Architecture specification and integration requirements

Grant Agreement N°: FP7 - SCP0 – GA – 2011 - 265647

Project Acronym: **ON-TIME**

Project Title: **O**ptimal **N**etworks for **T**rain Integration **M**anagement across **E**urope

Funding scheme: Collaborative Project

Project start: 1 November 2011

Project duration: 3 Years

Work package no.: WP7

Deliverable no.: D7.2; Revision 4

Status/date of document: Final, 29/04/2013

Due date of document: 30/04/2013

Actual submission date: 29/04/2013

Lead contractor for this document: NTT Data

Project website: www.ontime-project.eu

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision control / involved partners

Following table gives an overview on elaboration and processed changes of the document:

Revision	Date	Name / Company short name	Changes
1	06/02/2013	NTT DATA	First Draft
2	18/03/2013	NTT DATA	First release of a detailed event List with roles and responsibility Added WP5 Integration specification Revised WP4 Specification Revised TCS specification in order to handle data management from the TCS itself
3	22/04/2013	NTT DATA	Final Draft Revision for review Revised: ▪ State of the Art ▪ Data Entities ▪ Event Model ▪ Integration Interfaces with WP3, 4, 5 and 6.
4	26/04/2013	NTT DATA	Minor Reviews for submissions

Following project partners have been involved in the elaboration of this document:

Partner No.	Company short name	Involved experts
	NTT	Matteo Anelli, Bruno Ambrogio, Daniele Carcasole
	TUD	Thomas Albrecht
	UOB	John Easton

Executive Summary

This document details the architectural and technical approach to the integration of the ON-TIME distributed architecture.

Building on the data dictionary described in D7.1, this document will define and describe how modules will communicate each other within the ON-TIME architecture and how the data flows are modelled in a real-time environment.

In section 2 and 3, general requirements of the architecture are given, and basic principles used in it are described.

Extending the basic architecture principles, section 4 illustrate an analysis of some available state-of-the-art systems, often used to implement what is described in section 3.

Section 5 describes the architecture specifications, illustrating how it can be deployed, and some example scenarios that includes other WP's modules. It also gives a component-based view of the architecture.

In section 6, the data and event models are explained. An extensive list of the events generated by modules and by the platform is given, with publisher/subscriber and role/responsibility descriptions.

Finally, section 7 defines the Integration Requirements, explaining how TCS' and other WPs integrate with the WP7 architecture.

Table of contents

EXECUTIVE SUMMARY ... 3

TABLE OF CONTENTS ... 4

TABLE OF FIGURES ... 6

1 INTRODUCTION ... 7
- 1.1 Terminology ... 7

2 GENERAL REQUIREMENTS ... 8

3 BASIC ARCHITECTURE PRINCIPLES ... 9
- 3.1 Distributed architecture ... 9
- 3.2 Publish-Subscribe communication pattern ... 10
- 3.3 REST Web Services ... 12
- 3.4 Document-based DBMS ... 13

4 STATE-OF-THE-ART ANALYSIS ... 15
- 4.1 Queue Messaging Systems ... 15
  - 4.1.1 Description of the AMQP Protocol ... 16
- 4.2 REST vs. SOAP Web Services ... 18
- 4.3 Document-based Database Management Systems ... 18

5 ARCHITECTURE SPECIFICATIONS ... 21
- 5.1 Possible architecture scenarios ... 25
- 5.2 Component-based view of the architecture ... 26

6 DATA AND EVENT MODEL ... 27
- 6.1 Published Data Entities ... 27
- 6.2 Event Model ... 29
  - 6.2.1 ConnectionConflictEvent ... 31
  - 6.2.2 ConnectionScheduleAvailableEvent ... 32
  - 6.2.3 CrewConflictEvent ... 33
  - 6.2.4 CrewScheduleAvailableEvent ... 34
  - 6.2.5 LineDisruptionEvent ... 35
  - 6.2.6 LockedSwitchDirectionEvent ... 36
  - 6.2.7 PlatformDisruptionEvent ... 37
  - 6.2.8 RealTimeTrafficPlanAvailableEvent ... 38
  - 6.2.9 RollingStockChangeEvent ... 38
  - 6.2.10 RollingStockConflictEvent ... 39
  - 6.2.11 RollingStockDisruptionEvent ... 40
  - 6.2.12 RollingStockScheduleAvailableEvent ... 41
  - 6.2.13 SetRouteEvent ... 42
  - 6.2.14 SignalStateChangeEvent ... 43
  - 6.2.15 StationDisruptionEvent ... 44
  - 6.2.16 SwitchDisruptionEvent ... 45
  - 6.2.17 TDSectionOccupationEvent ... 46
  - 6.2.18 TDSectionReleaseEvent ... 47
  - 6.2.19 TemporarySpeedRestrictionsEvent ... 48

6.2.20 TrackDisruptionEvent 49
6.2.21 TrainEnterEvent 50
6.2.22 TrainExitEvent 51
6.2.23 TrainMassChangeEvent 52
6.2.24 TrainPassengerCountChangeEvent 53
6.2.25 TrainPathEnvelopeAvailableEvent 54
6.2.26 TrainPositionChangeEvent 55
6.2.27 TrainSpeedChangeEvent 56
6.2.28 TrainSuppressedEvent 57
6.2.29 UpdateConnectionScheduleEvent 58
6.2.30 UpdateCrewScheduleEvent 59
6.2.31 UpdateRollingStockScheduleEvent 60
6.2.32 UpdateRealTimeTrafficPlanEvent 61
6.2.33 UpdateTrainPathEnvelopeEvent 62

7 INTEGRATION REQUIREMENTS 63
7.1 Integration Interfaces 63
7.1.1 External Integration Interfaces 64
7.1.2 EventConsumer Interface 67
7.1.3 DataProvider Interface 68
7.1.4 Routing Interface 71
7.2 Integration with TCS 75
7.2.1 Integration with WP3 Modules 77
7.3 Integration with WP4 Modules 78
7.3.1 TrainPathEnvelopeComputation module (TPEC) 79
7.4 Integration with WP5 Modules 80
7.5 Integration with WP6 Modules 83

8 REFERENCES 86

Table of figures

Figure 1 - General architecture scenario 21
Figure 2 - Replicated AMQP Queues 22
Figure 3 - Distributed Event Processors 23
Figure 4 - MongoDB distributed environment 23
Figure 5 - Example scenario with distributed modules 25
Figure 6 - Component-based view of the architecture...26
Figure 7 - Data Entities as they are published by the DataProvider Service 27
Figure 8 - Events dispatched and consumed by the EventProcessor Interface...........29
Figure 9 - WP4 Integration Flow 78
Figure 10 - WP5 Integration Flow 80
Figure 11 - WP6 Integration Flow 84

1 INTRODUCTION

1.1 Terminology

TCS	Traffic Control System: a collection of systems that does monitor and manage the traffic system.
ICT	Information and Communications Technology
ONT	ONTIME project code on the project document repository
W3C	World Wide Web Consortium
WPx	Short code for a work package (e.g. WP7 refers to work package 7)
XML	eXtensible Mark-up Language
XSD	XML Schema Document
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
MoM	Message Oriented Middleware
API	Application Programming Interface
URI	Uniform Resource Identifier
JSON	JavaScript Object Notation
BSON	Binary JSON
DBMS	Database Management System
RDBMS	Relational DBMS
NoSQL	Category of non-relational DBMS
FIFO	First-In-First-Out
LAN	Local Area Network
WAN	Wide Area Network
TCP	Transfer Control Protocol
SSL	Secure Socket Layer
SOAP	Simple Object Access Protocol
AJAX	Asynchronous JavaScript And XML
SOA	Service Oriented Architecture
UML	Unified Modelling Language

2 GENERAL REQUIREMENTS

The rise of distributed systems and protocols paved the way to scale and architect **very complex** information systems. Railway management systems are very good candidates to use a distributed architecture, because of their natural **geographical distribution** and the need to be interconnected with bounding foreign systems to exchange data and operational information.

The **ON-TIME Project** proposes a distributed architecture to integrate different algorithms to solve typical problems of Railway Management Systems, such as timetabling, microscopic and macroscopic dynamic planning, resource management and scheduling. Since would be unfeasible to create a system capable of substituting current Train Management Systems, the purpose of the ON-TIME Architecture is to complement Train Control Systems extending their functionalities with a new layer of algorithms and real-time solutions to cope with the usually static planning of Train Control Systems.

They key-purpose of the architecture definition is to define a distributed, configurable and flexible infrastructure to exchange data and messages between different modules. The advantage of using a distributed architecture in this context is the ability to collect and exchange data on systems that are by their own nature **loosely coupled** (like timetabling management software and crew management software, for example) and create a coherent, dynamic communication context in which these information can be exchanged.

Data definition and software standards are equally important as the architecture itself. **Data and technical standards** must be implemented in order to easily integrate a collection of systems and to represent data in a way such that regional differences between neighbouring systems will have a small impact on the communication semantic. Since most of the European Countries have different processes and data standards, a **common data representation** is needed. In terms of data representation, as described also in D7.1, **open standards** must be used to encourage the adoption of the platform and to implement a uniform data representation that will supersede specific regional requirements.

Another key aspect is to treat modules as services that can be queried and interacted by other systems and users. This is a very important aspect of distributed systems: each functional module should be black-boxed and self-sufficient, to be easily replaced by another implementation using the same modular framework.

Since for real-time operational systems is paramount to have data consistency and to avoid synchronization issues, seeing systems as services opens the possibility to abstract their data as services as well.

Furthermore, in order to facilitate the **integration** with legacy and brand-new systems, the architecture must use open communication standards and integration frameworks suitable for **scaling** from *small-scale* up to *large-scale* distributed systems without compromising performance.

3 BASIC ARCHITECTURE PRINCIPLES

The architecture used in the ON-TIME Project must aim to be a modular, service-based and distributed computing architecture facilitating the integration of diverse ICT systems and exposing a range of algorithms via standard communication interfaces.

The basic architecture principles of the platform are:

- **Modularity.** Every module of the architecture is an abstract black box that defines a single functional unit. Modules that need information in real time consume events generated by the platform and by external sources, such as sensors or traffic control systems. Module abstraction allows easy substitution of differing implementations of similar functional units.
- **Event-Based.** Communication between different modules is managed by the exchange of events between the single modules and the architecture, removing any dependencies between the implemented functional units within a workflow.
- **Distributed.** State of the art application protocols will be employed to ensure that the platform can operate in distributed environments or in an independent, monolithic configuration.
- **Extensible.** XML representations will be used for events and data entities reducing the risk on dependencies on the choice of particular programing languages.

Non-real-time data (for example infrastructure data and the current timetable) will be provided via an on-demand, read-only public interface allowing it to be accessed as required by any system in the platform without the risk of data becoming corrupted.

3.1 Distributed architecture

Dealing with real-time and near real-time systems, where several entities are involved, each one either producing or consuming data, is naturally achieved by means of a distributed environment. This means that those several entities could be located not only on different systems, but also in different geographical areas.

Of course, such kind of an environment requires that the different systems must communicate each other in order to coordinate the management of computation tasks, increasing the complexity of its construction and setup. On the other hand, several advantages can be listed:

- **Fault-tolerance.** Replicating the main roles of the architecture among different systems makes easy to avoid stops of the architecture after a failure (either hardware or software one). In this way the availability of the system is increased, while the overhead of maintaining consistency among replicas should be taken into account when, for example, they stores data.
- **Scalability.** When the number of users or messages increases, the load of the entire architecture increases. This leads to higher latencies and possible message losses, not always admitted by applications. A distributed environment is more suitable for dealing with this increase of load, by means of, for example, adding a new node to the architecture and route some requests to it.

- **Performance.** Having more replicas allows load distribution among them, eventually reducing the service time for a single request, since the workload is generally maintained as balanced as possible among all nodes.

Furthermore, from a user point of view, a distributed architecture allows clients to integrate their modules with the whole system, while keeping them at their premises, and hence maintaining the full control on them.

3.2 Publish-Subscribe communication pattern

The well-known point-to-point communication pattern is not suitable for use in loosely-coupled architectures where several modules must communicate each other asynchronously, without having detailed information on network topology. In this type of architecture, multiple modules commonly require access to information generated by a single sender (multicast communications) and the run-time state of a process may change the message types it needs to complete its assigned task.

In these scenarios, where scalability and flexibility are of paramount importance, the communications management between modules should be left to a message-oriented middleware (*MoM*) employing a Publish-Subscribe pattern. In this pattern, messages (characterized by topic) are sent by message publishers to the middleware layer. Subscribers receive from the middleware only message types that they previously subscribed to. Subscriptions can be added and removed at any time.

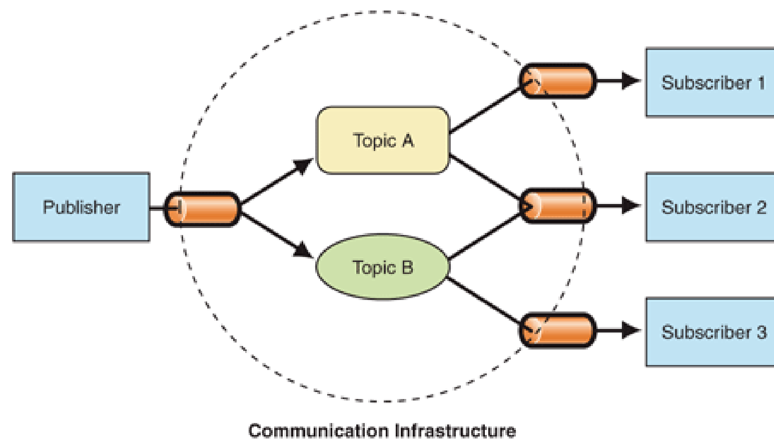


Figure 1 Communication Pattern

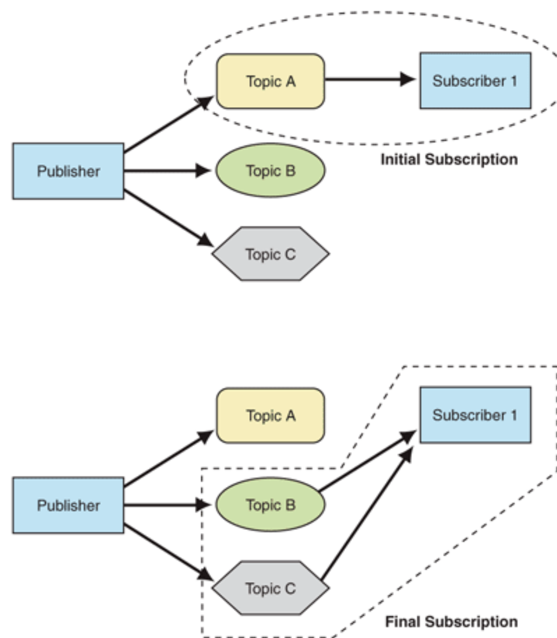


Figure 2 - Dynamic Subscriptions

Once a publisher sends a message to the platform, it must route it to all the right subscribers. Different approaches are possible:

- **Topic-based routing.** Subscriptions are issued expressing the topic of interest, i.e. the type of message, no matter what the message content will be. Message data can be unstructured.
- **Content-based routing.** Subscriptions are issued by imposing conditions on the content / values of the message. In this case, data contained within messages must be structured. Content-based routing requires an analysis of the content of each message; this is usually implemented by means of CEP (Complex Event Processing) systems, which typically offer not only content analysis but also content aggregation and filtering. This additional functionality can be useful in scenarios where large streams of events are present in the system but imposes an additional processing load on the system.

It is also common, for the communication middleware, to manage the following additional quality-control aspects of message passing within the system:

- **Reliability.** Once a message is sent to the architecture, eventually all the interested subscribers will deliver it. The middleware ensures that no message is duplicated, and no spurious message is created.
- **Durability.** The communication infrastructure could manage the dispatch of messages either in a durable, or non-durable way. The difference between them is evident when a crash of the communication system occurs. With the former, a message sent before the crash is still delivered to the subscribers once the communication system has been recovered.

3.3 REST Web Services

Short for Representational State Transfer, REST is an architectural style for developing web services. Since it relies on the HTTP protocol, it became the *de-facto* standard for web APIs architecture designs also for the simplicity of its usage and integration.

The HTTP protocol also implies that in REST architectures we have clients and servers. Like a classic request for a web page, clients initiate requests to servers that process them and return the appropriate responses. The main entity involved in REST requests is the representation of a resource that is an abstract document that may be addressed. Typically, data exchanged within requests is represented using the JSON format, that provides a compact representation, but every other hypertext valid media type can be used, such as XML.

In a general web API architecture using REST, implementing the so-called RESTful APIs, every resource (or collection of resources) is referenced by an URI and accessed by means of standard HTTP request types, maintaining their semantics:

- **GET.** Perhaps the most important one, used in the web context for downloading web pages. In the REST context, it is used to retrieve and list resources or, more precisely, a representation of them, for example JSON or XML data.
- **POST.** Along with GET, it is the other widely used request type, in the web context, for example whenever the user is compiling a form and sending its data to the server. In the REST context, a POST request to a specific resource URI it is used to replace that resource with a new one or, in case the resource is a collection of items, to create a new instance of that resource and add it to the collection.

These two request types are the most widely supported by web browsers, and hence the most well-known ones. However, HTTP specifications include also other request types, useful in a RESTful API context:

- **PUT.** Very similar to POST, it is used to replace the resource addressed by a specific URI with the one passed by parameter. This is true both for collections of resources and single-item resources. For this reason, POST is not generally used is case of single-item resources, since PUT better fits the semantic of the operation.
- **DELETE.** As the name implies, it is used to delete a resource addressed by a specific URI, regardless of its multiplicity.
- **HEAD.** Pretty similar to GET, but it does not retrieve the entire representation of the resource, but only the HTTP header with all the information included in it, such as the timestamp of the last modification of that resource or its content length. This kind of requests may be used by clients to implement features like caching and to avoid downloading the same resource representation twice.

GET and HEAD are defined as *safe*, since they do not make *side-effects* on data, but simply retrieve it. They are also defined as *idempotent*, since their multiple invocation give the same end result. DELETE and PUT are clearly not safe, since they change the state of one or more resources on the server. DELETE is also idempotent, since every

resource can be deleted at most once, and so is PUT, since after doing several times the same request, the server ends up with the same state that would be obtained with only one of them. POST is not definitively associated with these definitions: it depends on what it is used for.

Parameters to REST requests can be basically passed in two ways: appending them to the URI or writing them in the body of the request. The parameters passed by means of the former method are usually called GET parameters, while the ones passed by means of the latter are usually named POST parameters. The reason is simply because the former is the natural method to use for the GET request type, while the latter for the POST one. However, this association is not mandatory.

The lack of mandatory requirements for RESTful APIs is due to the fact that REST is an architectural style rather than a standard. A REST architecture is a combination of standards.

3.4 Document-based DBMS

Document-based databases are part of the NoSQL family of database management systems. The main concept behind them is to store data representing it as *documents* rather than *records*, typically represented in formats such as JSON, XML and BSON.

From an abstract point of view, a document and a collection, in a document-based database refers to a record and a table in a relational database, respectively. The main difference is in the ***flexibility*** of their structure. In a RDBMS, the schema of the data is rigid, defined at design time of the database, giving to each record a precise type (columns name and type), implied by its own table. Hence, all records within a table have the same structure.

On the other hand, in a document-based database, two documents of the same *logical* type, i.e. belonging to the same collection, may have slightly different data stored in, as shown in the following example:

```
Train1 = {number: 1234, type: “cargo”, mass: 1500}
Train2 = {number: 4321, type: “passenger”,
    commercialStops: [
    {name: “station1”, arrival: 16.00.00, departure: 16.02.00},
    {name: “station2”, arrival: 17.00.00, departure: 17.02.00}
    ]}
```

where a “cargo” train has not commercial stops, while “passenger” ones have.

This kind of flexibility clearly gives the advantage of reducing the ***sparseness*** of data: in relational databases, whenever different records in a single table may store different sets of columns, two methodologies are generally used: the structure of the table holds all the columns, records that store values only for a subset of them have a null-value for the remaining columns; acting the so-called decomposition of the schema, creating additional tables in order to minimize the presence of null-values. The latter method is generally preferred, because managing null-values is tricky and some problems could arise. In general, null-values waste storage space. Moreover, a column with null-values cannot be primary key and, last but not the least, there could be

problems in applying SQL aggregation functions on them, such as AVG, SUM and COUNT.

However, as the number of tables increases, the number of joins increases, when the database is queried for retrieving data. Joins are considered heavy operations, as they increase the ***time complexity*** of queries. Moreover, query execution times in relational databases increase with the number of records stored in tables. This can be often slightly mitigated with the use of indexes, but the effort to maintain them consistent with data in related tables must be taken into account.

Document-based databases usually refer to joins as *references* between documents. References are different from relational joins, since they can be evaluated only if needed. Joins, on the other hand, are calculated on the entire set of records, and filtering is done afterwards.

Another aspect to consider is that relational databases ship with the famous ACID properties for ***transactions***, which ensure some guarantees on data:

- **Atomicity.** The execution of a transaction is seen by the user as an atomic operation: either all actions are carried out or none are.
- **Consistency.** Each transaction, executed alone on the database, must preserve the consistency of the data. The DBMS assumes that consistency holds for each transaction.
- **Isolation.** Transactions must be isolated from the effects of other concurrently executing transactions. In other words, each transaction must see the system as if there are no other running transactions, i.e., there is no interleaving.
- **Durability.** Once the DBMS confirms that a transaction has been successfully completed, its effects must be permanent on the database, even if some crash occurs.

These properties give a powerful abstraction for developers, which do not have to cope with concurrency issues in their code. However, ensuring them is heavy, especially for consistency, when there is the need for ***scalability*** and ***availability***. ACID properties indeed imply that any read subsequent to a write on a same resource must return the version written by this latter.

Scalability is generally achieved, along with availability, by *replication* and *sharding* (or *partitioning*). At the end, the database system consists of several nodes, with data spread and replicated among them. This give fault-resilience (hence availability) and load balancing capabilities, as read operations can be done on any replicas. The problem in this is to maintain the consistency on these operations: ensure that a read returns *always* the last value written. This is the so-called *write-consistency*, and its implementation in a distributed environment hits the performances, since all read operations should be delayed till the last value written is propagated among all replicas.

Although it would be a desirable property, it is not needed by all datasets. Most of them can work well even if only *eventual-consistency* is ensured, that means that a read operation will return the last written value only since a point in time. Simply

speaking, this allows the propagation of a write to replicas to be done asynchronously, preserving performances.

Generally, document-based database systems only ensure eventual-consistency, since they are designed for distributed environments, like cloud computing applications.

4 STATE-OF-THE-ART ANALYSIS

In this section, we will illustrate a state-of-the-art analysis on software products implementing the technologies introduced ahead.

4.1 Queue Messaging Systems

- **RabbitMQ.** It is an open source Enterprise Messaging Queue server from VMWare, written in the Erlang language. Since it is one of the main implementations of the AMQP protocol (Advanced Messaging Queue Protocol), its architecture is broker-based, so every message is delivered to a central entity (*Exchange*) that routes it to the correct queue.

  RabbitMQ nodes can be organized in clusters, in order to achieve scalability and availability. Queues can be replicated (mirrored) in an active/passive fashion, with one master node and at least one slave node: if the master fails, a new one is elected among all slaves and the system continues to operate. RabbitMQ built-in clustering is not *partition tolerant* (since it ensures *strong consistency* and *availability*), so its usage is preferable on LAN environments, where connectivity between nodes can be considered stable. On WAN environments, RabbitMQ brokers can be distributed on different nodes by using the concepts of *federation* and *shovel*, that basically allows a broker to receive messages published to another broker, connected by the AMQP protocol. Of course, brokers can still be clustered locally. In these scenarios, *partition tolerance* and *availability* are achieved and, hence, only the *eventual consistency* can be globally guaranteed.

  Queues could be made *durable*, so that the containing messages are persisted on disk even in case of failures, on the Mnesia DBMS, written in Erlang as well.

  RabbitMQ runs on all major operation systems and clients for it could be written using several programming languages, such as Java, .NET, Ruby, Python, Erlang, PHP and C/C++. In addition, its setup is fast and easy.

- **Apache QPid.** Part of the Apache Software Foundation, it is an open source Enterprise Messaging system. Like RabbitMQ, it implements the AMQP protocol, so its architecture is broker-based. It provides brokers written in C++ and Java languages, with different features.

  Java Brokers

  Further to standard FIFO queues, also these kind of queues are supported:

- *Priority-Queues*: messages are delivered in the order based on their priority,
- *Sorted-Queues*: messages are delivered in the order based on the value of some message property,
- *Last-Value-Queues*: if two messages with the same key arrive, only the newer is kept, and thus read by clients.

Only two-node clusters are supported, where a node is the master and the other one is a replica. Every operation is performed by clients on the master node that propagates its state to the replica node. Availability of the system is preserved, in case of failures of a single node at once. QPid clustering copes with network partitions, in case of replica being disconnected from the master due to a failure of their network link, however there are some limitations. QPid indeed makes use of a failure detector system that is responsible for detecting node and network links failures. Working on a WAN environment, where latencies of message delivery and stability of connectivity is fairly low, the failure detector could give some *false positives*. This means that, if the master is erroneously considered faulty, the replica node becomes master as well, ending up with a cluster with two masters at the same time, i.e. a *split-brain* situation. QPid does not provide an automatic way to recover from such a situation.

Queues in QPid can be made durable, and their persistence can be configured to be carried on different DBMSs, such as MySQL, Oracle BDB JE and Derby.

C++ Brokers

This kind of broker is much more sophisticated than the Java counterpart. It supports FIFO and LVQ queues, and also active/active messaging clusters, where all nodes have the same queues, exchanges, messages and bindings. A consistent view of the system state is maintained across nodes in the cluster, so every node is actually able to process client requests. Thus, strong consistency and availability are preserved, hence clusters are not network partition tolerant.

Brokers can be connected across WANs by means of *federations*, generally used among clusters configured in high-availability.

QPid runs on all major operating systems and clients could be written using Java, Python, C++, .NET and Ruby programming languages.

4.1.1 Description of the AMQP Protocol

The AMQP Protocol is a networking protocol used in middleware applications to implement message-passing brokers between publishers and subscribers, in a distributed environment.

The main actors involved in the protocol are *Exchanges* and *Queues*. Publishers send messages to an exchange that will route them to the appropriate queues that have previously *bound* to it. Subscribers consume messages directly from queues.

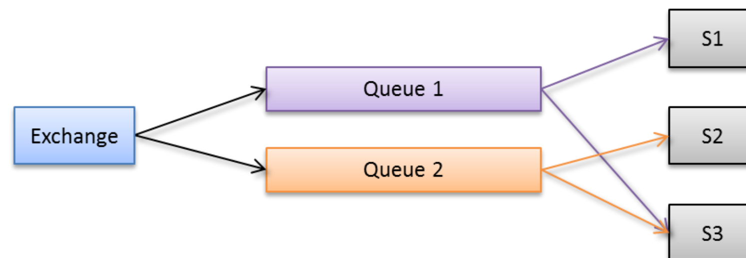


Figure 3 - AMQP example scenario

In figure above, messages arriving at the exchange are pushed into Queue 1 and Queue 2 based on some routing criteria defined on the exchange:

- **Direct.** Routing is based on the equality a key-string defined on messages.
- **Fan-out.** Messages are broadcasted to all queues, regardless any routing key.
- **Topic.** Similarly to direct, routing is based on the matching of a specific pattern on the routing key.
- **Headers.** Routing is based on multiple attributes defined as message headers.

Moreover, an exchange can be *durable* so that it will survive to a broker restart, or *transient*. They can also be configured to be deleted automatically by the system whenever all queues have finished using it.

