Accounting and Monitoring of AAI Services

Phase 4

Deliverable D4
Final Solution Documentation and Trial Results

The AMAAIS Partners

University of Zürich (UZH), Switzerland
SWITCH (SWITCH), Switzerland
ETH Zurich, 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: Final Solution Documentation and Trial Results
Type: Public
Editor(s): Christos Tsiaras
E-mail: tsiaras@ifi.uzh.ch
Author(s): Guilherme Sperb Machado (GM), Christos Tsiaras (CT), Andri Lareida (AL) Samuel Liniger (SL), Martin Waldburger (MW), Francesco Croci (FC) Matteo Corti (MC), Thomas Baerecke (TB), Burkhard Stiller (BS)
Doc ID: D4
Delivery Date: 30.04.2013

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
0.1	2013-03-18	CT	Deliverable created with content already compiled from AMAAIS members.
0.2	2013-03-22	MC, CT	Design and SMS and Printing sections are added.
0.3	2013-04-08	SL	Ethical hacking section is added.
0.4	2013-04-12	AL, TB	Stages, Persistency and Deployment sections are added.
0.5	2013-04-19	MC, FC	Updated text in SMS and Printing section.
0.6	2013-04-22	TB, AL, CT	Updated text and figures in all sections.
0.7	2013-04-23	GM	Input for the security section is added.
0.8	2013-04-24	GM	Text for the Trials section is added.
0.9	2013-04-29	MW, AL, CT, TB	General text revision.
1.0	2013-04-30	CT, BS	Final version released.

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 ETH Zurich—is to extend the current SWITCH Authentication and Authorization Infrastructure (AAI) with accounting and monitoring support. The project has been initiated in 2009, and with the completion of its fourth and final project phase at the time of publication of this deliverable, the final AMAAIS solution has become available. AMAAIS enables Shibboleth-based federations to aggregate, process, and store accounting and monitoring information. The deliverable at hand provides the documentation of the final solution. It, thus, constitutes the reference document for the AMAAIS project and its solution.

The deliverable addresses a technical audience with an interest in accounting and monitoring functionality for services offered in a federation environment. With this audience in mind, readers are recommended to have a look into the introduction chapter at the very beginning in order to quickly learn about the basic AMAAIS concepts and terminology of relevance as well as the interplay of AMAAIS core components (accounting client and accounting server) and AMAAIS accounting applications. For further interest in key design concepts and details on how these concepts are reflected in the system implementation, readers are recommended to initially browse through the respective two chapters—and to come back selectively to this content after studying the subsequent chapter on deployment guidelines.

The depicted steps regarding deployment will be of primary interest to the intended audience. They touch on all steps of relevance in a structured manner, ranging from technical pre-requisites to be met, explanations on the assembly of an accounting application, to information on available configuration options. These deployment guidelines also include an example deployment which may serve as a simple starting point. This information may be complemented by implementation details obtained from the set of three use cases adopted in the project—two of them (SMS and printing) applying service-specific accounting, while the third use case is service-independent in the sense that its accounting application creates events from Identity Provider log files.

This deliverable does, however, not only show how AMAAIS is designed, how it is implemented, and how AMAAIS can be deployed—it also shows that the AMAAIS project has produced an accounting and monitoring solution that is proven to work and to scale well with an increasing number of events to be treated. First, the AMAAIS implementation has been tested thoroughly and successfully with a test unit coverage of 77%. Second, AMAAIS successfully passed both a black-box and a white-box hacking attempt. Third, and and very important, long-term trials with operational conditions have been designed, conducted, and assessed for all three use cases considered in the project.

These trials have revealed bugs—all of which are fixed in the final solution—and they have revealed that each and every event entering the AMAAIS accounting system was persisted in the respective AMAAIS Server database in full and correctly. The trial for the Identity Provider use case focused in addition on scalability and performance: For each of the three defined levels of low, medium, and high load levels (measured by the amount of Identity Provider requests per time unit), the AMAAIS solution was able to perform its accounting tasks gracefully and correctly.

2 Introduction

The purpose of this deliverable is to document the highly configurable accounting solution provided by AMAAIS (Accounting and Monitoring of AAI Services). AMAAIS extends the SWITCH Authentication and Authorization Infrastructure (AAI) and its services offered by accounting and monitoring functionality. AMAAIS enables Shibboleth-based federations to aggregate, process, and store accounting and monitoring information. This deliverable targets a technical audience, in particular, IT specialists working in institutions and organizations, such as universities, hospitals, and government offices, with an interest to deploy an accounting system in a Shibboleth-based federation.

The accounting and monitoring information that AMAAIS handles covers the usage of an IdP (Identity Provider) or services provided by federation members. The interplay of AMAAIS Accounting Applications (AA) and AMAAIS core components is visualized in Figure 1. AAs parse log files or databases and create events. The AA calls the `AccountingClient` (AC) API, and the event will enter the AC pipeline. After passing along the AC pipeline, the event will be transmitted to one or multiple `AccountingServers` (AS), pass along the respective AS pipeline(s), and will be persisted to a database eventually.

Pipelines are composed from `Sources` and `Sinks`. `Sinks` receive and process `Events`, `Sources` produce `Events`. Pipelines can be constructed by combining `Source` and `Sink` components in any wished for order, so that events may be, for instance, queued, filtered, persisted, or transmitted. Highly configurable pipelines allow for immense flexibility. This is especially beneficial in federations, where different services are managed by different organizations, but with a need for an inter-operable accounting and monitoring.

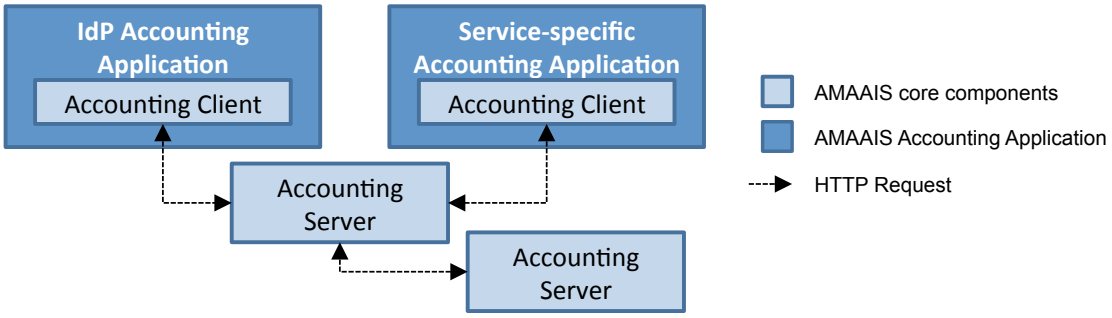

Figure 1: Interplay of AMAAIS Accounting Applications and AMAAIS Core Components.

The remainder of this document is structured as follows. The subsequent chapter describes the design of the AMAAIS solution by presenting key characteristics and concepts of the accounting system. This is followed by a system implementation chapter which explains all AMAAIS components with their respective functionality. Driven by these implementation details, a deployment guideline describes what is needed prior and during an AMAAIS deployment. This chapter gives also an overview on possible future extensions of AMAAIS. The deployment guideline is complemented with a chapter that describes in detail the use cases for which AMAAIS has been deployed, tested, and trialed as the primary accounting system. Testing, trial results as well as results from a conducted black-box and white-box hacking attempt are documented in a dedicated chapter, before the deliverable presents a summary and draws final conclusions.

3 Design

The `AccountingServer` and `AccountingClient` are based on a simple `Event` processing architecture. The main objects of these model are the `Event`, `Source`, and `Sink`. Any set of a number of `Sources` and/or `Sinks` define a pipeline (Figure 2) for both accounting client and server. The pipeline contains all the actions that will be taken for each `Event` during the accounting procedure, such as `Event` cloning, transmitting, storing etc.

AMAAIS include a set or instruments that handle those actions. However if more specific demands, such as accounting reports visualization is needed, the AMAAIS components can be extended in order to support further functionality.

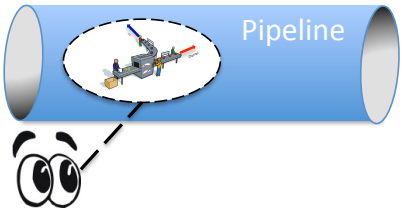

Figure 2: AMAAIS Pipeline.

3.1 Event

As its name suggests, an `Event` represents a measurable real-life event (*e.g.*, a system login, a print job, a door access card swipe) which needs to be accounted or monitored. An `Event` as shown in Figure 3 consists of two parts: a payload that contains the data describing the event and a collection of headers that allows additional metadata to be associated with the event.

The payload may be any message, or set of data, that describes the actual event being reported. In the case of a printing event this might include things like number of pages printed and number of pages printed in color.

The headers general provide metadata about the event. This might include things like a unique identifier for the event, state information accumulated during processing of the event, or a timestamp indicating when the event was produced.

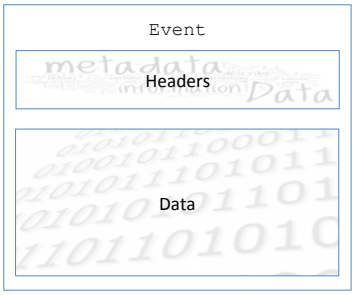

Figure 3: AMAAIS `Event`.

3.2 Source

A `Source` is an `Interface` that is implemented by any object that produces `Events`. `Sources` may, for example, read structured files such as log files, or database tuples, and produce `Events` based on the data they find within. Furthermore, every AMAAIS component that generates `Objects` of type `Event` is implementing the `Source Interface`, even if no access to log files or databases is needed. Such a mechanism is illustrated in Figure 4. Since a `Source` is inherently use-case specific, the developer of the accounting system will need to develop this.

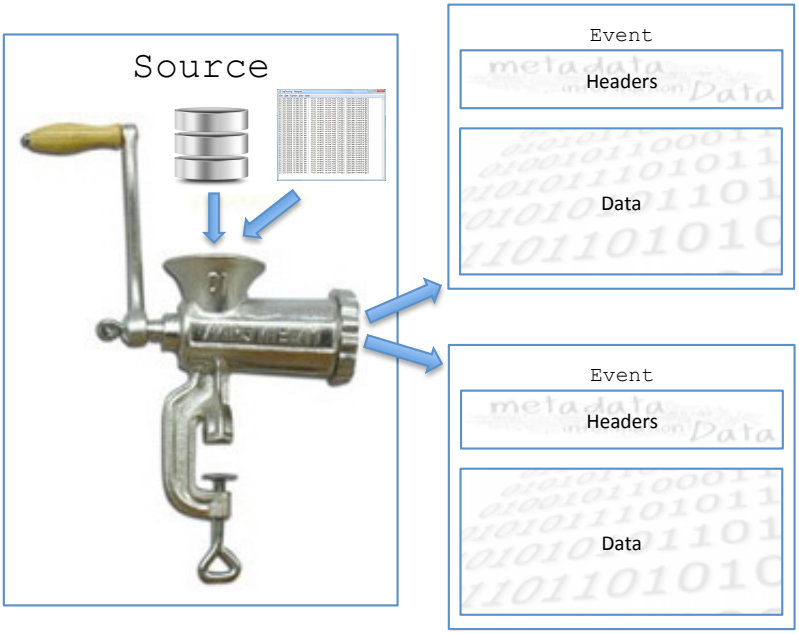

Figure 4: AMAAIS `Source`.

3.3 Sink

A `Sink` is an object that receives and processes `Events`. There are two subtypes of `Sink`, `Stage` and `Channel`, which distinguish the expected role that a `Sink` will play. An illustration of a `Sink` is presented in Figure 5.

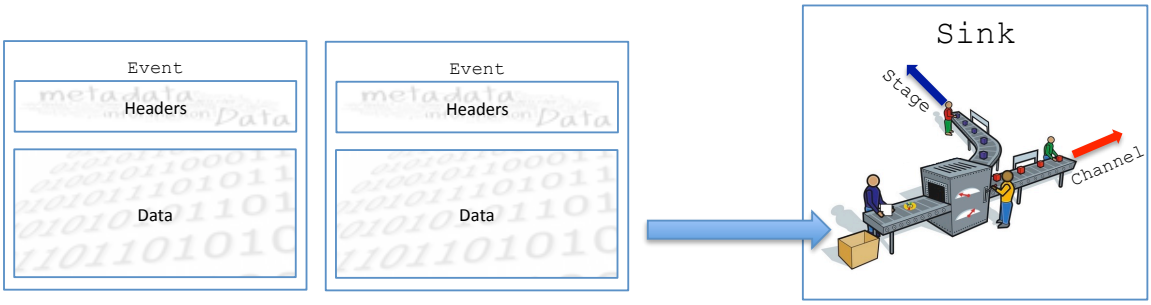

Figure 5: AMAAIS `Sink`.

3.3.1 Stage

A `Stage` (Figure 6), is a `Sink` that performs some computation upon the `Events` it receives. There is no limit to what a `Stage` can do but in general they usually either add information (*e.g.*, by looking up data in a database) to an Event or transform existing information (*e.g.*, by hashing data in order to obfuscate it). Furthermore, a `Stage` is the component that can transmit and persist `Events` to a database (DB).

There is an additional subtype of `Stage` known as the `SourceStage` which also implements `Source`. Most, but not all, implementations of `Stage` are actually implementations of `SourceStage` since they can both receive and publish `Events`.

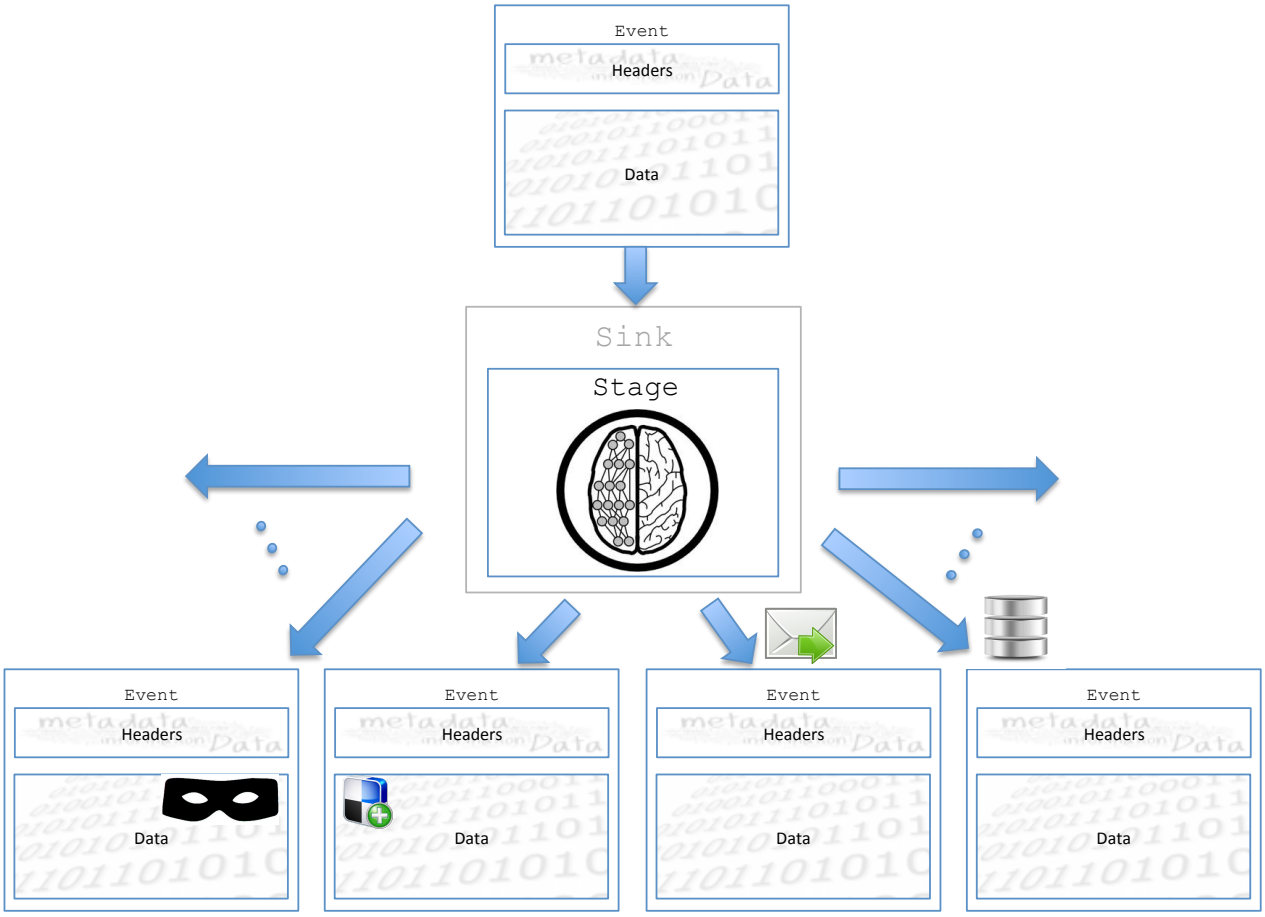

Figure 6: AMAAIS `Stage`.

3.3.2 Channel

A `Channel` is a `Sink` that is responsible for forwarding the messages it receives to the next processing step(s). So, in general a `Source` contains a channel to which it pushes `Events`. As illustrated in Figure 7 the `Channel` then forwards those messages to one or more `Stages`.

