INTEGRATION FRAMEWORK IMPLEMENTATION

Project Title	RESOLUTE
Project number	653460
Deliverable number	4.5
Version	16
State	FIRST RELEASE
Confidentially Level	PU
WP contributing to the Deliverable	4
Contractual Date of Delivery	M20 (31/12/2017)
Finally approved by coordinator	01-03-2017
Actual Date of Delivery	01-03-2017
Authors	Andrea Grifoni, Cristiano Costantini, Stefano Cigheri
Email	andrea.grifoni@thalesgroup.com, Cristiano.costantini@thalesgroup.com, stefano.cigheri@thalesgroup.com
Affiliation	THALIT
Contributors	All Partners

funded by the Horizon 2020
Framework Programme of the European Union

EXECUTIVE SUMMARY

This document introduces the RESOLUTE Integration Framework and the RESOLUTE Data Model used to share information inside the RESOLUTE architecture.

This document introduces the RESOLUTE Integration Framework and the RESOLUTE Data Model used to share information inside the RESOLUTE architecture. The structure of the document is the following:

Section 1 gives a brief introduction to the RESOLUTE integration framework.

Section 2 explains the problem of integration of multiple heterogeneous systems in a distributed environment, then presents the Enterprise Integration Patterns methodology.

Section 3 presents a general introduction to Web Services as the chosen technology to implement the RESOLUTE Integration Framework.

Section 4 introduces ServiceMix, an open source Enterprise Service Bus from Apache based on Karaf, Camel, CXF and ActiveMQ; Service Mix has been chosen as the RESOLUTE message router to allow the exchange of RESOLUTE's Canonical Data Model messages between the different RESOLUTE Systems.

Section 5 presents the Canonical Data Model and the main communication interfaces designed for the RESOLUTE project. This section describes in details all the steps that have been followed to obtain the Data Model (using TOGAF and NAF data views shared and filled by all partners) able to support the data produced by all RESOLUTE systems. The Canonical Data Model and its interface have been implemented using WSDL and XML Schema technologies.

Section 6 details specific aspects of the ESB implementation, in particular the COMET Message Exchange Pattern and the approach chosen to identify events and data produced by different RESOLUTE systems.

Finally, at the end of the document, an Annex presents the interfaces and data schemas produced.

PROJECT CONTEXT

Workpackage	WP4: Platform back-end
Task	Task 4.5 Integration framework
Dependencies	RESOLUTE reference architecture defined in D4.1.

Contributors and Reviewers

Contributors	Reviewers
THALIT, All Partners	CERTH, CMR

Version History

Version	Date	Authors	Sections Affected
1.0	28/02/2017	A. Grifoni (THALIT)	First Release
1.6	01-03-2017	P. Nesi (UNIFi)	closure

Copyright Statement – Restricted Content

Table of Contents

List of Figures

List of Tables

Abbreviations list

Abbreviation	Explanation
API	Application Programming Interface
CDM	Canonical Data Model
DSL	Domain Specific Language
DSS	Decision Support System
EIP	Enterprise Integration Patterns
ESB	Enterprise Service Bus
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
NAF	Nato Architecture Framework
NATO	North Atlantic Treaty Organization
NOV	Nato Operational View
NSV	Nato System View
OSGi	Open Service Gateway Initiative
POJO	Plain Old Java Object
REST	REpresntational State Transfer
SOA	Service Oriented Architecture
TOGAF	The Open Group Architecture Framework
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WSDL	Web Service Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition

1 INTRODUCTION

1.1 Scope

RESOLUTE Integration Framework is an open integration framework based on an Enterprise Service Bus using a web-services implementation methodology. This framework incorporates the features required to implement a Service-Oriented Architecture (SOA) and represents the mechanism that manages access to RESOLUTE applications and services, presenting a single, simple, and consistent interface.

To build the RESOLUTE Integration framework, services and interfaces have been developed to achieve integration and interoperability, according to specification and design outputs described in D4.1 - back-end reference architecture and specifications.

RESOLUTE Integration framework is developed in two iterations: a first prototype, described in this deliverable, implements the data model and the routing services on the bus, and a second, final release, that will include the complete set of defined functionalities.

1.2 Target audience

Audience of this deliverable are Pilot Site Leaders plus technological partners involved in the deployment of the RESOLUTE system in each of the pilot sites.

Besides, the deliverable is written for all stakeholders interested to learn to learn about EIP (Enterprise Integration Patterns) required for the deployment of the RESOLUTE framework as an application of the ERMG defined in the WP3.

2 THE PROBLEM OF INTEGRATION: EIP PATTERNS

In this section "the problem of integration" is presented and it is shown how this problem can be addressed, by using a well know approach based on Enterprise Integration Patterns (EIP).

Today's enterprise applications are complex systems that often incorporate a n-tier architecture with multiple independent applications that can run each one by itself and coordinate in a loosely coupled way.

The typical scenario comprises for several independent systems deployed in a distribute environment, each one producing (or consuming) data in its own data format and communication protocols.

Very often it happens that a system works with data defined according to proprietary data format and communication mechanism, relying on the usage of proprietary protocols. In other cases, different systems use different open standards to exchange data.

The "problem of integration" consists in finding the best approach to allow multiple heterogeneous systems to integrate functionalities and propose them to the final user through a unique entry point: the final user should see and use a unique application even if the offered functionalities come from the integration of several heterogeneous systems. In practice, the integration should be transparent for the final user.

Typically, to realize integration among different systems, it is required to find a mechanism that allows exchanging different types of data between heterogeneous and distributed systems. This problem is difficult to address because nowadays there is a huge variety of systems, proprietary (and not) data formats and communication protocols. To realize a dedicated application that performs data conversion among every possible format and/or system is almost impossible and could be very difficult to maintain.

So, how can the problem of integration be addressed? Fortunately, "Enterprise Integration Patterns" (EIP) can be used. What are EIP? They are a set of methodologies and best practices specifically designed to solve the problem of integration. As the problem of integration especially consists in moving data between systems, these theories explain in big details how to design and develop messaging solutions.

These patterns are presented in a book called "Enterprise Integration Patterns" [EIP] as shown in Figure 1.

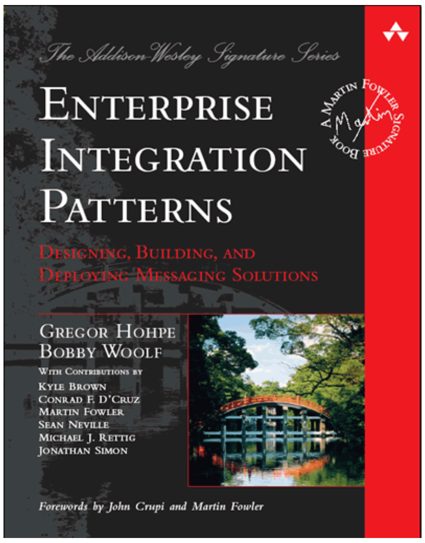

The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
A Martin Fowler Signature Book
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler

Figure 1 – Cover of the Enterprise Integration Patterns book from G. Hohpe and B. Woolf

EIP define the problem of integration in Information Technology as “the art of moving information from one application to another so they can work together”.

This concept is well clarified in Figure 2.

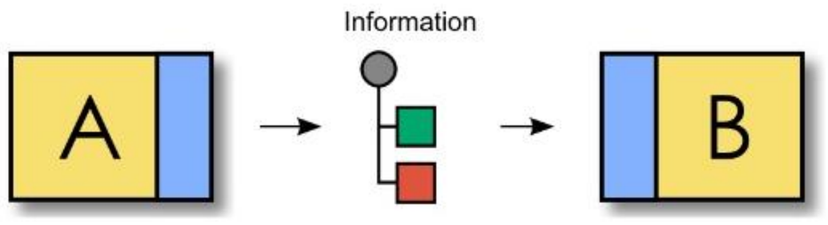


Figure 2 - Moving Information from System A to System B

Basically, let’s suppose we have two different systems, system A and system B; every system works with specific data/information defined in its own data format, in the picture represented with colours green for A and red for B. To allow system B to use the information produced by system A, we have to move this information from A to B and it is required that this information is translated from the data format of A to the data format of B, otherwise B won’t be able understand and use this information. In practice, it is necessary to convert the green information produced by A to the red information used by B.

EIP defines several different patterns and approach for integration. It starts by defining simple systems that require simple approaches to be integrated; then it starts to add complexity to the environment by adding new systems and new technical aspects that require modifying the integration strategy by changing approach or by adding new methodologies. In the end, it presents a complex huge system of systems and shows how it is possible to mix all the presented methodologies to obtain a full integration.

In the next paragraphs, a few relevant patterns are presented that can be used in the context of RESOLUTE project to realize the integration between the various systems provided by all the project partners.

2.1 Message Translator

In many cases, enterprise integration solutions route messages among existing applications, such as legacy systems, packaged applications, home-grown custom applications, or applications operated by external partners. Each of these applications is usually built around a proprietary data model.

Each application may have a slightly different notion of the Customer entity, the attributes that define a Customer and which other entities a Customer is related to. The application’s underlying data model usually drives the design of the physical database schema, an interface file format or a programming interface (API).

In many cases, the integration solution needs to be able to communicate with external parties using the ‘official’ data formats while the internal systems are based on proprietary formats.

How can systems using different data formats communicate with each other using messaging? They can use a special filter, a Message Translator, to translate one data format into another (and vice versa), as shown in Figure 3.

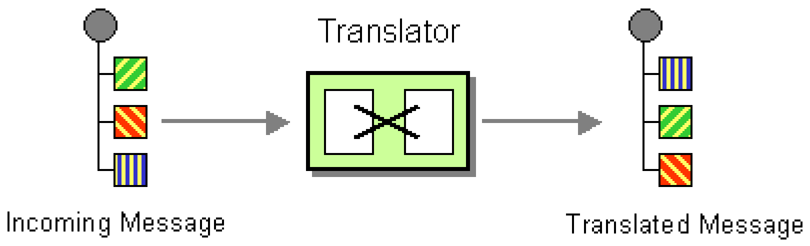


Figure 3 - Message Translator Pattern

The Message translator is a software component that receives messages in a specific data format and takes care to convert/translate this data format into another data format. Usually, the translator is able to do this job in both directions, i.e. to apply transformation from input source to destination source and to apply the reverse transformation from destination to source.

2.2 Message Broker

Very often the problem to realize routing mechanism, among systems to be integrated, arises: how is it possible to decouple the destination of a message from the sender and maintain central control over the flow of messages?

In this case, EIP recommend using a central Message Broker that can receive messages from multiple sources, determine the correct destination and route the message to the correct channel.

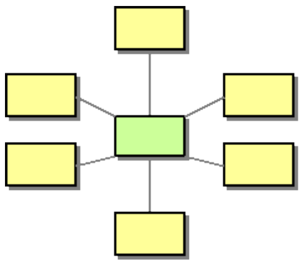


Figure 4 - Message Broker Pattern

Very often in the theory behind this pattern the terminology of “Producer” and “Consumer” and also “Publisher” and “Subscriber” is found.

Usually, “Producer” refers to the system(s) that produce a certain type of data according to a specific data format. “Consumer” instead, refers to the system(s) that uses that type of data for its purposes.

Since the producer systems could produce data in their own proprietary data format while the consumer systems could read data in another data format, a message translator could be required in order to make the message broker pattern work properly.

The words “Publisher” and “Subscriber” refer to, respectively, (a) the system that declares to the broker that it will send/publish information of a specific type, and (b) the system that declares to the broker that it will receive certain type of data.

In conclusion, the broker is the entity that receives all the data messages produced by all the systems and takes care to dispatch these data to the various subscriber systems based on the subscription list.

2.3 Canonical Data Model

The case of the Canonical Data Model is the one that puts together all the other patterns allowing to integrate several distributed heterogeneous applications: even if each application has its own internal data format and protocols, and each one is completely independent from all the others, according to this pattern all the applications are able to exchange data and to work together through messaging.

The basic question that justifies the usage of the "Canonical Data Model" pattern is the following: how can we minimize dependencies when integrating applications that use different data formats?

The answer is: by designing a Canonical Data Model that is independent from any specific application and ask each application to produce and consume messages in this common format.

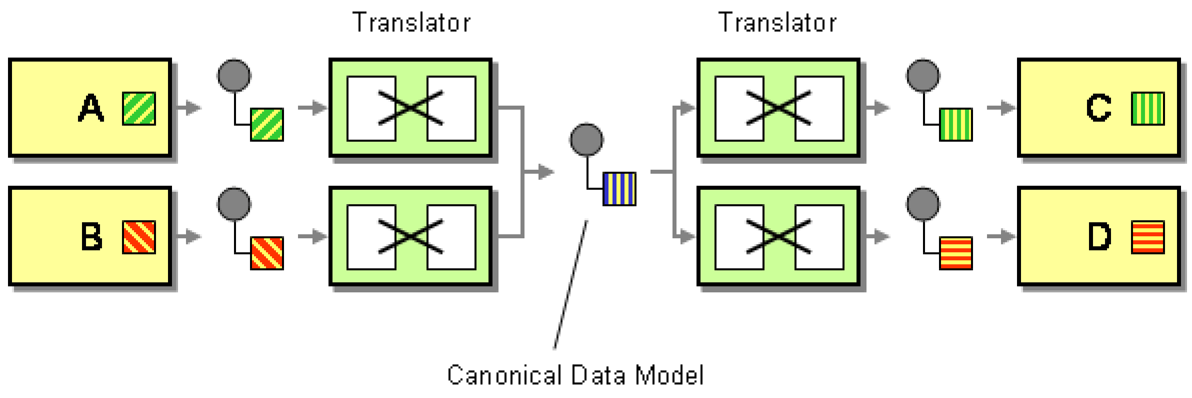


Figure 5 – Canonical Data Model Pattern

As shown in the picture, the Canonical Data Model provides an additional level of indirection between application's individual data formats. If a new application is added to the integration solution only transformation of the Canonical Data Model has to be created, independently on the number of applications that already exist.

This solution makes use of the Message Translator pattern, which is applied to every single system. The output of every translator is a set of data that altogether form the "Canonical Data Model". The Canonical Data Model can, thus, be seen as a comprehensive superset of data: in practice, if system A produces data related to "fruit" and system B produces data related to "meat" and system C produces data related to "fish" and system D produces data related to "vegetables", the Canonical Data Model is a superset of all these types of data: it contains data entities related to "fruit", "meat", "fish" and "vegetables".

By this way system A could send its data related to "fruit" and could ask data related to "vegetables" produced by system D. So, the Canonical Data Model is a pattern that allows bringing together in a unique common point the overall set of data of every single system that takes part in the bigger integration view. This approach is well shown in Figure 6.

