Service Oriented Architecture for Sensor Networks Based on the Ambient Computing Environment

Leon S. Searl

ITTC-FY2008-TR-41420-07

February 2008

Project Sponsor:
Oak Ridge National Laboratory

Copyright © 2008: The University of Kansas
2335 Irving Hill Road, Lawrence, KS 66045-7612
All rights reserved.
1 Overview

ACE SOA (Ambient Computing Environment)(Service Oriented Architecture) is the forth generation ACE implementation for providing distributed computing, media and sensing services to service consumers (clients) in a dispersed environment. ACE SOA is the infrastructure providing message and data communication, confidentiality, authenticity and permissions plus service discovery within an enterprise and between enterprises. ACE also provides a framework for developing new services and clients of services.

ACE SOA is a reimplementation of the original ideas of ACE but utilizing current technology and widely accepted Web Service specifications and publicly available implementations which are suitable for Sensor Networks. Some of the Web Service specifications in use are SOAP, the WS-X specifications and UDDIv3.

2 ACE SOA Infrastructure

ACE SOA is an infrastructure to allow Web Service based clients and services of one or more enterprises to interact using the following features:

- Provide means for service to publish its URL location and Web Service Interface for discovery by clients.
- Allow clients to discover service’s URL location and Web Service Interface.
- Provide a secure communication channel between clients and services.
- Provide mechanism for clients to subscribe to service ‘events’ or ‘alarms’.
- Authenticate a client to a service.
- Provide fine grain authorization of a client’s use of a service.
- Provide a framework for development of new clients and services.
- Establish a trust relationship between enterprises.
Figure 1 Federation of Services Overview

Figure 1 illustrates a simplified federation of clients and services. The client in this example is in ‘red’. The services are in blue. The ACE infrastructure provided services are the ‘Service Directory’, ‘Authenticate’ and ‘Authorize’ services. The remaining services have been developed by the enterprise.

Each federation has its own Service Directory (UDDI), Authentication service and Authorization service (although this example does not show an authorization service for enterprise A).

When a service starts up it publishes its location URL (host and port) and its Web Service interface (operations and operation arguments and argument types) to its Service Directory so that clients are able to find it. In Figure 1 the ‘Sense DB’ service of Enterprise B has published its information to the ‘Service Directory’ service.

Service Directories subscribe with each other for new inter-service publications. When a new service that can be used from outside the enterprise publishes with its Service Directory, the Service Directory then notifies any other subscribed Service Directories about the new service. In Figure 1 the Service Directory of Enterprise B has notified the Service Directory of Enterprise A of the service ‘Sense DB’ from Enterprise B.

A client can discover a service by querying its own Service Directory (query can be by service type or by some other distinguishing category published by a service). The service directory tells the client where the service(s) is and the interface (operations and arguments to the operations) needed to communication with the service. The service may be in the client’s own enterprise or it may be in another enterprise. In Figure 1 the Client has queried for a service that is in the ‘Sensor Database’ category and its Service
Directory has returned the location URL and interface for the ‘Sense DB’ from Enterprise B.

Services may also be clients of other services. In Figure 1 ‘Sense Proxy’ is a client of ‘Sensor’. Service ‘Sense Filter’ is a client of service ‘Sense Proxy’. Service ‘Sense DB’ is a client of ‘Sense Filter’, ‘Authorize’, ‘Authenticate’ and ‘Service Dir’.

The Enterprise A client communicates with the Sense DB service using encrypted messages. The ‘user’ of the client will be authenticated by ‘Sense DB’ using a certificate that the client obtained from its Authenticate service and passed along it’s message.

The service determines if a client/user is authorized to perform the requested operation in the message by querying its Authorize service.

In Figure 1 there is a remote service that is connected by a slow or intermittent connection so instead of communicating with Service Dir, Authenticate and Authorize services it has a ‘Sense Proxy’ service that handles that communication. Only the ‘Sense Proxy’ service is allowed to communicate with the remote service so the remote service can have its authentication and authorization information in local files.

