build/`.

7.6.3.2 Viewer web application

As a web interface the BIRT Viewer J2EE Application (see `http://www.eclipse.org/birt/phoenix/deploy/`) is used. The installation instructions are given on the web site: `http://www.eclipse.org/birt/phoenix/deploy/viewerSetup.php`. For the test bed installation the BIRT Viewer was installed as a web application in Tomcat 6.

7.6.4 Example report

The example bar chart below has been created using Eclipse Report Designer. The following steps are necessary to create the chart:

- Define data source
- Define data set
- Add chart to report page

The report design file in XML format is included in the sources of the first version of the AMAAIS release.

7.6.4.1 Data source

The data source is defined as follows:

Driver `com.mysql.jdbc.Driver (v5.1)`

Database URL `jdbc:mysql://127.0.0.1:3306/amaaisdb`

User name `<database username>`

Password `<password>`

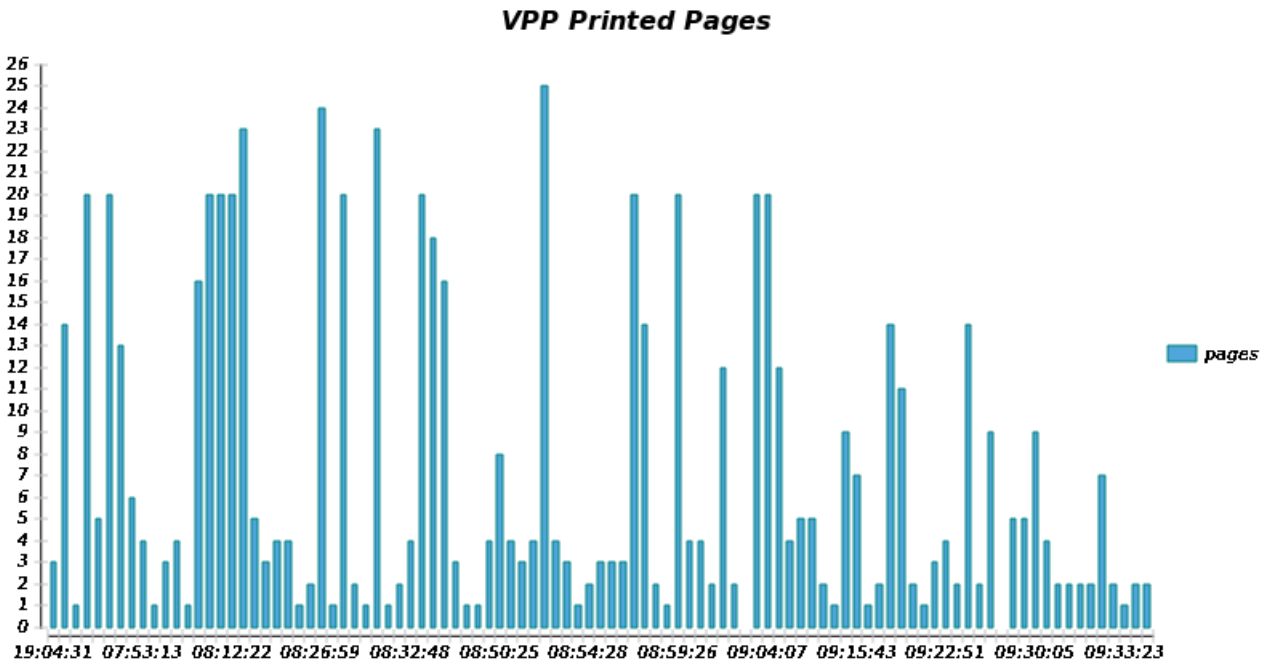

Figure 14: Chart of the example report vpp-pages

7.6.4.2 Data set

The data set uses the data source defined above and the following SQL query to get the visualized data:

```
SELECT ‘AccountingSessions‘.‘event-timestamp‘,
       AttributeInteger.‘attr-value‘ AS value
  FROM AccountingSessions LEFT JOIN AccountingRecords USING (‘session-id‘)
       LEFT JOIN AccountingAttributes USING (‘record-id‘)
       LEFT JOIN AttributeInteger USING (‘attr-id‘)
 WHERE AccountingSessions.‘service-id‘ = ’VPP’
       AND AccountingAttributes.‘oid‘=’2.16.756.1.2.7.1.3.1’;
```

7.6.4.3 Chart

The bar chart is defined with the following properties:

Chart Type 2-dimensional bar chart

Y axis attribute value of the attribute “printed pages”

X axis event timestamp

7.6.4.4 Report output

The chart of the report output is shown on figure 14.

