24-05-2011

Deliverable DJ3.3.2 Composable Network Services Framework and General Architecture: GEMBus

Deliverable DJ3.3.2 v1.0

Contractual Date:	31-12-2010
Actual Date:	24-05-2011
Grant Agreement No.:	238875
Activity:	JRA3
Task Item:	T3
Nature of Deliverable:	R
Dissemination Level:	PU
Lead Partner:	RedIRIS
Document Code:	GN3-11-002
Authors:	Diego R. Lopez (RedIRIS), Yuri Demchenko (UvA), Krzysztof Dombek (PSNC), Mary Grammatikou (GRNET), Roland Hedberg (UMU.SE), Jordi Jofre (i2CAT), Constantinos Marinos (GRNET), Pedro Martínez-Juliá (UMU.ES), Antonio-David Pérez-Morales (RedIRIS), Vassiliki Pouli (GRNET), Elena Torroglosa (UMU.ES), Maja Gorecka-Wolniewicz (UMK-PSNC), Tomasz Wolniewicz (UMK-PSNC), Bartłomiej Idzikowski (PSNC), Maciej Glowiak (PSNC), Shannon Milsom (DANTE)

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

This deliverable presents the framework and general architecture for GEMBus, the GN3 federated multi-domain service-oriented architecture. 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.

Table of Contents

Table of Figures

Table of Tables

Executive Summary

This document presents the framework and general architecture for GEMBus (GÉANT Multi domain Bus), the federated multi-domain service-oriented infrastructure being developed in the GN3 project. 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. It is founded on a Composable Service Architecture (CSA), based on a general framework for composite services, and on the industry adopted Enterprise Service Bus (ESB), extended to support dynamically reconfigurable virtualised services. The architecture addresses multi domain issues and distributed services composition and orchestration.

GEMBus follows SOA and ESB principles. Service-Oriented Architecture (SOA) allows managing, maintaining and accessing heterogeneous and geographically sparse resources in a unified way by providing standardised interfaces and common working environments to their users. The heterogeneous nature of these resources spans across not only different providers or administrative domains, but across different application domains as well, aiming (for example) at the integration of bandwidth reservation mechanisms with storage allocation procedures into what users should perceive as a single service.

One of the key components of SOA architecture is the ESB concept. The functionalities of an ESB are comparable to those of a physical bus that carries bits among devices in a computer. This means that in an architecture that uses an ESB, all communications are handled via the ESB, which acts as a broker between applications, enabling the integration of services and applications.

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. This enables them to concentrate on the direct implementation of business processes. Most current ESB frameworks are oriented to single enterprise deployment that relies on a central top authority. GEMBus aims to bring the advantages of these ESB frameworks into an open collaborative environment, taking a step further into federated infrastructures and supporting the definition of a multi-domain ESB infrastructure, a “bus of buses”.

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

Lifecycle management is an important part of the CSA and is key 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 complicated process. One of the objectives of GEMBus is to ease the integration of existing service platforms in the GÉANT infrastructure and user communities. This document describes the integration patterns that the GEMBus team identified during their experiments to define the aspects of the architecture and interface mechanisms GEMBus offers to application or computing elements using its services.

To conclude, the alignment of the GEMBus architecture proposal to the most relevant standards, as well as in the context of other activities within the GN3 project, is analysed. Finally, a detailed description of the most elaborated experiment so far, available as a practical case demonstrator of the GEMBus potential, is included in the Appendix. The work described there is a step towards achieving the challenge of building self-managed systems by providing the necessary services onto GEMBus.

1 Introduction

This document presents the architecture for the federated multi-domain service-oriented infrastructure under development in the GN3 project, GEMBus (GÉANT Multi-domain Bus), with the goal of enabling collaboration among the different actors in the European research networks (and beyond) by sharing services and allowing them to compose more complex ones. This architecture is based on a general framework for composable services, founded on the industry adopted Enterprise Service Bus (ESB) and extended to support dynamically reconfigurable virtualised services. The architecture addresses multi-domain issues and distributed services composition and orchestration.

The bus paradigm provides the additional advantage of freeing service developers from dealing with common aspects such as security, service discovery and message management. This enables them to concentrate on the direct implementation of business processes. Most (if not all) current frameworks are oriented to single enterprise deployment that though can be extremely complex, rely on a central top authority. GEMBus intends to bring the advantages of these frameworks into an open collaborative environment, taking a step further into federated infrastructures and supporting the definition of a multi-domain infrastructure, a “bus of buses”.

The experience in deploying federated architectures dictates a strict adhesion to the principle of keeping simplicity as the topmost design goal to ease integration of disparate participant infrastructures and to facilitate interoperation at the common agreed level. With this principle in mind, very few requirements are made for an infrastructure to become part of GEMBus. Most interoperation mechanisms are regarded as end-to-end issues, though GEMBus commits itself to provide mediation services for location, authentication, authorisation, accounting and composition. The components taking care of these mediation services are termed the GEMBus core. They are intended to provide support to services participating in GEMBus along their whole lifecycle.

The process for service integration can be complicated and time-consuming in many cases. This document describes the integration patterns that the GEMBus team has identified during the experiments run to decide the aspects of the architecture presented here. Conversely, discussion is provided on the interface mechanisms that GEMBus offers to any application or computing element willing to make use of its services.

To conclude, the alignment of the GEMBus architecture proposal to the most relevant standards, as well as in the context of other activities within the GN3 project, is analysed. Finally, a detailed description of the most elaborated experiment so far, available as a demonstrator of the GEMBus potential, is included in the Appendix. The work described there is a step towards achieving the challenge of building self-managed systems by providing the necessary services onto GEMBus.

2 General Architecture Framework for GEMBus

Service-oriented architectures constitute a significant step in the evolution of distributed computing. Based on the request/reply design paradigm for synchronous and asynchronous applications, SOA proposes the modularisation of application business logic and individual functions, providing them as services for consumer/client applications. Designing according to SOA principles consists essentially in connecting various services according to common business rules. This makes SOA especially suitable for loosely coupled distributed applications that communicate with each other to offer interoperability between distributed systems. It is a paradigm that enables interoperability between heterogeneous and diverse systems and reduces the complexity of administering their coordinated operation.

SOA encourages software components development for on-demand consumption and typically relies on using well-known high-level transport protocols[1], establishing itself as an extremely flexible connection technology. SOA principles also influence the business logic of services by encouraging good design, i.e., promoting modular and dispersed components that can be separately developed and maintained. Services exposed by the SOA integration capabilities can be composed into composite services spanning multiple domains and organisations that are, in turn, subject to further compositions. This provides an extremely powerful and flexible toolset to application developers.

SOA, viewed as a tool for the integration of distributed resources, plays a significant role, not only as manager of computational resources, but increasingly as aggregator of measurement instrumentation and pervasive large-scale data acquisition platforms, such as sensor networks. In this context, the functionality of service-oriented architecture allows managing, maintaining and exploiting heterogeneous and geographically sparse instrumentation and acquisition devices in a unified way, by providing standardised interfaces and common working environments to their users.

This is achieved through the properties of isolation from the physical network and from the peculiarities of the instrumentation, granted by standard middleware, together with secure and flexible mechanisms to seek, access and aggregate distributed resources. The service-oriented architectures concepts that build upon service-oriented architecture abstractions are typically specified using XML-based standards and recommendations. They have the following key characteristics.

[1] Web Services, using the HTTP protocol and structured mark-up languages, constitute the archetypal example of this approach.

SOA services communicate with formally defined messages. Communication among consumers and providers typically happens in heterogeneous environments with little or no knowledge about the other party. Implementers commonly build SOAs using web services standards, such as Simple Object Access Protocol (SOAP) [SOAP] and XML-based standard protocol developed by the World Wide Web Consortium (W3C) [W3C] and supported by the main IT players, that defines a standardised message envelope and a rather free message payload. Apart from SOAP, it is possible to implement SOA communication using any service-based technology, such as REST [REST]. REST services operate on URLs that may respond with XML messages as well as other formats. JavaScript Object Notation (JSON) [JSON] is an alternative to XML for message exchange.

SOA services are maintained in the enterprise by a registry that acts as a directory listing to store and publish services. SOA services have self-describing interfaces in platform-independent XML documents. Web Services Description Language (WSDL) [WSDL] is the standard one for service description when services are SOAP-based. Web Application Description Language (WADL) [WADL] is the standard XML-based format for RESTful web services. There is also Universal Service Description Language (USDL) [USDL], a language for formally describing the semantics of web services to become more practical, enabling users and applications to discover, process, deploy and synthesise services more easily.

2.1 Composable Service Architecture

Composable Service Architecture (CSA) [CSA] is the general framework on which GEMBus development is founded. It is based on the industry adopted Enterprise Service Bus (ESB) [ESB], extended to support dynamically reconfigurable virtualised services. It addresses multi-domain issues and distributed services composition and orchestration. CSA is based on the basic SOA architectural principles and services interaction models.

In its evolution and gradual development, GEMBus will adopt SOA best practices and comply with the Open Group Services Integration Maturity Model (OSIMM) [OSSIMM]. The OSIMM technical standard defines a grid of seven maturity level and seven dimensions that describe provisioned services and SOA related properties. The OSIMM maturity levels include:

- Silo
- Integrated
- Componentised
- Services
- Composable services
- Virtualised services
- Dynamically re-configurable services

The seven dimensions define different layers and aspects of the services such as:

- Business view
- Governance and Operations

- Methods
- Applications
- Architecture
- Information
- Infrastructure and Management

In the Applications dimension, the SOA-based applications deal with the different components and building blocks mapped to the above defined maturity levels:

- Modules (OSIMM1)
- Objects (OSIMM2)
- Components (OSIMM3)
- Services (OSIMM4)
- Applications composed of services (OSIMM5)
- Process integration via services (OSIMM6)
- Dynamic application assembly (OSIMM7)

Starting from the level OSIMM4 (services) the information or data are represented as “Information as a service” (OSIMM4), “Enterprise Business Data dictionary and repository” (OSIMM5), “Virtualised data services” (OSIMM6) and “Semantic data vocabularies” (OSIMM7). Table 2.1 provides a tabular summary of the service models at different OSIMM levels.

OSIMM **Levels and Dimensions**	**OSIMM1** **Silo**	**OSIMM2** **Integrated**	**OSIMM3** **Componentised**	**OSIMM4** **Services**	**OSIMM5** **Composable services**	**OSIMM6** **Virtualised services**	**OSIMM7** **Dynamically re-configurable services**
Business view	Isolated business lines	Business process integration	Componentised business	Componentised business offers services	Processes through services composition	Geographical independent service centres	Mixed match business and context-aware capabilities
Organisation	Ad hoc IT strategy and Governance	Ad hoc enterprise strategy and Governance	Common Governance process	Enabling SOA Governance	SOA and IT Governance Alignment	SOA and IT Infrastructure Governance Alignment	Governance through policy
Methods	Structured analysis and Design	Object Oriented Modelling	Component based development	Service Oriented Modelling	Service Oriented Modelling	Service Oriented Modelling for Infrastructure	Business Grammar Oriented Modelling
Applications	Modules	Objects	Components	Services	Applications composed of services	Process integration via services	Dynamic assembly, context-aware invocation
Architecture	Monolithic architecture	Layered architecture	Component architecture	Emerging SOA	SOA	Grid based SOA	Dynamically re-configurable architecture
Information	Application specific	LOB or enterprise specific	Canonical models	Information as a service	Enterprise Business Data dictionary and repository	Virtualised data services	Semantic data vocabularies
Infrastructure (and Management)	LOB Platform specific	Enterprise standards	Common re-usable infrastructure	Project based SOA environment	Common SOA environment	Virtual SOA environment, S&R	Dynamic sense, Decide and Respond

Table 2.1: Service models at OSIMM levels

A general SOA solution qualifies for OSIMM5/OSIMM6. The extensions proposed by CSA are intended to provide the functionality to achieve the highest level of maturity.

2.1.1 GEMBus in the CSA

CSA provides a framework for the design and operation of composite services provisioned on-demand. It is based on component service virtualisation which, in turn, is based on the logical abstraction of the component (physical/real) services and their composition. Composite services may also use orchestration services provisioned as a CSA infrastructure service to operate composite service specific workflows. One of the important components of the proposed architecture is the CSA middleware that should ensure smooth service operation during all stages of the service lifecycle.

To be included in the CSA infrastructure, component services need to implement an adaptation layer that is capable of supporting major CSA provisioning stages, in particular, service identification, service metadata (including the required security context) and provisioning session management. CSA also defines and implements special adaptation layer interfaces. This provides the necessary functionality to support dynamically provisioned control and management.

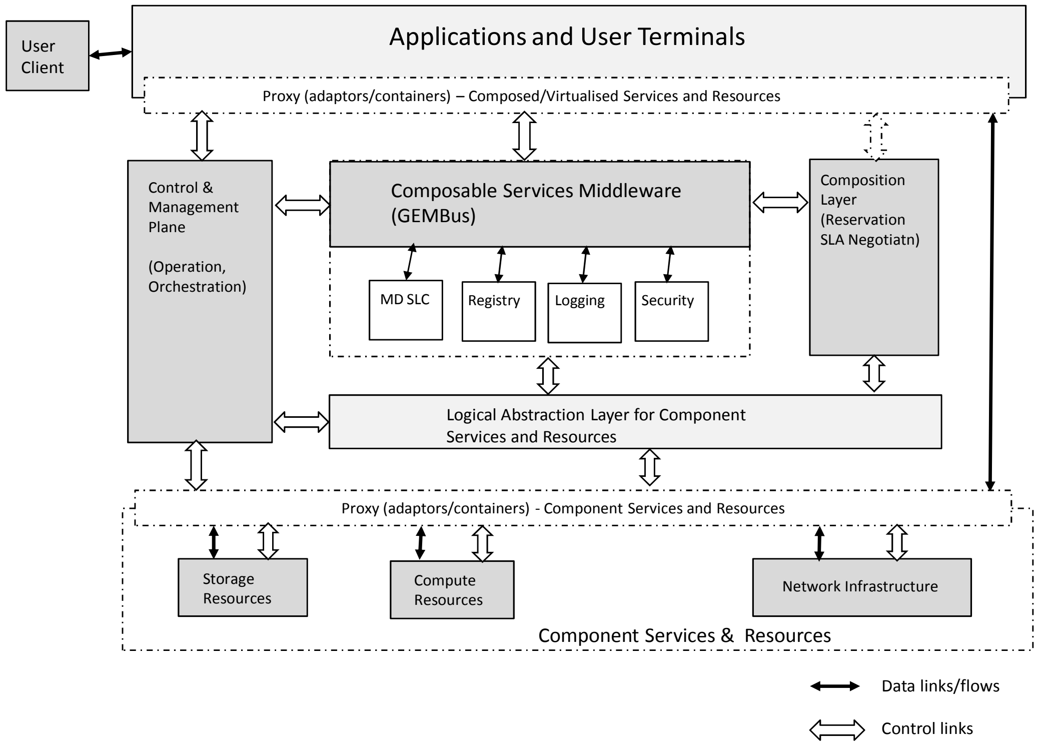

User Client
Applications and User Terminals
Proxy (adaptors/containers) – Composed/Virtualised Services and Resources
Control & Management Plane (Operation, Orchestration)
Composable Services Middleware (GEMBus)
Composition Layer (Reservation SLA Negotiatn)
MD SLC
Registry
Logging
Security
Logical Abstraction Layer for Component Services and Resources
Proxy (adaptors/containers) - Component Services and Resources
Storage Resources
Compute Resources
Network Infrastructure
Component Services & Resources
Data links/flows
Control links

Figure 2.1: Composable Service Architecture and main functional components

The middleware is a key component of the CSA that provides a common interaction environment for both component services and composite services. Besides exchanging messages, CSA middleware also contains/provides a set of common infrastructure services required to support reliable and secure (composite) service delivery and operation. The ongoing GEMBus development will provide a generic reference CSA middleware implementation. Figure 2.2 illustrates the functional architecture for GEMBus, including its three main functionality groups:

- The GEMBus Messaging Infrastructure (GMI). This includes a messaging backbone and other message handling supporting services such as message routing, configuration services, secure messaging and event handler/interceptors. The GMI is built on and extends the generic ESB functionality to support dynamically configured multi-domain services.
- GEMBus core infrastructure services. These support reliable and secure composable service operation and the whole service provisioning process. These include such services as Registry, Composition, Security and Logging, all provided by the GEMBus environment itself.
- Component services. Although typically operated by independent parties, they need to implement special GEMBus adaptors or use special plug-in sockets that allow their integration into the GEMBus CSA infrastructure.

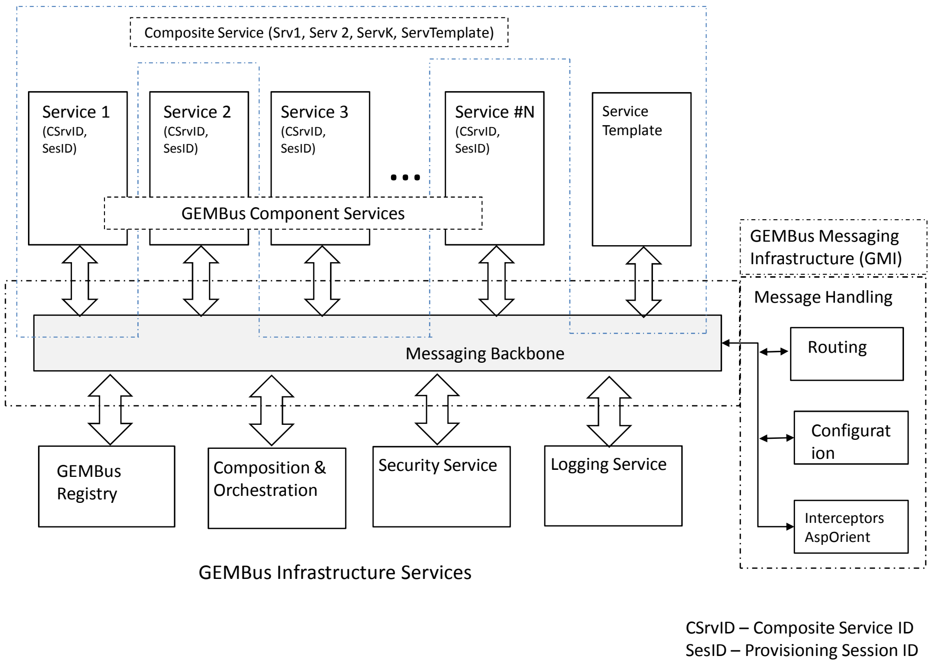

Composite Service (Srv1, Serv 2, ServK, ServTemplate)
Service 1 (CSrvID, SesID)
Service 2 (CSrvID, SesID)
Service 3 (CSrvID, SesID)
Service #N (CSrvID, SesID)
Service Template
GEMBus Component Services
GEMBus Messaging Infrastructure (GMI)
Message Handling
Messaging Backbone
Routing
Configurat ion
Interceptors AspOrient
GEMBus Registry
Composition & Orchestration
Security Service
Logging Service
GEMBus Infrastructure Services
CSrvID – Composite Service ID
SesID – Provisioning Session ID

Figure 2.2: GEMBus functional architecture

2.2 Realising GEMBus: Bus of Buses

The actual realisation of GEMBus as an SOA multi-domain middleware, able to support the deployment and composition of services spanning different management domains, calls for applying the federation mechanisms that have come to play a key role in the collaborative environment of current academic networking. Federation preserves management independence for each party as long as they obey the (minimum) set of common policies and technological mechanisms that define their interoperation. Metadata constitute the backbone of such federations, as they describe the components provided by each party and their characteristics in what relates to the federated interfaces.

The ESB paradigm for SOA implementation provides the additional advantage of freeing service developers of dealing with common aspects such as security, service discovery and message management. This enables them to concentrate on the direct implementation of business processes. Most (if not all) current ESB frameworks are oriented to single enterprise deployment that, although it can be extremely complex, relies on a central top authority. GEMBus intends to bring the advantages of ESBs into an open collaborative environment, taking a step further into federated infrastructures and supporting the definition of a multi-domain ESB, a “bus of buses”.

Figure 2.3Figure 2.3 depicts the general deployment architecture for GEMBus, where different ESB instances (of possibly different ESB frameworks) at different management domains are federated. Services in one of the instances can seamlessly access those services made available by the others via the federation.

A common service registry provides the metadata backbone for the federated bus. Service descriptions are updated at the common registry and made known at the participant instances through the local registries. They ensure absolute management independence for each participant. To support different ESB frameworks and implementation styles, the registry must provide a semantically rich internal format, supporting updates and queries by disparate means, while maintaining a coherent ontology for the services available at the federated ESB.

Since any service integrated in GEMBus offers its interfaces according to SOA principles, services integrated in other SOA frameworks (essentially web services) will be able to access GEMBus services as well by querying the registry using their own means, though the added value provided by GEMBus integration will not be available: neither integration can be complete, nor pure business logic orientation in service development seems attainable.

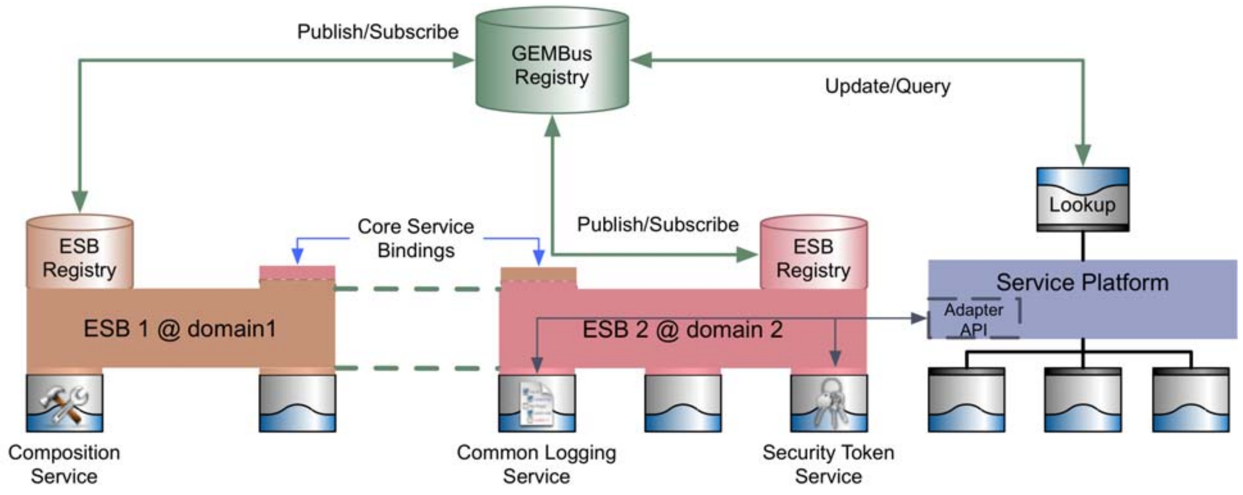


Figure 2.3: General GEMBus deployment architecture

Core infrastructure services can be deployed at any participating instance, providing additional properties on resilience, load balance and flexibility. The architecture supports the existence of *special-purpose* or *common* ESB instances to provide specific or somewhat closely related services, or as an additional security measure. In some cases, seamless access to these core services in such a distributed manner may require the implementation of specific interfaces as service binding components.

Component services can be statically deployed by their developers at a specific ESB instance or dynamically instantiated through service bundles available at specific repositories, connected with code repositories so the development activities can be integrated as well in the service lifecycle. OSGi [OSGI] is the most promising technology to achieve these goals. Such repositories have already been demonstrated elsewhere [OSAMI]. Through integration with the GEMBus registry, these repositories will become the basic support for service lifecycle management.

The experience in deploying federated architectures dictates a strict adhesion to the principle of keeping simplicity as the topmost design goal to ease integration of disparate participant infrastructures and to facilitate interoperation at the common agreed level. With this principle in mind, requirements for an SOA infrastructure willing to become part of GEMBus are limited to a few basic aspects related to the following:

- Messaging: in aspects associated with routing and traceability.
- Security: in what corresponds to the protocol used to exchange and collect statements.
- Service descriptions: in the specification elements to make them available at the registry.

All other interoperation mechanisms are regarded as end-to-end issues, involving the service consumer(s) and producer(s), though GEMBus commits itself to providing mediation services that, based on the registry, are able to deliver the following:

- Integrated authentication and authorisation.
- Service access and usage accounting.
- Composition mechanisms.

Any other service, interface or constraint on participating services shall be considered as added value that user applications are free to use or honour, configuring what can be seen as specific user communities inside GEMBus, either as part of specialised buses with a tighter coupling that takes advantage of the common GEMBus services or as a service cloud oriented to specific usage patterns or application domains.

3 The GEMBus Core

The GEMBus core is constituted by those elements that provide the functionality required to maintain the federation infrastructure, allowing the participant SOA frameworks to interoperate in accordance with the principles previously described. This section describes these elements, how they are used to establish the GEMBus foundations and how they can be used by the participating services.

The GEMBus core comprises 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 STS or the Workflow Server 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.

3.1 GEMBus Registry

The registry is an absolutely vital component of GEMBus. Distributed services cannot be used if there is no way to find them and learn what they can do. Since GEMBus is a multi-domain, multi-protocol environment the demands on the registry are especially severe.

3.1.1 Information Architecture

JRA3 T3 expects GEMBus to be used, at least initially, to bring together already existing services. This means that the GEMBus team must deal with service descriptions that are made in already existing formats such as WSDL and WADL. It is important to note that the task is to store these in the registry with metadata about the service.

Why is metadata needed?

WSDL, for example, can specify the operations available through a web service and the structure of data sent and received, but it cannot specify the semantic meaning of the data or semantic constraints on the data. This requires programmers to reach specific agreements on the interaction of web services and makes automatic web service composition difficult [SEWSR].

To store this extra information the Web Ontology Language (OWL) [OWL]/Resource Description Language (RDL) [RDF] combo has been selected. There are many reasons to use OWL, including:

- The larger expressiveness compared with (as an example) XSD (XML Schema Definition).
- Simpler version handling.
- The ease in which to add rules on top of the ontology using OWL.
- Facilitate more complex searches and reasoning algorithms.

As a consequence, service information can be published, retrieved and interlinked on the Web according to the principles of Linked Open Data [LINKEDDATA].

3.1.2 Service Ontology

At present, there is no clear candidate for the position of service ontology. There are many candidates with advantages and disadvantages. The eventual choice will not change the architecture of the service, although it will affect what information can be stored.

Apart from the service ontology, ontologies for describing organisations, persons, pricing, legal requirements, etc. are needed. Some of these can be built on existing ontologies such as those defined by the FOAF project [FOAF]. Others will be constructed as needed.

3.1.3 Import/Export

In many cases GEMBus will act as the glue between ESBs. The ESBs normally have a registry of their own, so a way must be defined for importing service descriptions from the ESB registry to the GEMBus registry. In addition, a way to export service descriptions from the GEMBus registry to an ESB registry must be defined. Both of these flows may be subject to filtering, so not all services descriptions will be shared by all the registries.

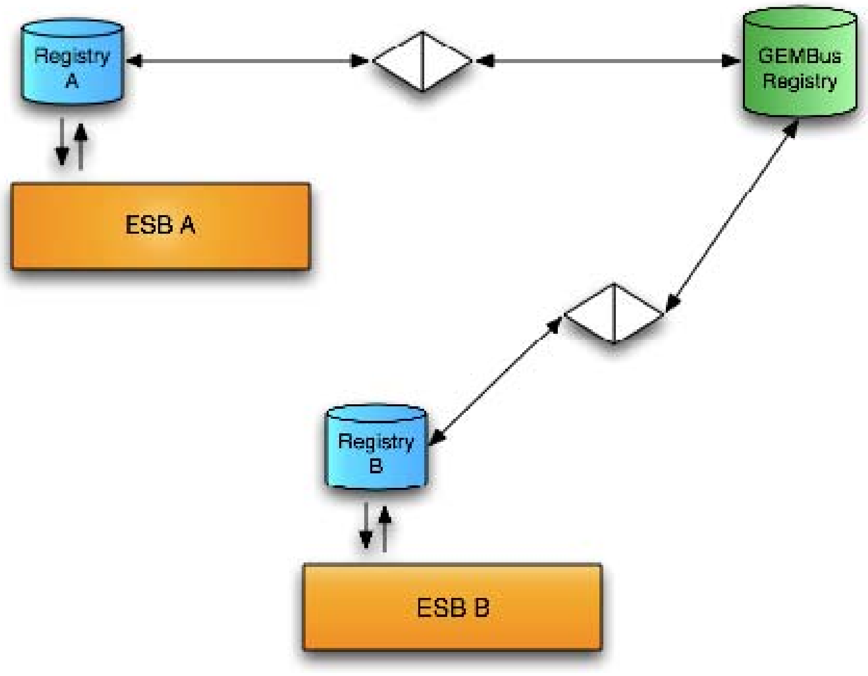


Figure 3.1: Registry imports and exports

Since the GEMBus registry provides more information about the services and their environment than the ESB registry does, the GEMBus registry must be accessible directly. As it is expected to work in an already existing environment, it is not appropriate to make bold assumptions as to which protocols people should use. The most commonly used protocols should be supported, such as the following:

- Atompub/Opensearch
- RESTful queries
- SPARQL

Note that when this document refers to the GEMBus Registry, it does not imply that only one is envisaged. It is most likely that there will be more than one. It is expected that these registries exchange information and synchronise. JRA3 T3 envisages using Pubsubhubbub [PUBSUBHUBBUB] for this communication to make it easy to set up a new registry and pull necessary service descriptions from existing registries.