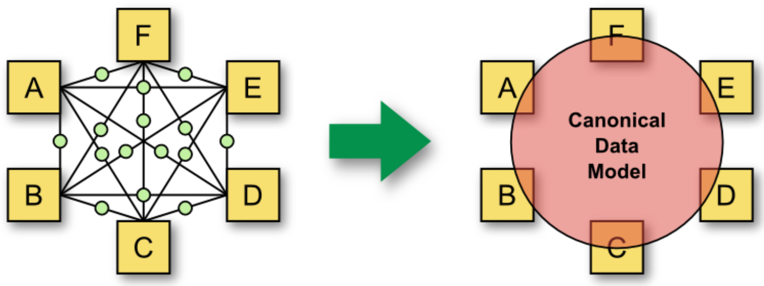


Figure 6 – Canonical Data Model Pattern – CDM Detail

The main advantages of the Canonical Data Model pattern are the following:

- The Canonical Data Model is independent from any specific application;
- The Canonical Data Model requires each application to produce and consume messages in its common format;
- The Canonical Data Model provides an additional level of indirection between application's individual data formats;
- If a new application is added to the integration solution only transformation between the Canonical Data Model has to be created.

This pattern is very often used together with the Message Broker pattern. In practice, producer systems produce some data and publish them to the Broker. The data are first translated from the proprietary data format into the Canonical Data Model representation, then they are inserted in specific messages and, finally, they are sent to the Broker.

The Broker has the list of the subscribed systems, i.e. it knows which are the systems that want to receive certain types of data. So, the broker sends the related messages to the right translators that take care to convert the data from the Canonical Data Model format to the proprietary data format of the receiver systems.

Usually in a real scenario, it happens very often that a producer is also a consumer and by using the Canonical Data Model together with the Message Broker, heterogeneous systems are able to be integrated by moving data from every producer system to every consumer that needs that data.

3 WEB SERVICES

The Canonical Data Model pattern presented in the previous section is a good methodology that can be adopted to realize the integration between several heterogeneous and distributed systems.

As presented in the previous section, this pattern works very well if used together with the Message Broker pattern; basically, the data exchange is based on messages that are sent/received to/by every system that participates to the integration architecture.

This pattern works as expected if there exist a mechanism to exchange messages: one of the most used and well documented way to exchange messages is to use Web Services. The usage of Web Services is also suggested by the EIP theory as a good way to enable messaging communication between several systems.

The next sections will present some theory about Web Services.

3.1 WSDL and XML Schema

The term "Web Services" refers to a collection of standards for communication between software applications which are specifically designed for achieving interoperability between different software platforms. Among several standards related to the Web Services, five of them are particularly important:

- XML (eXtensible Markup Language), together with "XML Schema", is used for describing data models, formats and data types. XML is also used as the basis for specifying other standards, like WSDL.
- HTTP (Hyper Text Transfer Protocol) is the main protocol that is used to implement Web Services over networks.
- WSDL (Web Service Description Language) is used to describe the functionalities offered by a Web Service: it describes service operations, input and output parameters and how the service is bound to a protocol and where the service is deployed.
- SOAP is an application protocol used to exchange Web Services data. It relies on other protocols to negotiate and transmit messages, usually HTTP, and transmit messages in XML format.
- UDDI (Universal Description Discovery and Integration) is an XML catalogue to register and locate Web Services: the register can be interrogated through SOAP messages to find and interact with Web Services.

XML and HTTP are general standards that existed before the formalization of web services while WSDL, SOAP and UDDI are specific to web services.

3.1.1 WSDL

WSDL is the only standard that is mandatory and its use distinguish a real Web Service from other communication technologies based on XML and HTTP. All the other standards are optional: UDDI, for example, is rarely used.

There are two versions of WSDL, version 1.1 and 2.0; although the last version is more synthetic, extensible and solves some issues, it has not gained wide support by development communities and it is recommended to keep working with version 1.1 for wider interoperability.

A WSDL is like a contract that describes the service and is exchanged as an XML file. The use of XML allows processing this contract automatically: this has driven to many Web Services software frameworks, for different operating systems and programming languages. Such frameworks provide automatic code generation from a WSDL file to client proxy's or server stub's classes that can be easily used to invoke or implement the services.

These frameworks take care of the underlying communication stack, making it easy to implement the interoperability between different platforms.

A WSDL could be split in two parts [2], as depicted in figure 7:

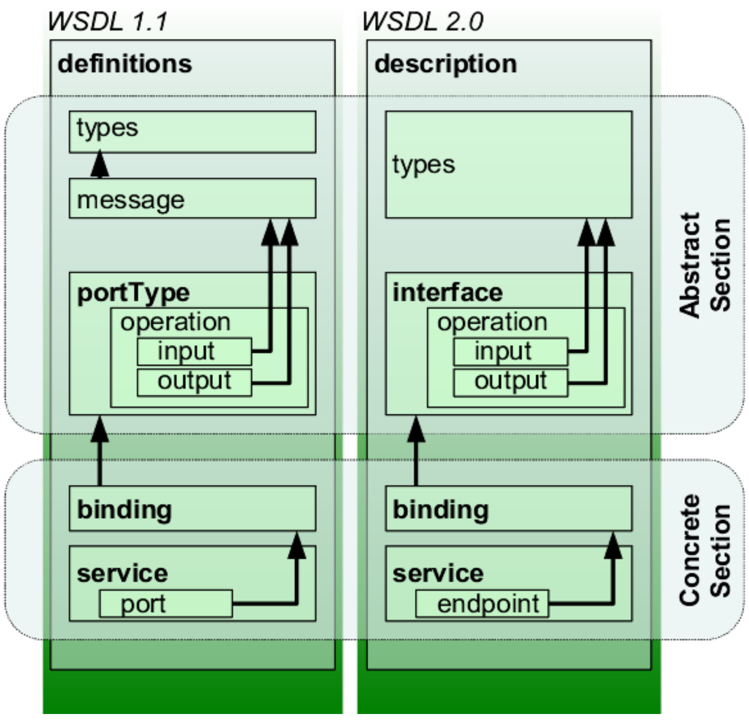


Figure 7 – WSDL structure

The abstract part defines the service's data types and the service's interface. The interface specifies a list of operations together with their input and output data, that define the functionalities provided by the service.

The WSDL can then include one or more "concrete" bindings for the interface. Bindings are a concrete specification of how to exchange the input and output messages with the service and allow implementing the abstract interface of the WSDL. For development purposes, most of Web Services define a SOAP 1.1 over HTTP (sometimes a SOAP 1.2) binding, which instructs the service's client and server implementation to exchange input and output XML messages using SOAP. SOAP 1.1 ensures more interoperability and it is the recommended choice for development.

The WSDL optionally ends with the location of the service network address. When using SOAP over HTTP, the address is an URL which specifies the service's endpoint.

3.1.2 XML Schema

WSDL has a strict connection with XML Schema standard, which is used to define the service data types. An XML Schema is an XML document that describes a set of rules to which a corresponding XML document must conform in order to be considered valid. Specifically, it defines the syntax of the XML document, what kind of XML tags it should contain, in which order these tags have to appear and the format applicable to the tags contents and attributes.

XML Schema includes primitive data types that maps to the main types available in most programming languages (strings, integers, floating point values, date values). An XML Schema complex type is structured as an aggregation of primitive data types or other complex structures.

This standard makes it possible to bind a schema complex type to classes of a programming language and perform automatic "marshalling"[1] to XML and "unmarshalling" of the XML.

Such capability is fundamental for development with Web Services as it allows developers to work with code in their native platform language and get rid of details of dealing with XML parsing and generation.

3.1.3 Web Service's Lifecycle

The layers of the WSDL enable to establish a service's design, development and deployment lifecycle that naturally meets the SOA service lifecycle and allocates clear responsibilities to the team members. The Table 1 [2] illustrates this lifecycle:

Table 1 - WSDL Life Cycle

Layer	WSDL 1.1	WSDL 2.0	Contents clear at	Defined by
Service interface	<types>, <message>, <portType>	<types>, <interface>	Solution design time	Solution managers, business process managers, service portfolio managers …
Service interface with namespaces and other details	<types>, <message>, <portType>	<types>, <interface>	Service design and implementation time	Service designers and/or implementers
Protocol	<binding>	<binding>	Configuration time	Infrastructure (ESB) team
Location	<service>	<service>	Deployment time, runtime	Operators

Such an approach makes it straightforward to implement interoperability. However, these standards are flexible and there is the risk that a poor design might negatively impact the development efforts: at design time, it is important to put some effort in order to preserve an agile and lean implementation of Web Service's code.

3.2 Alternatives to Web Services: REST and JSON

The use of Web Services based on WSDL and XML is the preferred approach to implement the Canonical Data Model EIP in a context like the RESOLUTE project where there are many different partners working to integrate their systems together.

The WSDL and the XML Schema provide in fact a strongly typed standardized reference for the system interface and they ease validation and integration by using automated tools for Web Services.

In today's Web solutions however, other protocols and data formats are more popular, in particular REST and JSON are more common and easier to use from Web or Mobile clients. Despite their popularity, JSON does not provide

[1] *Marshalling* is the process of transforming the memory representation of an object to a data format suitable for storage or transmission, and it is typically used when data must be moved between different parts of a computer program or from one program to another. The opposite, or reverse, of marshalling is called *unmarshalling*.

a standardized way to define a schema for interfaces, so its adoption as a Canonical Data Model would lead to having more communication issues.

To ease interoperability with Web and Mobile clients, it is possible, and it is foreseen as viable option if helpful, to use Apache Camel and implement a Message Translator to transform the XML messages into JSON equivalent messages using the Camel JSON Data Format. In order to make the JSON messages easy to be consumed by these clients, the Apache Camel Jetty or Servlet components can be used to publish an HTTP endpoint where the event will be published in JSON format.

With Apache Camel, already available in ServiceMix, enabling consuming the events and the services of the RESOLUTE Canonical Data Model over HTTP and JSON is just a matter of configuring a couple of routes.

4 EIP IN PRACTICE: ESB

In section 2 presented EIP and in particular the “Canonical Data Model” pattern is presented as the best candidate to realize integration between heterogeneous systems that shall share data produced in different data format types. It has also been said that this pattern works very well if there is a broker to support the messaging infrastructure required to exchange data.

In section 3 web services have been introduced as a good way to support messaging: a web service can be exposed by the broker and can be used by all the systems to send/receive the messages according to the Canonical Data Model representation.

It is clear from the previous sections that all the presented mechanism relies on the presence of a central system called broker, which is the core of the integration platform: this software application in fact is the only one that will receive all the messages sent by every system and will take care to dispatch them to the subscribers.

Moreover, it will take care of other important aspects like the secure delivery of the messages, the persistence of data, the security of transmissions and, in general, every aspect related to inference or logic to be applied on the data in order to enrich, transform them and/or create new ones.

This section covers the last gap that still requires to be explained, i.e. how is it possible to implement the broker? Very often the term broker can be replaced by “Enterprise Service Bus” (ESB), which is a software application that implements a communication system between mutually interacting software applications in a service-oriented architecture (SOA). Every application that uses the ESB can behave as server or client in turns, i.e. it can be both a producer and a consumer of data. ESB promotes agility and flexibility with regard to high protocol-level communication between applications.

More in detail:

- An ESB is the **infrastructure** of SOA;
- The ESB purpose is to provide **interoperability** combined with some additional services such as security, monitoring, inference, persistence etc.
- The ESB has several responsibilities such as:
- Providing connectivity
- Data transformation
- (Intelligent) routing
- Dealing with security
- Dealing with reliability
- Service management
- Monitoring and logging

There are several implementations of ESB that can be used to solve integration problems. Some are proprietary solutions, other ones are free licensed. Almost every implementation provides integration tools that can be used to implement EIP.

The RESOLUTE ESB in particular will be implemented using Apache ServiceMix, that is a free ESB produced by the open source community of the Apache Foundation. In the next subsections we will present the ServiceMix platform and its technological sub-components.

4.1 ServiceMix

Apache ServiceMix is a flexible, open-source integration container that unifies the features and functionalities of Apache Karaf, Apache Camel, Apache CXF and Apache ActiveMQ into a powerful runtime platform that can be used to build integrations solutions.

ServiceMix is a developer oriented ESB: integration is implemented with (little) Java coding and configuration of XML files.

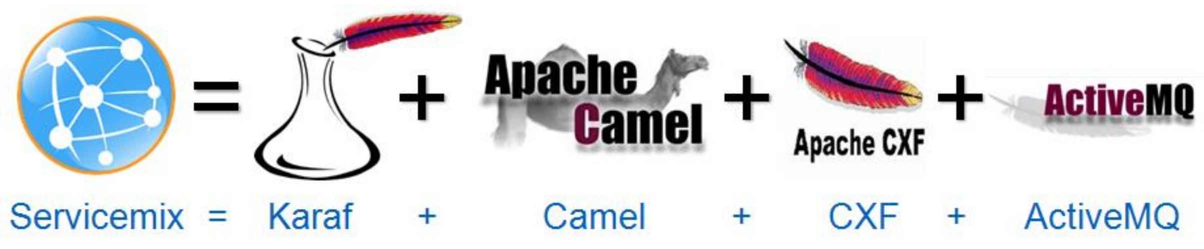


Figure 8 – ServiceMix

4.1.1 Apache Karaf

Apache Karaf is a small OSGI based server runtime which provides a lightweight container onto which several components and applications can be deployed.

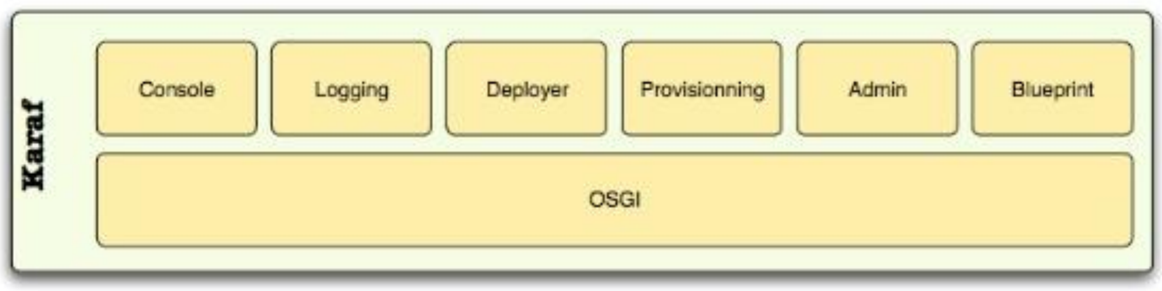


Figure 9 – Karaf Container Structure

In particular, Karaf can be scaled from a very lightweight container to a fully featured enterprise service: it's a very flexible and extensible container, covering all the major needs. These are some of the main features offered by Karaf:

- Complete Console: Karaf provides a complete Unix-like console where it is possible to completely manage the container by issuing dedicated commands.
- Dynamic Configuration: Karaf provides a set of commands focused on managing its own configuration. All configuration files are centralized: any change in a configuration file is noticed and reloaded.
- Advanced Logging System: Karaf supports all the popular logging frameworks (slf4j, log4j, etc…). Whichever logging framework is used, Karaf centralizes the configuration in one file.
- Provisioning: Karaf supports a large set of URLs where it is possible to install applications (Maven repository, HTTP, file, etc…). It also provides the concept of "Feature" which is a way to describe the application.
- Management: Karaf is an enterprise-ready container, providing many management indicators and operations via JMX.

- Remote: Karaf embeds an SSH server that allows to use the console remotely. The management layer is also accessible remotely.
- Security: Karaf provides a complete security framework (based on JAAS), and provides a RBAC (Role-Based Access Control) mechanism for console and JMX access.
- Instances: multiple instances of Karaf can be managed directly from a main instance (root).

4.1.2 Apache Camel

Apache Camel is an open source Java framework that can be used to take care of messaging, routing and Enterprise Integration Patterns realization.