8 Summary and Conclusions

The main achievement of AMAAIS Phase 2 was the complete development of AMAAIS components. As a result, the prototype produced respects all the requirements described in Phase 1, and all component functionalities refined during the beginning of Phase 2.

Summarizing, the current implementation confirms the proposal from Phase 1 that the architecture could be extended with new components, since the component's hierarchy (from the Meter to Accounting Server) is defined in a flexible manner – multiple instances of the same component (*e.g.*, Collectors) can be instantiated to avoid a huge demand, for example. Moreover, it is possible to exchange Accounting records between domains inside a federation, and also domains from different federations. The additional feature attached to the system is the forwarding of Accounting Records. The employed mechanism allows – in a well-known XML-based description format (since it is also used by Shibboleth's IdP implementation) – to express policies/rules to forward records. The component that evaluate forwarding rules was developed by AMAAIS.

Even if the prototype was not extensively tested nor deployed in a production environment, the service-dependent Collectors developed for SMS and Printing were successfully tested in a test-bed. Also, the service-independent Collectors demonstrated to run in compatibility with the Shibboleth implementation of SP and IdP (without log files set to debug mode).

Related to the enhancements, the AMAAIS team understands that the prototype version 1.0 – last version until the conclusion of this document – has its limitations (mainly regarding configuration parameters and security) and already planned to enhance it on AMAAIS Phase 3. The Phase 3 will be mainly focused on building a security mechanism to exchange Accounting Records, to come up with a solution related to exchange charging information, and refining the visualization and database parts.

Terminology

IdP: Identity Provider.

SP: Service Provider.

Institution: A large important organization such as a university that, in the scope of this document, provide identity credentials.

Resource: An available supply that can be used/consumed, and, in the scope of this document, provided by Service Providers.

AAI environment: The entire set of conditions under which the Authentication and Authorization Infrastructure is operated, as it relates to the hardware (*e.g.*, servers, network segments), operating platform (*e.g.*, Shibboleth), or operating system.

DS: Discovery Service.

Acknowledgement

This deliverable was made possible due to the large and open help of the members of the AMAAIS project. Many thanks to all of them.

References

[1] AMAAIS Project. *Accounting and Monitoring of AAI Services – Deliverable Phase 1*, October 2009. Available at: http://www.csg.uzh.ch/research/amaais. Visited on: Dec. 2010.

[2] B. Pfitzmann and M. Waidner. Federated identity-management protocols. *LECTURE NOTES IN COMPUTER SCIENCE*, 3364:153, 2005.

[3] AMAAIS Project. *Accounting and Monitoring of AAI Services – Project's Website*, June 2009. Available at: http://www.csg.uzh.ch/research/amaais. Visited on: Dec. 2010.

[4] OASIS. *Security Assertion Markup Language (SAML)*, August 2009. Available at: http://www.oasis-open.org/committees/security. Visited on: Dec. 2010.

[5] Internet2. *OpenSAML*, August 2009. Available at: http://www.opensaml.org. Visited on: Dec. 2010.

[6] AMAAIS Project. *AMAAIS Phase 1: Scenarios, Requirements, and High-Level Architecture*, October 2009.

[7] ETHZ Informatikdienste. *VPP – Verteiltes Printen und Plotten*. Available at: http://www.vpp.ethz.ch/. Visited on: Dec. 2010.

[8] ETHZ Informatikdienste. *SMS Gateway der ETH Zrich*. Available at: http://www.sms.ethz.ch/. Visited on: Dec. 2010.

[9] Internet2. *IdP Logging - Shibboleth 2 Documentation - Internet2 Wiki*, August 2009. Available at: https://spaces.internet2.edu/display/SHIB2/IdPLogging. Visited on: Dec. 2010.

[10] IETF – Internet Engineering Task Force. *RFC 2866 – RADIUS Protocol*, June 2000. Available at: http://www.ietf.org/rfc/rfc2866.txt. Visited on: Dec. 2010.

[11] IETF – Internet Engineering Task Force. *RFC 3588 – Diameter Base Protocol*, December 1997. Available at: http://www.ietf.org/rfc/rfc3588.txt. Visited on: Dec. 2010.

[12] OASIS. *Security Assertion Markup Language (SAML) – Core Document*, March 2005. Available at: http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf. Visited on: Jan. 2011.