2.1 Client/Service Communication

ACESOA clients communicate with ACESOA services utilizing the open web service standard SOAP (Service Operation Access Protocol) specification. ACESOA only uses version 1.2 of the SOAP specification. The implementation of SOAP used by ACESOA is the Apache Axis2 package.

![Figure 2 ACESOA Message Stack](image-url)
Figure 2 shows the SOAP message stack utilized by ACESOA. A message originates with the client application code, passes down through the SOAP layers to the underlying network. The network routes the message to the proper host and port. The message then passes up through the soap layers for the service until it reaches the service application code.

Each client and each service has ACESOA specific code that is used to interface with portions of the SOAP stack (labeled as ‘ACE Infra’).

The Serialize and XML layers adhere to the SOAP 1.2 specification for the message body. This is integral to Apache Axis2.

The WS-Authorization specification has not been written yet. Until the WS-Authorization is written ACESOA will utilize xACML (extendable Access Control Markup Language) to specify credentials and will make a best guess for the protocol for placing xACML into SOAP messages based a working draft WS-Authorization XML schema found on the web.

The Authentication layer uses the WS-Security mechanism for signing headers and messages. This is implemented by the Apache Rampart module.

The Encryption layer uses the WS-SecureConversation specification. This is implemented by the Apache Rampart module.

WS-Addressing layer follows the like named specification. This is integral to Apache Axis2.

**2.1.1 Client SOAP Stack**

To invoke a procedure on a service the client must first create the message body. Since ACESOA utilizes the Axis Axiom Object Model the client must create the message as an Axiom Object Model. The Object Model has the procedure name and procedure argument names and argument values placed in it.

The client passes the OM (message object model) to SOAP which then serializes the OM. The serialized object model is placed in XML in the SOAP schema.

On the client side WS-Authorization is skipped.

On the client side the Authentication layer (which is really a part of WS-Security) is given a certificate or token that has been provided by an enterprise authentication Certificate Authority (CA) through the ‘Ace Infra’ layer. The ‘Ace Infra’ layer handles this automatically for the client. The token or certificate identifying the user is placed in the message.

The message head and body is encrypted by WS-SecureConversation which is part of WS-Security. WS-Security in Axis2 to is provided by the Rampart module.
SOAP routing information (service being called, session in the service to use and the procedure being called) is placed in the message at the WS-Address layer.

The message is encapsulated and network routing information is added by the transport layer then the message is placed on the network.

2.1.2 Service SOAP Stack

A message from an ACESOA client, destined for an ACESOA service is first received by the transport layer from the network. The SOAP message is extracted from the transport message and delivered to the WS-Addressing layer.

The WS-Addressing layer determines from the message which service the message is for, which session within the service to deliver the message to and which procedure in the service session to invoke.

The header and body of the message are decrypted by WS-SecureConversation in the WS-Security layer. WS-Security is implemented by the Rampart package from Apache.

The authentication of the client user is determined by extracting the token or certificate from the message and verifying the token or certificate with the CA of the service’s enterprise. The communication with the CA (Authenticate service) is handled by the ‘Ace Infra’ layer. To reduce communication overhead with the Authenticate service the token or certificate is cached in the service with a limited cache lifetime. The service also has subscribed to the Authenticate service to be notified if the validity of the user’s token or certificate changes. Any failed authentication results in an error message back to the client.

The WS-Authorization layer determines which procedure and the arguments of the procedure that the client is trying to use. To determine if the client is authorized to invoke the procedure with the specified arguments the service’s Authorize service is queried. To reduce communication overhead with the Authorize service the user’s authorization credential is cached in the service with a limited cache lifetime. The service also has subscribed to the Authorize service to be notified if the validity of the user’s credential changes. Any failed authorization results in an error message back to the client.

The message is extracted from the XML and placed in an Axiom Object Model by the XML and serialize layers.

The appropriate method for invoking the procedure is called with the Axiom Object Model as the argument.

2.1.3 Event Notification

An Event is the asynchronous generation of data by a service (an event). Event Notification is the sending of Event data by asynchronous message (publishing) from a service to an event subscriber. An event generating service allows clients to send
subscription messages to it specifying the desired event and the destination for the event notification (publishing).