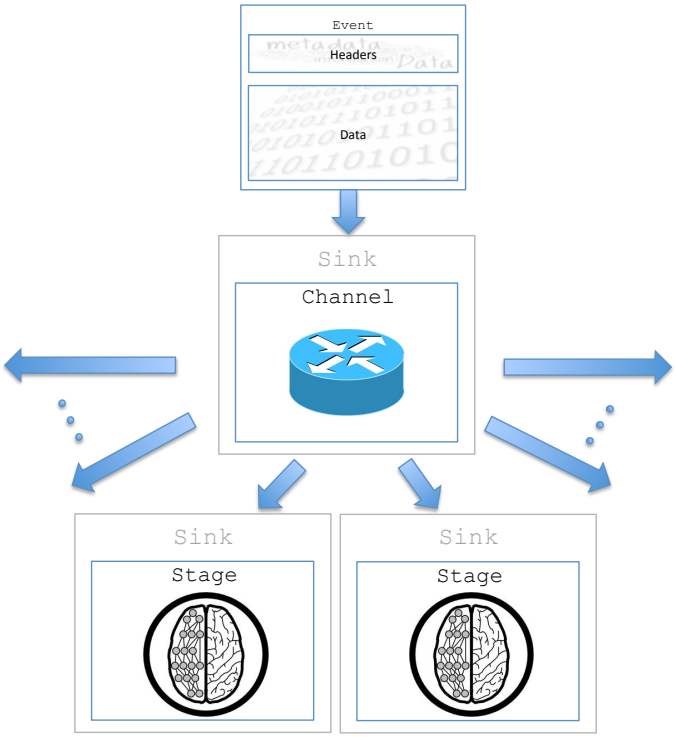

Figure 7: AMAAIS `Channel`.

3.4 Accounting Client

The `AccountingClient` is a `Source` that exposes an API that allows services to feed `Events` in to a configured event processing pipeline which ultimately publishes them to one or more `AccountingServers`. The `AccountingClient` is responsible for loading the employed event processing pipeline at startup time and reloading it when its configuration changes. An `AccountingClient` illustration can be seen in Figure 8.

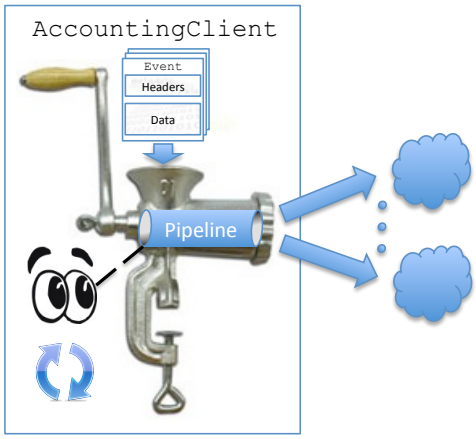

Figure 8: AMAAIS `AccountingClient`.

3.5 Accounting Server

The `AccountingServer` (Figure 9) is an `HttpServlet` that receives event information, usually in batches, from an `AccountingClient`. Like the `AccountingClient` it operates as a `Source` that feeds Events in to a processing pipeline. This pipeline will usually persist data to a database (from which reports are generated) and may publish these `Events` to other `AccountingServer`(s). Also, like the `AccountingClient`, it is responsible for (re)loading the employed event processing pipeline.

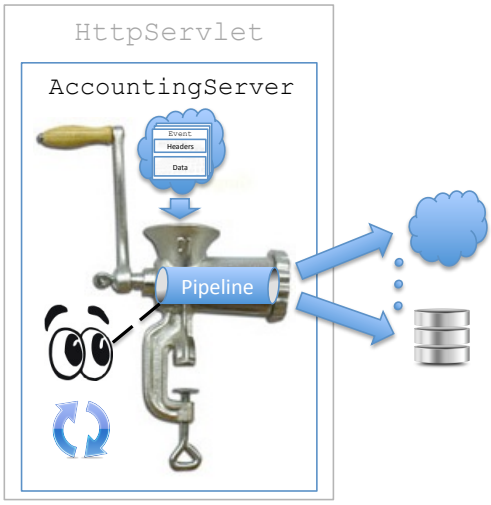

Figure 9: AMAAIS `AccountingServer`.

3.6 Metadata

The requirements of AMAAIS with its protocol require SAML (Security Assertion Markup Language) metadata extensions. Three new main types have been defined in order to support the AMAAIS protocol, namely `PublishEventRequest`, `AccountingEventConsumer`, and `AccountingEventProvider`. These three types reflect the message, the client, and the server, respectively. There are two more standrad types which were renamed, in order to fit the accounting terminology: `EventConsumingService` which is a SAML `Endpoint` and `SupportedEventType` which is of the standard type `anyURI`.

Since the AMAAIS metadata is an extension of SAML metadata it supports all the handling and signing options used by Shibboleth IdP and SP. Tools that generate metadata have to be extended though, in order to support the AMAAIS-specific features.

4 System Implementation

In this chapter the core of the AMAAIS and all the available components of the accounting system such as Channels, and Stages are briefly presented. Furthermore, significant aspects concerning the security of the system are discussed. At the end of this chapter the reader should be able to understand how to combine all the AMAAIS components in order to configure a personalized accounting system according to the needs of the specific scenario.

4.1 Accounting Client

The high level architecture of the AccountingClient has been described in Section 3.4. A key parameter of the AccountingClient design is the internal Spring [17] ApplicationContext that is held by the client. This ApplicationContext contains all the instantiated beans that make up the event processing pipeline. The AccountingClient initializes the ApplicationContext using the configuration files provided during the construction time. The files are monitored for changes and the ApplicationContext is reloaded in case of a change. When the ApplicationContext is reloaded a Sink with the name given during construction is looked up and used as the recipient of Events given to the AccountingClient. Thus, a new pipeline is configured and will be used for any future Events. Any unprocessed Events during the ApplicationContext reloading procedure, will be processed by the old pipeline. It is important for the developer to know that only three parallel pipelines will remain active in order to ensure performance of AMAAIS. If there are already three active pipelines, any unprocessed events in the oldest one will be lost when a new configuration reloading occurs. The configuration file reloading period is configurable during the AccountingClient initialization but it has to be larger than three seconds.

4.2 Accounting Server

As described in Section 3.5 the AccountingServer is an HttpServlet implementation. As only the POST binding for accounting message is defined, only the doPost method is implemented. In contrast to the AccountingClient, the Spring ApplicationContext is being loaded by the Servlet container. The configuration files to use are given as Servlet initialization parameters. Additionally, the incoming Security Assertion Markup Language (SAML) messages are appropriately validated according to the accounting message profile. This entails checking for replayed messages, validating signatures (if the message is signed), decrypting the message (if the message is encrypted), and checking the certificate used to establish the mutually-authenticated TLS connection, or for generating the signature against metadata in order to establish the trustworthiness of the AccountingClient.

The Server has three Status response codes and can provide a Reason for the respective code. The three Status codes are ACCEPTED, REJECTED, and ERROR, the Reason can be an arbitrary text detailing the Status. ACCEPTED means that the message passed all the

checks done by the server and the Events are further processed. REJECTED means that a certain seucrity check performed by the server failed and the whole message was discarded. ERROR means that something unexpected happened and therefore the message could not be processed. An ACCEPTED Status does not give any Reason since everything worked as expected. The REJECTED Status gives a Reason why the message was rejected, which can be a signature issue or a replayed message. The Error Status details the cause of the error by writing the Exception's message in the Reason.

4.3 Channels

There are three different Channels in the AMAAIS system. Each Channel can control the Event release rules, and/or the number, the type, and the order of the Event release Sinks. If a combination of each Channel's functionality is needed, a sequential selection of the corresponding Channels should be configured in the pipeline.

4.3.1 DirectChannel

The DirectChannel, which is illustrated in Figure 10, is a Channel that simply forwards the received events to the associated Sink. The use of this channel s not strictly necessary since the Source producing the Events can be wired directly to the Sink consuming the Events. However, the use of this channel is a clear statement that this is the intent and no further action, such as broadcast Events is needed. Furthermore, the implementation of the DirectChannel preserves the common model of Source and Sink connected via a Channel.

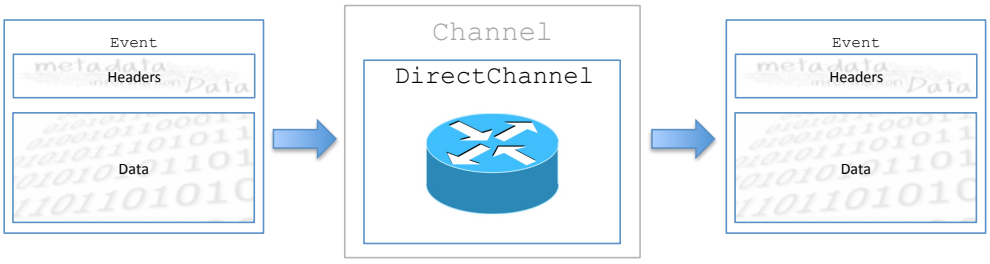

Figure 10: AMAAIS DirectChannel.

4.3.2 BroadcastChannel

The BroadcastChannel is a Channel that sends each received Event to one or more Sinks. If isCloningEvents is true and more than one Sink is registered, this channel will clone the Events it receives prior to pushing them to the associated Sinks. When an Event is cloned its payload remains the same while the ID_HEADER and the CREATION_INSTANT_HEADER are changed. In Listing 1 is shown how to enable, or disable the Events cloning functionality of the BroadcastChannel.

Listing 1: `BroadcastChannel` Cloning Configuration.

```
<bean id="broadcastchannel" class="ch.SWITCH.amaais.core.channels.
   BroadcastChannel">
       <constructor-arg>
              <list>
                     <!-- The list of Sinks goes here -->
                     <ref bean="sink1" />
                     ...
              </list>
       </constructor-arg>
       <!--If Events cloning is needed before broadcasting, then the
          following value should be set to 'true', in other case it should
          be set to 'false'-->
       <constructor-arg type="boolean" value="true"/>
</bean>
```

If `Events` are not cloned, great care must be taken while defining the pipeline in the Spring configuration files, so that subsequent processing steps do not interfere with each other. An illustration of the `BroadcastChannel` can be found in Figure 11.

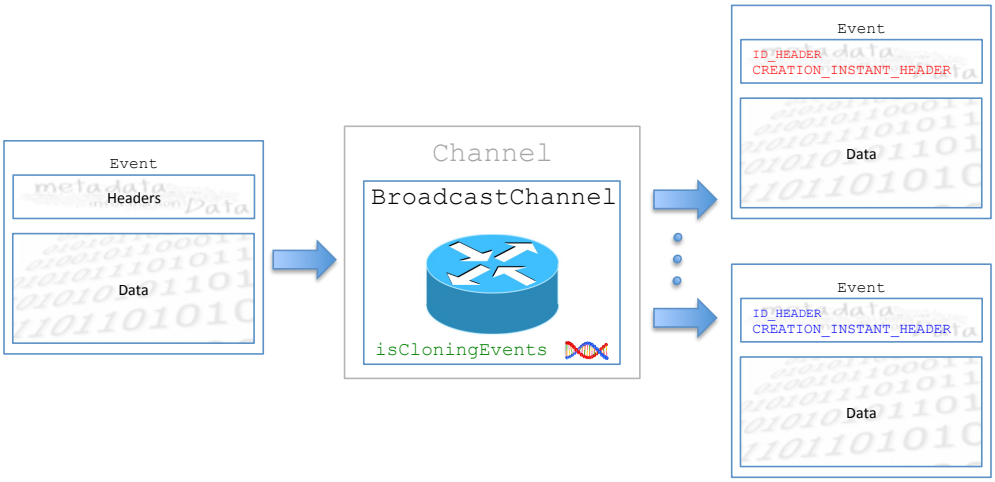

Figure 11: AMAAIS `BroadcastChannel`.

4.3.3 QueueChannel

The `QueueChannel` is illustrated in Figure 12. It is a `Channel` implementation that buffers `Events` until a given criterion is met, at which point all queued events are released to the associated `Sink`. Note, if the release criterion is based on queue size (*e.g.*, release the buffered events when there are 50 `Events`) the number of events received by the associated `Sink` may be greater than the triggering size as events may have arrived between when the criterion was checked and when the events were actually drained from the queue. However, one might think that it is not safe to wait for the queue size to be

reached before releasing the `Events` in the queue. Thus, it is mandatory to configure the `releasePeriod` parameter. When the first release criterion is met all the `Events` that are currently in the `QueueChannel` will be released.

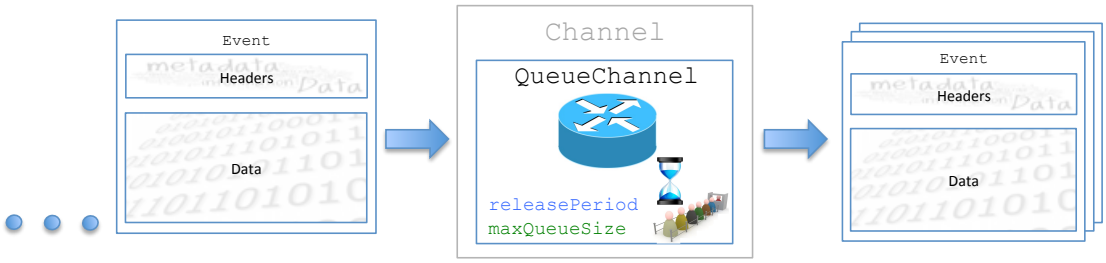

Figure 12: AMAAIS `QueueChannel`.

4.4 Stages

This section explains the responsibilities of the different `Stages` available in AMAAIS, such as sending `PublishEventRequests` to an accounting server or persisting `Events` in a database. Furthermore, example Spring configuration snippets are presented.

4.4.1 FilterAttributes

`FilterAttributes` is a `SourceStage` that accepts `Events` whose payload is a collection of `Attributes` and runs them through an `AttributeFilteringEngine` in order to filter attributes that should not be released. Reasons for not releasing an attribute exist manyfold, one example is privacy.

The Spring configuration of the `FilterAttributes Stage`, shown in Listing 2, is simple. Only two constructor arguments are needed, a `FilterEngine` and a `Sink` to which `Events` are passed on.

Listing 2: Example Spring Configuration for the `FilterAttributes Stage`.

```
<bean id="exampleFilterStage" class="ch.SWITCH.amaais.core.stages.
   FilterAttributes">
  <constructor-arg ref="exampleEngine" />
  <constructor-arg ref="theSink"/>
</bean>
```

The actual filtering configuration is done in the `FilterEngine` configuration, shown in Listing 3. A `FilterEngine` consists of several `AttributeFilterPolicy` beans. `AttributeFilterPolicy` consists of several `AttributeValueFilterPolicy` beans. By default, an attribute is filtered if there is no rule allowing its release. This implies that for each attribute which is going to be allowed through the filter there must be an `AttributeValueFilterPolicy`.

The AttributeValueFilterPolicy has two properties, which need to be set. The first property is the AttributeID and the second is the ValueMatcher. An AttributeID determines to which attributes this policy applies. A ValueMatcher determines to which Values this policy applies. AttributeIDs must match the attribute ID in the respective event. An example ValueMatcher is the allMatcher shown in Listing 3—allMatcher matches any Value. There are more sophisticated matchers such as ValueMatcher which can use regular expressions. See the Shibboleth documentation for details [7].

The AttributeFilterPolicy takes an ID, a Predicate (Google Guava), and a set of AttributeValueFilterPolicy as constructor arguments. The ID is just used as an identifier (used in logging) and can be chosen arbitrarily. The Predicate determines if a matched AttributeValue is released or not. If it evaluates to true it is released. The Predicate API [1] lists some further factory methods to create a Predicate; relevant ones are alwaysTrue and alwaysFalse.

The AttributeFilteringEngine configuration is simple. It needs an identifier and a set of AttributeFilterPolicy beans. The identifier is mainly used for logging and has no influence on the filtering process. The AttributeFilterPolicys have already been defined before and can be added as required.

Listing 3: Example Spring Configuration for Attribute Filtering.

```
<bean id="allMatcher" class="org.springframework.beans.factory.config.
   FieldRetrievingFactoryBean">
  <property name="staticField" value="net.shibboleth.idp.attribute.filtering
     .AttributeValueMatcher.MATCHES_ALL" />
</bean>

<bean id="exampleValuePolicy" class="net.shibboleth.idp.attribute.filtering.
   AttributeValueFilterPolicy">
  <property name="attributeId" value="exampleAttribute1" />
  <property name="ValueMatcher" ref="allMatcher" />
</bean>

<bean id="examplePolicy" class="net.shibboleth.idp.attribute.filtering.
   AttributeFilterPolicy">
  <constructor-arg type="String" value="exampleID" />
  <constructor-arg>
    <bean class="com.google.common.base.Predicates" factory-method="
       alwaysTrue" />
  </constructor-arg>
  <constructor-arg>
    <list>
      <ref bean="exampleValuePolicy" />
    </list>
  </constructor-arg>
</bean>

<bean id="exampleEngine" class="net.shibboleth.idp.attribute.filtering.
   AttributeFilteringEngine">
```

```
  <constructor-arg type="String" value="exampleEngine"/>
  <constructor-arg>
    <list>
          <ref bean="examplePolicy" />
        </list>
  </constructor-arg>
</bean>
```

4.4.2 ResolveAttributes

`ResolveAttributes` is a `SourceStage` that associates `AttributeEncoders` with all `Attributes`. `Encoders` are required for converting `Attributes` into XML. `Encoders` carry the protocol-specific name of the `Attribute` and the logic necessary to convert the Java objects that are the values of the Attributes in to the proper XML values (*e.g.*, Base64 encoding `byte[]` values and marking the resulting `<AttributeValue>` XML element as something that carries a Base64 encoded blob).

