16-03-2012

Deliverable DJ3.3.3
Composable Network Services: GEMBus Developments

Deliverable DJ3.3.3 v1.0

Contractual Date:	31-01-2012
Actual Date:	16-03-2012
Grant Agreement No.:	238875
Activity:	JRA3
Task Item:	T3
Nature of Deliverable:	R
Dissemination Level:	PU
Lead Partner:	RedIRIS
Document Code:	GN3-12-003
Authors:	Pedro Martinez-Julia (UMU.ES), Yuri Demchenko (UvA), Mary Grammatikou (GRNET), Roland Hedberg (UMU.SE), Jordi Jofre (i2CAT), Steluta Gheorghiu (i2CAT), Constantinos Marinos (GRNET), Stella Kafetzoglou (GRNET) Antonio-David Pérez-Morales (RedIRIS), Elena Torroglosa (UMU.ES), Marcin Dębowiak (PSNC), Łukasz Dolata (PSNC), Krzysztof Dombek (PSNC), Maja Gorecka-Wolniewicz (UMK-PSNC), Tomasz Wolniewicz (UMK-PSNC), Bartłomiej Idzikowski (PSNC), Maciej Glowiak (PSNC)

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7 2007-2013) under Grant Agreement No. 238875 (GÉANT).

Abstract

The aim of GEMBus is to enable collaboration between networks, share services and facilitate composition of more complex ones, establishing seamless access to the network infrastructure and services. This deliverable presents the framework for GEMBus, the GN3 federated, multi-domain service- bus. It details infrastructure developments, as well as how different services are integrated with GEMBus.

Table of Contents

Table of Figures

Executive Summary

This document presents the consolidation of the framework and general architecture for GEMBus (GÉANT Multi-Domain Bus), the federated, multi-domain, service bus infrastructure being developed as part of the GN3 project. The aim of GEMBus is to enable collaboration between networks, share services and facilitate service composition by establishing seamless access to the network infrastructure and services. The GEMBus architecture follows Service-Oriented Architecture (SOA) principles, which allows managing, maintaining, and accessing heterogeneous and distributed resources in a unified way by providing standardised interfaces and common working environments to their users. The GEMBus architecture has been designed to offer and support service composition (inherited by the SOA model), that is, the ability for an application to be turned into a reusable component. For example, GEMBus could be used to create a new service that automatically obtains digital certificates (by automatically contacting a certification authority), feeds them to a tool that generates scripts to configure users' devices (i.e. mobile phones) and makes the script available to download from a website. The alternative would be to provide a link to the Certification Authority (CA) that issues the server certificate and provides configuration and verification via the portal.

GEMBus is also aligned with the industry adopted Enterprise Service Bus (ESB) concept, which is extended to support dynamically reconfigurable and virtualised services as a general service bus architecture.

Most current ESB frameworks are oriented to single-enterprise deployments, which rely on a central, top administrative authority. GEMBus aims to bring the advantages of these frameworks to an open, collaborative environment, resulting in a further step towards the federation of infrastructures and the definition of a multi-domain service bus infrastructure, a "bus of buses".

GEMBus provides elements that maintain interoperability services for location, security, messaging and composition. These components form the core of GEMBus; they provide support to services participating in GEMBus through their whole lifecycle.

Lifecycle management is an important part of the Composable Service Architecture (CSA) and is important to the underlying design and operation of GEMBus. It is the basis for CSA provisioning and delivery service, incorporating service request, composition, deployment, operation, and decommissioning stages.

Service integration is a complex process. One of the objectives of GEMBus is to ease the integration of existing service platforms in the GÉANT network infrastructure and user communities. This document describes the service integration platforms identified by the team while defining aspects of GEMBus's architecture pertaining to service connection and support of dynamically reconfigurable and virtualised services.

1 Introduction

This document presents the ongoing results of the GEMBus (GÉANT Multi-Domain Bus) development. GEMBus is aligned with the industry-adopted Enterprise Service Bus (ESB) concept, which is extended to support dynamically reconfigurable and virtualised services as a general service bus architecture.

GEMBus is not a newly developed ESB; it builds on the assumption that each (GÉANT) domain may use a preferred ESB platform, or any other bus-type service communication or messaging environment, and that GEMBus would act as the link between these buses. In other words, ***GEMBus would 'federate' the buses***.

The bus paradigm provides the additional advantage of freeing service developers from dealing with common aspects such as authentication, authorisation, accounting, service discovery, and message management when developing their services. This enables them to concentrate on the direct implementation of business processes. Most (if not all) current frameworks are oriented to single enterprise deployments that, although complex, rely on a central top administrative authority.

GEMBus intends to bring the advantages of these frameworks into an open and collaborative environment, which will federate infrastructures and support the definition of a multi-domain infrastructure, a "bus of buses". Moreover, the GEMBus architecture also addresses multi-domain issues, distributed services composition and orchestration.

The experience of the research community when deploying federated architectures dictates strict adhesion to simplicity as the topmost design goal, to ease integration of disparate participant infrastructures and to facilitate interoperation at a common and agreed level. As a result, there are few requirements for an infrastructure to become part of GEMBus. Most interoperation mechanisms are regarded as end-to-end issues, although GEMBus is committed to provide mediation services for location, authentication, authorisation, accounting, and composition. The components that facilitate these mediation services are referred to as the GEMBus core. They are intended to provide support to participating GEMBus services throughout their lifecycle.

Service integration can be complicated and time-consuming. This document consolidates the definition of the integration patterns, starting with Deliverable 3.3.2*, Composable Network Services Framework and General Architecture: GEMBus*, and including those identified by the GEMBus team during experiments to establish the core components of the architecture (presented in this document). Discussion is also provided on the interface mechanisms that GEMBus offers to any application or computing element willing to make use of its services.

The document also provides design suggestions and describes an initial setup of the joint GEMBus/ESB technology testbed planned for the GN3 community.

2 GEMBus Architecture

2.1 Overview and Components

The GEMBus architecture follows **SOA principles,** namely:

- SOA services are independently managed and communicated via well-defined messages, typically using Simple Object Access Protocol (SOAP) or REpresentational State Transfer (REST) protocols.
- Each bus in a domain maintains a local registry that lists the services available in that domain.
- Each service is described in the local registry using a well-defined standard (typically one that is XML-based).
- Each service wishing to be accessed via GEMBus should comply with a set of minimum requirements that define the service repository.
- Client applications can request a service as a whole or just some of the functionalities offered by the service.

Following the evaluation of available ESB platforms (made at the start of the project, and detailed in deliverable DJ3.3.1), GEMBus selected FUSE [FUSE] as the preferred ESB platform for GEMBus implementation. The GEMBus architecture, described below, is therefore being developed using the FUSE platform extended by necessary messaging infrastructure configuration profiles for inter-domain GEMBus/ESB communication.

In order to provide the best support to a multi-domain services federation, GEMBus will include the following set of core components:

- **Federated Service Registry**: To talk to the local registries and announce the services available globally, allowing the ability to locate and obtain additional information about the services.
  - *Status*: Developed, in beta-version.
- **Service Repository**: To store service bundles[1] allowing their deployment via GEMBus. The GEMBus Service Repository will be accessible via a Web Interface, shell console and RESTful interface.
  - *Status*: Under development.

1 'Service bundles' refers to a group of services, typically in Open Grid Services Infrastructure (OGSi) format, but in this context it can mean any group of services.

- **Security Token Service***:* **(STS)***:* Is built as a WS-Trust (Web Services, Trust extension) implementation, issues, verifies and translates security tokens to allow the authentication of requesters in a federated, multi-domain environment. Requesters can use these tokens to request access to a service; in turn, the service checks the validity of the token before granting access to the requester.
  - *Status*: Basic functionality, extensions and improvements being worked on. A demonstrator is also available.

- **Composition Service**: To enable composition of services. This can be offered as a centralised service via the orchestration engine that is typically part of an ESB, or as an on-demand, deployed service, by downloading and deploying the necessary components locally using an OSGi service management framework/standard.
  - *Status*: Demonstrator available.

- **Accounting Service**: This service provides configurable and aggregated access to the GEMBus log-in service to support monitoring, auditing, diagnostics and troubleshooting.
  - *Status*: Under development.

Figure 2.1 provides a high-level view of the GEMBus core components described above.

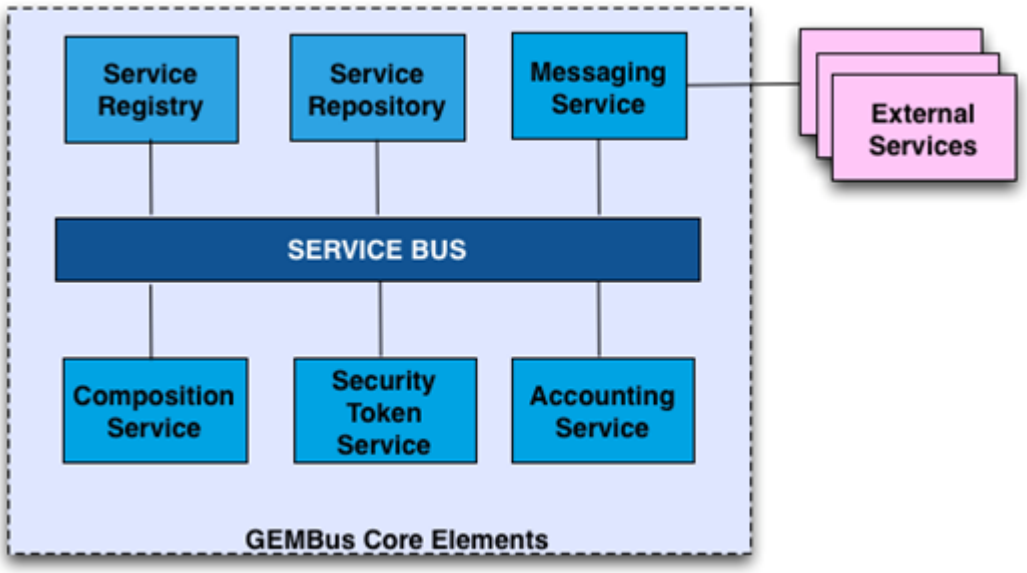

Figure 2.1: GEMBus core components

2.2 How Does GÉANT Benefit from GEMBus?

GEMBus provides an opportunity to access existing GÉANT services in a unified way and to create new, on-demand, customised services for user-specific needs or projects. Using standard interfaces and interactive service composition will allow users to focus on service functionalities without the need for details of service implementation and programming.

An added value of GEMBus is that users will be able to integrate GÉANT services with their commonly used workflows.

The well-structured way of interacting with services proposed by GEMBus will simplify the use of the core GÉANT services (security, accounting, monitoring, etc.) by any resource deployed in GÉANT.

2.3 Why Should You Connect Your Service to GEMBus?

In the past, when developing a service, developers needed to consider every aspect of service delivery, from instantiation in a server, to the mechanisms employed by a client to use it (including specific APIs, client libraries, and specific protocols for that service). It was also difficult to integrate the service with any other service in order to enhance the offered functionality and derive what is known as a "mashup".

Deploying a service in GEMBus means that the service does not need to include mechanisms to deal with specific instantiation aspects, such as the different protocols used by service consumers. It will also automatically be available for integration with other services (composition of services), but always be subject to the security constraints set by service administrators. Moreover, the service deployed via GEMBus gains federation capabilities, so services deployed in other domains are able to consume or be composed with that service.

Finally, inside the GÉANT community, when a service is deployed in GEMBus, it may take advantage of consuming, being consumed by, or being composed with any other service deployed in GEMBus. To illustrate its functionality, the library services (book catalogue and reservation) of a number of organisations may be plugged into GEMBus ‘as they are’, and then combined to build a larger and federated service with integrated authentication/authorisation, accounting and logging. It does not matter if the services ‘talk’ via different protocols, the framework will adapt them. Also, the resulting (composite) service may be extended with other services to offer PDF files of the books, comments or reviews from other users, etc. Another ‘real world’ example can be found in F-Ticks, a service that ‘emits’ RADIUS logging messages obtained from eduroam events. This service can also be plugged into GEMBus and extended to any client that demands it, together with specific filters to select the desired events and apply the necessary authentication/authorisation procedures to the access. Everything is transparent to the service developer, and the services are easily plugged in because the framework has the necessary mechanisms to compose them, filter messages, apply security, etc.

In summary, GEMBus permits service developers to focus on the business of the service and let the platform (GEMBus) provide the other aspects such as security, federation, composition, etc. Hence, by connecting a service to GEMBus, the chances for that service to become more widely used would be increased.

3 Consolidation of GEMBus Core Services

The GEMBus Core Services are composed of those elements that provide the functionality required to maintain the federation infrastructure, allowing the participant SOA frameworks to interoperate, in accordance with GEMBus principles. This section updates the description of these elements as they consolidate their functionality, through the definition of initial prototypes, and demonstrates how they are deployed as part of GEMBus, and how they can be used by participating services from other activities and tasks of GN3.

There are two types of elements, combined to provide the functional elements described below according to the functionalities of the service frameworks connected to GEMBus:

- Core components that form the federation fabric, enforcing its requirements in regard to service definition and location, routing of requests/responses and security. These elements are implemented by specific software elements and by extending and profiling the service frameworks to be connected.

- A set of core services that provide direct support to any service to be deployed in GEMBus, such as the registry, accounting service, security service, or the composition service described below. These core services are invoked by the core elements as part of their functions. They can be called from the code implementing any service deployed in GEMBus. Furthermore, as any other service taking part in the infrastructure, they are suitable to be integrated within composite services.

In this section we cover the consolidation of the core services, in order to provide an updated view of the state of the GEMBus framework, as well as any advances in service consolidation. The section provides a detailed description of the Composition and Security services that are considered the main enabling components for inter-domain services composition and federation.

3.1 Composition

Composition (based on independent specifications that can be combined to provide more powerful capabilities) allows for the creation of well-defined, composable web services that support security and reliability. These services specify the behaviour of the services necessary to support higher-level functionality.

There are four main steps that a developer needs to take to implement the "Simple Web Service Composition" demo, which are detailed on the demo site [GEMBus].

3.1.1 Composition Engine Requirements

3.1.1.1 *Mandatory Components:*

First, and only for the first time, the composition engine must be activated on the GEMBus server issuing the following command from the **karaf** console:

```
karaf@root> features:install ode
```

Also, if the user wants to use the **HelloWorld** service provided by GEMBus, the corresponding bundle must be installed:

```
karaf@root> features:install examples-cxf-osgi
```

There are additional components that are useful for testing, debugging and process modelling. Further details on these components and on their installation, may be found in Appendix B.

3.1.2 Example Scenarios

As introduced in the previous section, the existing services are simply plugged into GEMBus in order to build composed services. The composition engine is the component that facilitates a user's (here, the service developer) potential building of composite services inside GEMBus.

Continuing with the library example introduced in Section 2.3, the user only needs to load the library services, security service, and accounting service to the composition UI to be connected through the provided tools. In this example, the requests coming from the outside of the service should be connected to the authentication service in order to know if the request is from a valid client or not. After passing the security filter, the messages are sent to the accounting service in order to log the operation. Finally, the request will iterate through the different library services and a 'join' filter is set to add the results from the independent libraries, thereby building the response that is sent back to the requester.

Another example, the F-Ticks integration, follows the same process, using the security service to validate the client request and specify the filters to select the required log messages.

3.2 Security

The GEMBus Security Services must provide mechanisms to ensure security, privacy and simplicity for the communication that takes place within GEMBus architecture. Summarising the design detailed in deliverable DJ3.3.2, *Composable Network Services Framework and General Architecture* [DJ3.3.2] the security service is based on principles established by the WS-Security and WS-Trust. Web Services Security [WSS] is a communication protocol that provides the means for applying security to web services. It is a member of the WS-* family of web service specifications and was published by OASIS [OASIS]. WS-Trust [WST] is a WS-*

specification and OASIS standard that provides extensions to WS-Security, specifically dealing with the issue, renewal and validation of security tokens, as well as how to establish, assess (the presence of) and broker trust relationships between participants in a secure message exchange. WS-Trust defines the concept of Security Token Service (STS), the formats of the messages used to request security tokens as well as the required mechanisms needed for the exchange.

The GEMBus architecture makes use of an STS concept to offer all these security features. The functionality is divided in two different elements. First, the Ticket Translation Service (TTS) is responsible for generating valid tokens in the architecture according to the received credentials. This must include the support of current (standardised) authentication methods, as well as methods incorporated in the future. Second, token validation is performed by the Authorisation Service (AS). The validation process can also be associated with more complex processes of authorisation that imply attribute request and check security policies. If the token is valid, the AS provides an affirmative answer to the service.

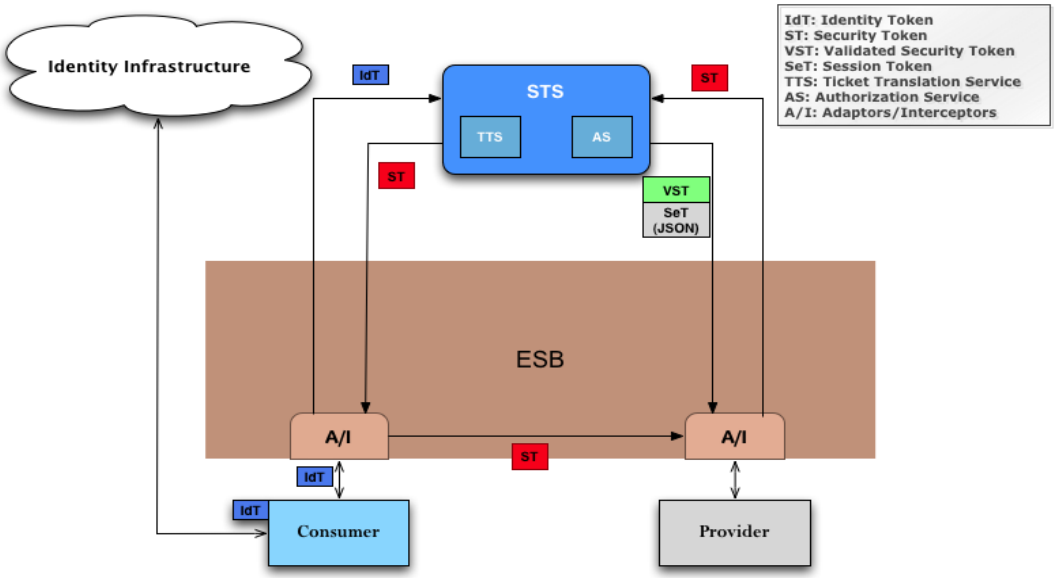

Figure 3.1: GEMBus integration scheme

Figure 3.1 illustrates a scenario in which a STS that has been extended with support for session tokens is integrated in the GEMBus architecture. In this example, the consumer obtains an identity token (a SAML assertion, for example) from an identity infrastructure. It then sends an authentication request to the STS using the identity token. The STS validates the consumer identity token and issues a Security Token (ST) to the consumer. With the new token, the consumer sends a request message to the provider that is intercepted by an element that extracts the ST and sends a token validation request to the STS. The AS module validates the consumer token and issues a response with a validated security token, together with an optional Session Token (SeT). Finally, the interceptor passes the message to the provider. It processes the consumer request and sends a response message to the consumer.