Event notification in ACE_SOA is based on the WS-Eventing specification and implemented by the Apache Savan module for Apache Axis2.

In Web based SOA, messages are either one way or in-out exchange. An in-out exchange is one message into a service and a single response to the client that originated the message. Clients must always initiate a message exchange. Clients may never receive a one way message nor be the recipient of the first message of an in-out exchange. Because of this Web Service restriction a client can not receive an Event Notification directly.

For a standalone client to receive an Event Notification it is necessary to embed a service server within the client. The service server then starts up a service that can receive Event Notifications. The client must register a callback class/method with the embedded service to be called when the embedded service receives an Event Notification. When the client sends a subscription request to an Event Notice generating service it includes in the request the URI of the embedded service as the delivery endpoint of the Event Notification.

When a ‘normal’ service wants to receive Event Notifications the above described scenario is not require. A ‘normal’ service runs within a server (such as Axis2 within Apache Tomcat) so all that is required of the service is to subscribe to the event with the delivery URI is its own endpoint.

The Savan module intercepts subscription messages for a service so it is not necessary to add anything to a service to handle subscriptions. When a service wishes to publish an event it makes a Savan API call with the event data as an argument. The Savan core code takes care of determining which subscribes are to be notified of the event and sends the event message to those subscribers.

### 2.2 Service Directory Service

The ACE Service Directory Service (ASD) stores location URL and interface information about currently running ACESOA services. Without the Service Directory, clients would have to ‘know’ the location (host, port and service name) and the interface (procedures and their arguments) of its desired services. This information would have to be hard coded or stored in configuration files for the client. If the interface or location of a service were to change the client code would have to be changed or the configuration file of the client would have to be changed. Changing client code would be impractical. Changing a configuration file would be cumbersome. With many different clients using the same service any service change would prohibitively difficult to manage.

ACESOA uses the [Web Service specification UDDI](Universal Description Discovery and Integration) version 3 for a standard interface to service discovery. The implementation used is the [OpenUDDI Server](which is based on the Novell UDDI)
Server. With UDDI v3, UDDI servers may be replicated and UDDI has the ability to register subscriptions for events.

The UDDI differs from the WFS (Web Features Service). The UDDI is intended to only store programmatic interface information about a service. The WFS is intended to store geographic features. The UDDI and WFS can be used together to locate and interface to a service in a specified geographic area. The UDDI can also be used to find the URL to the WFS.

Although there may be more than one Service Directory (UDDI) in an enterprise, generally a client or service will only use one Service Directory. There is no mechanism to ‘search’ for a Service Directory so the network location of an ASD for a client or service must be stored in a configuration file.

2.2.1 Service Publication

When an ACE service is starting up it must register its location and interface with the ASD. The Web Service specification for describing a service location URL and interface is WSDL (Web Services Description Language). In ACESOA WSDL v2.0 is used. Information from the WSDL for the service is stored in the UDDI server (ASD) as described in the OASIS Technical Note “Using WSDL in a UDDI Registry”.

In addition to the ‘standard’ WSDL information for a service the ACESOA infrastructure uses Java’s introspection to travel the service’s Java class hierarchy to find each inherited class and each implemented interface. The name of each inherited class, interface and the class’s name are stored in UDDI as added category values for the service’s interface under the category key named ‘acesoa:service-intf’. These categories are used by clients to search for services with a desired interface implementation. Example: All services that provide a temperature measurement implement the “org.tssn.service.sensor.temperature” interface. This interface name is stored in the ASD as a ‘acesoa:service-intf’ category for each service that implements it when the service is published by the ACE infrastructure as the service starts up. A client can ask the ASD for all services that have the ‘acesoa:service-intf’ category value of ‘org.tssn.service.sensor.temperature’.

The mechanism for a service to register with the ASD is automatic and does not have to be considered by a service developer. ACESOA utilizes the Apache Axis2 implementation of SOAP with its AxisObserver interface. When Axis2 deploys a service the ACESOA implementation of an AxisObserver, named AceServiceEventListener, is invoked. The ‘serviceUpdate’ method of the AceServiceEventListener extracts the WSDL from the service and information from the service’s configuration file and publishes the service’s location URL and interface to the ASD(UDDI).