Queues store messages that are consumed by applications. They have a name and some additional properties. As well as exchanges, they can be durable or transient and can be configured to be deleted when there are no active subscribers anymore. Furthermore, they can be created as exclusive queues, so that they can be used by only one connection and hence the queue will be deleted when this connection closes.

Exchanges and queues must be *declared* to the broker by issuing appropriate AMQP commands. After declaration, queues are bound to the exchanges that will use them, by issuing *binding* AMQP commands.

Reliable delivery could be implemented using an acknowledgment-based mechanism, where the broker is instructed to remove messages from queues only after the client sends back an acknowledgment. Multiple message acknowledgment can be implemented as well.

In case a client has successfully received a message, but it's not able to process it at that time, it can reject the message, by issuing a negative acknowledgment. However, a reject message cannot be issued for a set of messages. This is left to specific protocol implementations.

Connections with brokers are opened on top of TCP for reliable communications. Optionally, they can be secured using SSL. From the user point of view, a connection with a broker is a channel that is an abstraction of the real network connection. The

use of virtual channels instead of real TCP connections makes possible to have different logical connections on top of a single real one, letting clients to have multiple connections with brokers while using system resources for a single one.

Furthermore, brokers can host different logical environments, each one with its queues and exchanges, by using the concept of *virtual hosts*, very similar to the ones used by web servers.

4.2 REST vs. SOAP Web Services

REST and SOAP are two approaches for creating web services, each of having its advantages and disadvantages.

The main difference between them is that SOAP is a W3C standard, while REST is rather a collection of standards used together.

First of all, the access to REST web services is done by using standard URIs on HTTP/HTTPS, hence a web service call can be performed using any web-browser. On the other hand, making a SOAP call is not that easy, while it gives the possibility to use any transport protocol other than HTTP.

For data representation, SOAP relies on XML for messages representation, including their payloads. REST is more flexible in this sense, since it is not strictly associated with any representation language, although JSON is generally used. XML has the clear disadvantage to add overhead to information, while JSON does not. On the other hand, XML allows to have rigid specifications for exchanged types.

The use of formal contracts on data leads to reduce the maintainability of code, since whenever the contract changes, clients have to be reconfigured in order to be able to re-issue invocations.

REST communications can be secured by using HTTPS instead of HTTP. More security and encryption options are possible with SOAP.

The common scenario for REST employment is when totally stateless operations are needed, and caching facilities could be desirable. Of course, it use less bandwidth than SOAP and makes very easy the integration with commonly used web technologies, like AJAX. SOAP, on the other hand, offers the possibility to implement stateful operations with formal contracts on data structures on both sides of the communication. Moreover, SOAP allows the processing code to be executed asynchronously from the service invocation.

4.3 Document-based Database Management Systems

- **MongoDB.** It is the leading of document-based NoSQL database management systems, open source and written in C++. It is used by several applications, from content management, such as *SAP*, *SourceForge* and *Wordnik*, to news and media websites, such as *TheGuardian*, *Forbes* and *The New York Times*.

  It stores documents using a JSON-like format, offering dynamic schemas, as the type of DBMS can provide, and a full support for indexes, as they can be

created on any attribute of documents. Several indexes are available: single-key or multi-key, sparse or dense, hash-based or tree-based.

MongoDB supports transactional operations only on a single document. This means that any operation performed on a single document can be considered atomic. Concurrency control is performed by a *readers-writer locks* mechanism. Operations that involve different documents cannot be executed atomically with an out-of-the-box functionality. However, a *two-phase commit* approach could be implemented by the user to achieve transaction-like semantics.

Large aggregation tasks can be executed used a built-in *map-reduce* programming paradigm.

MongoDB is well suitable for applications that run on distributed systems, where there is the need for scalability as well as fault-tolerance. The concepts of *replication* and *sharding* are used. With the former, a single MongoDB node (called *primary*) is replicated on a number of other nodes (called *secondary*). Writes are all directed to the primary, while reads can be executed on any node within the replica set. The state of the primary node is reflected to the secondary ones asynchronously. MongoDB can be configured both to allow read operations only on the primary node, and to allow them on any node of the replica set. In the former case, *strict-consistency* is achieved, while in the latter only the *eventual-consistency* can be ensured.

Sharding refers to data partitioning as a way to achieve scalability, balancing data and load among different machines. With this methodology, a document collection is divided into *shards*, each one stored on a different machine. Write capacity can be increased, since different write operations can be executed on different machines at the same time.

Replication and sharding are generally used together to achieve high-availability (fault-tolerance) and scalability. A common approach is to shard collections of documents across a set of machines, and then replicate each of them. We achieve thus availability of the overall system. Machines are not constrained to be placed in the same data centre, since mirroring and sharding are possible also across WANs, preserving *partition-tolerance*. Split-brain situations are avoided by the implementation of a pessimistic approach: whenever a network partition is detected, only one partition remains available and continues to operate accepting read and write operations, while the others become unavailable. The survival partition is chosen carrying out a *quorum-based consensus*.

MongoDB has a very good documentation, and it can be used easily with almost any programming language using its drivers.

- **CouchDB.** Part of the Apache Software Foundation, is an open source document-based database management system, written in the Erlang language.

  Documents are represented in a JSON-like format and read and update operations on them can be issued by means of a RESTful HTTP API.

CouchDB is based on a Multi-Version Concurrency Control model, where read operations are never blocked and a new version of a database object is created every time that object is written.

Database replication is possible by means of a special DB, called “replicator”, which manages the replication of objects between two databases. Anyway, replication is intended to be used only for server clustering, in order to achieve some form of scalability on read operations.

At the moment, CouchDB is not well suited for distributed applications, since it does not support sharding and partition-tolerance.

Clients for CouchDB are available in several programming languages.

- **RavenDB.** A document-based database management system, open source, written on the .NET framework.

  It supports ACID transactions both on single and multiple documents operations. Scalability and Availability are achieved by the support of replication and sharding.

  The map/reduce programming technique can be used for complex aggregation tasks on data.

  Data is accessed by a RESTful HTTP API, and clients are available for .NET, giving the possibility to run LINQ queries on a database. Furthermore, it can be embedded easily in any .NET application.

5 ARCHITECTURE SPECIFICATIONS

The architecture will allow Traffic Control Systems to be integrated with Optimization Modules and Driver Advisory Systems.

A Traffic Control System could consist of several server units, each one belonging to different functional areas, such as commercial, traffic management, data storage and analysis. Generally, a TCS could be seen as a distributed environment, where different functional units can reside in different geographic areas. Furthermore, a TCS collects real-time data from trains on the field, by different possible communication media.

Driver Advisory Systems communicate with TCS and sends data to running trains, either directly or indirectly, passing through the TCS.

Optimizations Modules integrate within the architecture by means of the SOA architecture given by WP7.

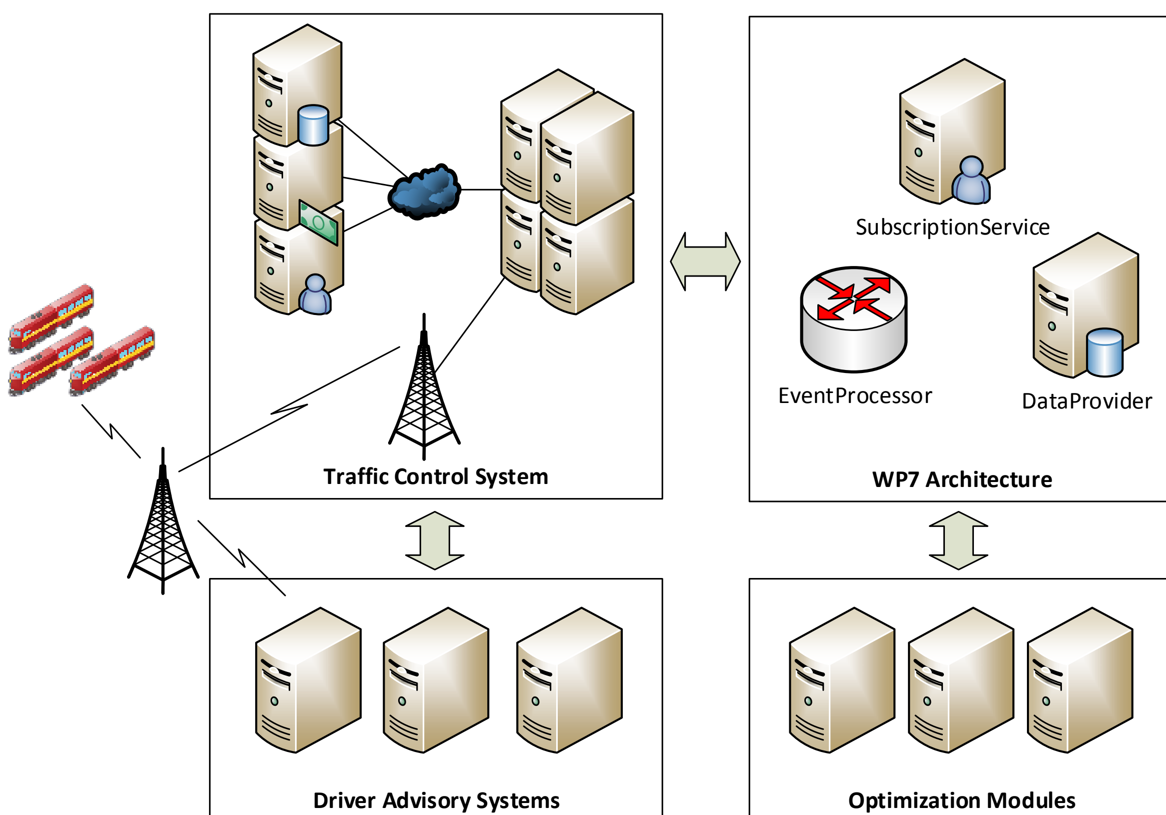


Figure 1 - General architecture scenario

As showed in the figure above the implementation of the WP7 architecture it is composed by three different nodes:

- **Subscription Service Node**

  This service will be provided by a custom Web Service written in .NET. The We Service model will guarantee the opportunity for other actors to call the service regardless of the technology used to implement them.

Before be able to process events, actors must register to the Subscription Service, indicating whether they are going to send or receive events from the architecture.

The service can accept or reject the requests basing on several criteria: formal correctness of the request, security policies and so on.

The DataProvider module will store subscriptions and un-subscriptions.

- **Event Processor Node**

  This layer will be responsible for receiving event messages from publishers and delivering them to subscribers, in a reliable way.

  RabbitMQ will be used to implement the MoM (Message Oriented Middleware) using reliable and durable message queues.

  Reliability is achieved by using RabbitMQ built-in clustering on LAN environments, hence replicating message queues on different nodes.

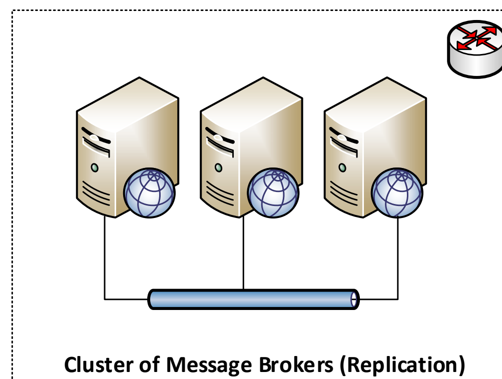


Figure 2 - Replicated AMQP Queues

If the system load requires the WP7 MoM to scale up, more clusters of message brokers can be added to the scenario, connecting them through a WAN network, hence on different geographic areas.

Using *federated* message brokers, we safely avoid the occurrence of split-brain situations.

Different subscriber clients can access the WP7 MoM through any Event Processor in the network, achieving thus load balancing.

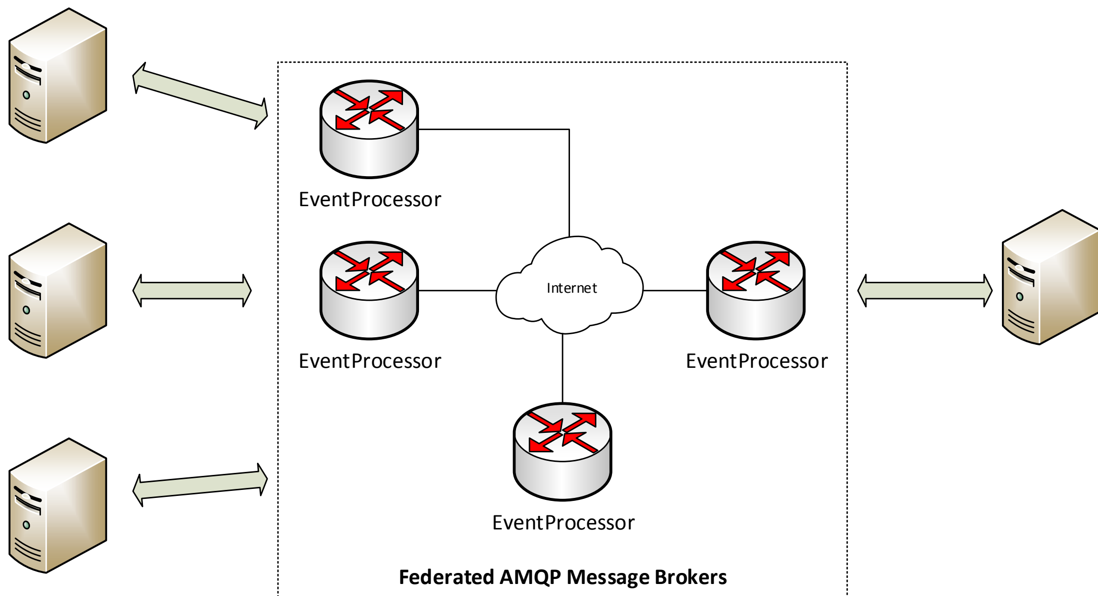


Figure 3 - Distributed Event Processors

- **Data Provider Node**

  The DataProvider node will be implemented using the MongoDB database management system. Both stand-alone and distributed solutions are possible.

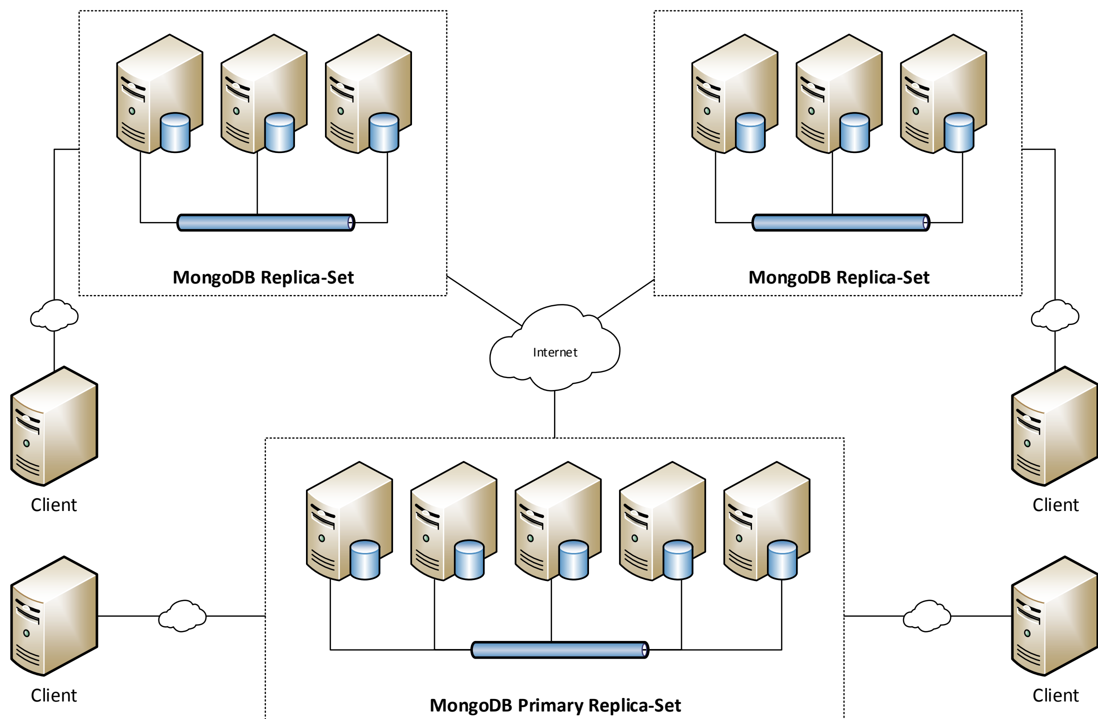


Figure 4 - MongoDB distributed environment

Similarly to concepts used in the EventProcessor node specifications, reliability in the DataProvider node is achieved using the MongoDB replication scheme.

In this way, different clusters of MongoDB node are connected each other using a WAN network, achieving hence scalability.

In order to let the system work in presence of network partitions and failures, one out of all MongoDB clusters (replica sets) must contains more nodes than others, in order to maintain a *majority* within it. In this way, MongoDB internal quorum-based algorithms can work safely on a WAN environment.

The Data Provider node will contain all subscription data, updated by the Subscription Service. The Event Processor will use those data to dispatch messages to correct recipients. Furthermore, it will contain the event catalogue, i.e. the list of possible events managed by the WP7 infrastructure.

Train-related data will be not stored by the Data Provider node; accesses to those information are redirected to the TCS. The architecture will act as a proxy for such data requests, abstracting hence an interface to the TCS.

5.1 Possible architecture scenarios

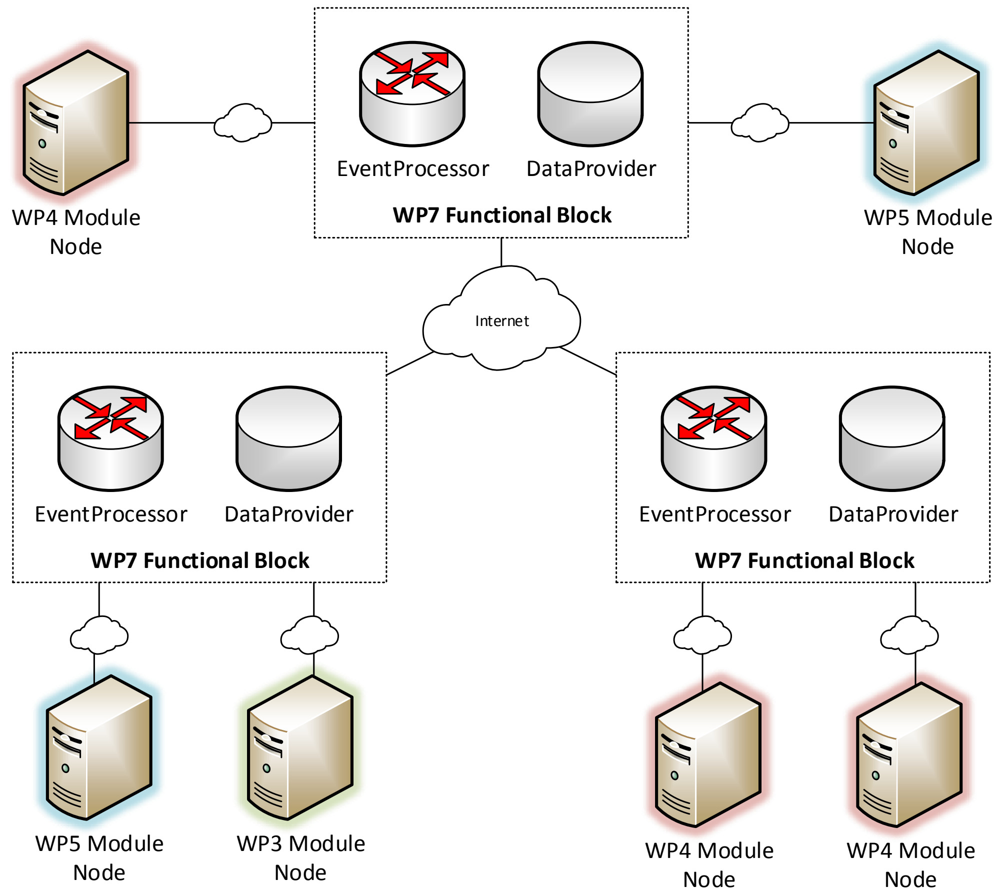


Figure 5 - Example scenario with distributed modules

From their side, optimization modules could be distributed as well, meaning that a single module could consist of several computational units that have different roles, even spread on different geographic areas.

Note that different nodes of a single module can communicate each other by means of the WP7 architecture, using appropriate event definitions and subscriptions.

This scenario is feasible within the WP7 architecture, achieving a **fully-distributed** system.

5.2 Component-based view of the architecture

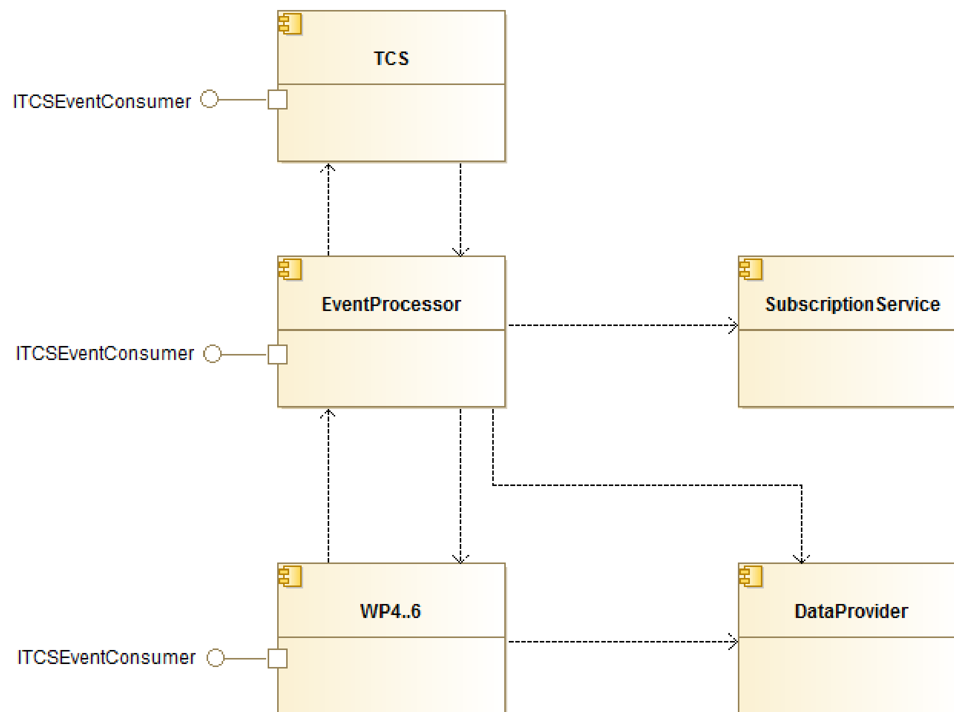


Figure 6 - Component-based view of the architecture

- **EventProcessor:** it will communicate with the TCS and WPs. During the dispatch of messages, it uses the SubscriptionService to access subscription data, and the DataProvider component to access data storage.

- **DataProvider:** provides a read-only data access to published data structures, updates to these structures are performed by special events generated by the optimisation modules concerned.

- **SubscriptionService:** manages the list of subscribers to specific event types within the architecture. The component will also provide the infrastructure to define routing rules (determine the services to which an event has to be forwarded). The component will store:
    - A list of event consumers. Every consumer registers specifying a list of events (defined by TypeId and Version number) it is subscribing to;
    - A list of event publishers. Every publisher will specify a list of events (again defined by TypeId and Version number) that is going to provide.

6 DATA AND EVENT MODEL

6.1 Published Data Entities

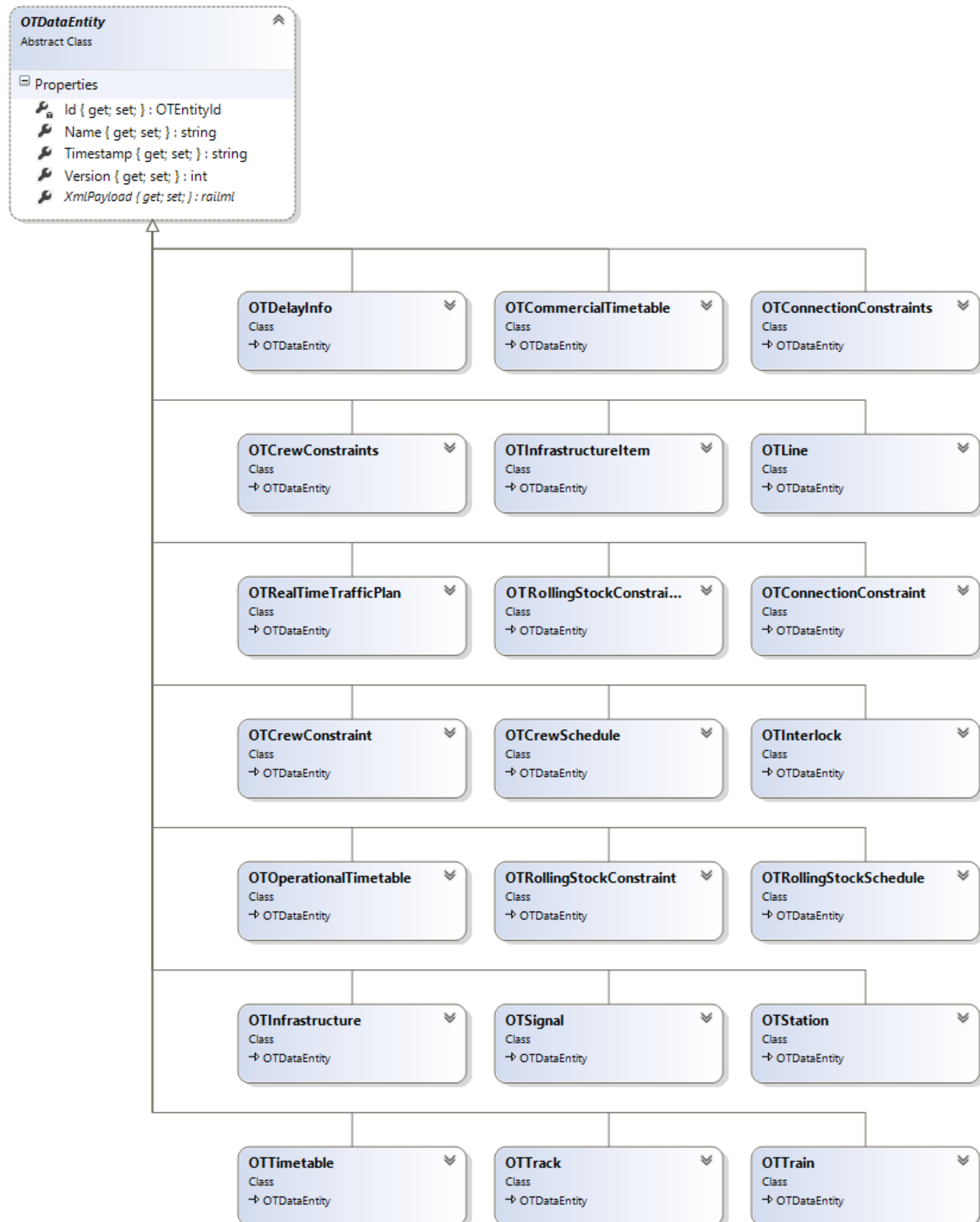


Figure 7 - Data Entities as they are published by the DataProvider Service

Since most of the data entities provided within the architecture are subject to updates in response to changes in the real-world state of the network, or the completion of

processing tasks by optimisation modules, the architecture will provide a way for modules to consume these data structures on-demand. To this end, the architecture will maintain an interface to a directory of latest versions / last valid releases of all published data entities. Updates will be performed via messages from data publishers. This approach will be of greatest importance for data entities that need to be periodically updated during a session (e.g., timetables).

The system will be based on published dates: whenever an optimization module will provide a new version of a data structure, the architecture will store it and will publish it for consumption by other services.