In addition to the flow described here, the service providers (SPs) deployed in GEMBus can validate the tokens by contacting the STS.

The STS concept is extended in the GEMBus architecture in order to provide additional functionalities. With the aim to improve the validation process, the STS is able to request attributes from external entities such as Attribute Authorities and Identity Providers. This new information could be used alongside of the client's in order to take an authorisation decision from Policy Points using eXtensible Access Control Markup Language (XACML).

3.2.1 GEMBus STS Design and Implementation

The architecture proposed by GEMBus is based on message exchanges performed by different services that can be connected in many ways. Since the ESB is the main integration mechanism provided by GEMBus, and because the ESB can also act as a service container, it is possible to develop and deploy a service directly on the bus. However, it is more interesting to exercise the integration capabilities of the ESB, such as interceptors, message routers and binding components. Whether deployed inside the bus or running as an external service, the STS can be used in a service composition to transparently provide its capabilities, using the aforementioned mechanisms.

The current version of the GEMBus STS is written in Java and designed to provide the basic components described above, as well as to offer points for possible extension in order to guarantee a future evolution of the GEMBus security services.

The software is divided in two parts:

- The WS-Trust application programming interface (API) library contains the WS-Trust model classes that represent the elements defined by the specification. Moreover, it also contains some utility classes to deal with WS-Trust requests and responses in a more simple way.

- The STS components, including a functional STS implementation. These components are extensible and it is easy to add new functionality or modify existing functionality of these components.

The STS does not issue tokens of a specific type. Instead, it defines generic interfaces that allow multiple tokens and providers to be plugged in. As a result, it can be configured to deal with various types of tokens, as long as a token provider exists for each token type.

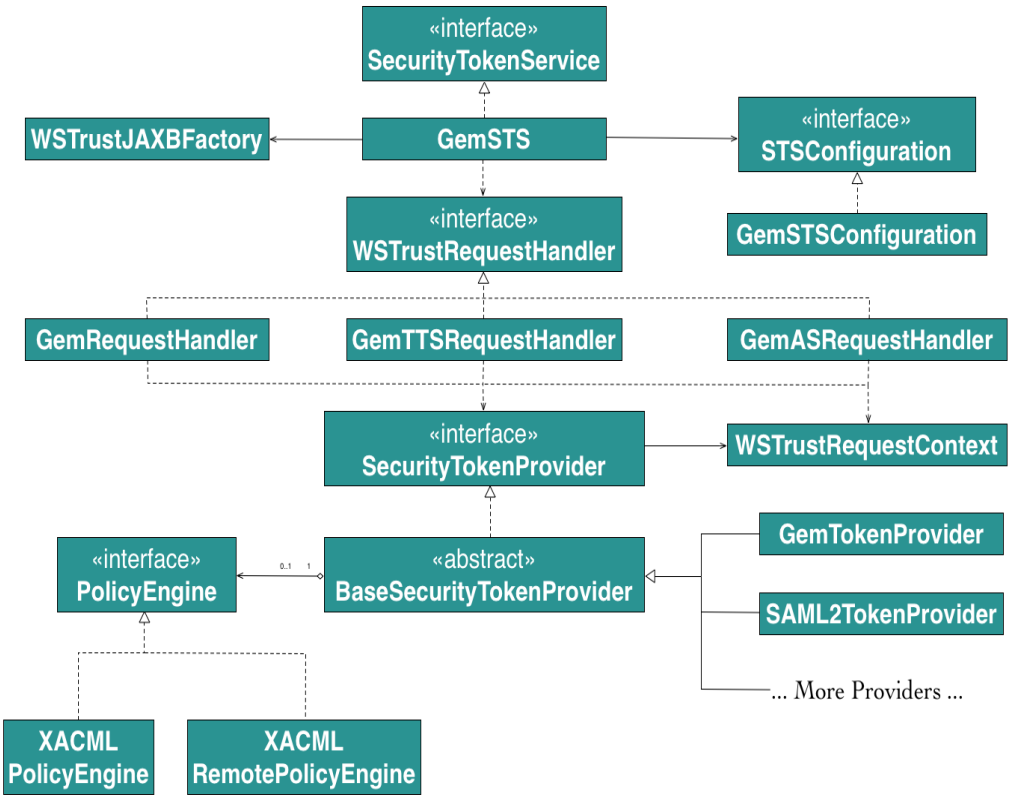

Figure 3.2: Class diagram of Security Token Service

As shown in Figure 3.2, the design is based on the use of interfaces to allow component extensibility. This way, the STS can be extended or modified by implementing these interfaces and changing its configuration.

The **GemSTS** is the STS web service, the component called by the clients who want to request, renew, cancel or validate a security token. It implements the **SecurityTokenService** interface, which in turn, extends the **javax.xml.ws.Provider** interface. The only method defined by the **SecurityTokenService** is the `invoke` method. This method takes a single parameter of type `Source`. The Source object allows for generic XML content to be transmitted to the Web Service. In this case, a WS-Trust request message.

The **WSTrustJAXBFactory** class is a utility class that is used by the STS to parse security token requests and marshal the security token responses. This class converts the XML request message and the Java object model, and then converts the Java objects to the XML response message that will be returned by the STS.

The **STSConfiguration** object is used to manage the configuration. This object is constructed by the **GemSTS** and contains all the configurations defined by the administrator. It contains information such as the default token TTL (time-to-live), the request handler, the token providers that can handle a specific token type, the identity providers upon which the STS relies, etc. The **STSConfiguration** interface provides a way to access this information. The concrete implementation of this interface contained in the library is the **GemSTSConfiguration** class.

The **WSTrustRequestHandler** manages all the business logic of the STS. Instances of this interface are responsible for actually handling the WS-Trust requests. When an STS receives a token request, it parses the request message and delegates the request handling to the **WSTrustRequestHandler** instance that has been configured. The handler uses either the STS configuration or another mechanism to find out which security token provider should be used to handle the token request and also to set values for properties that are absent in the WS-Trust request message. Specifically, the implementations of this interface contained in the STS library use the GEMBus registry to obtain the appropriate token type expected by a reliable service provider.

The library contains three implementations of this interface: **GemRequestHandler**, **GemTTSRequestHandler** and **GemASRequestHandler**. These handlers represent the three ways in which the **GemSTS** can be deployed:

- STS mode uses the **GemRequestHandler** and it is able to issue, renew, cancel or validate tokens.
- TTS mode uses the **GemTTSRequestHandler** and it is only able to issue, renew or cancel tokens.
- AS mode uses the **GemASRequestHandler** and it only validates security tokens.

The **WSTrustRequestContext** is a class that represents the security token request context. It contains all information that is relevant to processing the request. This class is used by the handlers and token providers to pass information between them, such as keys to use by the token providers, status of a token validation performed by a token provider, the security token issued by a token provider, etc.

The security token providers are responsible for handling the requests for a specific token type, using the information contained in the **WSTrustRequestContext** object. These providers implement the **SecurityTokenProvider** interface and they are plugged into the STS via configuration. Examples of token providers are **SAML2TokenProvider**, **GemTokenProvider**, **X509TokenProvider**, etc.

All providers must extend the **BaseSecurityTokenProvider** class, which in turn, implements the **SecurityTokenProvider** interface. This class supplies a policy engine to the providers, which can then use the policy engine to perform an authorisation in order to validate security tokens, either in validation requests (TTS) or before issuing new ones (AS).

As it has already been seen, the **PolicyEngine** interface is used to provide an authorisation mechanism. The **XACMLPolicyEngine** provides a XACML-based policy engine. It makes a XACML authorisation request to a XACML PDP (Policy Decision Point) using the information contained in the security token and it processes the result of that request. The **XACMLRemotePolicyEngine** is similar to the XACMLPolicyEngine, with a difference in that the latter makes a XACML authorisation request to a remote or external XACML PDP via SOAP. The policy engine used by each provider is set using the STS configuration.

3.2.2 Security Token Processing Example

Using the configuration shown in the previous section (with the **GemRequestHandler**), the processing of a security token issuance request (RST) is as follows:

1. A client sends a security token issuance request to **GemSTS**.

2. **GemSTS** parses the request message, generating a Java object model.

3. **GemSTS** reads the configuration file and creates the **STSConfiguration** object, if needed. Then it obtains a reference to the **WSTrustRequestHandler** from the configuration and delegates the request processing to the handler instance.

4. The **WSTrustRequestHandler** (**GemRequestHandler**) checks whether a security token exists in the request. This token represents the user on whose behalf the client is acting.

5. The **WSTrustRequestHandler** creates the **WSTrustRequestContext**, setting the relevant information from the request.

6. The **WSTrustRequestHandler** uses the **STSConfiguration** to get the **SecurityTokenProvider** that must be used to validate the token, according to the token type.

7. The **SecurityTokenProvider** instance processes the token validation and stores the validation status in the request context.

8. If the token validation is successful, the request handler uses the **STSConfiguration** to set default values when needed (for example, when the request does not specify a token lifetime value).

9. The **WSTrustRequestHandler** uses the **STSConfiguration** to get the **SecurityTokenProvider** that must be used to issue the new token based on the type of the token that is being requested. Then it invokes the provider, passing the **WSTrustRequestContext** as a parameter.

10. The **WSTrustRequestHandler** obtains the token from the context and constructs the WS-Trust response object containing the security token.

11. The **GemSTS** marshals the response generated by the request handler and returns it to the client.

The processing of a security token validation request is as follows:

1. A client sends a security token validation request to **GemSTS**.
2. **GemSTS** parses the request message, generating a Java object model.

3. **GemSTS** reads the configuration file and creates the **STSConfiguration** object, if needed. Then it obtains a reference to the **WSTrustRequestHandler** from the configuration and delegates the request processing to the handler instance.

4. The **WSTrustRequestHandler** creates the **WSTrustRequestContext**, setting the relevant information from the request.

5. The **WSTrustRequestHandler** uses the **STSConfiguration** to get the **SecurityTokenProvider** that must be used to validate the token based on the type of the token contained in the request. Then it invokes the provider, passing the **WSTrustRequestContext** as a parameter.

6. The **SecurityTokenProvider** instance processes the validation request and stores the validation status token in the request context.

7. The **WSTrustRequestHandler** obtains the status from the context and constructs the WS-Trust response object containing the status response.

8. The **GemSTS** marshals the response generated by the request handler and returns it to the client.

It is important to note that many different entities can act as clients to **GemSTS**. A client could be a web-service client that needs to obtain or renew a security token in order to access the service, but it could also be the web service itself trying to validate or cancel a token it has received.

The processes described in both the STS issue request and a security token validation request are illustrated in the following diagrams.

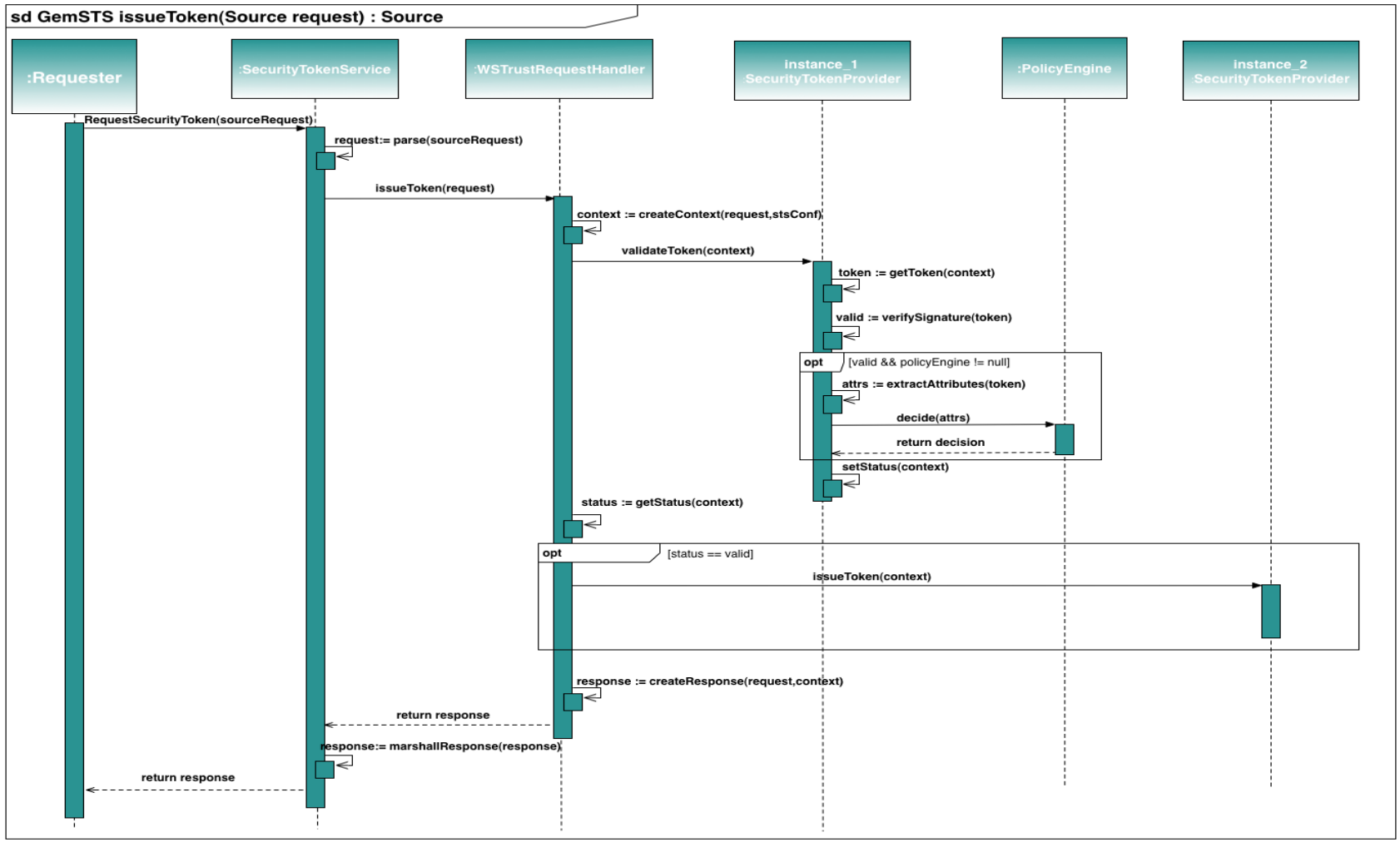

Figure 3.3: Security token validation request

3.2.3 STS Extension Points

The design of the STS defines several interfaces that provide extension points. Alternative implementations of these interfaces can be plugged in to the STS via configuration. The points where the STS can be extended are:

- **STSConfiguration**: This interface permits the STS to deal with other configuration files that do not follow the structure described in the previous sections.
- **WSTrustRequestHandler**: The STS business logic can be modified while implementing this interface, for example, supporting additional request processing.
- **SecurityTokenProvider**: This interface allows new token providers to plug into the STS in order to support additional token types.
- **PolicyEngine**: This interface allows different policy engines. For example, it is possible to have a policy engine based on rules in a plain text file by implementing this interface.
- **KeyHolder**: The KeyHolder allows the STS and token providers to load keys from different sources. For example, it is possible to have a KeyHolder which obtains the keys from plain text files by implementing this interface.

3.2.4 OpenID Connect

OpenID Connect [OIC] is a suite of lightweight specifications that provide a framework for identity interactions via RESTful APIs. These specifications permit the building of authentication and authorisation schemes based on existing (and external) identity infrastructures and also use GEMBus to provide authentication and authorisation services to other applications and services compatible with OpenID Connect. GEMBus can be aligned and integrated with other service platforms without requiring implementing specific security services or adapters for them.

OpenID Connect builds on two existing technologies: OpenID and OAuth2. OpenID Connect combines the resource access management support from OAuth2 [OAuth2] with the distributed identity management built into OpenID. It also extends the combined functionality by adding some features derived from the SAML2 [SAML2] deployments in higher education and research organisations. The specifications are being developed by the OpenID foundation [OpenID] and the process has started to standardise the OpenId Connect via the Internet Engineering Task force [IETF].

Within the OIC framework, user identity information is regarded as a single resource among others managed by the authorisation framework defined by OAuth2 [OAuth2].

3.3 GEMBus Registry

The GEMBus registry, sometimes referred to as the Federated Service Registry, is a vital component in the GEMBus architecture. Its main role is to expose unified information about the services available via a GEMBus instance. Information stored in the registry is collected by talking to the local ESB registries or having the services register themselves. The registry then announces the information globally, providing other services with the ability to locate and obtain additional information about participating services.

The information model used for the registry must be very easy to extend/modify. Using RDF ontologies as the common way of describing the information model gives GEMBus this flexibility. The registry has therefore been designed around a Resource Description Framework [RDF] database. A NoSQL database was also considered, however, this would have resulted in the inability to reuse de facto standard ontologies

There is a layer surrounding the RDF triple store that can translate to/from RDF graphs and an object-based representation. This layer allows us to accept, as well as export, the information in many formats. Applications can push information to the registry, as either JSON [JSON] or RDF/XML and they can PULL information from the registry as JSON, RDF/XML or HTML. The interface to the outside world is a REST-based web server with a built-in SAML2 SP.