All of the information needed by the service to locate and communicate with the UDDI (username, password) is stored in the service’s Axis2 service configuration ‘service.xml’ file. It is up to the administrator of the host running the service to ensure that the ‘service.xml’ file can not be read by unauthorized users of the host.
Since OpenUDDI (the implementation used for the ASD) utilizes Axis2 for its SOAP implementation, the SOAP message authentication and authorization mechanism used by ACE clients and services would be used with the UDDI. To avoid a chicken and egg problem the certificates for authentication signing of headers and messages are stored in a configuration file for the UDDI\(^1\).

### 2.2.1.1 ASD Service Alive Status\(^2\)

Previous version of ACE had a mechanism in the ASD and in the service infrastructure that provided a means of determining if a service that was registered with the ASD was still alive. The service would send a keep-alive message to the ASD periodically to tell the ASD that it was alive. The ASD had a timeout for each service. If the service did not send its keep-alive message to the ASD within the timeout period the ASD would remove the service information.

The UDDI has no ‘active service’ mechanism similar to past ACE ASDs. To have the UDDI only have published information of active services a ‘Active Status’ service is required. An Active Status service ‘pings’ each service listed in the ASD periodically to determine if it is alive. If the service fails to respond to N successive pings the Active Status service would un-publish the no longer responding service.

### 2.2.2 Service Discovery by Client

An ACESOA client uses the Service Directory service to discover the location URL and interface of one or more desired services. The ACESOA infrastructure provides an API for the client to call to perform the search. The method ‘ACEClient.findServices’ takes as an argument a string that is the name of a service interface that a service must implement in the UDDI search. The client must be coded to “know” how to interact with any service that implements the specified service interface. The ‘findServices’ method queries the UDDI (ASD) to search for all services that have published interfaces with categories that include the key name ‘acesao:service-intf’ and the category value as the specified service interface name. The UDDI returns the information it has on each service that matches the search.

The information returned by the ‘findServices’ method for each service found that matches the specified service interface includes:

- URL of the service
- Name of the service
- Namespace of the service (needed to create SOAP messages to the service).

\(^1\) The mechanism for Authorization is has yet to be determined

\(^2\) This service has not yet been implemented (November 7, 2007).
In a near future version of ACESOA, extended data stored in the UDDI for the service shall also be returned.

2.3 Authentication Service

NOTE: This infrastructure item is currently being integrated (November 7, 2007).

- Authentication service exists in each enterprise to provide certificates to clients to prove who the client user is.
- The authentication service verifies for a service the authentication certificates received by a service from a client.
- A trust relationship has to be established between enterprises so that Enterprise A will accept certificates issued by the authentication service of Enterprise B and visa-versa.

2.4 Authorization Service

NOTE: This infrastructure item has not yet been integrated (November 7, 2007).

- Services use authorization service to verify that the user specified in the client’s message authentication certificate has the authority to invoke the procedure specified in the message
- The WS-Authorization specification has not been written yet.
- Items used to determine authorization include:
  - Time of day
  - Service name
  - Host name
  - GEO Location of service
  - Name of user
  - Procedure Argument values

2.5 Remote Services

NOTE: This infrastructure item has not yet been integrated (November 7, 2007).

- Remote services are not directly connected to the internet.
- Frequently have limited bandwidth and limited duration connections to the home office. (examples: GSM, Satellite Phone).
Because of limited connections it is not practical to have remote services use the ASD, Authorization and Authentication services (too much communication overhead and out of date information).

Use a service proxy that is connected to the internet and the wireless comm Link to the remote service.

- Proxy is responsible for handling ASD publishing, authentication and authorization of clients.
- Proxy communicates with remote service using single/fixed authentication and authorization. Remote service compares authentication and authorization with local files instead of using ‘normal’ authentication and authorization services.
- Comm link between proxy and remote service is only ‘up’ during message exchange.

3 Client and Service Development

- The Apache Axis2 package is written in Java and thus clients and services developed for ACESOA are written in Java.
- Embedded services can use gSOAP and be written in C++.
- ACESOA has client and service Java code templates to ease development of new clients and services.