These data interface are read-only, scenario-based data access modules and will not be updatable from the DataProvider itself.

To update scenario data structures, the architecture need to receive an update event from the subscribers allowed to submit it.

For details about the ON-TIME data model, please refer to *D7.1 - Library of Data and Communication Models.*

6.2 Event Model

Figure 8 - Events dispatched and consumed by the EventProcessor Interface

Owing to the dynamic nature of the Train Management Systems that ON-TIME will need to interface to, the architecture will provide a middleware responsible for the collection, dispatch and classification of events.

Every event is composed by a payload, defined in XML (using RailML or a custom representation, backed-up by a proper XSD schema) and a series of attributes that helps the infrastructure and the consumers to quickly understand the nature and the domain of the information.

The classification of events allows event consumers to subscribe to specific class of events only, easing network loads and simplifying the integration patterns.

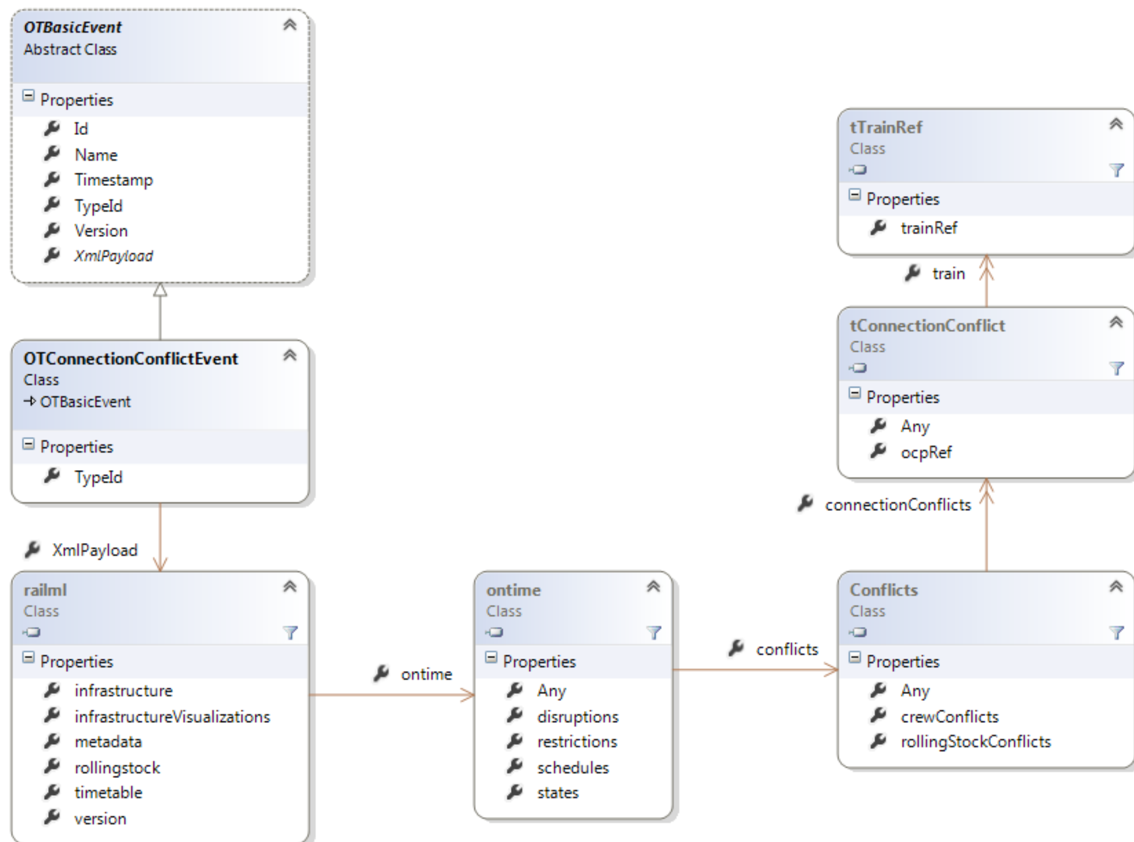
An event will be composed of the following attributes:

- **Id**: Unique Id of the Event
- **TypeId**: Unique Id that will classify the event
- **SenderId**: Unique Id of the sender of the message. Useful for routing and security
- **Version**: Version number of the event
- **Name**: Human-Readable name of the event
- **Timestamp**: the event timestamp
- **XmlPayload**: and XSD-backed XML fragment that does describe the event and contains information.

Note: to ease extensibility, the architecture will manage events in an agnostic way. Specifically, the contents of the events will not be considered as part of the routing task, instead classified will be based purely on TypeId and Version.

For a detailed data dictionary of the event model and RailML, please refer to *D7.1 - Library of Data and Communication Models.*

6.2.1 ConnectionConflictEvent

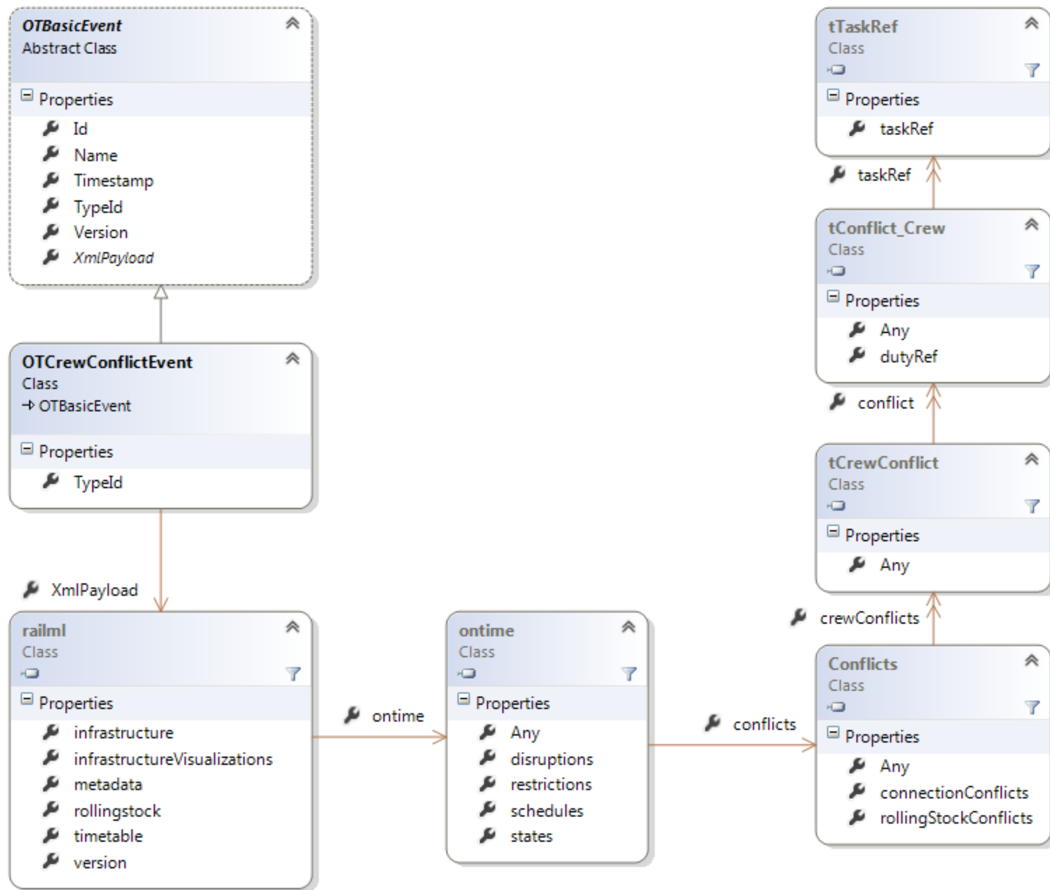


TypeId	{9F9D1034-E31B-4160-9632-06262877BF02}
Created By:	WP4
Consumed By:	WP5
Description:	Notifies that there's a connection conflict in the planning of several resources.
Data Payload	▪ List of TrainIDs affected by the conflict ▪ List of StationIDs affected by the conflict
Data Model Reference	CONNECTION_CONFLICT_EVENT = ENTITY_HEADER + { CONFLICTING_TRAIN } + { CONFLICT_TIME } + { CONFLICT_LOCATION }

6.2.2 ConnectionScheduleAvailableEvent

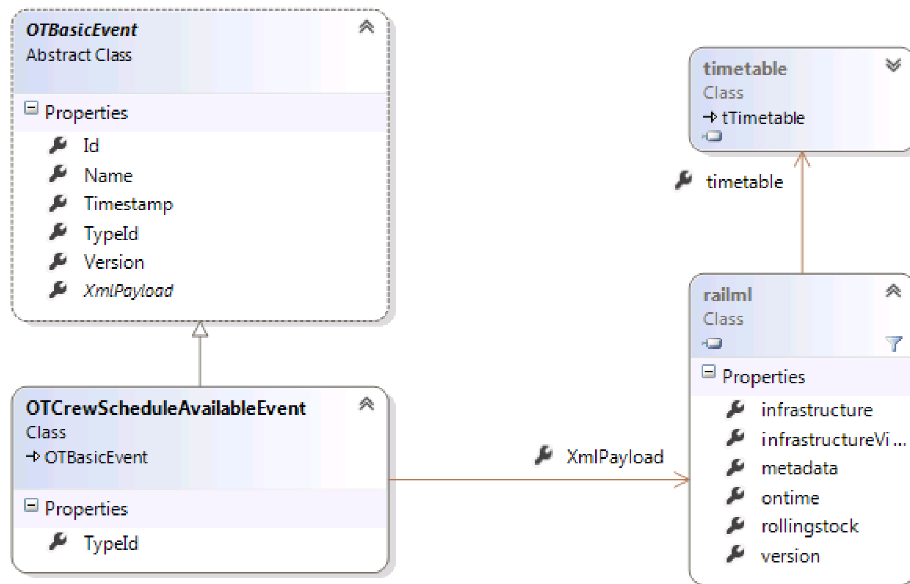
TypeId	{739B5024-4E67-492F-A75B-393F62615EFE}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that a new ConnectionSchedule has been published to the architecture and can be retrieved as Static Data via the Data Provider Interface.
Data Payload	▪ TrainIDs of the trains affected
Data Model Reference	CONNECTION_SCHEDULE_AVAILABLE_EVENT = ENTITY_HEADER + TRAIN_ID

6.2.3 CrewConflictEvent



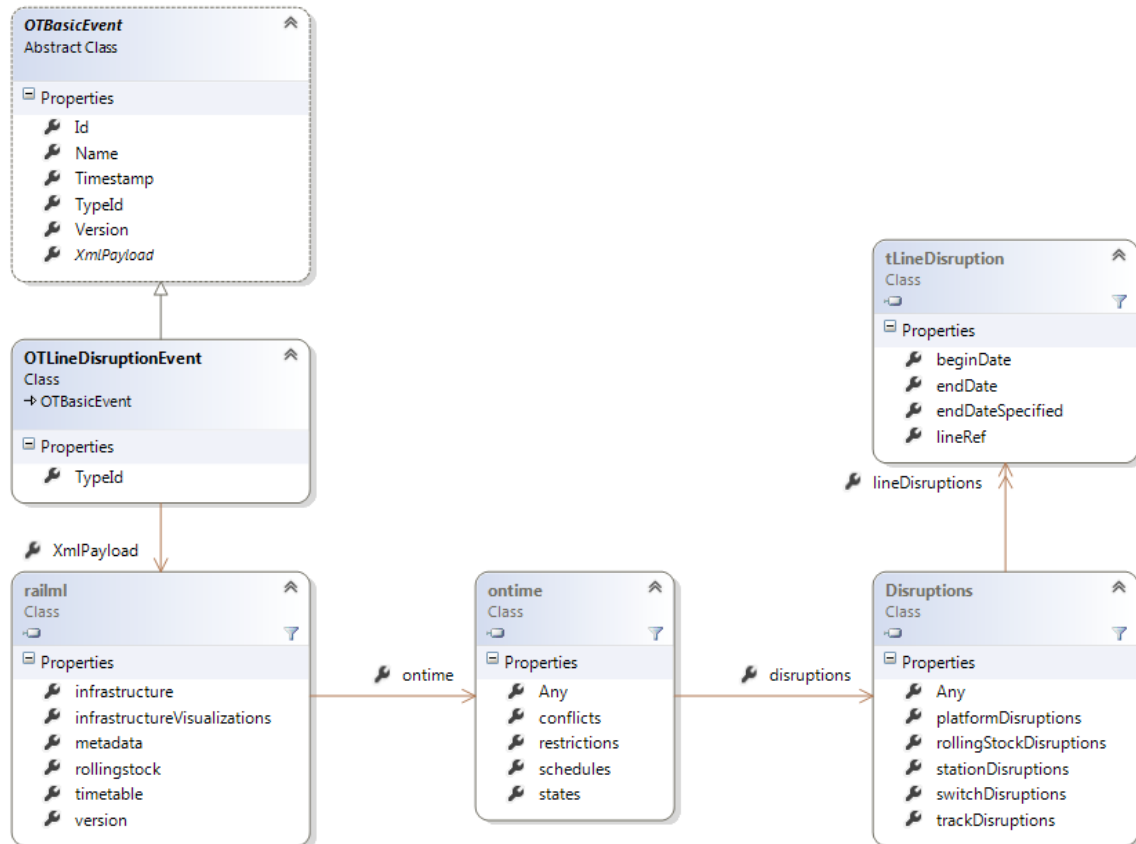
TypeId	{B6B1E7B5-991A-47F5-AE59-57329F0F3D11}
Created By:	WP4
Consumed By:	WP5
Description:	Notifies that there's a crew conflict arose during the re-planning of the Real Time Traffic Plan.
Data Payload	▪ List of Crew IDs that are in conflict with each other
Data Model Reference	CREW_CONFLICT_EVENT = ENTITY_HEADER + { CREW_DUTY_ID + { CREW_TASK_ID } }

6.2.4 CrewScheduleAvailableEvent



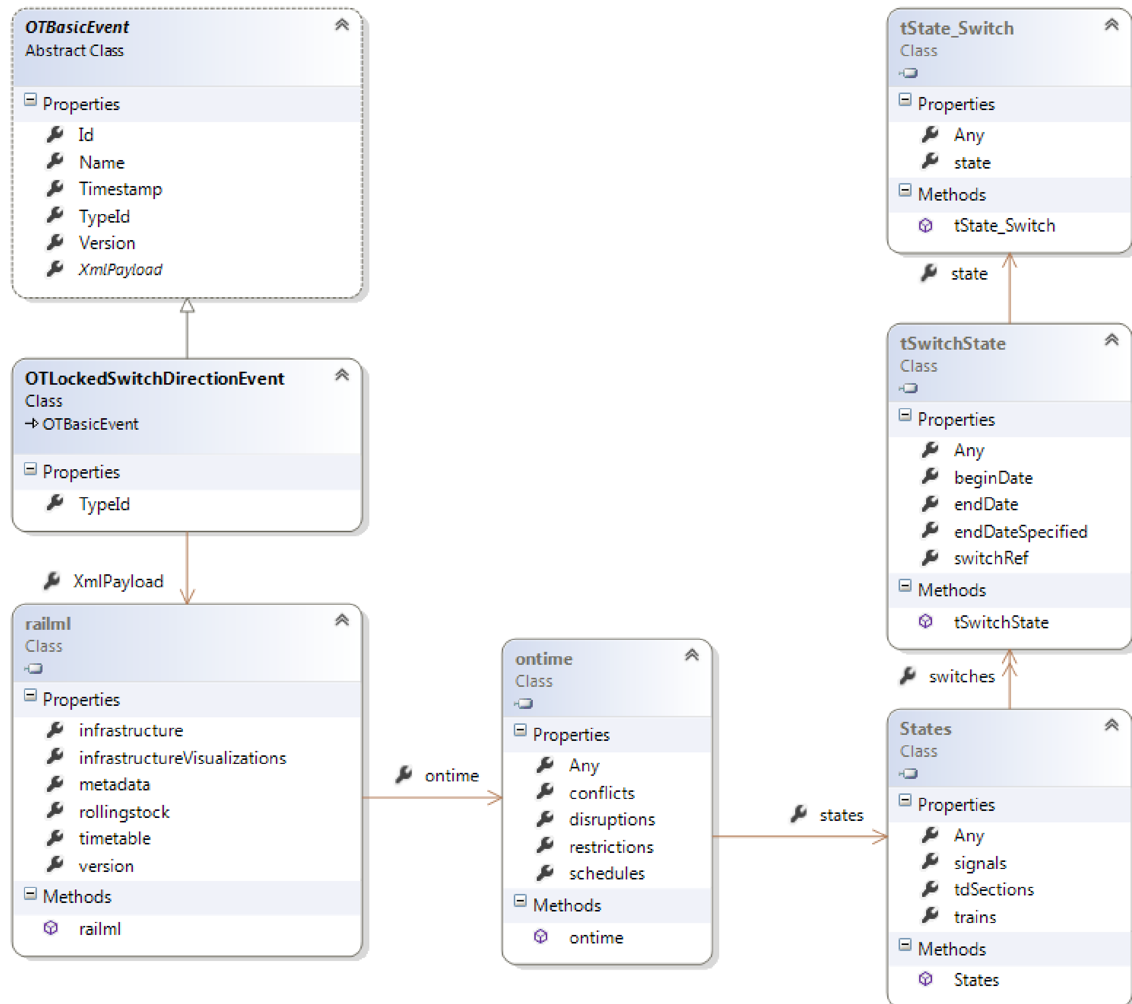
TypeId	{812B3FAB-809D-4E79-89D1-EA6888FCA6CA}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that a new CrewSchedule has been published to the architecture and can be retrieved as Static Data via the Data Provider Interface.
Data Payload	▪ ID of the timetable
Data Model Reference	CREW_SCHEDULE_AVAILABLE_EVENT = ENTITY_HEADER + TIMETABLE_ID

6.2.5 LineDisruptionEvent



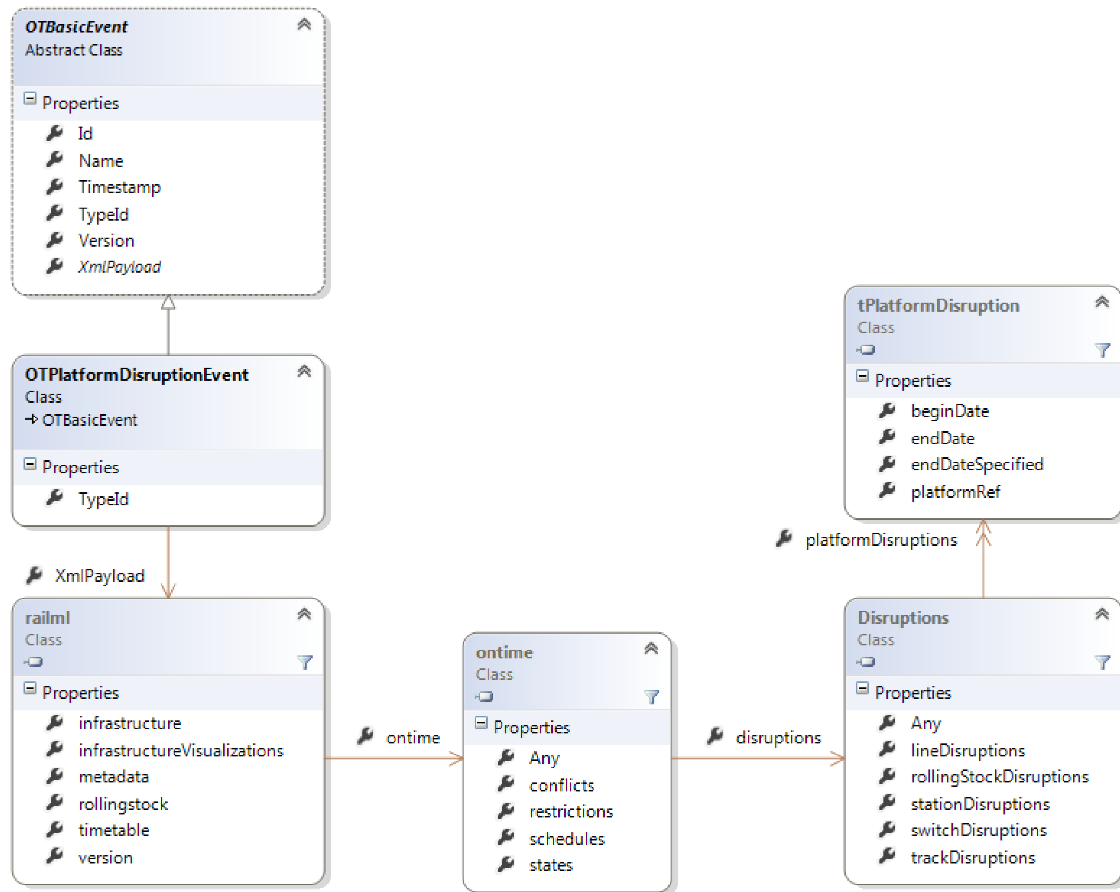
TypeId	{33FFEC5D-2B42-4294-BFB1-4463C37E7675}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that there's a line that has a disrupted.
Data Payload	▪ ID of the RailML Line representation ▪ Starting time of the disruption ▪ Time at which the disruption will end (for planned operations) ▪ Reason of the disruption
Data Model Reference	LINE_DISRUPTION_EVENT = ENTITY_HEADER + LINE + TIME_STAMP + (TIME_STAMP) + { 0-9 | A-z | . | , }

6.2.6 LockedSwitchDirectionEvent



TypeId	{9F32D213-5C9E-4419-9332-D82A606C7C06}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies a system that a switch has been locked to a direction
Data Payload	▪ ID of the switch ▪ Switch direction
Data Model Reference	LOCKED_SWITCH_DIRECTION_EVENT = ENTITY_HEADER + SWITCH_ID + SWITCH_DIRECTION + TIME_STAMP

6.2.7 PlatformDisruptionEvent

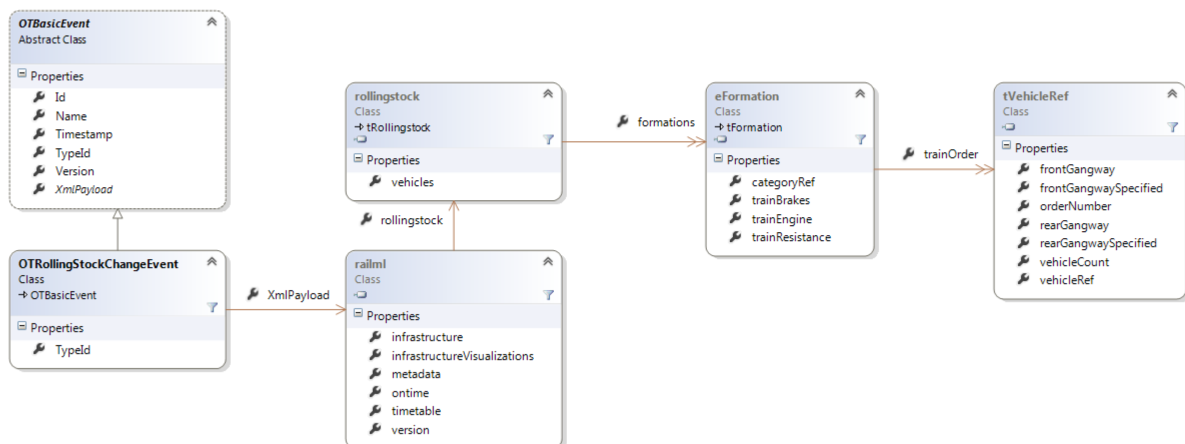


TypeId	{9BD541BD-3741-4122-83B7-7067B08E5F9B}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies the disruption of a platform
Data Payload	▪ ID of the Station containing the Platform ▪ ID of the Platform ▪ Starting time of the disruption ▪ Time at which the disruption will end (for planned operations) ▪ Disruption reason
Data Model Reference	PLATFORM_DISRUPTION_EVENT = ENTITY_HEADER + STATION_ID + PLATFORM_ID + TIME_STAMP + (TIME_STAMP) + { 0-9 | A-z | . | , }

6.2.8 RealTimeTrafficPlanAvailableEvent

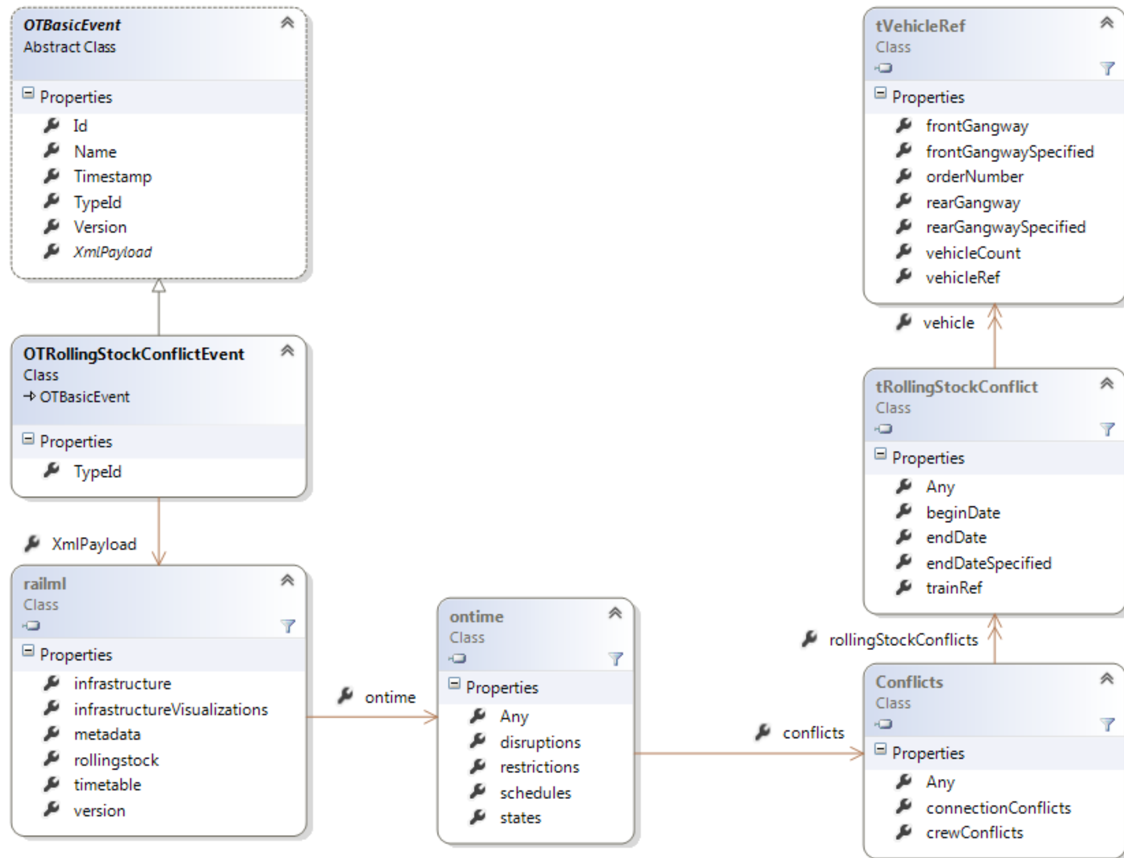
TypeId	{F8AB774F-4473-4655-8226-87EFE49BB74E}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that a new RealTimeTrafficPlan has been published to the architecture and can be retrieved as Static Data via the Data Provider Interface.
Data Payload	No data
Data Model Reference	REAL_TIME_TRAFFIC_PLAN_EVENT = ENTITY_HEADER