Note that when this document refers to the GEMBus Registry, it does not imply that only one instance of a registry is envisaged. It is most likely that there will be more than one. Specific application environment, organisations and/or National Research Networks might, for different reasons, decide to run their own instance of a registry. It is expected that these registries will also exchange information with each other.

Each registry is self-contained. There might be references in one registry to objects in another registry, but there is no support for distributing queries over several registries. Such a system could be built, however, this is outside of this project's remit.

3.3.1 The API

In order to facilitate the usage of the registry there must be a well-known application interface (API). This API is not primarily for human usage but used by applications/services to get access to the information, as well as to keep the information up-to-date.

3.3.1.1 *The REST Interface*

REpresentational State Transfer (REST) is a software architecture designed for use in distributed systems that is based on a set of rules that define how web standards, such as HTTP and URIs, are supposed to be used. It has been designed to facilitate simple interactions, using the following HTTP operations on objects stored in the Registry: GET (read), POST (create), PUT (update) and DELETE (delete).

GET (read)

GET is used to fetch information about an object, in effect, a 'read'. Depending on the content of the accept header of the query, this operation can return XML, JSON or HTML.

When used with the type of requested object defined, GET, will retrieve a listing of those objects:

- GET http://<host>/service gets you a list of all the services by identifier (URL).
- GET http://<host>/service/0123456789 will return information about a specific service.

POST (create)

POST is more like an order, and used to create an object in the registry. An RDF graph of the RDF/XML format is expected in the body of the call. A successful POST will result in a '201 Created' response.

PUT (update)

PUT is used to update the full content of the resource. Note that the object in the store will be exchanged for this updated information. Incremental updates are not supported. A RDF graph of the RDF/XML format is expected in the body of the call. A successful update will result in a "200 OK" response.

DELETE (delete)

DELETE will remove the object with the given identifier from the registry.

3.3.1.2 *The SPARQL Query Interface*

The GEMBus registry also supports SPARQL Protocol and RDF Query Language (SPARQL), a query language that can be used to search the registry for an object.

The format of a typical query would look like the following:

```
http://<host>[:<port>]/query?<sparql_query>
```

Note that the <sparql_query> part must be URL encoded. The response is delivered as raw SPARQLtext.
An example of a sparql_query is (formatted for readability):

```
   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
   PREFIX gembus: <http://geant.net/rdf/gembus#>
   PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>
   SELECT ?about ?gn ?sn
   WHERE {
        ?about rdf:type gembus:Contact .
        ?about vcard:n ?name .
        ?name vcard:given-name ?gn .
        ?name vcard:family-name ?sn .
}
```

A result to such a query could result in this (using an application/JSON representation).

[['http://localhost:8087/registry/contact/corky@example.com', 'Corky', 'Crystal'],
[http://localhost:8087/registry/contact/corky@example.com'), 'flo', 'Crystal']]

3.4 Accounting

In current networked architectures, accounting operations, which collect and record information for the relevant service calls, are implemented to provide necessary input for charging and billing systems. These charging and billing systems started with the use of simple models, such as time-based, volume-based or flat-pricing models [Kuhne11] and strove to provide the basic needs, such as completeness, correctness and security of the collected information, and accountability of the service user.

However, in today's dynamic, service-oriented environment, where scalability and reliability issues arise, convergent charging and billing offers new solutions that should also take into consideration the following desired features [Kuhne11]:

- Cost transparency, which refers to the premise that the user should always be informed about the cost of a service.
- Online charging, which requires real-time capabilities in order to support prepaid users without incurring a credit risk for the service provider.
- Easy introduction of new services, which implies a low cost of integrating new services, and is also important in terms of user acceptance.
- Synchronisation or better consolidation of charging processes, which refers to the case where the accounting data is generated at different places (for example, across several domains).
- Configurability, which refers to the possibility of controlling the way a user is charged for the consumed services, and the way to account for corresponding user activities.

From these prerequisites, we further derive the main characteristics of an efficient accounting system:

- Timely provisioning and processing the accounting data.
- Scalable infrastructure to allow fast, low-cost integration of new services.
- Cross-domain support to allow the aggregation of the accounting information for composed services.
- Management support to permit easy configuration of the recorded activities.

The accounting needs of computing services and networking resources have also been considered. The networking protocol Remote Authentication Dial In User Service (RADIUS) is an example for the networking domain that provides centralised authentication, authorisation, and accounting management for computers using a network [Rigney00]. Although the overall cost of an accounting and charging mechanism needs to be taken into account, the accounting aspect is limited and mainly focuses on the amount of time a networking connection is active and the amount of traffic created. In the area of grid computing, research has been targeted at the definition of a common usage record format. This work aims to allow a standardised exchange

of accounting information, and is driven by the Open Grid Forum (OGF). Two format recommendations [Götze] have emerged so far:

- The usage record format recommendation, focusing on grid-job-level usage accounting of computing resources, (i.e. CPU, storage) [Mach06].

- The aggregate accounting record format recommendation intended to wrap high-level accounting view information for cross-domain accounting in grid environments [Chen06].

Both standards solely address computing resources; additional resources may be added to an extension field without any standards. Further, more research has been done in order to extend usage accounting with regard to storage [Scibilia07] and software license [Mallman08], but a standard has yet to emerge.

However, these approaches deal with format-specific services, therefore a generic accounting solution that could handle arbitrary services and applications is missing. As a result, we propose a more flexible system for GEMBus that would provide usage accounting in any type of service-oriented environment. Apart from charging and billing, the information collected by GEMBus's accounting system could be useful also for trend analysis, or to assess future service usage such that a necessary enhancement of service capacities can take place before a bottleneck arises. In addition, such information could be used for auditing purposes, to verify the correct execution of processes by examining the individual operations of a process. Examples for such use are the cross-checking of the users accessing a service compared to the list of users that should have access to that service, or the detection of a violation of policies.

The accounting service solution for the service-oriented environment of GEMBus has to fulfil multiple requirements and must address the challenges derived from the heterogeneity and the complexity of the environment. Heterogeneity arises from the various existing types of services, e.g., database services, computing services, and the multitude of implementations available for each of them. Thus, new services or enhanced versions of existing ones should be integrated in GEMBus with the lowest cost possible and with a minimum amount of changes to the accounting infrastructure.

Apart from this issue, the relations with other accounting domains can lead to additional complexity. Users consuming a service might be part of the local ESB instance or originating from another ESB accounting domain. In this second case, the accounting service should enable the gathering of the information about complex services (resulting from the composition of other services). Therefore, the system must correctly identify the service being used and the consuming user. Being able to track such service interactions and being able to associate these interactions with the originating user, allows the recording of the accurate accounting information that would further ensure the appropriate charging and billing. Another typical problem for multi-domain environments is that different accounting systems possibly use the collected information in different ways; hence a common format of the accounting data is required [Bhushan01]. In this context, the GEMBus accounting service should be able to collect performance information and to forward it to a global GEMBus accounting infrastructure that would aggregate the records from the whole GEMBus platform. Moreover, only entitled personnel are allowed access to this information for further usage.

Taking into consideration all of the challenges identified above, we propose a GEMBus accounting solution that meets the following goals:

- Support cross-domain accounting, thus enabling GEMBus's multi-domain nature (to be resolved as part of a future service task).
- Support further enhancements of existing services and integration of new services into the accounting infrastructure.
- Ensure a scalable accounting infrastructure that is capable of providing its service without restrictions on the size of the service-oriented environment.
- Provide support for management of the accounting infrastructure, thus enabling the administrator to easily handle the GEMBus distributed system.
- Integration with a larger infrastructure, with multiple accounting domains (one per GEMBus ESB instance or domain).
- Exchange accounting information with other domains.
- Ensure the management of the distributed components of the accounting system to create a global GEMBus accounting system.
- Define a common format for the data records.

3.4.1 Implementation Aspects

Figure 3.4 illustrates the architecture of the accounting service for GEMBus

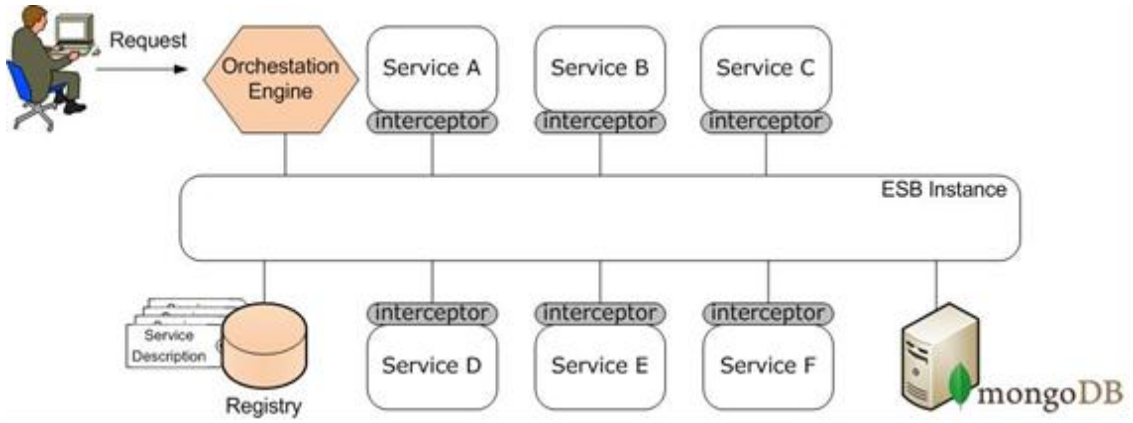

Figure 3.4: GEMBus accounting architecture

Apart from the existing core services of the GEMBus Registry and the Orchestration Engine, in order to provide accounting capabilities, we have selected the interceptor functionalities provided by the FUSE ESB implementation [CXFINTER]. The interceptors are Plain Old Java Objects (POJOs) that have access to the message data before it is passed to the application layer. Depending on the logical functions, the interceptors are grouped into **phases,** where each phase is responsible for a specific type of message processing. The phases are further aggregated into **chains**, as shown in the figure below.

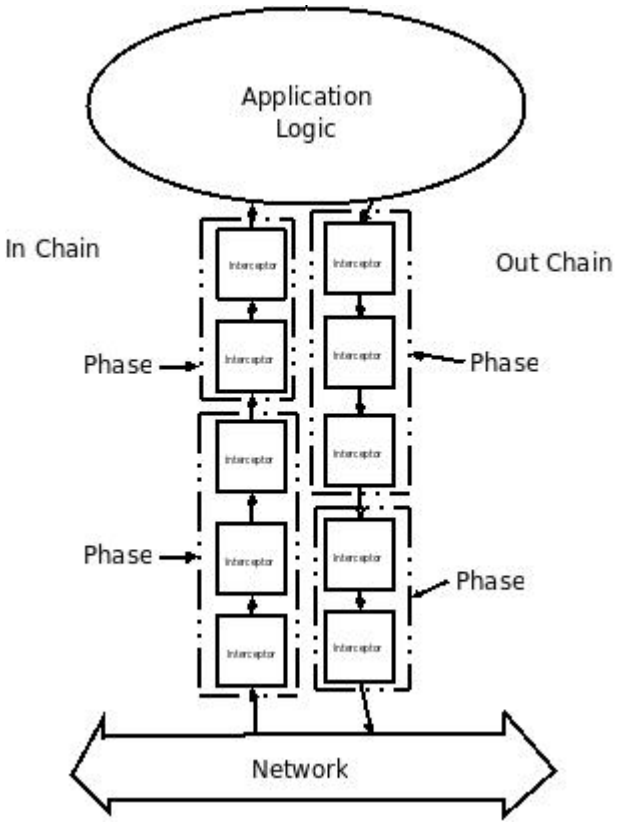

Figure 3.5: Apache CXF interceptor chains [ApacheInt]

The chains can be of three types, depending on the message they process: (a) **inbound** messages, (b) **outbound** messages, and (c) **error** messages.

In order to develop a new interceptor, one needs to specify the phase in which it will run, the chain to which it will belong, as well as to implement the interceptor's processing logic. Figure 3.6 illustrates the flow through an interceptor from the **handleMessage()** to the **handleDefault()** in case of an error.

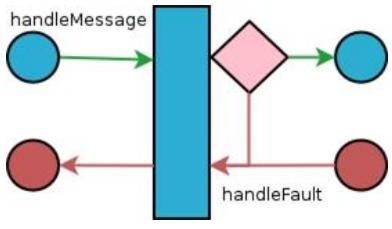

Figure 3.6: Flow through an interceptor [ApacheInt]

The processing logic of an interceptor message is placed in the **handleMessage()** method, which is called during normal message processing. If an error occurs during the execution of the interceptor's chain, the method **handleFault()** is called and cleans up any resources used by that interceptor.

For the proposed accounting system, we will add an interceptor at each service deployed in GEMBus. This interceptor should track both incoming and outgoing messages from the corresponding service. The interceptor could be included in the service container information, together with the security component, by adding the following lines to the beans configuration:

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-http-jetty.xml"
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-http binding.xml" />

<bean class="com.progress.pso.helloworld.HelloWorldImpl" id="helloWorldBean"

   <jaxws:endpoint              id="helloWorldWS"              implementor="#helloWorldBean"
address="http://0.0.0.0:8197/HelloWorldService/">
       <jaxws:inInterceptors>
         <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor" />
         <bean class="neat.geant.gembus.i2cat.accouting.InterceptorCXF"
       </jaxws:inInterceptors>