4 Scenario Examples

This section contains examples of how the ACESOA infrastructure, clients and services fit together.

4.1 Client/Service Messaging Example plus Remote Service Example

In this example an ACESOA client discovers a desired service that has published itself with the ASD. The service happens to be a Service Proxy for a remote service so the mechanism for communicating with a remote service is also shown in the example. The following interactions are described:

- Service publication

3 The content of this section will be written once the client/service authentication and authorization mechanisms have been implemented.
- Client search for a service via Service Directory (UDDI).
- Client user authentication.
- Client user authorization
- Client/Server communication
- Client/Remote Service communication
- Remote Service authentication
- Remote Service authorization

In this example the communication between a remote service proxy and a remote service is shown. This is different than ‘normal’ ACESOA communication since the assumption is that the communication link with the remote service is over a slow and time limited wireless link (example: GSM mobile phone).

Because of the limitations of the communication link the remote service keeps a local authentication file of trusted Certificate Authorities (CAs) and a local authorization file of credentials for a single user or only a very few users.

All communication with the remote service by clients must pass through the proxy service. The proxy service is the only entity that knows how to connect to the remote service and is the only user authorized to use the remote service (via the remote services authorization file).

In Figure 3 the initial state of the system is with the 3 ACESOA infrastructure services “Service Directory”, “Authenticate” and “Authorize” and the remote service “Sensor” running.
In Figure 4 the Sensor Proxy service starts up and is automatically published to the ASD by the ACESOA infrastructure. The service’s location URL and interface information (WSDL information) is stored in the ASD UDDI database for later query by a client looking for the service.
When starting up the Sensor Proxy Service obtains the information it needs about its remote service from its configuration file as show Figure 5. Included in the information is the remote service GSM phone number.

In order to communicate with the remote service the Sensor Proxy service must act as a client and obtain the remote service client authentication token/certificate as shown in Figure 6. To simply the remote service, the token/certification for only Sensor Proxy user is stored in the Authentication file.
A client that intends to use the “Sensor Proxy” service starts up as shown in Figure 7. The name of the client is “Sensor Filter”.

Figure 7 Client Startup
The client “Sensor Filter” wants to find the “Sensor Proxy” service as shown in Figure 8. The client knows the location of the ASD from an entry in its configuration file. An ACE API call is used by the client to perform the search query. The client specifies that it wants a service that implements the interface named “Sensor Proxy”. There could also be more query information such as the sensor type desired.

Figure 8 Client Query for Service

The ASD returns to the client those services that match the service query. In this example, shown in Figure 9, the service information returned to the client contains the WSDL information such as location (URL) and interface for the “Sensor Proxy” service.

---

4 The additional query information needed shall be reevaluated as more experience is gained with the sensor network.
In order for the service to recognize the client user the client must obtain an authentication token/certificate from a Certification Authority that the service trusts. In this case, as shown in Figure 10, the client queries the Enterprise A Authenticate service with a username and password for the client user. The message to the Authenticate service is encrypted by the ACESOA/Axis2 Rampart infrastructure so that the username and password are not easily seen on the network.
Figure 10 Client gets Authentication Token

Figure 11 shows the Authenticate service returning an authentication token/certificate after it has verified that the username and password provided by the client is valid.

Figure 11 Client Authenticate Token Return
The client, using the WSDL information obtained about the service from the ASD, composes and sends a data request to the service. The ACESOA/Axis2 infrastructure handles signing the message (authentication info) and encrypting the message before it is sent to the service. See Figure 12.

Figure 12 Client Request of Data from Service

Figure 13 Service Authentication Check
The ACESOA/Axis2 infrastructure checks the authentication token/certificate (signature) of the message to determine if it is authentic and from a trusted Certificate Authority. First the service infrastructure looks to see if the information for this user is already in the service authentication cache. If it is not then its sends a request to the ACESOA Authenticate service to verify the users certificate/token. See Figure 13.