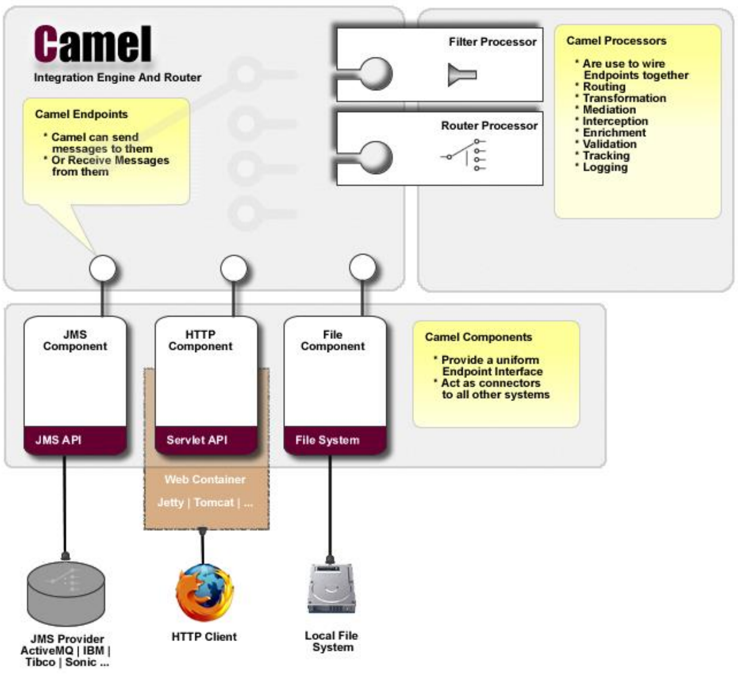


Figure 10 – Camel

Camel provides:

- concrete implementations of all the widely used Enterprise Integration Patterns (EIPs);
- connectivity to a great variety of transports and APIs;
- easy to use Domain Specific Languages (DSLs) to wire EIPs and transports together.

Camel uses URIs to work directly with any kind of Transport or messaging model such as HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF, as well as pluggable Components and Data Format options.

Camel is a small library with minimal dependencies for easy embedding in any Java application.

Camel provides support for Bean Binding and seamless integration with popular frameworks such as CDI, Spring, Blueprint and Guice. Camel also has extensive support for unit testing routes.

4.1.3 Apache CXF

Apache CXF is an open source services framework that helps to build and develop services using frontend programming APIs, like JAX-WS and JAX-RS. These services can speak a variety of protocols such as SOAP, XML/HTTP, RESTful HTTP, or CORBA and work over a variety of transports such as HTTP, JMS or JBI.

In particular, in RESOLUTE context, the CXF framework will be used to develop a Web Service with SOAP/WSDL and expose its endpoint over HTTP.

This service will be used by RESOLUTE Adapters to send/receive data in Canonical Data Model representation format.

4.1.4 Apache ActiveMQ

Apache ActiveMQ is one of the most popular and powerful open source message broker written in Java together with a full Java Message Service client; it is an implementation of JMS, a Java Message Oriented API for sending messages between two or more clients.

ActiveMQ supports many Cross Language Clients and Protocols, comes with easy to use Enterprise Integration Patterns and many advanced features while fully supporting JMS and J2EE.

ActiveMQ employs several modes for high availability and is celebrated for its flexibility in configuration: it supports a large number of transport protocols, including MQTT, REST, and WebSockets.

5 RESOLUTE CDM AND WSDL

5.1 TOGAF and NAF

In order to manage the complexity of the Integration in the RESOLUTE Architecture, it has been chosen to adopt an **Architecture Framework** approach to analyze and design the integration aspect between the system and the stakeholders involved. This approach ensures that the architecture design is taking into account the needs of the stakeholder when implementing the integration among the different systems.

For this purpose, adopted a hybrid approach based on TOGAF (The Open Group Architecture Framework) version 9 [TOGAF] and on NATO Architecture Framework (NAF) version 3 [NAF] has been adopted.

Even if the full approach is complex, only what has been needed to solve integration doubts has been used in RESOLUTE. For reference, TOGAF defines a process to design complex architectures based on 9 iterative phases.

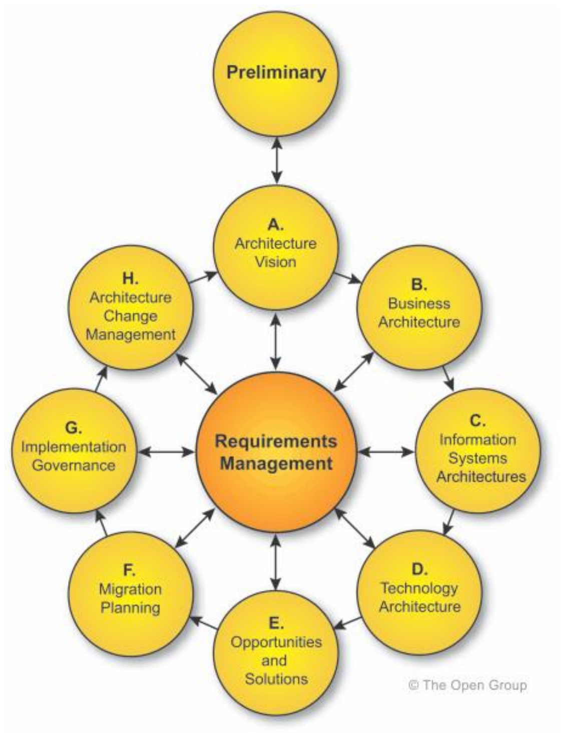

Preliminary
A. Architecture Vision
B. Business Architecture
C. Information Systems Architectures
D. Technology Architecture
E. Opportunities and Solutions
F. Migration Planning
G. Implementation Governance
H. Architecture Change Management
Requirements Management
© The Open Group

Figure 11 - The TOGAF process

On the other hand, NAF defines many sets of views that depict aspects of the architecture that ease analysis of interoperability issues, both at operational and at system level.

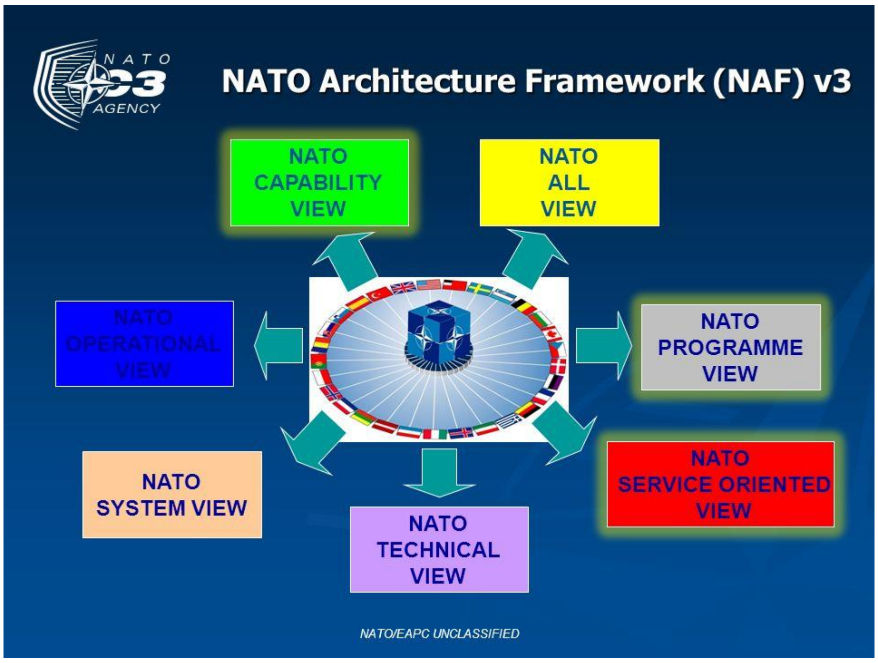


Figure 12 - Categories of Views from NAF v3

To keep the procedure simple, within RESOLUTE only a minimal set of views and phases has been adopted, in order to solve the specific integration issues of the project.

In particular, only two phases have been implemented from the TOGAF, and only 5 views of the NAF were required:

- “Business Architecture” phase:
- NOV 2 - Operational Node Connectivity Description
- “Information System Architecture” phase:
- NSV 1 - System Interface Description
- NSV 3 - Systems to Systems Matrix
- NSV 6 - Systems Data Exchange Matrix
- NSV 11 - System Data Model

All views have been implemented with a digital Spreadsheet; no specific enterprise Architecture Design tool has been used.

5.1.1 NOV2 - Operational Node Connectivity Description

5.1.1.1 Development Guidelines

Synthesis of the View definition and its specific implementation in RESOLUTE extracted from NATO Architecture Framework V.3 [3]:

The purpose of the Operational Node Connectivity Description is to illustrate the operational domain’s needs for information exchange in support of operational activities.

It depicts operational nodes with the needs of information exchange between those nodes.

More in detail, operational nodes produce or consume information and may represent an operational realization of capabilities. The connection between the nodes that represent the exchanged information describes the characteristics of the data or information, i.e. its substantive content, format (voice, imagery, text and message format, etc.), throughput requirements, security or classification level, timeliness requirement, and the degree of interoperability required for the exchange.

The information portrayed in the node connectivity models can be used to make decisions about what systems are needed to satisfy the operational needs of an organization or functional area. However, it must be understood that it is the operational needs that are illustrated and not the systems solutions.

Nodes are independent of material considerations; indeed, they exist to fulfil the missions of the enterprise and to perform its tasks and activities (operational processes, procedures, and functions). Use of nodes and need-lines supports analysis and design by separating process modelling and information requirements from the material solutions that support them.

In the spreadsheet of the RESOLUTE view, Operational Nodes are in headers of columns and rows, the needs of information exchange are written in the cells. The information exchanges defined in the spreadsheet are directional, with sender node on the table's rows, and receiver nodes on the columns. The operational nodes shown in an NOV-2 product may be internal nodes to the architecture, or external nodes that communicate with those internal nodes.

5.1.1.2 RESOLUTE Output View

Table 2 - NOV2 Operational Node Connectivity Description

Sender \ Receiver	Car Driver (user of "streets")	Central Decision Maker	Citizen	Evacuation Responsible	Road Traffic Responsible	Tramway Passenger	Tramway Responsible
Car Driver (user of "streets")				Position Profile data Media Content (image, video) Alarm signal Chat messages	Traffic Level Monitoring Position Profile data		
Central Decision Maker				Accept/Decline proposed evacuation/rescue plan.	Request of traffic deviation		Close Tramway Order Suppress Station Order Request for evacuation message display on Tramway Public Display and Announcement

Sender \ Receiver	Car Driver (user of "streets")	Central Decision Maker	Citizen	Evacuation Responsible	Road Traffic Responsible	Tramway Passenger	Tramway Responsible
Citizen				Position Profile data Media Content (image, video) Alarm signal Chat messages	Position Profile data		
Evacuation Responsible	Media Content (image, video), Alarm signal, Chat messages, Evacuation/rescue path	Evacuation Plan Rescue Team Plan	Media Content (image, video), Alarm signal, Chat messages, Evacuation/rescue path		Request for evacuation message display on Road Traffic Public Display Request of traffic deviation		Request for evacuation message display on Tramway Public Display and Announcement Request of station suppression Request to close line
Road Traffic Responsible	Traffic events	Traffic events Proposed actuation strategies		Traffic events Proposed actuation strategies			Proposed actuation strategies
Tramway Passenger				Position Media Content (image, video), Alarm signal, Chat messages			Position Profile data
Tramway Responsible		Tramway Status				Tramway Status Emergency Info	

5.1.2 NSV1 - System Interface Description

5.1.2.1 Development Guidelines

Synthesis of the View definition and its specific implementation in RESOLUTE extracted from NATO Architecture Framework V.3:

The purpose of the System Interface Description is to illustrate which systems collaborate in which way to support the operational domain's information and information exchange needs as defined in the Operational View. NSV-1 links together the NATO Operational View and the NATO System View by depicting which systems and system connections realize which information exchanges.

A system is defined as any organized assembly of resources and procedures united and regulated by interaction or interdependence to accomplish a set of specific functions. The term system in the NATO System View is used to denote software intensive systems. A system's services are accessed through the system's interfaces. In general, an interface is a contract between providers and consumers of (system) services. With regard to software intensive systems, the contract is a declaration of a coherent set of public system functionality.

NSV-1 is represented in RESOLUTE as a Spreadsheet table depicting which systems are used by which operational nodes and we will rely on NSV-3 to depict the System to System communication aspects.

5.1.2.2 RESOLUTE Output View

Table 3 - NSV1 System Interface Description

	CRAMSS Dashboard	Evacuation DSS	Mobility Supervisor - UTM DSS	Network Analysis	RESOLUTE app	Tramway System - UPT DSS
Car Driver (user of "streets")					x	
Central Decision Maker	x			x		
Citizen					x	
Evacuation Responsible	x	x		x		
Road Traffic Responsible	x		x	x		

Tramway Passenger					x	
Tramway Responsible	x					x

5.1.3 NSV3 - Systems to Systems Matrix

5.1.3.1 Development Guidelines

Synthesis of the View definition and it specific implementation in RESOLUTE extracted from NATO Architecture Framework V.3:

The Systems to Systems Matrix provides detail on the interface described in the NSV-1 subview for the architecture, arranged in matrix form.

An NSV-3 product allows a quick overview of the entire interface presented in multiple NSV-1 diagrams. The matrix supports a rapid assessment of potential commonalities and redundancies (or, the lack of redundancies).

NSV-3 is a summary description of the system to system interfaces identified in NSV-1. NSV-3 is similar to an NOV-3 information exchange matrix, with systems listed in the rows and columns of the matrix, and with cells indicating a system interface pair. Many types of interface characteristics can be presented in the cells of NSV-3.

In RESOLUTE view the NSV3 only depicts the connection between systems, data exchanged details is described by NSV6.

5.1.3.2 RESOLUTE Output View

Table 4 - NSV3 Systems to Systems Matrix

Sender \ Receiver	CRAMSS Dashboard	Evacuation DSS	Mobility Supervisor - UTM DSS	Network Analysis	RESOLUTE app	Tramway System - UPT DSS
CRAMSS Dashboard		X				
Evacuation DSS	X		X		X	X
Mobility Supervisor - UTM DSS	X	X				
Network Analysis			X			

RESOLUTE app		X			▨	
Tramway System - UPT DSS	X		X			▨

5.1.4 NSV6 - Systems Data Exchange Matrix

5.1.4.1 Development Guidelines

Synthesis of the View definition and it specific implementation in RESOLUTE extracted from NATO Architecture Framework V.3:

The Systems Data Exchange Matrix specifies the characteristics of the system data exchanged between systems. This product focuses on automated information exchanges (from NOV-3). Non-automated information exchanges, such as verbal orders, are captured in the NATO Operational View only. System data exchanges express the relationship across the three basic architecture entities of the NATO System View (systems, system functions, and system data flows) and focus on the specific aspects of the system data flow and the system data content. These aspects of the system data exchange can be crucial to the operational mission and are critical to understanding the constraints introduced by the implementation.