Listing 4 shows the configuration of a `ResolveAttribute Stage`. Two constructor arguments are used, where the first is an identifier and the second is a list of `AttributedDefinition` beans.

The `GenericAttributeDefinition` defined in Listing 4 can be used for every `Attribute`. The first constructor argument is the `AttributeId` which has to correspond to the name of an actual `Attribute`. There must be an `Encoder` defined for every `Attribute`. The second argument is the actual `Encoder`. At the moment only `XSStringEncoders` are supported. Therefore, this example configuration can be used for all `Attributes`.

Listing 4: `ResolveAttributes` Configuration.

```
<bean id="exampleAttributeDefinition" class="ch.SWITCH.amaais.common.
    attribute.definition.GenericAttributeDefinition">
  <constructor-arg type="java.lang.String" value="exampleAttribute2" />
  <constructor-arg>
    <list>
      <bean id="XSStringEncoder" class="net.shibboleth.idp.saml.impl.
          attribute.encoding.Saml2StringAttributeEncoder"/>
    </list>
  </constructor-arg>
</bean>

<bean id="exampleResolver" class=" net.shibboleth.idp.attribute.resolver.
    AttributeResolver">
  <constructor-arg type="java.lang.String" value="foo" />
  <constructor-arg>
    <!-- List of attribute definitions used for resolving attributes -->
    <list value-type="net.shibboleth.idp.attribute.resolver.
        BaseAttributeDefinition">
      <ref bean="exampleAttributeDefinition" />
```

```
    </list>
  </constructor-arg>
  <constructor-arg>
    <null />
  </constructor-arg>
</bean>
```

Additionally, it can be extended to lookup additional `Attribute`s and transform existing ones. This feature works like it works in the Shibboleth IdP, from where most code can be reused. Furthermore, the available `Encoders` can be extended. Shibboleth currently offers XML and binary data encoders which can be integrated if requirements arise.

4.4.3 GenerateAttributeStatement

`GenerateAttributeStatement` is a `SourceStage` that generates an `AttributeStatement` containing all the attributes from a given `Event`. `Event`s are transformed into SAML (version 2) `Assertion`s, wherefore the set of `Attribute`s in an `Event` corresponds to an `AttributeStatement`. The `AttributeStatement` is attached to the header data of the respective `Event`. The only configuration parameter is the `Sink` to which `Event`s are forwarded. This will usually be a `TransmitSamlMessage Stage`. The configuration is shown in Listing 5.

Listing 5: `GenerateAttributeStatement` Configuration.

```
<bean id="generateAttributeStage" class="ch.SWITCH.amaais.core.stages.
   GenerateAttributeStatement">
    <constructor-arg ref="TransmitSAML" />
</bean>
```

4.4.4 PersistEventsToDatabase

The `PersistEventsToDatabase Stage` is a `Sink` only and therefore it does not output `Event`s. The `Stage` is responsible for storing `Events` into a database. Multiple event types are supported by a single `PersistEventsToDatabase Stage`. Configuring the `Stage` in Spring is straight forward as it takes a Hibernate `SessionFactory` and a `Map` as constructor arguments, where the `Map` represents the mapping from event type to the corresponding `DAOFactory`.

Listing 6 shows an example Spring configuration of a `PersistEventsToDatabase Stage`. It shows the two constructor arguments, the first is a reference to a Hibernate `SessionFactory` bean. The map argument can be extended, what means that another key-value pair can be added to support a custom `Event Type`.

Listing 6: Example Spring Configuration for the `PersistEventsToDatabase Stage`.

```
<bean id="localDevPersistStage" class="ch.SWITCH.amaais.core.stages.
   PersistEventsToDatabase">
```

```
 <constructor-arg ref="localDevSessionFactory"/>
 <constructor-arg>
  <map>
   <entry key="DefaultEvent" value-ref="defaultEventDAOFactoryImpl" />
  </map>
 </constructor-arg>
</bean>
```

Listing 7 is an example of the Hibernate-related Spring configuration. The `SessionFactory` used in Listing 6 is defined in Listing 7. First, a `DataSource` needs to be defined, which is then used in the `SessionFactory` bean. The `DataSource` has all the relevant information about the backing database. Special care has to be taken on timeout settings, since they can cause problems when not matched to the database system. Important for custom event types is the `PackagesToScan` property of the `SessionFactory`: it defines the packages being scanned by Hibernate for DAOs. In case of a custom DAO to be used, it is necessary to add the containing package to the list.

Listing 7: Example Hibernate Spring Configuration.

```
<bean id="localDevDataBaseSource" class="com.mchange.v2.c3p0.
   ComboPooledDataSource" destroy-method="close">
 <property name="driverClass" value="com.mysql.jdbc.Driver"/>
 <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/amaais"/>
 <property name="user" value="user_name"/>
 <property name="password" value="passwd"/>
 <!-- these are C3P0 properties -->
 <property name="acquireIncrement" value="1" />
 <property name="minPoolSize" value="1" />
 <property name="maxPoolSize" value="25" />
 <property name="idleConnectionTestPeriod" value="300"/>
 <property name="checkoutTimeout" value="120"/>
</bean>

<bean id="localDevSessionFactory" class="org.springframework.orm.hibernate4.
   LocalSessionFactoryBean">
 <property name="dataSource" ref="localDevDataBaseSource"/>
 <property name="PackagesToScan">
  <list>
   <!-- add DAO implementation location here -->
   <value>ch.SWITCH.amaais.core.persistency</value>
  </list>
 </property>
 <property name="hibernateProperties">
  <props>
   <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
   <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
  </props>
 </property>
```

```
</bean>
```

4.4.5 TransmitSamlMessage

TransmitSamlMessage is a Stage that fetches the AttributeStatement from each event header, generates an Assertion containing all AttributeStatements, generates a PublishEventRecordRequest, POST-encodes the request, and sends it to the AccountingServer. This Stage is typically the end of an AccountingClient pipeline, since it sends Events to a server. Listing 8 shows the TransmitSamlMessage Stage configuration. The many constructor arguments are explained in the same order as they appear in Listing 8, additionally the issuerURL property is also explained:

RelyingPartyId This is the entityID of the EntityDescriptor for an AccountingServer found in the AMAAIS metadata.

MetadataProvider An OpenSAML metadata provider. A sample configuration is presented in Listing 9.

HttpClient This argument is there to make use of different HTTPClient implementations easy.

Credential Credential is used for signing Assertions in messages. A sample configuration is shown in Listing 10.

NumberOfAttempts This is the number of attempts the Stage takes to send a POST request, before failing.

IssuerURL This is the entityID of the EntityDescriptor for the sending AccountingClient found in the AMAAIS metadata.

Listing 8: TransmitSamlMessage Configuration.

```
<bean id="exampleTransmitSAML" class="ch.SWITCH.amaais.core.stages.
   TransmitSamlMessage">
  <constructor-arg type="String" value="exampleConsumer"/>
  <constructor-arg ref="exampleMetaDataProvider"/>
  <constructor-arg>
    <bean class="org.apache.commons.httpclient.HttpClient"/>
  </constructor-arg>
  <constructor-arg ref="credential"/>
  <constructor-arg type="int" value="3"/>
  <property name="issuerURL" value="exampleProducer"/>
</bean>
```

Listing 9 shows a basic MetadataProvider configuration. The one used here is file system-based. OpenSAML offers other types of MetadataProviders in the package org.opensaml.saml2.metadata.provider.

Listing 9: `MetadataProvider` Configuration.

```
<bean id="exampleMetaDataProvider" class="org.opensaml.saml.saml2.metadata.
   provider.FilesystemMetadataProvider" init-method="initialize">
  <constructor-arg>
    <bean class="java.io.File">
      <constructor-arg type="String" value="src/test/resources/examples/
         metadata.xml" />
    </bean>
  </constructor-arg>
  <property name="parserPool">
    <bean class="net.shibboleth.utilities.java.support.xml.BasicParserPool"
       init-method="initialize"/>
  </property>
</bean>
```

Listing 10 shows how a `Credential` bean can be defined in Spring. AMAAIS uses X509 credentials. A Java Key Store (JKS) is required for the creation of the bean. Therefore a convenience method to load such a JKS was created in the AMAAIS core. Arguments for creating the JKS bean are path to file, username, and password.

Listing 10: `Credential` Configuration.

```
<bean id="credential" class="ch.SWITCH.amaais.common.utils.KeystoreSupport"
   factory-method="getX509Credentials">
  <constructor-arg type="java.lang.String" value="/path/to/keystore.jks"/>
  <constructor-arg type="java.lang.String" value="username"/>
  <constructor-arg type="java.lang.String" value="password"/>
</bean>
```

4.5 Persistency

By default each `Event` is converted to a `Map` whose ID is the ID of the `Attribute` and whose value is the value of the `Attribute`. If the `Attribute` contains more than one value then the values are concatenated with a `valueSeparator` between each individual value.

This default design had to be extended during the final AMAAIS project phase, because the default storage would not allow to do advanced queries, such as queries based on number ranges. Therefore, a two component design was developed, which consists of a Data Access Object (DAO) and an according DAO Factory. The DAO is a concept which uses a simple Java object to represent a data structure. Java Persistency API (JPA) annotations are used to define data types and database parameters. The DAO Factory's responsibility is to create a DAO from an AMAAIS `Event`. It takes an `Event` and returns a DAO.

4.5.1 Persistency Design

Figure 13 depicts a class diagram of the persistency base design. The `PersistEventsToDatabase Stage` uses one to many abstract classes called `BaseEventDAOFactory`. The `Stage` maps event types to factories and uses the `create(...)` method to convert an `Event` to the corresponding DAO of type `BaseEventDAO`. `BaseEventDAO` contains the `Event` header data, while `BaseEventDAOFactory` sets these values. Thus, an extension of a DAO and a factory does not need to care about header data.

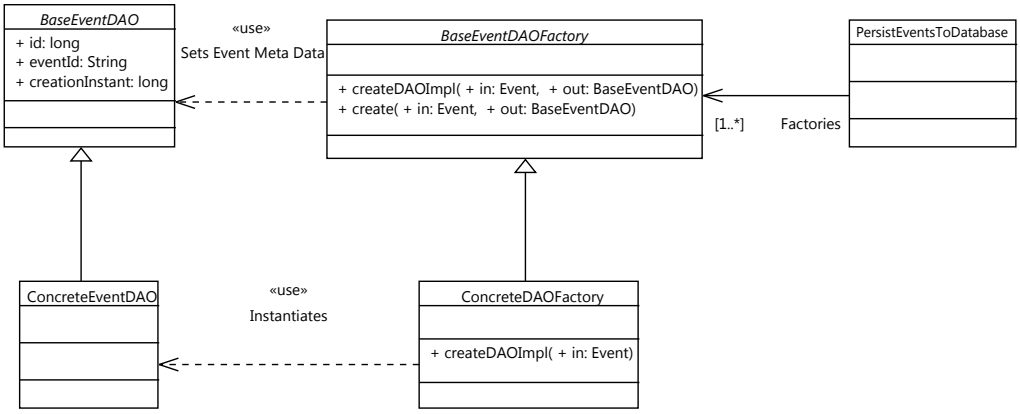

Figure 13: Class Diagram Showing the AMAAIS Default `DAO` Design.

`ConcreteEventDAO` and `ConcreteEventDAOFactory` are placeholders for an actual DAO implementation. One example of such an implementation is the `DefaultEventDAO` and `DefaultEventDAOFactory` which are explained in more detail in Sections 4.5.3 and 4.5.4.

4.5.2 Persistency Implementation

The implementation of the two abstract base classes introduced in the previous persistency design section is shortly described here, since their implementation is relevant for the development of custom extensions.

Listing 11: Implementation of the `BaseEventDAO` class.

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class BaseEventDAO {
  @Id
  @GeneratedValue(strategy=GenerationType.TABLE)
  @Column(name="ORM_ID")
  private Long id;

  @Column(name="EVENT_ID")
  private String eventId;
```

```
  @Column(name="CREATION_INSTANT")
  private Long creationInstant;
...
```

Listing 11 shows an excerpt of the `BaseEventDAO` class. Geter and seter methods are omitted but exist for each `Attribute`. Annotations are used from the JPA. For maximum compatibility no Hibernate-specific dependencies exist in either the DAOs or the factories. A DAO is a simple class without logic—by means of annotations, it becomes a mapping between Java and a database (the Object Relational Mapping, ORM).

Listing 12: Implementation of the `BaseEventDAOFactory` class.

```
public abstract class BaseEventDAOFactory {

  protected abstract BaseEventDAO createDAOImpl(Event<Set<Attribute>> event)
      ;

  public final BaseEventDAO create(Event<Set<Attribute>> event){
    BaseEventDAO baseDAO = this.createDAOImpl(event);
        baseDAO.setEventId(event.getHeaders().get(event.ID_HEADER).toString()
            );
        baseDAO.setCreationInstant(Long.parseLong(event.getHeaders().get(
            event.CREATION_INSTANT_HEADER).toString()));
        return baseDAO;
  }
}
```

Listing 12 shows the base implementation of the `BaseEventDAOFactory`, which has two methods. The first is abstract and acts as a template methode for subclasses to implement. Subclasses must implement this method in a way that it sets all the properties of `ConcreteBaseEventDAO`. The second method uses the first one to create the DAO; that way, the subclass controls the type of the DAO. The `create(...)` method then sets all the header properties defined in `BaseEventDAO` before returning it to the caller.

4.5.3 DefaultEventDAO

The default persistency implementation works out of the box and stores `Event` header data and any attribute and value. All `Attribute`s are converted into a map which uses the `Attribute` name as key for the concatenated `AttributeValue`s. If an attribute has multiple values, they are concatenated with a specified delimiter. No escaping is done in case the delimiter appears in the values therefore, a developer or user has to make sure that this won't happen.

Figure 14 shows the AMAAIS default database schema, which is a direct consequence of `DefaultEventDAO`'s Hibernate annotations. An `Event` is mapped to several `Attribute`s which use the `Event`'s primary key as a foreign key. This layout is simple to implement and can store arbitrary event types. However, when generating reports, this structure has significant drawbacks. Since `Attribute`s are all stored as the same data type, which is

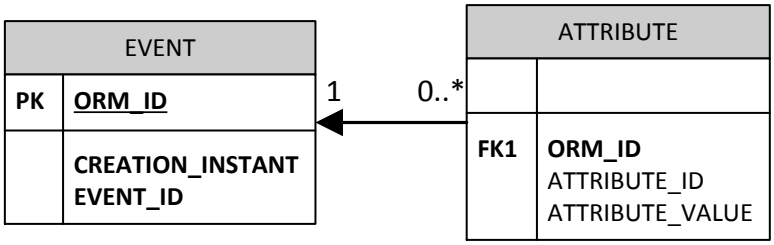

Figure 14: Default Database Schema.

`String`, possibilities for SQL queries are very limited. It is not possible to select `Events` based on a number or date range in an `AttributeValue`.

4.5.4 DefaultEventDAOFactory

`DefaultEventDAOFactory` performs the conversion from `Attributes` and `Values` to a `Map`. The `Map` is of type `<String, String>` where the key is the `Attribute` name and the value is the concatenated `AttributeValues` of the corresponding `Attribute`.

Listing 13: Implementation of the `DefaultEventDAOFactory` class.

```
<bean id="defaultEventDAOFactoryImpl" class="ch.SWITCH.amaais.core.
   persistency.DefaultEventDAOFactoryImpl">
  <constructor-arg value="[:]" />
</bean>
```

Listing 13 shows the Spring configuration of the `DefaultEventDAOFactory`. The configuration is straight forward, since the only constructor argument is the delimiter used to separate multiple `AttributeValues`.

4.6 Security

This section presents the AMAAIS security measurements in order to (1) transfer accounting events in a secure manner, and (2) avoid that accounting events become persisted twice in the `AccountingServer` database.

4.6.1 SSL Communication

In order to transfer the generated accounting events through the network ensuring that no intermediary malicious entity is capturing the traffic, the AMAAIS Accounting Protocol is encapsulated within an SSL communication. The SSL communication is established between a `TransmitSAMLMessage Stage` (which is typically used on the `AccountingClient` side) and an `AccountingServer`. The SSL communication is based on certificates, and the AMAAIS implementation enforces mutual client/server authentication. Mutual client/server

authentication means that both `AccountingClient` and `AccountingServer` need to have the respective certificates to communicate and accept the exchange of accounting events.

Therefore, the SSL connection has two roles: (1) authentication and (2) authorization. Authentication in the sense that the process of exchanging certificates ensures the identity of such entities, and authorization in the sense that the presence of valid certificates grants permission to entities to exchange accounting events.

Listing 14 illustrates how an SSL communication can be established between a `TransmitSAMLMessage Stage` and an `AccountingServer` running on Tomcat6. These steps are based on a Ubuntu 11.04 Linux distribution and Apache Tomcat6.

Step 1: Change "/etc/tomcat6/server.xml" to include:

Listing 14: Configuration of "/etc/tomcat6/server.xml".

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
             maxThreads="150" scheme="https" secure="true"
             clientAuth="true" sslProtocol="TLS"
             keystoreFile="server-keystore.jks" keystorePass="importkey"
             truststoreFile="server-truststore.jks" truststorePass="importkey"
/>
```

The "server-keystore.jks" file is a key store containing the public key and the private key of an `AccountingServer`. The "server-truststore.jks" file is a key store only containing the public key of the `AccountingClient`s that this accounting server trusts. In order to build a "server-truststore.jks", it is possible to use the "keytool" following the steps found in Section 4.6.2 and in the AMAAIS code[1].