**Figure 14 Verified Authentication**

If the authentication token/certificate that the “Sensor Proxy” sent to the Authenticate service was valid then the Authenticate service returns an “OK” to the “Sensor Proxy” service. To save time in the future the ACESOA infrastructure saves the client user’s authentication token/certificate in a local authentication cache so that the Authenticate service will not have to be queried for the next message from the same user. The authentication entry in the cache is given a timeout period. The entry in the cache is removed if the timeout occurs.

Note shown in Figure 14 is that the ACESOA infrastructure for the “Sensor Proxy” service subscribes to the Authenticate service so that the “Sensor Proxy” service is notified if the trustworthiness of a Certificate Authority changes.
Figure 15 Service Authorization of User

In Figure 15 the ACESOA is querying the Authorize service to determine if the user that has just been authenticated is authorized to invoke the procedure specified in the message. To save time the infrastructure provides an authorization cache. The authorization cache is first examined for a certificate/credential for the client user before trying the Authorize service.

Figure 16 Authorization Certificate/Credential
An authorization certificate/credential is returned by the Authorize service to the requesting service as shown in Figure 16. This certificate/credential is stored in a local cache by the ACESOA infrastructure so that the Authorize service will not have to be queried for the next message from the same client user. A timeout period is set for the cache entry. At the end of the timeout period the entry is removed from the cache.

Not shown in the figure is the subscription of the “Sensor Proxy” service to the Authorize service by the ACESOA infrastructure so that the service can be notified of any change in the authorization certificate/credential of the client user.

In Figure 17 the remote service Sensor is the service that does the actual measurements so the Sensor Proxy must connect to the remote service. The Sensor Proxy uses the phone number it obtained from its configuration to call the remote sensor using its attached GSM phone (not shown).

The Sensor Proxy service sends the request message ‘getData’ to the remote Sensor service. The ACESOA/Axis2 infrastructure includes the Sensor Proxy user’s authentication token/certificate in the message and encrypts the message. This is shown in Figure 18.

The remote service compares the certificate authority in the message’s authentication certificate/token and compares it with the trusted CA in its local Authenticate file. If the CA is trusted then the remote service checks to see if the Sensor Proxy user is allowed to invoke the ‘getData’ procedure by evaluating the credentials in the local Authorize file. If the user is authorized then the remote Sensor service obtains the measurement data.

Figure 17 Remote Service Connection
The remote service returns the data as shown in Figure 19. The ACESOA infrastructure encrypts the message. The Sensor Proxy service then passes the data on to the client that originally requested the data.
The Sensor Proxy tears down the GSM connection once the message transaction has completed as shown in Figure 20.

![Diagram](image)

**Figure 20 Remote Service Connection Termination**

In Figure 21 the measurement data is finally returned to the client.

![Diagram](image)

**Figure 21 Client Data Return**
4.2 ACESOA Federation Example

In this example a client in Enterprise B wants to query a Sensor DB service in Enterprise A. The majority of the interactions of clients and services are shown in this example. The following interactions are described:

- Inter-Enterprise authentication trust.
- Inter-Enterprise service publication subscription
- Service publication
- Client search for a service via Service Directory (UDDI).
- Client user authentication.
- Client user authorization
- Client/Server communication

Figure 22 Federation Initial State

In Figure 22 Enterprise A has its ACE Service Directory, Authenticate and Authorize service running. It also has a remote sensor that only communicates with the Sensor Proxy Service. The sensor data from the Sensor Proxy service is manipulated by the Sensor Filter service to be more useable by sensor clients.

For this scenario the service Sensor DB start up is shown. The purpose of the service is to store sensor data in a database and provide the sensor database data to clients inside and
outside Enterprise A. It will obtain its sensor data from the Sensor Filter service although the “Sensor DB”-“Sensor Filter” interaction is not shown in the example.

At some point in the future the client in Enterprise B will want to obtain data from the database in the Sensor DB service of Enterprise A.

![Diagram of Federation Authentication Trust](image)

**Figure 23 Federation Authentication Trust**

In order for a client in Enterprise B to authenticate itself to Enterprise A there must be a trust relationship between the Authenticate services of enterprise A and B (see Figure 23). This must be done by humans in the two enterprises by exchanging Certificate Authority signatures over a secure communication mechanism (example: encrypted and digitally signed email). In this example an administrator in Enterprise A takes the Certificate Authority signature that it has received from an administrator in Enterprise B and stores it in a database of trusted Certificate Authorities for the Authenticate service of Enterprise A.