6.2.9 RollingStockChangeEvent



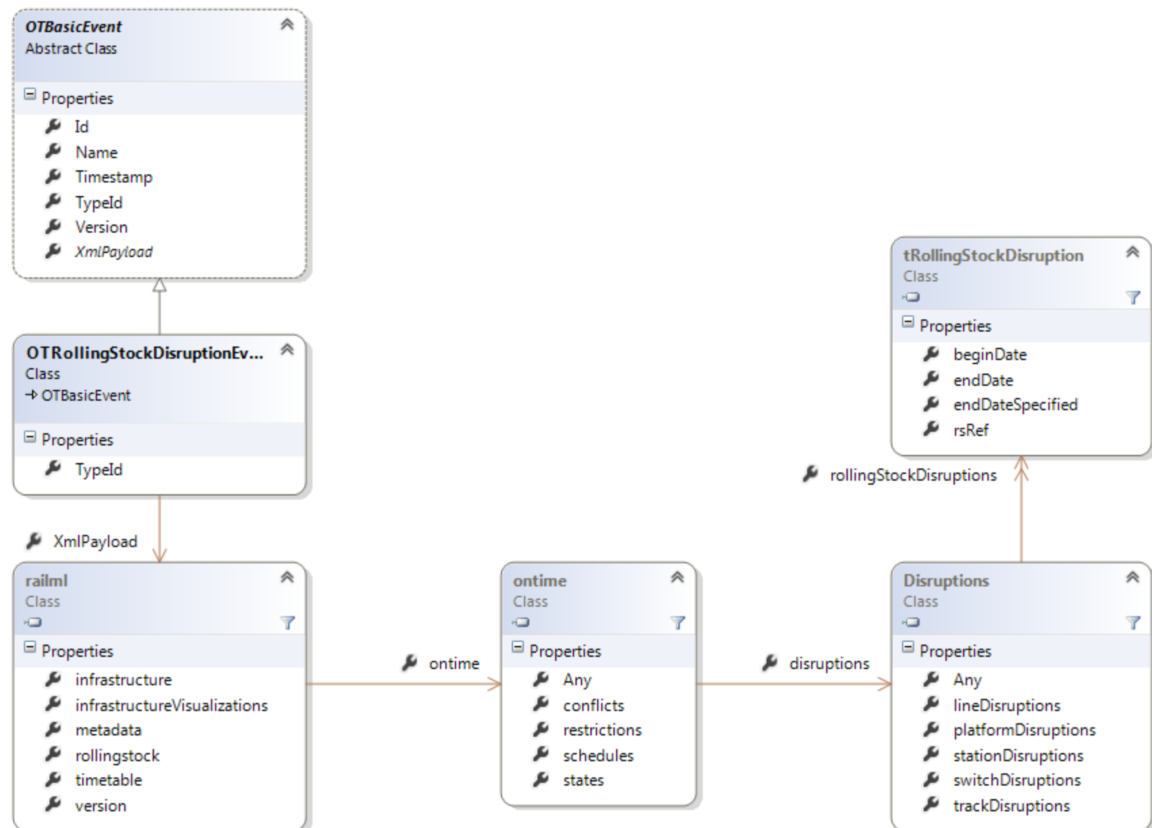
TypeId	{B4AED5E5-1667-442F-A02B-0FE6E5255B61}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that a Train has been incurred in a change of its composition
Data Payload	▪ ID of the Train ▪ RailML details of the new train composition
Data Model Reference	ROLING_STOCK_CHANGE_EVENT = ENTITY_HEADER + TRAIN_COMPOSITION

6.2.10 RollingStockConflictEvent



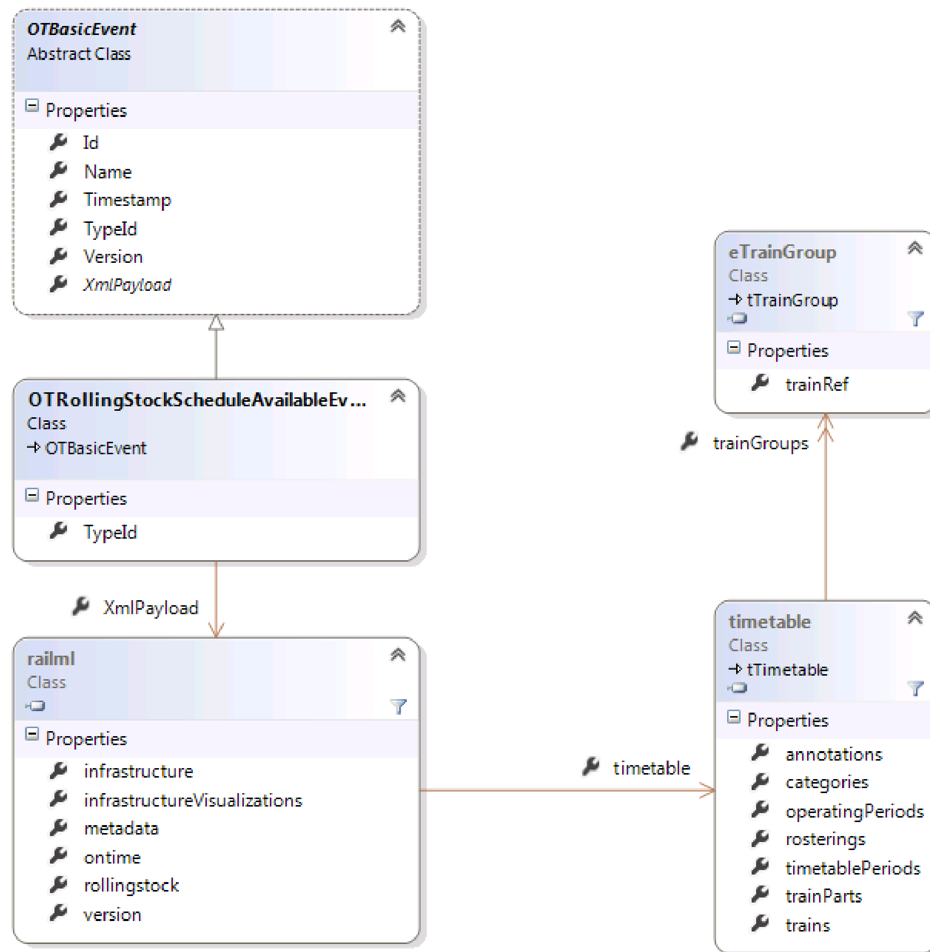
TypeId	{6F529394-A7DF-4C00-865F-725771CCC35C}
Created By:	WP4
Consumed By:	WP5
Description:	Notifies that a Train has been incurred in a rolling stock conflict.
Data Payload	▪ ID of the Train ▪ ID of the conflicting rolling stocks
Data Model Reference	ROLL-ING_STOCK_CONFLICT_EVENT = ENTITY_HEADER + CONFLICTING_ROLLING_STOCK

6.2.11 RollingStockDisruptionEvent



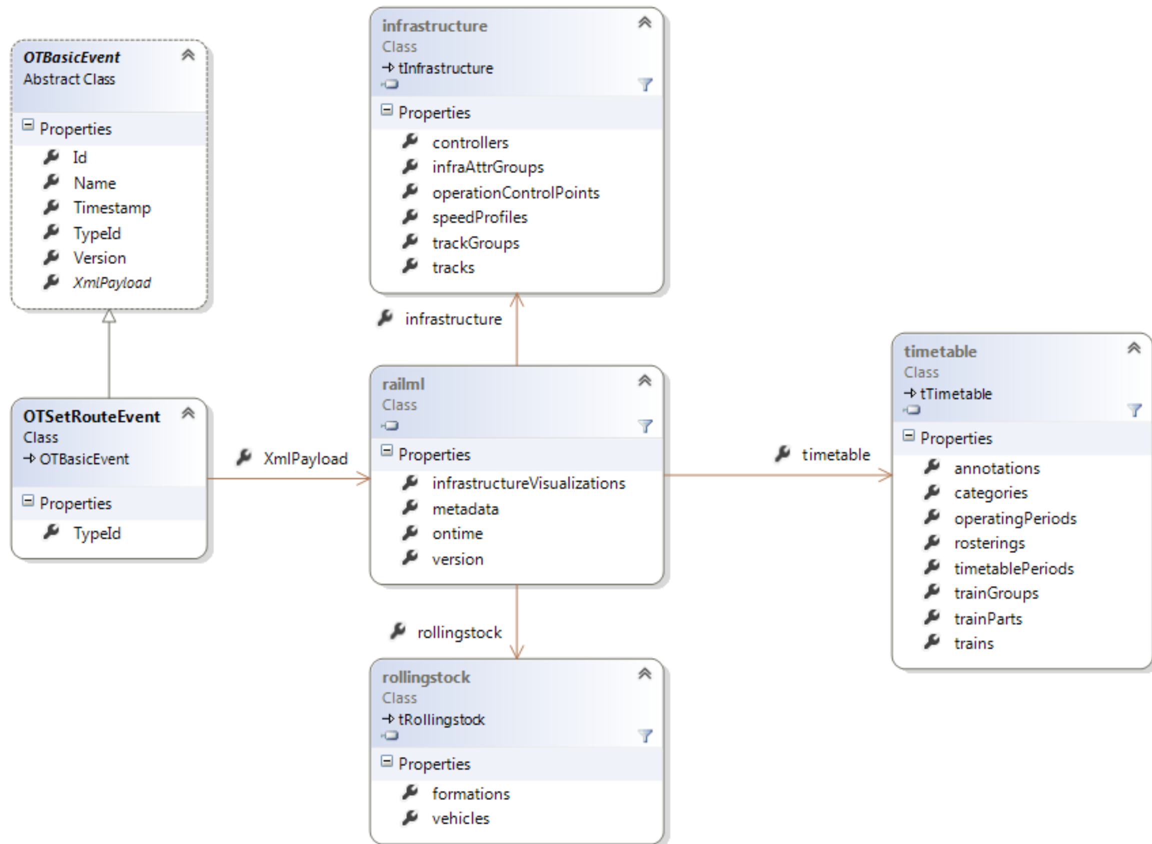
TypeId	{ED672C7E-D4A0-46D9-A07A-E8F68936244D}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that a train had a rolling stock failure.
Data Payload	▪ ID of the Train ▪ ID of the Vehicle that has incurred in failure ▪ Details about occurred failure (type, severity, etc.)
Data Model Reference	ROLL-ING_STOCK_FAILURE_EVENT = ENTITY_HEADER + TRAIN_ID + VEHICLE_ID + { ROLLING_STOCK_FAILURE_TYPE + ROLLING_STOCK_FAILURE_SEVERITY + (ROLLING_STOCK_FAILURE_LOCATION) }

6.2.12 RollingStockScheduleAvailableEvent



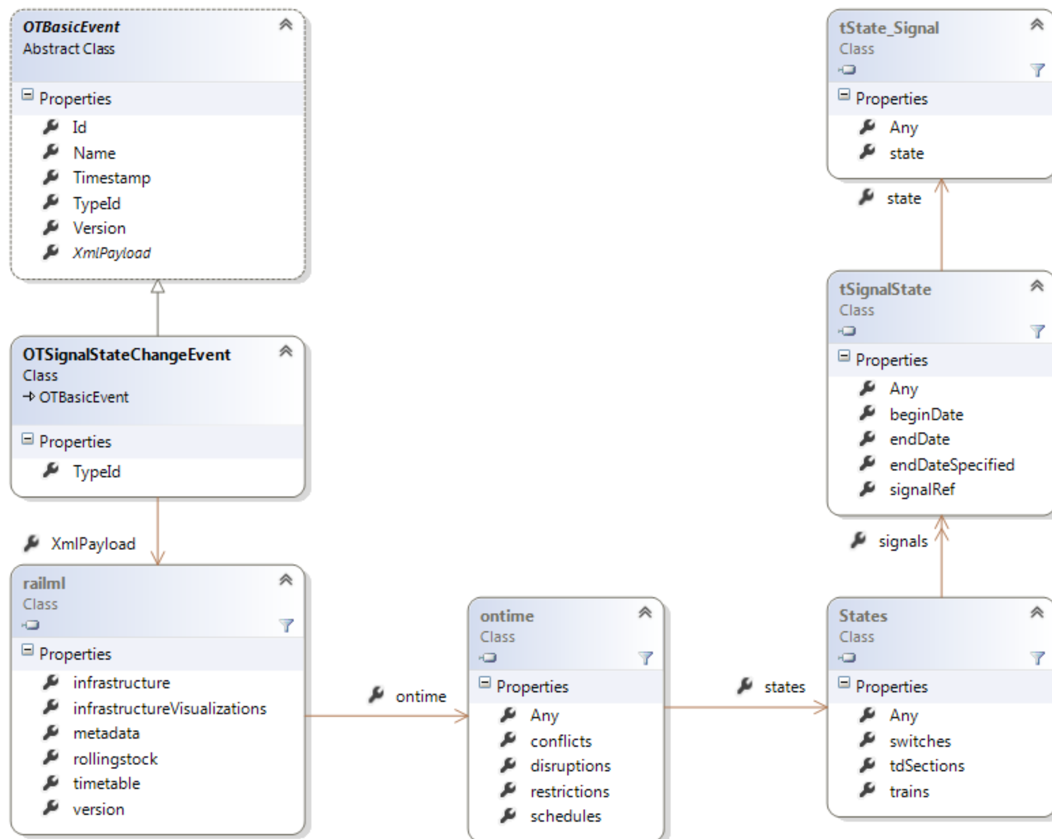
TypeId	{A900FA23-0161-4C14-9697-6F7C91F46E9B}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that a new RollingStockSchedule has been published to the architecture and can be retrieved as Static Data via the Data Provider Interface.
Data Payload	▪ List of TrainIDs affected by the new schedule
Data Model Reference	ROLLING_STOCK_SCHEDULE_AVAILABLE_EVENT = ENTITY_HEADER + {TRAIN ID}

6.2.13 SetRouteEvent



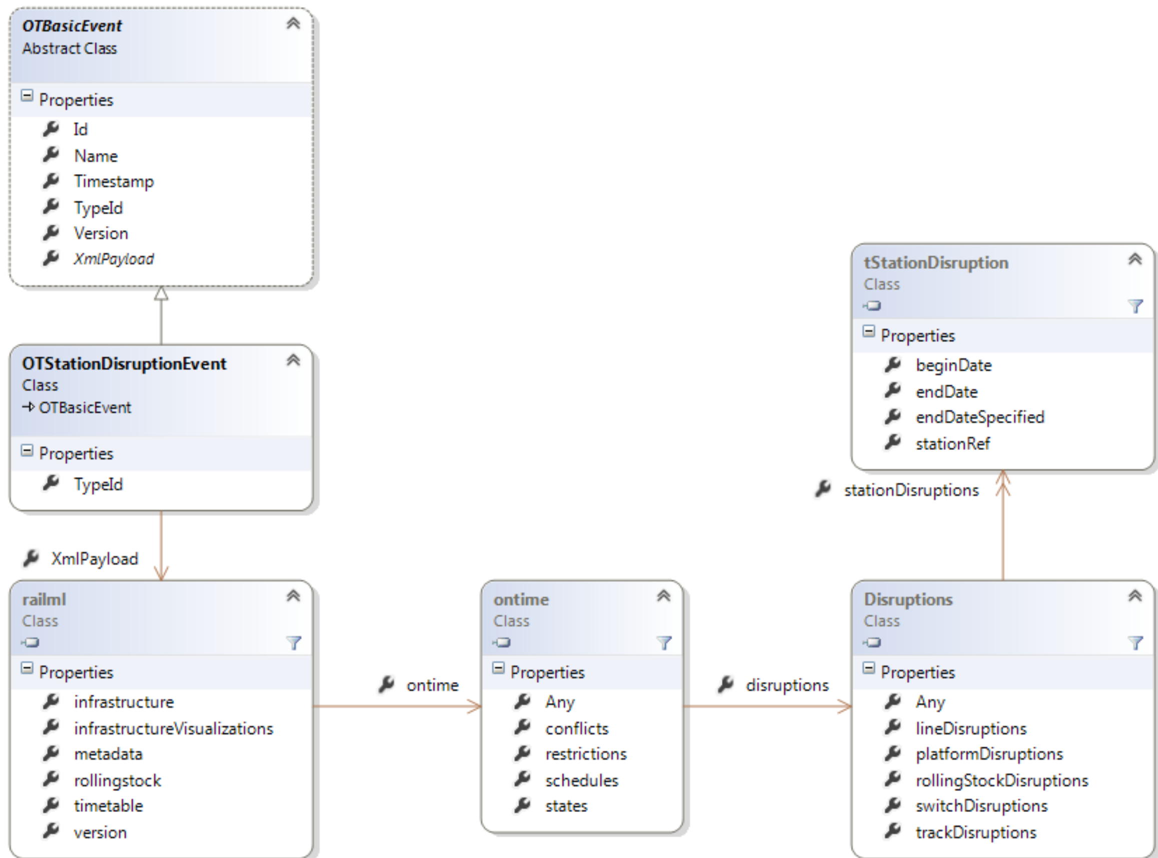
TypeId	{190C587D-D782-422A-AB3C-F876893CAD6E}
Created By:	WP4
Consumed By:	TCS
Description:	Notifies that a new route has been set.
Data Payload	▪ Route representation
Data Model Reference	SET ROUTE EVENT = ENTITY_HEADER + ROUTE_DETAILS

6.2.14 SignalStateChangeEvent



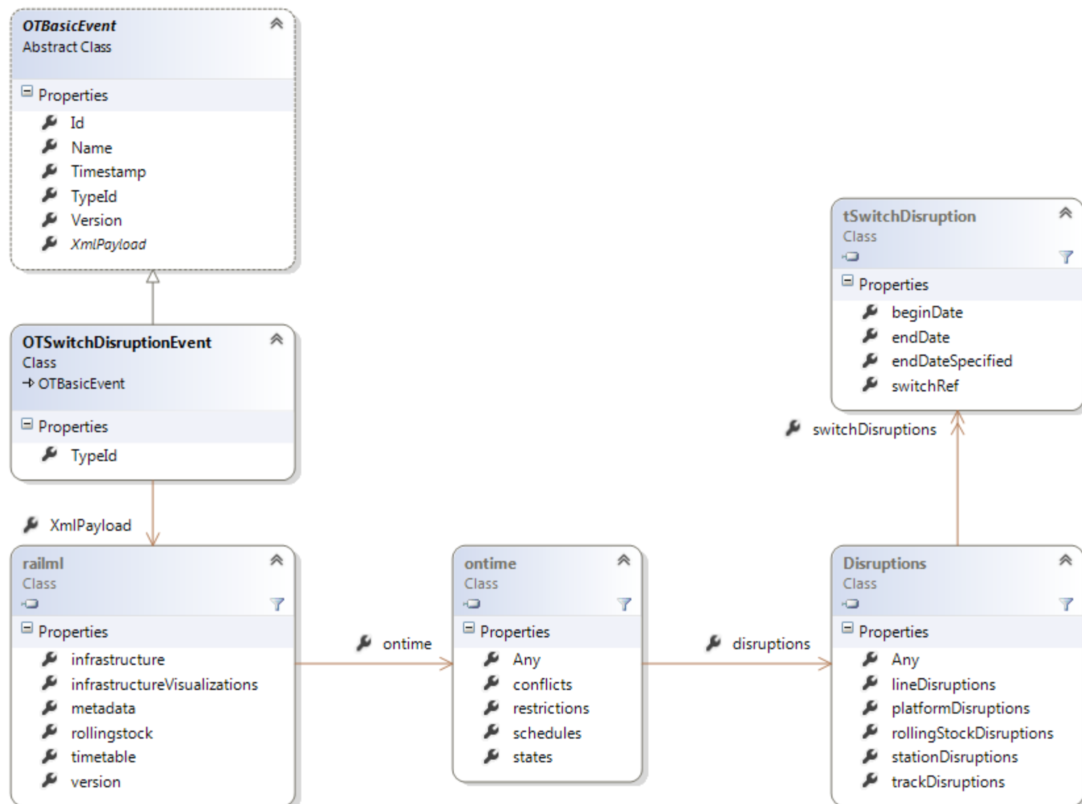
TypeId	{FE543362-1140-4F46-A30A-57666C48AA52}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies that a signal has changed its state.
Data Payload	▪ ID of the signal ▪ Signal aspect
Data Model Reference	SIGNAL_STATE_CHANGE_EVENT = ENTITY_HEADER + SIGNAL_ID + SIGNAL_ASPECT + TIME_STAMP

6.2.15 StationDisruptionEvent



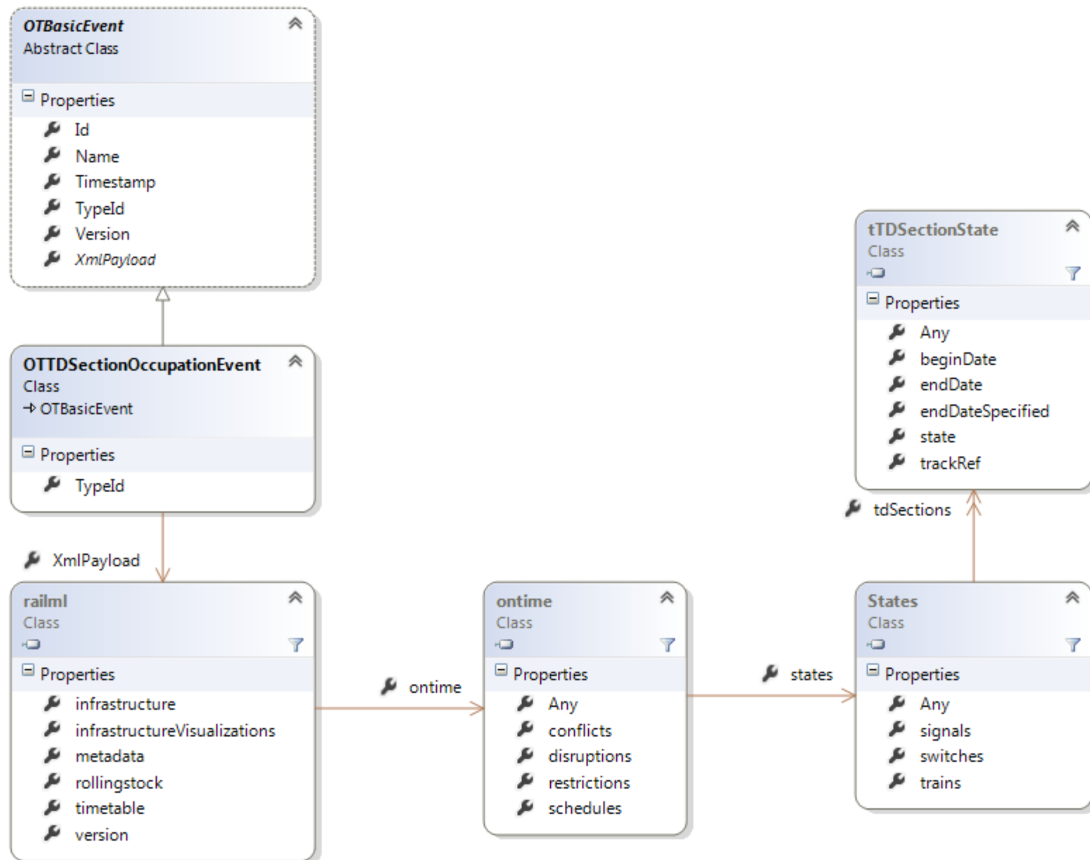
TypeId	{0304D51D-FCC3-4CFB-93C1-9C0DD408C815}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that there's a station that has been disrupted.
Data Payload	▪ ID of the Station ▪ Starting time of the disruption ▪ Time at which the disruption will end (for planned operations) ▪ Reason of the disruption
Data Model Reference	STA-TION_DISRUPION_EVENT = ENTITY_HEADER + STATION_ID + TIME_STAMP + (TIME_STAMP) + { 0-9 | A-z | . | , }

6.2.16 SwitchDisruptionEvent



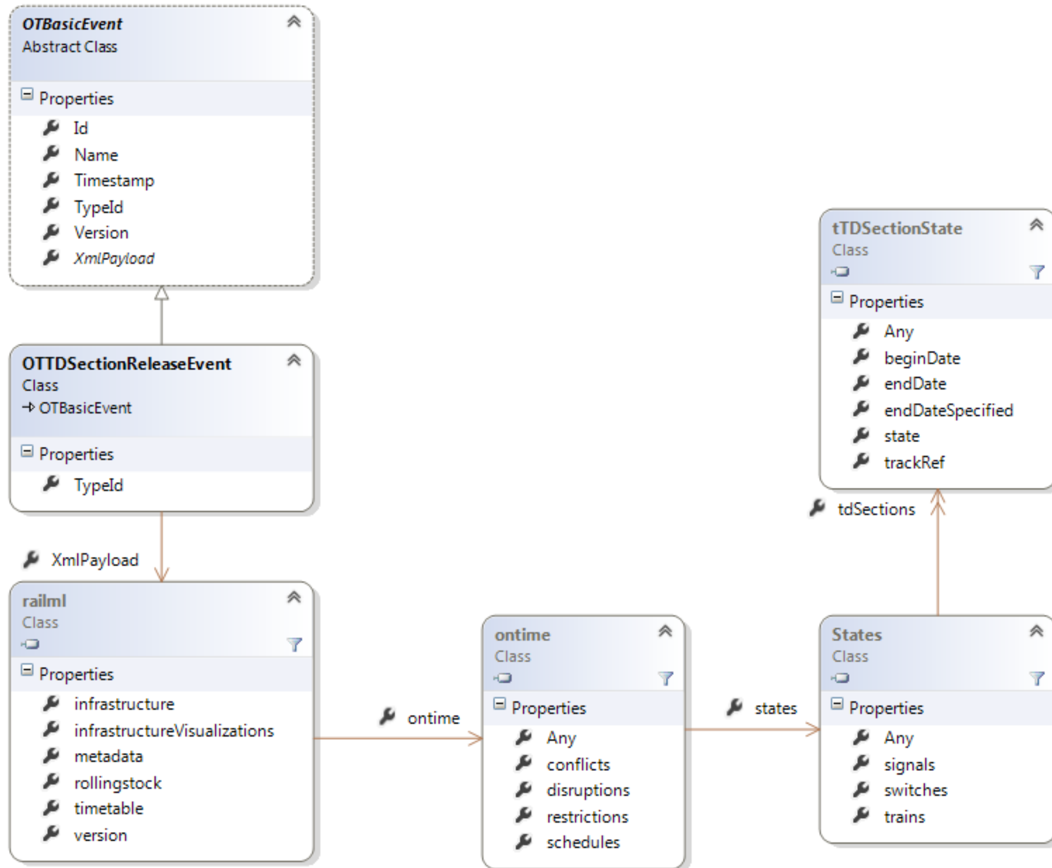
TypeId	{4CC9A382-6893-4B01-B36D-B098C3D165DA}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that there's a switch that has been disrupted.
Data Payload	▪ ID of the Switch ▪ Starting time of the disruption ▪ Time at which the disruption will end (for planned operations) ▪ Reason of the disruption
Data Model Reference	SWITCH_DISRUPTION_EVENT = ENTITY_HEADER + SWITCH_ID + TIME_STAMP + (TIME_STAMP) + { 0-9 | A-z | . | , }