Step 2: As an optional step, it is possible to set Apache Tomcat6 to redirect all HTTP connections to use SSL communication. This allows a higher security, but bringing the disadvantage that an `AccountingServer` needs to exchange keys with all `AccountingClient`s (or, `TransmitSAMLMessage Stage`s) that it communicates with. Listing 15 shows how to enable such "redirection" by including the "redirectPort" attribute in "/etc/tomcat6/server.xml", within the "connector" element that refers to normal HTTP connections.

Listing 15: SSL Redirection Configuration.

```
<Connector port="8080" protocol="HTTP/1.1"
             connectionTimeout="20000"
             URIEncoding="UTF-8"
             redirectPort="8443
/>
```

Also, add at the end of "/etc/tomcat6/web.xml" file, before the closing "</web-app>" element, what is shown in Listing 16.

Listing 16: Configuration of "/etc/tomcat6/web.xml".

```
<security-constraint>
       <web-resource-collection>
              <web-resource-name>Everything is HTTPS</web-resource-name>
              <url-pattern>/*</url-pattern>
       </web-resource-collection>
```

[1] `AuthSSLProtocolSocketFactory` class

```
        <user-data-constraint>
                <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
</security-constraint>
```

Step 3: Restart Apache Tomcat6, which can be executed in Ubuntu Linux 11.04 by means of "sudo /etc/init.d/tomcat6 restart".

Step 4: Deploy the `AccountingServer` WAR in Apache Tomcat6. Important: make sure that the Apache Tomcat6 server is using Java 7, otherwise the `AccountingServer` will not be deployed successfully.

Step 5: The configuration of the `TransmitSAMLMessage Stage` should present the instance of an `SSLKeysManagement` bean, setting the key store and the trust store file location and password. Listing 17 shows how to instantiate such configuration bean.

Listing 17: `SSLKeysManagement` Configuration.

```
<bean id="sslKeysManagementObj" class="ch.SWITCH.amaais.common.ssl.SSLKeysManagement">
        <property name="keyStorePath" value="file:///path/to/client-keystore.jks" />
        <property name="keyStorePassword" value="importkey" />
        <property name="trustStorePath" value="file:///path/to/client-truststore.jks" />
        <property name="trustStorePassword" value="importkey" />
</bean>
```

Moreover, within the "<TransmitSAMLMessage>" bean declaration, the property shown in Listing 18 should be added — referencing the `SSLKeysManagement` bean object instantiated in Listing 17. This Spring property performs a method call to enable SSL key management in `TransmitSAMLMessage`.

Listing 18: Configuration in the `TransmitSAMLMessage Stage`.

```
<bean id="exampleTransmitSAML" class="ch.SWITCH.amaais.core.stages.TransmitSamlMessage">
        ............
        ............
        <property name="SSLKeysManagement">
            <ref bean="sslKeysManagementObj" />
        </property>
</bean>
```

For `TransmitSAMLMessage`, the key store should contain the public key and private key of the `AccountingClient`. The trust store should contain the public key of the `AccountingServer`s that the `AccountingClient` communicates to. `AccountingServer` information can be acquired from the respective metadata. Certificate exchange needs to be handled manually in the current AMAAIS release.

It is important to note that the "keyStorePath" and the "trustStorePath" values should follow the URL standard. Therefore, if a local file is expressed, the "file://" prefix should be always used, followed by an absolute file path. For example: "file:///home/user/truststore.jks" or "file://c:/User/truststore.jks".

Step 6: The `TransmitSAMLMessage Stage` was modified to support "POST" redirects. "POST" redirection is not natively supported by Apache's `HttpClient` implementation.

However, to make the AMAAIS implementation more flexible and secure (thus, automatically accepting redirections to SSL), the “POST” method redirection was implemented and activated.

4.6.2 SSL Certificates

Use the following sequence of actions to generate a keystore, export a certificate, and import to a truststore file.

Generate a private key in keystore file:

```
keytool -genkeypair -alias certificatekey -keyalg RSA -validity 365
-keystore client-keystore.jks
```

Verifiy the created keystore file:

```
keytool -list -v -keystore keystore.jks
```

Export the certificate:

```
keytool -export -alias certificatekey -keystore client-keystore.jks -rfc
-file client-selfsignedcert.cer
```

Import the certificate in to the truststore file:

```
keytool -import -alias certificatekey -file client-selfsignedcert.cer
-keystore server-truststore.jks
```

Verify the created trust store file:

```
keytool -list -v -keystore server-truststore.jks
```

4.6.3 Generating AMAAIS Metadata Certificates

This section describes how certificates for the use with AMAAIS Metadata can be created. There are three steps in this process, namely creating the PEM files, converting them to DER format and finally importing them into a Java Key Store (JKS). Creating the .pem files can be done according to the SWITCH Service Provider configuration guide [6]. For PEM to DER conversion, `openssl` is used like this: `openssl x509 -outform der -in certificate.pem -out certificate.der`. The last step requires the Java keytool which comes with Java: `keytool -import -alias your-alias -keystore cacerts -file certificate.der`. The contents of the PEM-formatted files can be used in the `Metadata` as they appear in the files. The DER files are not needed anymore after the import, thus they should be deleted. A keystore should only be used by one `AccountingClient` or `AccountingServer` since it contains one or more private keys.

4.6.4 Replayed Messages

According to its design specification, the `AccountingServer` has the possibility to check for replayed messages. While this threat is handled by SSL connections, the threat remains with unencrypted setups.

For checking replayed messages, the `AccountingClient` integrates a random number into a message and then signs the whole message. The use of a random number ensures that it is almost impossible to have any false positives, meaning that a message is discarded albeit it was correct. For this to happen, the very unlikely case has to occur in which two messages have the exact same content and the exact same random number.

Upon receiving a message, the server can check if it received the same message signature already. This is done by putting the signature into a `HashSet`. If a signature is already in an existing set, the message is considered as replayed. Of course, a server can not keep hashes of every single message it ever received. The limiting factor in this context is memory consumption of the `HashSet`. Depending on server hardware the maximum value of signatures stored can be configured in the server configuration.

Listing 19 depicts the relevant property "maximumMessagesChecked" that can be set in the `AccountingServerServlet` configuration.

Listing 19: Maximum Replayed Messages Check Configuration.

```
<bean name="acct-server" id="acct-server" class="ch.SWITCH.amaais.server.
    AccountingServerServlet">
  <property name="eventSink"><ref bean="someStage"/>
  <property name="maximumMessagesChecked" value="1000"/>
</bean>
```

5 Deployment Guideline

The AMAAIS documentation, source code and libraries are currently available at http://www.csg.uzh.ch/research/amaais.html. In the near future, the same information will be moved to http://www.switch.ch/aai/amaais.

The deployment of AMAAIS requires in general the implementation of a specific Java accounting application, its installation on a server and the configuration of the AMAAIS `AccountingClient` and `AccountingServer` components. In the scope of AMAAIS, three accounting applications have been developed. They are described in detail in Sections 6.1.1 and 6.2.1.

5.1 Pre-requirements

The application has to be able to generate `Event`s. The underlying data can be obtained from any source, usually from a log file or database. The AMAAIS core runs on Java version 7 only. In fact, AMAAIS is dependent on the Shibboleth identity provider version 3 which requires Java 7. On the server side, one generally would want a database. Through the use of Hibernate as a persistency layer, there are many database distributions that can be used. Since the AMAAIS server is a `HttpServlet`, a container is necessary. This container needs to be able to load the Spring context. AMAAIS was tested with Tomcat, alternative containers are not officially supported. Also one needs to make sure that no firewall is blocking traffic on port 8080 between the AMAAIS `AccountingClient` and `AccountingServer`. However, the ports could be changed if 8080 is already used or not available for some reason.

5.2 Application Assembly

In order to build the application several dependencies have to be incorporated. As mentioned in the previous section, the most important dependency is the Shibboleth identity provider version 3 which is currently available as snapshot version only. Other dependencies include JUnit, commons-io and the AMAAIS software itself. An example maven dependency file can be found in Listing 20.

Listing 20: Maven Dependencies.

```
<dependencies>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>3.1.1.RELEASE</version>
                <exclusions>
                        <exclusion>
                                <artifactId>commons-logging</artifactId>
                                <groupId>commons-logging</groupId>
                        </exclusion>
                </exclusions>
        </dependency>
```

```
            <dependency>
                    <groupId>net.shibboleth.idp</groupId>
                    <artifactId>idp-attribute-api</artifactId>
                    <version>3.0-SNAPSHOT</version>
                    <scope>compile</scope>
                    <optional>false</optional>
            </dependency>
            <dependency>
                    <groupId>junit</groupId>
                    <artifactId>junit</artifactId>
                    <version>3.8.1</version>
                    <scope>test</scope>
            </dependency>
            <dependency>
                    <groupId>commons-io</groupId>
                    <artifactId>commons-io</artifactId>
                    <version>2.4</version>
            </dependency>
            <dependency>
                    <groupId>ch.SWITCH</groupId>
                    <artifactId>amaais</artifactId>
                    <version>0.0.26-SNAPSHOT</version>
                    <type>jar</type>
                    <optional>false</optional>
            </dependency>
    </dependencies>
```

The files shall be compiled as a jar file using the maven-jar-plugin and the maven-dependency plugin as shown in Listing 21.

Listing 21: Maven Plugins.

```
<plugins>
    <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                    <archive>
                            <manifest>
                                    <addClasspath>true</addClasspath>
                                    <classpathPrefix>lib/</classpathPrefix>
                                    <mainClass>put.your.main.class.here</mainClass>
                            </manifest>
                    </archive>
            </configuration>
    </plugin>
    <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>2.1</version>
            <executions>
                    <execution>
                            <id>copy-dependencies</id>
                            <phase>package</phase>
                            <goals>
                                    <goal>copy-dependencies</goal>
                            </goals>
```

```
                                <configuration>
                                        <outputDirectory>${project.build.directory}/lib</
                                            outputDirectory>
                                        <overWriteReleases>false</overWriteReleases>
                                        <overWriteSnapshots>false</overWriteSnapshots>
                                        <overWriteIfNewer>true</overWriteIfNewer>
                                </configuration>
                        </execution>
                </executions>
        </plugin>
</plugins>
```

The accounting application is linked to the AMAAIS `AccountingClient` by instantiating an object of the `AccountingClient` class and calling the `processEvents()` method. An appropriate `Event` class has to be created and passed on to this method.

5.3 Configurations

The `AccountingClient` is configured as a Spring bean. The configuration may be split in several files which are passed as a parameter to the main bean.

5.3.1 Pipeline Configurations

The entry point for a pipeline configuration at the `AccountingClient` is the `entrySink` bean which defines the object that first receives the `Event`s. All subsequent `Channel`s, `Source`s and `Sink`s are passed as constructor arguments. Every path from the `entrySink` has to end at a `Sink` that is not a `Channel` at the same time, that means that any processing does need to have a completely defined end. It is possible to create several pipeline streams, that means that an event may be pushed to several `Channel`s or `Sink`s. In the following, the parameters of each class are detailed.

A `BroadcastChannel` takes a list of `Sink`s as constructor arguments as shown in Listing 22.

Listing 22: `BroadcastChannel` Configuration.

```
<bean id="broadcastchannel" class="ch.SWITCH.amaais.core.channels.BroadcastChannel">
        <constructor-arg>
                <list>
                        <ref bean="sink1" />
                        <ref bean="sink2" />
                        <ref bean="sink3" />
                        ...
                        <ref bean="sinkn" />
                </list>
        </constructor-arg>
        <constructor-arg type="boolean" value="true"/>
</bean>
```

A `QueueChannel` takes a single `Sink` as first constructor argument. Further arguments are the type of the `List` used for queuing, the size of the queue and the timeout when events are sent even if the queue is not full. An example configuration is shown in Listing 23.

Listing 23: QueueChannel Configuration.

```
<bean id="queuechannel" class="ch.SWITCH.amaais.core.channels.QueueChannel">
        <constructor-arg ref="targetSink" />
        <constructor-arg>
                <bean id="eventQueue" class="java.util.LinkedList" />
        </constructor-arg>
        <!-- max Queue size -->
        <constructor-arg value="6" />
        <!-- max release period -->
        <constructor-arg value="60000" />
        <constructor-arg type="boolean" value="true"/>
</bean>
```

A DirectChannel takes a single Sink as constructor argument. Its configuration follows the format shown in Listing 24.

Listing 24: DirectChannel Configuration.

```
<bean id="directchannel" class="ch.SWITCH.amaais.core.channels.DirectChannel">
        <constructor-arg ref="targetSink" />
</bean>
```

5.4 Deployment Example

A straightforward configuration for the AMAAIS client would be to use a QueueChannel followed by a TramsmitSamlMessage Stage. Hence the client would just queue and sent messages without further processing.

Listing 25: Simple AccountingClient Configuration.

```
<bean id="entrySink" class="ch.SWITCH.amaais.core.channels.QueueChannel">
        <constructor-arg ref="transmitSink" />
        <constructor-arg>
                <bean id="eventQueue" class="java.util.LinkedList" />
        </constructor-arg>
        <!-- max Queue size -->
        <constructor-arg value="6" />
        <!-- max release period -->
        <constructor-arg value="60000" />
        <constructor-arg type="boolean" value="true"/>
</bean>

<bean id="exampleTransmitSAML" class="ch.SWITCH.amaais.core.stages.TransmitSamlMessage">
        <constructor-arg type="String" value="exampleConsumer"/>
        <constructor-arg ref="exampleMetaDataProvider"/>
        <constructor-arg>
                <bean class="org.apache.commons.httpclient.HttpClient"/>
        </constructor-arg>
        <constructor-arg ref="credential"/>
        <constructor-arg type="int" value="1"/>
        <property name="issuerURL" value="exampleProducer"/>
        <property name="encryptAssertions" value="false"/>
</bean>
```

On the server side, the most simple configuration would be a `QueueChannel` followed by a `PersistEventstoDatabase Sink`. The main bean here is a bean called `acct-server`. The configuration is done using properties instead of constructor arguments as shown in Listing 26. Inside the beans `PersistStage` and `LocalSessionFactoryBean`, the specific persistence classes have to be referenced.

Listing 26: Simple `AccountingServer` Configuration.

```
<bean name="acct-server" id="acct-server" class="ch.SWITCH.amaais.server.
    AccountingServerServlet">
        <property name="eventSink"><ref bean="serverQueueChannel"/></property>
        <property name="metaDataProvider"><ref bean="metaDataProvider"/></property>
        <property name="wantsAssertionSigned" value="true"/>
</bean>

<bean id="serverQueueChannel" class="ch.SWITCH.amaais.core.channels.QueueChannel">
        <constructor-arg ref="persistStage" />
        <constructor-arg>
                <bean id="eventQueue" class="java.util.LinkedList" />
        </constructor-arg>
        <!-- max Queue size -->
        <constructor-arg value="6" />
        <!-- max release period -->
        <constructor-arg value="60000" />
        <constructor-arg type="boolean" value="true"/>
</bean>

<bean id="persistStage" class="ch.SWITCH.amaais.core.stages.PersistEventsToDatabase">
        <constructor-arg ref="localDevSessionFactory"/>
        <constructor-arg>
                <map>
                        <entry key="DefaultEvent" value-ref="defaultEventDAOFactoryImpl" />
                        <entry key="YourEventType" value-ref="yourEventTypeFactoryImpl" />
                </map>
        </constructor-arg>
</bean>

<bean id="localDevSessionFactory" class="org.springframework.orm.hibernate4.
    LocalSessionFactoryBean">
        <property name="dataSource" ref="localDevDataBaseSource"/>
        <property name="PackagesToScan">
                <list>
                        <!-- add DAO implementation location here -->
                        <value>ch.SWITCH.amaais.core.persistency</value>
                        <value>your.persistency.package</value>
                </list>
        </property>
        <property name="hibernateProperties">
                <props>
                        <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</
                            prop>
                        <prop key="hibernate.hbm2ddl.auto">create</prop>
                </props>
        </property>
</bean>
```

5.5 Future Extensions

The `TransmitSamlMessage Stage` might be extended in order to notify the accounting application if needed. In more detail, the `POST` response of the `AccountingServer` might be `ACCEPTED`, `ERROR`, or `REJECTED` as described in Section Section 4.2. Thus, once the `POST` response status is received, if the response from the `AccountingServer` is either `ERROR`, or `REJECTED`, the `TransmitSamlMessage Stage` could notify the accounting application and handle the `Events` retransmission. In order to realize such solution it could be used the Java Observer [13] - Observable [12] architecture. The `TransmitSamlMessage` would then have to extend the `Observable` Java class, and the accounting application to implement the `Observer` Java `Interface`. Before the accounting application would push `Events` in the `AccountingClient` it could temporary save them. Once the `POST` response from the `AccountingSever` would be received, the `TransmitSamlMessage Stage` would invoke the `notifyObservers` method. Then the accounting application would invoke the `update` method and if needed retransmit the `Events`, or delete them from the temporary storage. Instead of temporary storage the accounting application might reparse the `Events` from their initial `Source`. An illustration this scenario is presented in Figure 15.