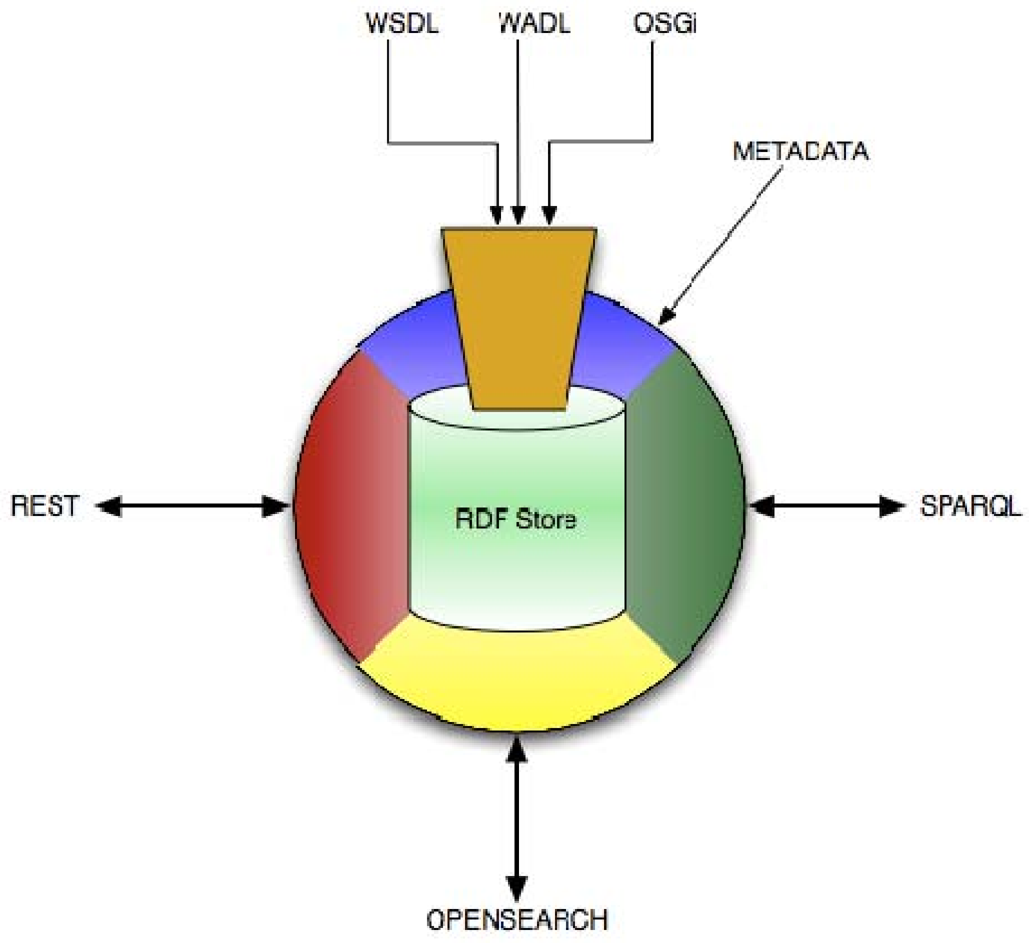


Figure 3.2: Inputs and outputs of the GEMBus Registry

3.2 GEMBus Messaging Infrastructure

Existing ESB frameworks and implementations incorporate a centralised model for message handling where a domain central message processor (that also provides inter-domain message routing when required) processes all intra-domain messages. Message processors in this case act as a collapsed messaging backbone. This means that to send or receive messages all services need to connect to one of the adapters supporting specific service related/defined message-level protocols. These adaptors are typically connected to an assigned port.

The GEMBus Messaging Infrastructure (GMI) extends this functionality in the multi-domain environment considered by GEMBus by means of the following major structural and functional components:

- A Message Processor, through adequately tailored standard ESB implementations.
- Service interfaces/adapters, connecting component services that may use different native interfaces. These include the mechanisms for message format transformations and data mappings, though these functionalities will typically be configured in separate ways.

- The context handling for messages and services, including session support.
- Aspect based interceptors and event handlers, provided as specific configurations to standard implementations.
- A Message Routing Module, by means of a standard message router deployed and configured accordingly.
- A Configuration Manager that supports the dynamically configuration of GMI specific components to enable a smooth GEMBus operation.

The Registry will store and provide the GMI components with service information on location, configuration and properties. These components will make use of the Registry either by using the local, internal ESB registries for local component services, or by directly querying the common GEMBus Registry service. These components will make direct use as well of the GEMBus Logging Service (GLOS) logging facility to track message exchange in a way that is transparent to service developers and operators. See section 3.5 for more information.

Figure 3.3 illustrates the GMI functional diagram and the interactions among its main functional components. The architecture is conceived to facilitate the maximum reuse of existing ESB frameworks and to achieve the required GMI functionality by adding new pluggable modules or providing appropriate configuration information to a typical ESB implementation. Figure 3.3 shows the functionality of the standard ESB implementation as available at the inner Message Processor.

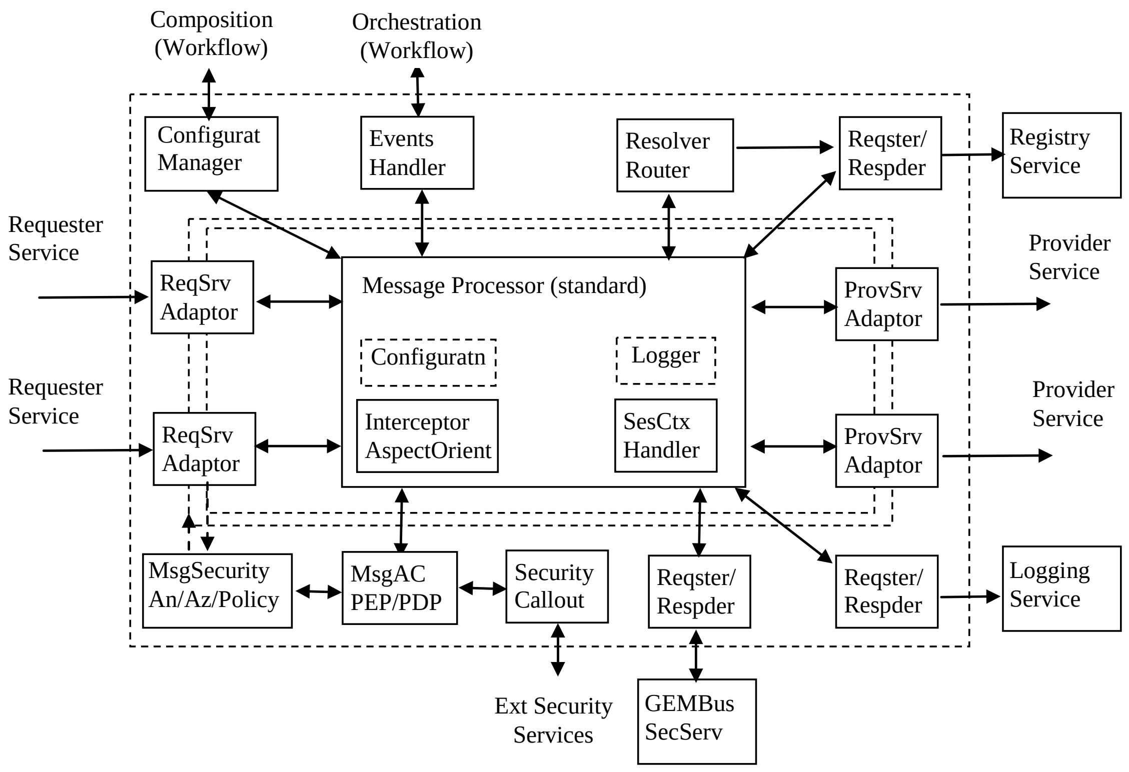


Figure 3.3: Main GEMBus Messaging Infrastructure (GMI) functional components

GMI provides a transparent re-configurable message exchange environment for service interaction and integration with the following core functionality:

- GMI configuration allows service virtualisation and dynamic configuration (without the whole GEMBus infrastructure re-deployment). Common GMI configuration patterns can be defined as GMI profiles, identified by designated names that can be referred by GEMBus services.
- GMI supports event driven services and allows configurable event recognition, alarming and recording, using standard ESB event handler and interceptors that can be configured via well-known mechanisms. This functionality is specifically required for, and targeted to, support service orchestration, workflow management, logging and accounting.
- GMI allows integration of SOAP and REST-based services by supporting the corresponding protocols. GMI supports transparent mapping between these two types of services, in particular for composite services to allow integration.

Routing takes place at the messaging layer and uses services identification, both as an endpoint reference (EPR) for SOAP messages and as a URL (used in RESTful services). Appendix A describes the conventions that define names and identifiers for the GEMBus EPRs, entities, elements, attributes and properties.

3.2.1 Security Considerations

The GMI provides the basic message level security using WS-Security mechanisms to protect message integrity and confidentiality. However, it also provides integration with the lower layer Transport level security and support for protocols as TLS/SSL. The GEMBus backbone could be run in a separate VPN domain connected to the participating domains. It is important to note that the collapsed backbone architecture requires the intra-domain messaging exchange environments to be secured and trusted at the messaging level.

The GMI security services are an integral component of the overall GEMBus infrastructure and they support the whole service lifecycle. Consequently, they support session-related security context management. GMI provides special functionality for security session context management by securely binding the session security context to the session ID. This is supported both at the GMI level and the GEMBus service level provided by the GEMBus Security Token Service (STS).

EPRs contain information critical to service operation. Therefore, all or some of their components must be protected. Integrity and confidentiality of the EPR in total or its elements can be achieved by applying standard XML Signature and XML Encryption methods.

3.3 GEMBus Security Services

Security mechanisms must comply with requirements that may conflict with security, privacy and simplicity of use. It is important that the security protocols deal with user attributes and related information in an appropriate manner, taking the conservative disclosure of attributes and abiding to user privacy policies whenever possible. It is also important that these directives are enforced by all entities, both in the infrastructure itself and in the participant services, dealing with user data in a consistent manner. From the point of view of services, is very important to protect information by ensuring the identity of consumers who use the services. The most adequate manner to satisfy these requirements relies on the use of a token that allows the transfer of security data along the exchanged messages.

3.3.1 General Design

The mechanisms needed to provide secure communications within the GEMBus architecture base their operation on the STS. This service, described in WS-Trust, makes it possible to issue and validate security tokens. The GEMBus STS will be aligned with the WS-Trust interoperability profile defined by SWITCH and the EMI [WSP].

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]. It is a flexible and feature-rich extension to SOAP to apply security to web services. The protocol specifies how integrity and confidentiality can be enforced on messages. It allows the communication of various security token formats, such as SAML [SAML], Kerberos [KERBEROS] and X.509 [X509]. Its main focus is the use of XML Signature and XML Encryption to provide end-to-end security. The protocol is officially called WSS and is developed via committee in Oasis-Open. It is associated with the following approved specifications: WS-Trust, WS-SecureConversation and WS-Policy.

WS-Trust [WST] is a WS-* specification and OASIS standard that provides extensions to WS-Security, specifically dealing with issuing, renewing and validating 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 an STS: A web service that issues security tokens as defined in the WS-Security specification.
- The formats of the messages used to request security tokens and the responses to those messages.
- Mechanisms for key exchange.

In the GEMBus STS, a different element carries each of these actions. The Ticket Translation Service is responsible for generating valid tokens in the system according to the received credentials. Token validation is performed by the Authorisation Service, which can also retrieve additional attributes or policy rules from other sources to perform the validation.

The Ticket Translation Service (TTS) mostly[2] relies on external identity providers that must verify the identity of the requester based on valid identification material. To support a large amount of services, the application of different authentication methods must be ensured. This must include the support of currently standardised authentication methods as well as methods incorporated in future. In this respect, there will be a direct usage of the eduGAIN identity federation services. eduPKI, TCS and other IGTF accredited identity infrastructures will be a key starting point.

The Authorisation Service (AS) is responsible for checking the validity of the presented tokens. In this case, the requester is usually a service that has received a token along with a request message and needs to check the

[2] Certain authentication mechanisms, such as those defined in the DAMe and Moonshot profiles, probably require a direct identification by the module.

validity of the token before providing a response. Checks carried out on the token can be related to issue date, expiration date or signature(s). This 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.

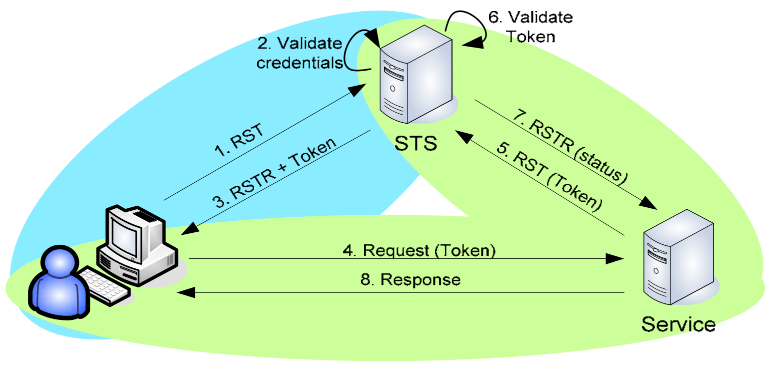


Figure 3.4: Generic use case for STS

Figure 3.4 depicts an example of the messages exchanged when one user tries to access a service using tokens to secure the connection.

First, the service consumer initialises and sends an authentication request to the STS. The STS then validates the consumer credentials and issues a security token to it. With the token, the consumer sends a request message including the token to the producer. The consumer sends the token to the STS to check its validity. After running its validation process, the STS sends a response with the status of the token to the producer, which processes it and replies to the consumer.

3.3.2 Token Description

The WS-Security specification allows a variety of signature formats, encryptions algorithms and multiple trust domains. It is open to various security token models, such as X.509 certificates, userid/password pairs, SAML assertions and custom-defined tokens.

The GEMBus TTS will support transformations among different token formats, according to service descriptions as stored in the Registry. Appropriate profile definitions will describe these formats. Nevertheless, the canonical GEMBus security token (applicable by default in all GEMBus-supported exchanges) is the relayed-trust SAML assertion originally defined within the GN2 project to provide identity information in scenarios where a service is acting on behalf of a user identified through an identity federation.

The SAML construct used in this case is able to convey information about the user accessing the producer. It fulfils two essential constraints:

- It must be bound to the consumer by the IdP, so it is possible to check that the information it contains about the user has been legally obtained.

- It must be bound to the producer by the consumer, so a potentially malicious producer cannot use this information to further impersonate either the consumer or the user.

To comply with these two requirements, the producer sends a SAML assertion expressing data related to the authentication with:

- A valid audience restricted to the resource it is addressed to, through a SAML `condition` element containing an URI uniquely identifying the resource.
- A statement that this specific method of relayed trust must be used to evaluate the assertion, through a specific value in the SAML construct identifying the subject confirmation method. This value is the URI in the eduGAIN namespace: `urn:geant:edugain:reference:relayed-trust`.
- The SAML assertion(s) received from the IdP as evidence for this confirmation process, as part of the SAML element `SubjectConfirmationData`.

A sample SAML assertion generated according to these procedures is shown in Appendix B.

3.3.3 Ticket Translation Service (TTS)

The ticket translation service (TTS) is responsible for issuing, renewing and converting security tokens, responding to consumer requests for issuing, renewing or converting security tokens for services that require it.

Each of these operations can only be done by the TTS (unlike security token validation that can be done either by the own service or at the framework integration elements such as interceptors, message routers or binding components).

The main TTS operations are:

- Issuing: To obtain a security token from an identity credentials (Identity Token).
- Renewing: To renew an issued security token.
- Converting: To convert a security token type to another security token type.

The TTS operation is as follow:

1. The consumer obtains an identity token (SAML Assertion, eduGAIN token, etc.) from an identity infrastructure.
2. The consumer sends a request for issuance, renewal or conversion to the TTS using either the Identity Token (issuance) or a Security Token (renewal or conversion).
3. The STS validates the consumer’s token (using security policies) and sends a security token to the consumer.

3.3.4 Authorisation Service

The AS is responsible for supporting the token validation functions, responding to requests for validating tokens of consumers and services that require it.

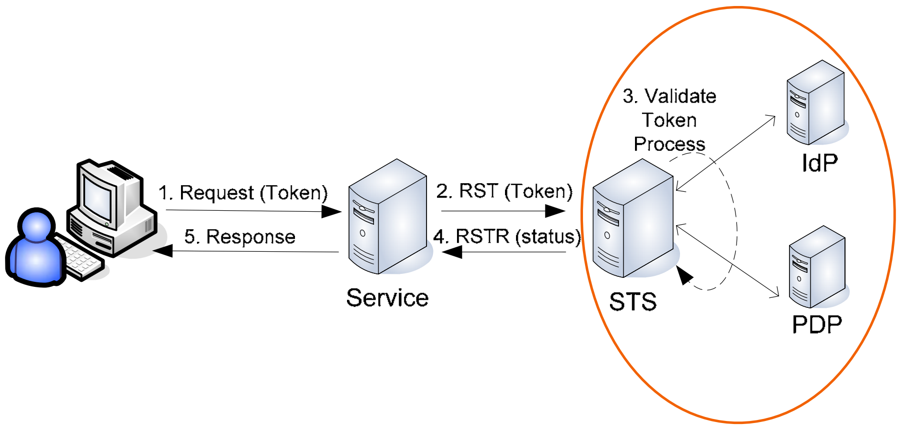


Figure 3.5: STS: Authorisation Service

The token validation process can be performed by the STS itself or act as a proxy redirecting the validation process to the external service that generated it. For external validation, the Authorisation Service consults an external service or IdP and forwards the response to the STS consumer. As Figure 3.5 shows, when the Authorisation Service itself performs validation, the process must verify the information contained in the token checking the issuer, issue and expiration date, signatures, etc. In addition to the token, the Authorisation Service can perform a more complex authorisation process, retrieving attributes related to the token subject and consulting a Policy Decision Point (PDP) for authorisation decisions.

3.3.5 Session Control between SP and STS

Session control is the process of keeping track of consumer activity across different levels of interaction with the producer.

Assuming that each message to a service is attached with a token that the service must validate at the Authorisation Service, this will very likely mean a high workload for the STS. The objective of managing GEMBus sessions is to speed up the security system performance without compromising security goals.

There are several ways to strengthen the validation of the tokens based on the idea of the sessions. First, optimisations can be applied to the token validation mechanism done by the STS. One proposal is that the STS temporarily stores a reference to each token validated. If, a short time later, the STS returns to receive a request for the same token, it does not need to revalidate the token. The idea is close to the use of a cache, obtaining a performance similar to a session as long as the reference is in the listing. This improvement has the advantage that it does not involve changes in the requesters that make use of STS.

On the other hand, it is possible to include a new type of token called session token that is returned to the requester after successful validation in the STS. The main feature of this type of token is rapid validation at the expense of lower security features compared to a normal token, though this can be alleviated (if not solved) by reducing its lifetime. When the requester makes a new request for validation to the STS, it can include the two tokens or just the Session Token. When the STS receives the query, the AS first checks the Session Token and, if it is valid, the STS can respond directly to expedite the process. In contrast to the previous mechanism, this method involves the adaptation of STS requesters to manage the Session Token.

3.3.6 Integration Flows

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 it can also act as a container, it is possible to develop and deploy a service directly on the bus. But it is more interesting to exercise its integration capabilities, 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 abovementioned mechanisms.

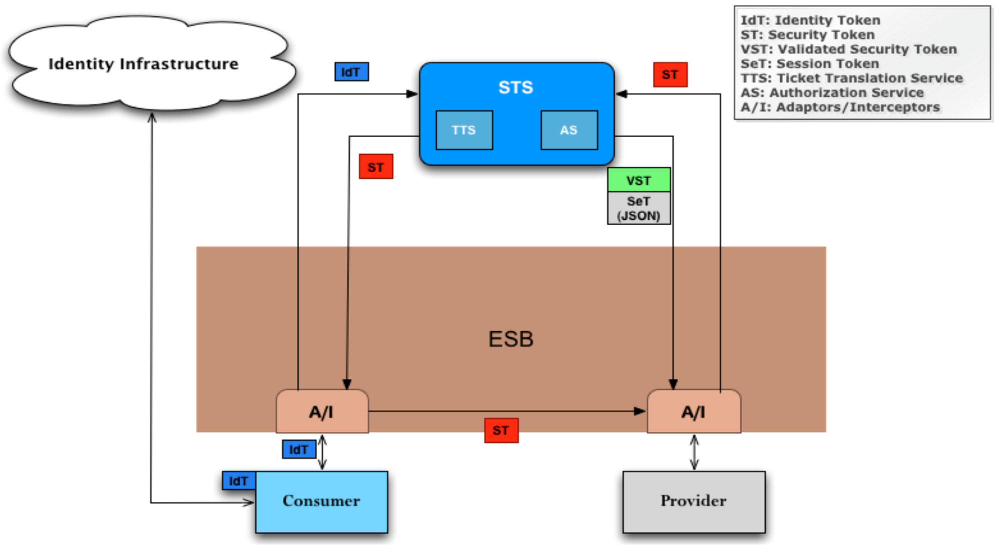


Figure 3.6: GEMBus integration scheme

Figure 3.6 illustrates a scenario in which a Security Token Service 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. Then it 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 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 SPs deployed in GEMBus can validate the tokens themselves by contacting the STS.

Figure 3.7 shows two more detailed interaction diagrams, illustrating how the federated ESB nature of GEMBus can be applied to get advantage of distributing the STS components in different bus instances and of existing identity infrastructures. It is important to note that whenever an external message enters the bus, it must be translated into internal format by an adapter. The reverse process should be done with messages that must leave the bus to be comprehensible to external entities.

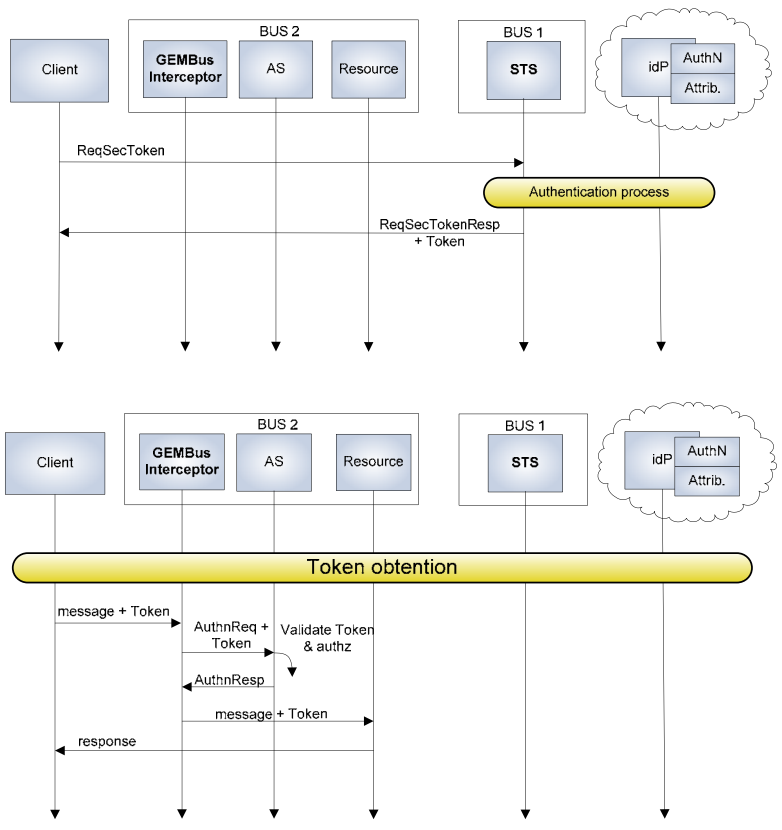


Figure 3.7: STS integration diagrams in a federated deployment

3.4 Composition

Based on an SOA, GEMBus will comprise a group of loosely coupled, reusable, composable and (probably) distributed services. There is a need for a feasible way to compose those services to build up more complex and smarter services. That functionality will be provided through the Composition core service, which will enable GEMBus to aggregate multiple simple services, as well as other compositions, into new services.

Figure 3.8 illustrates how service composition is offered as a service on itself, so composite services can be transparently accessed and instantiated by consumers.

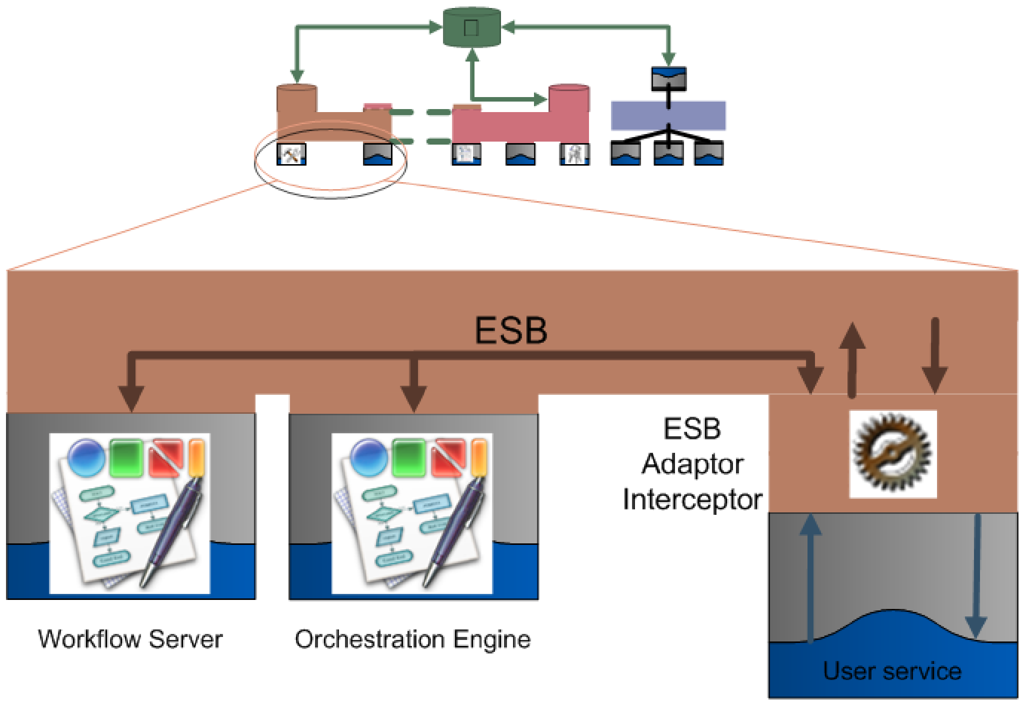


Figure 3.8: Accessing composition services

When talking about service composition, there are two main approaches, Choreography and Orchestration, of which the latter is the most interesting, as it is a way to implement choreographies. An orchestration is an executable description of services interactions and messages manipulation, so an execution engine and a description language are required.

Continuing with the standards basement of GEMBus, WS-BPEL [WSBPEL] is the OASIS standard for the specification of executable and abstract business processes. While the executable business processes model the actual behaviour of the interaction parties, abstract business processes are meant to provide a partially specified description of the interaction, hiding some of the required concrete details.

Once the language used to specify orchestrations is stated, a service to process and execute those specifications should be chosen. That service, previously called execution engine, is bundled out of the box with most ESBs, allowing for a seamless installation by the final GEMBus administrators. Apache ODE (Orchestration Director Engine) is the choice as execution engine within FUSE, the base framework for the current GEMBus development effort.