6.2.17 TDSectionOccupationEvent



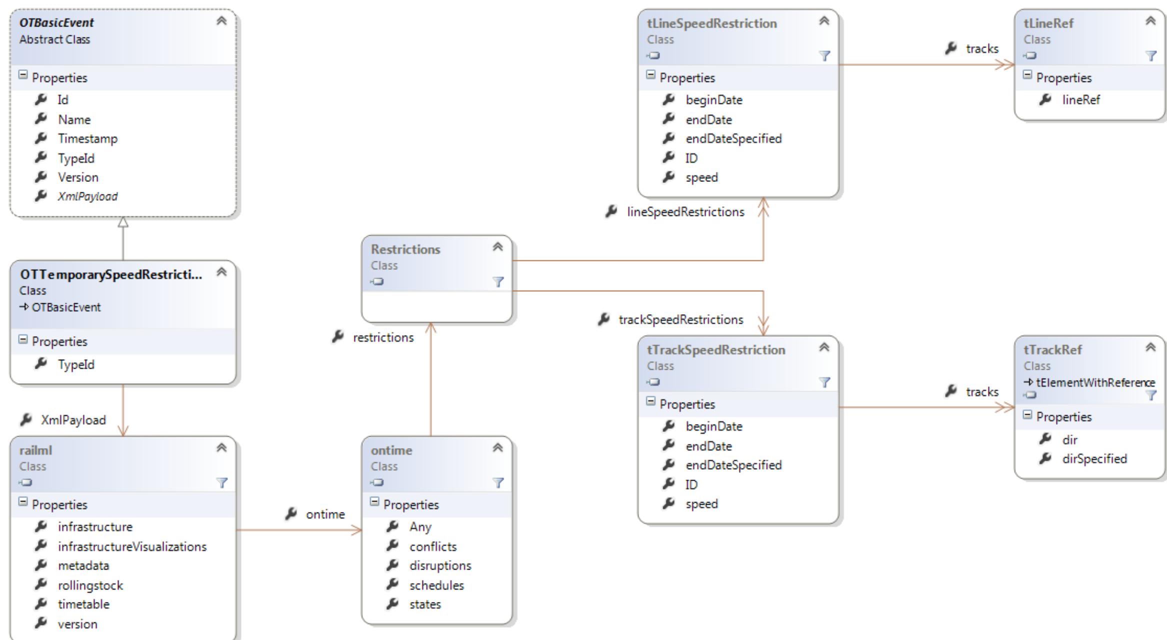
TypeId	{EE0D4827-2370-4FBC-980D-95BECA0BC6C1}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies that a track detection section has been occupied by a given train.
Data Payload	▪ ID of the section (unique) ▪ Train ID ▪ Time of occupation
Data Model Reference	TD_SECTION_OCCUPATION_EVENT = ENTITY_HEADER + BLOCK_ID + TRAIN_ID + TIME_STAMP

6.2.18 TDSectionReleaseEvent



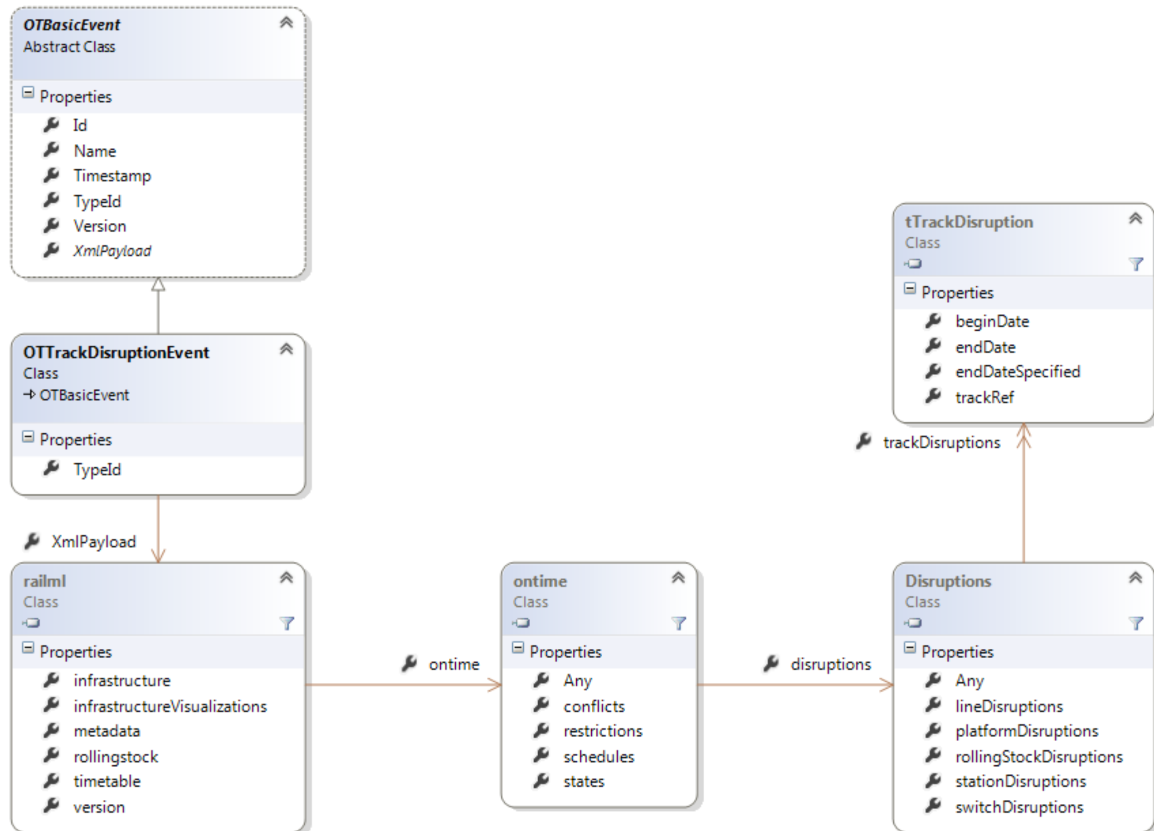
TypeId	{0CC7C8DA-29D4-4A8C-B774-480AA24B830B}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies that a track detection section has been released by a given train.
Data Payload	▪ ID of the section (unique) ▪ Train ID ▪ Time of release
Data Model Reference	TD_SECTION_RELEASE_EVENT = ENTITY_HEADER + BLOCK_ID + TRAIN_ID + TIME_STAMP

6.2.19 TemporarySpeedRestrictionsEvent



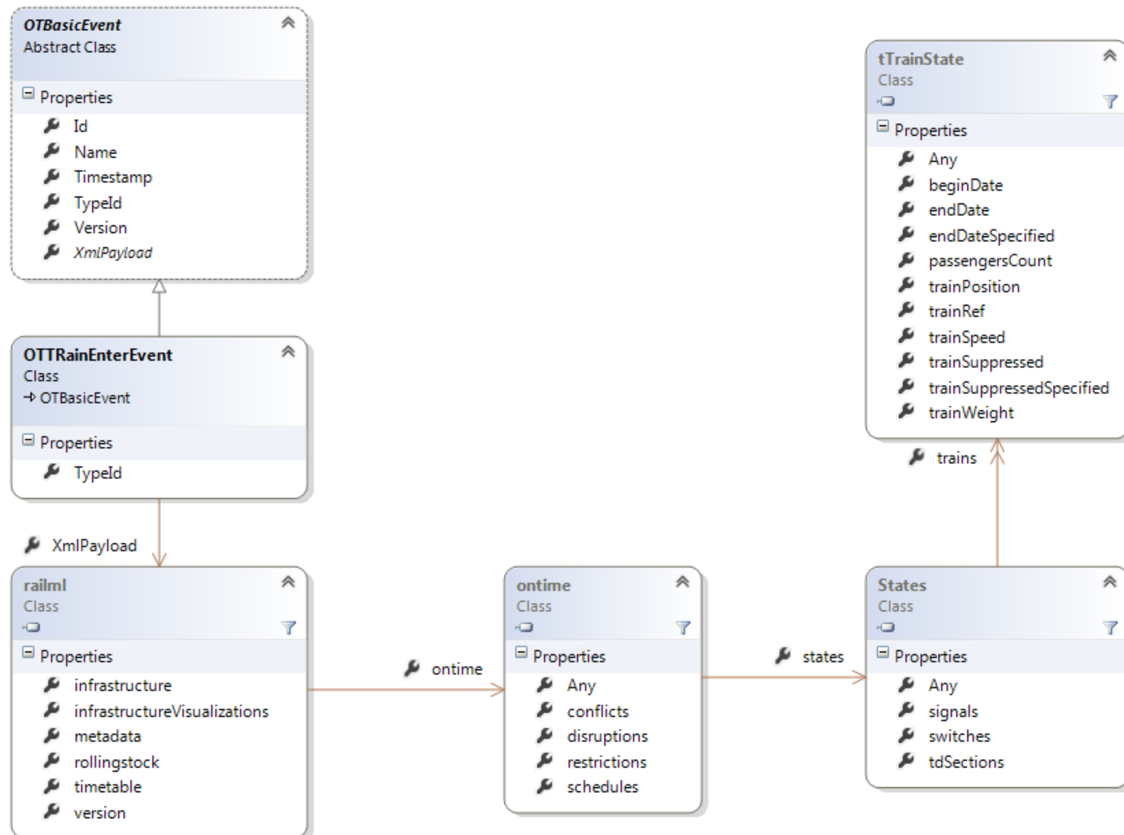
TypeId	{539785DA-EE7B-4670-8183-B16DAB6BA9D3}
Created By:	TCS
Consumed By:	WP4
Description:	Defines a temporary speed restriction on a specified set of tracks or lines.
Data Payload	▪ List of IDs of the RailML representation of Tracks and/or Lines ▪ Speed restriction value
Data Model Reference	TEMPORARY_SPEED_RESTRICTION = {LINE_ID} + {TRACK_ID} + SPEED_RESTRICTION

6.2.20 TrackDisruptionEvent



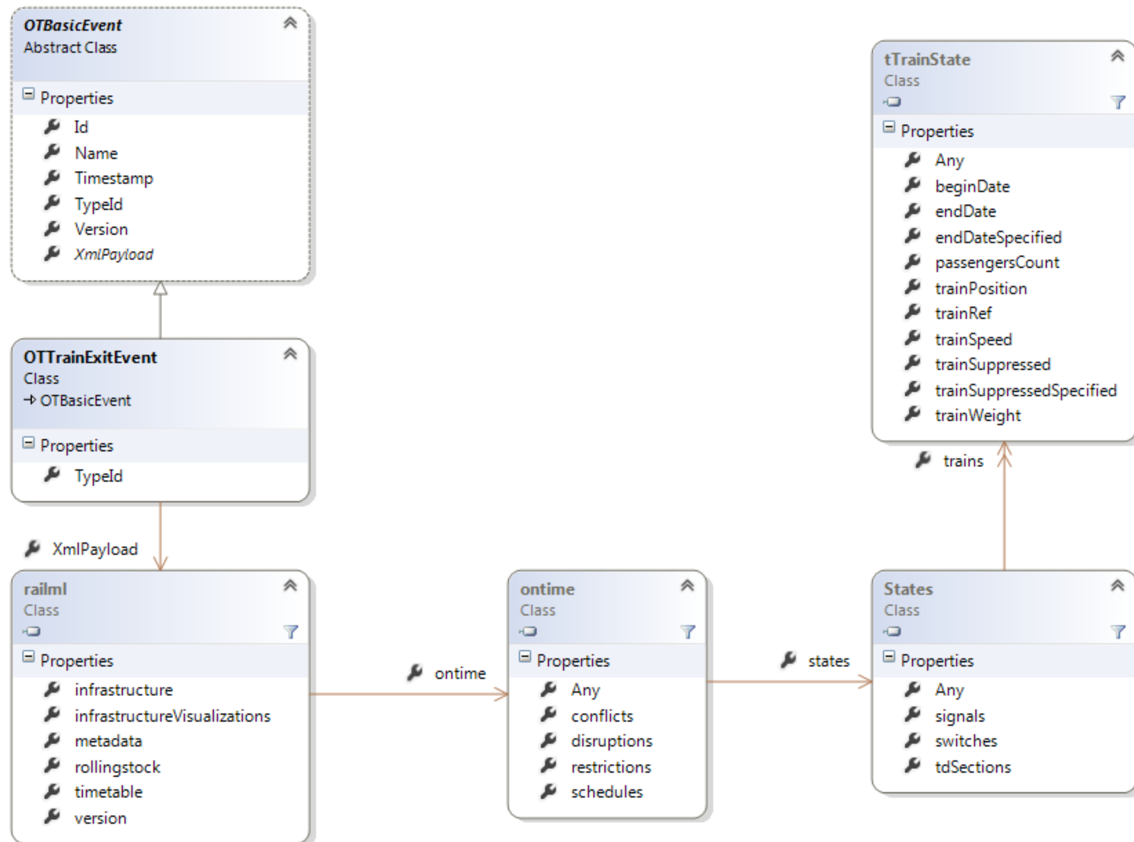
TypeId	{82A2C530-6C98-4800-8A75-E1EA80BE2D29}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that there's a track that has been disrupted.
Data Payload	▪ ID of the Track ▪ Starting time of the disruption ▪ Time at which the disruption will end (for planned operations) ▪ Reason of the disruption
Data Model Reference	TRACK_DISRUPTION_EVENT = ENTITY_HEADER + TRACK_ID + TIME_STAMP + (TIME_STAMP) + { 0-9 | A-z | , | . }

6.2.21 TrainEnterEvent



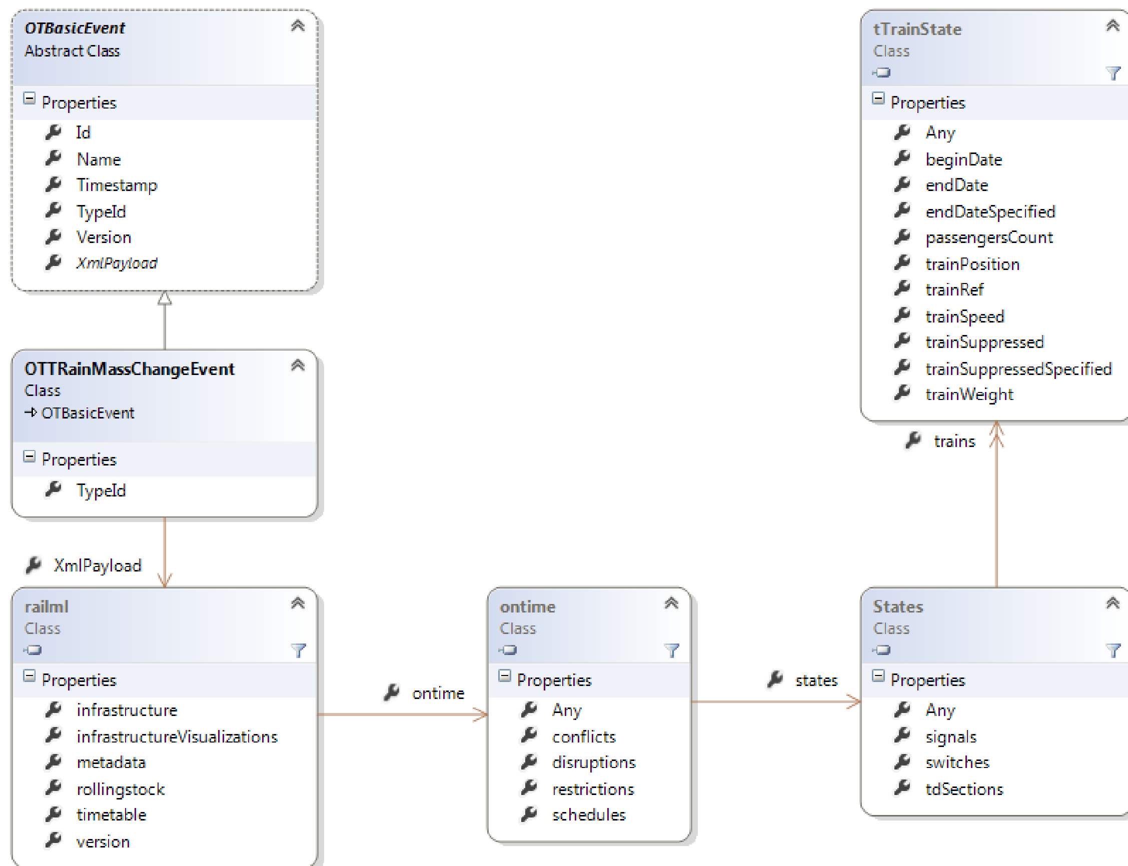
TypeId	{66879F3F-6C86-4761-828F-8816FEFDB2DD}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notify that a new Train has entered the scenario. Usually this event happens when a new train starts from a station.
Data Payload	▪ ID of the Train ▪ Time at which the train entered the scenario
Data Model Reference	TRAIN_ENTER_EVENT = ENTITY_HEADER + TRAIN_ID + TIME_STAMP

6.2.22 TrainExitEvent



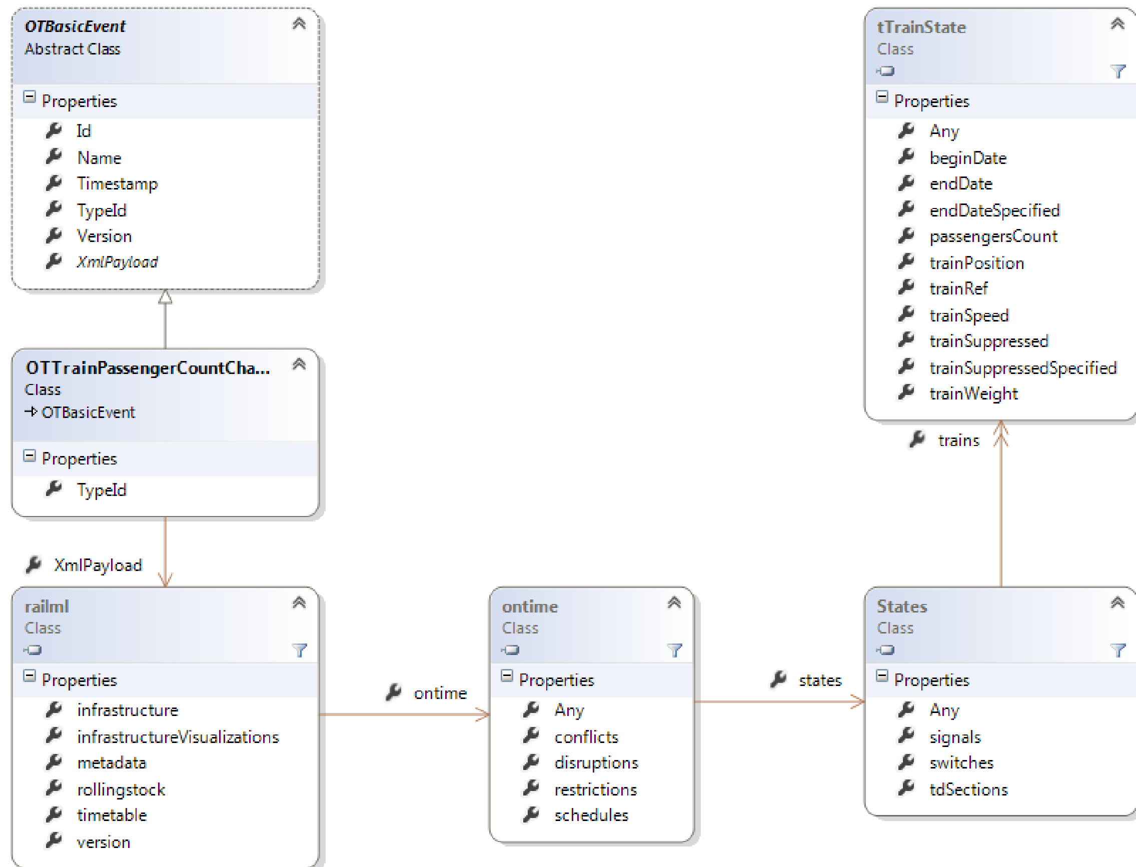
TypeId	{44051CCD-35C5-48FF-8E0A-9318920986A4}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notify that a new Train has exited the scenario. Usually this event happens when a train finishes its route or exits the scenar-io boundaries.
Data Payload	▪ ID of the Train ▪ Time at which the train left the scenario
Data Model Reference	TRAIN_EXIT_EVENT = ENTITY_HEADER + TRAIN_ID + TIME_STAMP

6.2.23 TrainMassChangeEvent



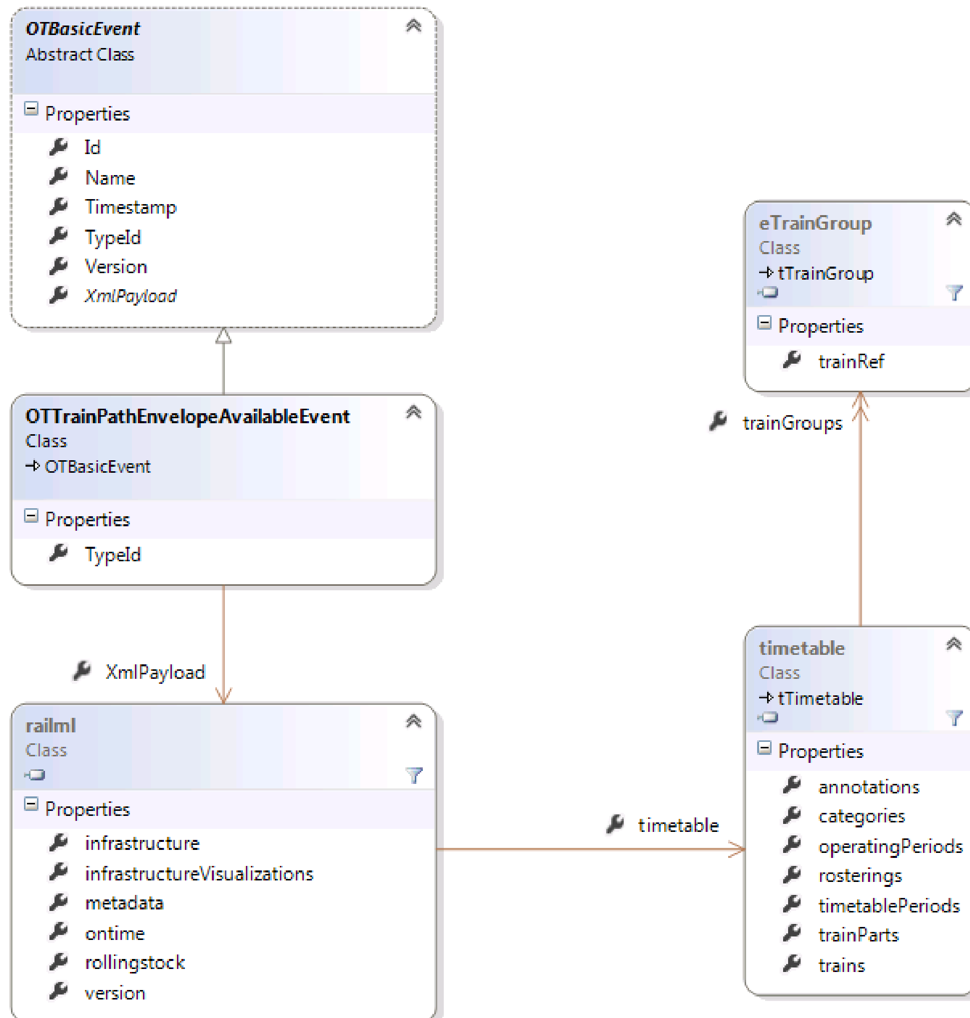
TypeId	{6F789BE4-4D2C-4BDA-835D-622FD236FD3F}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies that the mass of a train changed.
Data Payload	▪ ID of the RailML Train Representation ▪ New mass of the train
Data Model Reference	TRAIN_MASS_CHANGE_EVENT = ENTITY_HEADER + TRAIN_MASS

6.2.24 TrainPassengerCountChangeEvent



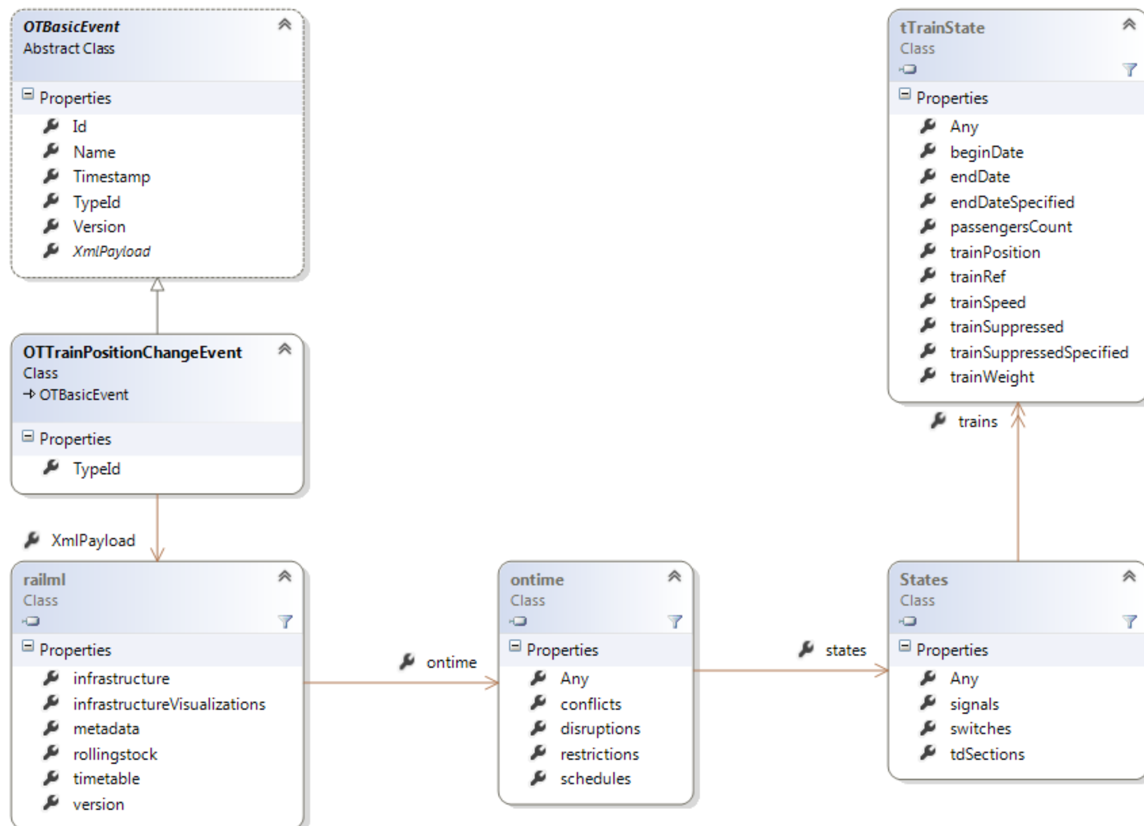
TypeId	{75C07B0C-A86D-4ADF-9922-8394DD6EDB14}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies that the passenger count of a train changed.
Data Payload	▪ ID of the Train ▪ New passenger count
Data Model Reference	TRAIN_PASSENGER_COUNT_CHANGE_EVENT = ENTI-TY_HEADER + TRAIN_OCCUPATION