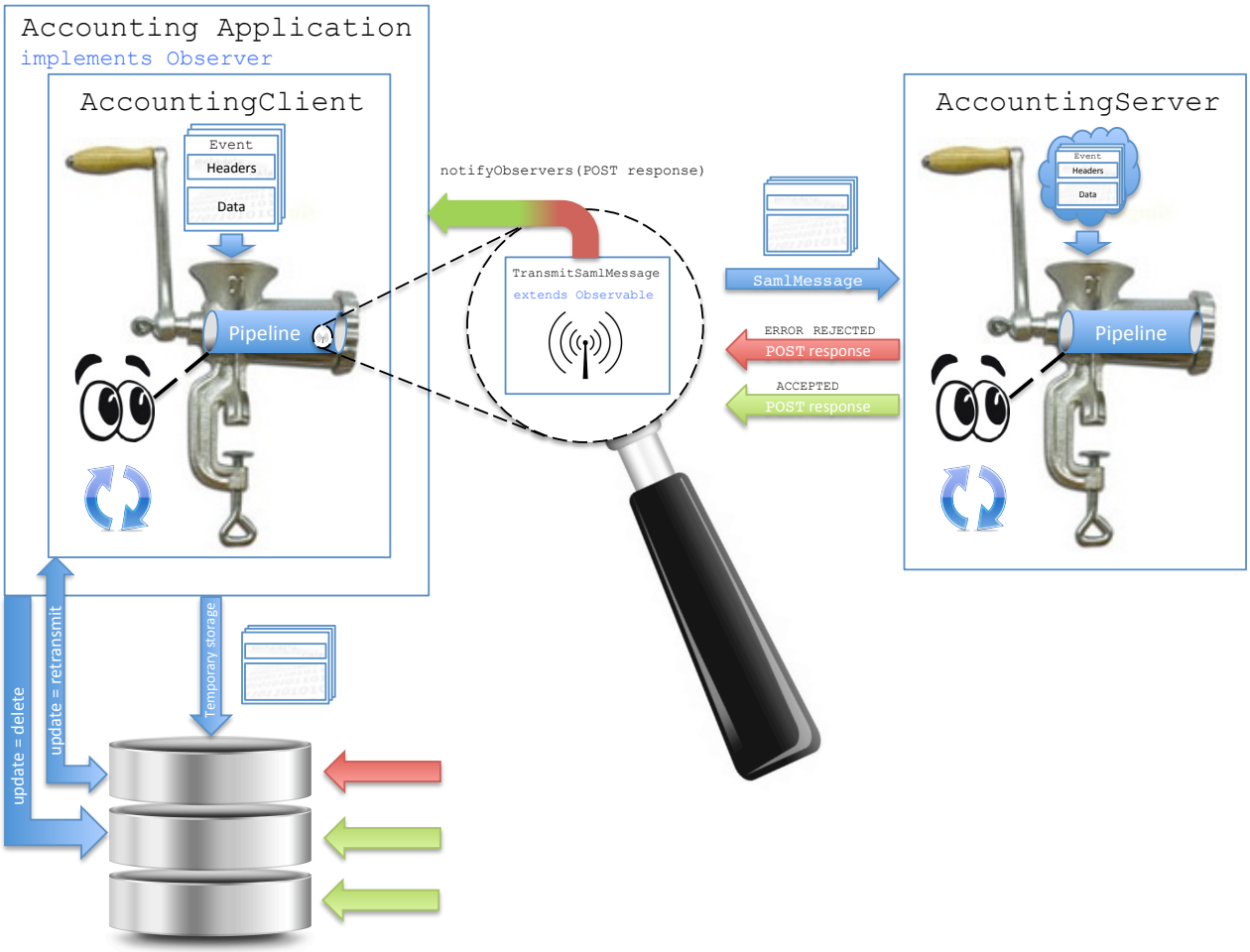

Figure 15: AMAAIS Extension.

Another future, but however important extension that the AMAAIS team recommends, is the use of a stable release of the Shibboleth IdP v3 code when it will become available. The decision to make AMAAIS compatible with the IdP v3 has been taken when there was no available release of the IdP v3. Thus during the AMAAIS development IdP v3 snapshots has been used.

6 Example Use Cases

This chapter presents the implementation of two AMAAIS accounting applications developed at ETH Zurich (Section 6.1) and one accounting application developed by SWITCH for an identity provider (Section 6.2).

The section contains a description of the implementation of the accounting application's core with the collection of the events from the source, the generation of the AMAAIS `Attribute`s and `Event`s and their transmission for each use case.

On the server side snippets of the corresponding persistency layers with the JPA DAOs for the three use cases are then described.

These examples guide the reader through several sample implementations providing a skeleton for the development of an AMAAIS accounting application.

6.1 SMS and Printing

This section presents two AMAAIS accounting applications developed at ETH Zurich to account for events on the university's distributed printing system (VPP [3]) and an SMS Web gateway [2].

The SMS web application generates an `Event` for each sent message (SMS) along with various attributes as the message length (in characters and messages) or the message destination (domestic or international).

The printing servers (VPP) generate an event for each printed job along with attributes describing for example the number of pages, the page size or if the page was black and white or colored.

The collected events are stored in both cases in a centralized logging database (Oracle for VPP and PostgreSQL for SMS). Both data structures are similar and the events are logged with a time stamp, a unique ID and the relevant attributes (*e.g.*, number of pages for the printing case or length of the message for the SMS application). As both accounting applications are basically the same a generalized example is presented in this section.

6.1.1 Accounting Application

Both the SMS and printing `Source`s (see Section 3.2) are implemented as *daemons* which check the logging databases on regular intervals to fetch new events. This is not a requirement: any solution which runs at given intervals would be equivalent (*e.g.*, standalone application run via cron or a JavaEE application using timers).

At each interval the collector performs the following operations:

- collect new events from the corresponding application logs (in this case a database)
  - read the last event time stamp from a persistent storage (*e.g.*, file system)
  - fetch all the log entries generated after the time stamp of the last run

  - for each log entry (database row) create a *payload* object containing the attributes
  - persist time stamp of the last log entry

- for each payload object the accounting application creates an `Event` with the corresponding set of `Attributes`: one for each property of the payload.
- the created `Events` are then sent to the next `Stage` with the `receive` method.

Listing 27 shows a simplified version of the main loop of the SMS source:

Listing 27: Collector.

```
package ch.ethz.app.sms;

// imports

public class ProduceSMSEvents
{

  // [...]

  @SuppressWarnings( "unchecked" )
  public void receiveSMSEvent()
  {

    // persist timing information

    // Collect the SMS data (from the logging DB)
    ArrayList< SmsPayload > smsPayloads = (ArrayList< SmsPayload >)(Object)collect();

    // Set the attributes
    for ( SmsPayload smsPayload : smsPayloads )
    {

      // Create a set of Attributes for the Event
      Set< Attribute > eventAttrs = new LinkedHashSet< Attribute >();

      eventAttrs.add( ProduceSMSEventsHelper.createAmaaisId( smsPayload.getSmsId() ) );
      eventAttrs.add( ProduceSMSEventsHelper.createUniqueIdAttribute( smsPayload.
          getSwissUniqueID() ) );
      eventAttrs.add( ProduceSMSEventsHelper.createTimestampAttribute( smsPayload.
          getTimestamp() ) );
      eventAttrs.add( ProduceSMSEventsHelper.createNumberOfSentSMSAttribute( smsPayload.
          getNumberOfSentSMS() ) );

      // other attributes

      // Create the Event specifying the client/application
      Event< Set< Attribute >> event = new Event< Set< Attribute >>( eventAttrs,
          SMS_EVENT_TYPE );

      // Forward the event to the stage...
      eventStage.receive( event );
    }
```

```
39
40   }
41
42 }
```

The different helper methods (from line 26) create an `Attribute` with a list of values (in this case always one) of type `AttributeValue` as follows (sample for the *sent SMS* attribute):

Listing 28: Collector's Helper Methods.

```
1  @SuppressWarnings( { "unchecked", "rawtypes" } )
2  public static Attribute createNumberOfSentSMSAttribute( int numberOfSentSMS )
3  {
4    Attribute attr = new Attribute( "SMS_SENT" );
5
6    Collection< AttributeValue< String >> valuesAttr = new ArrayList< AttributeValue< String
       >>();
7    valuesAttr.add( new StringAttributeValue( Integer.toString( numberOfSentSMS ) ) );
8
9    attr.setValues( (Collection< AttributeValue >)(Object)valuesAttr );
10
11   return attr;
12 }
```

Both `AccountingClients` need a `Channel` (see Section 3.3.2) to route the `Events` to the server. Listing 29 shows an instance of a `Stage` (injected by the Spring framework) where the events are sent using the `public void receive(final Event<T>... events);` method.

Listing 29: `Event` Receiver.

```
1  package ch.ethz.app.sms;
2
3  // imports
4
5  public class ReceiveSMSEvents<T> implements Stage<T>
6  {
7
8    /** The AccountingClient instance. */
9    private AccountingClient ac;
10
11   public ReceiveSMSEvents(
12     @Nonnull @NotEmpty final String sinkName,
13     @Nonnull @NonnullElements final Iterable<File> configurations,
14     @Nonnull final long period
15   ) throws IOException, InitializationException
16   {
17     ac = new AccountingClient(sinkName, configurations, period);
18   }
19
20   @SuppressWarnings("unchecked")
21   public void receive(Event<T>... events)
22   {
23     for (Event<T> event : events) {
24       ac.processEvents((Event<Set<Attribute>>) event);
25     }
```

```
  }

}
```

Listing 30 shows the application-specific Spring configuration file for the SMS application. The XML file specifies all the beans that have to be configured with the relevant values:

- The `Event` receiver (see Listing 29) with the constructor arguments (the `sinkName`, a list of `configuration` files and the `period`),
- the interval at which the main cycle is executed,
- the name of the file where the last processed event is persisted,
- other, omitted, use-case specific parameters (*e.g.*, the logging database access credentials).

Listing 30: Application-specific Configuration.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans
  xmlns = "http://www.springframework.org/schema/beans"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans http://
      www.springframework.org/schema/beans/spring-beans.xsd"
>

  <bean id="smsEventStage" class="ch.ethz.app.sms.ReceiveSMSEvents">
    <constructor-arg>
      <!-- Define the name of the bean id of the entry Sink -->
      <bean id="entrySink" class="java.lang.String">
        <constructor-arg type="String" value="entrySink" />
      </bean>
    </constructor-arg>
    <!-- Create the configuration file list -->
    <constructor-arg>
      <list>
        <!-- Create the File bean -->
        <bean id="accountingClientConfigFile" class="java.io.File">
          <constructor-arg>
            <!-- Declare the path of the configuration file -->
            <bean id="accountingClientConfigFilePath" class="java.lang.String
                ">
              <constructor-arg type="String"
                value="path/client-application.xml" />
            </bean>
          </constructor-arg>
        </bean>
```

```
      </list>
    </constructor-arg>
    <!-- Period that configuration files will be monitored -->
    <constructor-arg value="4000" />
  </bean>

  <!-- Time between two attempts to fetch an event from the DB -->
  <bean id="threadSleep" class="java.lang.Long">
    <constructor-arg type="String" value="3000" />
  </bean>

  <!-- File where the time of the last sent SMS is stored -->
  <bean id="timeLastSMSSent" class="java.lang.String">
    <constructor-arg type="String" value="path/TimeLastSMSSent.txt" />
  </bean>

  <!--
    Other application specific setting as the log database
    connection parameters
  -->

</beans>
```

6.1.2 DAO and DAO Factory

Each client must specify how the events will be persisted by the server (see Section 4.5). This is done by implementing an event DAO class extending `BaseEventDAO` (see Listing 31 for an example) and a DAO factory extending `BaseEventDAOFactory` mapping the event to the DAO object (see Listing 32).

Listing 31: Example DAO for the SMS Accounting Application.

```
package ch.ethz.app.sms.persistency;

// imports

@Entity
@Table(name="SMS_EVENT")
public class SMSEventDAOImpl extends BaseEventDAO
{

  // the unique identifier for an SMS event
  @Column(name="AMAAIS_ID")
  private Long acctSMSEventId;

  @Column(name="UNIQUE_ID")
  private String swissUniqueID;

  @Column(name="TIMESTAMP")
  private Long timestamp;
```

```

  @Column(name="SMS_SENT")
  private int numberOfSentSMS;

  // additional attributes

  // getters and setters for the property

}
```

As AMAAIS uses JPA, the DAO is a simple class annotated with `@Entity` and a field for each attribute that has to be persisted. For brevity, the example only shows some attributes namely `AMAAIS_ID`, `UNIQUE_ID`, `TIMESTAMP` and `SMS_SENT`. The corresponding getters and setters are omitted.

The corresponding sample DAO factory is shown in Listing 32.

Listing 32: DAO Factory.

```
package ch.ethz.app.sms.persistency;

// imports

public class SMSDAOFactoryImpl extends BaseEventDAOFactory
{

  @SuppressWarnings( "rawtypes" )
  @Override
  protected BaseEventDAO createDAOImpl( Event< Set< Attribute >> event )
  {

    SMSEventDAOImpl serviceDAO = new SMSEventDAOImpl();

    for ( Attribute attr : event.getPayload() )
    {

      Set< AttributeValue > attrValues = attr.getValues();

      Object value = null;
      for ( AttributeValue attributeValue : attrValues )
      {
        value = attributeValue.getValue();
      }

      switch ( attr.getId() ) {

        case "AMAAIS_ID":
        serviceDAO.setAcctSMSEventId(
          Long.parseLong((String)value)
        );
        break;

        case "UNIQUE_ID":
        serviceDAO.setSwissUniqueID( (String)value );
        break;

        case "TIMESTAMP":
```

```
39          serviceDAO.setTimestamp( Long.parseLong((String)value ));
40          break;
41
42          case "SMS_SENT":
43          serviceDAO.setNumberOfSentSMS(
44            Integer.parseInt((String)value)
45          );
46          break
47
48          // additional attributes
49
50        }
51      }
52
53      return serviceDAO;
54    }
55
56  }
```

The DAO factory simply creates a new instance of the corresponding DAO (line 13) and for each attribute in the payload (line 15) extracts the value (lines 20—24) and sets the corresponding property (from line 26).

6.2 Shibboleth Identity Provider

This section presents an AMAAIS accounting application developed at SWITCH to account logins at an SWITCHaai identity provider. Each time the user logs in to a service provider, the identity provider generates a line in the audit log with pipe separated values for various fields. As an example, there is the time of the login and the names of the attributes sent to the service provider. The audit log is in a machine readable format, that means structured using well-defined delimiters.

6.2.1 Accounting Application

The identity provider accounting application periodically parses the audit log of the identity provider and generates the corresponding `Event`s. The `Event`s are then pushed to the AMAAIS `AccountingClient`, sent to the server and finally persisted in a database. In the following, the flow of information, the class model and the configuration are detailed.

6.2.1.1 Information flow

When the application starts, it searches for a text file containing the time of the last event that was handled. Afterwards, it looks up all log files inside the specified log file directory that match “idp-audit” as prefix and ”.log” as suffix. In case, the time of the last event was found on the disk, it further restricts the selection to files that were updated after that time. Each of the remaining files is parsed line by line where each line represents an identity provider login. `Event`s that were already handled are discarded and for all others an event

is created and sent to the server. On the server side, the events are received and persisted using an appropriate DAO.

6.2.1.2 Class Model

The identity provider accounting application consists of three main classes: first, the application itself, which basically schedules the periodic execution of the `IdPParser`. The `IdPParser` contains the programming logic that loops through a set of files. Individual lines are parsed by the `LineParser` class.

6.2.1.3 Spring Configuration

The identity provider accounting application is intended to be reusable for other log-file based accounting applications. Therefore, most specific settings can be configured as Spring beans. This applies to the prefix and suffix all file names, the field names of the attributes, the delimiters (one for separating attributes and a second one for separating multiple values for the same attribute) and the path to the log files. These settings can be set inside two beans: one of class `IdPParser` and one of class `LineParser` (or a subclass of it). An example can be found in Listing 33.

Listing 33: Example Configuration.