Finally, a friendly environment to fill the gap between analysts and developers looks interesting. A language is required to model the processes graphically to allow analysts to express their knowledge and to transmit to developers what they want to do and how it should be done. That expressiveness is provided, once again following a standard language, thanks to BPMN [BPMN]. The resulting model in BPMN can later be transformed to an executable process in WS-BPEL by the developers. Once it is fully implemented and tested, the environment should also enable administrators to send the processes to production on the execution engine and to administer them. All that functionality will be unified in a single tool, shared by all the relevant stakeholders and built up from the extensively used, open source platform, Eclipse.

In addition to the orchestration service, interest in a workflow management system has risen. The concept of workflow, even having a lot in common with orchestration, have some differences that are not very clear, and authors cannot find a definition widely approved by the community. Avoiding the differentiation of both composition terminologies, a Workflow service will be presented. That service will consist of a set of tools oriented, but not limited to, the design and execution of scientific workflows and in-silico (via computer simulation) experimentation. That cross-platform set of tools will enable users to compose not only web services, but also local Java services (Beanshell scripts), local Java APIs, R scripts and import data from Excel or in CVS format.

3.4.1 Definitions and Technologies

Prior to going deeper into the details of the Composition services, following are some of the terms and technologies related to those services. It is not the intention to give a formal definition, but to explain how these terms are understood in the GEMBus architecture.

WS-BPEL: Web Services Business Process Execution Language (BPEL) is an XML-based executable and description language to define and implement business processes using web services. It is also capable of manipulating the messages and their content. BPEL tries to bring programming in the large to the world of web services. In addition, BPEL is used to implement business protocols.

BPMN: Business Process Modelling Notation (BPMN) is a graphical representation for Business Processes Modelling. BPMN is aimed to fill the gap between the different stakeholders that take place from the analysis of a business process to the implementation and management. It also provides a mapping to the underlying constructs of execution languages (BPEL).

Apache ODE: ODE is the execution engine for WS-BPEL bundled with FUSE, the current service framework in use within GEMBus. It is responsible for the web services communication stated in the processes, handling the data flowing between them and recovery from errors as described in the compensations.

Taverna: This is a Workflow Management System (WMS) that includes the Taverna Workbench (a desktop client application that is also able to run workflows), the Taverna Server (which allows for remote execution of workflows. the component inside GEMBus) and the Taverna Command Line Tool for terminal execution of workflows. Taverna is mainly oriented to scientific workflows and it has several services related to, but is not limited to, that community.

Orchestration: An orchestration is a recursive composition of services presenting the result as a new service. Orchestration depends on WSDL to represent the services interfaces and to provide the input and output points of the orchestration. Orchestrations are written in WS-BPEL and deployed by an execution engine. Orchestrations often cross domain/institution boundaries and are long-lived services, sometimes requiring hours or days to complete.

Workflow: A workflow is a single lane on an orchestration or a composition within the Taverna environment. In the Taverna domain, a workflow is also the execution unit (to be run on the GEMBus Taverna Server).

3.4.2 Service Details

The orchestration service will comprise the following:

- Execution Engine (Apache ODE)
- Design, implementation and management environment: an Eclipse-based tool.

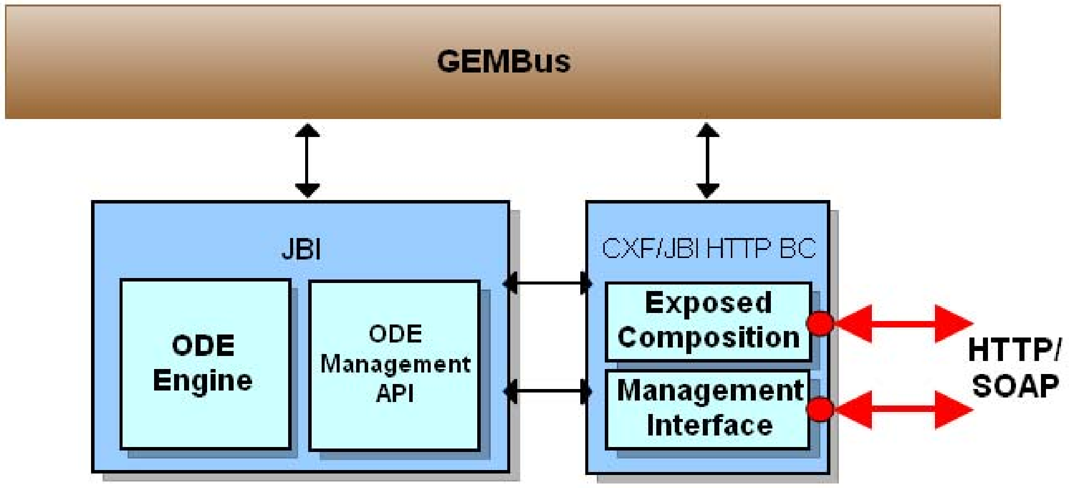


Figure 3.9: Orchestration service

There are two interfaces for management, `ProcessManagement` to manage deployed processes and `InstanceManagement` to manage the running instances of the deployed processes. The methods of those interfaces can be executed via the GEMBus Plug-in of the Eclipse tool. The initial version of the Eclipse tool is built up on the basis of Intalio Designer 6.0.3.050, which has some proprietary code added to the base Eclipse 3.4.1. This enables BPMN design and manipulation to transform it into a BPEL document.

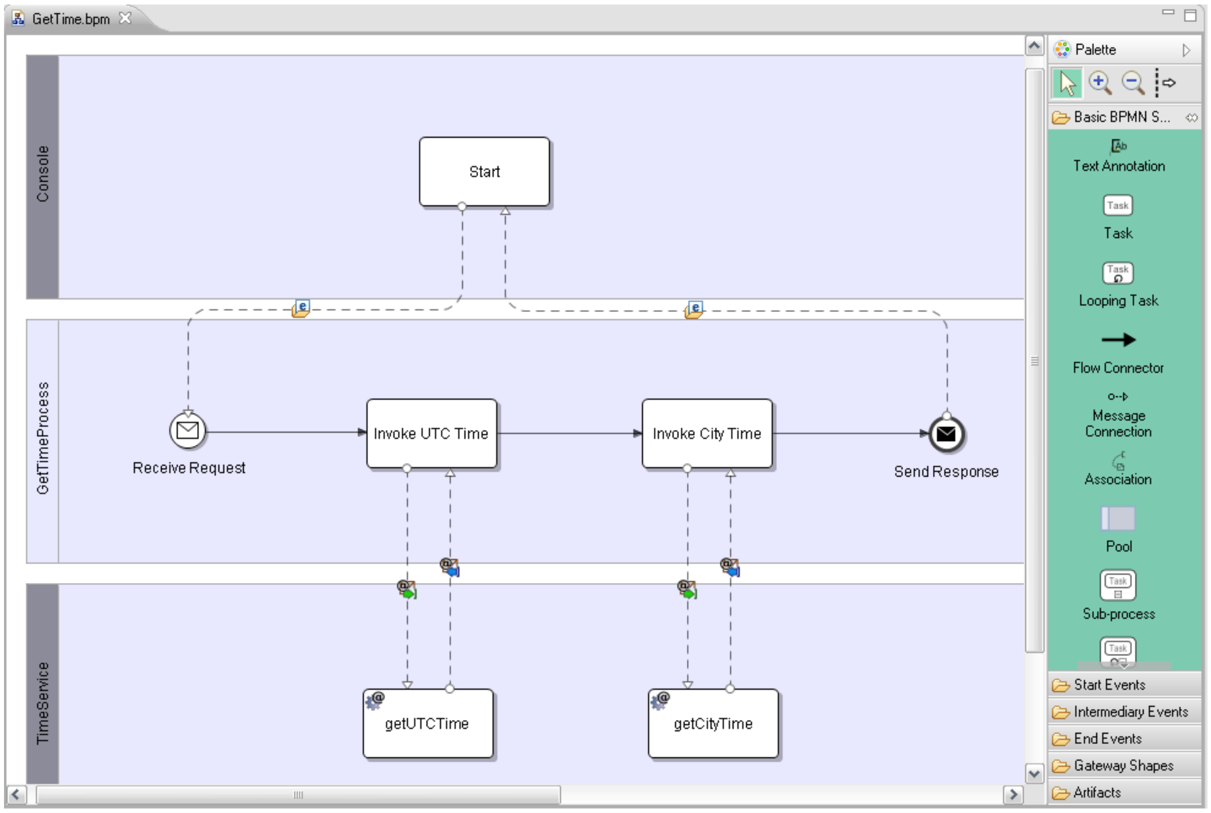


Figure 3.10: Orchestration design and management environment (Eclipse-based tool)

At the time of writing, the official OSGi version of Taverna Server has been rescheduled and is expected to be released in Q2 2011 (Alpha version for January). In the meantime, attempts to make the current release (v2.2.a1) comply with the OSGI standard have begun.

Taverna is in the process of re-engineering the core of the application to offer an OSGi-based distribution. This will enable developers to integrate the core in their applications or web portals. The intention is to deploy an execution server inside GEMBus. As usual, scientific simulation requires huge amounts of data to be processed and transferred, the incorporation of the execution server where the services live will enable stakeholders to use internal services and execute them on-site, resulting in a considerable bandwidth increase as the data now travels inside the GEMBus boundaries.

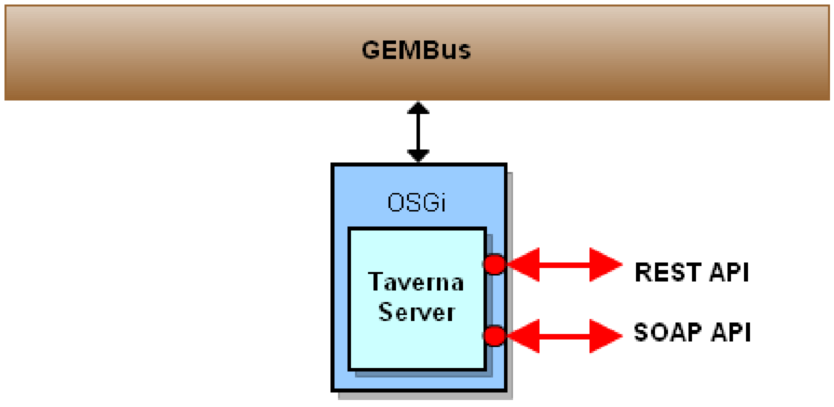


Figure 3.11: Workflow Execution Server within GEMBus with the WS APIs

Taverna also allows for local execution. This can be of interest for testing purposes and to simulate simple workflows that do not require large amounts of data and computation time. This local simulation is integrated into the desktop application, Taverna Workbench.

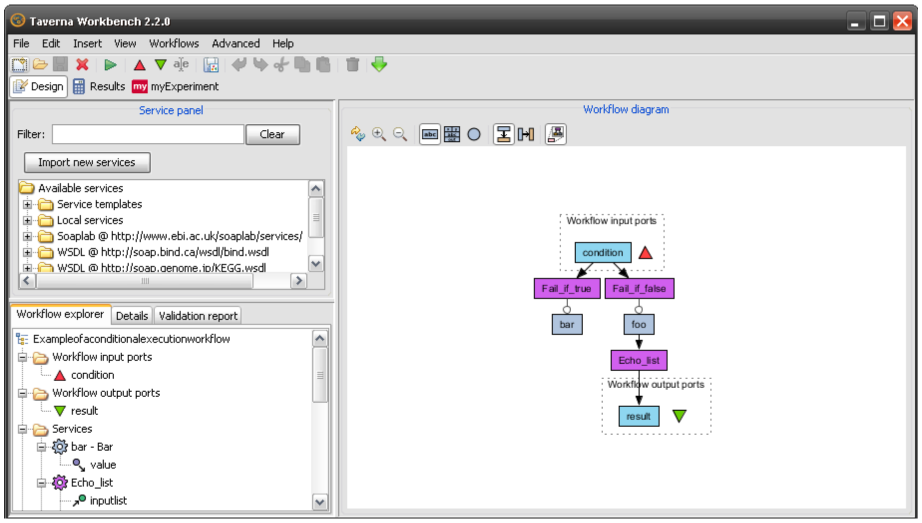


Figure 3.12: Taverna Workbench design and execution environment

3.5 Logging

Logs constitute an important source of information of service performance. They provide a great amount of data for monitoring and diagnostics purposes. When properly produced and processed, they can help to achieve efficient operations and to obtain a better understanding of service behaviour. As these logs comprise huge amounts of data they need special pre- and post-processing to actually generate useful information.

The rising paradigm of web services is creating a new model of interaction in the network service environment, which requires richer information than that captured in the traditional Web server logs. The source of information of Web server logs comes from links accessed by users. Such logs are not capable of capturing more complex interactions, such as service utilisation and/or composition, which are typical interactions supported by the web services technology. Moreover, the Web server log ends on the threshold of its own Web site. It does not cross the firewall's boundaries. It does not know about its partner's services, even if it is an important part of its services as the end user perceives it. Therefore, there is a need for new mechanisms for capturing, monitoring and logging the services usage that considers the specific needs of the processes supported by SOA.

The GEMBus multi-domain nature requires specific mechanisms for producing and processing meaningful log records. Figure 3.12 depicts the proposed architecture for the GLOS. The GLOS architecture consists of a Common Log Repository, where all logging data is stored and a GLOS service instance is deployed at every participating ESB.

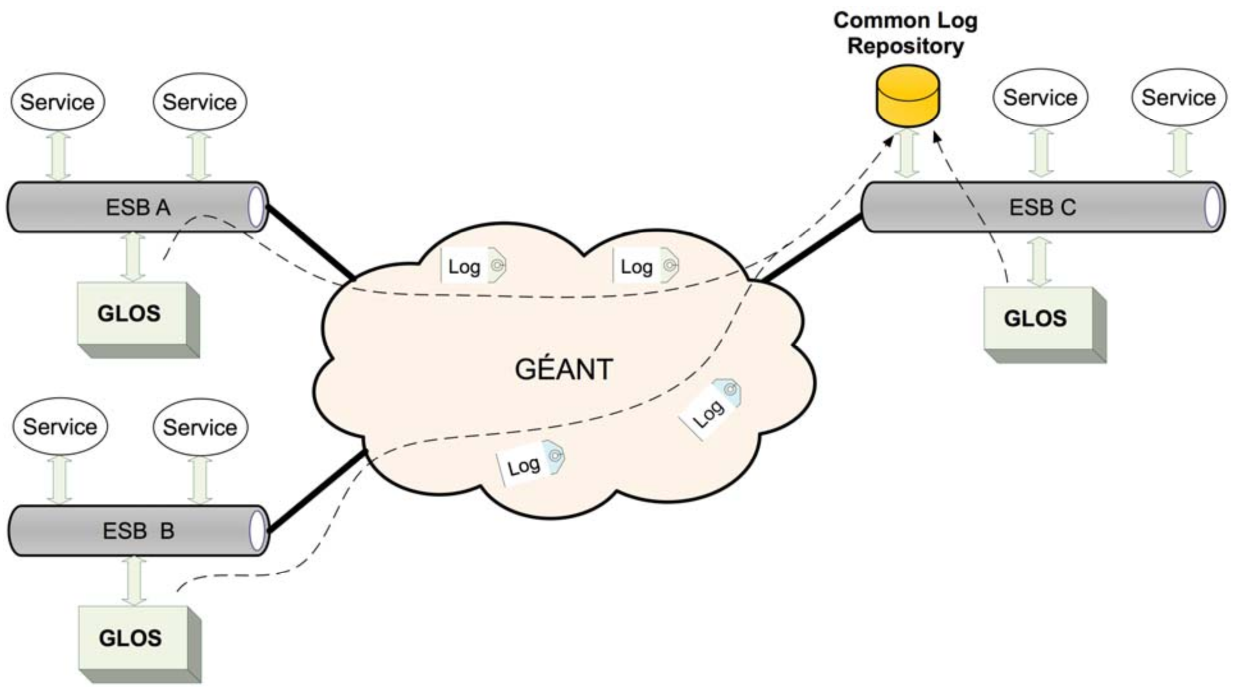


Figure 3.13: Common logging architecture

There are at least two ways to implement a GLOS. One of them is by changing the source code of each service willing to be integrated in GEMBus to call the incumbent logging service every time it is required. However, the main disadvantage to this solution is the lack of ownership over third parties' code. There is no guarantee that they will be willing to change it on another's behalf. Furthermore, modifying existing applications may be time consuming and error prone. An alternative is to use message interceptors (in most of the cases, SOAP messages). This is the approach that GEMBus takes. Figure 3.14 shows the functional modules of the GLOS service.

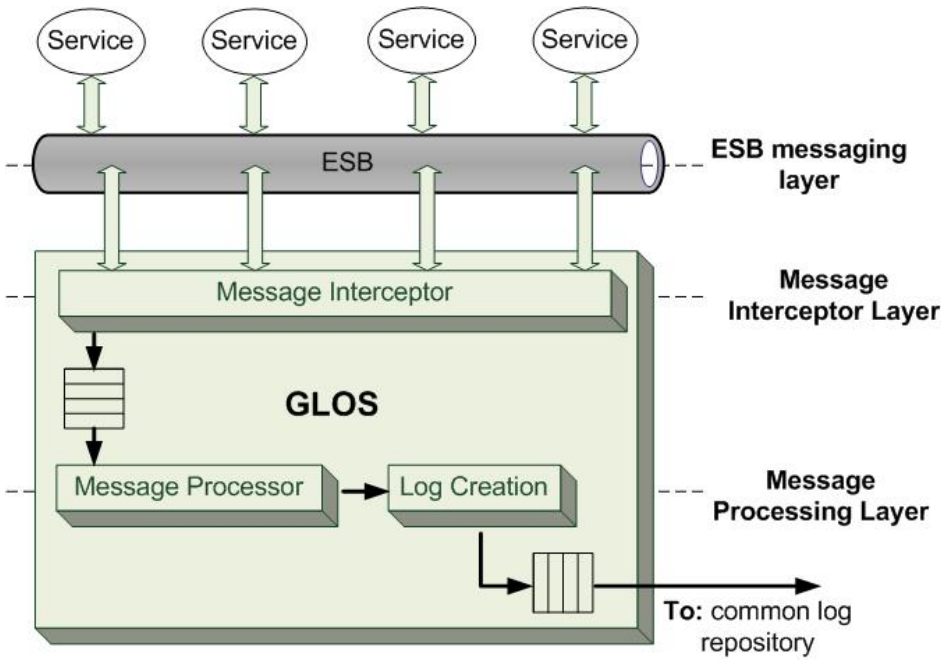


Figure 3.14: GLOS functional components

Services integrated in GEMBus use the message-oriented middleware (MOM) infrastructure offered by the GMI to send their messages. The function of GLOS is to catch and record every message exchanged through the GMI, as those messages are precisely the source of information to evaluate GEMBus services behaviour and performance.

The *Message Interceptor* module aims to catch every message exchanged through the GMI. GLOS will take advantage of the GMI message interceptors, extending them with specific capabilities for log data collection. The use of interceptors to implement GLOS prevents us from changing the service code, providing independence and flexibility on log management, allowing a log structure to be adapted according to eventual new needs without modifications on the existing services.

The *Message Processo*r module receives the intercepted message data from the *Message Interceptor* module and extracts the useful information to log using parsing mechanisms. The GLOS Message Processor needs to know the message envelope structure, as well as some agreed information transmitted in them, for logging purposes that must be acknowledged by services and the GMI.

The *Log Creation module* creates a log record for each intercepted message, saving the extracted information by the Message Processor and later forwarding it to the common log repository where all log data will be collected and where the logging analysis of the whole GEMBus infrastructure will be performed.

Consider SOAP interceptors as initial step, as SOAP-based services are currently the most common in web services scenarios. It is worth to explore which the structure is and the information carried in a SOAP message. A SOAP message is an ordinary XML document containing the following elements:

- *Envelope* element: identifies the XML document as a SOAP message.

- *Header* element: contains header information.
- *Body* element: contains call and response information.
- *Fault* element: contains errors and status information.

The SOAP header elements are optional and contain application-specific information. These optional SOAP headers allow extensions such as encryption, security, authentication, billing, correlations, etc. GLOS can extract useful logging information by analysing these header fields.

Figure 3.15 shows a fictional example of interacting web services, the *Infrastructure reservation* service. Users can request and reserve network connectivity (through a web portal) and computing resources at the same time. This example illustrates the message workflow for a user request. The user requests a network circuit between two Grid computing centres. The *Infrastructure reservation* service in turn uses the *BoD service* to provision network circuits and the *IT resource reservation* service to allocate computing resources from each Grid centre.

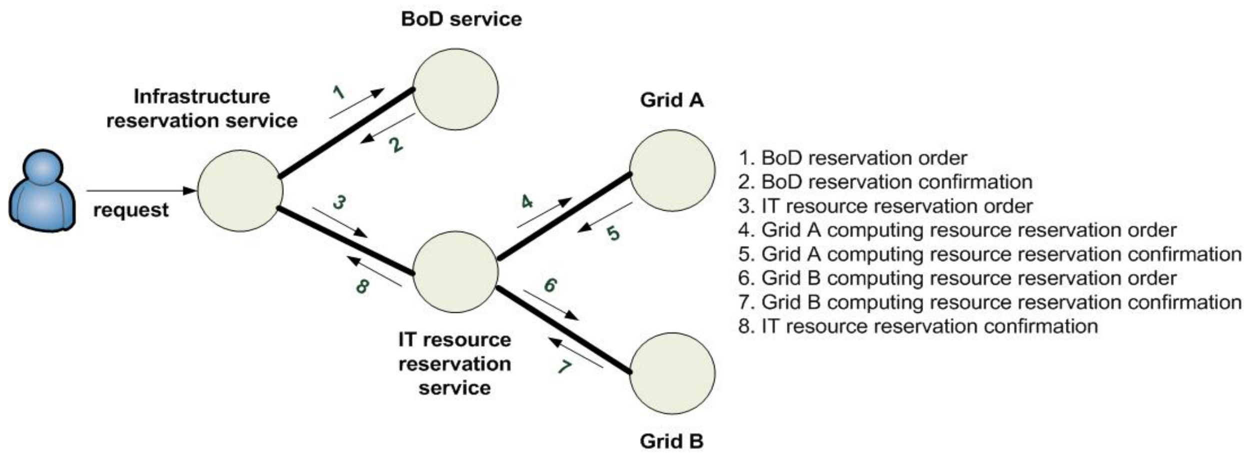

BoD service
Infrastructure reservation service
Grid A
request
IT resource reservation service
Grid B
1
2
3
4
5
6
7
8
1. BoD reservation order
2. BoD reservation confirmation
3. IT resource reservation order
4. Grid A computing resource reservation order
5. Grid A computing resource reservation confirmation
6. Grid B computing resource reservation order
7. Grid B computing resource reservation confirmation
8. IT resource reservation confirmation

Figure 3.15: SOAP messages exchanged among web services

As an example, the composite service described above wants to know how the component services are behaving. By studying and observing their behaviour, it would be able to optimise itself by either changing its component subservices or by instructing the existing component subservices to improve performance.

GLOS must fulfil the logging and monitoring functionality that the composite service needs to optimise its performance as a whole complex service, because it is actually seen as a single service by the end user. It is necessary to analyse end-to-end Quality of Service (QoS) and understanding bottlenecks or causes for failure of the service requests. Table 3.1 describes the types of message transactions between services to extract information.

Requirement Type	Requirement Description	Metrics Considered
Service Frequency of Use	Service use frequency. Periods of greater service use. Periods of greater data traffic.	Number of SOAP messages addressed to that service. SOAP messages size. SOAP messages date/time.
Service Availability	Time percentage in which the service is available.	Number of non-replied SOAP messages divided by the number of replied to SOAP messages.
Service Reliability	Percentage of well succeeded requests.	Number of failed SOAP requests divided by the number of well succeeded SOAP requests
Security	Percentage of secure services. Percentage of non secure services.	SOAP messages that carry security tokens divided by the number of SOAP messages without security messages.
Service Response Time	Time spend by the service to provide a reply to a given request.	SOAP message arrival time. SOAP message reply time.
Network Latency Time	Time spend by the message transiting over the network to reach the destination.	SOAP message sending time. SOAP message arrival time.

Table 3.1: Message transactions between services

In the *Infrastructure reservation service* example, the administrator has information to analyse some service performance parameters. For example, he will be able to extract statistics about service usage (frequency of use). Moreover, the administrator can determine whether or not the service has been available and reliable. If it has not, he can detect which sub-service is not performing well and take appropriate actions. Furthermore, the administrator can monitor if the service response times are within the normal values and, if not, take action such as hardware upgrade (the service may be overloaded and require a hardware upgrade). Finally, the system administrator can detect a poor service performance caused by an external factor such as high network latency.

To achieve this performance, GLOS takes advantage of the addressing and routing mechanisms provided by GMI through WS-Addressing message addressing properties and EPR. In particular, GLOS takes advantage of 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.

Using WS-Addressing it is possible to uniquely identify a message (*Unique message ID*), and identify who sent the message and its destination (*Source endpoint* and *Message destination*). It is possible to correlate message transactions and identify service workflows using the fields *Unique message ID* and *Relationship*. Correlating message transactions and identifying service workflows is a key capability for GLOS. Due to the amount of transactions going through GEMBus, it is required to identify to which service transaction each message belongs in order to do further log analysis. For each message, it is proposed to extract the following data and create a log record with the following fields:

```
[Timestamp] [Message ID] [Source EPR] [Destination EPR]
[Relationship to other messages] [Action] [Message size]
[Fault SOAP code (if any)] [SOAP body]
```

The log records will be saved using RDF ontology with the data fields described above. They will be sent to the common log repository to have a single point from which to compute logging analysis and control of GEMBus performance.

RDF is an emerging language and there are several reasons for selecting RDF ontology as a data model for GLOS:

- RDF is based on open source, languages and standards.
- A choice of a variety of serialisations and notations, including RDF/XML, N3, RDFa, Turtle, N-triples,
- Common framework and vocabulary for representing data structures and schema.
- Schema based on RDF can be extended and grown incrementally without impacting the existing data store.
- Because of conceptual closeness to the relational data model, it is possible to represent RDF in a relational database and vice versa. RDF has the ability to take advantage of historical RDBMs and SQL query optimisations.
- Use of a set-based semantics and queries. Via its SPARQL query language, easy mechanisms to drive faceted search and other browsing and viewing tools.

Note that RDF provides an easy way to extend the GLOS logging data model if necessary. Furthermore, it provides many advantages in the logging storage, where advanced queries provide a powerful tool to automate logging analysis.

3.5.1 Security and Privacy Considerations

The sort of records that GLOS will be able to collect, and that potentially could comprise the activity of all GEMBus-enabled services, can raise serious (personal and organisational) privacy concerns if a single centralised log repository is considered for the whole GEMBus infrastructure. It is important to highlight that the architecture does not require this unique repository, and that GLOS local deployments can decide to route logging messages to different repositories according to local policies and component and composite service requirements.

4 Service Lifecycle Model

Once the general GEMBus architecture has been discussed and the GEMBus core introduced, consider the lifecycle model that GEMBus will support for services deployed within it. Composable service lifecycle management is an important component of the Composable Service Architecture (CSA) proposed in section 2. It is key to the underlying design and operation of GEMBus. It is the basis for the CSA Service Delivery Framework (SDF), supported by the necessary Infrastructure Supporting Services (ISS) that provides consistent context management to the deployed services during their whole lifecycle.

Section 4.1 describes the CSA SDF. This is based on the TeleManagement Forum SDF [TMF-SDF] and extends it with additional stages that ensure consistent services context management during the whole lifecycle of the provisioned services. Figure 4.1 illustrates the main stages for service provisioning and delivery:

- **Service Request** stage, including SLA negotiation: The SLA can describe QoS and security requirements of the service along with information that facilitates authentication of service requests from users. This stage also includes the generation of a Global Reservation ID (GRI) that will serve as a provisioning session identifier and will bind all other stages and the related security context.

- **Composition/Reservation**: This stage comprises the Reservation Session Binding with GRI and provides support for complex reservation process in a potentially multi-domain environment. This stage may require access control and SLA/policy enforcement.

- **Deployment**: This stage begins after all component resources have been reserved and includes distribution of the common composed service context (including security context) and binding the reserved resources or services to the GRI as a common provisioning session ID. This stage can include an optional Registration and Synchronisation stage, specifically targeting possible scenarios in which the provisioned services are migrated or upgraded. In a simple case, the Registration stage binds the local resource or hosting platform run-time process ID to the GRI as a provisioning session ID.

- **Operation**: This stage is the main operational stage of the provisioned on demand composable services. Monitoring is an important functionality of this stage to ensure service availability and secure operation, including SLA enforcement.

- **Decommissioning**: This stage ensures that all sessions are terminated, data are cleaned up and session security context is recycled. It can also provide information to or initiate services usage accounting.