NSV-6 presents information about system data exchange from NSV-subviews, such as NSV-1, NSV-3 and NSV-4, but with additional data attributes and properties. NSV-6 also traces to the information exchanges detailed in NOV-3 that constitute the automated portion(s) of the NOV-3 information exchanges. The operational characteristics of the NOV-3 information exchanges are enhanced with the corresponding system data attributes and properties. It should be noted that a one-to-one relationship between NOV-3 information exchanges and NSV-6 data exchanges is not to be expected.

NSV-6 is described in tabular format where a single row represents a system data exchange between a pair of systems (from NSV-1). The table should be enhanced with data and data exchange specific attributes, such as periodicity, timeliness, throughput, size, information assurance, and security characteristics.

Ensure that an NSV-6 table is in accordance with NSV-1, NSV-4, and NSV-7 as well as NOV-2 and NOV-3.

In RESOLUTE the NSV6 is implemented as a spreadsheet table. Two extra columns have been added to track the output of the analysis and define if each specific data exchange is appropriated for passing by the ESB and if the data need to be geolocalized.

The production of this view has allowed identifying the data exchanges necessary to the RESOLUTE architecture: output from other views has been used to identify the Sending Nodes, the Receiving Nodes and the data, the resulting rows with unmatched column has been reviewed with the input of the previous views and have been excluded where no matching has been found. Furthermore, data exchanges where the Sending Node and the Receiving Node were the same, has been excluded and the resulting data has not been used for the data model definition as it is necessary only internally to the node.

5.1.4.2 RESOLUTE Output View

Table 5 - NSV6 Systems Data Exchange Matrix

Sending Node	Receiving Node	Data	Type	Periodicity (frequency)	ESB	Geo
CRAMSS Dashboard	Evacuation DSS	Accept/Decline Evacuation/Rescue Plan	Data	on event	Yes	No
CRAMSS Dashboard	Mobility Supervisor - UTM DSS	Request for Emergency messages on public displays	Com mand	on event	Yes	Yes
Evacuation DSS	RESOLUTE app	Chat message	Data	on event	No	No
CRAMSS Dashboard	Tramway System - UPT DSS	Emergency messages on public displays	Data	on event	Yes	No
CRAMSS Dashboard	Tramway System - UPT DSS	Request for Emergency messages on public displays	Com mand	on event	Yes	Yes
CRAMSS Dashboard	Tramway System - UPT DSS	Request of Tramway station suppression	Com mand	on event	Yes	Yes
CRAMSS Dashboard	Tramway System - UPT DSS	Request of Tramway traffic deviation	Com mand	on event	Yes	Yes
CRAMSS Dashboard	Tramway System - UPT DSS	Request to Tramway line close	Com mand	on event	Yes	Yes
CRAMSS Dashboard	Tramway System - UPT DSS	Tramway high-level decisions (suppress station or close)	Com mand	on event	Yes	No
Evacuation DSS	CRAMSS Dashboard	Evacuation Plan	Data	on event	Yes	Yes
Evacuation DSS	CRAMSS Dashboard	Rescue Team Plan	Data	on event	Yes	Yes
Evacuation DSS	Mobility Supervisor - UTM DSS	Emergency messages on public displays	Data	on event	Yes	Yes
Evacuation DSS	Mobility Supervisor - UTM DSS	Evacuation Plan	Data	on event	Yes	Yes
Evacuation DSS	Mobility Supervisor - UTM DSS	Rescue Team Plan	Data	on event	Yes	Yes
Evacuation DSS	Tramway System - UPT DSS	Emergency messages on public displays	Data	on event	Yes	No
Evacuation DSS	Tramway System - UPT DSS	Tramway high-level decisions (suppress station or close)	Com mand	on event	Yes	No
Mobility Supervisor - UTM DSS	CRAMSS Dashboard	Traffic Events	Data	on event	Yes	Yes

Sending Node	Receiving Node	Data	Type	Periodicity (frequency)	ESB	Geo
Mobility Supervisor - UTM DSS	CRAMSS Dashboard	Proposed actuation strategies	Data	on event	Yes	Yes
Mobility Supervisor - UTM DSS	Evacuation DSS	Proposed actuation strategies	Data	on event	Yes	Yes
Mobility Supervisor - UTM DSS	Evacuation DSS	Traffic Events	Data	on event	Yes	Yes
Network analysis	Evacuation DSS	Critical links (route sections in TPL, roads in road network)	Servi ce	on request	Yes	Yes
Network analysis	Evacuation DSS	Critical nodes (stops of TPL, crossroads of road network)	Servi ce	on request	Yes	Yes
Network analysis	Mobility Supervisor - UTM DSS	Critical links (route sections in TPL, roads in road network)	Servi ce	on request	Yes	Yes
Network analysis	Mobility Supervisor - UTM DSS	Critical nodes (stops of TPL, crossroads of road network)	Servi ce	on request	Yes	Yes
RESOLUTE app	Evacuation DSS	Chat message	Data	on event	No	No
RESOLUTE app	Evacuation DSS	Alarms	Data	on event	No	Yes
RESOLUTE app	Evacuation DSS	Mobile Device Location	Data	on event	No	Yes
RESOLUTE app	Evacuation DSS	Mobile Device Logged Profile	Data	on event	No	Yes
Tramway System - UPT DSS	CRAMSS Dashboard	Tramway status	Data	5 minutes' update	Yes	Yes
Tramway System - UPT DSS	CRAMSS Dashboard	Wifi user stats	Data	5 minutes' update	Yes	Yes
Tramway System - UPT DSS	Mobility Supervisor - UTM DSS	Tramway status	Data	5 minutes' update	Yes	Yes

5.1.5 NSV11 - System Data Model

5.1.5.1 Development Guidelines

Synthesis of the View definition and its specific implementation in RESOLUTE extracted from NATO Architecture Framework V.3:

The purpose of a data model is to enable analysis, design and implementation of the data presentation, handling and storage functionality of an information system.

A data model should not be confused with an information model. Although the distinction between the two is not clear, they at least serve different purposes. There is a grey area where information models and data models overlap. A data model is the representation of an information model in a form that is specific to a particular paradigm or theory on the representation, storage and handling of data, often reflecting a certain type of data storage or repository technology.

The NSV11 in RESOLUTE is implemented as a single view where the Logical data model entities are presented in a column as the result of the analysis from NSV6, and the Physical data model, associated to the logical data model into another column, solely links to the entity implemented in the RESOLUTE WSDL used on the ESB.

5.1.5.2 RESOLUTE Output View

Table 6 - NSV11 System Data Model

Data Type	Implementation
Alarms	Alarms
Chat message	Chat
Emergency messages on public displays	Command (with text)
Mobile Device Location	Resource (of type mobile) location
Mobile Device Logged Profile	Profile
Tramway high-level decisions (suppress station or close)	Command
Tramway status	Resource (of type tramway) status
Wifi user stats	Resource (of type Access Point) status
Traffic Events	Traffic Messages
Evacuation Plan	Evacuation Plan
Accept/Decline Evacuation/Rescue Plan	Command
Critical links (route sections in TPL, roads in road network)	Network
Critical nodes (stops of TPL, crossroads of road network)	Network
Proposed actuation strategies	Traffic Messages
Rescue Team Plan	Evacuation Plan
Request for Emergency messages on public displays	Command
Request of Tramway station suppression	Command
Request of Tramway traffic deviation	Command
Request to Tramway line close	Command

5.2 RESOLUTE CDM and XSD

The RESOLUTE Canonical Data Model (CDM) is a reference data model produced in order to achieve interoperability between the main systems integrated in the RESOLUTE project and in the pilots. The aim of this data model is to connect the system over a centralized Enterprise Service Bus (ESB) dedicate to exchange real-time events and avoiding creating a “spaghetti” architecture where each system chaotically talks with the others.

To model the RESOLUTE Canonical Data Model (CDM) all the inputs coming from the TOGAF/NAF views presented in the previous sections were modelled using UML (Unified Modelling Language) notation.

Several entities have been defined, presented here with UML Class Diagrams, one for every relevant type of data to be exchanged. Each entity may have one or more attributes that characterize better its behaviour and usage. For example, an “Alarm” entity has been defined that represents the alarm data to be exchanged between the systems. This entity is characterized by a name, a message, an open and close time and so on. Every field of the entity will be populated by the producer system and will be available for the consumer systems to be used.

The entities are implemented as an XML Schema: XML can be exchanged via a Web Service or other means of communication. A code-first approach has been used for the schema definition, thus the entities have been developed using java programming language and basically they are java POJO (Plain Old Java Object) that contains only fields and setter/getter methods on these fields. Using JaxB annotation, the entities are published as XML Schema and can been exchanged over heterogeneous systems.

The resulting UML class diagrams of the entities are presented here, showing that there are some basic entities that use or depend on other entities. Inheritance, extension and inclusion relations have been used widely in the creation of the data model.

The resulting RESOLUTE CDM consists of some generalist entity and some more specialist entity. The general entities are: Alarm, Resource, Person and Chat. The specific entities cover the concepts of Evacuation, of Traffic information and events, and the area of Network Analysis.

Many entities can be geo-referenced in RESOLUTE project so the concept of Coordinates and Location has been modelled independently, and it is used by entities of Alarm, Resource, by Evacuation nodes and plans, and by Traffic information and events.

Figure 11 shows the UML representation of these main generalist entities.

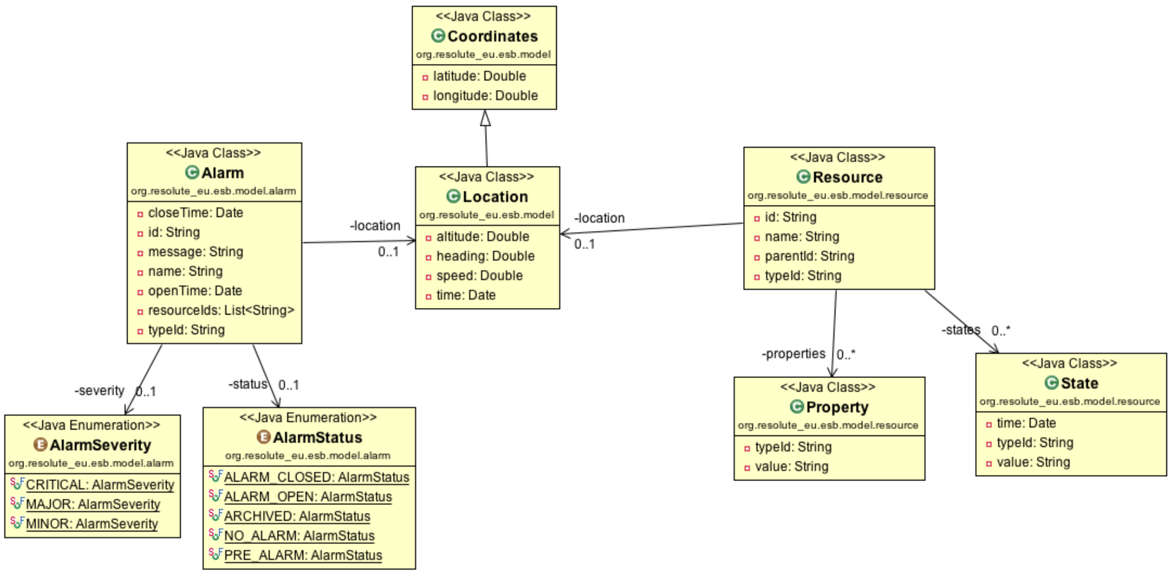


Figure 13 – Alarm and Resource entities CDM representation

The Alarm is an entity that can be used to exchange data related to alarms and alerts: it is defined by a unique identifier, a name, a type, a description message, an opening and closing time and a list of resources related to that alarm. It also has a severity and a status that are basically 2 enumerations of possible values. And, as said before, it may have a location that identifies exactly its real position in the environment.

This alarm could also have a relation towards one or more resources like for example if the alarm report is a Road Accident, it is possible to enrich the alarm with metadata for the adjacent road intersection that may be impacted by this alarm and attach as IDs of these nodes in the list of linked “resources”.

The Resource entity is used to model the concept of general device: it is defined by a unique identifier, a name, a type, a parent identifier (i.e. the id of the resource that can be considered hierarchically higher) and a location. It can also have one or more statuses and properties.

For example, a resource could be a bus that is moving in the street, thus it has its position and status (for example MOVING at time 15:00 and STOPPED at time 15:01), but may have also one or more properties (for example the “model” property with value “Breda AC12” or the “license_plate” property with value “CZ898NF”).

For modelling geo-referencing of information, the Coordinates and Location entities have been created; the Coordinates express the longitude, latitude of an object, in a mostly static approach. The Location entity extends the Coordinates with altitude, heading, speed, the time of the localization in order to make it possible to model the case of a vehicle moving at a certain speed in a precise time, and it can also be said that in that time the vehicle was at specific GPS coordinates and was moving with a certain heading.

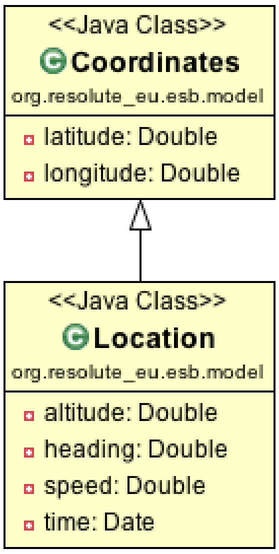


Figure 14 – Coordinates and Location entities

Regarding the profiling and the messaging data we have modelled two basic entities that are Person and Chat.

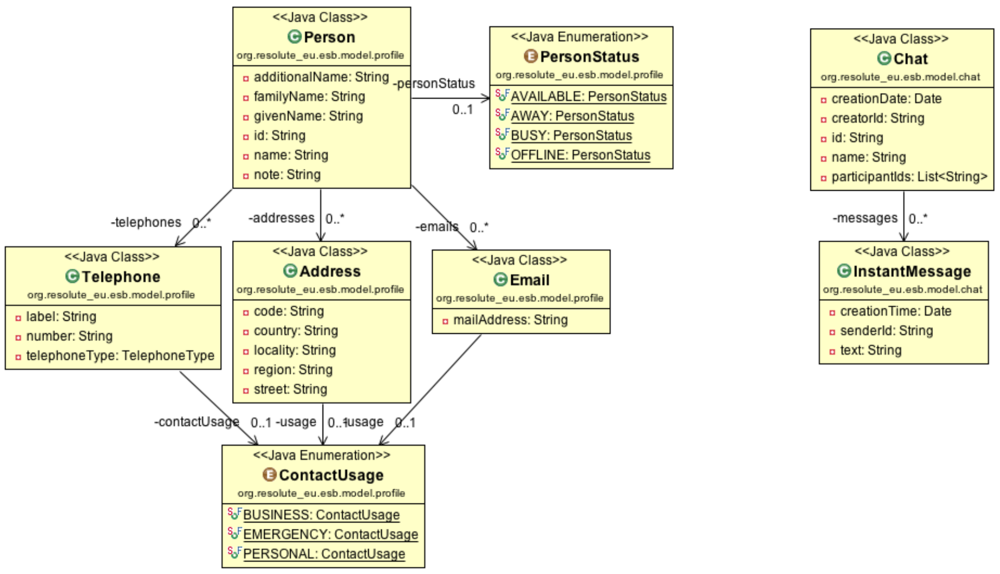


Figure 15 – Person and chat entities

A Person entity contains basically all the typical data for every person so it has some personal data such as the name, the family name, some note relatives to the person. Obviously, a person could have some telephones, some e-mails and also some address and all of these 3 entities could be used for business, emergency or personal purposes. This is the reason why the entities Telephone, Address and Email are contained by the entity Person in the form of list, and every one may have a ContactUsage entity.