       <jaxws:outInterceptors>
         <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor" />
         <bean class="neat.geant.gembus.i2cat.accouting.InterceptorCXF" />
       </jaxws:outInterceptors>
   </jaxws:endpoint>
</beans>
```

The messages captured by these interceptors are the main source of information for the accounting service, which generates a data log of relevant information that can be extracted from GEMBus. These logs need to be stored and kept for further analysis at a later time. We have chosen to deploy MongoDB [MongoDB], an open source, high-performance, schema-free, document-oriented database to store GEMBus logs. MongoDB is available free under the GNU Affero General Public License. The language drivers are available under an Apache License. MongoDB has official drivers for many languages and there are also a large number of unofficial drivers. Furthermore, it supports master-slave replication, where a slave copies data from the master and that data can only be used for reads or backup (not writes). There is a monitoring plug-in available for MongoDB and finally, several GUIs have been created by MongoDB's developer community to help visualise their data. All of these features make MongoDB well suited to GEMBus requirements.

The data recorded for each log includes the following fields:

- *UserID*: can be obtained from the token information provided by the security core service.
- *Action*: the current request made to the service and is traceable in the *Action* field of the WS-Addressing specification.
- *ServiceID*: identifies the service that is being consumed. This information is requested by the GEMBus registry.
- *Timestamp*: represents the time when the service is consumed, obtained for the local time of the machine.

- *MessageID*: a unique identifier for the message transaction extracted from the MessageID field of the WS-Addressing specification.

- *From*: an identifier that indicates which service sent the request. This information is useful, especially when it is necessary to trace service interactions.

As mentioned above, the GEMBus accounting solution relies on the use of WS-Addressing [WSADDRESS], a standardised way of including message routing data within SOAP headers. Instead of relying on network-level transport to convey routing information, a message utilising WS-Addressing may contain its own dispatch metadata in a standardised SOAP header. The addressing information relating to the delivery of a message to a web service is communicated through the Message Addressing Properties, and includes the following:

- Message destination: URI.
- Source endpoint: the endpoint of the service that dispatched this message (EPR).
- Reply endpoint: the endpoint to which reply messages should be dispatched (EPR).
- Fault endpoint: the endpoint to which fault messages should be dispatched (EPR).
- Action: an action value indicating the semantics of the message (may assist with routing the message) URI.
- Unique message ID URI.
- Relationship to previous messages (A pair of URIs).

The use of WS-Addressing fulfils the requirements of the GEMBus accounting service and is also suitable because it is a standard specification. As a result, it will be important for services deployed in GEMBus to enable the use of WS-Addressing.

3.5 Messaging Infrastructure

The GEMBus messaging service is a common component of all GEMBus installations or hosting platforms, which is based on an ESB that provides all necessary functionalities for services interaction and integration in the SOA.

The GEMBus messaging infrastructure is built on top of the standard Apache/FUSE messaging infrastructure that includes the following components [ApacheActiveMQ], [ApacheCamel]:

- FUSE Message Broker (Apache ActiveMQ) **messaging processor**.
- FUSE Mediation Router (Apache Camel) **normalised message router**.

These two components provide all necessary functionality for GEMBus component services integration and interaction, including inter-domain/inter-ESB communication. Actual service interconnection is defined by configuring the Message Broker and Message Router, as illustrated in Figure 4.1 and described in Section 4, GEMBus/ESB Testbed.

4 GEMBus/ESB Testbed

To demonstrate the GEMBus/ESB basic functionalities and experiment with inter-domain inter-services messaging, the GEMBus/ESB testbed has been created at University of Amsterdam (UvA). The testbed is implemented as a Cloud Platform as a Service (PaaS), where user services are deployed in preconfigured virtual machine (VM) with a preinstalled GEMBus/ESB platform. Besides supporting GEMBus component services development and integration, the testbed aims to facilitate GEMBus dissemination and wider adoption among GÉANT and NREN users.

Figure 4.1 shows the testbed structure and implementation details. The lower-layer infrastructure uses OpenNebula VM management environment. Each VM can run either a GEMBus instance or a FUSE ESB [FUSEESB] instance that may host one or more services. Each VM with an installed GEMBus/ESB environment can be considered as a single domain, alternatively, a few VM can belong to one domain.

Service interconnections are realised based on such common GEMBus/ESB functional components as Message Broker (based on Apache ActiveMQ) and Message Router (based on Apache Camel). Component services can be deployed in a GEMBus/ESB environment using VMs with preinstalled and pre-configured GEMBus/ESB instances. Final services interconnection topology can be created by pre-configuring the Message Broker and the Message Router at each GEMBus/ESB instance or dynamically changing their configuration after deployment and during run-time, which is supported by GEMBus messaging infrastructure.

Communication between GEMBus domains is carried out either over the underlying transport network infrastructure or using dedicated network infrastructure, which is provisioned as Network as a Service (NaaS). In the latter case, NaaS can be controlled via a dedicated GEMBus service. Current testbed implementation uses only underlying transport network infrastructure.

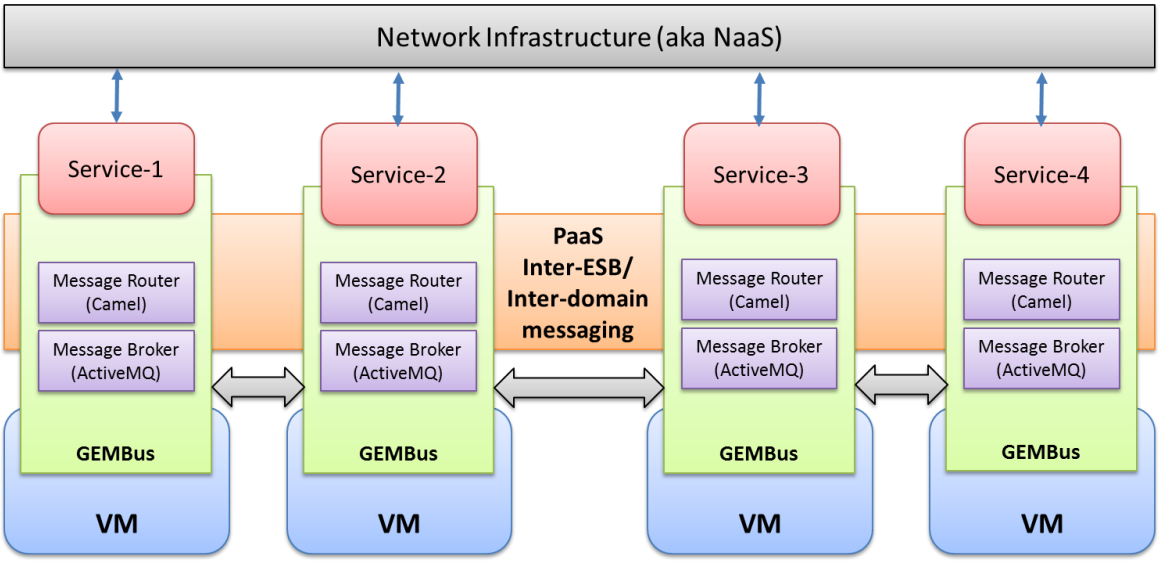

Figure 4.1: Testbed for GEMBus/ESB-based services composition (using Cloud PaaS service model)

Figure 4.2 below, illustrates topological relations between services in the testbed that includes inter-domain/inter-GEMBus signal and data links and control links between VMs and testbed PaaS infrastructure controller.

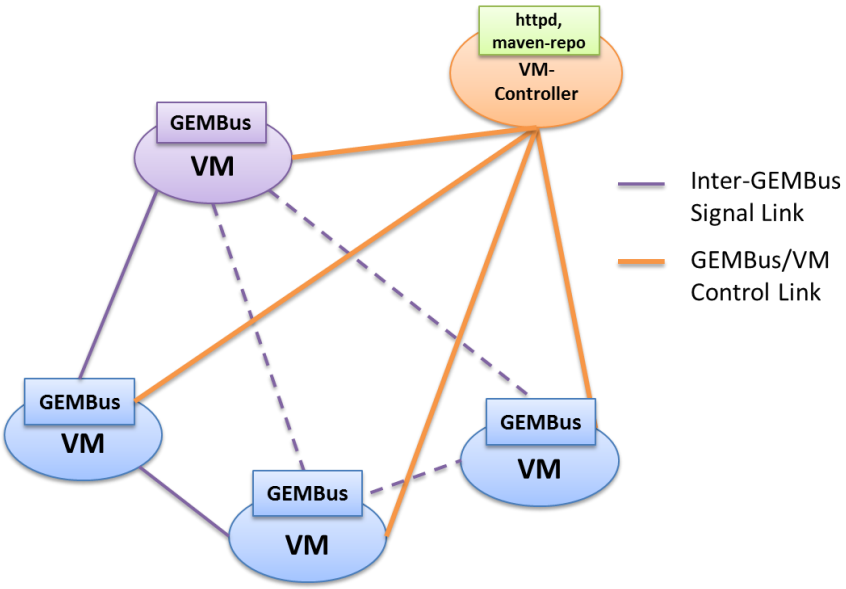

Figure 4.2: Testbed data and control interconnection topology

4.1 Example Demo Scenario (Multi-Domain User Services Deployment)

This section provides an example of the messaging infrastructure configuration used in the GEMBus testbed demonstration at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11).

Figure 4.3 presents services composition workflow for runtime demo using three source services generating samples of the time variable signal: rectangular pulse function, sinusoidal signal, and relaxation (or “saw”) generator. These basic services can be composed in different ways by using message routing function and processor service.

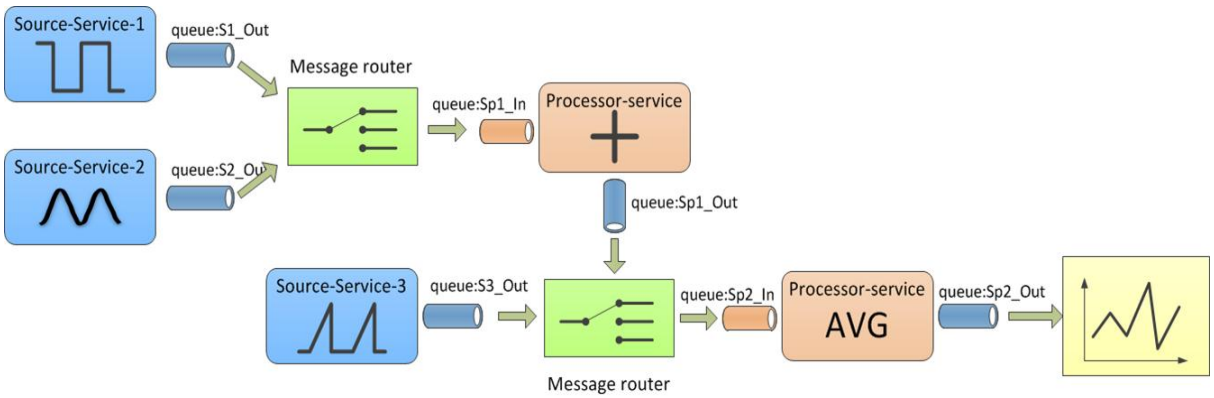

Figure 4.3: Demo scenario services composition

4.2 User Services Deployment

GEMBus user services can be deployed remotely by uploading a new service bundle to the **deployment** folder, which has a standard location **$GEMBus/deploy/.** Currently, this operation requires manual configuration, but in the future, it can be done with via a special user interface or client using either web browser or Eclipse client.

5 Service Characterisation and Deployment

One of the declared objectives of GEMBus is to ease the integration of existing service platforms, both in the GÉANT infrastructures and in the GÉANT user communities. The goals are to:

- Simplify the process of gaining access to those services by other user communities worldwide.
- Allow other services to leverage their resources to each other.
- Provide homogeneous mechanisms to ensure the evolution of the services.
- Simplify the integration with similar or related platforms in other spheres (commercial, governmental, etc.).

In order to achieve these goals, GEMBus worked on a common definition for service characterisation and thus let other services know how they should be built in order to fit within GEMBus. This was strengthened by the definition of reference services, as well as a brief guide to validate a service against GEMBus requirements.

5.1 Canonicalisation and Reference Services

5.1.1 Canonicalisation

Canonicalisation defines the procedure that has been followed in order to collect specifications for the different environments in which GEMBus is being installed. This data was collected through analytical surveys that were exchanged among the partners. Results showed that, in most cases, GEMBus has been installed in Linux-based servers. In the current version of GEMBus, the ESB layer is powered by the FUSE ESB and Tomcat with an AXIS SOAP [ApacheAXIS] container were used as application servers.

Surveys were also collected about the environment in which each partner was developing a service, a module or an interceptor. Results showed that the FUSE-Eclipse plug-in was used for the ESB development.

5.1.2 Reference Services

Reference services define a set of APIs that should be followed in order to be connected to GEMBus platform. Reference Services are the minimum pieces of code required in order to adapt external APIs for use on the GEMBus platform. This set of classes can be new modules, new interfaces, adaptors and also interceptors that

control the communication between GEMBus and an external user. *An external user could be anything from a single user to a multi-domain platform.*

As previously described, in order to support a multi-domain ESB/Bus federation, the GEMBus has developed the following core services:

- A **Federated Service Registry**, which is responsible for "talking" to the local registries and announcing the services available globally, allowing them to be located and obtaining additional information about the services.

- A **Service Repository**, which is responsible for storing service bundles, and allowing their deployment on the local instance of GEMBus.

- A **Security Token Service**, which is built as WS-Trust implementation, which verifies and translates security tokens to allow the authentication of requesters in federated multi-domain environment. Requesters can use these tokens to request access to a service; the service then checks the validity of the token before granting access to the requester.

- A **Composition Service** that enables the composition of services. It can be offered as a centralised service via the orchestration engine that is typically part of an ESB or as an on-demand deployed service, by downloading and deploying the necessary components.

- The **Accounting Service**, which is the service that provides configurable and aggregated access to the GEMBus logging service to support services such as monitoring, auditing, diagnostics and troubleshooting.

A developer will face a number of issues while trying to connect to each of these services during Reference Service implementation. If a new external service needs to be integrated with GEMBus, it should communicate with the corresponding service interceptor. There are two service interceptors, the messaging and the registry, which are prerequisite implementations for every new user/platform that wishes to connect to GEMBus.

Figure 5.1 details the different Service Interceptors connect to the corresponding Core Services and the Reference Container.

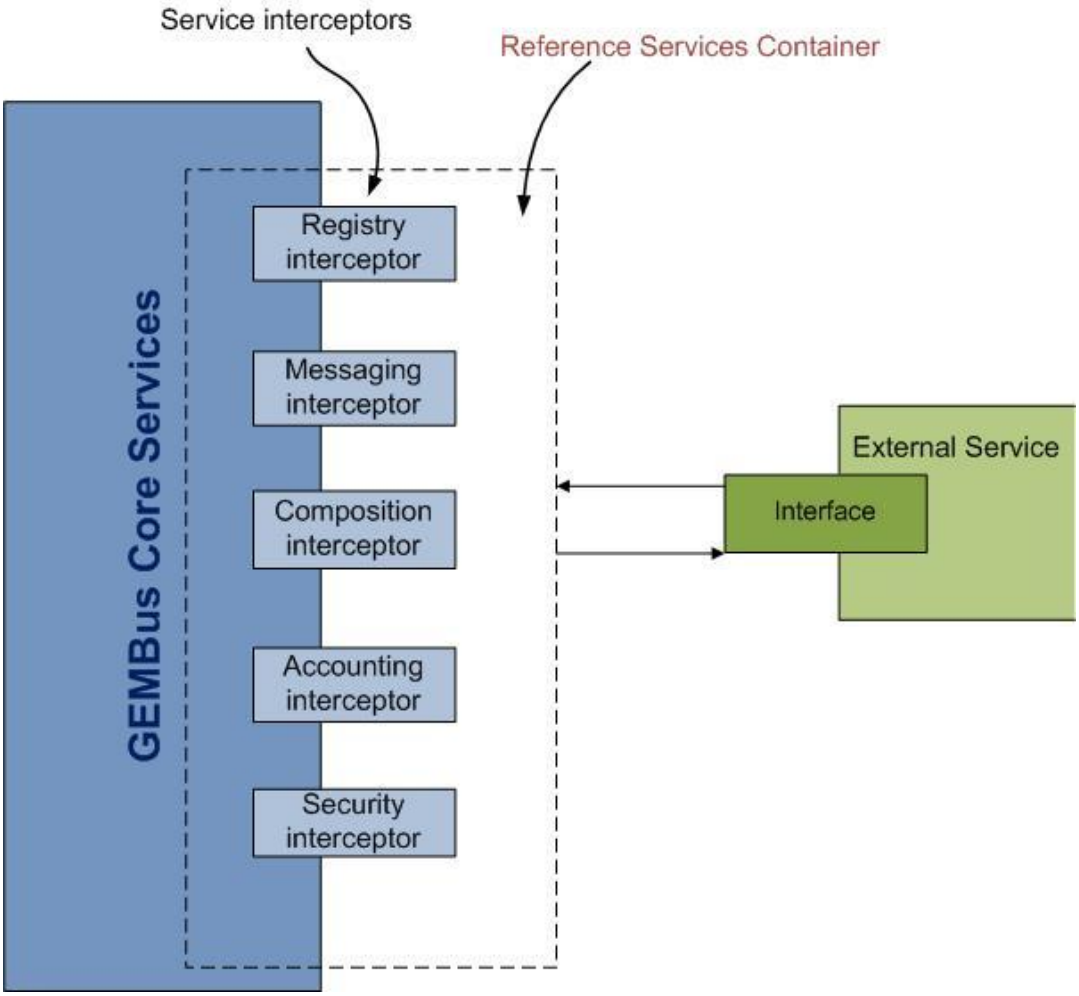

Figure 5.1: Service interceptor, core service and reference services container connections

5.2 Service Validation

The design and development of a complex architecture, as proposed in GEMBus, involves the use and experimentation with multiple tools for the implementation and testing of the newly proposed services. The division of tasks in different work areas involves different approaches to resolving problems. At this point it is helpful to pool the current approaches from all the developers in order to select the best mechanism with which to validate the services developed within the architecture.

GEMBus plans to compile and incorporate a knowledge base for sharing experiences and solutions from developing, debugging and validation tasks. In addition, this information is expected to serve as a source of information for future GEMBus developers.

For debugging and validation of services and the GEMBus architecture (in addition to previous design work), we have used different tools to monitor all levels: from the network layer (e.g. Wireshark) to the level of

communications between bus services (e.g. soapUI). All this work on tools and plug-ins will be compiled and published within the GEMBus cookbook, as well as be included as part of the final results of the project.

Although the central part of the work is focused on validation services, the idea is to use this knowledge sharing to produce guidelines on different development tools such as Eclipse or other integrated development environment (IDE), or on other tools such as the Eclipse plug-in for GEMBus, which is used by our developers.

5.3 Service Repositories

The GEMBus Service Repository is based on the OSGi Bundle Repository (OBR), which is a proposal for the specification internally referred to as RFC-0112 in the OSGi Alliance [OSGi Bundle Repository]. The FUSE ESB's Apache Felix OSGi Bundle Repository provides a service that can automatically install a bundle with its deployment dependencies from a bundle repository, thereby simplifying the use and deployment of available bundles. The OBR RFC-0112 proposed specification defines an XML format for repositories of OSGi bundles and an OSGi framework service to access and use a repository. The GEMBus Service Repository implementation has a modular architecture and contains the listed blocks:

- **Service repository daemon**: allows for easy start/stop/restart of all the modules and creates a Linux daemon as well as provide logging and configuration.
- **XML refresher (BIndex tool)**: module that uses the BIndex tool [OSGi Bindex] to periodically generate the XML description file.
- **Website interface**: module that serves the Web GUI interface.
- **REST interface**: module that serves the RESTful interface.
- **Repository storage**: module that stores the bundle binaries.

Figure 5.2 illustrates the architecture of the GEMBus Service Repository.

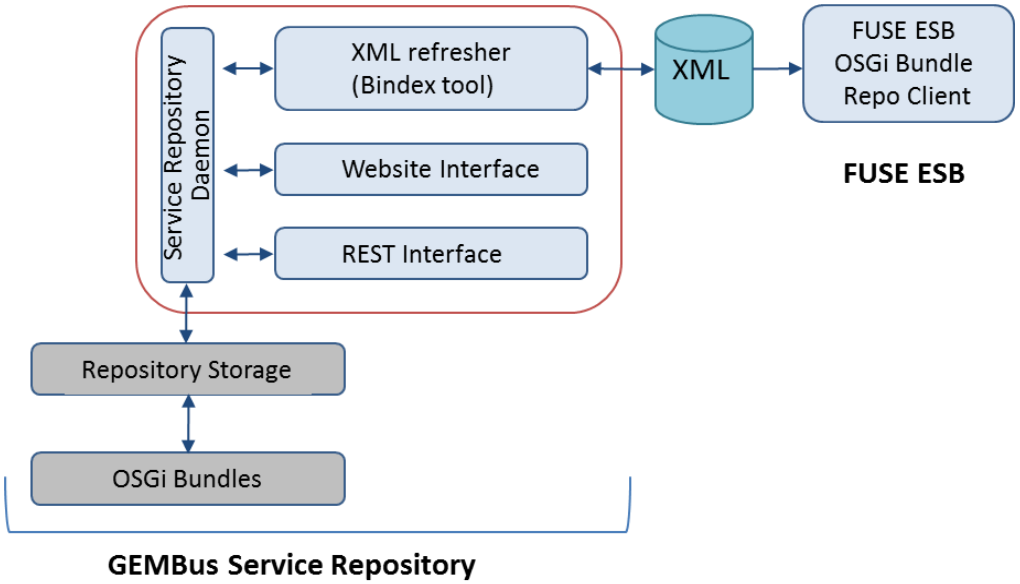

Figure 5.2: GEMBus Service Repository architecture

All the modules are implemented in Python, except from the BIndex tool, which has been included as a Java program.

The **OBR repository** file is an XML-based representation of bundle meta-data. It contains information of the provided capabilities and required dependencies. As previously mentioned, the detailed description of RFC-0112, the OBR meta-data format is available to download from the OSGi website [OSGi Bundle Repository].

A resource can provide any number of capabilities. A capability is a typed set of properties, such as:

```
<capability name='package'>
     <p n='package'  v='org.foo.bar'/>
     <p n='version'  t='version'    v='1.0.0'/>
</capability>
```

A requirement is a typed LDAP query over a set of capability properties

```
 <require   extend='false'  multiple='false'
    name='bundle'  optional='false'
    filter='(&amp;(symbolicname=perfsonarNMWGAdapter)(version&gt;=0.0.0))'>
Require Bundle perfsonarNMWGAdapter; 0.0.0
</require>
```

The **OBR XML description** file can be generated using the BIndex tool with a “-d” parameter and specified path to the already compiled bundle binaries. Below, you can see an example of such generating process.