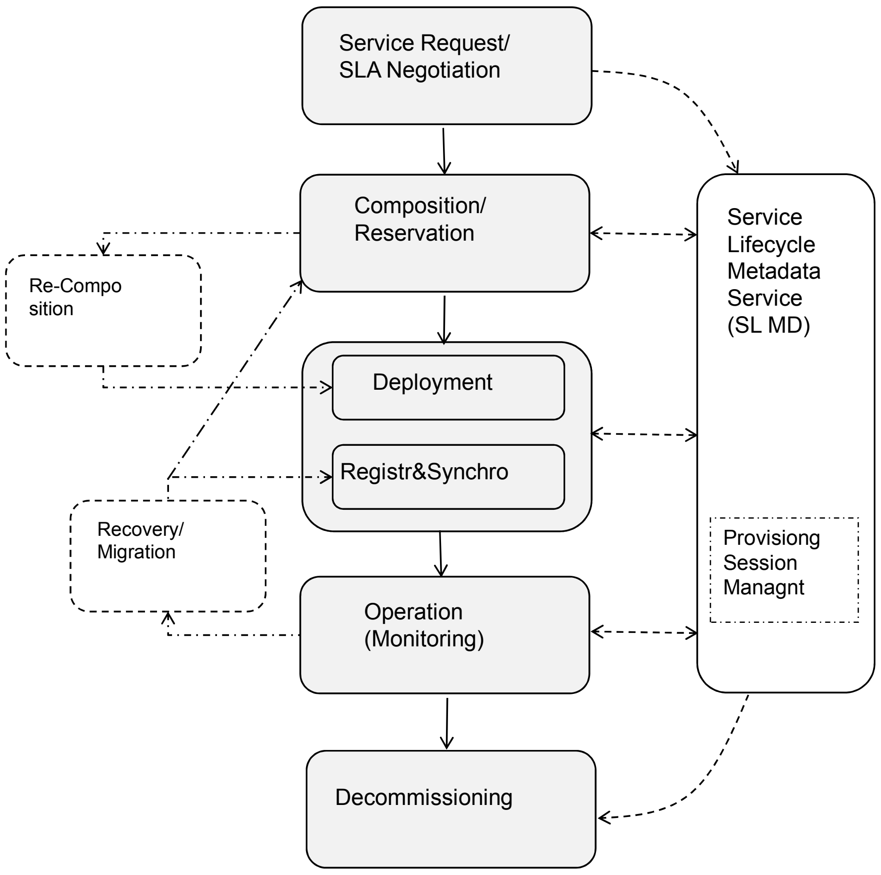


Figure 4.1: Workflow for on-demand provisioning of composable services

Two additional stages can be initiated from the Operation stage and/or based on the running composed service or component services state, such as their availability or failure:

- **Re-composition** or **Upgrade**: This stage allows incremental infrastructure changes.
- **Recovery/Migration**: This stage can be initiated by the consumer and the provider. This process can use service lifecycle metadata to initiate a full or partial resource re-synchronisation. It may also require re-composition.

4.1 Infrastructure Services to Support CSA SDF

The implementation of the proposed SDF requires a number of special Infrastructure Support Services (ISS) able to consistently support lifecycle management as part of the CSA middleware. The most basic one is the Registry mechanism already described as part of the GEMBus core, providing service registration and discovery. Beyond this, two additional services are required to support the lifecycle model:

- **Service Lifecycle Metadata Repository**: This keeps the service metadata during the whole service lifecycle, including service properties, service configuration information and service state.
- **Service and Resource Monitor**: This provides information about services and resources state and usage.

5 Integration Patterns

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 goal is to:

- Ease gaining access to those services by other user communities worldwide.
- Allow other services to leverage on and/or be composed with them.
- Provide homogenous mechanisms to make these services evolve.
- Simplify the integration with similar or related platforms in other spheres (commercial, governmental, etc.).

Following the experiments the GEMBus team executed to define the architecture introduced in this document, three different patterns for service integration have emerged. This section presents the main principles on which these integration patterns are based. It analyses their application to certain flagship GÉANT services and provides recommendations for future service integrations.

The service integration process in many cases can be complicated, time-consuming and cause other problems due to the requirement to integrate many types of services. Some of them provide interfaces based on standards but many others provide rather specific interfaces. Moreover many service frameworks that do not provide any service-oriented interfaces are developed in various programming languages (e.g. C++, Java, Python, Ruby, etc.). The ESB framework simplifies the integration process providing many standard-based modules ready to be used for binding existing services to the bus. Integration of the other interface types can be implemented as adaptors on which the GEMBus team is working. At the time of writing, the GEMBus framework considers three types of adaptors:

- Single Adaptor: One adaptor of many integrated services.
- Per-service Adaptors: Each service is integrated by separate adaptor.
- Publish/Subscribe: Services publish data. They are classified and can be received by subscribers.

5.1 Single Adaptor: AutoBAHN Case

The single adaptor pattern is the case where one adaptor is used for many integrated services. This pattern supports the interaction between a service consumer on a local ESB and a service provider on a foreign ESB. The logic to translate the ESB behaviours (for example security, assured delivery, logging, etc.) to support that interaction is built into the single adaptor, then bound to the adaptor at design time.

AutoBAHN integration follows the single adaptor pattern. The AutoBAHN (Automated Bandwidth Allocation across Heterogeneous Networks) [AUTOBAHN] is an inter-domain network reservation system capable of setting up on-demand dedicated circuits across heterogeneous domains irrespective of the underlying technologies. It is developed in GÉANT2 and provides communication through web services and/or a graphic user interface (UI).

The current prototype implementation of the AutoBAHN integration follows the single adaptor integration pattern. This pattern is used because the AutoBAHN service provides a single interface towards the user and does not require any extra mechanisms or mappings for the methods of the interface to be called. The integration is achieved via the GEMBus platform through the creation of an adaptor that enables the communication of the AutoBAHN network reservation system with the user, on one hand and with other composable services, on the other, e.g. monitoring services, bandwidth allocation and network topology services.

Figure 5.1 illustrates the AutoBAHN Client adaptor.

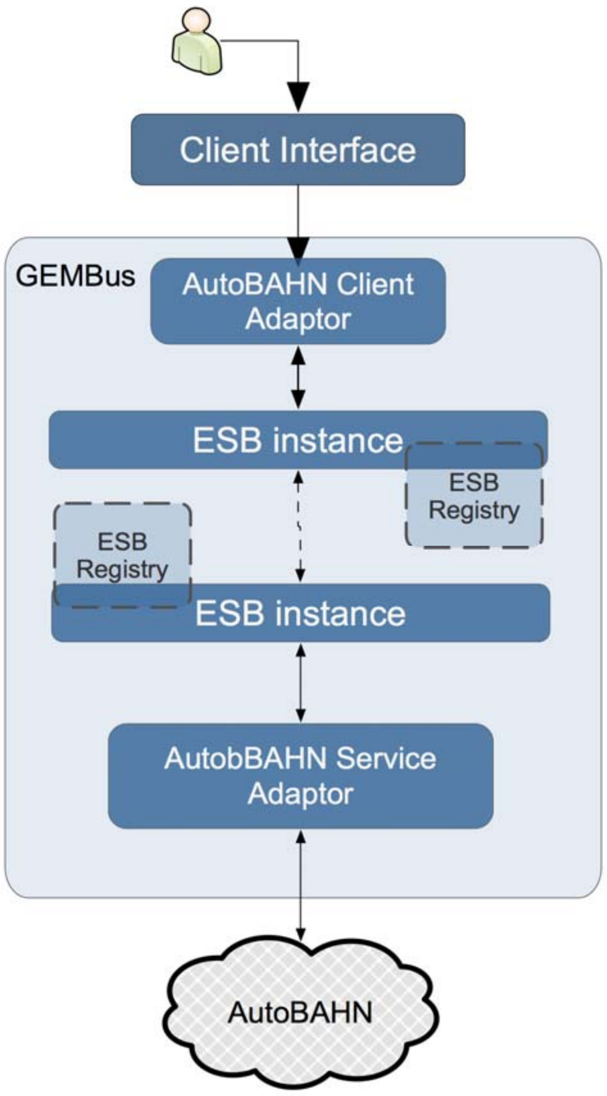


Figure 5.1: Integration of network reservation systems with other composable services

The architecture schema, with the modules required to provide the AutoBAHN integration into GEMBus, is described as follows:

Client Interface: This is the front-end to the client that receives the user request parameters, handles the reservation request and/or cancellation, and/or modification, and forwards the request to the AutoBAHN Client Adaptor.

AutoBAHN Client Adaptor: This adaptor adopts the Single Adaptor pattern. It communicates with the Client Interface and handles the user request for the path reservation/cancellation/modification. It receives the network reservation parameters, such as source IP, destination IP, reservation time period and network guarantees (such as bandwidth, delay, etc.) and forwards them to GEMBus, which will then provide them to the AutoBAHN Service Adaptor.

AutoBAHN Service Adaptor: This adaptor is responsible for deploying the AutoBAHN service inside the GEMBus platform and publishing it as a composable service available for others to see and call it. This adaptor also acquires the reservation parameters from the GEMBus and calls the AutoBAHN reservation system with the specific user request.

Through the AutoBAHN Service Adaptor the AutoBAHN reservation service becomes a composable service inside an ESB registry. It can be orchestrated or composed with other services, such as with a monitoring service in which it can forward the reservation parameters to monitor the network path.

The Single Adaptor pattern used in the GEMBus case provides a translation of the AutoBAHN service interface into a compatible user interface and, by transforming user data into appropriate forms, translates the user calls into calls to the original interface. This pattern is efficient and can be effectively followed in cases where future services have a single interface and their integration into GEMBus can be easily achieved by mapping this interface into a different one, user-friendly and with a similar functionality.

5.2 Per-Service Adaptors: The perfSONAR Case

This section describes the concept of the integration pattern based on per-service adaptors, as well as its advantages and disadvantages. It proposes a method to classify the per-service integration pattern.

The concept behind this integration pattern assumes that each service is implemented as a separate adaptor. This solution provides several advantages, such as easy management of services. Since the ESB framework manages the lifecycle of the adaptors the process related to deployment as well as stopping and restarting each of the services will be easy to perform. Having each of the services in separate adaptors also provides easy access restriction, keeping statistics of the service usage or logging all service events. This pattern also offers an easy way for clients to access the services. The use of ESB binding elements makes it possible to put out many types of ready-to-use endpoints (such as SOAP or REST protocols) for clients, as well as easy reconfiguring.

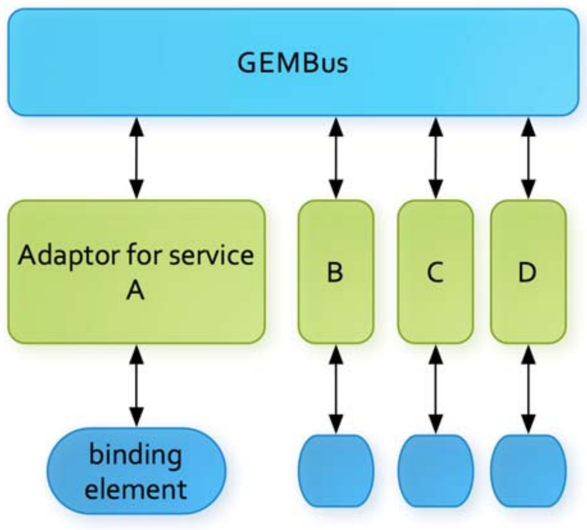


Figure 5.2: Concept of the per-service adaptors integration pattern

5.2.1 Integration Methods

During the experiments with this GEMBus integration pattern, three different integration methods have been identified, depending on the type of adaptors that can be implemented:

- Adaptors using the current provided services.
- Adaptors based on the service source code.
- Adaptors based on newly developed services.

The first type of integration method uses the current provided services. It is based on an evolutionary, rather than a revolutionary, approach. It is important to note that (in this method) the communication with the current running services cannot be implemented in all cases on standard protocols. In this case, an additional adaptor is necessary (known here as an interceptor) which translates between the current running service and the ESB internal protocol. Figure 5.3 shows the structure of adaptors based on this method.

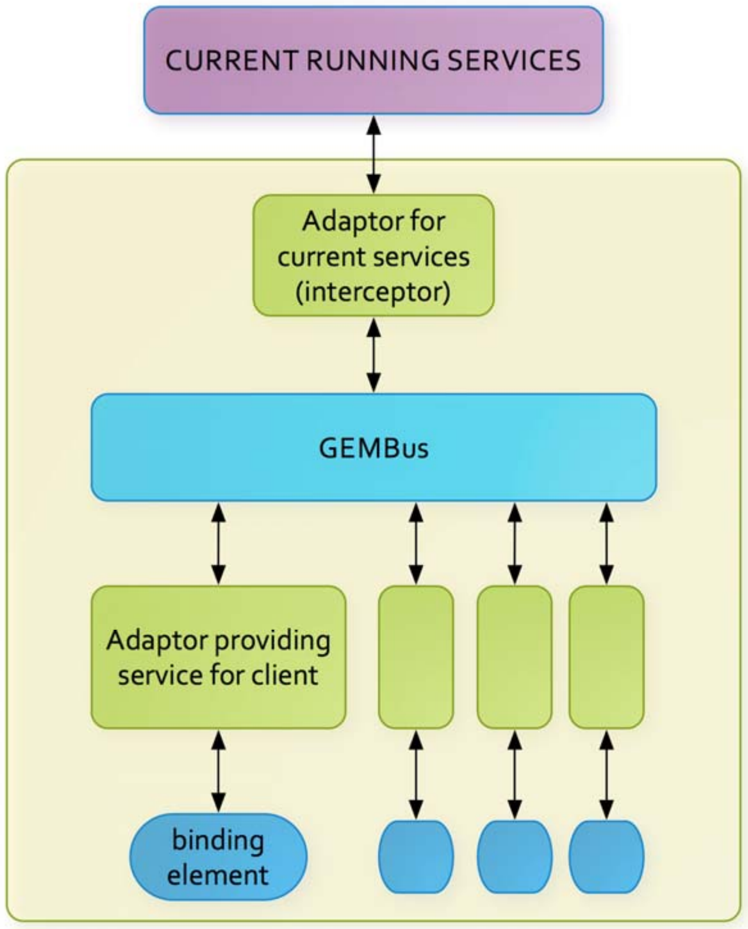

CURRENT RUNNING SERVICES
Adaptor for current services (interceptor)
GEMBus
Adaptor providing service for client
binding element

Figure 5.3: Per-service adaptors based on current running services

The second type of integration method uses the source code of the service to provide it through the ESB. In this case, there must be some requirements defined for the adaptor to be able to register with the GEMBus and provide services for clients. Figure 5.4 shows the adaptors based on this method.

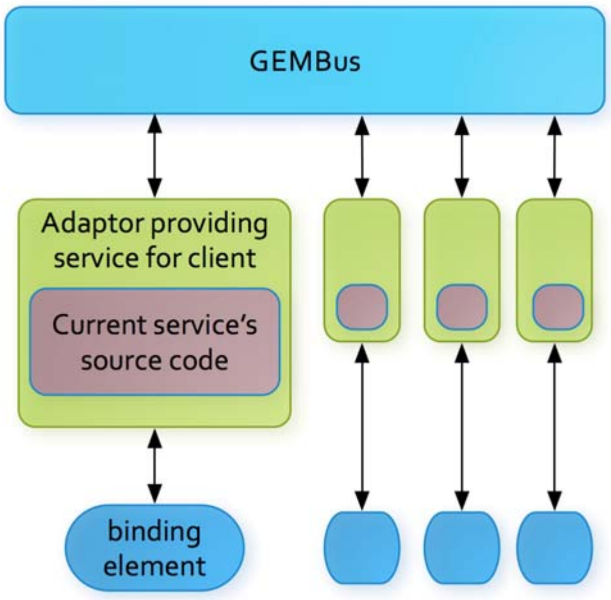

GEMBus
Adaptor providing service for client
Current service's source code
binding element

Figure 5.4: Per-service adaptors based on current service source code

The third integration method requires developing a new service from scratch. This method can produce the most optimised adaptors and service integration with GEMBus. In some cases, this method may be better than the second method. Figure 5.5 shows the adaptors based on this method.

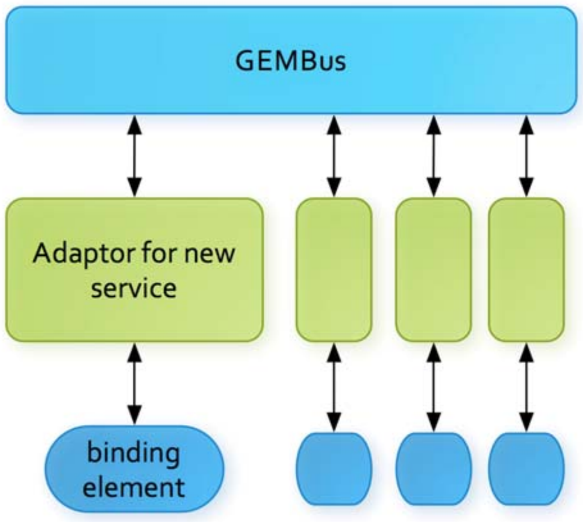


Figure 5.5: Per-service based on newly developed services

5.2.2 Prototype

The current prototype used to experiment with the integration of the GÉANT perfSONAR services (Performance Service Oriented Network monitoring architecture) [PERFSONAR]. The integration of perfSONAR is based on the per-service integration pattern because of the perfSONAR software architecture, implementing it by means of adaptors using the current provided services. At the time of writing, the services already integrated with GEMBus are the perfSONAR Command Line Measurement Point (CLMP) and RRD Measurement Archive (RRDMA) services. Figure 5.6 shows conception (left) and prototype (right) of the integration GÉANT perfSONAR services within GEMBus.

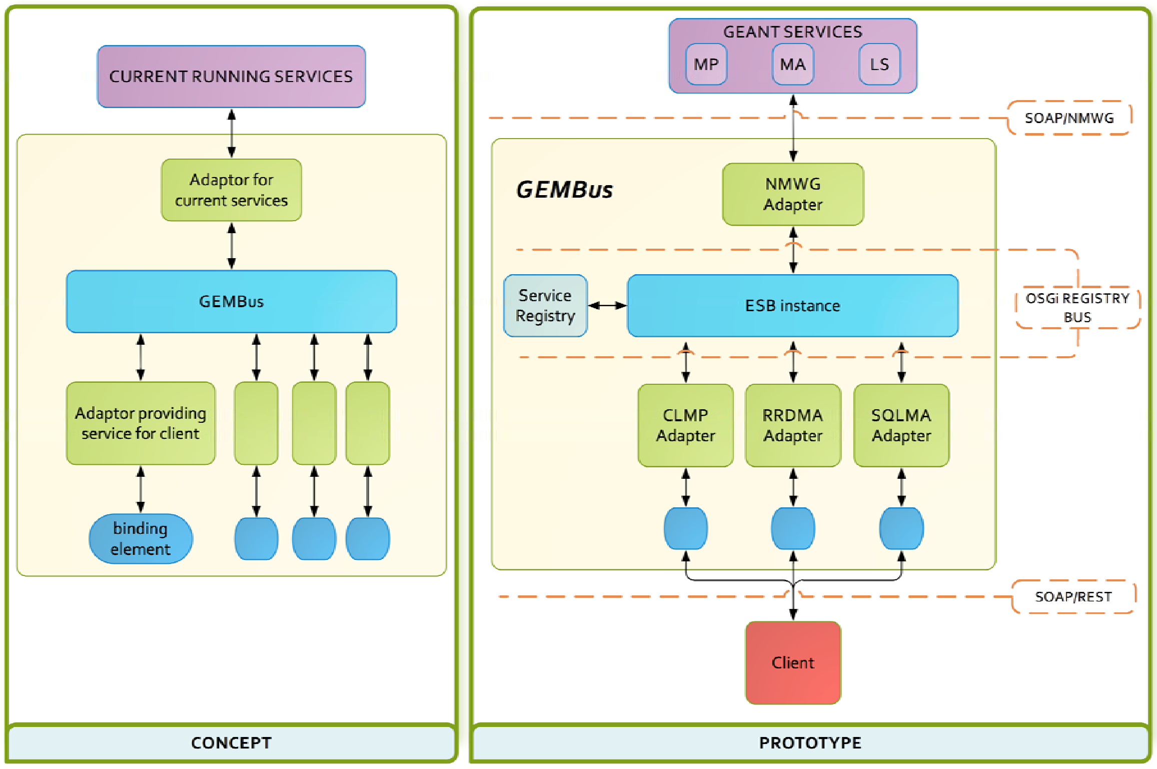


Figure 5.6: perfSONAR integration in GEMBus using a per-service adaptor pattern

5.3 Publish/Subscribe: The eduroam Case

eduroam® [EDUROAM] is a well-known and relatively mature service run within GN3 SA3 T2. Network access provided by eduroam is based on RADIUS authentication passing through the eduroam infrastructure. In the eduroam model, authentication is equivalent to authorisation to use the service. In GN2, it was planned to investigate how the bare authentication process could be extended with the use of additional SAML channels (the DAMe project [DAME]). This is currently being pursued within the GEMBus scenario, exploring the additional ideas discussed below. Splitting the process in well-defined blocks guarantees reusability of the code and concepts.

eduroam places much emphasis on users' privacy by allowing users to hide their true electronic identity within the visited network. Work done in GN3 JRA3 T1 introduces a permanent opaque user identity that can identify the user within a given visited site. The main use case of this extension is to enable a speedy reaction in cases of network incidents. This approach also opens a new possibility to add functionality for certain eduroam visitors, without a need to create local access accounts for them. The identifier introduced in eduroam is carried as the Chargeable-User-Identity (CUI) RADIUS attribute. To generate different identifiers for different eduroam service providers, it is expected that these service providers will send a domain that identifies them in the Operator-Name RADIUS attribute.

The main strength of eduroam is its simplicity, which leads to quality of the service and its wide adoption. When designing any extensions of eduroam, care must be taken not to introduce any negative effects. New interfaces between eduroam and GEMBus should be lightweight and non-blocking. Great care must be exercised to preserve the privacy profile of eduroam.

Native eduroam-generated events are limited to successful or unsuccessful authentication and can be published by home institution, visited institution or a relaying party (eduroam proxy). Additional messages, generated to GEMBus by supporting systems, could include authorisation attributes. By adding a new information exchange channel one can also add an equivalent to a forced log-off message, something that the RADIUS protocol does not provide. All messages should contain the value of the CUI.

Potential subscribers to eduroam login events can be:

- Local IdP services used for storing local statistics and for keeping information about current users;
- Global statistics collectors and analyzers.

eduroam service providers can consume authorisation messages to offer additional services.

Service providers can use log-off messages to immediately terminate the user's association to the network.

eduroam involves a number of server implementations. The simplest method with which to interface them with an ESB is to provide a syslog listener. This collects information sent by the servers and republishes it to GEMBus.

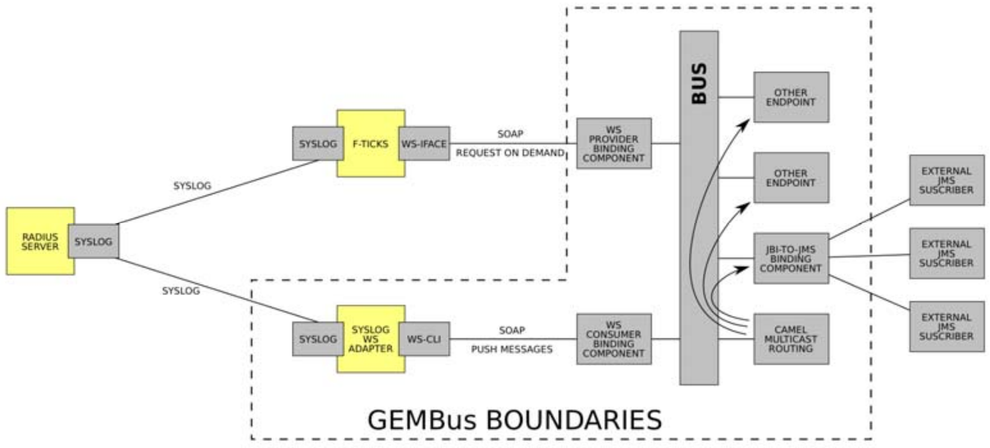


Figure 5.7: eduroam syslog in a publish-subscribe pattern

5.3.1 Case Study

5.3.1.1 *Case 1: Enabling a printing service for certain visiting users*

When an academic guest visits a university for an extended period of time, it may be desirable to provide him/her with some additional local services, such as access to local printers. There may be a number of methods to do this locally

- Assigning the user to a privileged VLAN.
- Opening access on a local firewall.
- Sending a message to a local ESB based printing service.

Regardless of the method selected, access control must recognise and authorise the user. The printing service is used here simply as an example. Many other local services can be enabled in a similar way.

The general idea in this scenario is based on the fact that a visited institution is able to correlate the user's machine hardware address, the user's CUI and the user's machine IP address attribute value. Correlation between the first two happens during authentication and can be obtained from the RADIUS server, while the IP assignment process correlates the latter two. A well-managed network should block any attempt of an unauthorised IP change. On the other hand, a MAC address change will result in re-authentication that will renew the MAC-CUI and IP-MAC associations. Access rights to the service attached to a given CUI value may be derived automatically from the user's IP or MAC address.

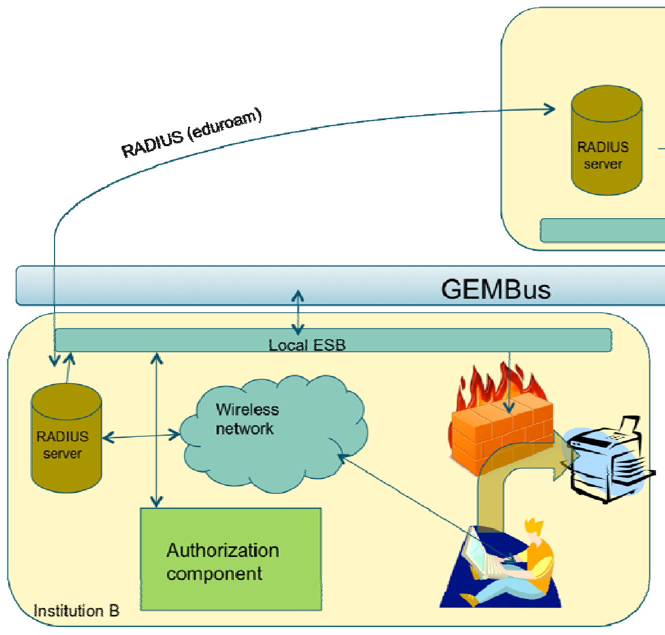


Figure 5.8: Enabling access to a local resource relying on eduroam

Two scenarios are proposed for how the authorisation may be done.

1. **Authorisation decision passed from the home institution of the user**: Upon authentication, the user's IdP may look up an authorisation table and send a GEMBus message containing additional authorisation attributes. The addressee of the message is looked up in the GEMBus registry, based on the value of the Operator-Name attribute contained in the Access-Request RADIUS message. This solution requires that some additional trust relationship be established between the visited institution and user's home institution. It also requires that both sides have access to GEMBus.
2. **Authorisation decision made within the visited network**: As has been observed earlier, once a given CUI has been assigned some additional rights, the rest of the access mechanism can happen automatically. Since the actual CUI value is not known to the user, the mechanism of coupling the user and CUI is crucial. The following proof-of-concept solution is proposed:

- The visiting user, after having authenticated to eduroam, accesses a local registration web page and asks for permission to use the service, providing an e-mail address.
- The web server sends an email to the user's address with either a password or a one-time URL.
- The user accesses the provided confirmation page.
- The WEB server obtains the user's IP, correlates it to the CUI and saves the CUI-email pair in a local database.
- Additional domain-specific verification confirms that the user with a given email address is indeed authorised to use the printing service and the result is recorded in the local authorisation system.

The actual access to the service can be realised in various ways, but a proof-of-concept implementation is under construction using the ESB approach described above. Tools created in the process will be applied to other GEMBus interactions in the future.

5.3.1.2 *Case 2: Identifying eduroam abuse by the same user logging at several locations in the same time*

This case has been studied in detail and implemented as a proof-of-concept in the experiment described in detail in Appendix A. The requirement is that in cases when it is observed that the same user appears at the same time in a number of locations, the user should be disconnected from eduroam either in all locations or all except the most recent ones. Appendix A shows how this result can be achieved by sending GEMbus messages between eduroam parties.