For example, the person “Mario Rossi” could have 2 telephones, one for business and one for personal usage, could have also 2 e-mails one for business and one for personal usage; and in the same way he could have associated two addresses, one of its own home (personal usage) and the other one of the place where he works (business).

Every person has also a unique identifier that univocally identifies him/her: this is useful not only to avoid duplicates of the same person in the platform but also to avoid the case of two different people that have the same name/family name.

The Chat entity is used to model the classic message data. A chat is like a room identified by its name and id; there is a creator of the chat a timestamp of the chat and a list of participants: every participant is a person so basically, it is enough to add the list of ids of all persons to identify who are the participants to the chat.

Then, every time that someone sends a message, it is added to the list of messages of the chat. Every message has only three basic fields that are the text of the message, the senderId (also in this case it will be the id of a person) and the creation timestamp of the message.

As an example, let’s suppose there are 3 people called “Mario Rossi”, “Mario Bianchi” and “Sergio Verdi” with ids 1, 2 and 3 respectively; Mario Rossi is the creator of the Chat and all the people participate to the chat; the Chat entity will have the creatorId and participantsIds set in this way:

- creatorId: 1;
- participantIds: 1,2,3;

A new InstantMessage entity will be generated every time a new message is sent by a participant and this entity will be added to the list of messages of the Chat. Every message will have as senderId the id of the participant that has sent the message (i.e. 1 or 2 or 3).

The specific entities of the data model describe the objects to be used to exchange Evacuation, Traffic and Network Analysis information.

The entities used to exchange Evacuation information are the EvacuationPlan and the EvacuationNode:

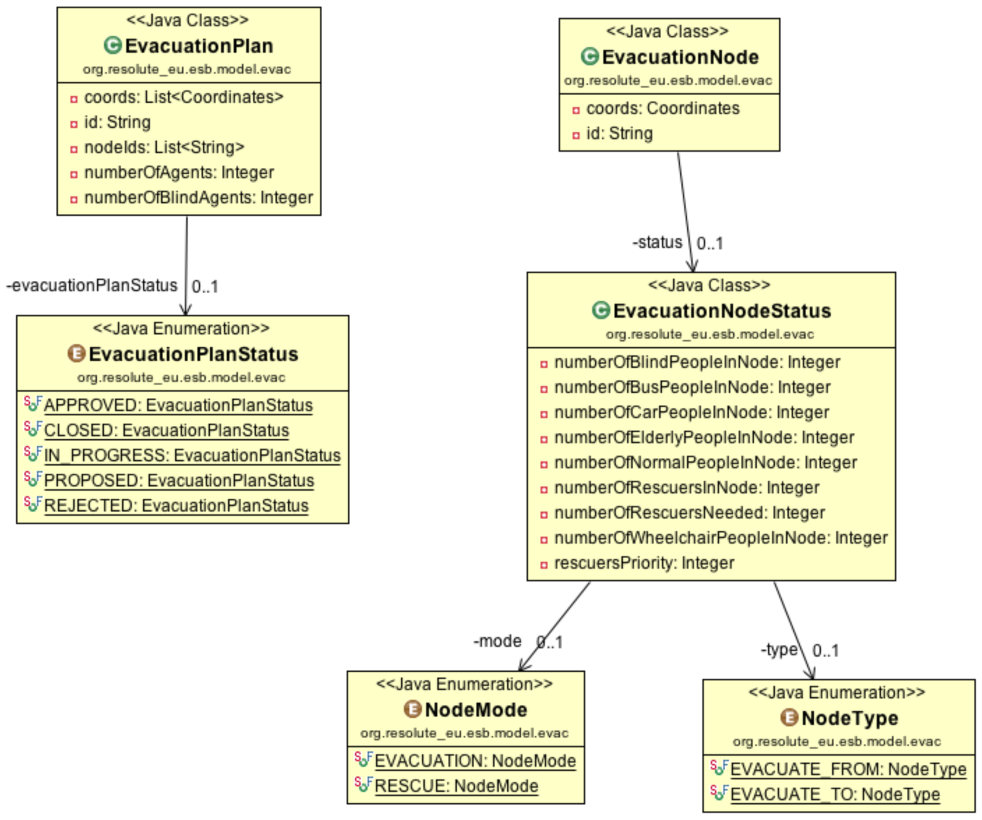

<<Java Class>>
EvacuationPlan
org.resolute_eu.esb.model.evac
coords: List<Coordinates>
id: String
nodeIds: List<String>
numberOfAgents: Integer
numberOfBlindAgents: Integer
-evacuationPlanStatus 0..1
<<Java Enumeration>>
EvacuationPlanStatus
org.resolute_eu.esb.model.evac
APPROVED: EvacuationPlanStatus
CLOSED: EvacuationPlanStatus
IN_PROGRESS: EvacuationPlanStatus
PROPOSED: EvacuationPlanStatus
REJECTED: EvacuationPlanStatus
<<Java Class>>
EvacuationNode
org.resolute_eu.esb.model.evac
coords: Coordinates
id: String
-status 0..1
<<Java Class>>
EvacuationNodeStatus
org.resolute_eu.esb.model.evac
numberOfBlindPeopleInNode: Integer
numberOfBusPeopleInNode: Integer
numberOfCarPeopleInNode: Integer
numberOfElderlyPeopleInNode: Integer
numberOfNormalPeopleInNode: Integer
numberOfRescuersInNode: Integer
numberOfRescuersNeeded: Integer
numberOfWheelchairPeopleInNode: Integer
rescuersPriority: Integer
-mode 0..1
-type 0..1
<<Java Enumeration>>
NodeMode
org.resolute_eu.esb.model.evac
EVACUATION: NodeMode
RESCUE: NodeMode
<<Java Enumeration>>
NodeType
org.resolute_eu.esb.model.evac
EVACUATE_FROM: NodeType
EVACUATE_TO: NodeType

Figure 16 – EvacuationPlan and EvacuationNode entities

The EvacuationPlan models a plan to be executed to in order to perform evacuation from an area. The EvacuationNode models the status of a node (an intersection between roads) that can be in a status that needs to be evacuated or needs a rescue team. Also, a node may be the origin of the evacuation (EVACUATE_FROM) or the destination of the evacuation (EVACUATE_TO) where people must be moved in order to achieve safety.

The entities used to exchange Traffic information are the TrafficInformation and the ActuationStrategy:

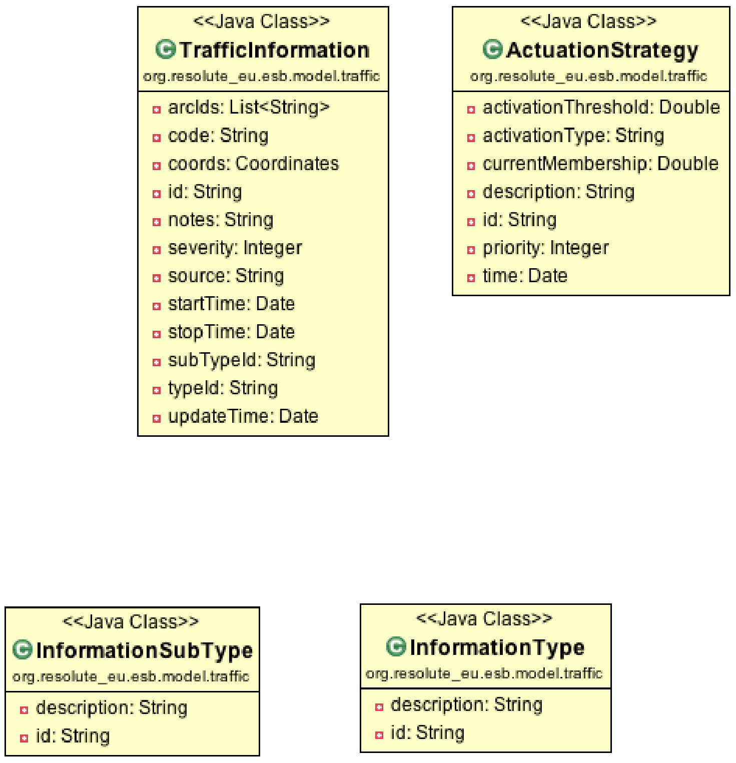


Figure 17 – TrafficInformation and ActuationStrategy entities

The TrafficInformation object is used to model the status of current traffic events that will be displayed on the dashboards. The name TrafficInformation has been chosen instead of the name TrafficEvent in order to avoid ambiguity with the TrafficEvent object used in the WSDL to transport TrafficInformation objects and their updates.

The TrafficInformation has a typeId and a subTypeId fields: these two values refer to objects, InformationType and InformationSubType, which may be exchanged to share the specific descriptions of these fields.

The ActuationStrategy object is used to exchange information about the actual strategies adopted to address the current traffic status, and it enables displaying on the dashboard the active strategies.

The entities used to exchange Network Analysis are the NetworkAnalysis, the EdgeMeasure, the NodeMeasure and the ElementEvaluation:

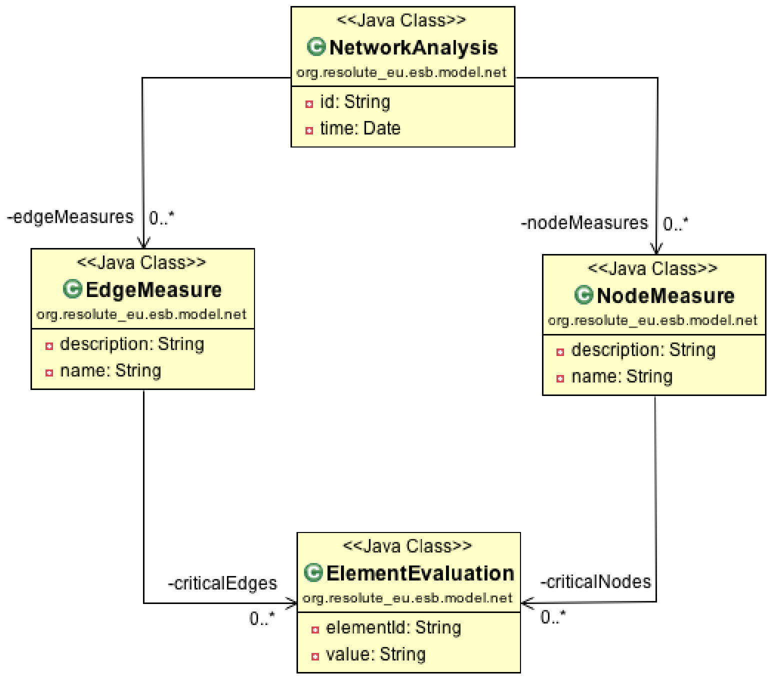


Figure 18 – NetworkAnalysis entities

It is important to underline that all the presented entities contain the minimum set of information required to implement all the defined scenarios and to exchange the complete set of data produced by every system. The CDM can be refined and modified every time that there is the need to add a new type of data to the integration platform or there is the need to modify or remove an existing type of data.

More in general if a new system joins the platform and this system produces new data that can be consumed by other system, the CDM can be modified to satisfy this need. Usually when a platform is up and running for some times and all the systems are working properly, there will be no more changes or the changes will be really minimal.

As said before, all the entities have been modelled using java in the form of POJO. By using the tools presented in section 4 and in particular Apache CXF, it is possible to generate the XML-Schemas (XSD files) to represents these java object in XML notation. In practice the CDM for the RESOLUTE project is a set of XML-Schema files, which define all the rules and naming conventions to be used to create valid messages in the CDM format representation.

This way, every adaptor is able to generate an XML message in the form of the CDM XSD format and fill this message with the relevant data produced by the external system.

A valid XML CDM message is presented in Figure 17 that presents some example data for an alarm.

```xml
<alarmDefinition
    xsi:type="Alarm"
    id="com.thalesgroup.it/alarms/fire/01"
    typeId="org.resolute-eu/alarm_types/fire"
    name="Fire Alarm"
    message="smoke presence in area 1"
    openTime="2017-02-15T14:30:00.000+02:00"
    status="PRE_ALARM"
    severity="MAJOR">
    <location longitude="10.482345" latitude="4.437890" />
    <resourceIds>
        <resourceId>com.thalesgroup.it/resources/smoke_detector/bosh_01</resourceId>
    </resourceIds>
</alarmDefinition>
```

Figure 19 – Alarm example XML message in CDM representation

In the example, all the fields of the alarm have been set with fake data, but the format of the message is syntactically correct: if a real adaptor want to send alarm data generated by the external system, it shall generate an XML message in the same format of the example and simply replace the fake data with the real one.

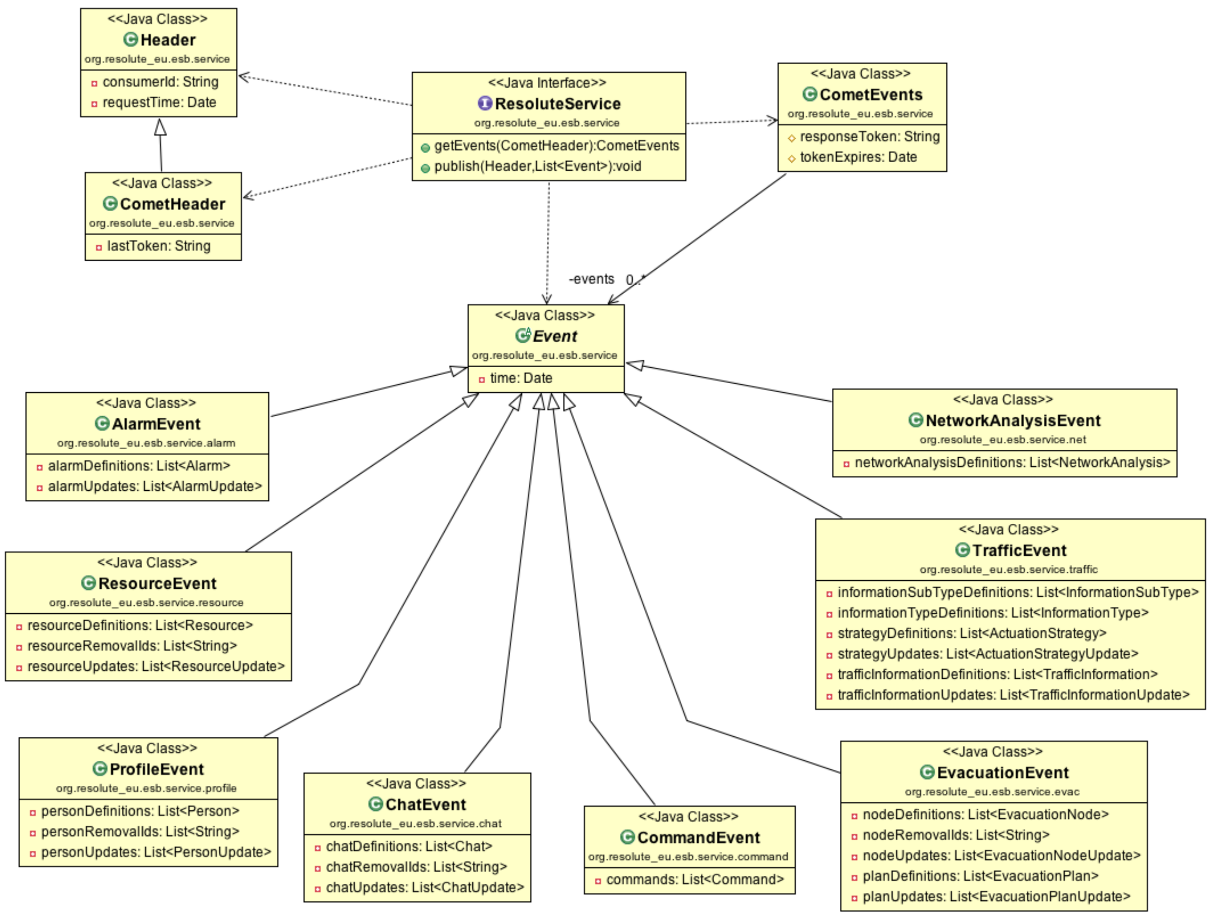
Once the XML message has been created, the adapter can use the interface exposed by the platform web service to send it: the interface design is explained in the following section.