In Figure 24 the ASD of Enterprise A is sending a subscription request to the ASD of Enterprise A. This subscription request tells the ASD of Enterprise A that the ASD of Enterprise B should be notified of any services that publish with Enterprise A’s ASD. This reduces internet traffic since clients query their own ASD to find services in other enterprises.

The ASD of Enterprise A may also subscribe to the ASD of enterprise B but that is now shown in this example.

An enterprise will want to keep some services private (intra-enterprise). In this case there would be two ADSs in the enterprise. One ASD would be for services that allow inter-
enterprise usage. The other ASD would be for service with intra-enterprise usage only (not shown).

Figure 24 Federation ASD subscription

In Figure 25 the Federation A service ‘Sensor DB’ has been started. The startup could have been initiated by a host computer booting up or a human could have started it manually.

Figure 25 Federation Sensor DB Service Startup
In Figure 26 the service Sensor DB publishes internet location information and interface to the public (inter-enterprise) ASD so that clients outside Enterprise A are able to find it.

![Diagram showing Federation Service Publication]

In Figure 27 the ASD of Enterprise A has received the publication of the Sensor DB service. For each ASD that has subscribed to it, Enterprise A’s ASD publishes the information for the Sensor DB service. In this example it publishes the service information to Enterprise B’s ASD since it had previously subscribed to Enterprise A’s ASD.

Although not shown in the figures, service Sensor DB queries the ASD of Enterprise A for the Sensor Filter service. The ASD returns the location and interface information for the Sensor Filter service and the Sensor DB service connects as a client to the Sensor Filter service to obtain sensor readings to store in the Sensor DB database.
The client in Enterprise B wants to find and query the Sensor DB service in Enterprise A as shown in Figure 28. The client sends a service query request to its Enterprise B ASD looking for services that implement the Sensor DB interface.
In response to the client's query for service information, Enterprise B’s ASD returns the location URL and interface information for the Sensor DB service that it knows about in Enterprise A. This is illustrated in Figure 29.

Figure 29 Federation Client Query Service Information

The client must authenticate itself with the Sensor DB service in Enterprise A. To do this it must have a certificate or token that will be accepted by Enterprise A. Since the trust relationship has been established between the Authenticate services of Enterprise A and Enterprise B, the client can request its certificate/token from its Authenticate service as shown in Figure 30. In the token request to the Authenticate service, the client provides a username and encrypted password for identification of the client user.
The Authenticate service, once it verifies the validity of the username and password, returns a certificate/token to the client to use to authenticate itself with services. This is shown in Figure 31.

As shown in Figure 32, using the network location information returned by the ASD, the client creates a message containing the location URL and name of the service and adds the procedure request ‘getData’. The authenticate token is automatically added to the
message and the header and body of the message are encrypted by the ACESOA infrastructure and Apache Rampart Axis2 module.

Figure 32 Federation Client Request of Service

The service Sensor DB has received the ‘getData’ request from the client as shown in Figure 33. The ACESOA infrastructure first checks the local authentication cache in the service to see if the client user is already known, if the user is not known then the Enterprise A Authenticate service is queried to authenticate the client user by checking the user’s token/certificate to see if it was issued by a trusted Certificate Authority.
For this scenario, as shown in Figure 34, the Authenticate service tells the Sensor DB service that the user is really who he/she claims to be. The Sensor DB stores the user’s authentication information in a cache to be used for future queries by the same user to save time by skipping the authentication with the Authenticate service. The entry in the cache is given a limited lifetime and then removed from the cache at the end of the lifetime.

Not shown in the figure is that the Sensor DB has subscribed with the Authenticate service to be notified if a Certificate Authority is no longer trusted. When this notification happens all certificates/tokens cached by the service that were issued by the no longer trusted CA are removed from the cache.
Figure 34 Federation Authenticate OK Result

Once the client user has been authenticated the service has to determine if the user is allowed to invoke the requested operation ‘getData’. The Sensor DB sends a query to the Authorize service to get the authorization credentials/certificate of the user if the service does not already have the credential in a local credential cache. This request is shown in Figure 35.