5.3.1.3 *Case 3: Enabling eduroam statistics*

GN3 JRA3 T1 and GN3 SA3 T2 have provided a new statistic tool: F-Ticks [FTICKS]. The concept is based on sending syslog messages from authenticating (or intermediate proxy) server to a central collection point, where they get anonymised and merged. GEMBus can serve as an enabler for additional analysis tools if F-Ticks messages are published and can be subscribed. There are various levels of data anonymisation based on trust relationship between the publisher and a subscriber. An initial service will be built by setting up a syslog publisher and listening to messages forwarded from the F-Ticks service.

6 Accessing GEMBus

GEMBus is conceived as an infrastructural service. It intends to be able to incorporate services and allow building more complex ones by composition, in addition to providing any component or application deployed in the GÉANT ecosystem with access to the services it integrates, without necessarily requesting it to be accessible through the GEMBus federated SOA.

This section describes common approaches suitable to be employed by applications (or any service element) willing to take advantage of GEMBus services. These were a consequence of experiments carried out by the GEMBus team to define and validate the architecture, as well as some results (most notably, the AC demonstrator; see section C.1).

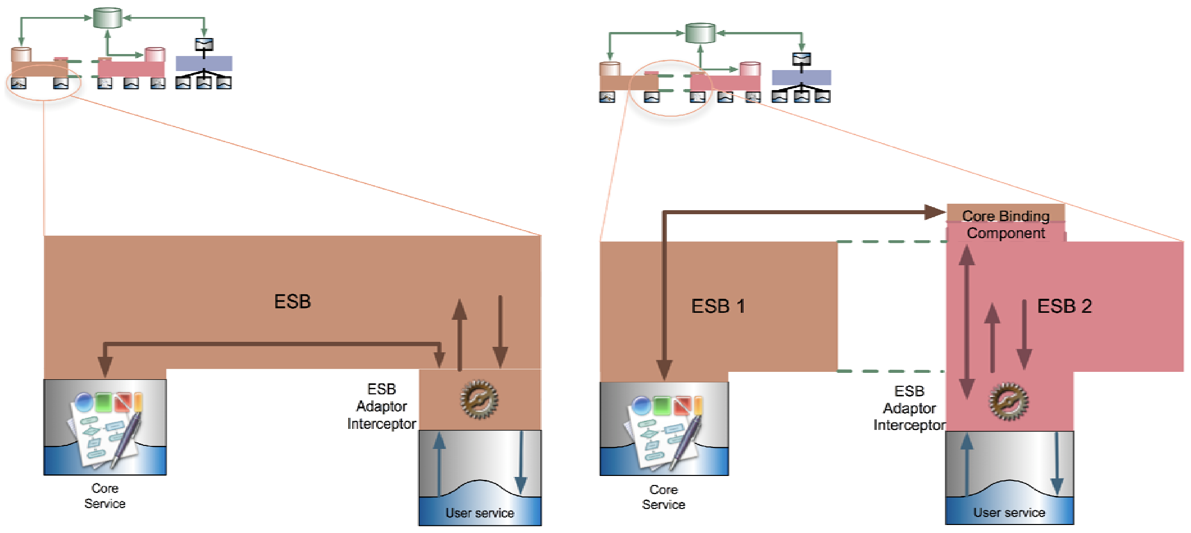


Figure 6.1: Accessing GEMBus through direct ESB integration

Figure 6.1 illustrates accessing GEMBus through ESB integration. The first mechanism is the most direct one and corresponds to a fully integrated element in an ESB instance participating in the GEMBus federated infrastructure. The application component is able to access transparently any GEMBus service located in the same ESB instance or in a separate one. The GEMBus core adaptors and binding components provide the necessary mechanisms to make this interaction seamless, so developers can concentrate in the element business logic and leave particular aspects related to service access (security, routing, data adaptation, etc.) to deployment options established according to the particular element usage and general ESB configuration defaults provided by the infrastructure administrators.

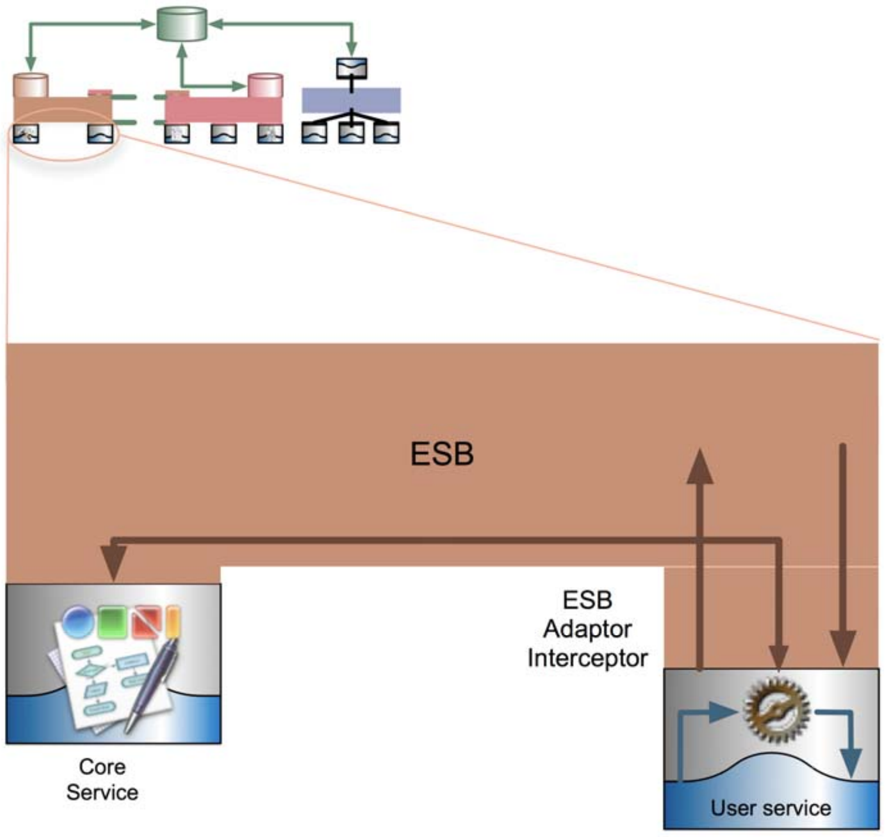


Figure 6.2: Accessing GEMBus at the API level within an ESB

There may be many other cases where a finer control on the usage of GEMBus core services is required or in which the developers want to specifically apply only a part of those services in their element. Since GEMBus core services are integrated with the ESB supporting platform, they can be accessed by directly invoking its API. They can be viewed from the user element as part of the web services framework on which the ESB is built (for example, Apache Spring in the current FUSE ESB framework).

In fact, any mix of this approach is possible. An element could rely on the participating ESB for message routing (as an example) and use the GEMBus security service by direct API calls. Note that this flexibility comes at the price of a higher dependency on the particular implementation details and required libraries used for a given GEMBus core service, so the evolution of the element is not only driven by the business logic requirements.

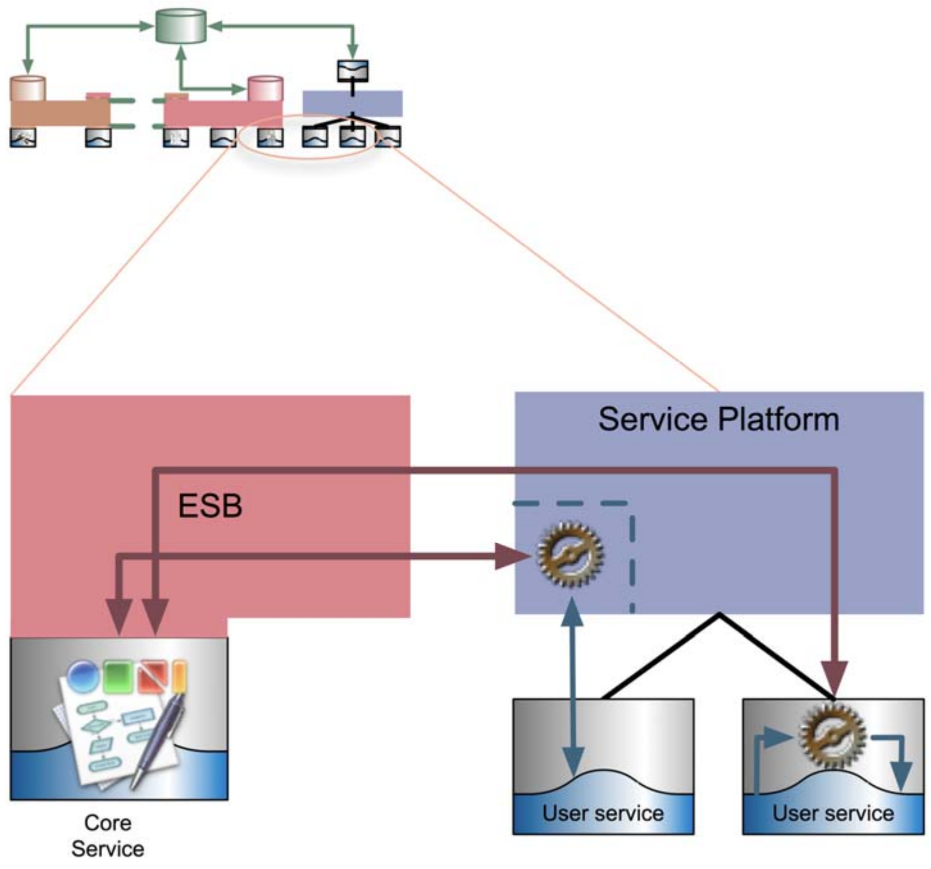


Figure 6.3: Accessing GEMBus from outside an ESB

GEMBus services can be accessed by other elements not directly integrated in any participating ESB, as long as the service platform on which the user element runs incorporates the necessary API to access GEMBus services (or alternatively they are incorporated into the service bundle), and the GEMBus Registry is used to look up the necessary data for locating and accessing the requested services. The first condition is similar to that which was described for the previous approach, while the second requires the implementation of at least one of the access methods provided the Registry, either to make real-time queries or periodical updates.

This latter approach covers several interesting use cases, allowing the application of core GEMBus services in environments such as access portals. GEMBus acts as a bridge between base GÉANT services and the mechanisms providing access to them to the general user community.

7 GEMBus in Relation to Other Architectures

This section describes the overall CSA, which was introduced in section 2, together with its practical realisation through GEMBus in the framework of related standards (particularly those related to the Open Service Environment (OSE) concepts, as defined by ITU-T Next Generation Network (NGN) [NGN], the IPSphere Framework [IPSPHERE] and OGSA [OGSA]. Finally, the alignment of the proposed GEMBus architecture to parallel efforts to formalise business process in GÉANT will be analysed.

7.1 GEMBus in the Context of Standards

The CSA can be described as a framework for implementing the Open Service Environment (OSE) concept, as defined by ITU-T Next Generation Network (NGN), which demonstrates the present trend to use service-oriented concepts in the modern telecommunication industry.

GEMBus relies on a CSA that adopts the layering approach defined in the Web Services Architecture and extends it with additional lower and upper layers to reflect use cases specific to Internet and telecommunications services. Figure 7.1 shows the CSA layers. They are described as follows:

1. **Networking Layer**. This layer provides the capability to apply technologies typical in distributed enterprise applications, such as VPNs.
2. **Transport Layer**. This layer defines functionality specific for service communication such as transport layer security (using TLS/SSL protocols), assigning service types to specific ports, etc.
3. **Messaging Layer**. This layer defines functionality related to message handling such as message routing, message format transformation, etc.
4. **Virtualisation Layer**. This layer is split into a Logical Abstraction Layer and a Composition Layer. This layer provides functionality to compose services and support/drive their interaction (e.g. with workflows) to ensure application interactions.
5. **Application Layer**. This layer hosts application-related protocols and represents applications, where the major goal is application related data handling.

Security services are applied at multiple layers to ensure consistent security. Management functions are also present at all layers and can be seen as the management plane, similar to the NGN reference model.

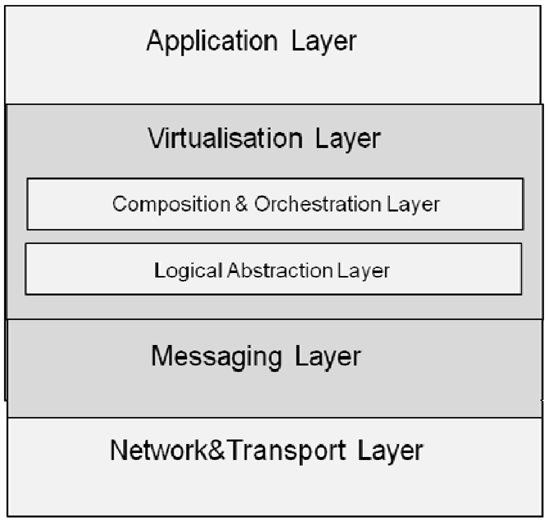


Figure 7.1: CSA Layering

A number of the recent ITU-T recommendations related to the NGN provide a basis for transport/network and Information Technology (IT) convergence based on NGN. The NGN concept is introduced by ITU-T as a next step in creating a global information infrastructure. The NGN reference model, according to ITU-T Y.2011 Recommendation, suggests the separation of the transport network and application services and defines them as NGN service stratum and NGN transport stratum consisting of a User plane, a Control plane and a Management plane. Any modern networking environment is characterised by the integration between services and network infrastructure, increasing use of Internet protocols for inter-service communication, services digitising and integration with higher-level applications. The ITU-T Recommendations Y.2012 and Y.2201 specify high-level requirements and functional architecture of the NGN Release 1. The described NGN service architecture proposes a service and network separation principle and defines functional components of the Transport stratum and Service stratum. The NGN Y.2012 architecture also defines the Application Network Interface (ANI) that provides an abstraction of the network capabilities and is used as a channel for applications to access network services and resources.

The NGN convergence service model is defined by ITU-T Recommendation Y.2232 and suggests the major scenario by means of web services. The NGN OSE defined by ITU-T Recommendation Y.2234 is based on web services and actually implements basic SOA principles in defining a service integration model. The definition of the OSE and web services convergence model is targeted to provide a common framework for both application and provider service developers.

The Y.2234/Y.2201 NGN OSE is required to satisfy such requirements as independence from transport network providers, independence from manufacturers, location transparency, network transparency and protocol transparency. The OSE should provide the following capabilities to support effective services integration and operation:

- Service coordination.
- Interworking with service creation environment.
- Service discovery.
- Service registration.

- Policy enforcement.
- Development support.

Development support suggests that OSE should support the full lifecycle of components, ranging from installation, configuration, administration, publishing, versioning, maintenance to removal.

In a natural step, another set of ITU-T standards prescribe NGN convergence model based on web services and require NGN capabilities to support OSE. Web service-enabled NGN transport networks provide a native environment for integrating applications, services and resources that can be provisioned on demand.

The IPSphere Framework is a framework for abstracting and composing multi-stakeholder telecommunications-based services both within and between service providers. Service abstraction describes the business and technical characteristics of a service and its constituent service elements, which are sets of resources offered by different providers. Service composition identifies and selects elements that satisfy these technical and business requirements. The IPSphere Service Structuring Stratum provides support for structuring, executing and assuring these services.

The IPSphere Framework can be viewed as a set of mechanisms needed for any user to obtain a final service composed of several elements belonging to distinct infrastructure/network providers. The interface in the highest layer offered by the IPSphere Framework is used for provisioning complex services by means of orchestrating/selecting simple elements.

GEMBus infrastructure can be mapped easily with the IPSphere Framework. As GEMBus is based on SOA, its services are characterised by having well-defined service endpoints, which allows loose coupling between them. Furthermore, GEMBus relies on its core services for registering, orchestrating and composing complex services according researchers needs.

It must be taken into account that IPSphere is a framework under strong evolution and enhancement, and because of this, no reference API has been implemented. Currently, TMForum is working on the second specification release.

Based on the SOA, the Grid SOA was developed because the Grid Architects decided that there were enough requirements in their services-oriented architecture that were not met by the Service Oriented standards. The Open Grid Forum (OGF) was formed to standardise a services-based infrastructure for the Grid environments, named OGSA (Open Grid Services Architecture). OGSA represents an evolution towards a Grid system architecture based on web services concepts and technologies by introducing semantics and capabilities to web services, statefulness, stateful interactions, transient instances, lifetime management, introspection and notification of state changes at the resources.

Generally OGSA provides state-aware, continuous availability for service applications, data and processing logic. It is based on architecture that combines horizontally scalable, database-independent, middle-tier data caching with intelligent parallelisation and an affinity of business logic with cache data. This enables newer, simpler and more-efficient models for highly scalable service-oriented applications that can take full advantage of service virtualisation and event-driven architectures that exist in grid environments. It provides uniform mechanisms to discover and query resources, while it simplifies the administration and management of heterogeneous systems and resources by offering common management and virtualisation capabilities.

CSA conforms to the existing Grid standards, especially OGSA, and is suitable to integrate heterogeneous systems by incorporating both OGSA services among them and extending OGSA infrastructures with additional services. This is achieved through the core GEMBus services described earlier for security, registration, accounting and composition, and to the integration patterns described by supporting a service lifecycle model for dynamic provisioning.

7.2 GEMBus in the Context of GÉANT Business Architectures

JRA2 T1 has made a thorough analysis of multi-domain business processes in a peer-to-peer relationship for interactions in the GÉANT ecosystem [DJ2.1.1]. Figure 7.2 shows the operations corresponding to the decomposition of GÉANT-NREN business processes in both NREN to NREN interactions and NREN to GÉANT interactions.

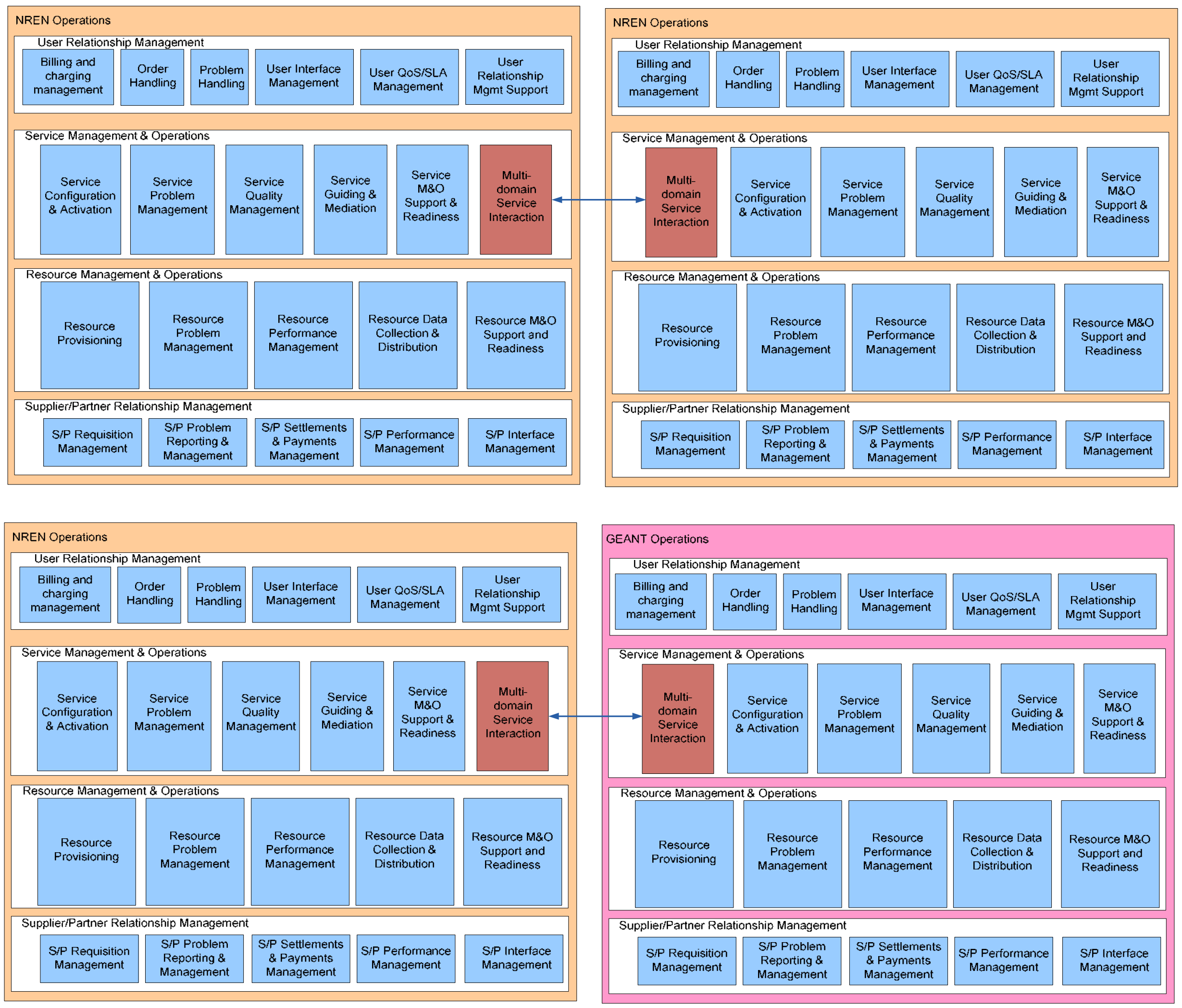

NREN Operations
User Relationship Management
Billing and charging management
Order Handling
Problem Handling
User Interface Management
User QoS/SLA Management
User Relationship Mgmt Support
Service Management & Operations
Service Configuration & Activation
Service Problem Management
Service Quality Management
Service Guiding & Mediation
Service M&O Support & Readiness
Multi-domain Service Interaction
Resource Management & Operations
Resource Provisioning
Resource Problem Management
Resource Performance Management
Resource Data Collection & Distribution
Resource M&O Support and Readiness
Supplier/Partner Relationship Management
S/P Requisition Management
S/P Problem Reporting & Management
S/P Settlements & Payments Management
S/P Performance Management
S/P Interface Management
NREN Operations
User Relationship Management
Billing and charging management
Order Handling
Problem Handling
User Interface Management
User QoS/SLA Management
User Relationship Mgmt Support
Service Management & Operations
Multi-domain Service Interaction
Service Configuration & Activation
Service Problem Management
Service Quality Management
Service Guiding & Mediation
Service M&O Support & Readiness
Resource Management & Operations
Resource Provisioning
Resource Problem Management
Resource Performance Management
Resource Data Collection & Distribution
Resource M&O Support and Readiness
Supplier/Partner Relationship Management
S/P Requisition Management
S/P Problem Reporting & Management
S/P Settlements & Payments Management
S/P Performance Management
S/P Interface Management
NREN Operations
User Relationship Management
Billing and charging management
Order Handling
Problem Handling
User Interface Management
User QoS/SLA Management
User Relationship Mgmt Support
Service Management & Operations
Service Configuration & Activation
Service Problem Management
Service Quality Management
Service Guiding & Mediation
Service M&O Support & Readiness
Multi-domain Service Interaction
Resource Management & Operations
Resource Provisioning
Resource Problem Management
Resource Performance Management
Resource Data Collection & Distribution
Resource M&O Support and Readiness
Supplier/Partner Relationship Management
S/P Requisition Management
S/P Problem Reporting & Management
S/P Settlements & Payments Management
S/P Performance Management
S/P Interface Management
GEANT Operations
User Relationship Management
Billing and charging management
Order Handling
Problem Handling
User Interface Management
User QoS/SLA Management
User Relationship Mgmt Support
Service Management & Operations
Multi-domain Service Interaction
Service Configuration & Activation
Service Problem Management
Service Quality Management
Service Guiding & Mediation
Service M&O Support & Readiness
Resource Management & Operations
Resource Provisioning
Resource Problem Management
Resource Performance Management
Resource Data Collection & Distribution
Resource M&O Support and Readiness
Supplier/Partner Relationship Management
S/P Requisition Management
S/P Problem Reporting & Management
S/P Settlements & Payments Management
S/P Performance Management
S/P Interface Management

Figure 7.2: Business process decomposition, NREN-NREN, NREN-GÉANT (source: JRA2 T1)

This analysis places particular emphasis on the Multi-domain Service Interaction business processes as unique processes within the GÉANT-NREN environment, missing in current standards but needed for the establishment of multi-domain services in a non-commercial (heterarchical or peer) environment. It is obvious that GEMBus can play a key role in these processes, enabling a direct implementation of the business logic by providing support for core services: service location and access, security, composition and monitoring. Furthermore, the rest of the processes could be implemented by component and composite services, or even by dedicated buses federated through GEMBus. GÉANT operations could be constructed by multi-domain composition.

Appendix A GMI Addressing and Routing Mechanisms

Routing in GEMBus takes place at the messaging layer and uses services identification both as an EPR for SOAP messages and as a URL (used in RESTful services). The following sections describe the conventions that define names and identifiers for the GEMBus EPRs, entities, elements, attributes and properties.

A.1 Namespaces

GEMBus uses dedicated URI namespaces for expressing names, identifiers and properties of the GEMBus entities, elements and attributes. In the appropriate cases the same naming scheme can be applied to enumerated values and attributes.

GEMBus naming allows the use of two types of namespace expressions: URN (Uniform Resource Name) and URI (Uniform Resource Identifier). They are designed to support direct mapping between them.

The URN namespace format is defined as a branch of the GÉANT namespace `urn:geant` specified in RFC4926 and uses the following prefix:

```
urn:geant:gembus
```

The URL namespace format uses the prefix http://geant.net/namespace/gembus/.

Optionally, URL namespace will be resolved into GÉANT website URLs containing GEMBus schema definitions.

The GEMBus namespace allows the definition of lower level namespace branches for specific groups of services or attributes. The current list of top-level namespace branches includes:

- `Protocol`: for elements and attributes in any protocol defined or extended by GEMBus.
- `Security`: for elements and attributes related to the GEMBus security architecture and services.
- `Service`: for elements and attributes related to the services included in GEMBus.

Additional top-level branches will be added as necessary.

A list of all assigned namespace branches shall be maintained within a special metadata file that can be directly used by GEMBus components. GEMBus related SOAP message header elements shall use the default namespace prefix `gembus`. XML schemas defined for other GEMBus components (e.g. GEMBus protocol, metadata, etc.) shall use namespace prefixes constructed in the following way:

```
gembus{branch}
```

Table A.1 provides some initial namespace prefix definitions.

Prefix	**XML Namespace**	**Comment**
`gembus`	`urn:geant:gembus`	This is the default prefix
`gembusp`	`urn:geant:gembus:protocol`	GEMBus message level protocol
`gembussec`	`urn:geant:gembus:security`	Security related elements and attributes

Table A.1: Namespace prefix definitions

A.2 Endpoint References

GEMBus defines its own endpoint reference format and schema in accordance with the WS-Addressing endpoint reference definition, which provides support for the following scenarios, as stated in [WS-Addressing]:

- Dynamic generation and customisation of service endpoint descriptions.
- Identification and description of specific service instances that are created as the result of stateful interactions.
- Flexible and dynamic exchange of endpoint information in tightly coupled environments where communicating parties share a set of common assumptions about specific policies or protocols that are used during the interaction.

According to the WS-Addressing specification, EPRs can be used to complement WSDL `<service/>` elements to allow easy exchange and update service endpoint information including dynamic policy assignment and dynamic configuration information.

WS-Addressing defines the EPR element in the following form:

```
<wsa:EndpointReference>
    <wsa:Address>xs:anyURI</wsa:Address>
    <wsa:ReferenceProperties>... </wsa:ReferenceProperties> ?
    <wsa:ReferenceParameters>... </wsa:ReferenceParameters> ?
    <wsa:PortType>xs:Qname</wsa:PortType> ?
    <wsa:ServiceName PortName="xs:NCName"?>xs:Qname</wsa:ServiceName> ?
    <wsp:Policy> ... </wsp:Policy>*
</wsa:EndpointReference>
```

GEMBus EPRs use the top-level elements defined by the WS-Addressing specification and introduce new elements down in the hierarchy to support the required GEMBus functionality.
The GEMBus EPR profile conforms to the minimal requirements principle discussed in section 2. It is intended to allow easy mapping to service addresses expressed in the form of a URL string to provide transparency between SOAP-based and RESTful services. The following considerations apply to the GEMBus EPR element definitions:

- The content of the `<wsa:ReferenceProperties>` element defines configurable parameters of the composable services that remain constant for the whole service lifecycle period.
- The content of the `<wsa:ReferenceParameters>` defines variables and session related data that are subject to change during service operation.
- All GEMBus defined elements and attributes should have a (registered) ID in the form of an FQN (Fully Qualified Name) under the GEMBus namespace.
- All GEMBus defined elements will use the namespace prefix `gembus:service`.
- It is a decision of the service developer to include a service name into the element name. However it is recommended that the element's FQN uses a namespace branch defined for the service or service type.