```
<bean name="idpparser" class="ch.SWITCH.idp.IdPParser">
        <constructor-arg>
                <!-- Define the name of the bean id of the entry Sink -->
                <bean id="entrySink" class="java.lang.String">
                        <constructor-arg type="String" value="entrySink" />
                </bean>
        </constructor-arg>
        <!-- Create the configuration file's list -->
        <constructor-arg>
                <!-- Create the configuration file's list -->
                <list>
                        <!-- Create the File bean -->
                        <bean id="accountingClientConfigFile" class="java.io.File">
                                <constructor-arg type="java.lang.String" value="example-
                                    client-attribute-definition.xml"/>
                        </bean>
                        <bean id="accountingClientConfigFile" class="java.io.File">
                                <constructor-arg type="java.lang.String" value="example-
                                    client-attribute-filter-config.xml"/>
                        </bean>
                        <bean id="accountingClientConfigFile" class="java.io.File">
                                <constructor-arg type="java.lang.String" value="test-run-
                                    persistence-config-c3p0.xml"/>
                        </bean>
                        <bean id="accountingClientConfigFile" class="java.io.File">
                                <constructor-arg type="java.lang.String" value="example-
                                    client-application.xml"/>
                        </bean>
                        <bean id="accountingClientConfigFile" class="java.io.File">
```

```
                            <constructor-arg type="java.lang.String" value="example-
                                client-credentials.xml"/>
                        </bean>
                </list>
        </constructor-arg>
        <!-- Period that configuration files will be monitored -->
        <constructor-arg value="4000" />

        <!-- Reference to the LineParser-Bean -->
        <property name="lp" ref="lineparser" />
        <!-- Location of the log files, relative path -->
        <property name="directory" value="SampleIdPLogsSWITCH/" />
        <!-- Prefix of all log files to parse -->
        <property name="filenameprefix" value="idp-audit" />
        <!-- Suffix of all log files to parse -->
        <property name="filenamesuffix" value=".log" />
        <!-- Name of the file where the time of the last event handled is stored -->
        <property name="timestampfilename" value="lasttimestamp.txt" />
</bean>