```
# java -jar bindex.jar -d  /.m2/repository/net/geant/gembus/          # command
for generating the XML
# cat repository.xml                                          # content of the XML
<?xml version='1.0' encoding='utf-8'?>
...
<resource id='perfsonarCLMPAdapter/0.0.1.SNAPSHOT' ... >
    <description>
      Adapter for perfsonar command line measurement point services
    </description>
     ...
    <capability name='bundle'>
      <p n='manifestversion' v='2'/>
      <p n='presentationname' v='Perfsonar Command Line Measurement Point'/>
      <p n='symbolicname' v='perfsonarCLMPAdapter'/>
      <p n='version' t='version' v='0.0.1.SNAPSHOT'/>
    </capability>
...
```

The **GEMBus Service Repository** provides an XML Refresher module that automatically generates the XML description file to the current contents of the bundles in the repository.

The **FUSE ESB Kernel** console provides interactive access to OBR using the “obr” subshell commands. These commands can be used to deploy user bundles, following the creation of a bundle repository metadata file.

```
karaf@root> features:install obr                          # installing the new
feature - repository client
karaf@root> obr:addUrl file://esb/repository/repository.xml     # adding new url
to XML description file
karaf@root> obr:list                                  # listing the available
bundles in repository
…
Perfsonar Command Line Measurement Point (0.0.1.SNAPSHOT)
Perfsonar NMWG Adapter (0.0.1.SNAPSHOT)
Perfsonar RRD Measurement Archive (0.0.1.SNAPSHOT)
```

The **repository** client has an access to a federated set of repositories via the **Repository Admin** service.

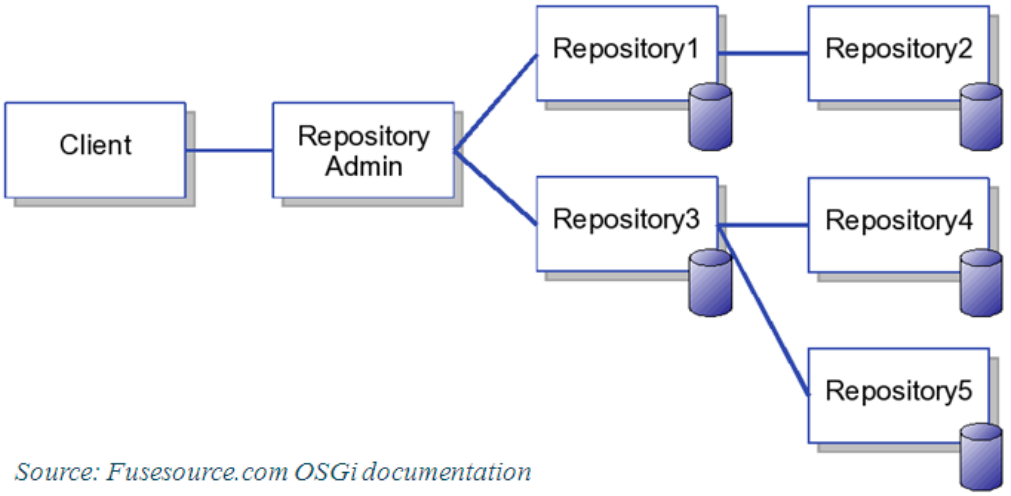

Figure 5.3: Repository client

The **REST interface** and **website interface** are implemented as an application for the **WSGIService** framework [WSGIService] and are run as a module for **Service Repository daemon**.

The **REST interface** is in the implementation stage and will provide all the functionalities to deploy and discover list bundles and to get detailed repository resource information. It will also allow the obtaining information on capabilities, requirements and other meta-data of given bundle or package.

Below, you can see the HTTP REST protocol specification of the Service Repository REST interface.

HTTP message, method and URI	Message Content	Abstract Message
Bundle push / deploy		
HTTP Request PUT http://gembus.geant.net/repository/bundle/{bundle-id}	Bundle JAR	Bundle Push
HTTP Response • 200 OK • 201 Created • 400 Bad Request • 401 Unauthorized • 503 Service Unavailable	–	Bundle Push Result
HTTP Request GET http://gembus.geant.net/repository/bundle/{bundle-id}	--	Bundle Deploy Request
HTTP Response	Bundle JAR	Bundle Deploy Response

HTTP message, method and URI	Message Content	Abstract Message
• 200 OK • 404 Not Found		
Bundle discovery		
HTTP Request GET http://gembus.geant.net/repository/bundle	–	Bundle List Request
HTTP Response • 200 OK • 404 Not Found	XML described by bundle-advertisement XSD	Bundle List Response
HTTP Request GET http://gembus.geant.net/repository/bundle/{bundle-id}/capabilities GET http://gembus.geant.net/repository/bundle/{bundle-id}/requirements	–	Bundle Capabilities and Requirements Request
HTTP Response • 200 OK • 404 Not Found	XML described by bundle-advertisement XSD	Bundle Capabilities and Requirements Response
Bundle information get / modify		
HTTP Request GET http://gembus.geant.net/repository/bundle/{bundle-id}/description GET http://gembus.geant.net/repository/bundle/{bundle-id}/size etc.	–	Bundle Information Request
HTTP Response • 200 OK • 404 Not Found	XML described by bundle-advertisement XSD	Bundle Information Response
HTTP Request POST http://gembus.geant.net/repository/bundle/{bundle-id}/description POST http://gembus.geant.net/repository/bundle/{bundle-id}/size	XML described by bundle-advertisement XSD	Bundle Information Modify Request
HTTP Response • 200 OK • 201 Created • 400 Bad Request • 401 Unauthorized • 503 Service Unavailable	--	Bundle Information Modify Response

Table 5.1: HTTP REST protocol specification

5.4 Flagship Applications: F-Ticks

Following the demonstration of GEMBus's potential presented in Deliverable DJ3.3.2, *Composable Network Services Framework and General Architecture: GEMBus*, the next step has been to find real-world applications to integrate with GEMBus services and show the initial functionalities of the framework to other interested partners in order to promote the collaboration and disseminate the results obtained throughout the project.

The F-Ticks service has been developed within GN3-JRA3-T1, and implemented by GN2-SA3-T2 (eduroam). It is a data collection service designed for a very simple implementation at eduroam participating organisations. The service consists of two main elements: a **data publisher**, which sends a record of every eduroam authentication attempt, and a **central data collector**, which reads all records and stores them in a database. Access to the database and data presentation are separate tasks. This original F-Ticks uses Syslog protocol for data logging as the transport mechanism for messages. Syslog is easily coupled with authenticating RADIUS servers, which makes the publisher setup as straightforward. The ease of publisher setup is absolutely crucial since the service can be successful only if it is widely implemented.

5.4.1 Adding F-Ticks to GEMBus

The addition of F-Ticks to GEMBus use cases has grown from a number of reasons:

- The F-Ticks message feed could be published as a simple and universally accessible data stream within GEMBus, for the use of parties studying eduroam statistics.

- Composition of complex services is the main purpose of GEMBus, but it has also been observed that the complexity of the service itself may obscure the details of the GEMBus integration, therefore F-Ticks, which is easy to comprehend, can serve as a good example of a complex composite service.

- The F-Ticks example must use all core components of GEMBus: translation of a standard service protocol (Syslog) into GEMBus messages, the publish/subscribe service, the repository (for finding F-Ticks, publishing the endpoint and comprehending the output), security services if the access should be restricted or if the level of data anonymisation is dependent on access rights.

- As GEMBus grows in popularity it may be natural to use it as a transport mechanism, even for F-Ticks. This could be done either by using the existing protocol translators locally at the authenticating server or even to implement direct GEMBus hooks into popular RADIUS servers.

The current GEMBus implementation has created the interface to the original F-Ticks service, where Syslog messages are fed into the GEMBus component, transformed into GEMBus messages and published inside the bus. Local subscriber services have also been implemented and will be turned into the final example when the GEMBus composition is ready for use.

It must be clearly stated that it has never been a goal to produce a mature F-Ticks statistics analyser. Such a tool is out of scope of this activity. The example subscriber is expected to serve as a “Hello World” application, which can be used as the starting point for a number of specialised consumer systems.

5.4.2 Implementation

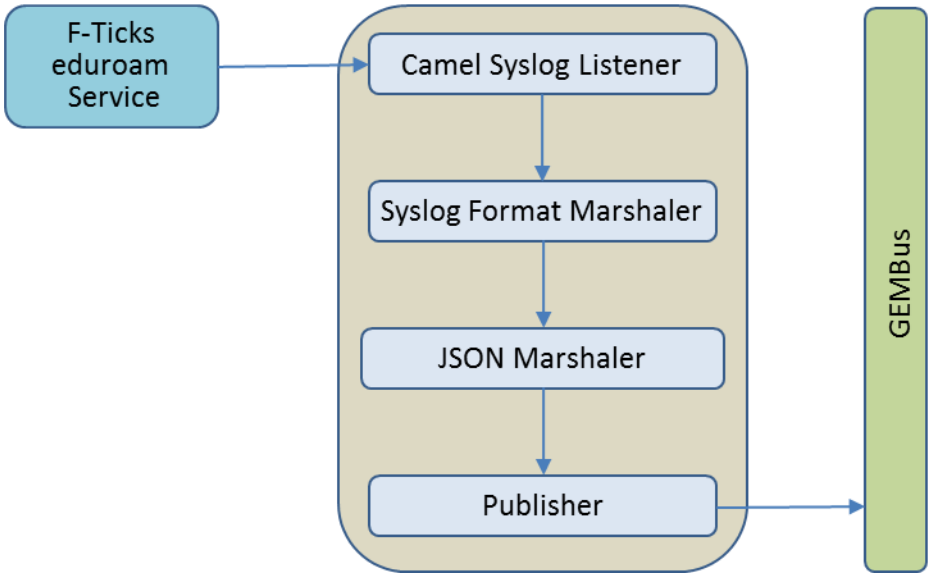

Figure 5.4: F-Ticks framework

Implementation of the F-Ticks Syslog listener bundle is based on Apache Camel framework and on the definition of routing rules using Java DSL. The bundle was created using one of Camel Maven archetypes: **camel-archetype-java**, which is designed to create a new Maven project for Camel routes.

The default Camel class for the definition of routes, **MyRouteBuilder**, implements the following actions:

- Reads syslog messages from the defined endpoint using **camel-mina transport** component (udp://: //localhost:10514).
- Uses **SyslogDataFormat** and **SyslogMessag**e classes from camel-syslog to process messages.
- Unmarshals syslog message using **SyslogDataFormat** class.
- Processes a message as a F-Ticks message using **FTicks** class, which analyses message fields and constructs FTicks class data filled with the appropriate information from message.
- Marshals F-Ticks data to JSON format.
- Sends JSON data are sent using **Apache ActiveMQ** to a **JMS Topic** (multiple consumers can get the message) activemq:topic:fticks.

The Camel Context configuration in the **camel-context.xml** file defines the **activemq** bean.

FTicks class handles F-Ticks data:

- Sets appropriate data fields.
- Uses **org.apache.camel.component.syslog.SyslogMessage** class methods to get log message, time stamp, remote address, etc.

5.4.3 Result

FTicks syslog listener bundle forwards all received F-Ticks messages to the established ActiveMQ endpoint, using publish-subscribe logic.

FTicks object contains the following fields:

- remoteaddress
- timestamp
- fticksid
- fticksrealm
- fticksviscountry
- fticksvisinst
- ftickscsi
- fticksresult

Example result messages:

```
{
"gembus.fticks.data.FTicks":
{
"remoteaddress":"\/127.0.0.1:38358",
"timestamp":"2011-10-12 10:24:30.911 CEST",
"fticksid":"F-TICKS\/eduroam\/1.0",
"fticksrealm":"wlan.mnc003.mcc260.3gppnetwork.org",
"fticksviscountry":"PL",
"fticksvisinst":"BIAMANTLS",
"ftickscsi":"10:f9:ee2d532e40c36aa6084be5c10ef2220",
"fticksresult":"FAIL"
}
}

{
"gembus.fticks.data.FTicks":
{
```

```
"remoteaddress":"\/127.0.0.1:38358",
"timestamp":"2011-10-12 10:26:12.958 CEST",
"fticksid":"F-TICKS\/eduroam\/1.0",
"fticksrealm":"gumed.edu.pl",
"fticksviscountry":"PL",
"fticksvisinst":"TORMANTLS1",
"ftickscsi":"00:1f:3c4ca8815e418795c3b06fceec9aef3",
"fticksresult":"OK"
}
}
```

6 Current General Status and Future Work

As of January 2012, GEMBus is in a beta version; there are some demonstrators available, but there is not yet a complete GEMBus testbed that can be offered.

A complete GEMBus system could be deployed in a number of different ways. For instance, all GEMBus core components could be offered by a single domain or different domains could operate some of the core components. Figure 6.1 depicts a possible scenario in which GEMBus could be deployed.

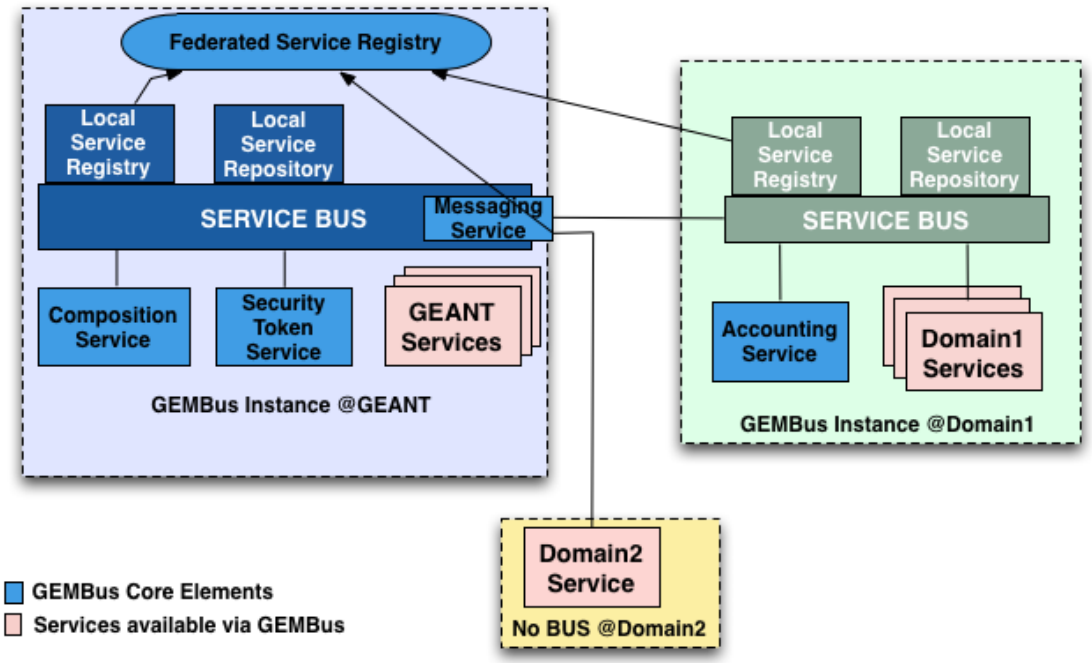

Figure 6.1: Using GEMBus in a multi-domain scenario

6.1 Further Developments

It is important to stress that GEMBus is a research project, and as such, is still under development. However, GEMBus is following a clearly defined development path.

During the next phase of GN3 (2012–13), development will focus on offering a test bed for GEMBus that will be available to all project partners, as well as to those communities that have expressed an interest in testing GEMBus, such as the CLARIN and eScience Projects. Meeting this objective is not easy since it requires the integration of components that have been developed separately. We are in contact with the company behind the main integration bus we are using, FUSESource (detailed in Section 6.3), in order to facilitate the task of putting together the components, as well as get a deeper knowledge of that product.

In addition to the pilot, a cookbook is currently being written by the GEMBus team and will be available to the community in the spring of 2012, which will facilitate the usage and wider take-up of GEMBus.

6.2 The GEMBus Cookbook

The GEMBus cookbook provides guidance for GN3 users on the use of GEMBus, as well as deploying new services to expand its use. This document explains not only the nature and operation of GEMBus but also offers a step-by-step tutorial for using and adding new services.

The cookbook document comprises six chapters, covering all necessary aspects of the GEMBus platform.

Chapter 1 provides an introduction to the GEMBus platform, as well as instructions on how to install GEMBus onto a user's domain.

Chapter 2 deals with the services offered on the GEMBus platform, and more specifically, defines GEMBus services, and presents the GEMBus Core Services 2, a set of services that provide direct support to any service to be deployed in GEMBus. The core services that have been developed throughout the GN3 are:

- The Service Registry, which is responsible for storing service bundles, and allows their deployment on the local instance of the GEMBus.
- The Messaging Service is responsible for service interaction and integration with the key aspect of message transparency (whereby the sender may elect to leave some aspects of the message completely visible to potential readers, while encrypting other parts of the message for a trusted subset of readers).
- The Composition Service, which enables the composition of services.
- The Accounting Service that provides configurable and aggregated access to the GEMBus logging service to support monitoring, auditing, diagnostics and troubleshooting.

- The Security Token Service, which is built as WS-Trust implementation, issues, verifies and translates security tokens to allow the authentication of requesters in a federated, multi-domain environment.

In addition to the above, this chapter will also provide the reference service framework, as derived from the Reference Service Workshop. This workshop has been set up between the GRNET, UvA and UMU and with the aim to define the reference framework.

Chapter 3 illustrates how to access the GEMBus platform. It will present a general overview of how to use GEMBus Core Services with Reference Services. The material for this chapter will be produced as an outcome of the Reference services workgroup.

Chapter 4 is the main step-by-step guide for developers to create new services. It presents the pre-requirements for creating a new service, and setting up a GEMBus environment. The pre-requirements have been gathered during the canonicalisation task, and will be presented in specific terms of operating and programming environment used. Moreover, it will provide interface details for the newly created service to ensure compatibility with the Reference Framework. Such alignment with the Reference framework will guarantee access to the GEMBus core services.

After creating a new service, developers will have to integrate it with the GEMBus platform, as described in detail in Chapter 5. Because of this, the F-Ticks service could be included as a canonical example for developing and deploying a new service to the GEMBus platform. Moreover, a guide for validating a service will be included based on the survey / questionnaire distributed by RedIRIS.

Finally, Chapter 6 provides several answers to frequently asked questions (FAQ). GRNET is examining the idea of distributing a questionnaire to developers and users of the GEMBus platform, which could be used to discover common problems faced during any step of installing GEMBus, accessing its services, and developing and deploying new services.

Overall, it is our belief that the GEMBus cookbook will provide a useful guide for users and developers of the GEMBus platform.

6.3 FUSE Source Collaboration Proposal

As mentioned in the previous sections of this document, GEMBus has developed the GEMBus Core Components, of which the GEMBus Federated Service Registry or simply the GEMBus Registry is one of the most vital.

The Registry communicates with local registries (operating in the local domains) and announces the global availability of a service.

After an evaluation of the available ESBs (done at the beginning of the project in 2009), the GEMBus team selected FUSE as the preferred ESB platform. It is important to note that the FUSE ESB is widely used in the

research community, for instance, CERN is also using it to run the operational grid activities of the Large Hadron Collider.

Using FUSE as the preferred ESB base for GEMBus development, has worked pretty well in most of the aspects, however, some problems have emerged while federating different ESBs. To achieve its aim the GEMBus registry would have to be able to access to the information stored in the registries of the participating ESBs.

Not surprisingly, the registry in a FUSE ESB is geared to be used within one instance of the FUSE ESB, and therefore holds information in a format that is not suitable for exposure to other ESBs, especially if the other ESB is not a FUSE ESB (which should be considered to be the assumption in GEMBus).

To solve this problem, the GEMBus team approached a commercial company named FUSESource that offers support for FUSE ESB (which is an open source product). FUSESource clearly has more in-depth knowledge about FUSE than the GEMBus team; furthermore, FUSESource involvement would ensure that the functionalities to enable the export of data from the FUSE registry is permanently added into the FUSE product.

FUSESource has acknowledged the issues, and would (upon compensation), be able to solve it.

At the time of writing there are ongoing discussions between the GEMBus team and FUSESource, to quantify the work and investigate the feasibility of a possible collaboration and advance GEMBus developments.

In the absence of an agreement with FUSESource, the GEMBus team would be able to develop an API to export data stored in the local ESB registries to the GEMBus registry, however, this solution would not be desirable without the support of the FUSE community. In fact, the GEMBus team would have to spend a considerable amount of time developing such an API, which would be valuable only if FUSE ESB would accept the development and integrate it as part of future FUSE ESB releases. The involvement of FUSESource would enable this process.

Appendix A Implementation Details and Code

A.1 STS Configuration

The Security Token Service configuration is based on a XML file, where it can be configured all the classes used by the STS and other important parameters. An example of **STS configuration** file is shown below.

The STS configuration includes elements for:

- **Issuer name in generated tokens**.
- Mode: STS, TTS, AS.
- Default token TTL.
- **Keys used to verify and sign tokens**.
- Supported token providers.
    - Policy engines (PDPs) applied for authorisation.

The following lines show an example of a STS config file:

```
<GemSTSxmlns="urn:geant:gembus:security:config:0.1STSName="urn:geant:gembus:security:sts:gemsts"
TokenTTL="1800">
        <KeyHolderClassName="net.geant.gembus.security.key.impl.KeyHolderImpl”>
        <Property Name="KeyStoreURL" Value="/security/gn3-sts.jks" />
        <Property Name="KeyStorePassword" Value=”stspassword" />
        <Property Name="PrivateKeyAlias" Value=”stsprivkeyalias" />
        <Property Name="PrivateKeyPassword" Value=”stsprivkeypassword" />
        <Property Name="PublicKeyAlias" Value=”stspubkeyalias" />
        </KeyHolder>
        <RequestHandler>
        net.geant.gembus.security.wstrust.GemRequestHandler
</RequestHandler>
        <PolicyEngines>
        <PolicyEngine Id="InternalPolicyEngine”
ClassName="net.geant.gembus.security.wstrust.policy.xacml.XACMLPolicyEngine">
        <Property Name="PolicyFile" Value="/policy/GemSTSPolicy.xml" />
        <Property Name="Resource" Value="GemSTS" />
        <Property Name="Action" Value="authorize" />
        </PolicyEngine>
        <PolicyEngine Id="ExternalPolicyEngine”
ClassName="net.geant.gembus.security.wstrust.policy.xacml.XACMLRemotePolicyEngine">
        <Property Name="PDPEndpoint” Value="http://155.54.210.139:9999/axis2/services/PdpService" />
        <Property Name="Resource" Value="SampleServer" />
```

```
        <Property Name="Action" Value="access" />
        </PolicyEngine>
        … More policy engines used by the token providers …
        </PolicyEngines>
        <TokenProviders>
        <TokenProvider
ProviderClass="net.geant.gembus.security.wstrust.token.providers.saml.v2.SAML2TokenProvider”
TokenType="urn:oasis:names:tc:SAML:2.0:assertion" TokenLocalName="Assertion”
TokenNS="urn:oasis:names:tc:SAML:2.0:assertion" PolicyEngine="ExternalPolicyEngine" />
        <TokenProvider
ProviderClass="net.geant.gembus.security.wstrust.token.providers.gembus.GemTokenProvider”
TokenType="urn:geant:gembus:security:token:1.0:gemtoken” TokenLocalName="GemToken"
TokenNS="urn:geant:gembus:security:token:1.0:gemtoken" PolicyEngine="InternalPolicyEngine"/>
… More token providers …
        </TokenProviders>
        <ServiceProviders>
        <ServiceProvider Name=“http://www.geant.net/sp1”
TokenType="urn:geant:gembus:security:token:1.0:gemtoken" />
        <ServiceProvider Name=“http://www.geant.net/sp2”
TokenType="urn:oasis:names:tc:SAML:2.0:assertion" />
        … More known service providers …
        </ServiceProviders>
        <IdentityProviders>
        <IdentityProvider Name="http://www.geant.net/idp1">
        <KeyHolderClassName="net.geant.gembus.security.key.impl.KeyHolderImpl">
                        <Property Name="KeyStoreURL" Value="/security/idps.jks" />
                        <Property Name="KeyStorePassword" Value=”idpspw" />
                        <Property Name="PublicKeyAlias" Value=“idp1" />
        </KeyHolder>
        </IdentityProvider>
        … More trusted identity providers …
</GemSTS>
```

The most salient configuration elements are:

- **GemSTS**: the root element. It defines the properties that allow the STS administrator to set default values:
    - **STSName**: a String representing the name of the security token service. If not specified, the default **GemSTS** value is used.
    - **TokenTTL**: the token lifetime value in seconds. If not specified, the default value of 3600 (1 hour) is used.
- **KeyHolde**r: this element and all its sub-elements are used to configure the keyholder that will be used by GemSTS to sign and validate tokens. The implementation provided in the library uses the Java keystore as backend. Properties such as the keystore location, its password, the signing (private key) alias and password and public key alias are all configured within this element.
- **RequestHandler**: this element specifies the fully qualified name of the **WSTrustRequestHandler** implementation to be used.
- **TokenProviders**: this element specifies the **SecurityTokenProvider** implementations that must be used to handle each type of security token and the **PolicyEngine** used by each provider. In the example implementations, there are two providers: one that handles **urn:oasis:names:tc:SAML:2.0:assertion** tokens and uses a remote XACML policy engine, and

another that handles **type urn:geant:gembus:security:token:1.0:gemtoken**tokens, and uses an internal XACML policy engine.

- **ServiceProviders**: this element specifies the token types that must be used for each service provider (the Web service that requires a security token). When a WS-Trust request does not contain the token type, the **WSTrustRequestHandler** must use the service provider endpoint to find out the type of token that must be issued. First, the handler queries the GEMBus registry to obtain the token type for the service provider. If the GEMBus registry does not contain the token type for the service provider, then the **ServiceProviders** configuration section is used.

- **IdentityProviders**: this element specifies the identity providers in which the STS trusts. Any security token issued by these identity providers can be used to obtain a new security token. Each identity provider contains a **KeyHolder** element, where it is specified the place where to obtain the public key that will be used to validate the token signature. Properties such as keystore location, its password and the public key alias are configured in this section. Here the private key alias and password are not necessary, because the token providers only need the public key in order to validate the token.

- **PolicyEngines**: this element specifies the policy engines that can be used by the token providers when the tokens are validated. The library contains two implementations of the **PolicyEngine** interface. The **XACMLPolicyEngine** requires properties like the policy file and the default values for resource and action in order to can perform a XACML request, whereas the **XACMLRemotePolicyEngine** requires the PDP endpoint besides the default values for resource and action to perform a XACML request using SOAP.

All these elements are used by the STS and its components. The **WSTrustRequestHandler** uses the **TokenTTL** when no **Lifetime** has been specified in the WS-Trust request. It creates a **Lifetime** instance that has the current time as the creation time and expires after a specified number of seconds.

The **KeyHolder** is used by the **GemSTSConfiguration** to access the configured keystore and provides the STS private and public key and the identity providers' public key to the **SecurityTokenProvider** when it needs to sign or validate a security token.

The **PolicyEngines** element contains the policy engines that can be used by the token providers to take an authorisation decision on the security token.

The **TokenProviders** elements used by the **GemSTSConfiguration** to obtain the **SecurityTokenProvider** that must be used to handle a WS-Trust request that specifies the token type. **The WSTrustRequestHandler** calls the **getProviderForTokenType**（String tokenType） method of **STSConfiguration** to obtain a reference to the appropriate **SecurityTokenProvider**.

The **ServiceProviders** element is used by the **GemSTSConfiguration** to obtain the **SecurityTokenProvider** that must be used to handle a WS-Trust request that does not specify the token type. In this case, the request message must identify the service provider endpoint. The **GemSTSConfiguration** first locates the token type of the service provider using the service provider name defined in the **ServiceProvider** elements, and then locates the **SecurityTokenProvider** using the **TokenProvider's** elements. The **WSTrustRequestHandler** calls the **getProviderForService**（String serviceName） method of **STSConfiguration** to obtain a reference to the appropriate **SecurityTokenProvider**.

The **IdentityProviders** element is used by the **GemSTSConfiguration** to obtain the public key of an identity provider. The **SecurityTokenProvider** calls the **getIdentityProviderPublicKey(StringidentityProviderName**) method of **STSConfiguration** to obtain a reference to the identity provider public key. The provider will use the public key to validate the signature of the security token within the **validateToken** method of **SecurityTokenProvider**.

A.2 Message Broker Configuration

The listing below provides example of Message Broker configuration using static IP addresses. According to the services composition workflow, the Message Broker, containing Source Service 1 and Source Service 2 queues, needs to register with Message Broker containing Processor Service 2.

```
<blueprint ...>
  <broker ...>
    <networkConnectors><networkConnector name="ncNeptue"
                  uri="static:(tcp://192.168.56.102:62001)" duplex="true">
      </networkConnector></networkConnectors>
    <transportConnectors>
      <transportConnector name="openwire" uri="tcp://localhost:61616"/>
      <transportConnector name="stomp" uri="stomp://localhost:61613"/>
    </transportConnectors>
  </broker>
  <bean id="activemqConnectionFactory"
            class="org.apache.activemq.ActiveMQConnectionFactory">
       <property name="brokerURL" value="tcp://localhost:61616" />
  </bean>
  <service ref="pooledConnectionFactory" interface="javax.jms.ConnectionFactory">
    <service-properties>
      <entry key="name" value="localhost"/>
    </service-properties>
  </service>
</blueprint>
```

An example Message Router configuration in beans.xml, and a Message route description in Java Domain Specific Language (DSL) is provided below.

Message Broker configuration in beans.xml

```
<beans ….>
<bean id="jms" class=
"org.apache.camel.component.
jms.JmsComponent">
  <property name="connectionFactory">
    <bean class= "org.apache.activemq.
ActiveMQConnectionFactory">
      <property name="brokerURL"
value="tcp://localhost:61616"/>
    </bean>
  </property>
<route>
    <from uri="jms:S1_Out"/>
    <to uri="jms:Sp1_In"/>
</route>
<route>
    <from uri="jms:S2_Out"/>
    <to uri="jms:Sp1_In"/>
</route>
<route>
    <from uri="jms:S3_Out"/>
    <to uri="jms:Sp2_In"/>
</route>
<route>
    <from uri="jms:Sp1_Out"/>
    <to uri="jms:Sp2_In"/>
</route>
</camelContext>
</beans>
```

Message Router configuration in Java DSL:

```
from(“jms:S1_Out”).to(“jms:S3_In”);
from(“jms:S2_Out”).to(“jms:S3_In”);
from(“jms:S3_Out”).
bean(LogSignal.class, “log(${body})”);
```

Figure A.1: Message router configuration

Appendix B Installation of Additional Composition Components

B.1 Installation of Testing and Debugging Components

There are some components that can be handy for testing and debugging. They can be easily installed issuing:

```
karaf@root> features:install ode-commands

karaf@root> features:install examples-ode-helloworld

karaf@root> features:install examples-ode-ping-pong
```

It is highly recommended to use the provided *Eclipse Plug-in for GEMBus* to interact with the server during the rest of the process. To install and activate all the features needed by the Plug-in a simple change must be done on one configuration file **$GEMBus/etc/config.properties** ($GEMBus points to the GEMBus installation root directory) replacing the line:

```
org.osgi.framework.bootdelegation=org.apache.karaf.jaas.boot,sun.*,com.sun.*,javax.transaction,javax.transaction.*
```

with the following instruction:

```
org.osgi.framework.bootdelegation=org.apache.karaf.jaas.boot,sun.*,com.sun.*,javax.transaction,javax.transaction.*,javax.xml.stream,javax.xml.stream.*
```

After this change, configuration must be updated or the server restarted.

Next, proceed to the manual installation of the deployment-api bundle. This can be done copying the file **deployment-api-2.0.0.jar** that can be obtained in the GEMBus SVN, to the deployment folder **$GEMBus/deploy/**. Now we are ready to use the deployment capabilities of the **Eclipse Plug-in** *for* GEMBus.

The next component to install is the Plug-in (in the Eclipse installation). It is also possible to use it within the Intalio Designer[2]. The process is the same on both IDEs, just copy the **Plug-in** file and the required

[2] Intalio|Designer is property of Intalio, Inc.

dependencies (**/plugins/** folder in the svn) on the **/plugins/** folder of the IDE. Restart the IDE if it is already open. Now the Plug-in may be opened by double clicking on any .jar or .zip file. (An alternative method to start the Plug-in will be implemented in the future.)

The Plug-in will start with an overview window showing some configuration parameters. First, enter the URL where the GEMBus server is running, leave the port parameters as they are unless you have modified them, and check that the deployment-api is correctly accessed by clicking on **Test Connection** (notice that a warning message with missing management-api will be shown).

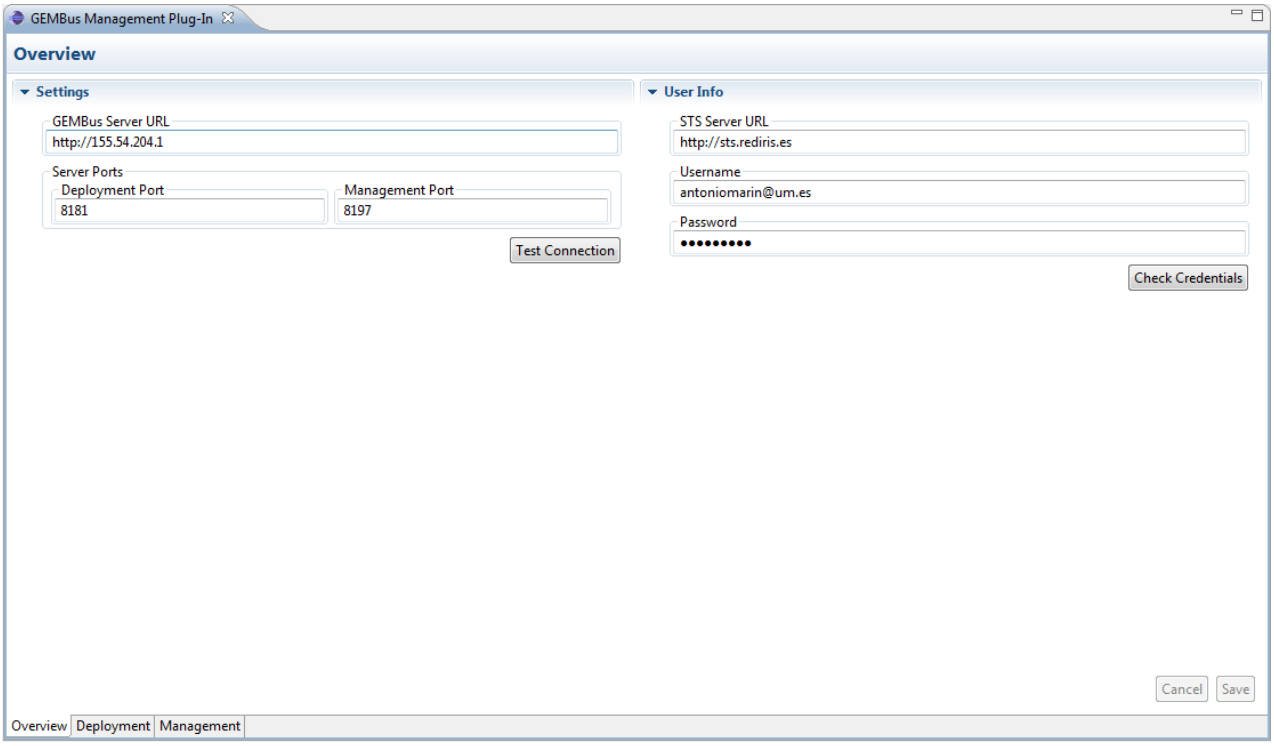

Figure B.1: Eclipse plug-in for GEMBus: Overview window

Now select the **Deployment** tab, where a list of local bundles (All the .jar and .zip files contained in the Project) will be shown. Copy the **management-api-1.0.0.jar** file (Obtained from the svn) to the Project and click the **Refresh** Button on the Local Bundles section, the bundle will be shown, and may now be deployed by selecting and clicking the **Deploy** Button.

The remote bundles list will be updated showing the new deployed bundle.

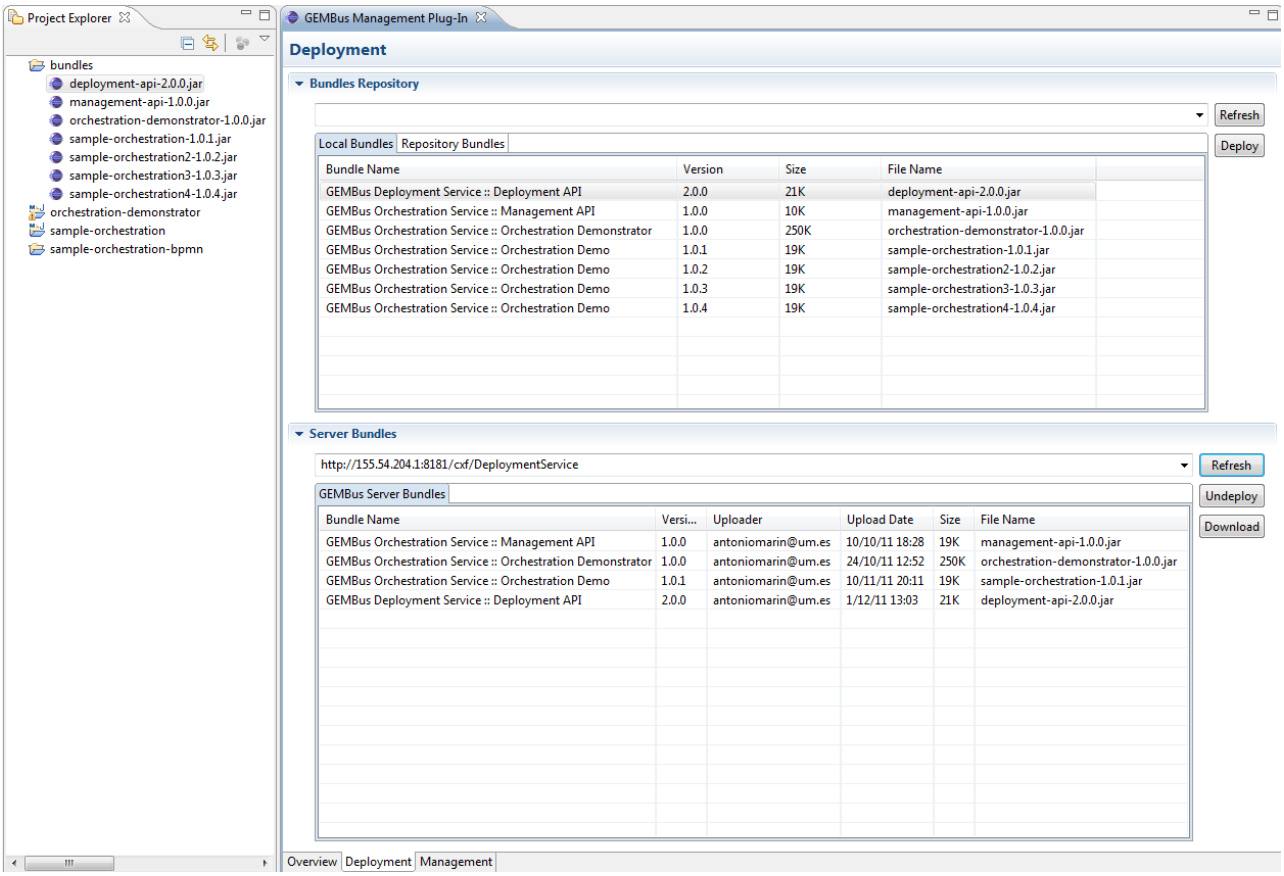

Figure B.2: Eclipse plug-in for GEMBus: Deployment window

Using this window, it is possible to deploy, un-deploy or download the bundles stored in the GEMBus server.

B.2 Process Modelling

B.2.1 Intalio Designer Installation

The latest version of the design IDE can be obtained for use with different architectures from the download section of the Intalio website [Intalio]. Install the software following the setup process and finally, install the *Eclipse Plug-in for GEMBus* if desired (recommended). When launching the IDE, the user will be prompted for his/her Intalio credentials, however, these are not mandatory, and may be skipped during the install.

B.2.2 Process Implementation

Once the IDE is installed, create a new Intalio Designer Business Process Project and name it **sample-orchestration-bpmn**. Next, create a new Business Process Diagram and name it **SampleProcess**.

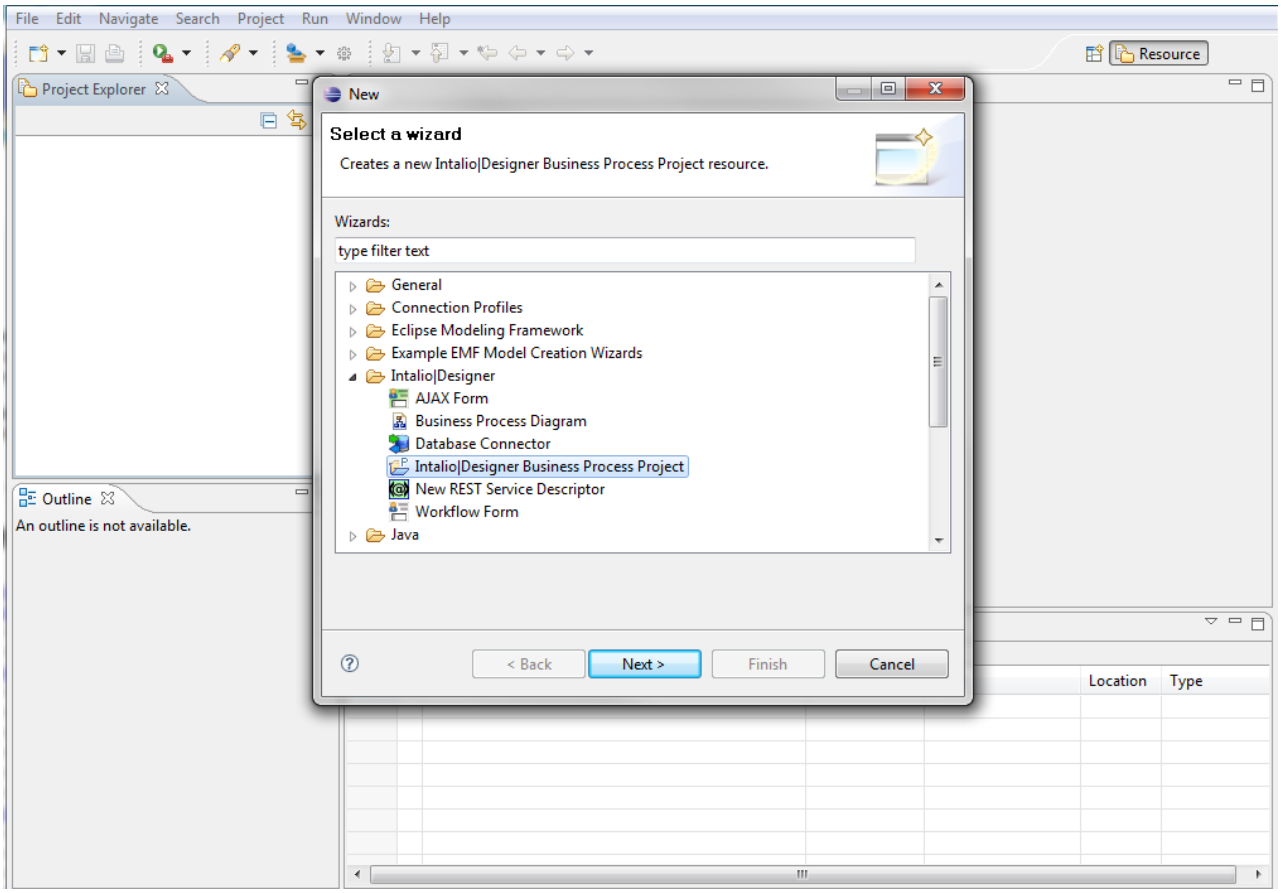

Figure B.3: Intalio Designer: New project wizard

Next, locate the Web Service Definition Language (WSDL) files of the web services that are going to be called in the composition. These can be found on the GEMBus svn. Also include the **XML Schema Definition (XSD)** for message typing.

Once all the necessary files have been collected, the modelling of the process can begin. All the modules needed are available in the **palette**, or in the **context** menu, which can be activated by hovering the mouse over the Process Editor window.

After this, create two new pools and label them **Process** and **Services**. Also rename the existing one **Client**. Set the Client and Services pools to non-executable by right clicking on them. Rename the existing task **GreetingService**.

Following this, add the remaining tasks, connections, and input/output events to the process editor as shown in Figure 3.4. For more info on how to create the diagram please refer to the tutorials section of the Intalio website.

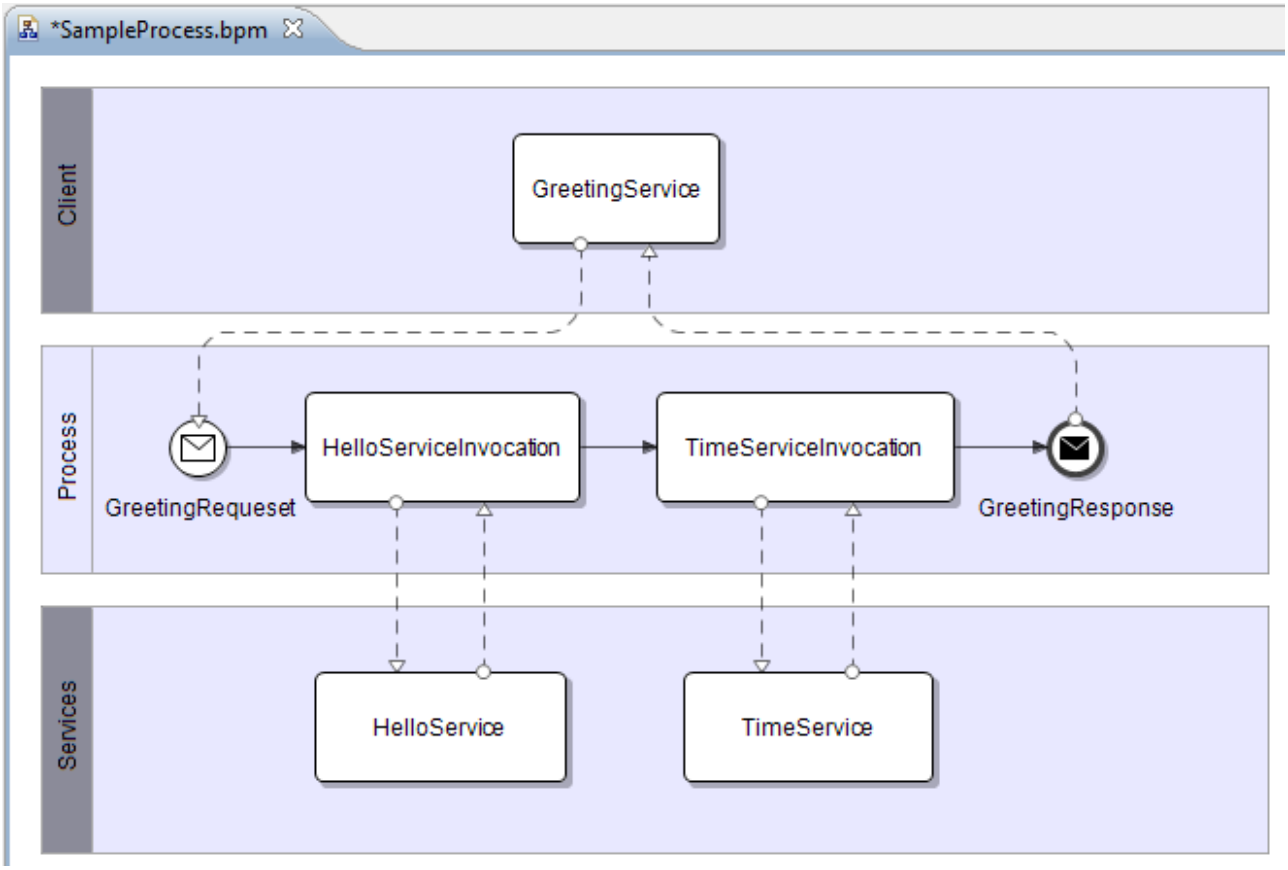

Figure B.4: Intalio Designer: Process Editor

Expand the WSDL files in the Process Explorer to see the methods implemented by the service. Drag and drop the **sayHi** operation to the **HelloService** task, and select **Provide operation**. Do the same with the **getUTCTime** operation to the **TimeService** task.

Finally, the input/output messages need to be mapped to pass parameters and results. The parameters are the input required by each service for its correct operation. That input may come as an output from other services, the resulting process input, or manipulation combination of the aforementioned, so those parameters should be passed by connecting the corresponding inputs/outputs represented as variables.

First, select the **HelloServiceInvocation** task. Using the Mapper view, a list of variables will appear on the left and right-hand sides. Here, we need to link the parameter coming from the **GreetingRequest** to the **HelloService**. Click on the right-hand border of the **$thisGreetingRequestRequestMsg.body** variable, and a link cursor will appear, click on the left border of the **helloWorldSayHiMsg.paramer$** variable on the right-hand side. Next, connect the output from the **HelloService** with the output of the **TimeService**, add some text to make it user-friendly, and connect to the **GreetingResponse**. Then select the **GreetingResponse** event, to display a new set of variables in the Mapper.

First, create three new operators by clicking the red box on top of the Mapper view, and fill them in, as shown in Figure B.5. Finish connecting every variable and operator as depicted in Figure B.5.

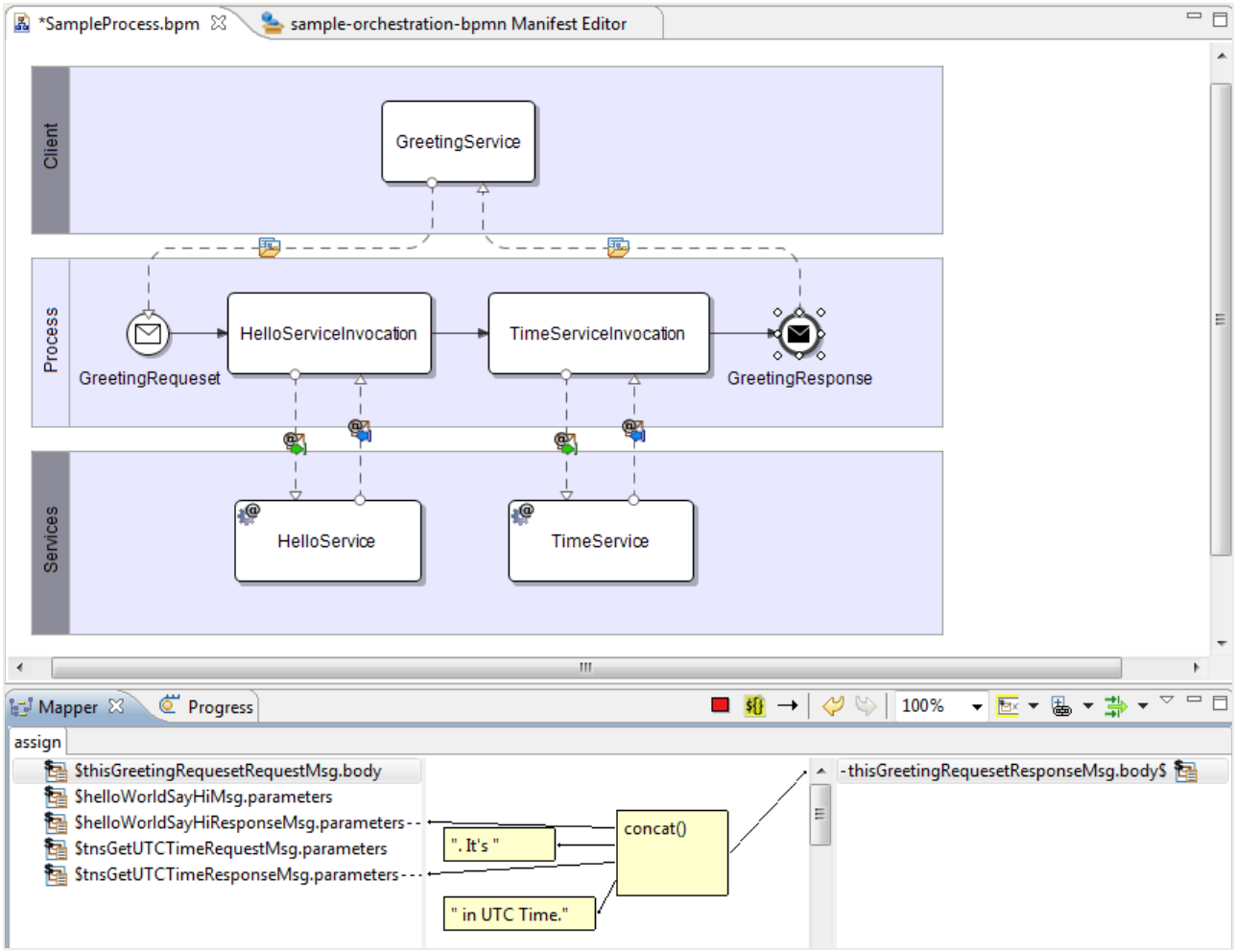

Figure B.5: Intalio Designer: Mapper View

B.3 Code and OSGi Bundle Generation and Deployment

If left in default mode, Intalio Designer automatically builds upon any project changes, but some files need to be created through the Deploy/Archive bundle, which launches the builders needed to generate all the necessary content. At this step, the server URL is not relevant for the Deploy method, and may be set to any invalid URL, or use the Archive method to ignore that parameter. Set the target namespace (urn:/net.geant.gembus.service.composition.orchestration.examples) and click Deploy / Archive, depending on the method selected.

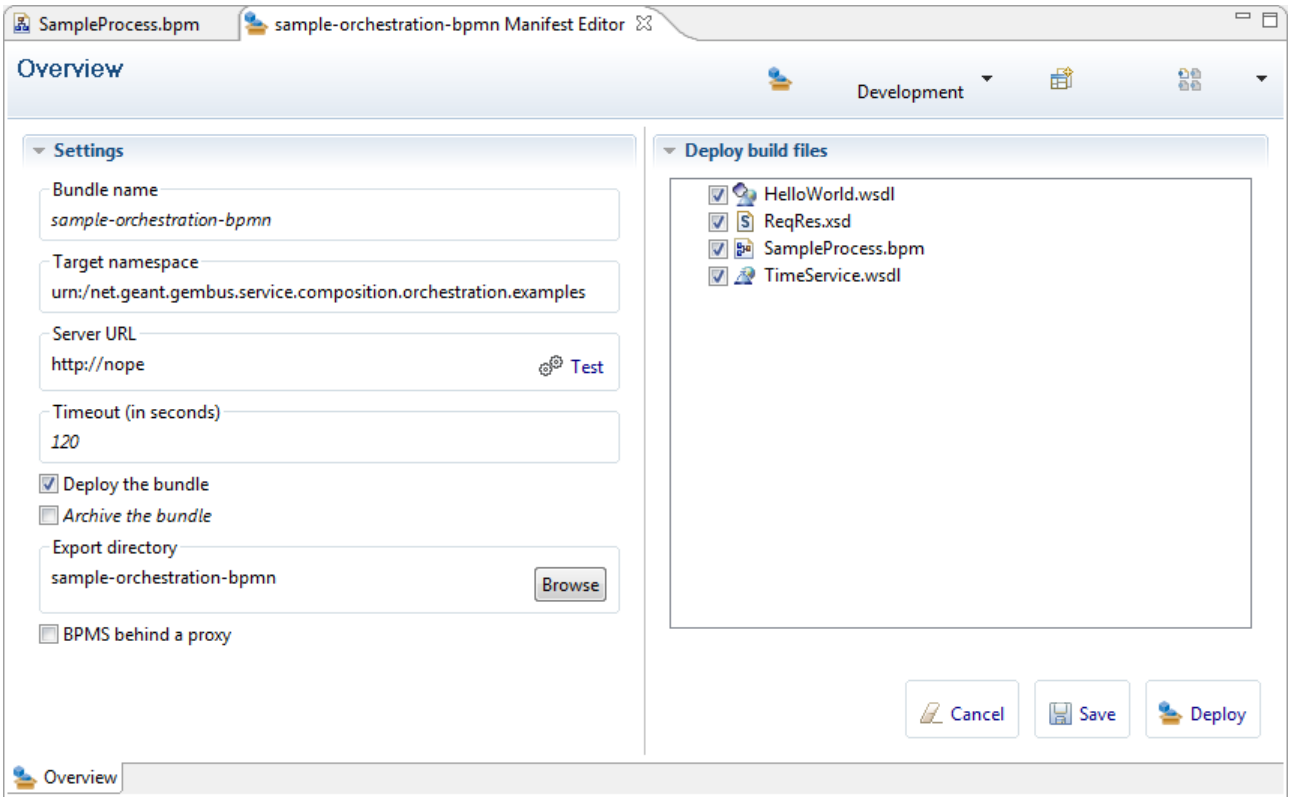

Figure B.6: Intalio Designer. Project Manifest Editor

After a few seconds, the **build** folder will be filled with all the necessary files to create a deployable bundle. We need to import the OSGi Maven project for ODE. Import the project sample-orchestration from the Subversion (SVN) and copy the content of the **build** folder to the **OSGi project,** under the **src/main/resources/META-INF** folder.

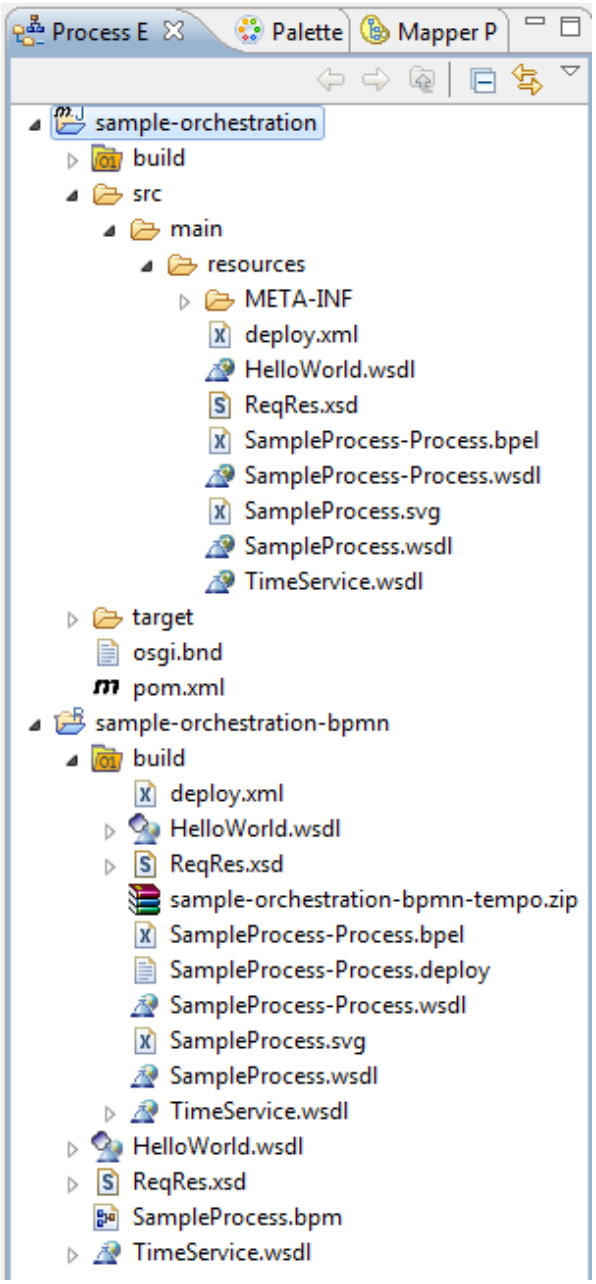

Figure B.7: Intalio Designer: Process Explorer

Note that some files are not needed at this stage, such as the tempo.zip (**sample-orchestration-bpmn-tempo.zip**), or the source deploy file (**Sample-Process-Process.deploy**). Configure the endpoints to be exported/imported to the Java Business Integration (JBI) bus, so the composition engine can use/expose them. In that sense, the **src/main/resources/META-INF/sptring/beans.xml** file needs to be configured to create providers for the invoked web services, as well as a consumer to expose the resulting composition to the outer world via an Apache CXF binding component.