5.3 RESOLUTE WSDL

The previous section has introduced the CDM representation of the data for the RESOLUTE project. The data produced by the systems external to the platform, shall be at first translated in the CDM data format and only then can be sent to the platform. The adaptor is the software component that takes care to do this job.

This section describes the interfaces exposed by the platform web service that can be invoked by the adaptor to send the produced data. The platform exposes also an interface that can be invoked to get the data produced by other systems: by this way every external system can also be a consumer of the information produced by the others (or by the platform itself).

As for the data model, the interfaces have also been modelled in Java, according to UML principles; Figure 18 shows the UML representation of the interfaces.



<<Java Class>>
Header
org.resolute_eu.esb.service
consumerId: String
requestTime: Date
<<Java Interface>>
ResoluteService
org.resolute_eu.esb.service
getEvents(CometHeader):CometEvents
publish(Header,List<Event>):void
<<Java Class>>
CometEvents
org.resolute_eu.esb.service
responseToken: String
tokenExpires: Date
<<Java Class>>
CometHeader
org.resolute_eu.esb.service
lastToken: String
-events 0..*
<<Java Class>>
Event
org.resolute_eu.esb.service
time: Date
<<Java Class>>
AlarmEvent
org.resolute_eu.esb.service.alarm
alarmDefinitions: List<Alarm>
alarmUpdates: List<AlarmUpdate>
<<Java Class>>
NetworkAnalysisEvent
org.resolute_eu.esb.service.net
networkAnalysisDefinitions: List<NetworkAnalysis>
<<Java Class>>
ResourceEvent
org.resolute_eu.esb.service.resource
resourceDefinitions: List<Resource>
resourceRemovalIds: List<String>
resourceUpdates: List<ResourceUpdate>
<<Java Class>>
TrafficEvent
org.resolute_eu.esb.service.traffic
informationSubTypeDefinitions: List<InformationSubType>
informationTypeDefinitions: List<InformationType>
strategyDefinitions: List<ActuationStrategy>
strategyUpdates: List<ActuationStrategyUpdate>
trafficInformationDefinitions: List<TrafficInformation>
trafficInformationUpdates: List<TrafficInformationUpdate>
<<Java Class>>
ProfileEvent
org.resolute_eu.esb.service.profile
personDefinitions: List<Person>
personRemovalIds: List<String>
personUpdates: List<PersonUpdate>
<<Java Class>>
ChatEvent
org.resolute_eu.esb.service.chat
chatDefinitions: List<Chat>
chatRemovalIds: List<String>
chatUpdates: List<ChatUpdate>
<<Java Class>>
CommandEvent
org.resolute_eu.esb.service.command
commands: List<Command>
<<Java Class>>
EvacuationEvent
org.resolute_eu.esb.service.evac
nodeDefinitions: List<EvacuationNode>
nodeRemovalIds: List<String>
nodeUpdates: List<EvacuationNodeUpdate>
planDefinitions: List<EvacuationPlan>
planUpdates: List<EvacuationPlanUpdate>

Figure 20 – RESOLUTE Interfaces CDM representation

The basic entity for the service interfaces is "RESOLUTEService": this entity defines two interfaces, "publish" and "getEvents" that can be used respectively to send a list of events to the platform and to get a list of events from the platform.

The event entity is an abstract entity that has only a timestamp field to be used to set the date and time associate to the creation of the event itself.

This abstract entity can be specialized by concrete realizations, in particular one for every basic data type, i.e. an AlarmEvent for the alarm data, a ResourceEvent for the resource data, a ProfileEvent for the person data and a ChatEvent for the chat data. Every specialized event can be used to contain several different information about the basic data type, such as the creation (called definition) of new data, the modification (called update) of existing data and the deletion (called removal) of existing data.

The basic pattern followed by each specific event is to transport push notifications with the data in the payload of the event. There are 3 types of potential information in the events:

- Definitions: a new object is defined on the system or the object is completely re-defined;
- Updates: lightweight updates for frequently changing fields;
- Removals: lists of IDs of the object removed and no more available in the system.

Figure 19 illustrates the pattern applied to the ResourceEvent

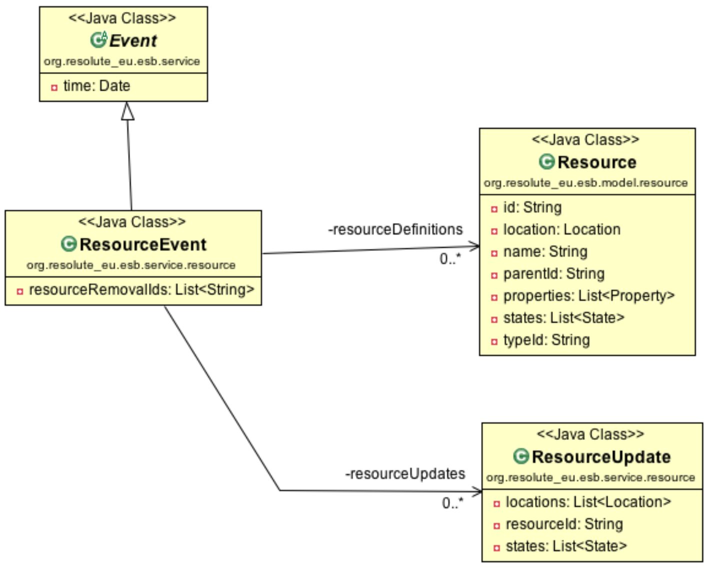


Figure 21 – Example of the event pattern applied to the Resource objects

Every type of information (definition, update or removal) added to an event, is added in the form of a list in order to send multiple data in just one event. For example, it is possible to add to a ResourceEvent a list of new resources, a list of updates of existing resources and a list of removal of existing resources.

The approach about definition, updates and removal is replicated for every event type, except for the alarm because it is not allowed to remove an alarm, but only to create and update it. After its management, the alarm can be closed and archived but it will never be removed from the platform, in order to allow for external systems or platform operators to retrieve the history of past alarms.

The publish interface accepts as input a list of event entities, so it is possible to send for example 2 o 3 alarm events and a resource event and 2 profile events in just one message. Every single event could contain a list of one or more definition, update, removal of its data type. So, it is really up to the producer to decide on the amount of data to send in a single message. It could also be possible to send a single event with just one definition (or update or removal) of a single instance of alarm.

This approach allows the sender to have full granularity on the data sending, as it can aggregate multiple pieces of information or just send single data items. The amount of data, together with the frequency of publishing, depends strongly on how the subsystem works: thanks to this mechanism the platform allows covering every need of the sender.

The publish interface accepts also another entity as input: the Header. This entity is simply used to declare the identity of the data sender. There are in fact 2 fields of particular interest, which are the “consumerId” and the "requestTime". The consumerId is the unique identifier of the sender of the message (for the platform, the sender is first of all a system that consumed the service exposed through the interface), while the requestTime is the timestamp of the interface invocation.

The getEvents interface is used to require to the platform the list of events. Events are sent by producer systems in real time, so platform events continuously grow over time. If a consumer system asks for events at a certain time, the platform will give the events it has at that time; if the same system make a new request after 5 minutes, it shall get only the new events arrived in the last 5 minutes and not also the events that have been already considered before.

To manage this communication mechanism, the platform uses a methodology called Comet [4]; according to this methodology, when a system asks for events, the platform returns the events (if present); if not present, the platform waits for new events to be received and when a new one arrives, the response is being sent. If no events are received in a configurable time interval (usually 30 seconds), the platform returns an empty message, and the consumer can issue a new request. This assures that the consumer will receive for sure some events back if present in the platform, or will wait at maximum 30 seconds and will then be able to ask again for new events. In practice, COMET is an evolution of polling where the waiting time is controlled by the server.

The getEvents interface is implemented to work exactly in this way: to invoke this interface the consumer shall specify as input parameter a CometHeader that extends the basic Header entity but adds the "lastToken" field. This field is a string generated by the server at the first invocation of getEvents from a consumer system. The token is associated to the consumer and when the consumer calls the getEvents, this token should be specified in the CometHeader. At this point the platform knows the identity of the consumer and, based on the token value, it is able to know the timestamp of the last call and which are the new events received from the time of the last request and from the time of the new one.

Based on this information, the platform will return a CometEvents entity that contains the list of the events arrived from the last call up to the new invocation time together with a responseToken. This token that is the "lastToken" will be used by the consumer in the next getEvents invocation. By this way, the communication can continue to go on time by time.

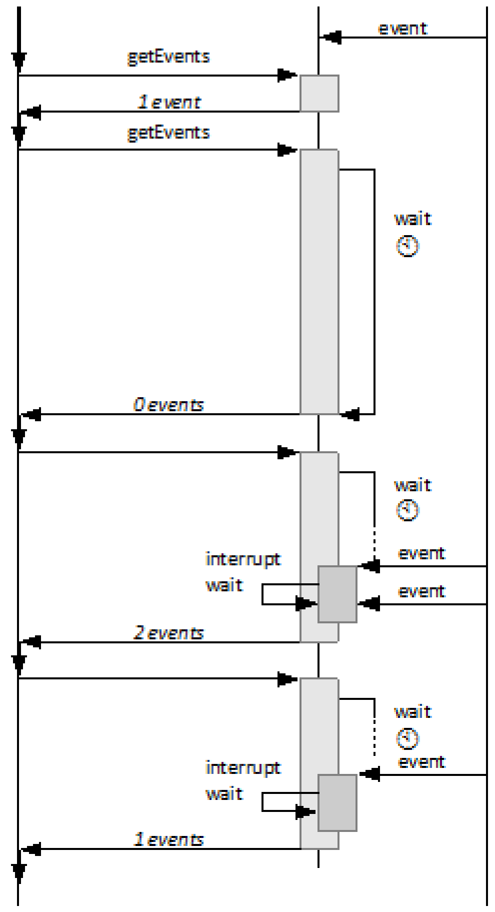


Figure 22 – COMET Message Exchange Pattern

There is another entity that still requires an explanation: the CommandEvent. This entity is an event exactly like the AlarmEvent, ResourceEvent etc, but unlike the other type of events it doesn't contain data produced by external systems, like alarms, resource, etc.

On the contrary, it contains a list of "Command": command is another entity that has been modelled specially to send command/directives to the platform or to external systems. It is a very simple entity: it contains only the list of identifiers of the receivers of the command and the type of the command; it is supposed that all the systems that participates to the integration agree together on the types of command in order to know what everyone is supposed to do when it receives a command of a certain type. There is also a specialization of the command entity that is the TextualCommand that simply add a textual field to be used to specify better some parameters for the command or to add some notes.

Using CXF, the interfaces entities are defined within a WSDL (Web Service Definition Language) file that imports also the XML Schema files generated to define the RESOLUTE CDM.

The Web Service exposed by the platform is created using software tools that parse the WSDL + XSD files and generate the defined interfaces.

The WSDL + XSD files can also be released to the RESOLUTE developers that can use them to generate, inside the adapter, a client application that will connect to the platform Web Service and will invoke the exposed interfaces to publish and get events.

Using CXF all the messages generated by the adaptors can be translated in the XML CDM format representation and encapsulated in a SOAP Envelope. Then the adaptor can send this SOAP Message over HTTP to the Web Service endpoint exposed by the platform.

The platform will receive this SOAP message, will read the content of the SOAP Envelope and will invoke the service implementation of the proper interface (publish or getEvents): the content of the SOAP Envelope (i.e. the XML message) will be passed as input to the interface implementation.

According to the policies and rules defined in the platform (i.e. persistence, dispatching of data, inference on data and so on) the interface implementation software will be able to manage the message properly and compute all the required actions.

6 RESOLUTE ESB IMPLEMENTATION

This chapter presents, in detail, how the RESOLUTE ESB has been implemented.

RESOLUTE architecture relies on open common communication interfaces used to collect data and run control actions on respective actuation system. It is based on a three-layer architecture model:

- Presentation layer
- Mission Critical layer
- Data Management layer

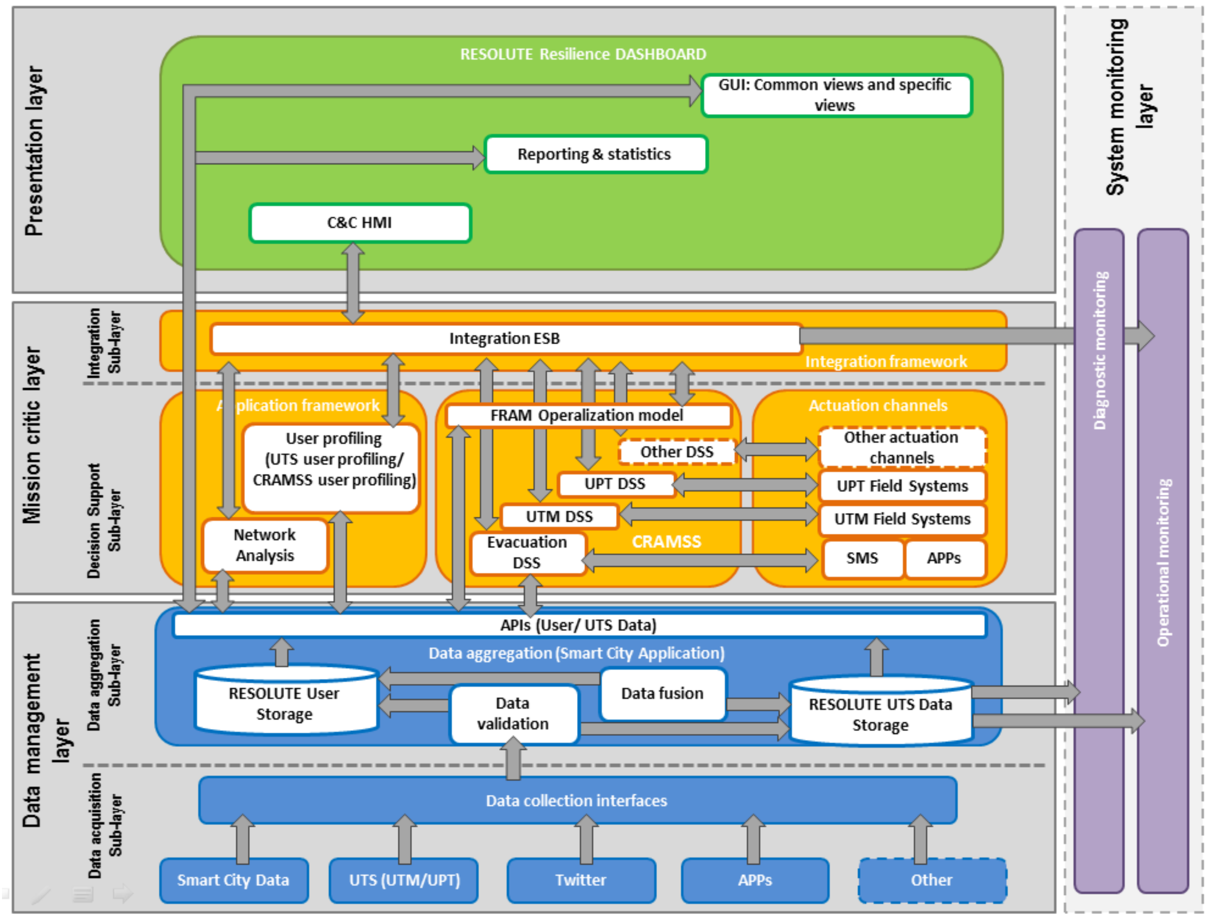


Figure 23 – RESOLUTE Architecture

For all the details about the architecture, refer to Deliverable D4.1 – Back-end reference architecture and specifications [RESOLUTE-D41].