Table A.2 and Table A.3 list all currently specified service properties and parameters and are provided primarily for illustration purposes. Complete lists will emerge with the practical GEMBus implementation and testing.

Property name	**Element name and attributes**	**Property ID (FQN including namespace)**	**Comment**
`DomainID`	`<gembus:DomainID>`	`urn:geant:gembus:service:property:domain-id`	Service domain expressed in DNS format
`ServiceRegistryKey`	`<gembus:ServiceRegistryKey>`	`urn:geant:gembus:service:property:registry-key`	Unique key assigned to registry
`AnyProperty`	`<gembus:AnyProperty>`	`urn:geant:gembus:service:property:property-any`	Just template

Table A.2: Service properties

Parameter name	**Element name and attributes**	**Parameter ID (FQN including namespace)**	**Comment**
`SessionID`	`<gembus:SessionID>`	`urn:geant:gembus:service:parameter:session-id`	Session type or relation to other session(s)
`SessionDuration`	`<gembus: SessionDuration >`	`urn:geant:gembus:service:parameter:session-duration`	
`AnyParameter`	`<gembus:AnyParameter>`	`urn:geant:gembus:service:parameter:parameter-any`	Just template

Table A.3: Service parameters

When translating an EPR into a URL string the following conventions are used:

- URL string is composed as follows: `domainName/service/serviceName/{propertyKeyValuePairs}/{parameterKeyValuePairs}`
- `propertyKeyValuePairs` defines the configurable parameters of the service
- `parameterKeyValuePairs` defines variables and session related data
- Lists of key-value pairs are separated with semicolon: `;`
- No quotation marks are allowed in the URL string.

The following example illustrates the EPR definition for a `LanguageAssess` service in the CLARIN [CLARIN] domain:

```
<wsa:EndpointReference>
   <wsa:Address>http://clarin.geant.net/languageassess</wsa:Address>
    <wsa:ReferenceProperties>
       <gembus:domainID>clarin.geant.net</gembus:domainID>
<gembus:ServiceRegistryKey>K2349456076</gembus:ServiceRegistryKey>
    </wsa:ReferenceProperties>
    <wsa:ReferenceParameters>
       <gembus:sessionID>173945623490764234854</gembus:sessionID>
       <gembus:sessionDuration>8460</gembus:sessionDuration>
    </wsa:ReferenceParameters>
    <wsa:PortType>gembus:LanguageAssessPortType</wsa:PortType>
    <wsa:ServiceName PortName="LanguageOfText">
         urn:geant:gembus:clarin:textlanguage</wsa:ServiceName>
</wsa:EndpointReference>
```

The above example can be mapped to the following URL string:

http://clarin.geant.net/languageassess/textlanguage/
ServiceRegistryKey=K2349456076/
sessionID=173945623490764234854;sessionDuration=8460

When sending a SOAP message to this service, the contents of the message information header blocks should be similar to:

```
<S:Envelope
     xmlns:S=”http://www.w3.org/2003/05/soap-envelope”
     xmlns:wsa=”http://schemas.xmlsoap.org/ws/2004/08/addressing”
     xmlns:gembus=”urn:geant:gembus”>
  <S:Header>
  <wsa:MessageID>
      message-id:aaaabbbb-cccc-dddd-eeee-wwwwwwwwwwww
  </wsa:MessageID>
  <wsa:RelatesTo>
      message-id:aaaabbbb-cccc-dddd-eeee-ffffffffffff
  </wsa:RelatesTo>
  <wsa:To S:mustUnderstand=”1”>
      http://clarin.geant.net/languageassess
  </wsa:To>
  <wsa:Action>
      http://clarin.geant.net/languageassess/LanguageOfText
  </wsa:Action>
  </S:Header>
  <S:Body>
    Text to be evaluated according to CLARIN schema/format
  </S:Body>
</S:Envelope>
```

Appendix B Sample GEMBus Security Token

A sample SAML assertion following the procedures described in section 3.3.2 for a given consumer with identifier:

```
urn:geant:gembus:component:perfsonarclient:NetflowClient10082.
```

acting on behalf of a user that it is identified by an IdP with identifier:

```
urn:geant:edugain:be:uninett:idp1
```

and connecting to a producer identified by:

```
urn:geant:gembus:component:perfsonarresource:netflow.uninett.no/data
```

should have a SAML 2.0 content as the one displayed below (some line breaks and indentation have been added to improve readability):

```
<?xml version="1.0" encoding="UTF-8"?>
<Assertion xmlns="urn:oasis:names:tc:SAML:2.0:assertion"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:oasis:names:tc:SAML:2.0:assertion
   file:/Users/andreas/Documents/UNINETT/AAISpecs/SAML-2.0/oasis-sstc-saml-
schema-assertion-2.0.xsd"
   Version="2.0" ID="100001" IssueInstant=”2006-12-03T10:00:00Z”>
  <Issuer>
    urn:geant:gembus:component:perfsonarclient:NetflowClient10082"
  </Issuer>
<!-- An audience restriction, that will restrict this security token to be
valid for one single resource only. -->
  <Conditions>
    <AudienceRestriction>
      <Audience>
        urn:geant:gembus:component:perfsonarresource:netflow.uninett.no/data
      </Audience>
    </AudienceRestriction>
  </Conditions>
```

```
    <Subject>
      <NameID>aksjc7e736452829we8</NameID>
      <SubjectConfirmation Method=”urn:geant:edugain:reference:relayed-trust”>
        <SubjectConfirmationData>
          <Assertion xmlns="urn:oasis:names:tc:SAML:2.0:assertion"
           xmlns:xsi="http://www.w3.org/2006/XMLSchema-instance"
           Version="2.0" ID="_200001" IssueInstant="2006-12-03T10:00:00Z">
            <Issuer>urn:geant:edugain:be:uninett:idp1</Issuer>
<!-- This inner assertion is limited to only be valid for the client performing
the WebSSO authentication. This inner assertion cannot be reused or used at all
by others than the NetflowClient10082 instance. But NetflowClient10082 can use
it as an evidence when used inside an assertion issued by NetflowClient10082
using the relayed-trust confirmationMethod. -->
            <Conditions>
              <AudienceRestriction>
                <Audience>
                  urn:geant:gembus:component:perfsonarclient:NetflowClient10082
                </Audience>
              </AudienceRestriction>
            </Conditions>
<!-- This is the inner Subject and authNstatement proving the authentication
itself. These elements and attributes must be identical in the inner and outer
assertion:
     - Assertion/Subject/NameID
     - Assertion/AuthnStatement@AuthenticationMethod
     The inner assertion confirmation Method must be
urn:oasis:names:tc:SAML:1.0:cm:bearer. -->
            <Subject>
              <NameID>aksjc7e736452829we8</NameID>
              <SubjectConfirmation
Method=”urn:oasis:names:tc:SAML:2.0:cm:bearer”/>
            </Subject>
            <AuthnStatement AuthnInstant="2006-12-03T10:00:00Z">
              <AuthnContext>
                <AuthnContextClassRef>
                  urn:oasis:names:tc:SAML:2.0:ac:classes:Password
                </AuthnContextClassRef>
              </AuthnContext>
            </AuthnStatement>
            <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
<!-- Signed by the IdP (or Home Bridging element) -->
```

```
              <SignedInfo>
                <CanonicalizationMethod Algorithm="…"/>
                <SignatureMethod Algorithm="…"/>
                <Reference>
                  <DigestMethod Algorithm="…"/>
                  <DigestValue/>
                </Reference>
              </SignedInfo>
              <SignatureValue/>
            </Signature>
          </Assertion>
        </SubjectConfirmationData>
      </SubjectConfirmation>
    </Subject>
  <!-- The authNstatement issued by the client itself -->
    <AuthnStatement AuthnInstant="2006-12-03T10:00:00Z">
      <AuthnContext>
        <AuthnContextClassRef>
          urn:oasis:names:tc:SAML:2.0:ac:classes:Password
        </AuthnContextClassRef>
      </AuthnContext>
    </AuthnStatement>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <!-- Signed by client -->
      <SignedInfo>
        <CanonicalizationMethod Algorithm="…"/>
        <SignatureMethod Algorithm="…"/>
        <Reference>
          <DigestMethod Algorithm=".."/>
          <DigestValue/>
        </Reference>
      </SignedInfo>
      <SignatureValue/>
    </Signature>
  </Assertion>
```

Appendix C Practical Case: AC Prototype

The architecture depicted in this document is not simply a product of technology reviews or based on previous experiences on related projects. The team developing GEMBus has performed different experiments to assess the feasibility of the proposals presented here. This section describes the most elaborate experiment so far, available as a demonstrator of the GEMBus potential.

The work described here is a step towards achieving the challenge of building self-managed systems, by providing the necessary Autonomic Computing (AC) services onto GEMBus.

C.1 Autonomic Computing

Autonomic Computing (AC) is an initiative to develop computer systems capable of self-management to overcome the increasingly complicated task of managing the rapidly growing number of different distributed computing resources. This task is hampered by the rapidly growing complexity, dynamism and heterogeneity of computer systems. AC systems are defined as “computing systems that can manage themselves given high-level objectives from administrators” [AC]. This definition encompasses the key principle behind AC: administrators (humans) set the rules (policies) by which systems should be guided and those systems are responsible for enforcing them.

At present, most network management is done using certain methods for monitoring and configuring networked elements, either virtual or physical equipment. These methods are normally used by specific software that permits administrators to manage multiple elements from a central place, but almost all tasks need human intervention. As the number of network elements grows, this task is becoming more and more complicated to accomplish and a new management paradigm is needed in this field. AC presents a good solution to achieve the necessary self-management capabilities in modern networks and services.

AC defines four essential tasks:

- **Monitor**: Collect several state data from the environment.
- **Analyse**: Examine the collected data to get a more accurate current high-level state of the environment.
- **Plan**: Decide if the current environment state is valid and, if not, decide what actions should be taken to reach a valid state.
- **Execute**: Make the environment elements to perform the corresponding changes to the decisions.

These tasks, together with the knowledge acquired over time, perform the whole lifecycle of the autonomic management.

Section C.2 describes the basic architecture has a service for each of these tasks:

- Collector.
- Analyzer.
- Decider.
- Changer.

This separation of concerns in different services provides flexibility and scalability to the system. For example, to support more types of events, a system only needs an enhanced Data Collector. To support more complex environment analysis, only new analysis expressions are required. The effort required to overcome infrastructure changes in the environment is rather limited. Moreover, its services are able to live in different machines and be replicated to scale, attending as many events as needed.

In the particular case of AC, there is the self-management of networked systems and services. It is one of the key challenges of the Future of Internet (FI) [FI]. Some approaches try to cover the self-management requirements, such as the *Self-organising Management Overlays for Future Internet Services* [SOMO]. Other approaches have emerged from Cloud Computing [VCLOUD] and virtualisation in general [VIRSYSMNG], but most of them leave management autonomy out of their goals.

Providing AC services in the GEMBus framework can benefit the behaviour of other services and applications connected to the bus, gaining self-management capabilities, thus overcoming the increasing complexity of network services as a starting point to meet the challenges of AC [ACOV].

C.2 Basic Architecture

The basic architecture, introduced in section C.1, aims to provide the necessary services and logic for building autonomic network management solutions that ensure compliance with the policies set by network administrators throughout the organisation and eventually beyond its boundaries. This architecture was originally designed to be the foundation of the AC services for the GEMBus framework. Here it is used to illustrate the applied design principles.

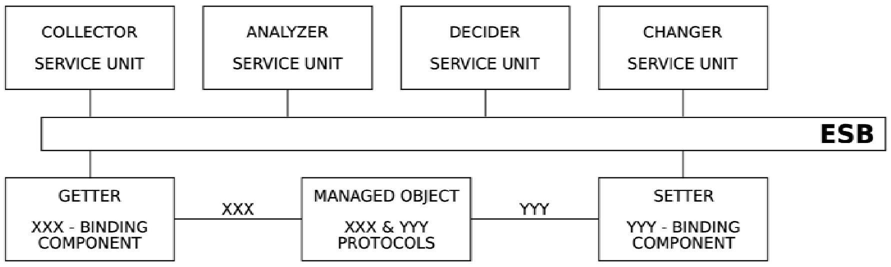

COLLECTOR SERVICE UNIT
ANALYZER SERVICE UNIT
DECIDER SERVICE UNIT
CHANGER SERVICE UNIT
ESB
GETTER XXX - BINDING COMPONENT
XXX
MANAGED OBJECT XXX & YYY PROTOCOLS
YYY
SETTER YYY - BINDING COMPONENT

Figure C.1: Basic autonomic network management architecture

Figure C.1 shows the components involved in the management architecture: the ESB, the main services (Data Collector, Data Analyser, Decider and Changer) and a managed object that is connected through the binding components used to adapt its interaction protocols. Each component is described in the following paragraphs.

The ESB works as a communication bus and service container, so it is used to integrate the different components of the architecture. Current implementations are based on Apache ServiceMix ESB [SERVICEMIX]. The services are packaged as a Service Assembly and may contain Service Units (SU) and Binding Components (BC).

Two integration technologies are tightly bound to ESBs: JBI and OSGi. Java Business Integration (JBI) is a specification for an approach to implementing an SOA. It is built on a message-centered model; a key component in JBI is the Normalised Message Router (NMR). The NMR delivers messages among services, which are connected through endpoints. JBI defines two types of services, Binding Component (BC) and Service Engine (SE). The former component type is intended for connecting external services and adapting their protocol, while the latter is intended to hold business models and other tools. OSGi is a standard that provides the definition of a complete component model and lifecycle management tool, acting as component and service container. It permits a user to remotely install, start, stop, update and uninstall components without needing to restart the platform. OSGi also manages a registry with the service or services offered by each component, so component dependencies can be resolved dynamically.

The Managed Object represents an abstract object that is connected to the architecture. It can be any network equipment (physical or virtual) that can offer an API for reading its configuration and state as well as changing its configuration. The way a managed object is connected to the bus depends on its capabilities. Certain managed objects that support the ESB protocol could be directly connected to the bus while others need to be connected through binding components that adapt certain protocols to the bus protocol. Once a protocol is provided by a binding component, any managed object which uses that protocol can be connected to the bus. The architecture described here represents two binding components, one for the protocol used to receive messages from the managed objects and the other for the protocol used to send messages to the managed objects. Certain combinations of protocol and managed objects can support the two operations, so only one binding component is needed to connect the object to the bus.

The architecture uses BCs to permit the interaction with managed objects that cannot be directly connected to the ESB. On one hand, the *Getter* is a BC that permits other connected services to retrieve configuration and state data from managed objects from outside the ESB. On the other hand, the *Setter* is a BC that permits other connected services to send commands to managed objects from outside the ESB. There are BCs that can be used to read and write data so only one is needed to interact with the managed objects it exposes. This situation makes that the getter and setter binding components mere conceptual artifacts that can live alone or integrated into other services or components.

The Data Collector is responsible for the data gathering and initial manipulation. It provides the collected data in a common format to the rest of components, performing the necessary transformations. The data acquired can be either *pulled* or *pushed*, depending on the managed object or its configuration. To achieve these objectives, this service interacts with the binding components that act as interfaces to the managed objects. Then, the data-flow between a Managed Object and the Data Collector can be in multiple supported standard protocols, such as SNMP. Those protocols are adapted by the BCs, but the Collector adapts the data format if required by other services.

The Data Analyser receives events and reports desired states to other services, such as the Decider. In the current prototype, it has been built with Esper, a Complex Event Processor (CEP) [ESPER], which is set to process received XML messages with SNMP entries to detect and report *saturated* and *unsaturated* states.

Previous state	**Current State**	**Action**
unsaturated	unsaturated	no action
unsaturated	saturated	change route to L2
saturated	saturated	no action
saturated	unsaturated	change route to L1

Table C.1: Decider state/action mapping

The Decider service determines the actions to be performed by managed objects for each state change from system policies. GEMBus previous architecture holds policies in a simple state-action mapping, as Table C.1: Decider state/action mapping

shows. This mapping represents two example policies:

- Priority of the main link to send the outgoing traffic.
- Correct saturation using the alternative link.

The Changer service ensures that each managed object receives and performs the actions set by the Decider. It can use many protocols and message formats, such as SNMP, SSH and HTTP. GEMBus previous architecture works with SSH protocol for sending messages and shell commands for action definitions.

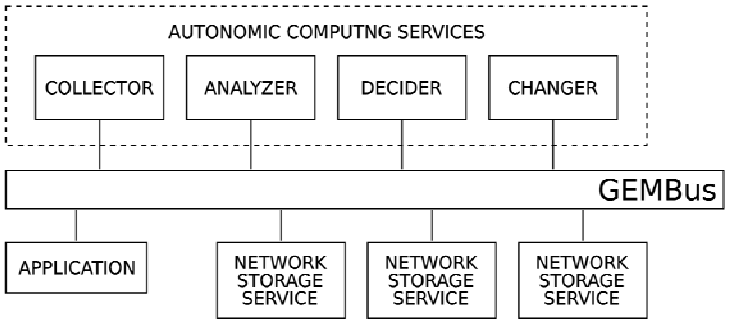


Figure C.2: Autonomic Computing services within GEMBus

Finally, the architecture is designed to support several managed objects connected to the ESB, so many managed objects can be involved in the autonomic management process. For example, Figure C.2 depicts how the same AC services used in the autonomic management architecture can work inside GEMBus. Here, each network storage service acts as a managed object, and the application is the emergent service that offers the composed network storage service and urges the AC services to manage them. As a result, the whole storage system is self-managed.

C.3 Multi-domain Architecture

To get an enhanced architecture for the AC services included in GEMBus, it was necessary to evolve the basic architecture discussed in section C.2. This included adding new necessary components and changing the behaviour of some existing services. In addition, as one of GEMBus goals is to permit multiple domains to cooperate, the task included communicating services from different ESB instances. Some new components are focused on this task.

Before introducing the new components, it is necessary to discuss the new behaviour of the whole architecture. The new strategy consists in gathering (collecting) necessary data from designated elements of the environment to build an environment description set. Then, this information is analysed to find, on the one hand, new environment records to add to the environment description set or, on the other hand, new values for existing records. This more complete set is then used to ask the Policy Manager, a new service, for the correctness of the environment. If the environment state is right, no action is taken. Otherwise, the Policy Manager will point out to the necessary actions that should be taken to fix any problem found in the environment. Finally, the corresponding elements are informed with the decision and they will comply with it.

To follow this new strategy, both existing components and their connections have been changed. The most significant change in the component wiring is in the Collector-Analyser-Decider communications. The Collector sends the data it receives to the Analyser but now, instead of sending its results directly to the Decider, the Analyser sends them again to the Collector. The new steps in the data-flow are as follows:

1. The Collector receives messages describing the environment. The messages can be sent from different sources and can contain one or more registers. The Collector sends those messages to the Analyser.
2. The Analyser, capable of detecting complex events comprising several messages, receives the environment information and sends back a new message to the Collector if it detects any new event related to an additional environment state.
3. The Collector receives new messages from the Analyser and updates its environment description set.
4. The Collector, for each received message, composes a new message with the environment description set and sends it to the Decider. It is desirable that, after a period of time without receiving any message, the Collector sends a reminder to the Decider, again with the environment description set.
5. The Decider asks the Policy Manager for the correctness of the environment it received. With the result obtained from the Policy Manager and the environment description set received from the Collector, it composes a new message and sends it to the Changer.
6. The Changer communicates the actions (obligations) determined by the Policy Manager to the elements that should perform them.

Other new components and services are added to the architecture to support and promote AC services federation. They are divided into two groups: *local* (passive) components and *remote* (active) components. These compose the global new architecture, but they are kept conceptually separated to maintain their different roles. These two groups are described as follows:

C.3.1 Local/Passive Components

Besides adding new components, functionality of the existing components has been slightly changed to meet this new data-flow. This section describes the structure and definition of the main elements of the new architecture. These main elements are meant to live locally, near the other services and components used to build the emergent system. They are labelled local (passive) components because they do not act by themselves. They are only responsible for information retrieval and processing. They do not perform actions.

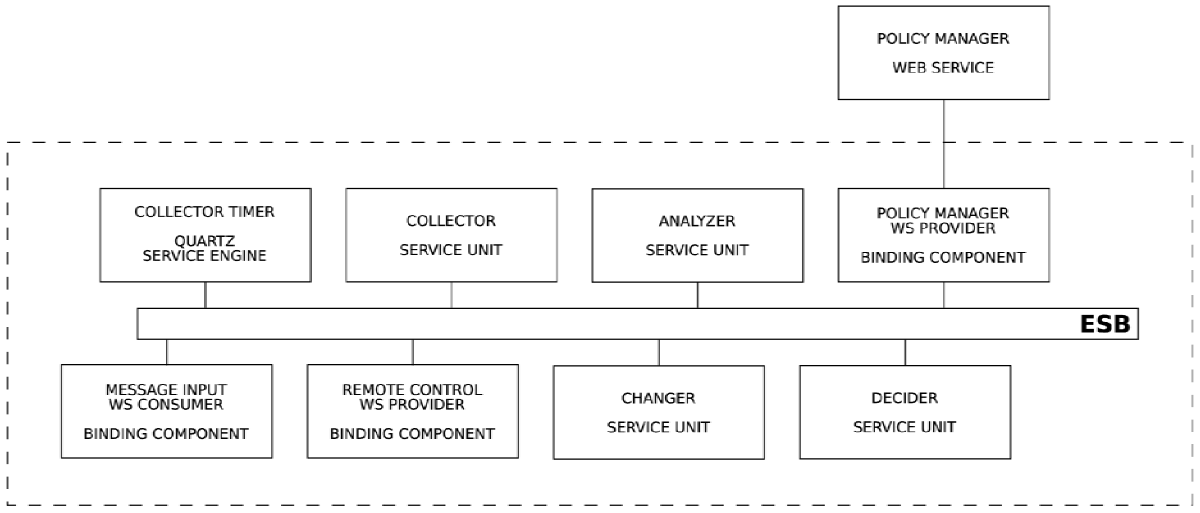


Figure C.3: Enhanced architecture (Autonomic Computing services)

Figure C.3 shows the new architecture which depicts both new components and the components inherited from the old architecture described in the previously. Although their behaviour has been slightly changed, the main services remain in the new architecture. A Policy Manager has been added to the main services, which include Collector, Analyser, Decider, Policy Manager and Changer. The other components shown in Figure C.3 are included for interfacing with other elements, such as Policy Manager WS Provider, Message Input WS Consumer and Remote Control WS Provider, as well as helper services, such as Collector Timer. The behaviour of each component is described as follows:

C.3.1.1 *Enterprise Service Bus*

This component did not change its behaviour because it is only the service container and the communication bus. It hosts the JBI-based NMR that is used to deliver messages among components. In the new architecture Apache ServiceMix ESB 3.0 to FUSE ESB 4.2 was changed. Although it is based on Apache ServiceMix 4.0, it integrates many other technologies and standards. It remains an open-source solution and is supported by a private corporation and fulfils most of the needs of many interested organisations.

C.3.1.2 *Collector*

This component now has more functionality than in the previous architecture. It still receives information from the different elements of the environment, which is then sent to the Analyser. As stated previously, it now builds an environment description set that is completed with messages sent by the Analyser and sends it to the Decider. Moreover, it can receive messages from the Collector Timer that urge it to build a new message with the environment description set and send it to the Decider. The Collector is built in a generic manner, and it can handle three types of messages. The first type is the raw state message that is usually received through the Message Input WS Consumer, coming from outside the ESB. The second message type is an environment description message that contains concrete environment description registers, which it must include in its environment description set, overwriting the existing ones if their key/name matches. Finally, the Collector can handle the messages sent by the Collector Timer that prompt it to resend the environment state message to the Decider.

C.3.1.3 *Collector Timer*

This component is built with the Quartz Service Engine included in FUSE ESB to send periodic messages to the Collector.

C.3.1.4 *Analyser*

The functionality of this component is almost the same as it was in the previous architecture definition, keeping its Esper-based CEP. It receives raw state entries and uses its CEP to find complex events. If they match the defined statements, it generates a new environment message with the result of the analysis and sends it to the Collector.

C.3.1.5 *Decider*

This component has completely changed its behaviour. It was a basic proof-of-concept in the previous architecture, and as such it only had a state-action map to decide what to do in some situations. Now the Decider gets the environment description set and builds a request message to ask the Policy Manager about the suitability of the environment. The Policy Manager, depending on the policies previously stored by the administrators, sends a *Permit* or *Deny* response, together with obligations if necessary. The contents of this response are included in a new message with other environment description registers, such as the subject and destination of the action, which can contain many values. Finally, this message is sent to the Changer.

C.3.1.6 *Policy Manager*

This component is based on an external web service, which itself is based on XACML [XACML], using the implementation offered in XACML-Light [XACMLLIGHT]. It is set by administrators using its interface to manage XACML policies. To interact with this service, the other services use the Policy Manager WS Provider.

C.3.1.7 *Policy Manager WS Provider*

This component is a binding component that adapts the messages sent through the ESB to the external Policy Manager web service. This is specifically through the NMR using JBI, so the requests and responses of the WS are encoded in JBI. The Decider uses it to contact the current Policy Manager.

C.3.1.8 *Changer*

This component receives the orders sent by the Decider and sends them to the corresponding destinations indicated in the *action-destination* field of the order message.

C.3.1.9 *Remote Control WS Provider*

This component is a BC that converts the JBI messages received through the ESB into SOAP requests. It receives the messages sent by the Decider and sends them to the correspondent component in the remote domain. This is one of the key components involved in the federation, together with the next component and the other (remote) components defined in section C.3.2. This component is connected with the Remote Control WS Consumer that lives in a remote domain.

C.3.1.10 *Message Input WS Consumer*

This is the other key component to obtain multi-domain federation of AC services. It is a BC used to receive external SOAP messages, adapt them to the ESB and send the new messages to the Collector. In the entire architecture, this component can receive messages from many sources, but there is a remote component to connect to this one: the Message Input WS Provider.

C.3.2 Remote/Active Components

This section describes the remote (active) services. They do not live in the same place as the other components. They initiate the management process by sending the environment state messages to the passive components. They terminate it by performing the commands urged by the other components. These components are key to obtaining the federation capabilities. These components are responsible for crossing the domain boundaries and sending messages to remote AC services.

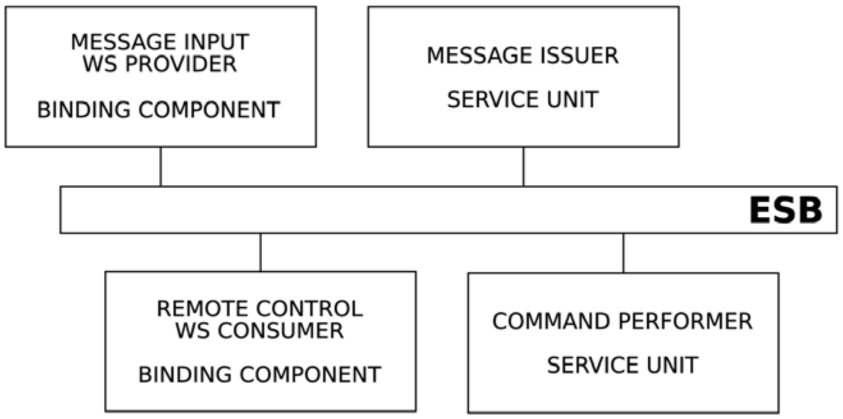


Figure C.4: Remote Autonomic Computing services

Figure C.4 shows the remote components. It shows Message Input WS Provider and Remote Control WS Consumer that are used to interact with the outside. It also shows Message Issuer and Command Performer, providing the actual functionality. Sections C.3.2.1 through C.3.2.4 describe each of these components.

C.3.2.1 *Message Issuer*

This component generates environment state messages and sends them to the *local domain* through the Message Input WS Provider. This component may be incarnated by many components and services, so it is desirable for it to be located near the information source, thereby reducing the delay between the information generation and the delivery of its environment state message.

C.3.2.2 *Message Input WS Provider*

This component uses a SOAP BC and is connected with the Message Input WS Consumer that is located in the *local domain*. This component is mainly used by the Message Issuer.

C.3.2.3 *Remote Control WS Consumer*

This component uses a SOAP BC to receive the orders urged by the *local domain* services. It is connected with the Command Performer, so it adapts SOAP messages to JBI for sending them to the Command Performer.

C.3.2.4 *Command Performer*

This component receives orders and performs them. Many components and services may incarnate the role of this one, but it is either a conceptual or a concrete element. When received orders involve several elements, this component should be instantiated alone and be instructed to route the orders to their destination.