<bean name="lineparser" class="ch.SWITCH.idp.LineParser">
        <!-- The delimiter between attributes, note that the | character has to be escaped
             -->
        <property name="delimiter" value="\|" />
        <!-- The delimiter between several values for the same attribute -->
               <property name="subdelimiter" value="," />
        <!-- Field names for all attributes, these have to match the identifier in the
            attribute definition and attribute filter configuration -->
         <property name="header" value="Event time,Request Binding,Request Id,Relying
             Party Id,Message Profile Id,Asserting Party Id,Response Binding,Response Id,
             Principal Name,AuthNMethod,Released Attribute Ids,Name Identifier,Assertion
             IDs" />
         <!-- Position of the date/time field inside a parsed line -->
         <property name="timestampfield" value="0" />
</bean>
```

6.2.2 DAO and DAO Factory

The data access object (DAO) encapsulates the data layer for storage in the database. The identity provider accounting application has two DAO-classes. The main class `IdPEvent` and a helper class for a multi valued attribute, the list of AAI-attributes submitted in the assertion. Most attributes are stored as `String`s with the exception of the date/time of the event.

6.2.3 Visualization

The visualization component aims at aggregating and displaying statistical information from the AMAAIS server. Several reports have been created for the identity provider

use case. For other use cases, reports can easily be designed using the eclipse BIRT framework[2].

6.2.3.1 Prerequisites

The visualization components requires the following environment on the server: It should be installed on the same machine as an AMAAIS server or on a machine where the database is duplicated. The environment for the dynamically generated reports is a Tomcat web application server. Furthermore, the eclipse BIRT runtime has to be installed. The appropriate JDBC drivers for database access complete the setup.

To view reports a computer with a standard web browser is sufficient.

6.2.3.2 Reports

Reports can be created with the BIRT report designer and deployed to the server. The report data is obtained directly from the database. Aggregations can be done on any numerical data including data derived from the count function. Data can be displayed either using its absolute value or proportions, over time or categories. In summary, the framework is very general and allows very many different types of reports. In the following some useful reports for the identity provider statistics are presented as examples.

The first report series depicts the overall usage of an identity provider over time. The first report (Figure 16) shows a bar diagram of the number of logins for each year. The user can subsequently drill-down into a year by clicking the corresponding bar to obtain a monthly view (Figure 17). Clicking on a bar in the monthly view brings up a daily view (Figure 18) for the selected month. The monthly and daily views may also be called directly. In that case, the application will ask the user to provide the parameter (*i.e.* the year or month, respectively).

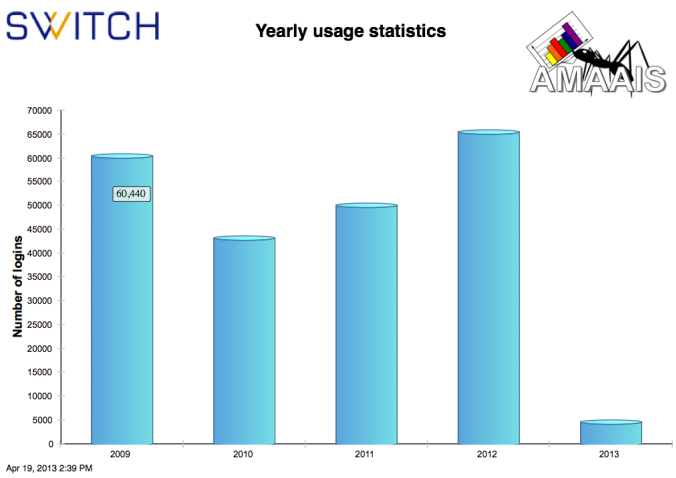

Figure 16: Yearly Usage Statistics from an Identity Provider.

[2]http://www.eclipse.org/birt/phoenix/ (last visited on April 19, 2013)

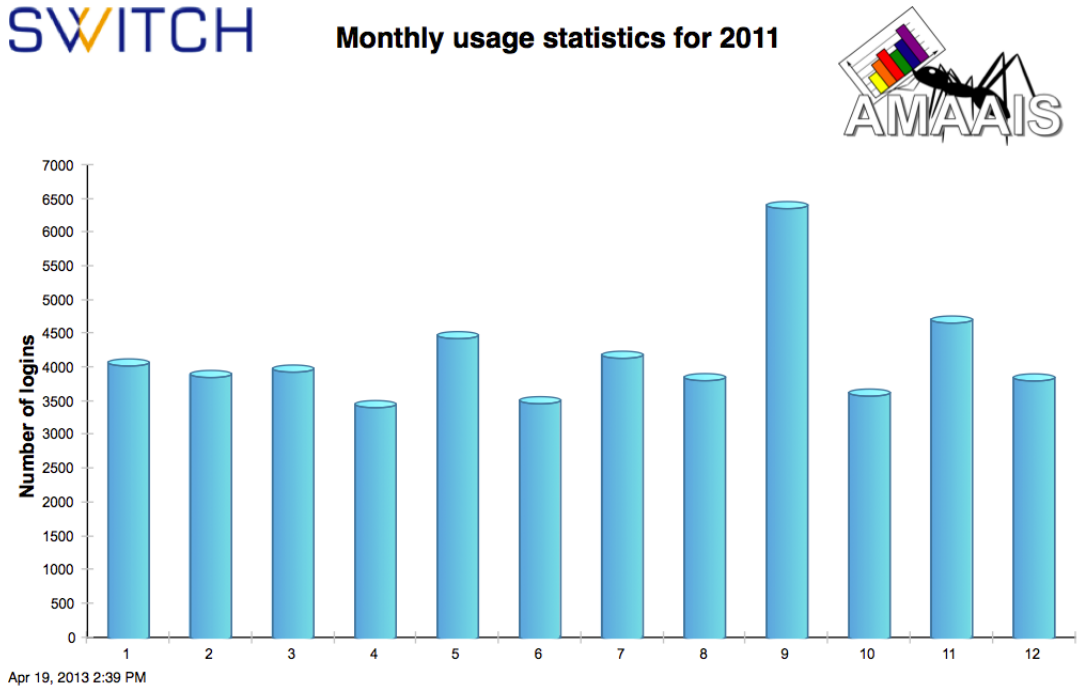

Figure 17: Monthly Usage Statistics for the Year 2011 from an Identity Provider.

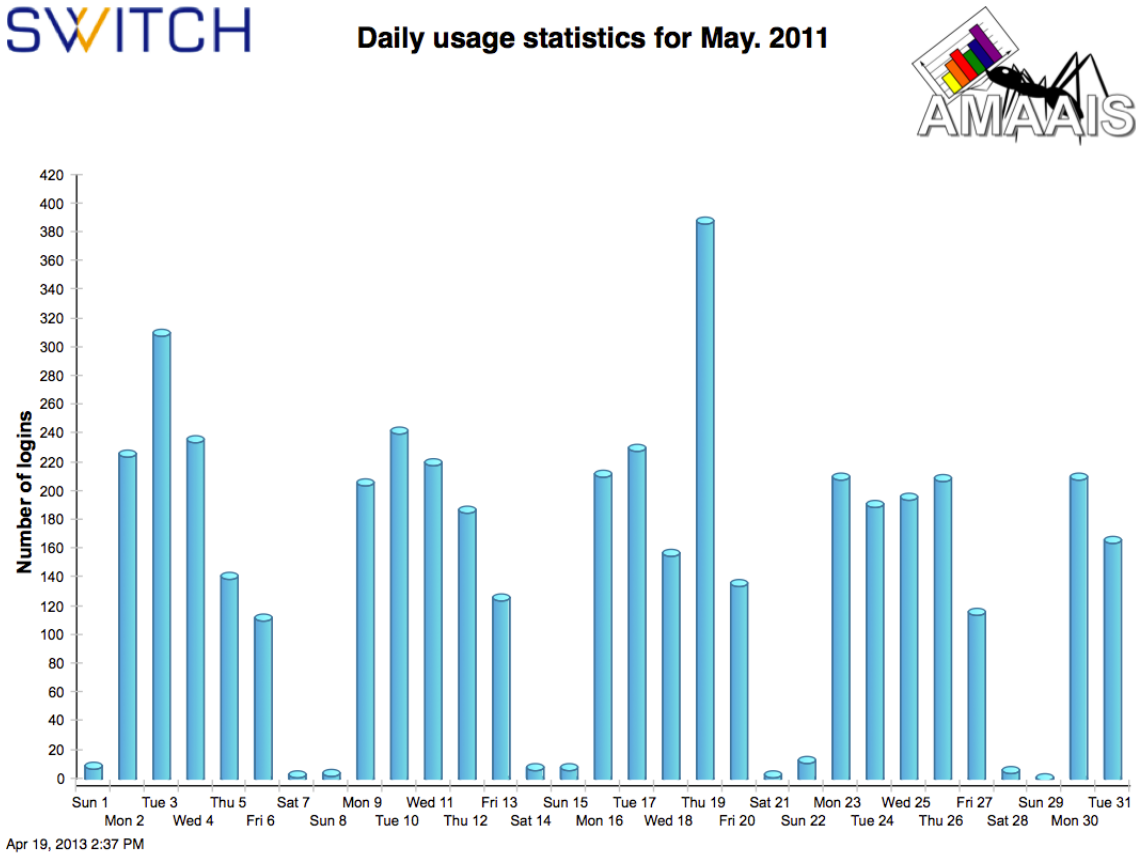

Figure 18: Daily Usage Statistics from an Identity Provider for May 2011.

The second report series illustrates the evolution of different login methods over time. In this case, a normalized bar diagram is used, that is each bar has exactly the same size independently of the total number of logins. The user can easily see the proportion of logins done with each login method (*e.g.*, username/password login, certificate login). The series consist of two reports for different time intervals. The user can drill down on the yearly report (Figure 19) to obtain a more detailed view (Figure 20) of the selected time frame.

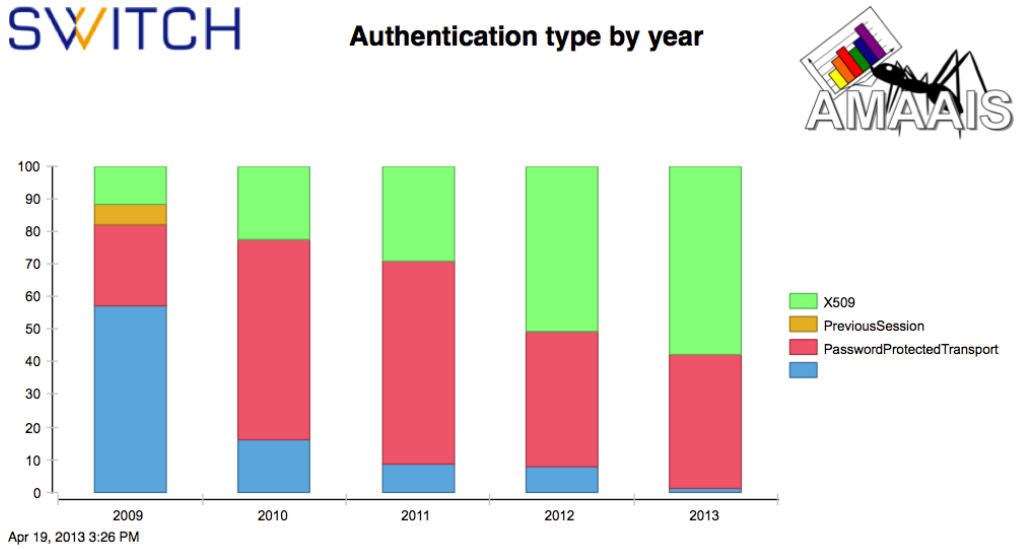

Figure 19: Authentication Type for an Identity Provider in a Yearly Comparison.

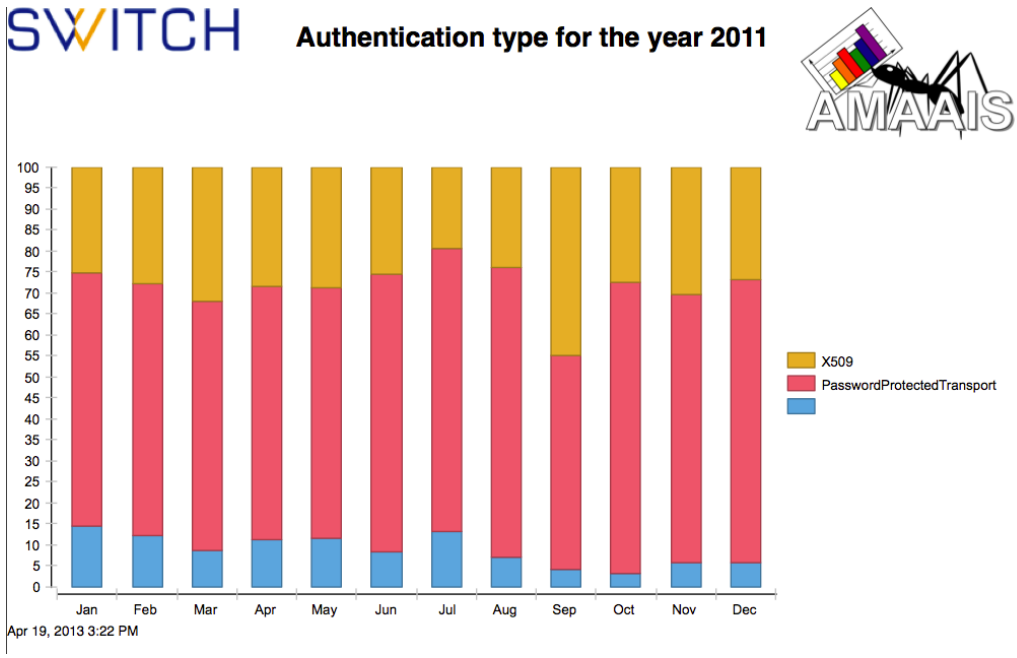

Figure 20: Authentication Type for an Identity Provider in a Monthly Comparison for the Year 2011.

The third report type displays the most frequently released attributes (Figure 21). It shows a horizontal bar chart starting with the most frequently attribute and gives the absolute number of times this attribute was released as a result.

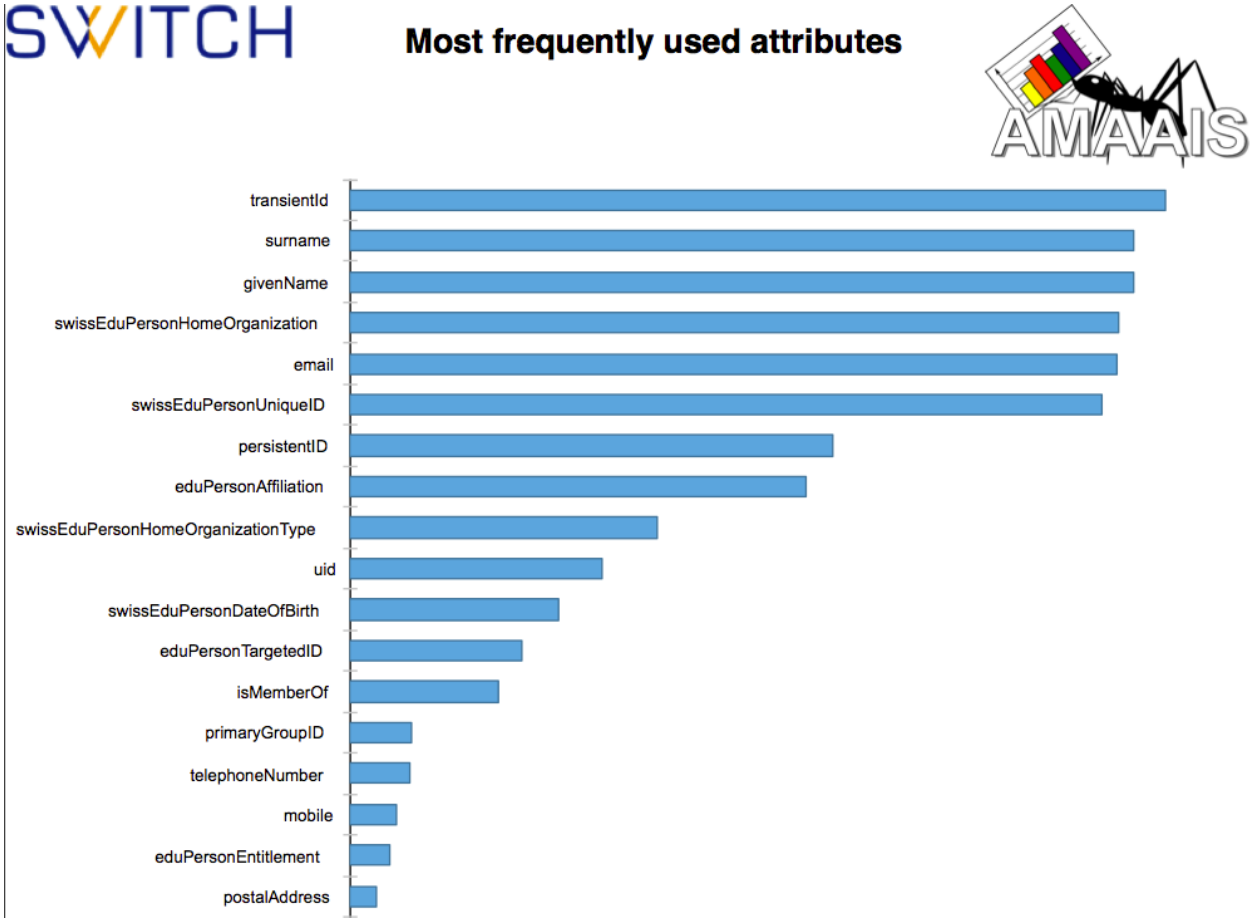

Figure 21: Overview of the Frequency of the Most Frequently Used Attributes.

7 Evaluation

In order to evaluate AMAAIS in qualitative and quantitative terms, Unit tests and trials were compiled. Unit tests prove that each fraction of the code operate as expected considering a given input. The trials give a more robust and solid evaluation view of the AMAAIS software, since it encapsulates end-to-end functionalities. Furthermore, the trials stress the AMAAIS solution in order to prove the scalability of the system.

7.1 Code Testing

The AMAAIS code is tested extensively through Unit tests for every method, of most in the AMAAIS core, and support classes. For the `Stage`s more complex integration tests have been written as TestNG tests in order to test the collaboration of `Stage`s. Furthermore, Unit tests have been created also for the AMAAIS consistency test classes that have been used during the AMAAIS trials. That results a 77% AMAAIS code coverage. The Shibboleth compatible testing framework TestNG [19] has been used for the AMAAIS Unit test.

7.2 Trials

The trials were based on the three AMAAIS use cases: SMS, Printing (Section 6.1), and the IdP (Section 6.2). This section presents the results of each use case trial phase, observing three major dimensions: (1) scalability, (2) data consistency, and (3) data completeness. While the SMS and Printing trials only focus on (2) and (3), the IdP trial mainly focus on (1). This decision was made due to the accounting `Event`s generation intensity inherent to the chosen use cases. On one hand, SMS and Printing use cases have `Event`s constantly being generated, but not likely presenting a high volume or bursts during a short time frame. On the other hand, the IdP use case may present a high volume of accounting `Event`s being generated in a short time frame, which may represent scalability issues in the `AccountingClient` and `AccountingServer` side. Related to (2) and (3), the SMS and Printing use cases are more prone to present data inconsistencies and incompleteness due to the fact that SMS and Printing `Event`s receive input from users. Therefore, *e.g.*, mal-formed or intentionally malicious inputs can bring unexpected `Event`s to the `AccountingServer`, not only impacting on the accounting but also possibly having charging impacts.

Data consistency and completeness related to `Event`s was checked as follows. The accounting application hashed the content of `Event`s and generated a local file (in the `AccountingClient` side) containing the `Event` Identifier with the corresponding hashed `Event` payload. This generated file is called *check file*. In the `AccountingServer` side, the same process was performed: for each `Event` persisted in the AMAAIS database, the `Event` payload was hashed and associated to its `Event` Identifier, generating a local *check file* with such information (in the `AccountingServer` side). In the end, both generated files were compared in order to achieve conclusions related to data consistency and completeness.

Related to data completeness, if one `Event` Identifier is located in the *check file* on the `AccountingClient` side but not in the *check file* on the `AccountingServer` side it means that such `Event` may be lost, and is considered as a "fail" `Event` result. If the `Event` Identifier is located in both files (just once), it is considered a "successful" `Event` result.

Related to data consistency, the same comparative process occur, but not only looking to the `Event` Identifier, but also comparing if the `Event` payload hash is the same in both *check files*. If a `Event` payload hash is different, then it is considered a "fail" `Event` result. If the `Event` payload hash is the same in both files for that given `Event` Identifier, it is considered a "successful" `Event` result.

7.2.1 SMS Trial

The SMS use case was deployed as follows for the trial:

Deployment Set-Up: The Virtual Machine `amaais1.ethz.ch` was hosting the SMS `AccountingClient` as well as the SMS accounting application (Section 6.1.1). The Virtual Machine `amaais2.ethz.ch` was hosting the `AccountingServer` and the AMAAIS database. Both Virtual Machines were placed within the ETHZ network.

Trial Start Date: Fri Apr 05 12:02:10 GMT+02:00 2013

Trial End Date: Tue Apr 23 14:34:30 GMT+02:00 2013

The SMS use case trial had the following results:

Events Generated: 57213

Data Consistency Results:

- Events Checked: 57213
- Events Failed: 0

Data Completeness Results:

- Events Checked: 57213
- Events Failed: 0

7.2.2 Printing Trial

The Printing use case was deployed as follows for the trial:

Deployment Set-Up: The Virtual Machine `amaais1.ethz.ch` was hosting the Printing `AccountingClient` as well as the printing accounting application (Section 6.1.1). The Virtual Machine `amaais2.ethz.ch` was hosting the `AccountingServer` and the AMAAIS database. Both Virtual Machines were places within the ETH Zurich network.

Trial Start Date: Tue Apr 09 14:28:42 GMT+01:00 2013

Trial End Date: Tue Apr 23 13:35:28 GMT+02:00 2013

The Printing use case trial had the following results:

Events Generated: 16464

Data Consistency Results:

- Events Checked: 16464
- Events Failed: 0

Data Completeness Results:

- Events Checked: 16464
- Events Failed: 0

7.2.3 IdP Trial

The IdP use case was deployed as follows for the trial:

Deployment Set-Up: The Virtual Machine `obelix.switch.ch` was hosting the IdP `AccountingClient` as well as the IdP accounting application (Section 6.2.1). The Virtual Machine `asterix.switch.ch` was hosting the `AccountingServer` and the AMAAIS database. Both Virtual Machines were places within the SWITCH network.

Since scalability is the main focus within the IdP trial, real Shibboleth IdP logs were collected from a on-production SWITCH machine and artificially copied to the `obelix.switch.ch` Virtual Machine. This step was done since the IdP accounting application could not be installed in a real on-production SWITCH machine due to stability issues that could be raised in the trial period. Therefore, the IdP logs used for the trial have 60440 IdP `Event`s, and the trial observed the following:

- How much time does it take to process all the `Event`s? *I.e.*, how much time all the `Event`s can be parsed with the IdP accounting application, sent from the `AccountingClient` to `AccountingServer`, and persisted to the AMAAIS database?
- How much CPU load, RAM, and network traffic was used in the machines hosting the IdP accounting application and `AccountingClient`, as well as the `AccountingServer` would present during the `Event` processing?

The results were collected after constant runs and showed a result average pattern. The IdP use case trial had the following scalability results:

Time to Process Events: 39 minutes and 55 seconds to process all 60440 events. **Time Rate to Process Events:** 1514 `Event`s per minute.

IdP Accounting Application and `AccountingClient` — obelix.switch.ch:

- CPU load: Figure 22 shows that `obelix` Virtual Machine reached a maximum peak of 60% of CPU load when the `Events` were being processed.

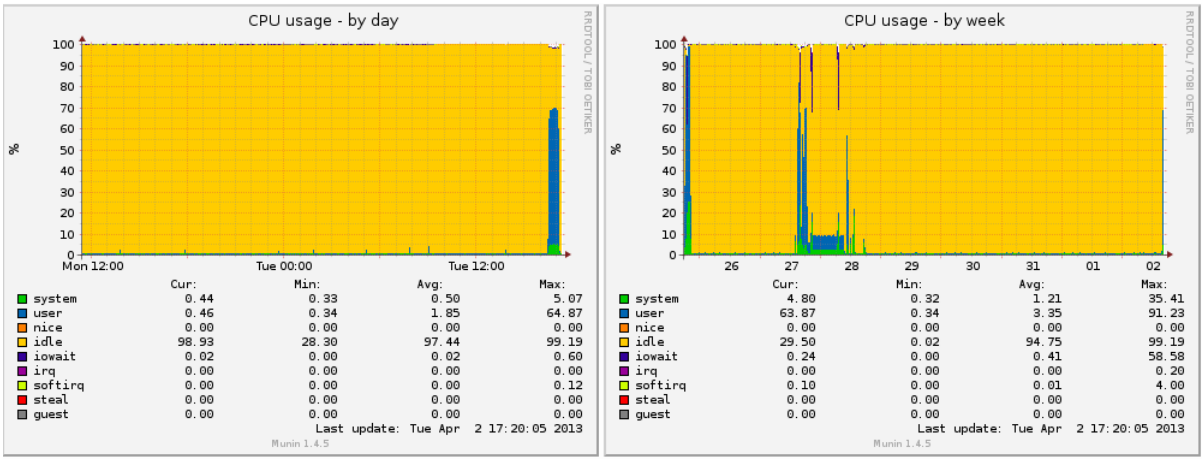

Figure 22: `obelix.switch.ch` CPU Load Usage.

- RAM usage: Figure 23 shows that `obelix` Virtual Machine reached a maximum of 700 MByte exactly when the `Events` were being processed.

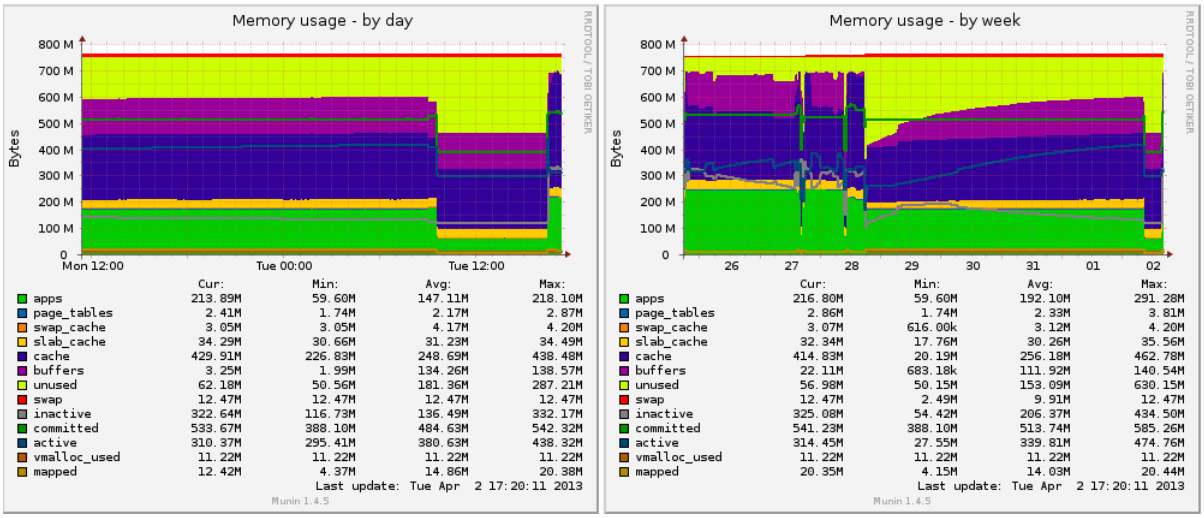

Figure 23: `obelix.switch.ch` RAM Usage.

- Network traffic: Figure 24 shows that `obelix` Virtual Machine reached a peak of 1.2 MBits per second of outbound traffic exactly when the `Events` were being processed.

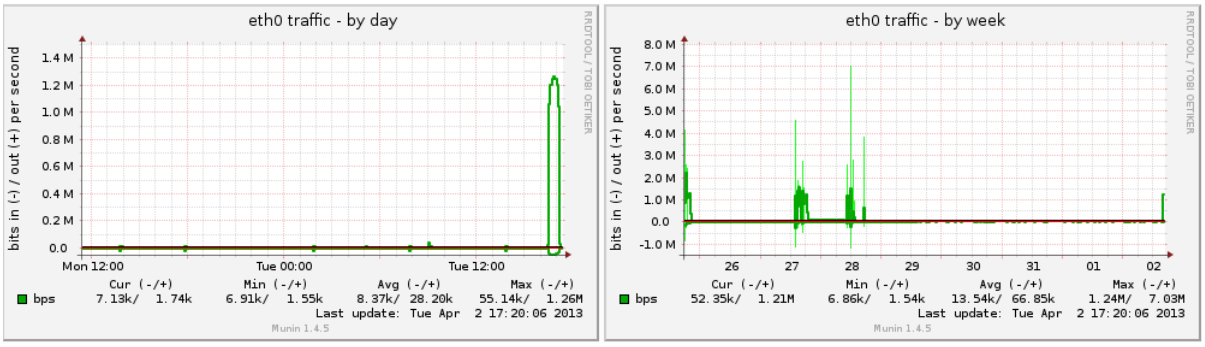

Figure 24: `obelix.switch.ch` Network Traffic.

Accounting Server — asterix.switch.ch:

- CPU load: Figure 25 shows that `asterix` Virtual Machine reached a maximum peak of 35% of CPU load when the `Event`s were being received and persisted.

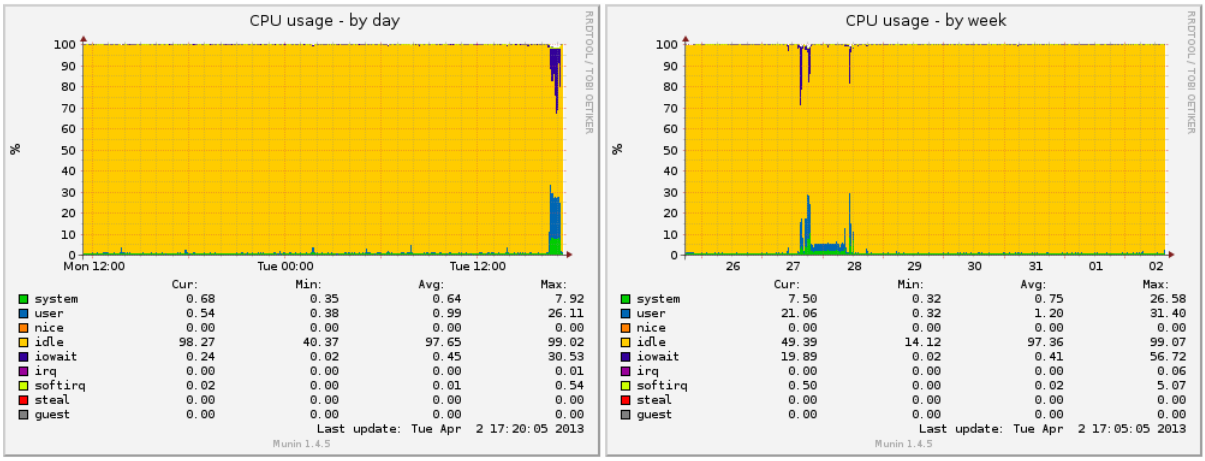

Figure 25: `asterix.switch.ch` CPU Load Usage.

- RAM usage: Figure 26 shows that `asterix` Virtual Machine did not have any drastic change of RAM utilization when the `Event`s were being received and persisted. The RAM usage remained stable considering the last few hours and also considering the last few days before the trial run.

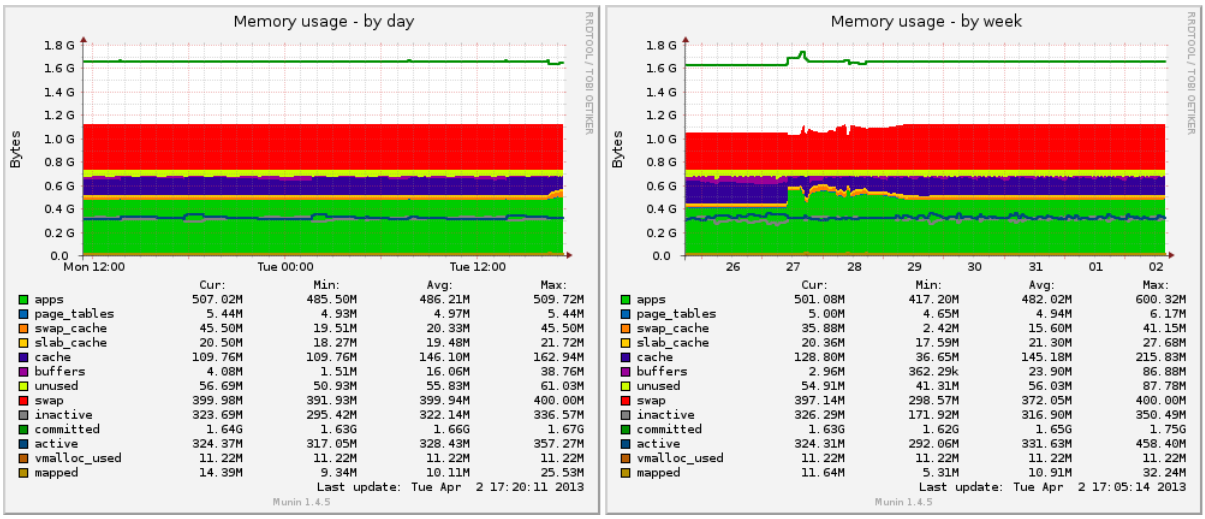

Figure 26: `asterix.switch.ch` RAM Usage.

- Network traffic: Figure 27 shows that `asterix` Virtual Machine reached a peak of 1.2 MBits per second of inbound traffic exactly when the `Events` were being received.

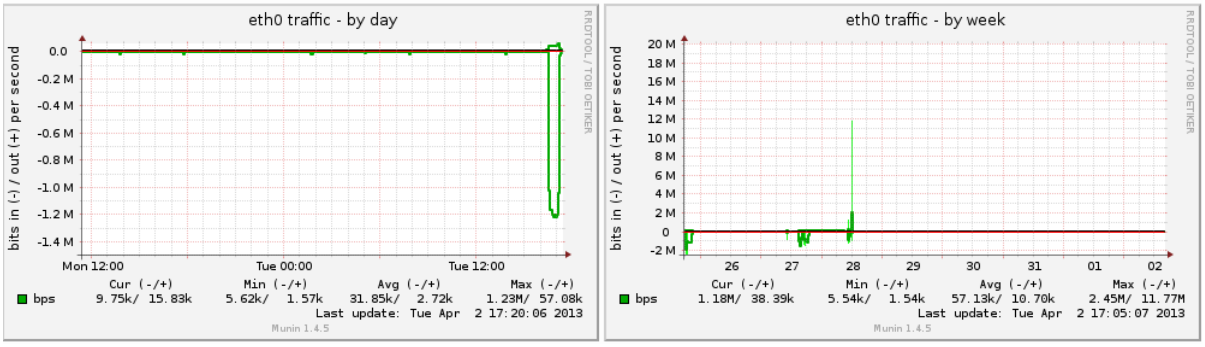

Figure 27: `asterix.switch.ch` Network Traffic.

In overall terms, the scalability results showed that the AMAAIS `AccountingClient` and `AccountingServer` are able to generate and receive, respectively, an average of 1514 `Events` per minute without presenting a high CPU, RAM, and network consumption.

7.3 Security

Today the term hacking is usually used in a negative sense. In the early beginning of the digital age the term hacker had two different meanings. A first definition said, that hacker is a person who enjoys learning the details of computer system. And the second definition stated, that a hacker is a person who programs enthusiastically, or who enjoys programming rather than just theorizing about programming [16] [14]. Today the term ethical hacker is usually used to distinguish hackers with a criminal intention of hackers

that are officially allowed to attack a system, to reveal vulnerabilities. Palmer [14] describes three main questions an ethical hacker wants to answer:

1. What can an intruder see on the target systems?
2. What can an intruder do with that information?
3. Does anyone at the target notice the intruder's attempts or successes?

The goal of this section is to present the technics and findings during the black-box and white-box hacking on an AMAAIS process which aimed to reveal system vulnerabilities. For the evaluated weaknesses countermeasures proposed.

Additional to this technical tests also a social based security assessment took place. To do this part in a structured manner the NIST Information Technology Security Assessment Framework [11] was used. NIST already provides a sample questionnaire to assess an organization. There is also a self assessment guide, where more questions are provided [18]. Due to the big coverage, only a selection of questions were considered. The NIST questionnaire that was generated for the AMAAIS evaluation need can be found in the Appendix of this document.

7.3.1 Black-box Hacking

The goal of this step was to find out, if a potential malicious user, who has no insight to the server, could attack the system and crash it. This approach assumes that the ethical hacker has approximately the same knowledge like a potential criminal hacker [5]. To start the black box testing phase in a structured manner, it was chosen to be done according to the five phases described by Hafele [5]. This approach starts with the phase of initial reconnaissance. This includes the investigation, if there are any readily available public information about the target system. The second phase is called service determination, or scanning phase. Hafele refers in this context to Namji [10] who writes that the activities of this phase are identification of listening services and ports that are operational on the system to evaluate. The third phase is the enumeration phase. The goal is to identify open network services for possible exploit. The next phase is gaining access. This includes many different attack techniques such as password cracking, and denial of service (DoS) attacks [10]. There are many possible forms of DoS attacks, but the goal is always to prevent, or limit the availableness of a service by use limited, or non-renewable resource; delete data or configurations, destroy computer or communications facilities [9]. The fifth phase is called privilege escalation. This stage assumes, that the hacker has gained access. Now the goal is to gain administrative or root level, to get complete control of the network [5].

For the initial reconnaissance, the web was searched for information about the AMAAIS project. As result the official website `amaais.switch.ch` was found. All the documents that are publicly available before this deliverable, are not useful to a criminal hacker, because some architecture has already been changed in Phase 4 of the AMAAIS project. Thus, during the black-box hacking section, no AMAAIS vulnerabilities has been revealed. However, it is recommended to protect AMAAIS related documents, in order to hide any

sensitive data that could be helpful information for a criminal hacker, such as AMAAIS system running servers (*e.g.*, `amaais1.ethz.ch` and `amaais2.ethz.ch`). Furthermore, technical details, such as the `AccountingServer` is running as a Java Servlet, could be used in combination with port scan [4] techniques in order to reveal future AMAAIS running servers. Thus, an ungenerous public insight to the AMAAIS detail is essential.

7.3.2 White-box Hacking

During the white-box hacking phase, information about the architecture and help how to use the AMAAIS server were provided to an ethical hacker. The goal was to reveal potential weaknesses of the AMAAIS system, that users with an insight to the infrastructure and architecture might get to abuse. The structure of the white box hacking process was chosen to be similar to the black box testing phase. Because the initial reconnaissance was already done, the process should start with phase three immediately.

The first step of the white box testing included the setup of the updated AMAAIS server and an example `AccountingClient`. Thereby it was possible to capture an example process, where an `AccountingClient` is sending data to the server (Listing 34). The first goal was to re-inject such captured packets with original, or modified data. Alexander [9] calls this a replay-attack. He defines it as a passive eavesdropping where the attacker is duplicating the message he has captured. This attack was chosen, because authentication protocols for applications like bank transactions are endangered [9]. In this case the captured data was re-injected with netcat. The AMAAIS server rejected this packet successfully. The result of this is shown in Listing 35.

Listing 34: Section of an `Event` Captured with Wireshark.