RESOLUTE shall be capable to manage events and alarms in real-time, thus it is necessary to adopt a system that allows publishing and receiving these events. In the previous sections the methodologies and technologies that has brought to the implementation of a web service that expose the required interfaces to respond to these needs have been presented

Moreover, the Enterprise Service Bus (ServiceMix) used to host this web service has been presented; the ESB is the software component that provides a message broker where all the RESOLUTE components (Fram, DSSs) will connect to publish and receive events.

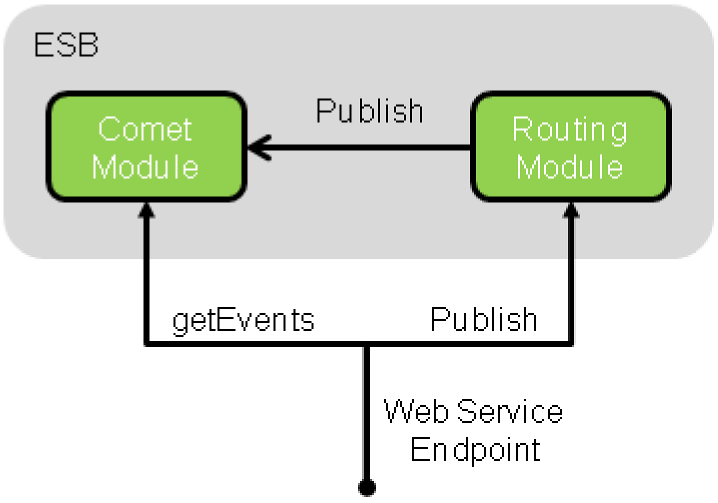


Figure 24 – ESB Messaging Representation

This message broker, as shown in the picture 24 is basically consisting by two software modules, the Routing Module and the Comet Module. Basically, the publication of events reaches the Routing Module. This module can dispatch the events to other ESB modules that can use the received data to perform different actions; the Routing Module sends also the received events to the Comet Module.

All the getEvents requests performed by external systems are sent directly to the Comet Module that can respond with the list of required events, previously received by the Routing Module.

As shown in the previous sections, the ESB adopts the Canonical Data Model Pattern to minimize inter-dependencies among all the heterogeneous applications (each one with its own internal data format and protocols) that takes part to the overall platform.

The ESB is not only responsible of receiving and dispatching events: it also includes other functionalities like persistence, data elaboration/transformation and inference on data. These functionalities are performed by dedicated software modules which run inside Karaf.

The main bundle implemented in ServiceMix is the one that deploys the Web Service to be invoked by every other system in order to send and receive the data events. This bundle deploys the web service at a configurable endpoint: external systems can use the publish and getEvents interfaces by connecting to this endpoint, in order to send/receive Canonical Data Model entities.

Once an event is sent to the platform, it is persisted in the platform database in order to be available for future requests. The event could also be modified or used by other bundles to perform several activities.

The simplest scenario is the following one:

- System A generates some data regarding an alarm;
- The Adaptor related to System A get this data, creates the “Alarm” Entity as defined in the Canonical Data Model and sets all the related attributes using the setter methods of this entity;
- The Adaptor creates an “AlarmEvent” and put the “Alarm” entity in the list of definition of the event;
- The Adaptor connects to the platform endpoint and invoke the “publish” interface; it passes as input of the invoked method the generate AlarmEvent and a Header in which specify that the consumerId is “system A”.
- The AlarmEvent reach the platform: the entry bundle passes this event to the storage bundle that persist this event in the platform database.

- The Adaptor related to System B connects to the web service endpoint and invoke the “getEvents”. The “AlarmEvent” entity created in the previous points in returned.
- The Adaptor related to System B parses the message, translate the relevant data in the data format of System B and send these data to System B that can use them.

A more complex scenario could imply the management of the alarm through the interaction with functionalities guaranteed by the Alarm Manager bundle; in the end this interaction implies an update of some fields of the alarm during time and the consumer systems could retrieve these updates in order to follow the evolution of the alarm.

Other scenarios, more complex than the previous ones, could imply also the execution of specific commands or the generation of new data based on certain states and/or values of the alarms.

The capacity to issue commands, to generate new data, to manage dedicated procedures and so on, can be available in the platform based on the installed bundles and the configuration of rules and parameters: this strongly depends on project scenarios and requirements.

The ESB is thus organized in several independent modules that basically receive some data in input, perform some actions and produce some data as output. Some of these modules could be the alarm manager module, the localization engine module, the resource manager module and so on. The orchestration of the data flow is possible thanks to Apache Camel that allow defining routes of data types between several bundles.

So, from the WS entry point of the ESB, the events data follow a specific flow defined in the Camel routes that brings them to the various software modules and from the software modules to other software modules and so on. Some of the data elaborated by the modules could also be the input of presentation modules in order to arrange the GUI and the data to be shown to the platform operators.

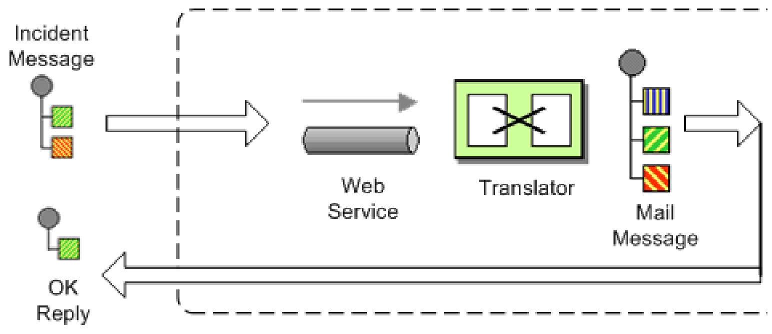


Figure 25 – Camel Route Example

For example, the previous picture shows how it is possible through the usage of camel routes to get the alarm messages generated by a system and send an email to inform the right people. In this example the incident message is sent through a Web Service to a translator module that transforms it to a textual mail message; from there the new message is sent though the mailing system to the right recipient. This is just an example of a very basic route, but this simple mechanism can be replicated for multiple software components, allowing the creation of complex scenarios.

6.1 RESOLUTE IDs specification

Every entity defined in the CDM representation (Alarm, Resource, Person and Chat) has an id field that is always used to univocally identify the data inside the platform. In practice, it is required that every entity that lives in the platform is unique. If a system publishes an alarm with id “1” and another system publishes a completely different alarm with the same id “1”, the platform will assume that the second alarm is a replacement of the previous one and will overwrite it.

Every producer of data has to set the ids for its produced entities: in practice, every system is responsible for the ids of the data it produces, but assuring uniqueness of ids could be difficult, as a system cannot know which are the ids of all the other entities produced by external systems.

Within RESOLUTE it has been decided to follow an approach to simplify this task, in particular by adopting a specific naming convention on ids.

The following points clarify this convention:

- Every ID is a URI (Uniform Resource Identifier), so they shall be put in the form of URI, i.e. lowercase characters separated by the "/" character. No space characters are allowed in a URI: underscore character "_" can be used in place of the space character.
- Every ID shall start with the following prefix string "urn:rixf:"
- Every ID shall specify, after the prefix the system/partner namespace; example: "com.thalesgroup.it"
- Every partner SHALL put ONLY its own namespace.
- After the partner namespace, every ID can contain strings that identifies the type of data sent, eventually the system that produced the data and some numbers or characters to be set to maintain uniqueness.

Some examples of valid IDs for an alarm:

- "urn:rixf:eu.RESOLUTE.partner1/alarms/flooding/alarm01"
- "urn:rixf:eu.RESOLUTE.partner1/alarms/flooding/flooding_alarm_17"
- "urn:rixf:eu.RESOLUTE.partner1/alarms/mugnone/zone_a/2016_12_15/001"

Some examples of NOT valid IDs for an alarm:

- "abc-25-Hwy" (it misses "urn:rixf"; it misses the partner namespace; it contains uppercase characters);
- "urn:rixf:eu.RESOLUTE.partner1/alarms/tramway dss/alarm17" (it contains white spaces);

IMPORTANT: It is the responsibility of every partner to maintain internally uniqueness of its own IDs in order to avoid on the platform the replacement of old alarms with new ones. Please take care that sometimes sending a duplicated id is a good way to replace fake/error values with the right ones.

7 CONCLUSIONS

In complex projects with several different partners, systems and software applications, achieving a flawless integration is difficult.

A direct integration of systems, will lead to many one-to-one inhomogeneous connections, thus huge efforts in terms of maintenance and management costs. Moreover, these architectures are not flexible enough to respond and adapt quickly to changes required by project's needs during pilot definition and execution.

To avoid all these risks and to implement the RESOLUTE Integration Framework, we have proposed an approach based on EIP (Enterprise Integration Patterns) methodologies that represents the state of the art in heterogeneous and distributed systems integration. The RESOLUTE approach to integration is based on the usage of well-known EIP patterns like Message Broker, Message Translator and a Canonical Data Model that has been defined through an extensive work shared between all the partners.

The RESOLUTE Integration Framework has been implemented using ServiceMix (the open source ESB from Apache) and other state of the art technologies (CXF, Camel, Karaf and ActiveMQ) achieving a flexible and extensible integration

This approach will guarantee the ability to evolve and adapt continuously the RESOLUTE Integration Framework during the rest of the project.

8 REFERENCES

References	
EIP	Gregor Hohpe - Bobby Woolf, "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", Addison Wesley, 2003
TOGAF	The Open Group, "TOGAF® Version 9.1", Van Haren Publishing 2011
NAF	North Atlantic Treaty Organization (NATO) "NATO Architecture Framework Version 3.0", 2007
COMET	Wikipedia "Comet (Programming)" https://en.wikipedia.org/wiki/Comet_(programming)
RESOLUTE-D41	D4.1 - Back-end reference architecture and specifications.