C.4 Adding Self-management Capabilities to eduroam Services

This section describes how AC services can be connected with other services to build a self-managed system or add self-management capabilities to an existing system, using eduroam as an example. As eduroam is a multi-domain service, it can benefit from AC services to achieve cross-domain autonomic management, becoming a self-managed system. It is proposed to instruct the local AC services of an organisation to detect if the same user is connected in two or more remote organisations and, if so, to send disconnection messages to the remote AC services of the organisations on which that user is connected.

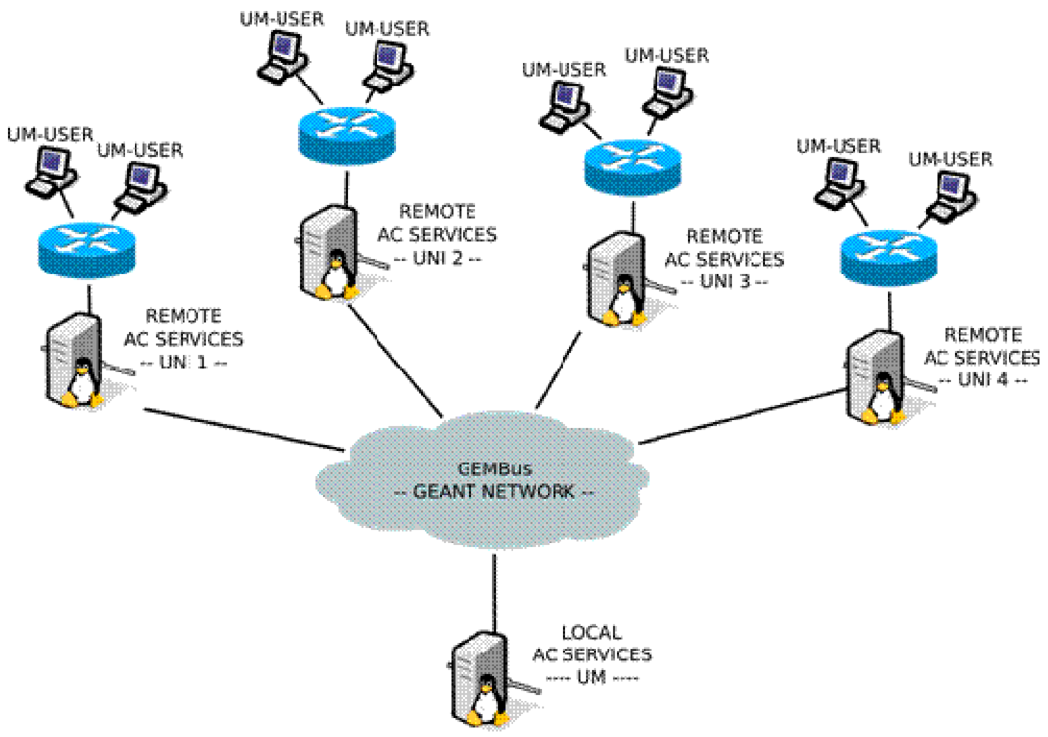

UM-USER
UM-USER
UM-USER
UM-USER
UM-USER
UM-USER
UM-USER
UM-USER
REMOTE AC SERVICES -- UNI 1 --
REMOTE AC SERVICES -- UNI 2 --
REMOTE AC SERVICES -- UNI 3 --
REMOTE AC SERVICES -- UNI 4 --
GEMBus -- GEANT NETWORK --
LOCAL AC SERVICES ---- UM ----

Figure C.5: Network environment

Figure C.5 shows the operating network environment for the self-managed system described here. It depicts some different domains involved in eduroam. The local domain (*UM*) is the owner of the many roaming users connected in the remote domains (UNI 1, UNI 2, UNI 3 and UNI 4). All domains are connected through the GÉANT network.

In this scenario, the AC services deployed in the remote domains report that UM users are connected to their infrastructure. The local AC services receive and process the information to send back orders if necessary. This behaviour is achieved by instructing eduroam services or infrastructure to report the roaming user connection to their corresponding AC service which is the Message Issuer or directly to the Message Input WS Provider that is connected to AC services at UM. Moreover, the Command Performer must also be connected with the eduroam service that controls the user connection so it can perform the possible actions sent by UM AC services.

Since eduroam services are tied to their own responsibilities, a good point to make the connection between them and AC services resides in the logging stage. All or maybe some of the logging entries should be formatted and sent to the proper AC service. This way, AC services can be instructed to keep a log of all messages it receives and store the sender, date-time and message content. This can be done by building a logging service and setting proper policies in the Policy Manager to match the desired messages and attach the logging action as an *obligation* to the answer.

C.5 Evaluation

This section describes the experiments performed to evaluate the feasibility of the proposed architecture, showing first the current solution built upon the architecture and how components exchange messages. It then describes the simulation environment used to test the solution.

To perform the experiments, an implementation was made for each component of the architecture. A solution was built for local domain and another different solution was built for remote domains. Both local and remote domain instances are based on the same ESB (FUSE), so the difference is in the services that are installed on each ESB instance. As described previously, AC services are classified as local and remote services. Local services are deployed in the local-domain ESB and remote services are deployed in the remote-domain ESBs.

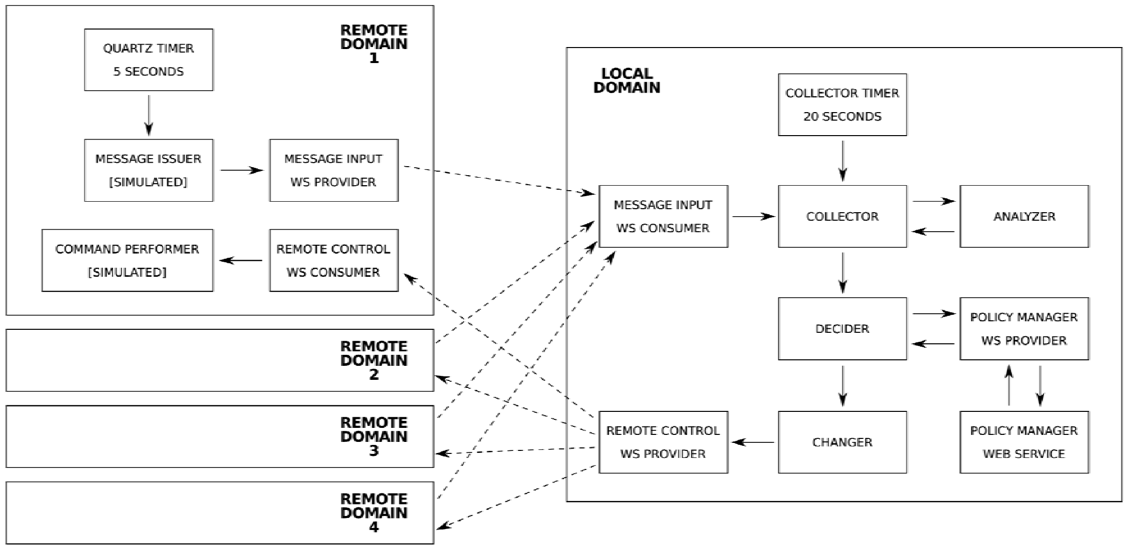

REMOTE DOMAIN 1
QUARTZ TIMER 5 SECONDS
MESSAGE ISSUER [SIMULATED]
MESSAGE INPUT WS PROVIDER
COMMAND PERFORMER [SIMULATED]
REMOTE CONTROL WS CONSUMER
REMOTE DOMAIN 2
REMOTE DOMAIN 3
REMOTE DOMAIN 4
LOCAL DOMAIN
COLLECTOR TIMER 20 SECONDS
MESSAGE INPUT WS CONSUMER
COLLECTOR
ANALYZER
DECIDER
POLICY MANAGER WS PROVIDER
REMOTE CONTROL WS PROVIDER
CHANGER
POLICY MANAGER WEB SERVICE

Figure C.6: Component integration and message flow

Figure C.6 presents the message-flow of the different components. It shows four remote domains and a local domain, but it only shows the internals of remote domains for the first one.

For the purpose of the tests, a Quartz Timer is added to feed the Message Issuer component every five seconds and instruct the same component to send a predefined message to the Message Input WS Provider, announcing that a roaming user is connected to its domain. There are three different remote users: Alice will be connected to remote domain 1; Bob will be connected to remote domain 2 and Chuck will be connected to both remote domains 3 and 4. In this configuration, with all services running at the same time, the AC services must notice that Chuck is connected in more than one place. Starting and stopping the Quartz Timer component of the different servers simulates different situations, such as the users connecting at different times.

In the local domain, the Analyser is configured to detect that more than one user is connected to different remote domains. This is done with the following Esper statement:

```
select ‘
  <set>
    <multiplaces>true</multiplaces>
  </set>
  <merge>
    <action-subject>’ || a.subject || ‘</action-subject>
    <action-destination>’ || a.source || ‘, ‘ || b.source || ‘</action-
destination>
  </merge>
‘ as newRow1
from pattern [
  every a=entry() -> b=entry(
    subject = a.subject,
    source != a.source
  ) where timer:within(5 min)
] .win:length(10)
```

It simply detects that an entry is followed by another entry with the same *subject* but different *source*. It returns an XML-code portion that, as described previously, is sent to the Collector. The code instructs the Collector to set to *true* a new environment entry, called *multiplaces,* and to merge the values of *action-subject* and *action-destination* with the same environment entries already added in the past. If the entries to set already exist, the Collector will replace them with the new ones. If the entries to merge do not exist, the Collector will add them as new entries. In contrast, if the entries to merge already exist, the Collector will add the new contents to the old entry.

Also in the local domain, the Policy Manager is set with a policy to make the system react when the Analyser reports the exceptional situation. As described previously, the Policy Manager is based on XACML. The policy is described in its XML-based language as follows:

```
<Policy xmlns=”urn:oasis:names:tc:xacml:2.0:policy:schema:os”
    PolicyId=”anm:policy:remote:allow”
    RuleCombiningAlgId=
      “urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides”>
  <Description>
    Grants access to resource anm:resource:remote
  </Description>
  <Target/>
```

```
    <Obligations>
      <Obligation FulfillOn=”Deny” ObligationId=”anm:obligation”>
        <AttributeAssignment
          AttributeId=”anm:obligation:attribute:action”
          DataType=”http://www.w3.org/2001/XMLSchema#string”
        >DISCONNECT</AttributeAssignment>
      </Obligation>
    </Obligations>
    <Rule RuleId=”anm:rule:remote:allow” Effect=”Permit”>
      <Condition>
        <Apply FunctionId=”urn:oasis:names:tc:xacml:1.0:function:not”>
          <Apply FunctionId=”urn:oasis:names:tc:xacml:1.0:function:boolean-is-
  in”>
            <AttributeValue DataType=”http://www.w3.org/2001/XMLSchema#boolean”>
              true
            </AttributeValue>
            <Apply FunctionId=”urn:oasis:names:tc:xacml:1.0:function boolean-
  bag”>
              <EnvironmentAttributeDesignator
                AttributeId=”anm:environment:multiplaces”
                DataType=”http://www.w3.org/2001/XMLSchema#boolean”
              />
            </Apply>
          </Apply>
        </Apply>
      </Condition>
    </Rule>
    <Rule RuleId=”anm:rule:remote:deny” Effect=”Deny”/>
  </Policy>
```

As expressed in the policy description, if there is an environment attribute called *multiplaces* set to *true*, the response is *Deny* and is attached with an obligation called *action* and set to the value *DISCONNECT*. The rule is defined in reverse logic to cover a wider range of cases of the boolean condition.

Finally, in this implementation, the Command Performer service only logs the messages it receives without performing any action.

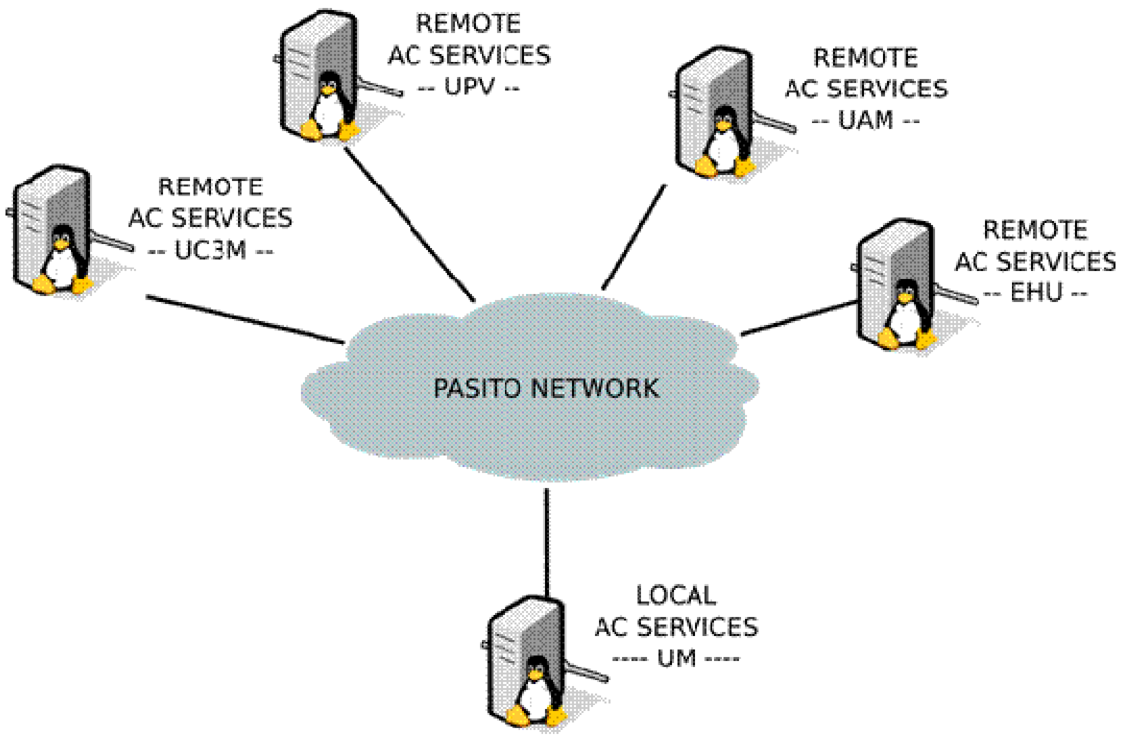


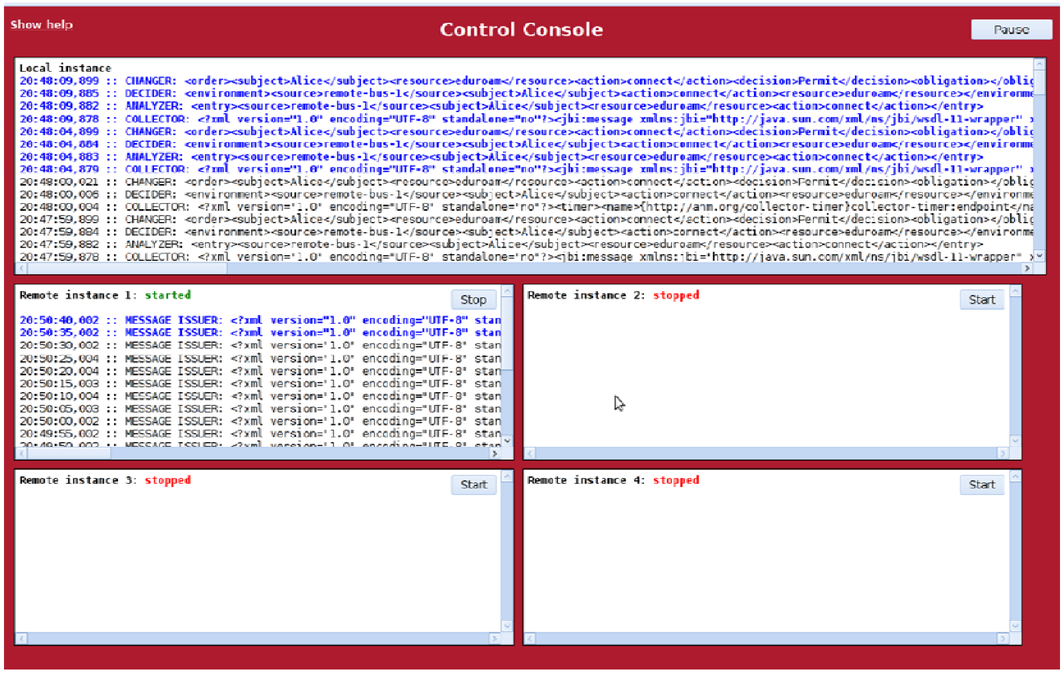
Figure C.7: Simulated network environment

Figure C.7 shows the test network topology used in the GEMBus evaluation. To tie an environment to GEMBus objectives, the evaluation scenario was deployed in five servers connected with the PASITO network [PASITO], the experimental infrastructure for services and protocols provided by RedIRIS. In addition to one local server, the remaining four were placed in other universities distributed throughout the country. The four remote servers, acting as remote domains, run the remote AC services. The local server runs the local AC services. To demonstrate how this scenario behaves in a multi-domain, federated ESB environment, both remote and local systems contain a complete FUSE ESB installation.

The main goal was to get a feasibility approach of the proposed services. After deploying whole systems in the servers, local AC services that wait for remote messages were started. All remote AC services were started except the Quartz Timer ones. After a while, the Collector service received some keep-alive messages sent by the Collector Timer service, but as the environment was right, no action was prompted. Once the whole system was uniformly running, the Quartz Timer services were started in order, waiting for a few messages between them. Then, the remote systems reported that Alice was being connected in remote domain 1, Bob in remote domain 2 and Chuck in remote domains 3 and 4. Finally, when the local AC services noticed that Chuck was connected in two domains at the same time, a message was sent to the remote AC services of domains 3 and 4, ordering them to disconnect Chuck. Log messages kept by the ESB of each server were reviewed to follow the testing process.

C.6 Results

This section describes the results obtained from the test performed to evaluate the proposed architecture. This starts with a brief description of the web application developed to easily obtain the log messages of all ESB instances involved in the evaluation, then showing the messages exchanged by the services during the test performed. As stated previously, the test simulates the concurrent remote connection of the same user in two different domains. The message emitter service of each domain is started in order, so initially only normal messages are received. They continue to be received until starting the service of the fourth domain that emits the messages indicating that a second Chuck is connected (a second user able to identify him/herself as Chuck). At this point, the system reacts and sends a message to the third and fourth domains, indicating that Chuck must be disconnected.



```
Local instance
20:48:09,899 :: CHANGER: <order><subject>Alice</subject><resource>eduroam</resource><action>connect</action><decision>Permit</decision><obligation></oblig
20:48:09,885 :: DECIDER: <environment><source>remote-bus-1</source><subject>Alice</subject><action>connect</action><resource>eduroam</resource></environme
20:48:09,882 :: ANALYZER: <entry><source>remote-bus-1</source><subject>Alice</subject><resource>eduroam</resource><action>connect</action></entry>
20:48:09,878 :: COLLECTOR: <?xml version="1.0" encoding="UTF-8" standalone="no"?><jbi:message xmlns:jbi="http://java.sun.com/xml/ns/jbi/wsdl-11-wrapper" x
20:48:04,899 :: CHANGER: <order><subject>Alice</subject><resource>eduroam</resource><action>connect</action><decision>Permit</decision><obligation></oblig
20:48:04,884 :: DECIDER: <environment><source>remote-bus-1</source><subject>Alice</subject><action>connect</action><resource>eduroam</resource></environme
20:48:04,883 :: ANALYZER: <entry><source>remote-bus-1</source><subject>Alice</subject><resource>eduroam</resource><action>connect</action></entry>
20:48:04,879 :: COLLECTOR: <?xml version="1.0" encoding="UTF-8" standalone="no"?><jbi:message xmlns:jbi="http://java.sun.com/xml/ns/jbi/wsdl-11-wrapper" x
20:48:00,021 :: CHANGER: <order><subject>Alice</subject><resource>eduroam</resource><action>connect</action><decision>Permit</decision><obligation></oblig
20:48:00,006 :: DECIDER: <environment><source>remote-bus-1</source><subject>Alice</subject><action>connect</action><resource>eduroam</resource></environme
20:48:00,004 :: COLLECTOR: <?xml version='1.0' encoding="UTF-8" standalone='no"?><timer><name>{http://anm.org/collector-timer}collector-timer:endpoint</na
20:47:59,899 :: CHANGER: <order><subject>Alice</subject><resource>eduroam</resource><action>connect</action><decision>Permit</decision><obligation></oblig
20:47:59,884 :: DECIDER: <environment><source>remote-bus-1</source><subject>Alice</subject><action>connect</action><resource>eduroam</resource></environme
20:47:59,882 :: ANALYZER: <entry><source>remote-bus-1</source><subject>Alice</subject><resource>eduroam</resource><action>connect</action></entry>
20:47:59,878 :: COLLECTOR: <?xml version='1.0' encoding="UTF-8" standalone='no"?><jbi:message xmlns:jbi="http://java.sun.com/xml/ns/jbi/wsdl-11-wrapper" x
```

```
Remote instance 1: started

20:50:40,002 :: MESSAGE ISSUER: <?xml version="1.0" encoding="UTF-8" stan
20:50:35,002 :: MESSAGE ISSUER: <?xml version="1.0" encoding="UTF-8" stan
20:50:30,002 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
20:50:25,004 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
20:50:20,004 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
20:50:15,003 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
20:50:10,004 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
20:50:05,003 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
20:50:00,002 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
20:49:55,002 :: MESSAGE ISSUER: <?xml version='1.0' encoding="UTF-8' stan
```

```
Remote instance 2: stopped
```

```
Remote instance 3: stopped
```

```
Remote instance 4: stopped
```

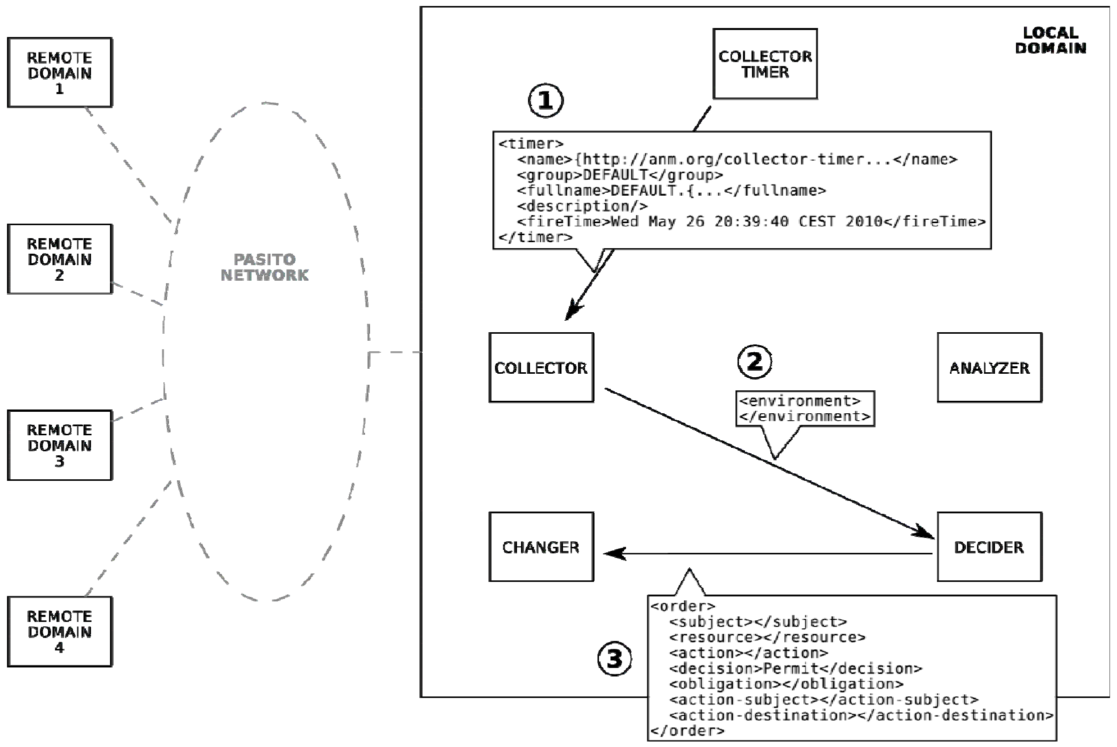
Figure C.8: Control console web application with a few messages per server

The test environment incorporated the OSGi feature that allows remote connections using SSH protocol to get the log messages of each ESB instance. Figure C.8 shows a screenshot of the application. It has a box for each server to display its log messages. All boxes are updated every five seconds. A box is dedicated to the local instance, and each one of the four remaining boxes is dedicated to each remote instance and has a label indicating such. In addition, there is a pause/restart button that prevents the web being updated, but the services are still running in the background. Moreover, there are buttons to start and stop the remote message emitters, as well as a label that indicates if it is started or stopped. The new messages on each box display in blue, so when they are updated, it is easy to see what happened from the last update. Because of the slow interaction channel with the ESB instances, all requests were performed in parallel. Because of the large size of the original HTTP response, *gzip* compression was used to send that response to the client browser.

The following sections describe the message exchanges produced by AC services during the test as well as the overall behaviour of the system in each step. The messages were grabbed from the logs after each action, so there is a record of the exchanged messages of the system when the local instance was the only one started and the exchanged messages after the startup of each remote domain.

C.6.1 Initial State Scenario

The first step in the test is the startup of each ESB instance. After that, the only message exchanges are caused by the Collector-Timer service, which sends a message every 20 seconds to keep alive the other services and make them refresh their state or perform any necessary operation.



```
<timer>
  <name>{http://anm.org/collector-timer...</name>
  <group>DEFAULT</group>
  <fullname>DEFAULT.{...</fullname>
  <description/>
  <fireTime>Wed May 26 20:39:40 CEST 2010</fireTime>
</timer>
```

```
<environment>
</environment>
```

```
<order>
  <subject></subject>
  <resource></resource>
  <action></action>
  <decision>Permit</decision>
  <obligation></obligation>
  <action-subject></action-subject>
  <action-destination></action-destination>
</order>
```

Figure C.9: Message exchanges, periodically generated messages

Figure C.9 shows a simplified test scenario, depicting whole domains as boxes and, for the local domain, only the services involved in the message exchanges. In addition, it shows the message exchange prompted by the Collector-Timer service, which is the first message exchange. The information sent by this service is not important because the exchange only serves to wake up the Collector service. In this exchange, the entire message is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<timer>
  <name>{http://anm.org/collector-timer}collector-timer:endpoint</name>
  <group>DEFAULT</group>
```

```
    <fullname>DEFAULT.{http://anm.org/collector-timer}collector-
  timer:endpoint</fullname>
    <description/>
    <fireTime>Wed May 26 20:39:40 CEST 2010</fireTime>
  </timer>
```

There are many fields in the message, but they are not used as noted previously. The second message is sent by Collector service to Decider service. The environment is empty because remote services did not send a message:

```
  <environment></environment>
```

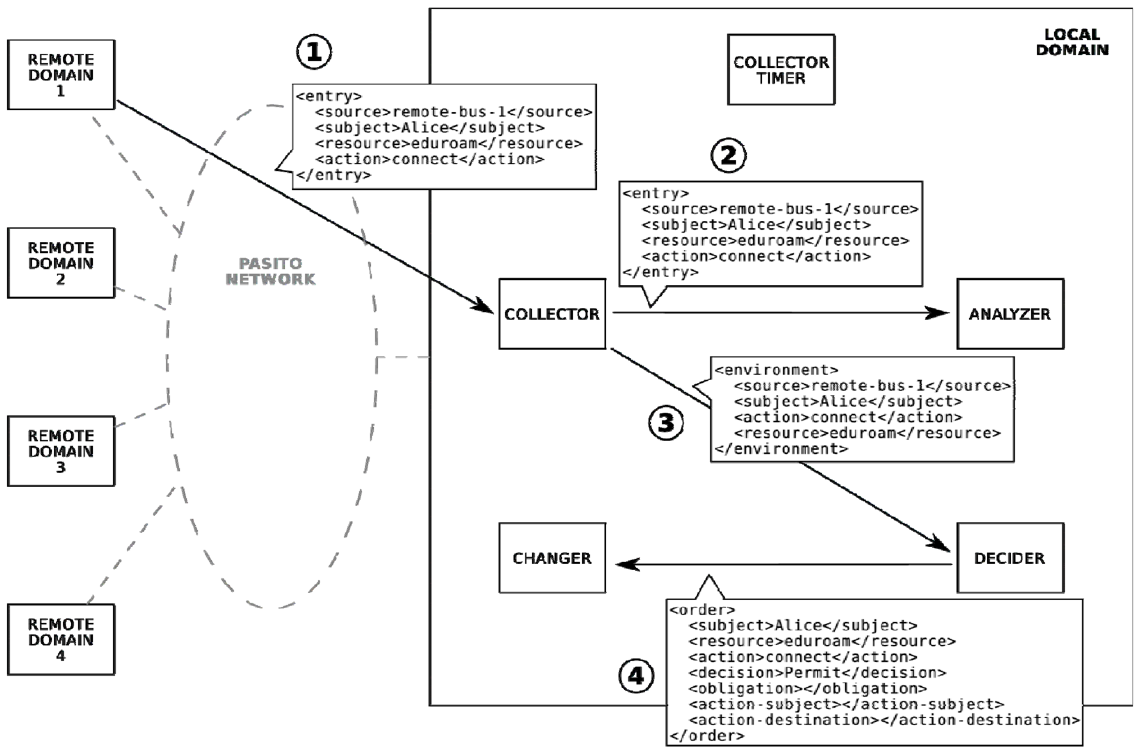
The Decider asks the Policy Manager to determine the correctness of that environment. With the response of the Policy Manager, the Decider builds a message and sends it to the Changer. This is the final message exchange and its content is as follows:

```
  <order>
    <subject></subject>
    <resource></resource>
    <action></action>
    <decision>Permit</decision>
    <obligation></obligation>
    <action-subject></action-subject>
    <action-destination>
    </action-destination>
  </order>
```

There are many fields, but most of them are empty in the present operation. The *subject*, *resource*, *action*, decision and obligation are obtained directly from the policy response but the action-subject and *action-destination* fields are managed by the Decider and obtained from the environment. As the environment is empty, these two fields are also empty.

C.6.2 First Remote Domain Started (Alice)

The next step in the tests is to start the first remote domain. In this scenario, the local domain starts receiving messages from the outside. The Collector service receives those messages, with the messages sent every 20 seconds by Collector-Timer. Remote domains send a message every five seconds. This domain sends messages reporting the connection to Alice.