6.2.25 TrainPathEnvelopeAvailableEvent



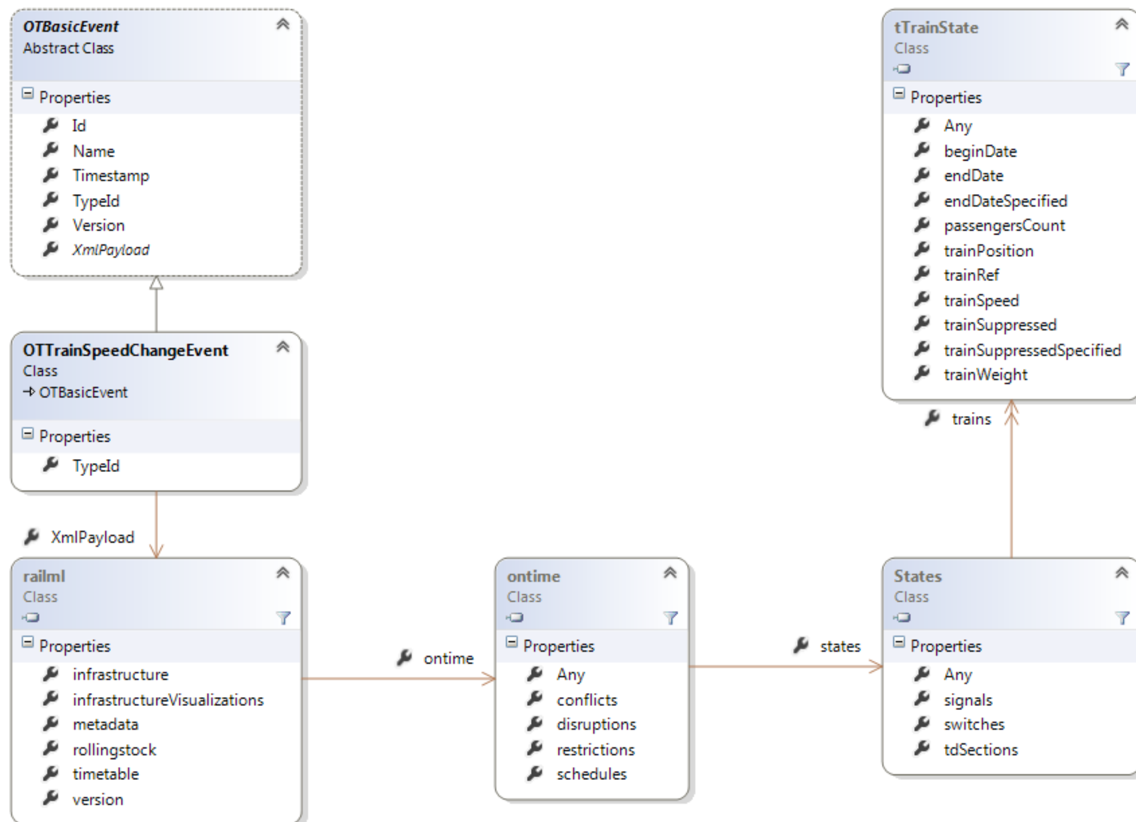
TypeId	{4681FF8F-63BB-4CA5-B9A6-195EEAC7A54D}
Created By:	TCS
Consumed By:	WP6
Description:	Notifies systems that a new train path envelopes are available
Data Payload	▪ List of TRAIN_ID whose Train Path Envelope has been up-dated.
Data Model Reference	TRAIN_PATH_ENVELOPE_AVAILABLE_EVENT = ENTI-TY_HEADER + { TRAIN_ID}

6.2.26 TrainPositionChangeEvent



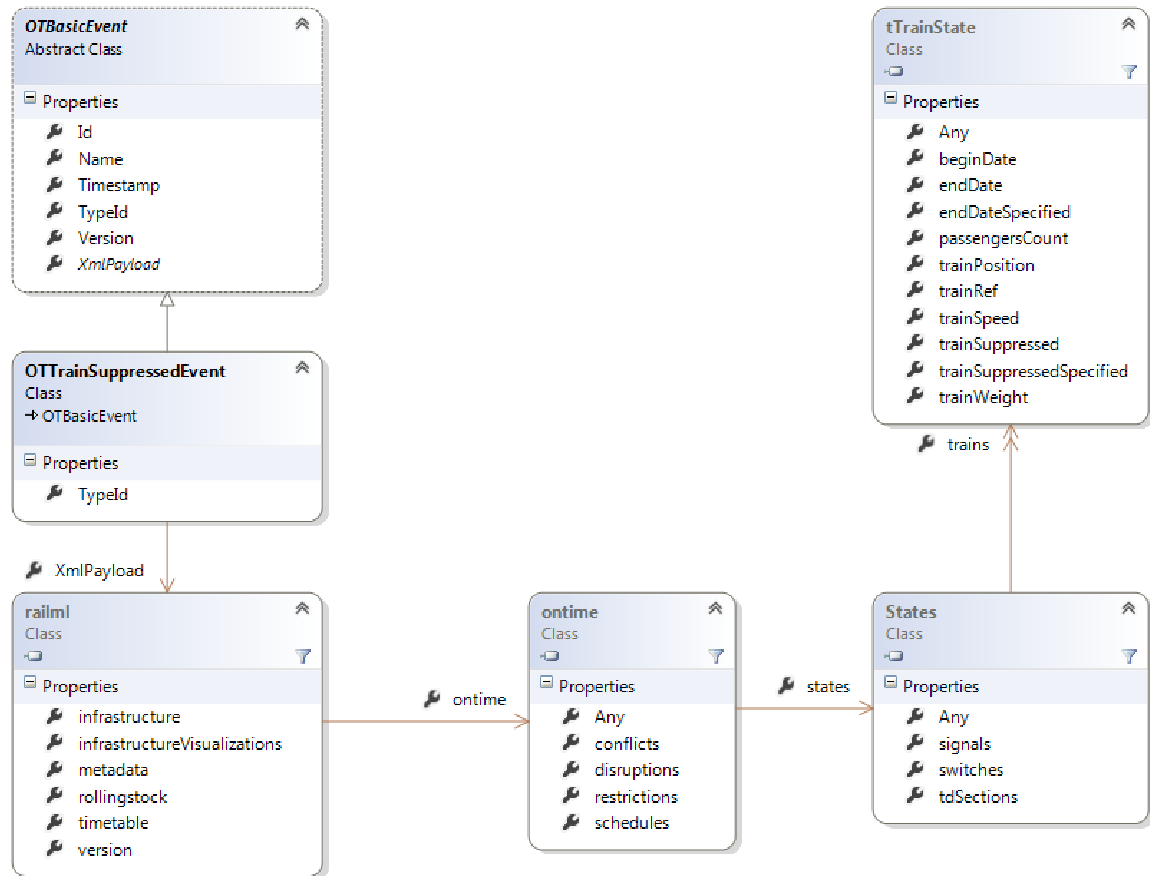
TypeId	{5D77BD3C-7DAA-4939-8148-0F41C5E094A7}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies that the position of a train changed.
Data Payload	▪ ID of the Train ▪ Position of the train head ▪ List of the occupied tracks
Data Model Reference	TRAIN_POSTITION_CHANGE_EVENT = ENTITY_HEADER + TRAIN_POSITION

6.2.27 TrainSpeedChangeEvent



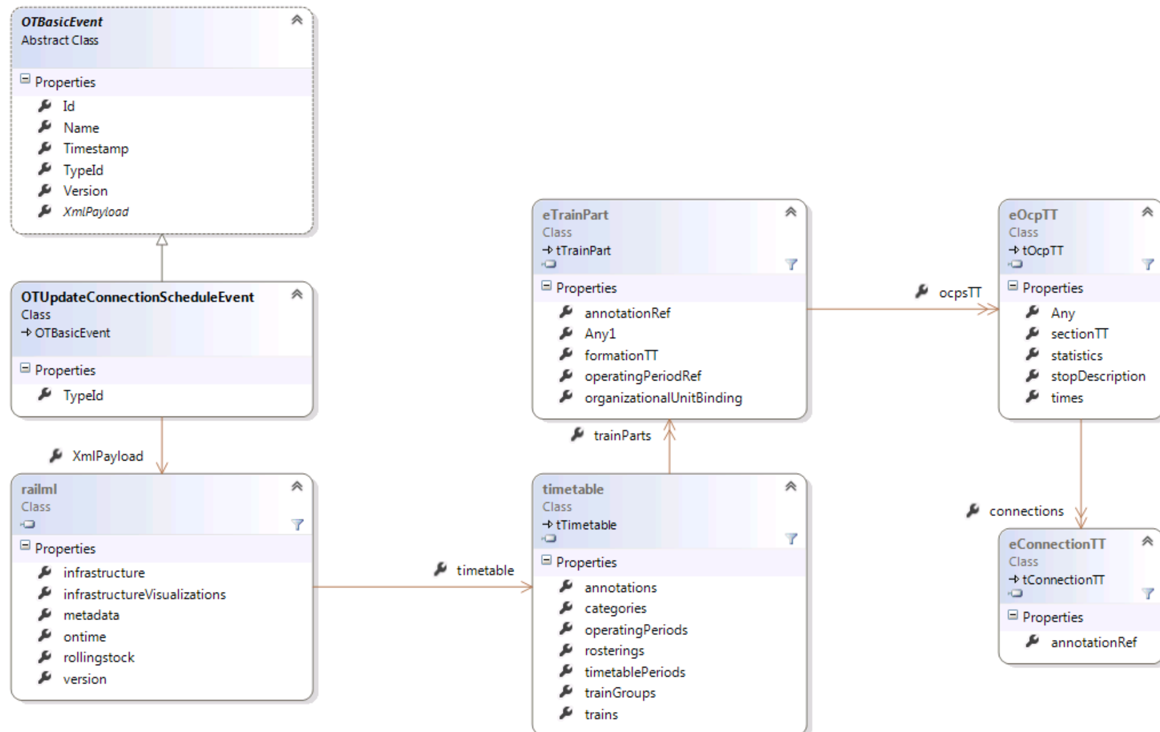
TypeId	{25485129-F540-4646-92E4-9AB375C114AA}
Created By:	TCS
Consumed By:	WP4
Description:	Notifies that the speed of a train changed.
Data Payload	▪ ID of the Train ▪ New speed of the train
Data Model Reference	TRAIN_SPEED_CHANGE_EVENT = ENTITY_HEADER + TRAIN_SPEED

6.2.28 TrainSuppressedEvent



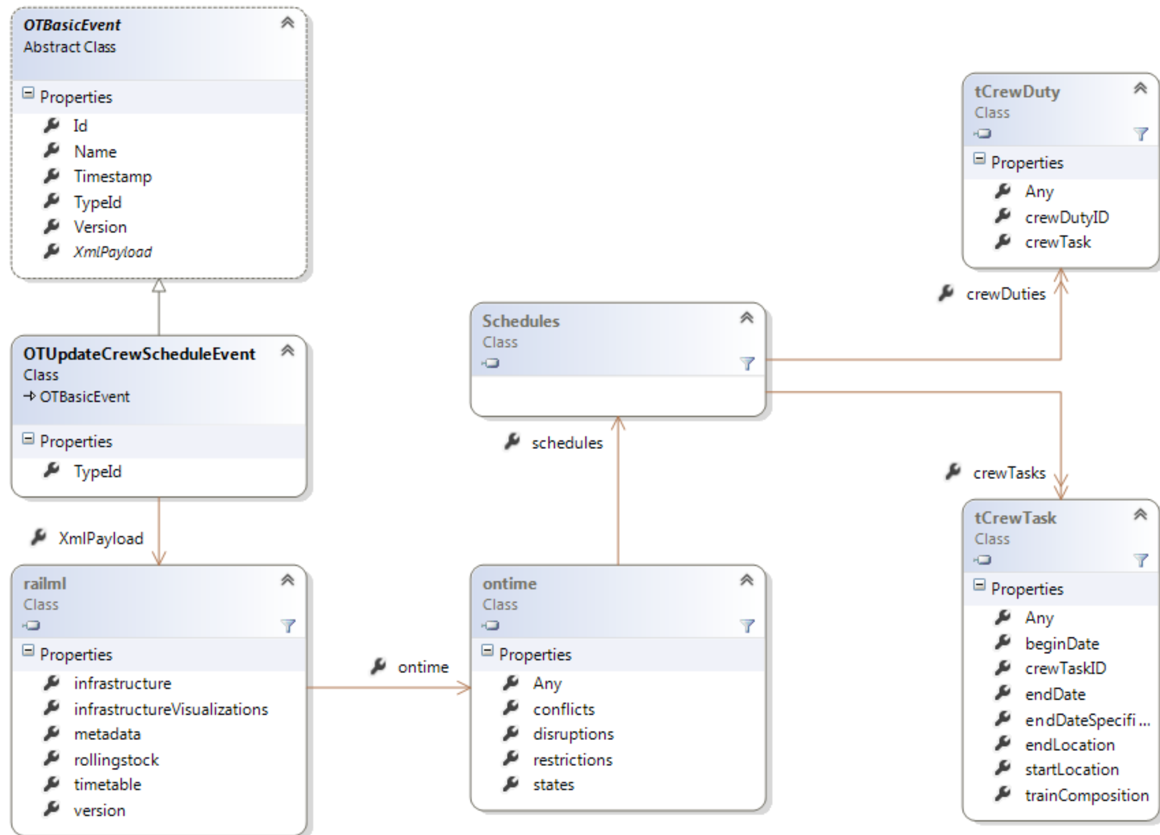
TypeId	{883CD302-5A04-456A-A5C9-5C303D94BE8E}
Created By:	TCS
Consumed By:	WP4, WP5
Description:	Notifies that a Train has been suppressed.
Data Payload	▪ ID of the Train ▪ Time at which the train has been suppressed
Data Model Reference	TRAIN_SUPPRESSED_EVENT = ENTITY_HEADER + TRAIN_ID + TIME_STAMP + { 0-9 | A-z | , | . }

6.2.29 UpdateConnectionScheduleEvent



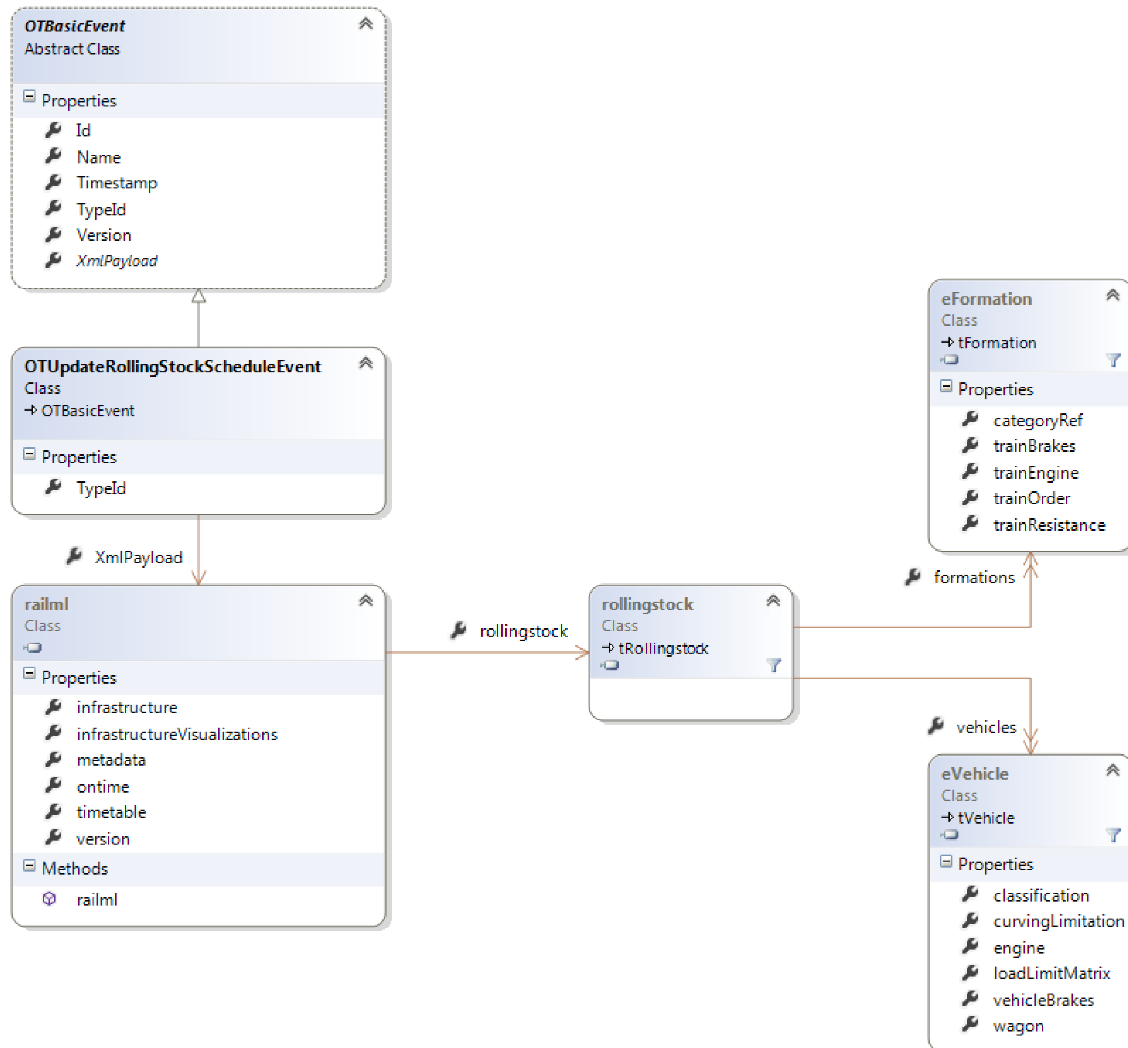
TypeId	{B5FC4768-8076-4DCB-A5D2-8AF120C0CD94}
Created By:	WP5
Consumed By:	TCS
Description:	Publishes a new ConnectionSchedule to the platform.
Data Payload	▪ New ConnectionSchedule
Data Model Reference	UPDATE_CONNECTION_SCHEDULE_EVENT = ENTITY_HEADER + {CONNECTION}

6.2.30 UpdateCrewScheduleEvent



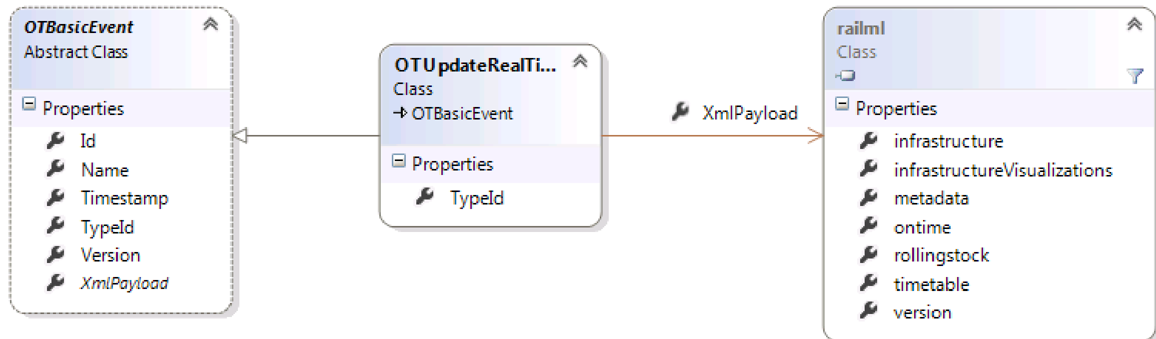
TypeId	{FB574C5E-2A21-4221-A5B1-78A8D81501EF}
Created By:	WP5
Consumed By:	TCS
Description:	Publishes a new CrewSchedule to the platform.
Data Payload	▪ New CrewSchedule
Data Model Reference	UPDATE_CREW_SCHEDULE_EVENT = ENTITY_HEADER + {CREW TASK} + {CREW DUTY}

6.2.31 UpdateRollingStockScheduleEvent



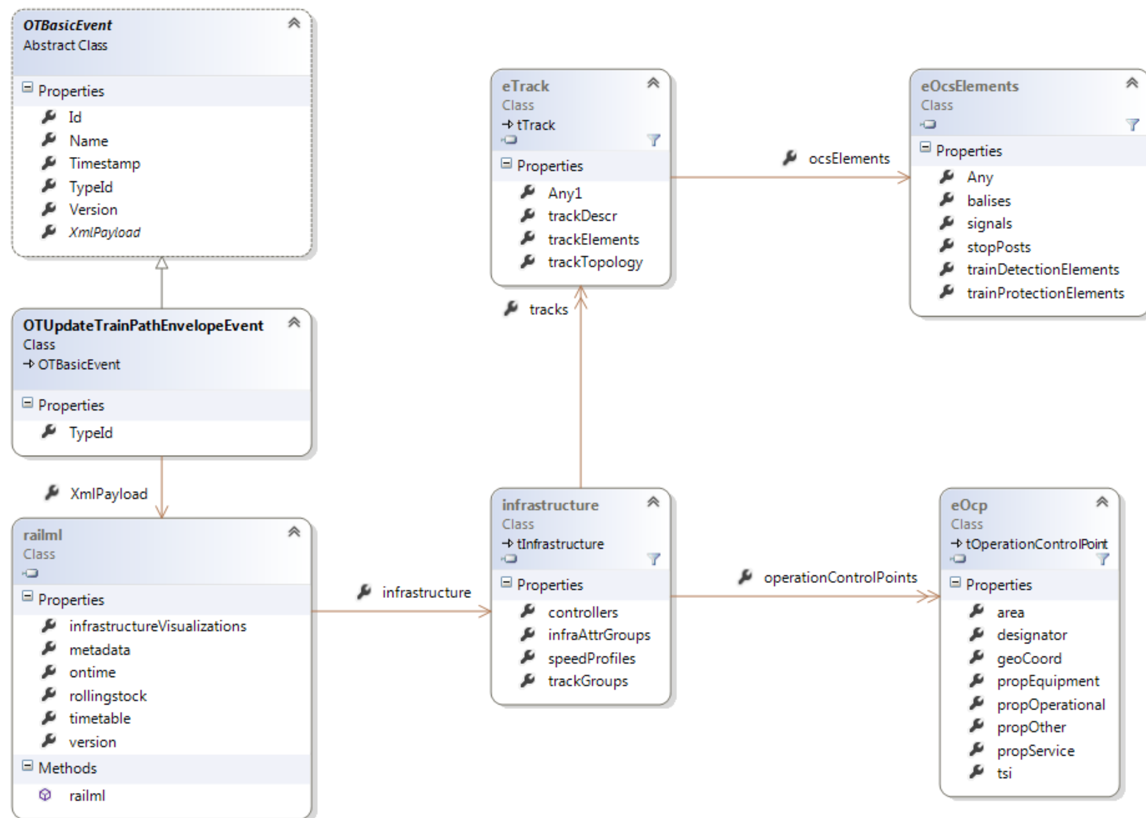
TypeId	{8650BA4B-5786-4736-AD67-C1FABAA00B90}
Created By:	WP5
Consumed By:	TCS
Description:	Publishes a new RollingStockSchedule to the platform.
Data Payload	▪ New RollingStockSchedule
Data Model Reference	UPDATE_ROLLING_STOCK_SCHEDULE_EVENT = ENTITY_HEADER + {VEHICLE} + {TRAIN COMPOSITION}

6.2.32 UpdateRealTimeTrafficPlanEvent



TypeId	{07693469-9F7A-480D-B808-6DB21E4A9CC6}
Created By:	WP4
Consumed By:	TCS
Description:	Publishes a new traffic plan to the platform. Note that there's a difference between a UpdateRealTimeTrafficPlanEvent and a RealTimeTrafficPlanAvailableEvent. The former actually publishes the new data on the platform, the latter will just notify the modules that a new traffic plan is available as Static Data.
Data Payload	▪ New traffic plan
Data Model Reference	UPDATE_REAL_TIME_TRAFFIC_PLAN_EVENT = ENTITY_HEADER + REAL_TIME_TRAFFIC_PLAN + TIME_STAMP

6.2.33 UpdateTrainPathEnvelopeEvent



TypeId	{DB06D35A-7714-4CBF-AF7A-CBA7E4C01A60}
Created By:	WP4
Consumed By:	TCS
Description:	Publishes a new train path envelope to the system.
Data Payload	▪ Train path envelopes with time windows of signal passing (at critical positions) ▪ Stopping part of real-time traffic plan
Data Model Reference	**UPDATE_TRAIN_PATH_ENVELOPE_EVENT = ENTITY_HEADER + TRAIN_PATH_ENVELOPE**

7 INTEGRATION REQUIREMENTS

7.1 Integration Interfaces

Different modules of the On-Time Architecture will communicate using standard interfaces. For sake of clarity, we will represent these interfaces as standard UML interfaces definitions but, in most of the cases, they have to be interpreted as web Services interfaces of distributed systems.

Since the rise of Web 2.0 technologies, the difference between static and distributed interfaces are becoming more and more subtle, some languages doesn't make any difference between the two, since communication frameworks and high-level programming languages can abstract the communication layer from the functional specifications.

Since the architecture is conceived to be modular and flexible, remote interfaces can become native and vice-versa, the architecture definition poses no limit on how the modules can be coupled.

Functionally, the On-Time architecture adhere to a minimalist design pattern. Even though there are several guidelines that the modules must comply to, they are very simple building blocks that can be used in different ways to achieve the desired results. In addition, the technology used allows for cross-language, cross-OS and cross-technologies integration without too much effort.

The architecture will distinguish modules. Modules have been conveniently named against the WP name of the project. A single WP can have multiple modules and different implementations (for example different WP4 Modules can exists, with different kind of optimization and planning algorithm). It is responsibility of the architecture to keep communications between the modules and forward data and messages to the correct endpoints.

Basically, a module must carry on these functions:

- Be able to subscribe for events.
- Be able to publish events.
- Be able to access the Static and Operational Data Provider of the architecture.

These three functional characteristics are the basics at which a module must adhere to be able to carry out its functions.

To be able to receive events, a module need to implement the EventConsumer Interface. The EventConsumer Interface it is the basic requirement to process messages and it is shared both by modules than by the architecture (since also the architecture must be able to receive messages from modules).

To access Operational and Static Data a module must know the location of a Data Provider Service. This service is an abstraction between different systems that stores and manages data for railway systems, called Train Control System or TCS.

Finally, to be able to register as a subscriber or a publisher of events the module will need to know the location of a SubscriptionService.

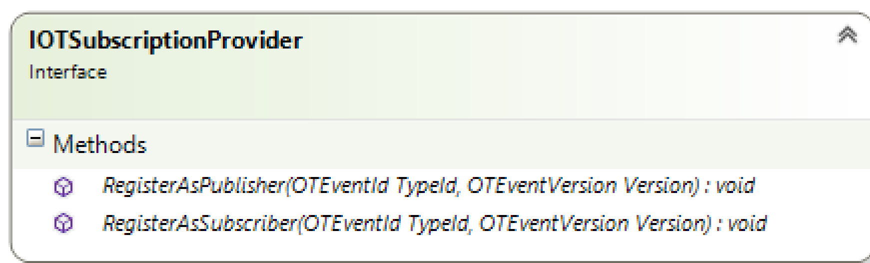
In the following chapters, we will explain the external integration interfaces and the internal integration interfaces of the architecture.

The External integration interfaces are shared between modules and the architecture, the Internal integration interfaces are maintained between the different modules of the architecture itself.

7.1.1 External Integration Interfaces

Publish/Subscribe Interface

Provided by: Architecture



The Publish/Subscribe interface gives modules within the system the ability to register their interest in receiving notifications describing system state changes.

The behaviour of a publishing registration request is as follows:

- The module interested to dispatch events will send a tuple [TypeId, Version]
- A validation of the request is performed
- If the request is valid, the platform will register the endpoint and will send a unique token id to as a response
- From this moment on, the module is allowed to publish events to the platform

The behaviour of a subscription registration request is as follows:

- A module will contact the P/S Interface endpoint sending one or more tuples: [TypeId, version]
- A validation of the request is performed
- If all the issued tuples [TypeId, version] are available and the request is valid, the SubscriptionService will acknowledge the subscriptions;
- If one of the issued tuples [TypeId, version] is not available, then the request is not valid, and the SubscriptionService will respond with an error.

Please note: from the publishing module point of view, there is only a single endpoint where to send events. If a module that is attempting to publish data is not registered or has a token that is no longer valid, any event it issues will be automatically discarded by the communication middleware.