Integration Framework Implementation Annex 1

RESOLUTE XML SCHEMA AND WSDL

1 MODEL XML SCHEMA

```
<?xml version="1.0" encoding="utf-8"?><xs:schema
xmlns:resolutemodel="http://org.resolute-eu.com/esb/model"
xmlns:tns="http://org.resolute-eu.com/esb/model"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
targetNamespace="http://org.resolute-eu.com/esb/model" version="1.0">
<xs:complexType name="Resource">
    <xs:sequence>
      <xs:element minOccurs="0" name="location" type="tns:Location"/>
      <xs:element minOccurs="0" name="properties">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="property"
type="tns:Property"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="states">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="state"
type="tns:State"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="parentId" type="xs:anyURI"/>
    <xs:attribute name="typeId" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="Location">
    <xs:complexContent>
      <xs:extension base="tns:Coordinates">
        <xs:sequence/>
        <xs:attribute name="altitude" type="xs:double"/>
        <xs:attribute name="heading" type="xs:double"/>
        <xs:attribute name="speed" type="xs:double"/>
        <xs:attribute name="time" type="xs:dateTime"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="Coordinates">
    <xs:sequence/>
    <xs:attribute name="latitude" type="xs:double" use="required"/>
    <xs:attribute name="longitude" type="xs:double" use="required"/>
  </xs:complexType>
<xs:complexType name="Property">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="typeId" type="xs:anyURI" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
<xs:complexType name="State">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="time" type="xs:dateTime"/>
        <xs:attribute name="typeId" type="xs:anyURI" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
<xs:complexType name="Alarm">
    <xs:sequence>
      <xs:element minOccurs="0" name="location" type="tns:Location"/>
      <xs:element minOccurs="0" name="message" type="xs:string"/>
      <xs:element minOccurs="0" name="resourceIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="resourceId" type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="closeTime" type="xs:dateTime"/>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="openTime" type="xs:dateTime"/>
    <xs:attribute name="severity" type="tns:AlarmSeverity"/>
    <xs:attribute name="status" type="tns:AlarmStatus"/>
    <xs:attribute name="typeId" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="Chat">
    <xs:sequence>
      <xs:element minOccurs="0" name="messages">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="message"
type="tns:InstantMessage"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="participantIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="participantId" type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="creationDate" type="xs:dateTime"/>
    <xs:attribute name="creatorId" type="xs:anyURI"/>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
<xs:complexType name="InstantMessage">
    <xs:sequence/>
    <xs:attribute name="creationTime" type="xs:dateTime" use="required"/>
    <xs:attribute name="senderId" type="xs:anyURI" use="required"/>
    <xs:attribute name="text" type="xs:string"/>
  </xs:complexType>
<xs:complexType name="Person">
    <xs:sequence>
      <xs:element minOccurs="0" name="addresses">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="address"
type="tns:Address"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="emails">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="email"
type="tns:Email"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="telephones">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="telephone" type="tns:Telephone"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="additionalName" type="xs:string"/>
    <xs:attribute name="familyName" type="xs:string"/>
    <xs:attribute name="givenName" type="xs:string"/>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="note" type="xs:string"/>
    <xs:attribute name="personStatus" type="tns:PersonStatus"/>
  </xs:complexType>
<xs:complexType name="Address">
    <xs:sequence/>
    <xs:attribute name="code" type="xs:string"/>
    <xs:attribute name="country" type="xs:string"/>
    <xs:attribute name="locality" type="xs:string"/>
    <xs:attribute name="region" type="xs:string"/>
    <xs:attribute name="street" type="xs:string"/>
    <xs:attribute name="usage" type="tns:ContactUsage" use="required"/>
  </xs:complexType>
<xs:complexType name="Email">
    <xs:sequence/>
    <xs:attribute name="mailAddress" type="xs:string" use="required"/>
    <xs:attribute name="usage" type="tns:ContactUsage" use="required"/>
  </xs:complexType>
<xs:complexType name="Telephone">
    <xs:sequence/>
    <xs:attribute name="contactUsage" type="tns:ContactUsage"/>
    <xs:attribute name="label" type="xs:string"/>
    <xs:attribute name="number" type="xs:string" use="required"/>
    <xs:attribute name="telephoneType" type="tns:TelephoneType"/>
  </xs:complexType>
<xs:complexType name="EvacuationNode">
    <xs:sequence>
      <xs:element minOccurs="0" name="coords" type="tns:Coordinates"/>
      <xs:element minOccurs="0" name="status"
type="tns:EvacuationNodeStatus"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="EvacuationNodeStatus">
    <xs:sequence/>
    <xs:attribute name="mode" type="tns:NodeMode"/>
    <xs:attribute name="numberOfBlindPeopleInNode" type="xs:int"/>
    <xs:attribute name="numberOfBusPeopleInNode" type="xs:int"/>
    <xs:attribute name="numberOfCarPeopleInNode" type="xs:int"/>
    <xs:attribute name="numberOfElderlyPeopleInNode" type="xs:int"/>
    <xs:attribute name="numberOfNormalPeopleInNode" type="xs:int"/>
    <xs:attribute name="numberOfRescuersInNode" type="xs:int"/>
    <xs:attribute name="numberOfRescuersNeeded" type="xs:int"/>
    <xs:attribute name="numberOfWheelchairPeopleInNode" type="xs:int"/>
    <xs:attribute name="rescuersPriority" type="xs:int"/>
    <xs:attribute name="type" type="tns:NodeType"/>
  </xs:complexType>
<xs:complexType name="EvacuationPlan">
    <xs:sequence>
      <xs:element minOccurs="0" name="coords">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="coord"
type="tns:Coordinates"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="nodeIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="nodeId"
type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="evacuationPlanStatus"
type="tns:EvacuationPlanStatus"/>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="numberOfAgents" type="xs:int"/>
    <xs:attribute name="numberOfBlindAgents" type="xs:int"/>
  </xs:complexType>
<xs:complexType name="NetworkAnalysis">
    <xs:sequence>
      <xs:element minOccurs="0" name="edgeMeasures">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="edgeMeasure" type="tns:EdgeMeasure"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="nodeMeasures">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="nodeMeasure" type="tns:NodeMeasure"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="time" type="xs:dateTime"/>
  </xs:complexType>
<xs:complexType name="EdgeMeasure">
    <xs:sequence>
      <xs:element minOccurs="0" name="criticalEdges">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="criticalEdge" type="tns:ElementEvaluation"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
<xs:complexType name="ElementEvaluation">
    <xs:sequence/>
    <xs:attribute name="elementId" type="xs:anyURI" use="required"/>
    <xs:attribute name="value" type="xs:string"/>
  </xs:complexType>
<xs:complexType name="NodeMeasure">
    <xs:sequence>
      <xs:element minOccurs="0" name="criticalNodes">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="criticalNode" type="tns:ElementEvaluation"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
<xs:complexType name="InformationSubType">
    <xs:sequence>
      <xs:element minOccurs="0" name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="InformationType">
    <xs:sequence>
      <xs:element minOccurs="0" name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="ActuationStrategy">
    <xs:sequence>
      <xs:element minOccurs="0" name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="activationThreshold" type="xs:double"/>
    <xs:attribute name="activationType" type="xs:string"/>
    <xs:attribute name="currentMembership" type="xs:double"/>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="priority" type="xs:int"/>
    <xs:attribute name="time" type="xs:dateTime"/>
  </xs:complexType>
<xs:complexType name="TrafficInformation">
    <xs:sequence>
      <xs:element minOccurs="0" name="arcIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="arcId"
type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="coords" type="tns:Coordinates"/>
      <xs:element minOccurs="0" name="notes" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="code" type="xs:string"/>
    <xs:attribute name="id" type="xs:anyURI" use="required"/>
    <xs:attribute name="severity" type="xs:int"/>
    <xs:attribute name="source" type="xs:string"/>
    <xs:attribute name="startTime" type="xs:dateTime"/>
    <xs:attribute name="stopTime" type="xs:dateTime"/>
    <xs:attribute name="subTypeId" type="xs:anyURI"/>
    <xs:attribute name="typeId" type="xs:anyURI"/>
    <xs:attribute name="updateTime" type="xs:dateTime"/>
  </xs:complexType>
<xs:simpleType name="AlarmSeverity">
    <xs:restriction base="xs:string">
      <xs:enumeration value="CRITICAL"/>
      <xs:enumeration value="MAJOR"/>
      <xs:enumeration value="MINOR"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="AlarmStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="ALARM_CLOSED"/>
      <xs:enumeration value="ALARM_OPEN"/>
      <xs:enumeration value="ARCHIVED"/>
      <xs:enumeration value="NO_ALARM"/>
      <xs:enumeration value="PRE_ALARM"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="ContactUsage">
    <xs:restriction base="xs:string">
      <xs:enumeration value="BUSINESS"/>
      <xs:enumeration value="EMERGENCY"/>
      <xs:enumeration value="PERSONAL"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="PersonStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="AVAILABLE"/>
      <xs:enumeration value="AWAY"/>
      <xs:enumeration value="BUSY"/>
      <xs:enumeration value="OFFLINE"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="TelephoneType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="FIXED"/>
      <xs:enumeration value="MOBILE"/>
      <xs:enumeration value="OTHER"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="NodeMode">
    <xs:restriction base="xs:string">
      <xs:enumeration value="EVACUATION"/>
      <xs:enumeration value="RESCUE"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="NodeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="EVACUATE_FROM"/>
      <xs:enumeration value="EVACUATE_TO"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="EvacuationPlanStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="APPROVED"/>
      <xs:enumeration value="CLOSED"/>
      <xs:enumeration value="IN_PROGRESS"/>
      <xs:enumeration value="PROPOSED"/>
      <xs:enumeration value="REJECTED"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

2 SERVICE XML SCHEMA

```
<?xml version="1.0" encoding="utf-8"?><xs:schema
xmlns:resolutemodel="http://org.resolute-eu.com/esb/model"
xmlns:resoluteservice="http://org.resolute-eu.com/esb/service"
xmlns:tns="http://org.resolute-eu.com/esb/service"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
targetNamespace="http://org.resolute-eu.com/esb/service" version="1.0">
<xs:import namespace="http://org.resolute-eu.com/esb/model"
schemaLocation="resolute_schema2.xsd"/>
<xs:element name="getEventsIn" type="tns:getEventsIn"/>
<xs:element name="getEventsOut" type="tns:getEventsOut"/>
<xs:element name="publishIn" type="tns:publishIn"/>
<xs:element name="publishOut" type="tns:publishOut"/>
<xs:complexType name="publishIn">
    <xs:sequence>
      <xs:element minOccurs="0" name="header" type="tns:Header"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="event"
type="tns:Event"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="Header">
    <xs:sequence/>
    <xs:attribute name="consumerId" type="xs:anyURI" use="required"/>
    <xs:attribute name="requestTime" type="xs:dateTime" use="required"/>
  </xs:complexType>
<xs:complexType name="CometHeader">
    <xs:complexContent>
      <xs:extension base="tns:Header">
        <xs:sequence/>
        <xs:attribute name="lastToken" type="xs:string"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType abstract="true" name="Event">
    <xs:sequence/>
    <xs:attribute name="time" type="xs:dateTime"/>
  </xs:complexType>
<xs:complexType name="ResourceEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
          <xs:element minOccurs="0" name="resourceDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="resourceDefinition" type="resolutemodel:Resource"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="resourceRemovalIds">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="resourceRemovalId" type="xs:anyURI"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="resourceUpdates">
            <xs:complexType>
              <xs:sequence>
```

```
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="resourceUpdate" type="tns:ResourceUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="ResourceUpdate">
    <xs:sequence>
      <xs:element minOccurs="0" name="locations">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="location"
type="resolutemodel:Location"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="states">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="state"
type="resolutemodel:State"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="resourceId" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="AlarmEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
          <xs:element minOccurs="0" name="alarmDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="alarmDefinition" type="resolutemodel:Alarm"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="alarmUpdates">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="alarmUpdate" type="tns:AlarmUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="AlarmUpdate">
    <xs:sequence>
      <xs:element minOccurs="0" name="additionalResourceIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="additionalResourceId" type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="location"
type="resolutemodel:Location"/>
      <xs:element minOccurs="0" name="message" type="xs:string"/>
      <xs:element minOccurs="0" name="removedResourceIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="removedResourceId" type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="alarmId" type="xs:anyURI" use="required"/>
    <xs:attribute name="closeTime" type="xs:dateTime"/>
    <xs:attribute name="openTime" type="xs:dateTime"/>
    <xs:attribute name="severity" type="resolutemodel:AlarmSeverity"/>
    <xs:attribute name="status" type="resolutemodel:AlarmStatus"/>
  </xs:complexType>
<xs:complexType name="CommandEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
          <xs:element minOccurs="0" name="commands">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="command" type="tns:Command"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="Command">
    <xs:sequence>
      <xs:element minOccurs="0" name="recipientIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="id"
type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="typeId" type="xs:anyURI"/>
  </xs:complexType>
<xs:complexType name="TextualCommand">
    <xs:complexContent>
      <xs:extension base="tns:Command">
        <xs:sequence>
          <xs:element minOccurs="0" name="text" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="ChatEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
```

```
          <xs:element minOccurs="0" name="chatDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="chatDefinition" type="resolutemodel:Chat"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="chatRemovalIds">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="chatRemovalId" type="xs:anyURI"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="chatUpdates">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="chatUpdate" type="tns:ChatUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="ChatUpdate">
    <xs:sequence>
      <xs:element minOccurs="0" name="additionalMessages">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="additionalMessage" type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="additionalParticipantIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="additionalParticipantId" type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="conversationName" type="xs:string"/>
      <xs:element minOccurs="0" name="removalParticipantIds">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="removalParticipantId" type="xs:anyURI"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="conversationId" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="ProfileEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
```

```
          <xs:element minOccurs="0" name="personDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="personDefinition" type="resolutemodel:Person"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="personRemovalIds">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="personRemovalId" type="xs:anyURI"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="personUpdates">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="personUpdate" type="tns:PersonUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="PersonUpdate">
    <xs:sequence/>
    <xs:attribute name="personId" type="xs:anyURI" use="required"/>
    <xs:attribute name="personStatus" type="resolutemodel:PersonStatus"/>
  </xs:complexType>
<xs:complexType name="EvacuationEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
          <xs:element minOccurs="0" name="nodeDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="nodeDefinition" type="resolutemodel:EvacuationNode"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="nodeRemovalIds">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="nodeRemovalId" type="xs:anyURI"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="nodeUpdates">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="nodeUpdate" type="tns:EvacuationNodeUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="planDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="planDefinition" type="resolutemodel:EvacuationPlan"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="planUpdates">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="planUpdate" type="tns:EvacuationPlanUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="EvacuationNodeUpdate">
    <xs:sequence>
      <xs:element minOccurs="0" name="status"
type="resolutemodel:EvacuationNodeStatus"/>
    </xs:sequence>
    <xs:attribute name="nodeId" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="EvacuationPlanUpdate">
    <xs:sequence/>
    <xs:attribute name="evacuationPlanStatus"
type="resolutemodel:EvacuationPlanStatus"/>
    <xs:attribute name="planId" type="xs:anyURI" use="required"/>
  </xs:complexType>
<xs:complexType name="NetworkAnalysisEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
          <xs:element minOccurs="0" name="networkAnalysisDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="networkAnalysisDefinition" type="resolutemodel:NetworkAnalysis"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="TrafficEvent">
    <xs:complexContent>
      <xs:extension base="tns:Event">
        <xs:sequence>
          <xs:element minOccurs="0" name="informationSubTypeDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="informationSubTypeDefinition"
type="resolutemodel:InformationSubType"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="informationTypeDefinitions">
```

```
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="informationTypeDefinition" type="resolutemodel:InformationType"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="strategyDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="strategyDefinition" type="resolutemodel:ActuationStrategy"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="strategyUpdates">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="strategyUpdate" type="tns:ActuationStrategyUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="trafficInformationDefinitions">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="trafficInformationDefinition"
type="resolutemodel:TrafficInformation"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element minOccurs="0" name="trafficInformationUpdates">
            <xs:complexType>
              <xs:sequence>
                <xs:element maxOccurs="unbounded" minOccurs="0"
name="trafficInformationUpdate" type="tns:TrafficInformationUpdate"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
<xs:complexType name="ActuationStrategyUpdate">
    <xs:sequence/>
    <xs:attribute name="strategyId" type="xs:anyURI" use="required"/>
    <xs:attribute name="currentMembership" type="xs:double"/>
  </xs:complexType>
<xs:complexType name="TrafficInformationUpdate">
    <xs:sequence/>
    <xs:attribute name="informationId" type="xs:anyURI" use="required"/>
    <xs:attribute name="stopTime" type="xs:dateTime"/>
    <xs:attribute name="updateTime" type="xs:dateTime"/>
  </xs:complexType>
<xs:complexType name="publishOut">
    <xs:sequence/>
  </xs:complexType>
<xs:complexType name="getEventsIn">
    <xs:sequence>
      <xs:element minOccurs="0" name="header" type="tns:CometHeader"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="getEventsOut">
    <xs:sequence>
      <xs:element minOccurs="0" name="getEvents" type="tns:CometEvents"/>
    </xs:sequence>
  </xs:complexType>
<xs:complexType name="CometEvents">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="event"
type="tns:Event"/>
    </xs:sequence>
    <xs:attribute name="responseToken" type="xs:string" use="required"/>
    <xs:attribute name="tokenExpires" type="xs:dateTime"/>
  </xs:complexType>
</xs:schema>
```

3 WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="ResoluteServiceService"
targetNamespace="http://org.resolute-eu.com/esb/service"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://org.resolute-eu.com/esb/service"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <wsdl:types>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <import namespace="http://org.resolute-eu.com/esb/service"
schemaLocation="resolute_schema1.xsd"/>
</schema>
  </wsdl:types>
  <wsdl:message name="publishResponse">
    <wsdl:part name="parameters" element="tns:publishOut">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getEventsResponse">
    <wsdl:part name="parameters" element="tns:getEventsOut">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="publish">
    <wsdl:part name="parameters" element="tns:publishIn">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getEvents">
    <wsdl:part name="parameters" element="tns:getEventsIn">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="ResoluteService">
    <wsdl:operation name="publish">
      <wsdl:input name="publish" message="tns:publish">
    </wsdl:input>
      <wsdl:output name="publishResponse" message="tns:publishResponse">
    </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getEvents">
      <wsdl:input name="getEvents" message="tns:getEvents">
    </wsdl:input>
      <wsdl:output name="getEventsResponse"
message="tns:getEventsResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="ResoluteServiceServiceSoapBinding"
type="tns:ResoluteService">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="publish">
      <soap:operation soapAction="urn:#publish" style="document"/>
      <wsdl:input name="publish">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="publishResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getEvents">
      <soap:operation soapAction="urn:#getEvents" style="document"/>
      <wsdl:input name="getEvents">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="getEventsResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ResoluteServiceService">
    <wsdl:port name="ResoluteServicePort"
binding="tns:ResoluteServiceServiceSoapBinding">
      <soap:address location="http://localhost:9090/ResoluteServicePort"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```