```
POST /amaais-acct-server HTTP/1.1
User-Agent: Jakarta Commons-HttpClient/3.1
Host: 127.0.0.1:8080
Content-Length: 14465
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?><acct:PublishEventRequest xmlns:acct="urn:mace:
    switch.ch:doc:accounting:profiles:1.0" xmlns:xsd="http://www.w3.org/2001/ XMLSchema"
    IssueInstant="-190184542-11-06T02:51:29.373Z" Version="2.0"><ds:Signature xmlns:ds="
    http://www.w3.org/ 2000/09/xmldsig#">
<ds:SignedInfo>
<ds:CanonicalizationMethod Algorithm="http://www.w3.org/ 2001/10/xml-exc-c14n#"/>
<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/ xmldsig#rsa-sha1"/>
```

Listing 35: Re-injected Packet.

```
nc 127.0.0.1 8080 < original.txt
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
AMAAIS_STATUS: REJECTED
AMAAIS_REASON: Replayed
Content-Length: 0
Date: Tue, 02 Apr 2013 16:22:03 GMT
```

In the next replay-attack attempt included an original message modification. The first modification involved “ds:SignatureValue” of the element “ds:Signature” of the “acct:PublishEventRequest”. The first character of the element value was replaced by another character. Afterwards this modified content was sent again with netcat. This time the server accepted it. The next modification included all “ds:SignatureValue”-elements. Again just the first char of all these values was were replaced. This time as it is shown in Listing 36, the server did not accept the packet with the reason “Untrusted Assertion”.

Listing 36: Rejected Packet.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
AMAAIS_STATUS: REJECTED
AMAAIS_REASON: Untrusted Assertion
Content-Length: 0
Date: Fri, 22 Mar 2013 13:07:31 GMT
```

Concluding this section, during the white-box hacking several packet modifications, as well as a syn flooding test (DoS attack) took place and no significant threats for AMAAIS were found. However, the ethical hacking period revealed some minor bugs in the code which have been identified and fixed. The findings for both the black-box and the white-box hacking are summarized in the following table.

Table 1: Conclusion.

Approach	**Test description**	**Result**
Black-box hacking	Initial reconnaissance	Unprotected AMAAIS Deliverables =>Protect future AMAAIS related documents
White-box hacking	Replay Attack:Original packet	Rejected by the server
	Replay Attack: Modification of “ds:SignatureValue”	Accepted =>Bug fixed
	Replay Attack: Modification of all “ds:SignatureValue”’s	Rejected by the server
	Replay Attack: Modification of “ds:DigestValue”	Accepted =>Bug fixed
	Replay Attack: Modification of all “ds:DigestValue”’s	Rejected by the server
	Replay Attack: Modification of Xml structure	Rejected by the server
	DoS attack	Server blocked due to hardware limitations =>No significant threat

8 Summary and Conclusions

This document determines the fourth and last deliverable of the AMAAIS project. It becomes available at a time when the AMAAIS project sees its formal end. Accordingly, this deliverable provides the documentation of the final AMAAIS solution. This includes the AMAAIS design and core concepts used, its system implementation, guidelines to deploy AMAAIS, details on the three use cases adopted in the project, and finally evaluation results both in terms of testing and trials conducted as well as a security assessment.

Parts of these contributions have been covered in previous AMAAIS deliverables. For instance, design considerations were an important focus in deliverable D2 [8], or suited use cases were initially investigated in deliverable D1 [15]. While many concepts presented in previous deliverables remain valid, it is important to note that this deliverable D4 represents the final AMAAIS solution: Deliverable D4, thus, constitutes the reference document for the AMAAIS project and its solution.

In this light, this deliverable provides relevant information for a technical audience with an interest in accounting and monitoring functionality for services offered in a federation environment. Based on a short introduction into the basic concepts and terminology of relevance, the interplay of AMAAIS core components (`AccountingClient` and `AccountingServer`) and AMAAIS accounting applications was presented. Such presentation of key design concepts of AMAAIS was further deepened by explaining how `Source`s and `Sink`s help facilitate highly configurable pipelines for queuing, filtering, persisting, or transmitting events in an `AccountingClient` or `AccountingServer`.

The AMAAIS system implementation maps these key design concepts and their distinctive functionality to the various channels and stages that the solution offers. For each channel and stage type supported, this deliverable provided sample Spring configurations. Furthermore, the persistency framework and supported security mechanisms (signing, encrypted communications between `AccountingClient` and `AccountingServer`, and replayed messages) were explained.

Driven by these notions of the AMAAIS system design and implementation, the interested audience was given guidelines on deploying AMAAIS, which was complemented by detailed implementation information obtained from the set of three use cases adopted in the project—two of them (SMS and printing) applying service-specific accounting, while the third use case is service-independent in the sense that its accounting application creates events from Identity Provider log files.

In conclusion, this deliverable shows, on the one hand, how AMAAIS is designed, how it is implemented—both in terms of its core components as well as in terms of use case-driven implementation work—and how AMAAIS can be deployed. On the other hand, this deliverable shows that the AMAAIS project has produced an accounting and monitoring solution that is proven to work and to scale well with an increasing number of events to be treated. First, the AMAAIS implementation has been tested thoroughly and successfully. Second, AMAAIS successfully passed both a black-box and a white-box hacking attempt. Third, and probably most importantly, long-term trials with operational conditions have been designed, conducted, and assessed for all three use cases considered in the project.

These trials have revealed bugs—all of which are fixed in the final solution—and they have revealed that each and every event entering the AMAAIS accounting system was persisted

in the respective AMAAIS server database, while the trial assessments proved that none of these events was altered in a non-intended way by AMAAIS. The trial for the Identity Provider use case focused in addition on scalability and performance: Even under high load (measured by the amount of Identity Provider requests per time unit), the AMAAIS solution was able to perform its accounting tasks gracefully and correctly.

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. Moreover, thanks to Chad LaJoie (Shibboleth Java Components Lead at Internet2), Phil Smart and Rhys Smith (Identity and Access Management at Cardiff University) for the ideas and discussions in the AMAAIS development mailing list.
AMAAIS has been carried out as part of the “AAA/SWITCH e-infrastructure for e-science” program under the leadership of the Swiss National Research and Education Network (SWITCH), and has been supported by funds from the State Secretariat for Education and Research (SER).

References

[1] Bourrillion K., "Class Predicates," http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/base/Predicates.html, (Accessed:) Apr 2013.

[2] ETH Zürich, "SMS Gateway der ETH Zürich," http://www.sms.ethz.ch/, (Accessed:) Apr 2013.

[3] ETH Zürich, "VPP – Verteiltes Printen und Plotten," http://www.vpp.ethz.ch/, Jan 2013.

[4] Gruman G., *OS X Mountain Lion Bible*, 1st ed. John Wiley & Sons, Sep 2012.

[5] Hafele D., "Three Different Shades of Ethical Hacking: Black, White and Gray," SANS Institute, Tech. Rep. SANS Institute, GSEC Practical Assignment, Version 1.4b, Feb 2004, http://www.sans.org/reading_room/whitepapers/hackers/shades-ethical-hacking-black-white-gray_1390.

[6] Hämmerle L. (SWITCH), "SWITCH Shibboleth Service Provider (SP) 2.5 Configuration Guide," https://www.switch.ch/aai/docs/shibboleth/SWITCH/2.5/sp/deployment/configuration.html#3, Jan 2012.

[7] La Joie C., Cantor S., "IdPAddAttributeFilter Matching Rules," https://wiki.shibboleth.net/confluence/display/SHIB2/IdPAddAttributeFilter#IdPAddAttributeFilter-MatchingRules, (Accessed:) Apr 2013.

[8] Machado G. (Edt.), "AMAAIS Phase 2: Architecture Design and Implementation," Accounting and Monitoring of AAI Services (AMAAIS) Project, Tech. Rep. Deliverable D2, Jan 2011, http://www.csg.uzh.ch/research/amaais/amaais-d2.pdf.

[9] Michael A., *Netzwerke und Netzwerksicherheit - Das Lehrbuch*, 1st ed. mitp-Verlag, Oct 2006, ch. 2.4 Attacken und Gegenmassnahmen, pp. 113–194, (in German).

[10] Najmi, *How Hackers/Crackers Break Into Your System?*, http://www.techiwarehouse.com/engine/17249a96/How-Hackers/Crackers-Break-Into-Your-System, Aug 2004.

[11] National Institute of Standards and Technology (NIST), *Federal Information Technology Security Assessment Framework*, http://csrc.nist.gov/drivers/documents/Federal-IT-Security-Assessment-Framework.pdf, Nov 2000.

[12] Oracle Inc., "Class Observable," http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html, (Accessed:) Apr 2013.

[13] Oracle Inc., "Interface Observer," http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html, (Accessed:) Apr 2013.

[14] Palmer C., "Ethical hacking," *IBM Systems Journal*, vol. 40, no. 3, pp. 769–780, Mar 2001, http://dx.doi.org/10.1147/sj.403.0769.

[15] Racz P., Machado G., Waldburger M. (Eds.), "AMAAIS Phase 1: Scenarios, Requirements, and High-Level Architecture," Accounting and Monitoring of AAI Services (AMAAIS) Project, Tech. Rep. Deliverable D1, Oct 2009, http://www.csg.uzh.ch/research/amaais/amaais-d1.pdf.

[16] Raymond E., *The New Hacker's Dictionary*, 3rd ed. The MIT Press, Oct 1996.

[17] SpringSource Community, "What is Spring?" http://www.springsource.org/, Apr 2013.

[18] Swanson M., "Security Self-Assessment Guide for Information Technology Systems, NIST," U.S. Department of Commerce, Computer Security Division, Information Technology, National Institute of Standards and Technology (NIST), Tech. Rep. NIST Special Publication 800-26, Nov 2001.

[19] TestNG, "TestNG," http://testng.org/, Mar 2013.

Appendix

Adapted NIST questionnaire

Category of Senitivity	Confidentiality	Integrity	Availability
High			
Medium			
Low			

Specific Control	L.1Policy	L.2 Procedures	L.3 Implemented	L.4 Tested	L.5 Integrated	Risk Based Decision Made
Personnel Security						
Are all positions reviewed for sensitivity level?						
Are mechanisms in place for holding users responsible for their actions?						
Authentication						
Are passwords, tokens or biometrics used?						
Do passwords contain alpha numeric, upper/lower case and special characters?						
Are passwords changed at least every ninety days or earlier if needed?						
Is there guidance for handling lost and compromised passwords?						
Are passwords transmitted and stored with one-way encryption?						
Is there a limit to the number of invalid access attempts that may occur for a given user?						
Risk Management						
Are risk assessments performed and documented on a regular basis or whenever the system, facilities, or other conditions change?						
Has data sensitivity and integrity of the data been considered?						
Have threat sources, both natural and man-made, been identified?						
Has a list of known system vulnerabilities, system flaws, or weaknesses that could be exploited by the threat sources been developed and maintained current?						

Specific Control	**L.1Policy**	**L.2 Procedures**	**L.3 Implemented**	**L.4 Tested**	**L.5 Integrated**	**Risk Based Decision Made**
Risk Management						
Has a mission/business impact analysis been conducted?						
Review of Security						
Has the system and all network boundaries been subjected to periodic reviews?						
Has an independent review been performed when a significant change occurred?						
Are routine self-assessments conducted?						
Are tests and examinations of key controls routinely made, *i.e.*, network scans, analyses of router and switch settings, penetration testing?						
Physical and Environmental Protection						
Is access to facilities controlled through the use of guards, identification badges, or entry devices such as key cards or biometrics?						
Does management regularly review the list of persons with physical access to sensitive facilities?						
Are unused keys or other entry devices secured?						
Is suspicious access activity investigated and appropriate action taken?						
Are sensitive data files encrypted on all portable systems?						