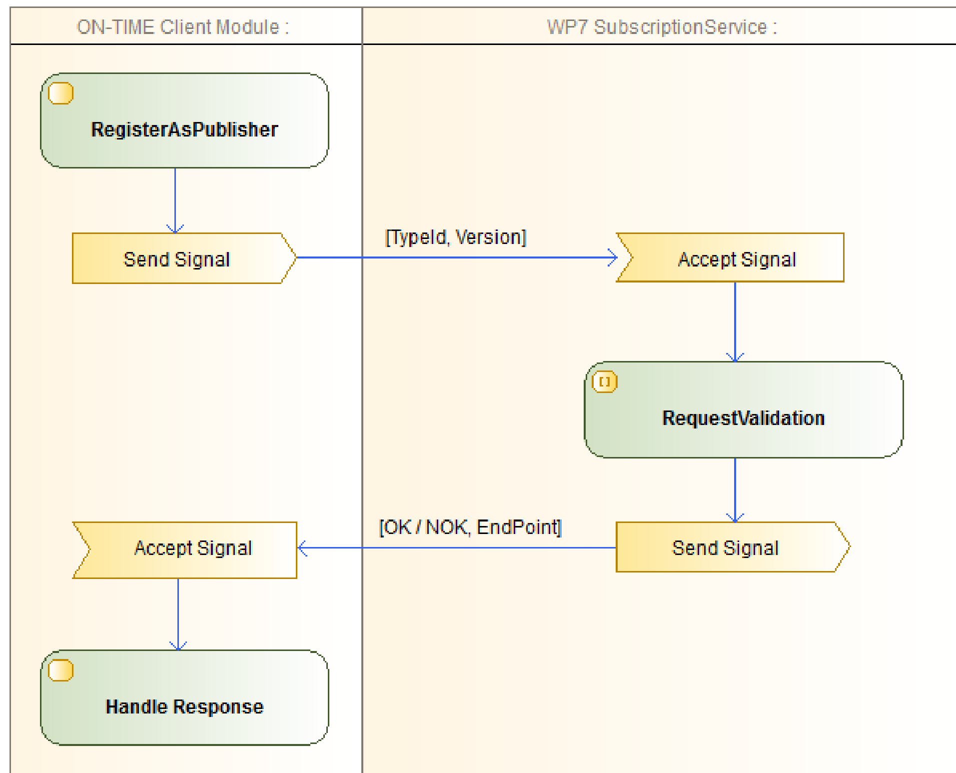


Figure 4 Register as Publisher - Activity diagram

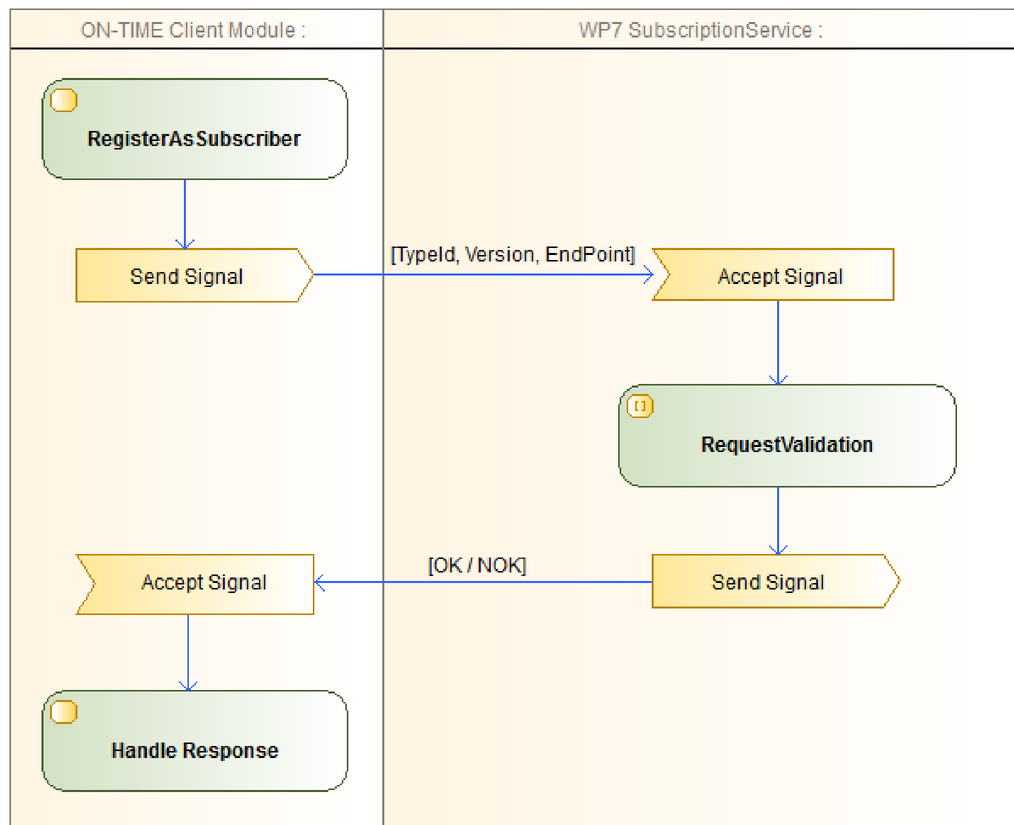


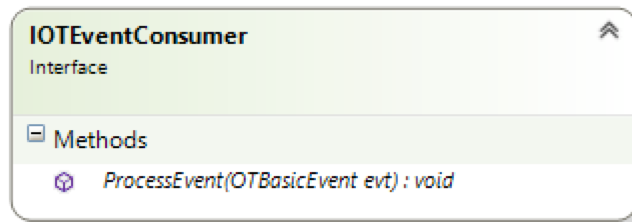
Figure 5 Register as Subscriber - Activity diagram

`RegisterAsSubscriber` informs the platform that a module wants to be registered as a subscriber for a given event id and version number. The platform stores the data about the subscriber and creates a new entry in the communication bus for the module.

The **`RegisterAsPublisher`** message informs the platform that a module needs to publish events to the system. Upon receipt of the message, the platform creates a logical link and a broadcast message queue for the new event stream (if not already present). It is important to note that multiple systems from the same logical module can register as event publishers. In that case, the platform will not guarantee causal ordering on those messages.

7.1.2 EventConsumer Interface

Can be implemented by: Architecture, Modules



Every module within the architecture (including optimization plugins) that needs to process events, has to implement the IOTEventConsumer Interface. This interface abstracts an endpoint that consumes an ON-TIME event, which is always a derived class of OTBasicEvent.

The IOTEventConsumer interface should behave asynchronously, ensuring that the application is not blocked by the receipt of events. As ON-TIME is using topic-based routing, the architecture won't process events based on their content but will just route them to endpoints as requested by module subscriptions.

The **`ProcessEvent`** message is an endpoint to receive messages. To avoid a blocking connection, the method should be implemented as a non-blocking event receiver that should do as little as possible before relinquishing control to the architecture message-dispatching thread.

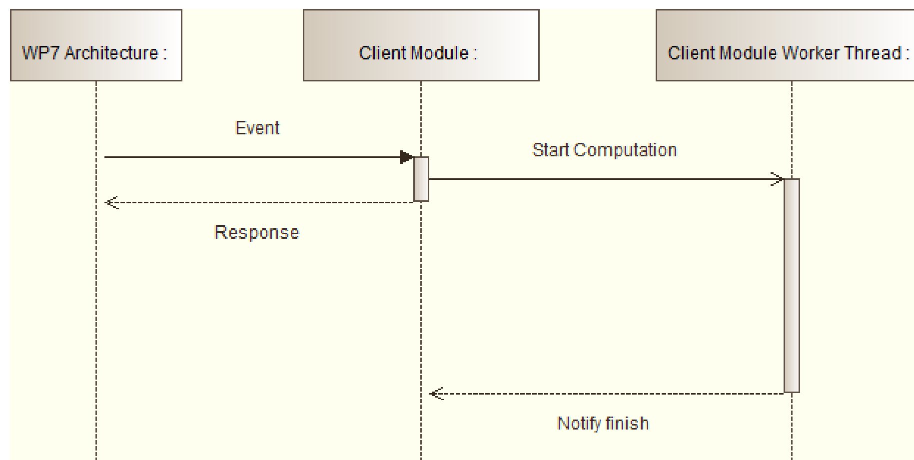


Figure 6 Non-Blocking behaviour

Web Services have synchronous message models and in any case, algorithm computation should be linked to the ProcessEvent interface. What should be done in the message is to check for data validity and correct model representation, scheduling the processing of the message payload in another, internal to the module thread.

7.1.3 DataProvider Interface

Implemented by: Architecture

IOTDataProvider
Interface

Members

- GetCommercialTimeTables(DateTime refDate, OTEntityId[] ids) : OTCommercialTimetable[]
- GetCommercialTimeTables(OTEntityId[] ids) : OTCommercialTimetable[]
- GetConnectionConstraints(DateTime refDate, OTEntityId[] ids) : OTConnectionConstraint[]
- GetConnectionConstraints(OTEntityId[] ids) : OTConnectionConstraint[]
- GetCrewConstraints(DateTime refDate, OTEntityId[] ids) : OTCrewConstraint[]
- GetCrewConstraints(OTEntityId[] ids) : OTCrewConstraint[]
- GetCrewSchedule(DateTime refDate, OTEntityId[] ids) : OTCrewSchedule
- GetCrewSchedule(OTEntityId[] ids) : OTCrewSchedule
- GetInfrastructreUnavailability() : OTInfrastructureItem
- GetInfrastructreUnavailability(DateTime refDate) : OTInfrastructureItem
- GetInfrastructureData() : OTInfrastructure
- GetInfrastructureData(DateTime refDate) : OTInfrastructure
- GetInterlocks(DateTime refDate, OTEntityId[] ids) : OTInterlock[]
- GetInterlocks(OTEntityId[] ids) : OTInterlock[]
- GetLineDelayInfo(DateTime refDate, OTEntityId[] ids) : OTDelayInfo[]
- GetLineDelayInfo(OTEntityId[] ids) : OTDelayInfo[]
- GetLines(DateTime refDate, OTEntityId[] ids) : OTLine[]
- GetLines(OTEntityId[] ids) : OTLine[]
- GetOperationalTimeTables(DateTime refDate, OTEntityId[] ids) : OTOperationalTimetable[]
- GetOperationalTimeTables(OTEntityId[] ids) : OTOperationalTimetable[]
- GetRealTimeTrafficPlan() : OTRealTimeTrafficPlan
- GetRealTimeTrafficPlan(DateTime refDate) : OTRealTimeTrafficPlan
- GetRollingStockConstraints(DateTime refDate, OTEntityId[] ids) : OTRollingStockConstraint[]
- GetRollingStockConstraints(OTEntityId[] ids) : OTRollingStockConstraint[]
- GetRollingStockSchedule(DateTime refDate, OTEntityId[] ids) : OTRollingStockSchedule
- GetRollingStockSchedule(OTEntityId[] ids) : OTRollingStockSchedule
- GetSignals(DateTime refDate, OTEntityId[] ids) : OTSignal[]
- GetSignals(OTEntityId[] ids) : OTSignal[]
- GetStationDelayInfo(DateTime refDate, OTEntityId[] ids) : OTDelayInfo[]
- GetStationDelayInfo(OTEntityId[] ids) : OTDelayInfo[]
- GetStations(DateTime refDate, OTEntityId[] ids) : OTStation[]
- GetStations(OTEntityId[] ids) : OTStation[]
- GetTimetableDelayInfo(DateTime refDate, OTEntityId[] ids) : OTDelayInfo[]
- GetTimetableDelayInfo(OTEntityId[] ids) : OTDelayInfo[]
- GetTimetables(DateTime refDate, OTEntityId[] ids) : OTTimetable[]
- GetTimetables(OTEntityId[] ids) : OTTimetable[]
- GetTracks(DateTime refDate, OTEntityId[] ids) : OTTrack[]
- GetTracks(OTEntityId[] ids) : OTTrack[]
- GetTrainDelayInfo(DateTime refDate, OTEntityId[] ids) : OTDelayInfo[]
- GetTrainDelayInfo(OTEntityId[] ids) : OTDelayInfo[]
- GetTrains(DateTime refDate, OTEntityId[] ids) : OTTrain[]
- GetTrains(OTEntityId[] ids) : OTTrain[]

The `IOTDataProvider` Interface is a public interface used to access read-only scenario data.

The **IOTDataProvider** is an abstraction of the TCS systems that ON-TIME will interact with, and provides methods by which modules within the platform may access data managed by those systems.

General conventions: The methods will always be provided with two basic signatures: one to access the current data, and one to access the data at a given DateTime. In the latter case, the system will return the last data structure published at a date and time lesser than or equal to the *refDate* provided. Methods that can be used to access one or more data entities given their IDs will implement this behaviour: if the IDs array is populated, they will return information on the specific items, if the ID array is null (or empty), they will return data for all the items valid at that given time.

```
OTInfrastructure GetInfrastructureData();
OTInfrastructure GetInfrastructureData(DateTime refDate);
```

Provides a RailML representation of the current static infrastructure. The static infrastructure contains all the following information:

- Tracks
- Lines
- Stations
- Signals
- Interlocks

```
OTRealTimeTrafficPlan GetRealTimeTrafficPlan();
OTRealTimeTrafficPlan GetRealTimeTrafficPlan(DateTime refDate);
```

Provides information about the traffic plan, including:

- Active lines
- Current Trains
- Timetables

```
OTCrewSchedule GetCrewSchedule(OTEntityId[] ids);
OTCrewSchedule GetCrewSchedule(DateTime refDate, OTEntityId[] ids);
```

Provides a data interface to the crew schedule of an abstracted crew management system. The method will report the crew schedule for the current or reference day.

```
OTRollingStockSchedule GetRollingStockSchedule(OTEntityId[] ids);
OTRollingStockSchedule GetRollingStockSchedule(DateTime refDate, OTEntityId[] ids);
```

Provides a resource plan for all the current planned rolling stock and railway movements.

```
OTInfrastructureItem GetInfrastructreUnavailability();
OTInfrastructureItem GetInfrastructreUnavailability(DateTime refDate);
```

Provides access to a list of all sections of the infrastructure that are current unavailable for use.

```
OTTrack[] GetTracks(OTEntityId[] ids);
OTTrack[] GetTracks(DateTime refDate, OTEntityId[] ids);
```

Returns RailML information of a given tracks.

```
OTLine[] GetLines(OTEntityId[] ids);
OTLine[] GetLines(DateTime refDate, OTEntityId[] ids);
```

Returns RailML information of a given set of Lines.

```
OTStation[] GetStations(OTEntityId[] ids);
OTStation[] GetStations(DateTime refDate, OTEntityId[] ids);
```

Returns RailML information of a given sets of Stations.

```
OTTrain[] GetTrains(OTEntityId[] ids);
OTTrain[] GetTrains(DateTime refDate, OTEntityId[] ids);
```

Provides RailML information of a given sets of Trains.

```
OTSignal[] GetSignals(OTEntityId[] ids);
OTSignal[] GetSignals(DateTime refDate, OTEntityId[] ids);
```

Provides RailML information of a given sets of Signals, with their current states.

```
OTInterlock[] GetInterlocks(OTEntityId[] ids);
OTInterlock[] GetInterlocks(DateTime refDate, OTEntityId[] ids);
```

Provides details for interlock data entities, with their current switching states.

```
OTTimetable[] GetTimetables(OTEntityId[] ids);
OTTimetable[] GetTimetables(DateTime refDate, OTEntityId[] ids);
```

Provides access to a given sets of timetables defined in the TCS. Different Timetables may refer to different “layers” of a same, global timetable, or other types of classifications associated to a Timetable.

```
OTOperationalTimetable[] GetOperationalTimeTables(OTEntityId[] ids);
OTOperationalTimetable[] GetOperationalTimeTables(DateTime refDate, OTEntityId[]
ids);
```

Provides access to a given sets of operating timetables present in the TCS. Different Timetables may refer to different “layers” of a same, global timetable, or other types of classifications associated to a Timetable. An Operating Timetable refer to a timetable that is current in execution, thus, to a given time of the day, this Timetable will have a part that is consolidated and has the real arrival and departure time at stations and a part (after the current date and time at which the timetable was requested) that will still consist of the planned stops.

```
OTCommercialTimetable[] GetCommercialTimeTables(OTEntityId[] ids);
OTCommercialTimetable[] GetCommercialTimeTables(DateTime refDate, OTEntityId[] ids);
```

Provides access to a given sets of commercial timetables present in the TCS. Different timetables may refer to different “layers” of a same, global timetable, or other types of classifications associated to a timetable. A commercial timetable lists only stops that are useful from a commercial point of view, such as stops that are used to embark or disembark passengers.

```
OTConnectionConstraint[] GetConnectionConstraints(OTEntityId[] ids);
OTConnectionConstraint[] GetConnectionConstraints(DateTime refDate, OTEntityId[]
ids);
```

Gets the connection constraints that are currently valid.

```
OTRollingStockConstraint[] GetRollingStockConstraints(OTEntityId[] ids);
OTRollingStockConstraint[] GetRollingStockConstraints(DateTime refDate, OTEntityId[] ids);
```

Gets the rolling stock constraints that are available at the given time.

```
OTCrewConstraint[] GetCrewConstraints(OTEntityId[] ids);
OTCrewConstraint[] GetCrewConstraints(DateTime refDate, OTEntityId[] ids);
```

Gets the crew constraints that are available at the given time.

```
OTDelayInfo[] GetTrainDelayInfo(OTEntityId[] ids);
OTDelayInfo[] GetTrainDelayInfo(DateTime refDate, OTEntityId[] ids);
OTDelayInfo[] GetLineDelayInfo(OTEntityId[] ids);
OTDelayInfo[] GetLineDelayInfo(DateTime refDate, OTEntityId[] ids);
OTDelayInfo[] GetStationDelayInfo(OTEntityId[] ids);
OTDelayInfo[] GetStationDelayInfo(DateTime refDate, OTEntityId[] ids);
OTDelayInfo[] GetTimetableDelayInfo(OTEntityId[] ids);
OTDelayInfo[] GetTimetableDelayInfo(DateTime refDate, OTEntityId[] ids);
```

Get the delay information aggregated by: Trains, Lines, Station or Timetable Ids.

7.1.4 Routing Interface

Provided by: Architecture

The Routing interface is used internally within the messaging layer and allows the architecture to determine the endpoints subscribed to specified event types. Whenever a new message is received, the architecture checks the routing interface for a list of interested modules (those specifying the correct TypeId and Version at subscription-time) and dispatches the event as appropriate.

Events can be sent to the architecture by choosing between two different approaches. Client modules can integrate directly using the AMQP protocol, pushing event messages to WP7 message exchanges, as shown in the following picture:

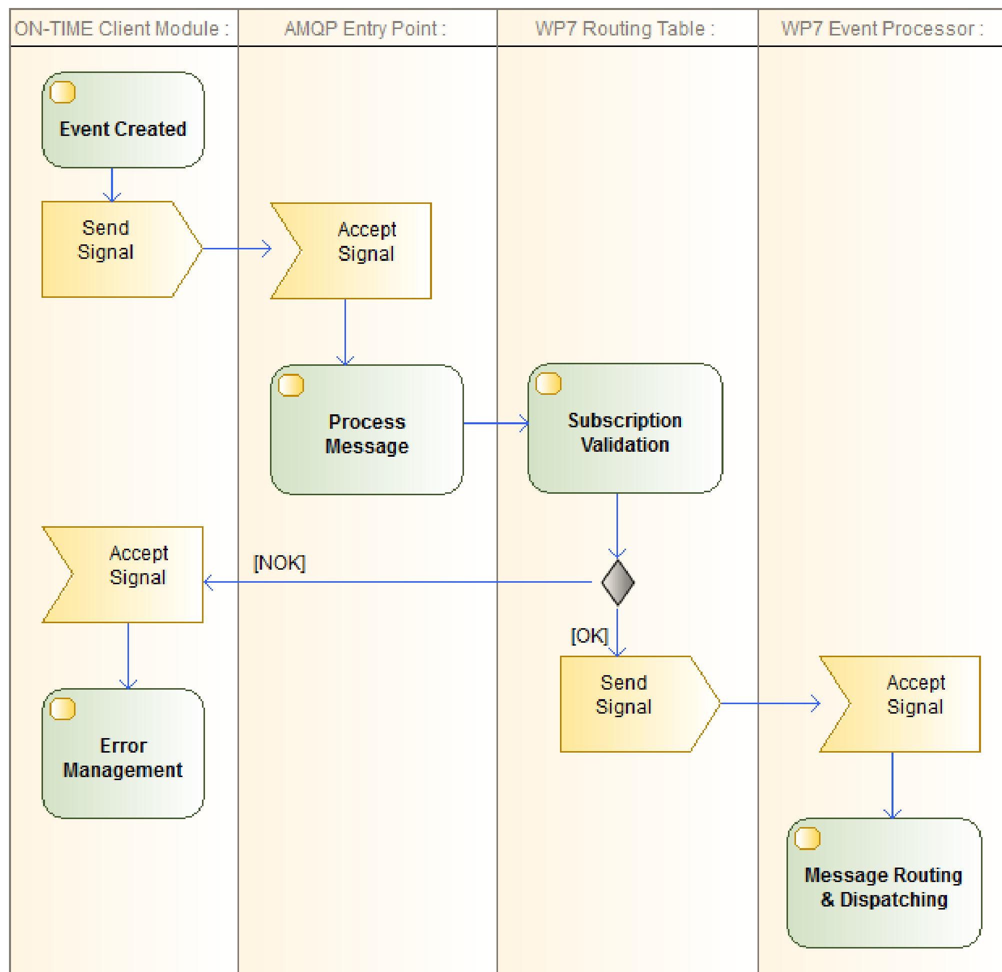


Figure 7 Publishing using AMQP

This approach is preferrable for modules that have a high rate of message publishes, since it does not require any additional layer on the AMQP protocol.

On the other hand, a slight effort should be spent to build an AMQP client implementation. For applications that are not going to have lots of messages to publish, a REST integration point will be possibile. This approach adds to the previous one an additional layer that exposes a REST web service and communicates with the internal WP7 AMQP message queues. Clients make PUSH requests to the REST web service, and it will internally push the received message to the queues. The rest of the processing remains invariate.

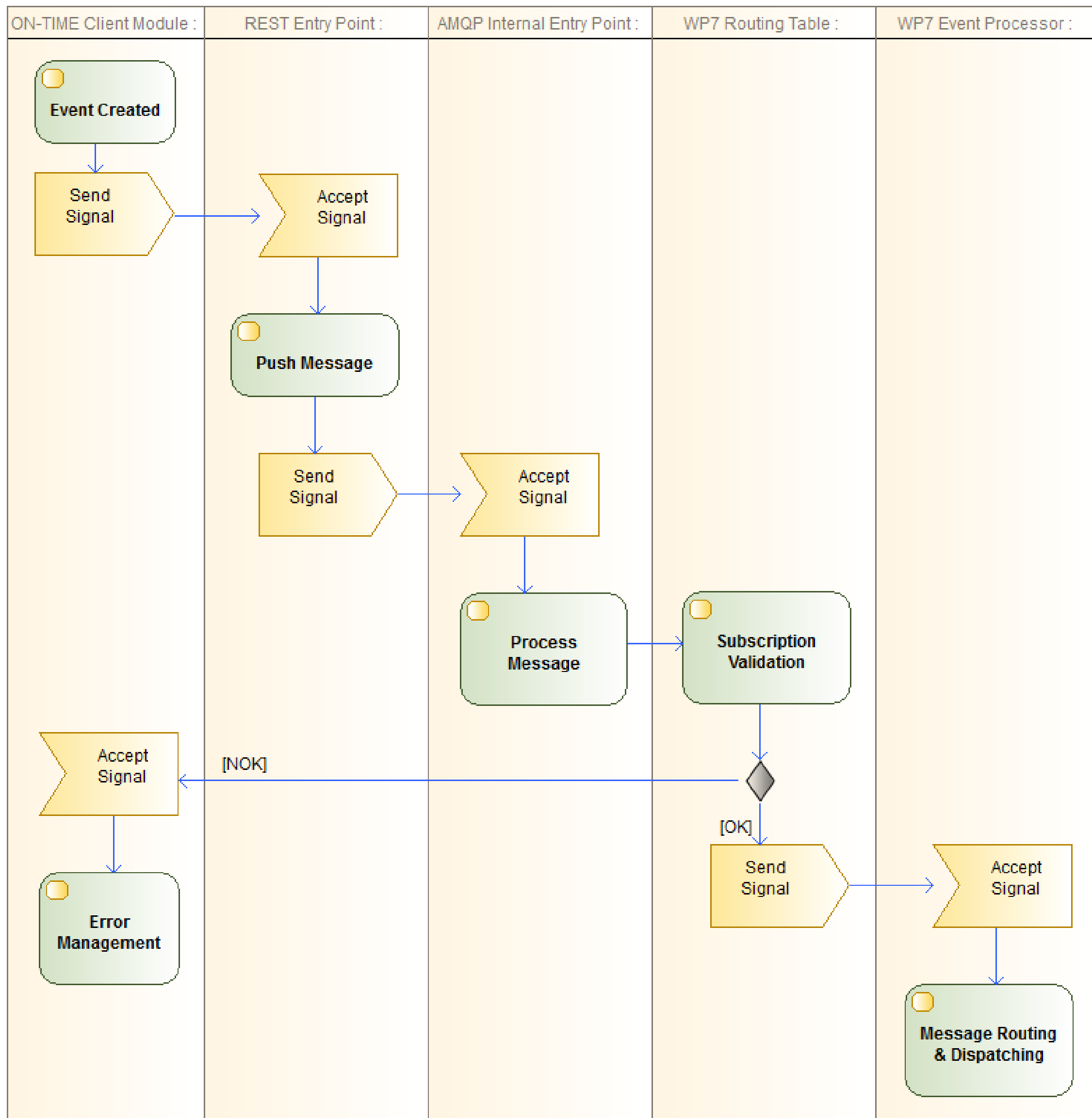


Figure 8 Publishing using REST WP7 Entry Point

For subscriber clients, the possible approaches are the same. Clients can either integrate directly by AMQP, or expose a REST interface that will be used by the architecture to deliver messages.

In the following picture, a direct integration is shown. The client must be AMQP-capable and implement an asynchronous receiver client, as described in the EventConsumer paragraph.

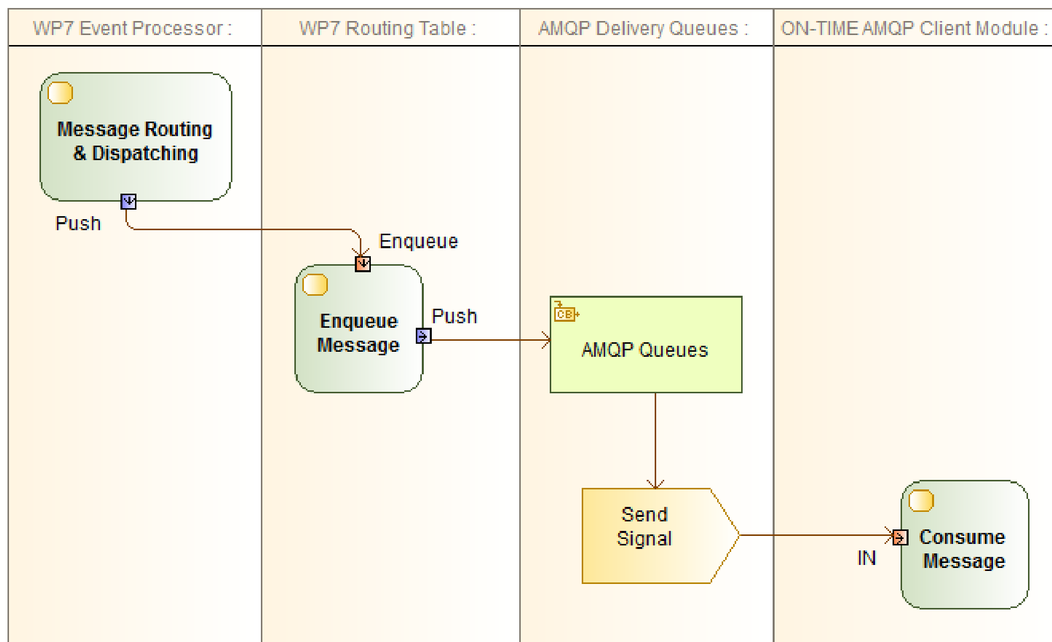


Figure 9 Message delivery to clients using AMQP

The following picture shows the other mechanism that can be used. On the approach seen above, an additional layer is added, that will implement the REST integration point from the architectural side.