```
<cxfbc:consumer
   xmlns:sampleorchestration="urn:/net.geant.gembus.service.composition.orchest
ration.examples/SampleProcess/Process"
   wsdl="classpath:SampleProcess-Process.wsdl"
   locationURI="http://155.54.204.1:8197/SampleOrchestration/"
   targetService="sampleorchestration:CanonicServiceForClient"
   targetEndpoint="canonicPort" />

<cxfbc:provider xmlns:hws="http://cxf.examples.servicemix.apache.org/"
   wsdl="classpath:HelloWorld.wsdl"
   locationURI="http://localhost:8181/cxf/HelloWorld"
   service="hws:HelloWorldImplService"
   endpoint="HelloWorldImplPort"
   interfaceName="hws:HelloWorldImplServiceSoapBinding"
   useJBIWrapper="false"
   useSOAPEnvelope="false"
   schemaValidationEnabled="false" />

<cxfbc:provider xmlns:ts="http://ws.intalio.com/TimeService/"
   wsdl="classpath:TimeService.wsdl"
   locationURI="http://ws.intalio.com/TimeService/"
   service="ts:TimeService"
   endpoint="TimeServiceSoap"
   interfaceName="ts:TimeServiceSoap"
   useJBIWrapper="false"
   useSOAPEnvelope="false"
   schemaValidationEnabled="false" />
```

This step is a JBI requirement, and the file provided in **src/main/resources/META-INF/sptring/beans.xml** from the svn project can be used.

Now we are ready to create the bundle for deployment. This is done using Maven, so any method used to generate the bundle will be valid. For best results, install Eclipse IAM [Eclipse IAM] following the instructions for Eclipse 3.4 (Intalio Designer is built upon it). This will create a file under the **target** folder, which is the deployable bundle. The Eclipse Plug-in can be used to deploy it on the server.

B.4 Processes and Instances Management

If everything goes as expected, the resulting composition should be up and running. To check this, use the **management** tab of the **Eclipse Plug-in for GEMBus**. Figure B.8 shows the deployed process, with a resume of the Instances and some available operations.

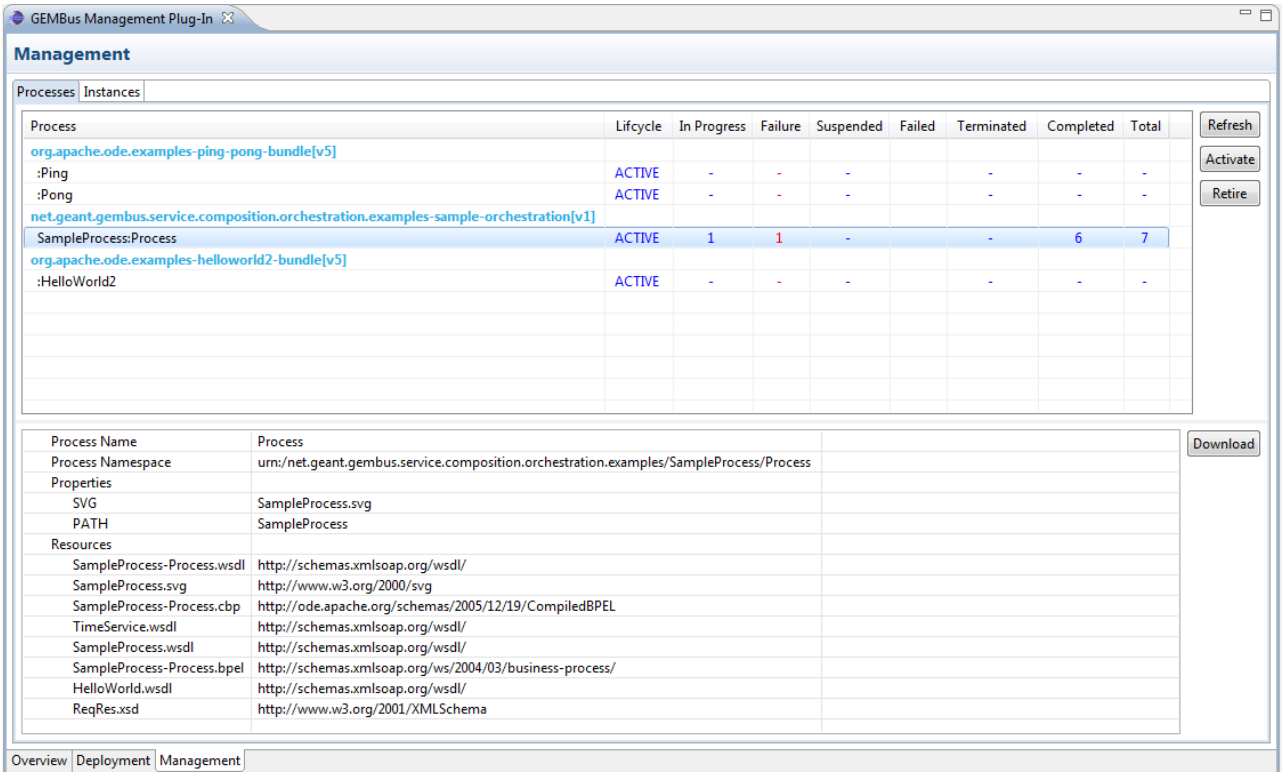

Figure B.8: Eclipse Plug-in for GEMBus. Management window

Below the **Process** resume table, the selected process details are displayed and some operations are also available, such as download the process resources, view the diagram (If it is available), etc.

References

[ApacheCamel]	http://camel.apache.org/
[ApacheInt]	Fuse ESB. Developing Apache CXF Interceptors. Version 4.3 August 2010.
[ApacheActiveMQ]	http://activemq.apache.org/
[ApacheAxis]	http://ws.apache.org/axis/
[Bhushan01]	*Federated Accounting: Service Charging and Billing in a Business-to-Business Environment.* Bhushan, B., Bhushan, B. Tschichholz, M. Leray, E. Donnelly, W., IEEE/IFIP International Symposium on Integrated Network Management, 2001.
[Chen06]	Chen, X., Khan, A. and Kant, D. Aggregate Accounting Record Recommendation. *Open Grid Forum (OGF)* 2006.
[CLARIN]	http://www.clarin.eu/external/
[CXFINTER]	*Developing Apache CXF Interceptors,* 2010.
[DJ3.3.2]	*Composable Network Services Framework and General Architecture* http://www.geant.net/Media_Centre/Media_Library/Pages/Deliverables.aspx
[**eduroam**]	http://www.eduroam.org/
[**Eclipse IAM**]	http://code.google.com/p/q4e/wiki/Installation
[FUSE]	http://fuse.sourceforge.net/
[GEMBus]	Demo available at http://gembus.inf.um.es:8181/OrchestrationDemonstrator/
[Götze]	Extensible and Scalable Usage Accounting in Service-oriented Infrastructures based on a Generic Usage Record Format. Joachim Götze, Tino Fleuren, Bernd Reuther, Paul Müller, WEWST '11, September 14, 2011, Lugano, Switzerland
[GRNET]	http://www.grnet.gr/default.asp?pid=1&la=2
[IETF]	Internet Engineering Task Force, http://www.ietf.org
[Intalio]	http://www.intalio.com/downloads http://community.intalio.com/tutorials-5.2/implementing-your-first-process-in-5.2-beginner.html
[JSON]	http://www.json.org/
[Kuhne11]	*Charging and Billing in Modern Communications Networks – A Comprehensive Survey of the State of the Art and Future Requirements.* Kühne, R., Huitema, G. and Carle, G. 2011, IEEE Communications Surveys & Tutorials
[Mach06]	Mach, R., Lepro-Metz, R. and Jackson, S. Usage Record Format Recommendation. *Open Grid Forum (OGF).* 2006.
[MongoDB]	MongoDB. *http://www.mongodb.org/*
[Mallmann08]	*SmartLM Grid-friendly Software Licensing for Location Independent Application Execution.* Mallmann, D., Martrat, J. and Ziegler, W**.** s.l. : inSiDE, Spring, 2008. 6(1).
[OASIS]	http://www.oasis-open.org/

[OAuth2]	OAuth2, http://oauth.net/2
[OIC]	The OpenID Connect specification, http://openid.net/connect
[OpenID]	OpenID foundation, http://openid.net
[OSGi Bundle Repository]	http://www.osgi.org/download/rfc-0112_BundleRepository.pdf
[OSGi BIndex]	http://www.osgi.org/Repository/BIndex
[RDF]	http://www.w3.org/TR/2004/REC-rdf-mt-20040210/
[RDF/XML]	http://www.w3.org/TR/REC-rdf-syntax/
[REST]	Representational state transfer http://en.wikipedia.org/wiki/Representational_state_transfer
[Rigney00]	Rigney, C. RADIUS Accounting . *Request for Comments of Internet Engineering Task Force (IETF), RFC 2866.* 2000.
[SAML]	The Security Assertion Markup Language, http://saml.xml.org
[SAML2]	http://saml2int.org/
[Scibilia07]	*Accounting of Storage Resources in gLite Based Infrastructures.* Scibilia, F. s.l. : 16th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2007.
[WSADDRESS]	WS-Addressing. *http://www.w3.org/Submission/ws-addressing/*
[WSGIService]	http://packages.python.org/WsgiService/
[WSS]	http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
[WST]	http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html

Glossary

API	Application Programming Interface
AS	Authorisation Service
BPMN	Business Process Modelling Notation
CA	Certification Authority
CLARIN	Common Language Resources and Technology Infrastructure project
CLMP	Command Line Measurement Point
CSA	Composable Service Architecture
eduroam	Roaming confederation aiming to provide mutual roaming network access to its members
EPR	Endpoint reference (for SOAP messages)
ESB	Enterprise Service Bus
F-Ticks	Federated Ticker System, statistic tool from GN3-JRA3-T1 and GN3 SA3 T2
FUSE	Filesystem in USErspace
GEMBus	GÉANT Multi-domain Bus
GNU	GNU's Not Unix (A free, Unix-like operating system)
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ID	Identity
IdP	Identity Provider
IP	Internet Protocol
ITU-T	International Telecommunication Union, Telecommunication Standardisation Sector
JBI	Java Business Integration
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
NaaS	Network as a Service
NoSQL	Database not based on SQL, typically not relational database
NREN	National Research and Education Network
OASIS	Organisation for Advancement of Structured Information Standards
OBR	OSGi Bundle Repository
ODE	(Apache) Orchestration Director Engine
OGF	Open Grid Forum
OSGi	Open Services Gateway Initiative (now OSGi Alliance)
PaaS	Platform as a Service
PDP	Policy Decision Point
perfSONAR	PERFormance Service Oriented Network monitoring Architecture
POJO	Plain Old Java Object
QoS	Quality of Service

RADIUS	Remote Authentication Dial-In User Service (IETF standard)
RDF	Resource Description Framework
RDF/XML	The XML syntax for RDF
REST	Representational State Transfer
RST	Request Security Token
SAML	Security Assertion Markup Language (OASIS standard)
SeT	Session Token
SOA	Service-Oriented Architectures
SOAP	Simple Object Access Protocol
SP	Service Provider
SPARQL	Query language from the W3C for searching data defined in the RDF format
ST	Security Token
STS	Security Token Service
SVN	Subversion
TTL	Time to Live
TTS	Ticket Translation Service
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
W3C	World Wide Web Consortium
WS	Web Services
WSA	Web Services Architecture
WSDL	Web Services Description Language
WSS	Web Services Security
WST	WS-Trust
XACML	eXtensible Access Control Markup Language
XML	eXtensible Mark-Up Language
XSD	XML Schema Definition