```
<entry>
  <source>remote-bus-1</source>
  <subject>Alice</subject>
  <resource>eduroam</resource>
  <action>connect</action>
</entry>
```

```
<entry>
  <source>remote-bus-1</source>
  <subject>Alice</subject>
  <resource>eduroam</resource>
  <action>connect</action>
</entry>
```

```
<environment>
  <source>remote-bus-1</source>
  <subject>Alice</subject>
  <action>connect</action>
  <resource>eduroam</resource>
</environment>
```

```
<order>
  <subject>Alice</subject>
  <resource>eduroam</resource>
  <action>connect</action>
  <decision>Permit</decision>
  <obligation></obligation>
  <action-subject></action-subject>
  <action-destination></action-destination>
</order>
```

Figure C.10: Message exchange, remote domain report

Figure C.10 shows the message exchanges provoked by services in this scenario. As in the previous scenario, it only shows the elements involved in the message exchanges. Although they are not displayed in the diagram, this scenario also uses wake-up messages that are not part of the AC services. They are used just to prompt the remote domain to send its corresponding message. Message Issuer receives the following message inside the remote domain:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<timer>
  <name>{http://anm.org/quartz-emitter}quartz-emitter:endpoint</name>
  <group>DEFAULT</group>
  <fullname>DEFAULT.{http://anm.org/quartz-emitter}quartz-
emitter:endpoint</fullname>
  <description/>
  <fireTime>Thu May 27 10:20:15 CEST 2010</fireTime>
</timer>
```

As in the previous domain, no field for this timer message is used. The entire message is used to prompt the remote domain to send a message to the local domain, specifically to the Collector service, indicating that a user is connected. In this case, as the remote domain is the first one, Alice is that user. The entire message reads as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```
<jbi:message
      xmlns:jbi=”http://java.sun.com/xml/ns/jbi/wsdl-11-wrapper”
      xmlns:msg=”http://anm.org/message-input”
      xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
      xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
      name=”DeliveryMessageRequest”
      type=”msg:DeliveryMessageRequest”
      version=”1.0”>
  <jbi:part>
    <tns:DeliveryMessage xmlns:tns=”http://anm.org/message-input/types”>
      <tns:message>
        <entry>
          <source>remote-bus-1</source>
          <subject>Alice</subject>
          <resource>eduroam</resource>
          <action>connect</action>
        </entry>
      </tns:message>
    </tns:DeliveryMessage>
  </jbi:part>
</jbi:message>
```

The Collector service receives this entire message.. Originally, it was received as a SOAP request by Message Input service, but then translated to JBI and sent through the ESB to the Collector service. As a result, this message has all JBI envelopes (*jbi:message*, *jbi:part*) as well as the envelopes defined by Message Input service (*tns:DeliveryMessage*, *tns:message*). The message body, its important part, is delimited by *entry* tag. There are four fields in the message. The first field, *source*, establishes the source domain or ESB instance for the message; in this case it is *remote-bus-1*. The field *subject* then establishes who (element or person) provoked the dispatch of this message. The field *action* determines the action that was performed by the element or person. Finally, the field *resource* determines which resource was involved in the action.

There is no mandatory field in this message. All fields under *entry* tag will be added to the environment description by the Collector service for further processing. With this, the system gains flexibility and generality.

After receiving the message, the Collector service then updates the environment description. In addition, the same message content is sent to the Analyser, but in a raw message. The content of this message is as follows:

```
<entry>
  <source>remote-bus-1</source>
  <subject>Alice</subject>
  <resource>eduroam</resource>
  <action>connect</action>
```

```
</entry>
```

The content received by the Analyser service is the same as in the previous message. This content is processed by the Complex Event Processor (CEP), and must be initialised if that was not done previously. If the CEP finds something wrong, it will send back a message to the Collector service. As the configuration set in the Analyser service is to detect users connected in more than one domain, this scenario does not dispatch the expression, so it does not send a message.

After sending the *entry* message to the Analyser service, the Collector builds the environment description set, composes a message to represent it and sends this message to the Decider. The message is as follows:

```
<environment>
  <source>remote-bus-1</source>
  <subject>Alice</subject>
  <action>connect</action>
  <resource>eduroam</resource>
</environment>
```

As the local domain received only one message, the environment only has the fields received on that message. With this environment description, the Decider service asks for its correctness to the Policy Manager. To build the policy request, the Decider uses the *subject*, *resource* and *action* items from the environment description to set the main policy request fields. It also uses the other environment items to set the environment entries of the policy request. The policy response sent by the Policy Manager is used by the Decider to build and send an *order* message to the Changer service. The message is similar to the one shown in C.6.1, but with the difference that the value of field *subject* is *Alice* (line 2), the value of the field *resource* is *eduroam* and the value of the field *action* is *connect*. In contrast with the message received by the Changer service in the previous scenario, the *subject*, *resource* and *action* fields are filled with the same values found in the environment description set. As the field *action-destination* of the order is empty, the Changer service does nothing with it.

C.6.3 Second Remote Domain Started (Bob)

This section describes the message exchanges provoked by the startup of the Quartz Timer service of the second remote domain. This scenario is almost the same as that in section C.6.2, with the exception that in this scenario the *source* field of the entry sent by the remote domain is set to *remote-bus-2* and the *subject* field is set to *Bob*. The content of the message received by the Collector service is similar to the previous scenario, with the commented differences in lines 2 and 3.

In addition, the remaining messages have the same differences as those in the previous scenario. The last message that is sent to the Changer service is similar to the one in section C.6.1, but with the difference that the value of the field *subject* is *Bob*, the value of the field *resource* is *eduroam* and the value of the field *action* is *connect*. As happens in the previous scenario, in this scenario, the Changer service does not send any message because the field *action-destination* is not fulfilled.

C.6.4 Third remote Domain Started (Chuck)

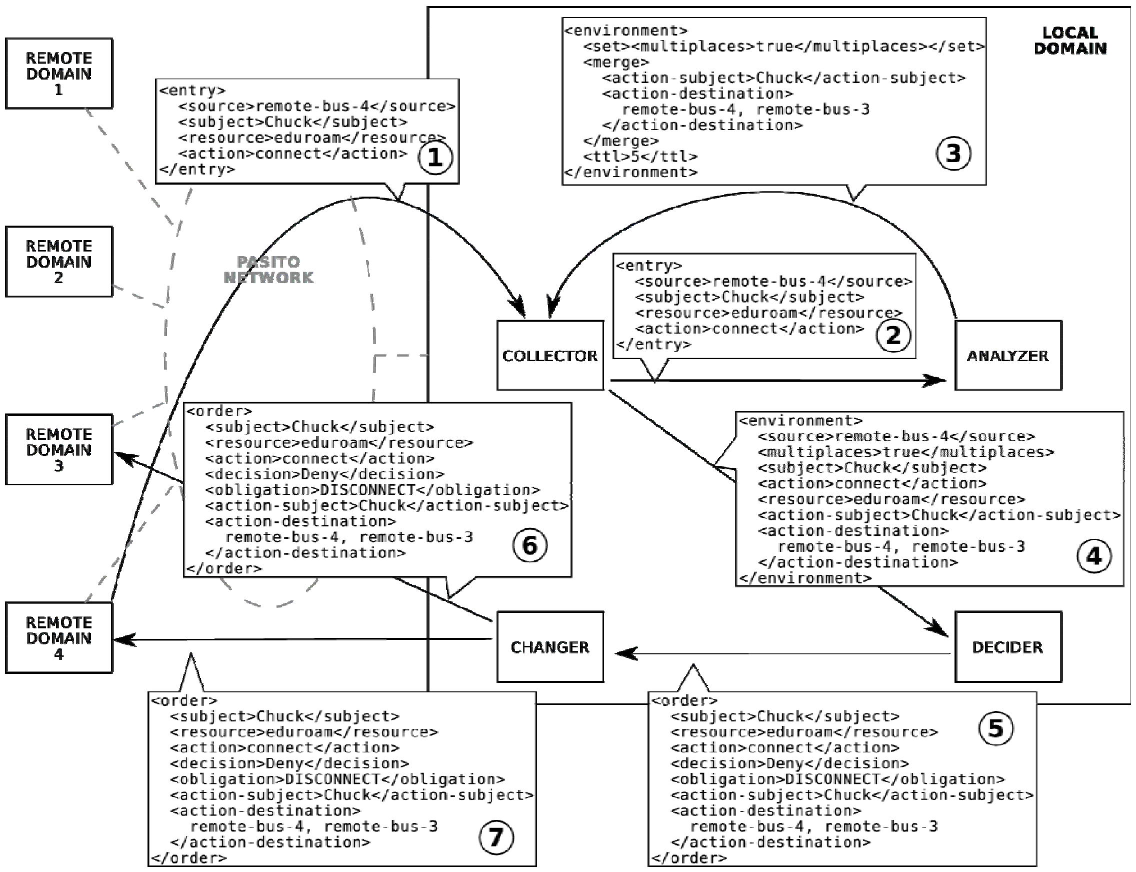
The resulting scenario after starting the Quartz Timer for the third domain is almost the same as the two previous scenarios, but again with differences in the *subject* and *source* fields. For this reason, this scenario is used to note certain particularities of the behaviour of the services. First, the content of the message, without its enclosure, received by the Collector service is similar to the one shown in C.6.2, with the value *remote-bus-3* in the *source* (line 2) and *Chuck* in the *subject* (line 3). The values of the other fields are the same as those of the previous scenarios. This message is then sent to the Analyser service and the updated environment description set is sent to the Decider service. The content of the message sent to the Decider service is as follows:

```
<environment>
  <source>remote-bus-3</source>
  <subject>Chuck</subject>
  <action>connect</action>
  <resource>eduroam</resource>
</environment>
```

Note that for each *entry* message received, as it has the same fields as previous messages, the environment description set is updated and appears that only the last message was processed. The *memory* is kept by the CEP of the Analyser service. If those fields were different for each remote domain, avoiding the overlapping, the environment description set items would be never overwritten. The environment description set should represent the *current* state of the system at any moment. Finally, after asking the Policy Manager service, the content of the message sent to the Changer service is similar to that in C.6.1, but with the *subject*, *resource* and *action* fields filled with the values *Chuck*, *eduroam* and *connect*. Again, as the *action-description* field is not filled, this *order* message is not sent to any remote domain. Note that the *Permit* decision is not processed because, although the environment is right, the Policy Manager is able to report *obligations*. The final recipient of the *order* message has the responsibility of understanding and performing those obligations.

C.6.5 Fourth Remote Domain Started (Chuck, again)

This section describes the fourth and final scenario. It happens just after the startup of the fourth domain. The user associated to that domain is again Chuck. AC services should detect that Chuck is connected in two different domains and to take part on its resolution, sending the appropriate orders to the remote domains which are determined by the Policy Manager service.



```
<entry>
  <source>remote-bus-4</source>
  <subject>Chuck</subject>
  <resource>eduroam</resource>
  <action>connect</action>
</entry>
```

```
<entry>
  <source>remote-bus-4</source>
  <subject>Chuck</subject>
  <resource>eduroam</resource>
  <action>connect</action>
</entry>
```

```
<environment>
  <set><multiplaces>true</multiplaces></set>
  <merge>
    <action-subject>Chuck</action-subject>
    <action-destination>
      remote-bus-4, remote-bus-3
    </action-destination>
  </merge>
  <ttl>5</ttl>
</environment>
```

```
<environment>
  <source>remote-bus-4</source>
  <multiplaces>true</multiplaces>
  <subject>Chuck</subject>
  <action>connect</action>
  <resource>eduroam</resource>
  <action-subject>Chuck</action-subject>
  <action-destination>
    remote-bus-4, remote-bus-3
  </action-destination>
</environment>
```

```
<order>
  <subject>Chuck</subject>
  <resource>eduroam</resource>
  <action>connect</action>
  <decision>Deny</decision>
  <obligation>DISCONNECT</obligation>
  <action-subject>Chuck</action-subject>
  <action-destination>
    remote-bus-4, remote-bus-3
  </action-destination>
</order>
```

```
<order>
  <subject>Chuck</subject>
  <resource>eduroam</resource>
  <action>connect</action>
  <decision>Deny</decision>
  <obligation>DISCONNECT</obligation>
  <action-subject>Chuck</action-subject>
  <action-destination>
    remote-bus-4, remote-bus-3
  </action-destination>
</order>
```

```
<order>
  <subject>Chuck</subject>
  <resource>eduroam</resource>
  <action>connect</action>
  <decision>Deny</decision>
  <obligation>DISCONNECT</obligation>
  <action-subject>Chuck</action-subject>
  <action-destination>
    remote-bus-4, remote-bus-3
  </action-destination>
</order>
```

Figure C.11: Message exchange, remote domain report, duplicated subject

Figure C.11 shows the resulting scenario after the startup of the fourth domain. This scenario is different from the previous scenarios and there are substantially more message exchanges. As in the other scenarios activated by remote domains, the Collector service receives an entry message to update its environment description set. The message without envelope (only message content) is similar to the one shown in C.6.2. The content of this message is the *entry* item with its typical fields. The values are specific for the fourth domain, so it has *remote-bus-4* as *source* (line 2), *Chuck* as *subject* (line 3), *eduroam* as *resource* (line 4) and *connect* as *action* (line 5). This content is sent again to the Analyser service in a message with the same content. Although this message is very similar to those sent in the previous scenarios, now the Complex Event Processor (CEP) of the Analyser service dispatches a pattern recognition event because it has a statement that detects a user connected in more than one remote domain. The Analyser handles the event dispatched by the CEP and sends a message to the Collector service to update the environment with the new information collected from the different entries received in time. This message is as follows:

```
<environment>
  <set>
    <multiplaces>true</multiplaces>
  </set>
  <merge>
    <action-subject>Chuck</action-subject>
```

```
      <action-destination>remote-bus-4, remote-bus-3</action-destination>
    </merge>
    <ttl>5</ttl>
  </environment>
```

This message indicates to the Collector service that it has to set a new environment description entry called *multiplaces* with the value *true*. If that entry already exists, the Collector overwrites it. Moreover, the message urges the Collector to merge two more entries into the environment description set. The first item is the *action-subject* with value *Chuck* and the second is *action-destination* with value *remote-bus-4, remote-bus-3*. If the environment description set does not have these items, the Collector will add them as new. Otherwise, if they already exist, the Collector will merge their values into the existing items. Finally, the message indicates that these items should have a time-to-live of five messages, which indicates that these values should be discarded after sending them to the Decider more than five times without being refreshed. All these environment updates requested to the Collector are prompted by the CEP statement, so they can be configured during the system construction.

After processing the message described above and performing the requested operations, the Collector service builds a new message from the updated environment description set and sends it to the Decider. This new message is as follows:

```
  <environment>
    <source>remote-bus-4</source>
    <multiplaces>true</multiplaces>
    <subject>Chuck</subject>
    <action>connect</action>
    <resource>eduroam</resource>
    <action-subject>Chuck</action-subject>
    <action-destination>remote-bus-4, remote-bus-3</action-destination>
  </environment>
```

Note that, although the first items are very similar than those from previous scenarios, there are three new items in the environment, prompted by the Analyser service. The new items are: *multiplaces* with value *true*, *action-subject* with value *Chuck* and *action-destination* with value *remote-bus-4, remote-bus-3*. The remaining items have the same value set from the previous *entry* message received by the Collector. With this message, the Decider builds a policy request, using the method described previously. It sends it to the Policy Manager to assess the correctness of the environment. This time, the Policy Manager does not emit a positive response but a rejection response, so the value of the *decision* field is *Deny*. The Decider does not change its behaviour depending on the *decision* field. It just builds a message using entries from the environment description set and from the policy response. The message it sends to the Changer is similar to the following:

```
  <order>
    <subject>Chuck</subject>
    <resource>eduroam</resource>
    <action>connect</action>
    <decision>Deny</decision>
    <obligation>DISCONNECT</obligation>
    <action-subject>Chuck</action-subject>
    <action-destination>
      remote-bus-4, remote-bus-3
    </action-destination>
  </order>
```

This time, the *action-destination* field of the message received by the Changer is not empty, so it delivers this *order* to the destinations indicated in that field. This order has four important fields and three less important. The important ones are *decision*, *obligation*, *action-subject* and *action-destination*, with the values *Deny*, *DISCONNECT*, *Chuck*, *remote-bus-4, remote-bus-3*, respectively. This means that the environment is unacceptable, so the subject *Chuck* must be disconnected from the domains 3 and 4. The other less important fields are informative. They describe the rest of the environment and can be useful to correct it. The *order* message is finally sent to the indicated remote domains, so the Changer sends the same message as shown above. The only difference between the message sent to domain 3 and the message sent to domain 4 is the namespace used in the envelope to match the destination service namespace (*xmlns:msg*). The messages are received by the Command Performer service instantiated in remote domains. This service is responsible for performing the obligation that is set in the corresponding field of the *order* message. The Command Performer can use the information in the other fields if it is necessary. Since eduroam services were not used in this test, this implementation only sends an entry to the system to indicate the message was received. If it were actually connected to eduroam, the Command Performer service would transform the message, using utilities provided by the ESB framework (such as routing, filtering and binding components), and send it to the corresponding network element that performs the indicated obligation in the real equipment.

References

[AC]	*The Vision of Autonomic Computing* Jeffrey O. Kephart and David M. Chess IEEE Computer, 36(1):41–50, 2003
[ACOV]	*Autonomic Computing: An Overview* Manish Parashar and Salim Hariri Proceedings of the International Workshop on Unconventional Programming Paradigms, pages 257–269, Springer, 2004
[AUTOBAHN]	http://www.geant.net/Events/ICT2010/Pages/AutoBAHNAnOverview.aspx http://www.geant2.net/server/show/nav.756
[BPMN]	http://www.bpmn.org/
[CLARIN]	http://www.clarin.eu
[CSA]	*GEANT3 Project Deliverable DJ3.3.1: Composable Network Services use cases* D. Lopez and I. Thomson, (January 2010) http://www.geant.net/Media_Centre/Media_Library/Media%20Library/GN3-09-198-DJ3_3_1_Composable_Network_Services_use_cases.pdf
[DAME]	http://dame.inf.um.es/
[DJ2.1.1].	*Information schemas and workflows for multi-domain control and management functions* Note: awaiting publication.
[eduGAIN]	http://www.edugain.org/
[EDUROAM]	http://www.eduroam.org/
[ESB]	http://searchsoa.techtarget.com/tutorial/ESB-Tutorial *Enterprise Service Bus* Chappell, D. (2004, June), O'Reilly, 247 pp *Exploring the Enterprise Service Bus, Part 2: Why the ESB is a fundamental part of SOA* G. Flurry and R. Reinitz (2007) http://www.ibm.com/developerworks/webservices/library/ar-esbpat2/
[ESPER]	http://esper.codehaus.org/
[FI]	*Future internet = content + services +management* J. Schonwalder, M. Fouquet, G. Rodosek and I. Hochstatter IEEE Communications Magazine, 47(7):27–33, 2009
[FOAF]	http://www.foaf-project.org/
[FTICKS]	http://monitor.eduroam.org/f-ticks/
IGTF	http://www.igtf.org
[IPSPHERE]	http://www.tmforum.org/ipsphere
[JSON]	www.json.org/
[KERBEROS]	http://www.kerberos.org

[LINKEDDATA]	http://linkeddata.org/
[NGN]	http://www.itu.int/en/ITU-T/gsi/ngn/
[ODE]	http://www.ode.apache.org/
[OASIS]	http://www.oasis-open.org/
[OGSA]	http://www.globus.org/ogsa/
[OSAMI]	http://www.osami-commons.org/
[OSGI]	http://www.osgi.org/Main/HomePage
[OSSIMM]	http://www.opengroup.org/soa/source-book/osimm_summary/ https://www.opengroup.org/projects/osimm/uploads/40/17990/OSIMM_v0.3a.pdf/
[OWL]	http://www.w3.org/TR/owl-features/
[PASITO]	http://www.rediris.es/proyectos/pasito/index.html.en
[PERFSONAR]	http://www.geant.net/Services/NetworkPerformanceServices/Pages/perfSONARMDM.aspx]
[PUBSUBHUBBUB]	http://code.google.com/p/pubsubhubbub/
[RDF]	http://www.w3.org/TR/rdf-schema/
[REST]	REST services operate on URLs, which may respond with XML messages as well as other formats. JavaScript Object Notation (JSON) REST: L. Richardson and S. Ruby. (May 2007). RESTful Web Services. O'Reilly Media, Inc. ISBN-10:0596529260
[SAML]	http://saml.xml.org/
[SERVICEMIX]	http://servicemix.apache.org/home.html Apache ServiceMix ESB 3.0 to FUSE ESB 4.2
[SEWSR]	*Semantically Enabling Web Service Repositories* http://sweet-dev.open.ac.uk/war/Papers/mmaRepositoriesReport.pdf
[SOA]	http://www.oasis-open.org/committees/tc_cat.php?cat=soa http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N. and Weerawarana, S. (2002). Unravelling the Web services web: an introduction to SOAP, WSDL and UDDI. IEEE Internet Computing
[SOAP]	http://www.w3.org/TR/soap/ http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
[SOMO]	*Self-organising Management Overlays for Future Internet Services* Lawrence Cheng, Alex Galis, Bertrand Mathieu, Kerry Jean, Roel Ocampo et al. Proceedings of the 3rd IEEE International Workshop on Modelling Autonomic Communications Environments, pages 74–89. Springer-Verlag, 2008
[TAVERNA]	http://www.taverna.org.uk/
[TMF-SDF]	http://www.tmforum.org/browse.aspx
[TMF SDF]	http://www.tmforum.org/ManagementWorld2008/SDFOverview/5036/Home.html
[USDL	http://www.w3.org/2005/Incubator/usdl/ http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1530890
[VCLOUD]	*Harnessing Cloud Technologies for a Virtualized Distributed Computing Infrastructure* Alexandre di Costanzo, Marcos Dias de Assunção and Rajkumar Buyya IEEE Internet Computing, 13(5):24–33, 2009
[VIRSYSMNG]	Towards a Service Management System in Virtualized Infrastructures Roman Belter Proceedings of the 2008 IEEE International Conference on Services Computing, pp 47–51
[W3C]	World Wide Web Consortium www.w3.org

[WADL]	http://www.w3.org/Submission/wadl/
	http://wadl.java.net/
[WS-Addressing]	http://www.w3.org/Submission/ws-addressing/
[WSDL]	http://www.w3.org/TR/wsd/
	Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S. (2002). Unravelling the Web services web: an introduction to SOAP, WSDL and UDDI. IEEE Internet Computing
[WSS]	http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
	Web Services Security: SOAP Message Security 1.1 (WS-Security 2004) OASIS Standard Specification, 1 February 2006
	http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
[WSP]	*Web Services Policy 1.5 – Framework*
	W3C Recommendation 04 September 2007
	http://www.w3.org/TR/ws-policy/
[WSBPEL]	http://www.oasis-open.org/committees/wsbpel/
	http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
[WST]	*WS-Trust 1.3. OASIS Standard*. 19 March 2007
	http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html
	http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html
[X509]	http://www.itu.int/rec/T-REC-X.509/en
[XACML]	http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml
[XACMLLIGHT]	http://xacmllight.sourceforge.net/

Glossary

AC	Autonomic Computing
ANI	Application Network Interface
API	Application Programming Interface
AS	Authorisation Service
AutoBAHN	Automated Bandwidth Allocation across Heterogeneous Networks
BC	Binding Components
BoD	Bandwidth-on-Demand
BPEL	Business Process Execution Language
BPMN	Business Process Modelling Notation
CEP	Complex Event Processor
CLARIN	Common Language Resources and Technology Infrastructure project
CLMP	Command Line Measurement Point
CSA	Composable Service Architecture
CUI	Chargeable-User-Identity
CVS	Concurrent Versions System
DAMe	GN2 Project
	Deploying Authorisation Mechanisms for Federated Services in the eduroam Architecture
DNS	Domain Name Service
e.g.	For example…
eduGAIN	AAI confederation created in GN2 JRA5
	for the purpose of interconnecting a set of national and community-wide AAI federations
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
FI	Future of Internet (FI)
FOAF	Friend Of A Friend
FQN	Fully Qualified Name
GEMBus	GÉANT Multi-domain Bus
GLOS	GEMBus Logging Services
GMI	GEMBus Messaging Infrastructure
GRI	Global Reservation ID
gzip	GNU zip (compression utility)
HTTP	Hypertext Transfer Protocol
ID	Identity

Identity Federation	Federated AAI containing multiple IdPs trusted by the members of the federation
IdP	Identity Provider
i.e.	In other words…
IGTF	International Grid Trust Federation
IP	Internet Protocol
IPSphere	TM Forum's framework for rapid service delivery
ISS	Infrastructure Support Services
ITU-T	International Telecommunication Union, Telecommunication Standardisation Sector
JBI	Java Business Integration
JSON	JavaScript Object Notation
Kerberos	Network authentication protocol
MAC	Message Authentication Code
MOM	Message-oriented Middleware
NGN	Next Generation Network
NMR	Normalised Message Router
NREN	National Research and Education Network
OASIS	Organisation for Advancement of Structured Information Standards
ODE	(Apache) Orchestration Director Engine
OGF	Open Grid Forum
OGSA	Open Grid Services Architecture
OSAMI	Open Source Ambient Intelligence
OSE	Open Service Environment
OSGi	Open Services Gateway Initiative (now OSGi Alliance)
OSSIMM	Open Group Services Integration Maturity Model
OWL	(Web) Ontology Language
PASITO	Telecommunications Service Analysis Platform
	Experimental infrastructure for services and protocols provided by RedIRIS
perfSONAR	PERFormance Service Oriented Network monitoring Architecture
QoS	Quality of Service
RADIUS	Remote Authentication Dial-In User Service (IETF standard)
RDBM	Relational Database Management System
RDF	Resource Description Framework
RDL	Resource Description Language
REST	Representational State Transfer
RRDMA	Measurement Archive
RST	Request Security Token
RSTR	Request Security Token Response
SAML	Security Assertion Markup Language (OASIS standard)
SDF	Service Delivery Framework
SE	Service Engine
SeT	Session Token
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
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

SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
STS	Security Token Service
SU	Service Units
TLS	Transport Layer Security
TMF	TeleManagement Forum
TTS	ticket translation service
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
USDL	Universal Service Description Language
VLAN	Virtual Local Area Network
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WMS	Workflow Management System
WS	Web Services
WSA	Web Services Architecture
WS-BPEL	Web Services Business Process Execution Language (BPEL)
WSDL	Web Services Description Language
WSP	Web Services Policy
WSS	Web Services Security
WST	WS-Trust
X.509	ITU-T computer networking standard covering electronic directory services, digital certificates
XML	eXtensible Mark-Up Language
XSD	XML Schema Definition