Note shown in the figure is that the Sensor DB has subscribed to the Authorize service to be notified when credentials change.

The credentials in the Authorize service have to have been setup by an administrator some time in the past.
Figure 35 Federation Get Authorization

Figure 36 shows the returned of a user’s authorization certificate/credential by the Authorize service to the Sensor DB service. The Sensor DB service caches the certificate/credential to use with future requests by the users until the cache entry times-out or the Authorize service notifies the service of a change in authorization certificate/credential.

Figure 36 Federation Returned Authorization
The final act in a client from Enterprise B requesting data from a service in Enterprise A is for the service to return the requested data as shown in Figure 37. The ACESOA/Axis2 infrastructure encrypts the return message and sends it to the client then it is decrypted and given to the client code.

![Figure 37 Federation Service Data Return to Client](image)

### 4.3 Event Notification Examples

The following examples illustrate the messaging that occurs to subscribe to, publish and consume Event Notifications. Note that in these examples it is assumed that service discovery, authentication and authorization are occurring but is not shown.

The WS-Event specification implementation used in the ACE environment is Savan. Savan provides and API for clients to send subscription messages to an eventing service. Savan intercepts subscription messages coming into an eventing service so not external API is needed to receive subscriptions. Services use a Savan API to publish events. There is no API to receive published event notification messages.

#### 4.3.1 Service – Service Notification

In Figure 38 the messages involved with having one service subscribe to and be the consumer of Event Notifications of another service are shown. This is the simplest Event Notification scenario.

Message 1 is the Sensor Filter Service using the client Savan API to send an Event Notification subscription to the service Sensor. Included in the subscription is a filter to determine which events the consumer is interested in. Also included is the ReplyTo: field.
which is the URI of the endpoint to send Event Notifications to for this subscription. In this case the ReplyTo is the Sensor Filter service endpoint.

The Savan WS-Eventing Axis2 module intercepts the subscription in the Sensor service and stores the subscription information (hidden from the service Sensor business implementation).

When an event occurs (message 2 originating from some other source) the service Sensor business implementation places the event data into a data binding (class) and then calls the Savan PublishEvent API with the event data. The Savan implementation then handles checking each subscription filter to determine which consumers are to be sent Event Notifications. The subscriptions that pass the filter check then have the event data sent (published) to the ReplyTo: found in the subscriptions (message 3 in Figure 38).

Sensor Filter receives the event notification as a ‘normal’ operation message.

![Figure 38 Service - Service Event Notification](image)

### 4.3.2 Client - Service – Service Notification

This scenario (Figure 39) involves a client subscribing to events in a service on behalf of another service. This is a common scenario where a user is using a client application to tell a service where to send Event Notifications.

In this case the eventSubscription message ReplyTo field from the client contains the URI of the Sensor Filter service. See section 4.3.1 for more details of the messages and the implementation tasks of the remainder of the messaging.

![Figure 39 Client - Service - Client Notification](image)
4.3.3 Client – Service Notification

In the scenario shown in Figure 40 the desire is to have Event Notifications delivered to a client. Clients can not receive notifications directly (as discussed in section 2.1.3). This restriction requires a service to be embedded in the client to receive the notifications. The embedded service then passes the notification data to the client.

In order for a service to run it must be handled by a server. In this scenario the client starts an embedded server (1: in the figure), specifying the name of a desired service to start. The server then starts the specified embedded service.

The client registers a callback class (usually itself) with the embedded service to be used when a notification is received by the embedded service (2: in the figure).

Messages 3:, 4: and 5: in the figure are the same as described in section 4.3.2 except that the ReplyTo: field of the subscription message is the URI endpoint of the embedded service. The client obtains the embedded service endpoint from from the server.

When the Embedded Service receives the notification it looks up the callback class (previously registered by the client) associated with the type of notification data received. It then calls the callback method of the callback class with the notification data as the argument (6: in the figure). If the callback class was the client class then the client directly receives the notification data.

Figure 40 Client - Server Notification