This layer will add an internal module that receives event messages from the WP7 AMQP queues and makes PUSH requests to the REST web service exposed by the client. The endpoint of the client-side web service is specified at subscription-time by the module.

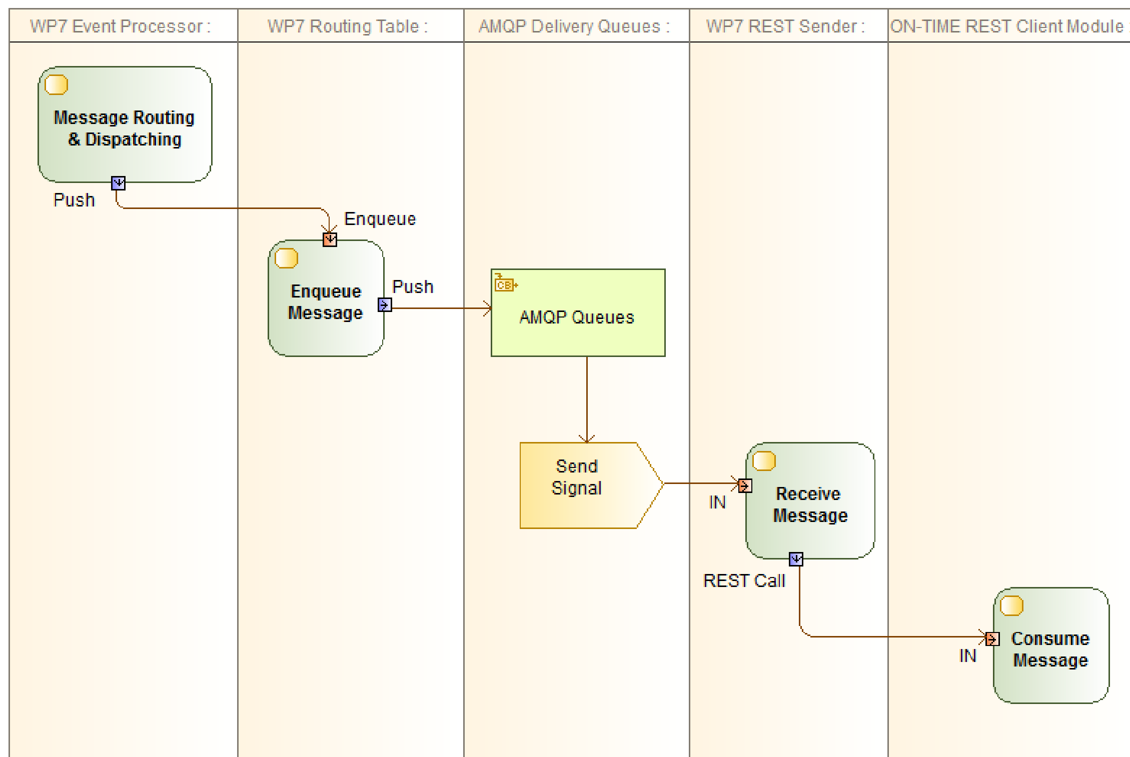


Figure 10 Message delivery using client-side REST web service

7.2 Integration with TCS

The TCS abstraction of ON-TIME is a system (or a collection of systems) that manages the operation and the monitoring of railroad operations, rolling stocks and crews. The TCS is the owner of a great deal of information, such as:

- The physical infrastructure and its state
- The Timetables (both operational and commercial) and their maintenance
- The Rolling Stock Database and its management
- The Crew Database and its management
- The Sensor Network that collects data and monitors the operation

Owing to the crucial, safety-critical role of the TCS in the operation of the railway infrastructure, it is important that the ON-TIME system should not interfere with the management of these data structures. The central nature of the TCS in railway operation makes the TCS abstraction an ideal location for the repository of all "master data" regarding the railway infrastructure and its available resources (in particular in terms of ensuring the most up-to-date view of the world available at any given moment).

The TCS Abstraction will also be responsible for the dispatch of events notifying system modules of the current traffic state and any other changes in the system.

From a functional perspective, the TCS will dispatch the following types of event:

Movements

Movement events will help to track a train and to locate it within the infrastructure. Since different modules need to track the train movements at different level of detail, movement events are classified using one of several frames of reference:

- ***Stations.*** Train Id/Station Id tuple to notify the entering or exiting of a train from a station.
- **Signals.** Whenever a train passes a signal, the train Id and signal Id are notified.
- ***Track Circuits.***
- ***Axle Counters.***
- ***Switches.***

Disruptions

Disruption events deal with the availability/unavailability of different parts of the infrastructure:

- ***Line.***
- ***Track Section.***
- ***Platform.***
- ***Station.***
- ***Switch.***

Trains

This event category relates to the availability of trains and variations in their composition (for example the splitting of 6-car units into 2, 3-car units).

- ***Availability.*** When a new train "enters" the scenario (it becomes available for traffic) or has been suppressed.
- ***Composition.*** The physical composition of a train: number of carriages and so on.

Signals

Signal events will notify the platform of changes in signal state.

- ***State.*** The state of a signal.

As with other modules, upon its activation the TCS will publish a list of the event types it will start to dispatch using the SubscriptionService interface.

The TCS abstraction must be able to receive input from other modules in the system. Since a TCS is a real-time, event-based system (or a collection of systems), key-data entities in the traffic model will be updated using event notifications.

The events that the TCS Abstraction will be able to consume are:

- **UpdateRealTimeTrafficPlan.** This event will include all the details for a new traffic plan of a given node or scenario. It will include new timetables, train scheduling and lines. A train is considered as suppressed if it does not appear in the new traffic plan.

- **Update Constraints**:
  - Crew
  - Rolling Stock
  - Infrastructure
- **Update Disruptions:** Normally disruptions are something that happens in the field. For the sake of demonstration and for other operational needs, disruptions may be planned using the same events that will regulate field operations, including:
  - LineDisruptionEvent
  - TrackDisruptionEvent
  - PlatformDisruptionEvent
  - StationDisruptionEvent
  - SwitchDisruptionEvent
- **Update Trains.** Trains may be placed into the scenario (for example when they begin service), removed from it (when they are exiting the scenario boundary or finishing services) or when they need to be suppressed due to operational (or optimization) needs. Train update events include:
  - TrainEnterEvent
  - TrainExitEvent
  - TrainSuppressedEvent
  - RollingStockChangeEvent

From the perspective of the involved information transfers, most train update events will be identical to those generated by the TCS.

As with other modules in the platform, the TCS abstraction should implement the IOTventConsumer interface in order to be able to receive and manage events coming from the platform.

7.2.1 Integration with WP3 Modules

Since the creation of a complete time-table is not a real-time task but it's a more complex, long, running task, the WP3 output (timetables of a whole year, for example) will be treated as a RailML input for the TCS systems that deals with timetabling operations.

This means that the WP3 modules will provide a RailML timetable as the initial state of the system. Other modules (like WP4) will use and modify this timetable using the Event Model and Data Processor Services of the architecture.

7.3 Integration with WP4 Modules

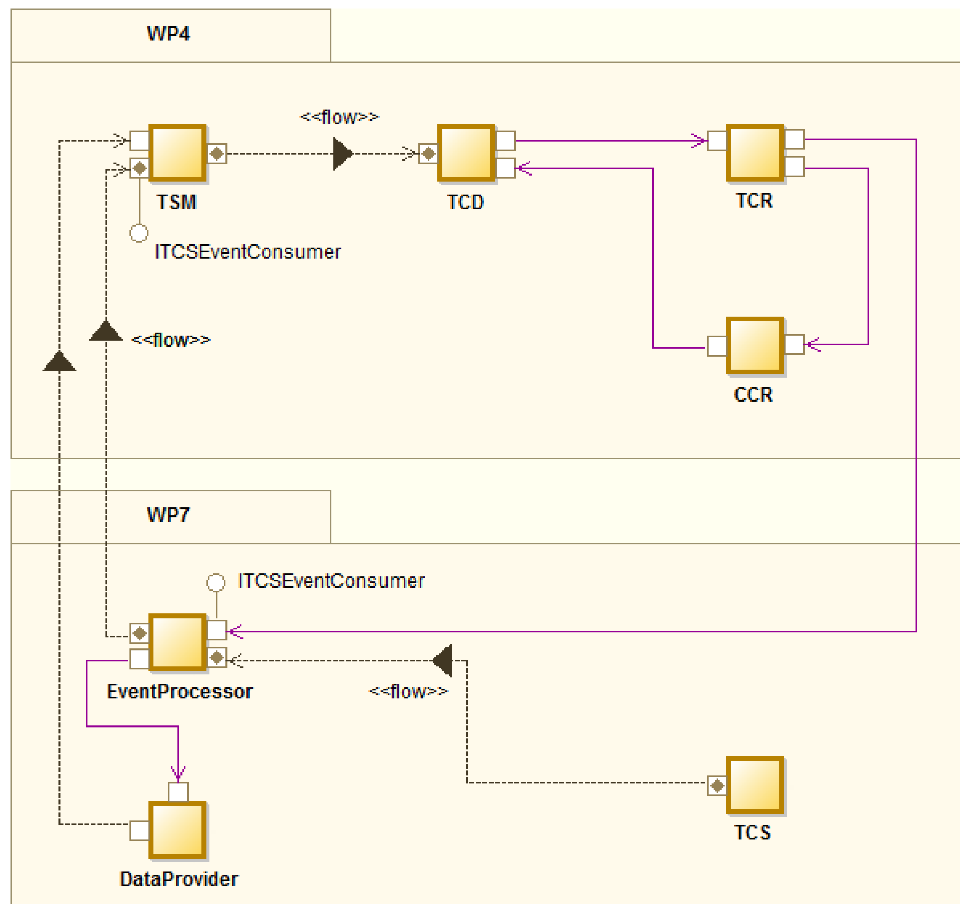


Figure 9 - WP4 Integration Flow

In the context of WP4, the architecture will manage the flow of information between the abstracted TMS and the underlying WP4 System. The WP4 implementation has a single entry point that communicates with the architecture, the TSM (Traffic State Monitoring) module. This module will receive events from the architecture and will process them.

Output will be performed by the TCR (Traffic Conflict Resolution) module that will generate update events for the current timetable, list of train orders and constraint set. These updates will be published and returned to the TMS.

Data communication will be implemented via two different dynamic mechanisms: event-based for real time data exchange and request-based for static and published data structures.

WP4 will consume events of the following types:

- **TDSectionOccupationEvent**
- **TDSectionReleasingEvent**
- **SignalStateChangeEvent**

- **LockedSwitchDirectionEvent**
- **SetRouteEvent**
- **TemporarySpeedRestrictionEvent**
- **InfrastructureUnavailability**
    - Treated implicitly, by issuing modifications to the static infrastructure data.
- **Disruptions events:**
    - RollingStockDisruptionEvent
    - LineDisruptionEvent
    - PlatformDisruptionEvent
    - StationDisruptionEvent
    - SwitchDisruptionEvent
    - TrackDisruptionEvent
- **NewRealTimeTrafficPlanEvent**
    - Used to notify modules that a new traffic plan has been issued. The new traffic plan data will be provided by the TCS systems)
- **TrainRelatedMeasurementsEvents**
    - TrainMassChangeEvent
    - TrainPassengerCountChangeEvent
    - TrainSpeedChangeEvent
    - TrainPositionChangeEvent

WP4 will generate events of the following types:

- **UpdateRealTimeTrafficPlanEvent**
- **SetRouteEvent**
- **RollingStockConflictEvent**
- **CrewConflictEvent**
- **ConnectionConflictEvent**

Conflicts are generated when the real time traffic planning modules cannot provide a good solution to the scheduling problem to be resolved. Connection conflicts may arise when there are disruptions at the infrastructure level.

Infrastructure disruptions are not modelled as events and will be managed at static data level. If a track becomes unavailable, it should be marked as such on the static data (or removed from the data itself). Since WP4 needs to refresh its infrastructure data as often as it runs, this approach may be suitable.

7.3.1 TrainPathEnvelopeComputation module (TPEC)

The TPEC module computes trains path envelopes for trains, starting from a commercial timetable, the current real-time traffic plan and DAS parameters. The result will be, for each train, a sequence of time windows, indicating minimum and maximum passing times for every signal in the train path.

The module will be called by WP4 internally, so no events coming from the architecture are consumed.

As output, the **TrainPathEnvelopeAvailableEvent** is generated.

7.4 Integration with WP5 Modules

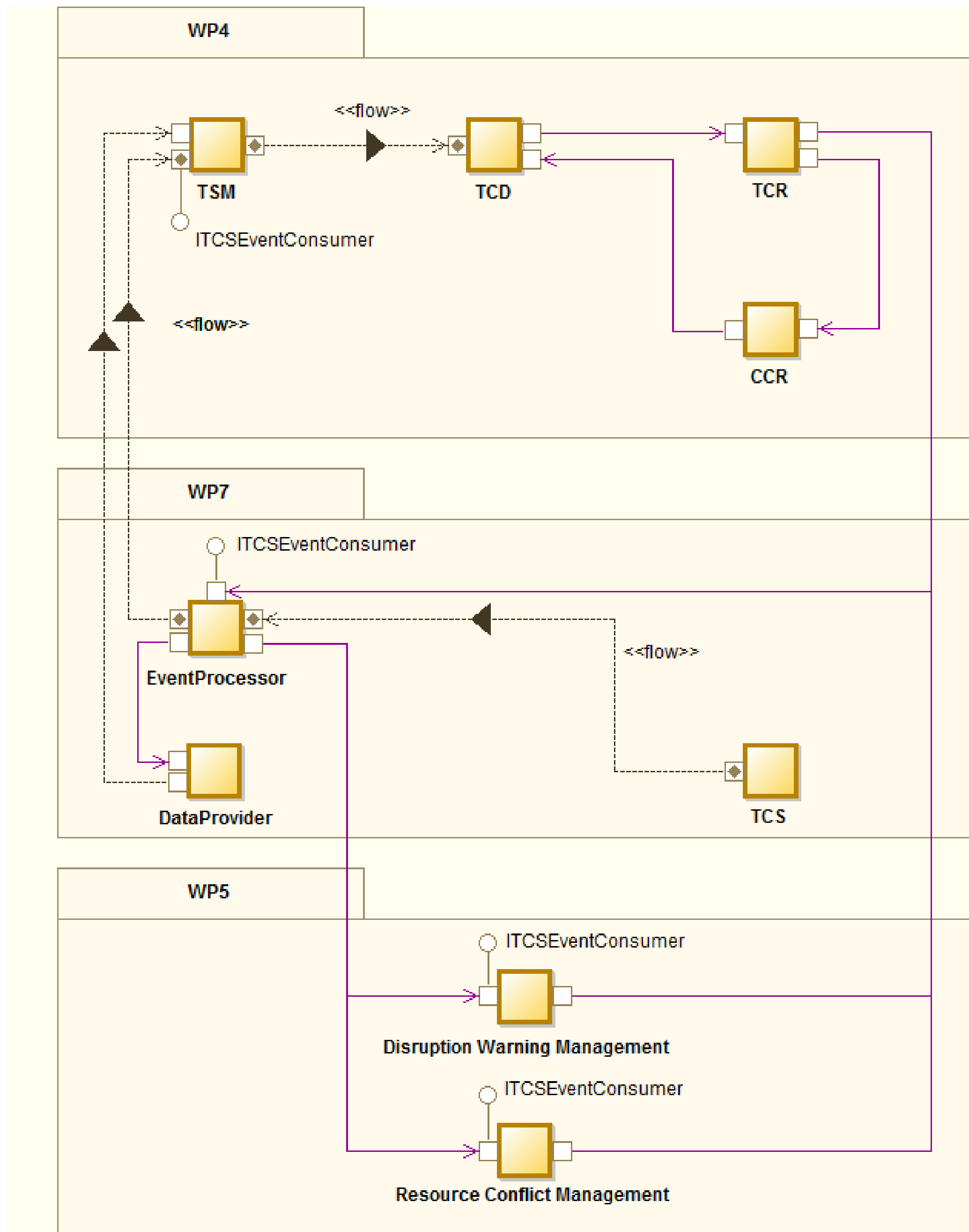


Figure 10 - WP5 Integration Flow

In the context of WP5, the architecture will manage the flow of information between the abstracted TCS and the underlying WP4 System. It will be responsible for the resolution of conflicts that arise because of re-planning operations within WP4 or that are due to changes in the state of the real-world rail network.

In the context of the data structures within the architecture, WP5 will mainly manage resource conflicts such as:

- Crew Conflicts

- Rolling Stock Conflicts
- Connection Conflicts

In addition to conflicts detected from WP4, WP5 will also manage disruption events generated by the TCS. These events could be generated by other systems or simply be collected from the field. The architecture will provide an abstraction by which these messages can be collected.

The WP5 modules will also be responsible for contingency planning in the event of great disruptions. Generally, these plans are not stored in a data structures in the TCS because emergency crews define them as the emergency develops.

WP5 has a strong interconnection with WP4. In keeping with the distributed nature of the system, the modules will communicate via events as described previously.

Here is an activity diagram of the interactions between WP4 and WP5:

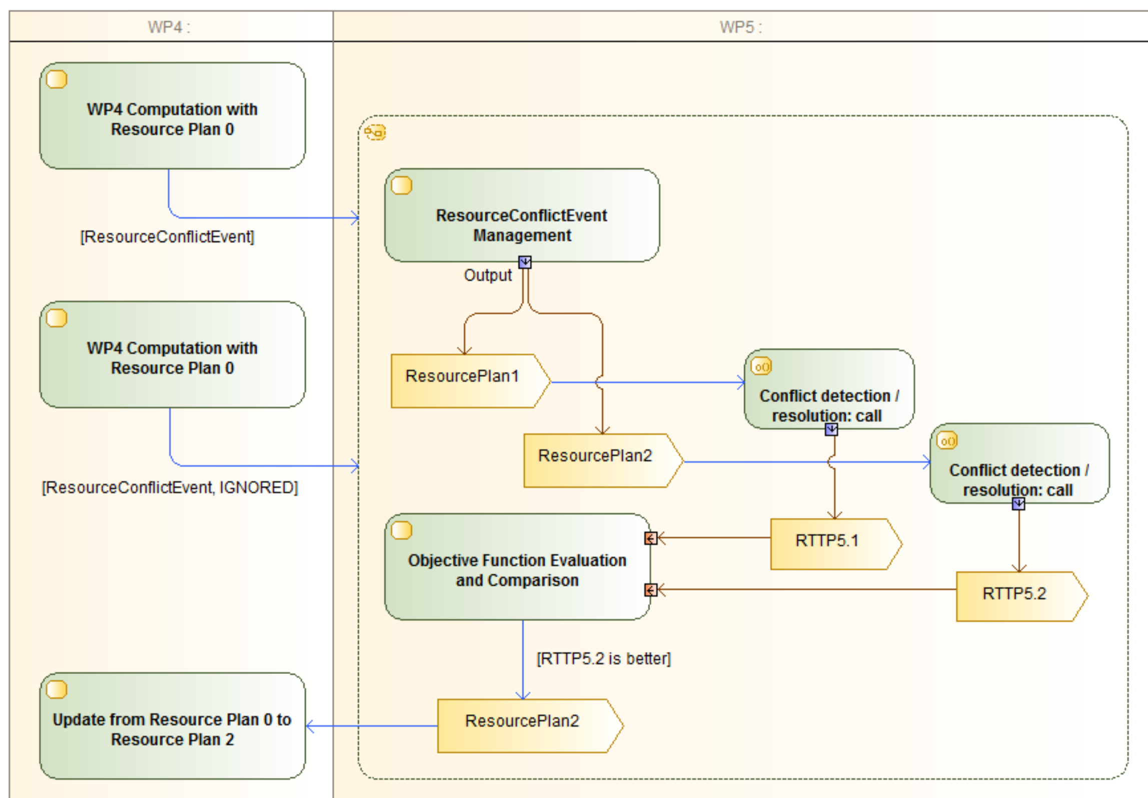


Figure 11 - WP5-WP4 interactions

It is important to note that during the execution of WP5, all other conflicts are ignored and WP4 will continue to work with the old resource plan until the WP5 will publish a new, optimal, one. Access to the new schedules are made using the DataProvider Service of the architecture, to avoid concurrency and data consistency issues. The DataProvider, in fact, will publish only the latest version of the resource schedules to other modules.

Another important feature is that WP5 will try to use WP4 algorithms to produce several resource optimizations alternatives to choose the best one for a given task.

WP5 modules will handle these events to intercept resource conflicts coming from the TCS that will collect them from the field:

- ConnectionConflictEvent
- CrewConflictEvent
- RollingStockConflictEvent

To notify the TCS that there is a change in the resource schedule, WP5 will update the schedules using these events:

- UpdateConnectionScheuduleEvent
- UpdateCrewScheduleEvent
- UpdateRollingStockScheduleEvent

In response to the updates, the TCS will broadcast events to notify other modules that newer versions of Schedules are available:

- ConnectionScheduleAvailableEvent
- CrewScheduleAvailableEvent
- RollingStockAvailableEvent

7.5 Integration with WP6 Modules

According to the functional specification of WP6, Driver Advisory systems have different deployment and functional models. The main concern integrating driver advisory system is to define how the data interface and the flow of information is managed between different technical solutions.

We can provide three different type of architecture:

- **Server-Side management of information.** Data is handled in a centralized server. Every client (the driver advisory console) will connect to the server to receive driving aids. This kind of integration has the most potential in terms of services offered since the server-side infrastructure is more powerful and can access additional services. Communication-wise, the solution is weak because the clients need to have a constant connection (or a connection with few interruptions) to receive data that have to be displayed to the drivers.
- **Client-Side management of information.** Data is downloaded in raw format from a repository and elaboration is made inside the client on-board the train. This solution is the most robust in terms of communication because the connection with a repository has to be made only when the data is needed (it can be done using the communication infrastructures present at the stations) and all the other process can be carried on offline. The solution need more powerful clients and robust protocols able to carry data with very low bandwidth requirements.
- **Hybrid management of information.** In this scenario, some operations are server-side and other are client-side. In most cases, this is the right trade-off since this approach will allow for a rich server-side data management and a more agile client-side data management when the locomotive is offline, for example.

Since all these approach are industry standard solutions, the architecture needs to comply with these different interaction models without penalizing any of them.

It is important to stress that, from the architectural point of view, the most important part is managing a consistent data flow and efficient, standard channels of communication. Looking at the requirements for WP6 modules, we can locate two different kind of data: static and dynamic infrastructure and operational data, data that can be changed according to events (in this case, the train envelopes produced by the WP4 modules).

To maintain a general, coherent implementation of the data flow, WP6 models need to know when a train envelope change and have APIs to recover these data, if needed.

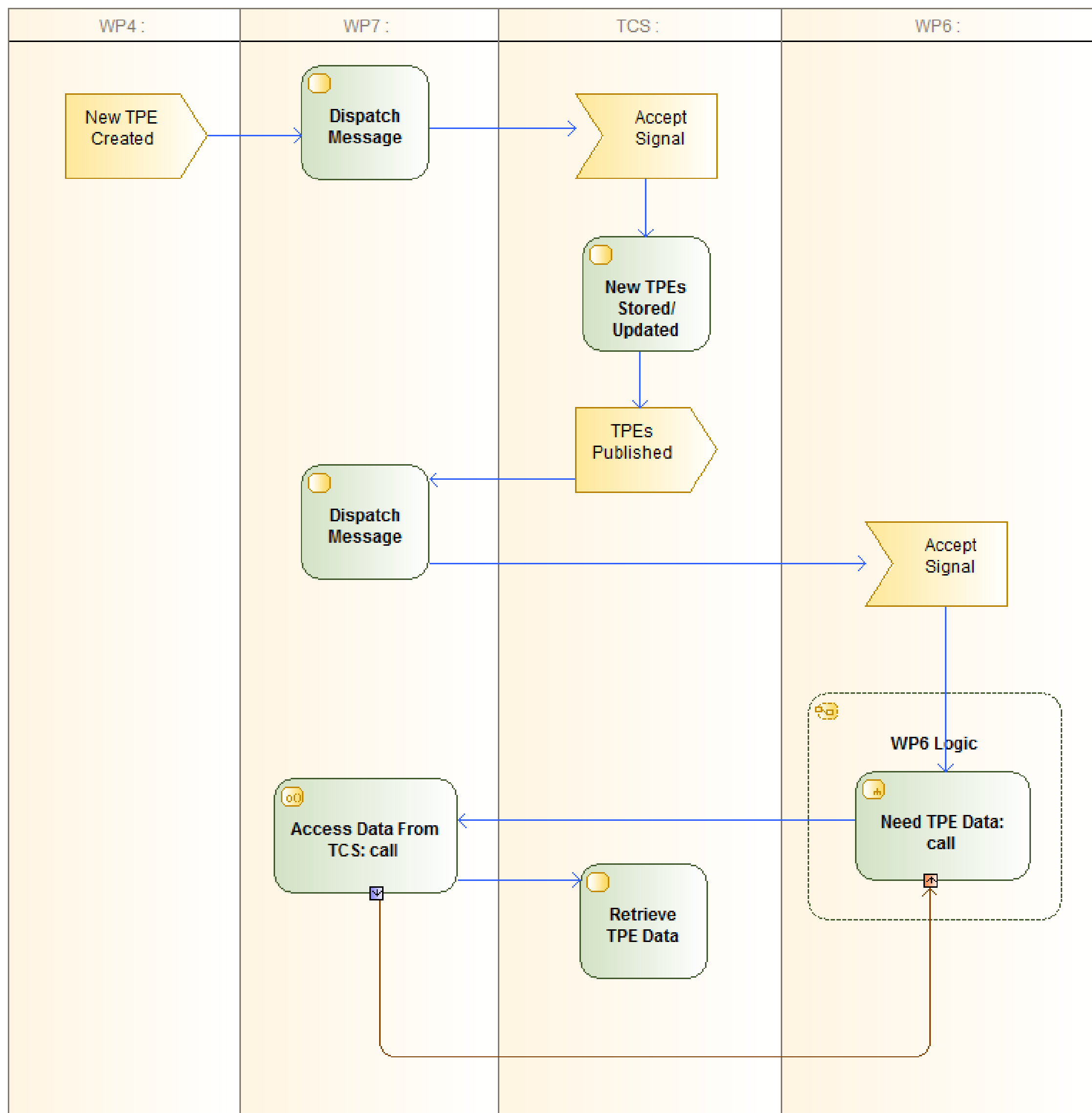


Figure 11 - WP6 Integration Flow

From a process point of view, when WP4 creates a new Train Envelope, it will notify the architecture. The availability of a new collection of train envelopes will trigger this chain of actions:

1. The Architecture Event Processor will forward the UpdateTrainPathEnvelopeEvent to the TCS system that subscribed it.
2. The TCS system consumes the Event Payload and stores the new Train Path Envelope, ensuring data consistency.
3. The TCS system publish a TrainPathEnvelopeAvailableEvent to the architecture.
4. The WP6 module will consume the TrainPathEnvelopeAvailableEvent and will take notice of the TRAIN_IDs included in the payload.

5. When the internal logic of the WP6 module will require it, the WP6 module can ask the Architecture Data Provider for the TPE associated to a TRAIN_ID for which it has received a notification of availability.

The TCS will store TPEs in a transactional, FIFO-like pattern. This means that every data transaction will be an atomic, first come, first served communication.

Since events have data that can be used to order communication streams (such as timestamps, sender information and so on), it's left to the WP6 implementation to provide means to implement message-ordering mechanics or other data consistency patterns, since they can vary upon the different kind of implementation a WP6 module may have.

8 REFERENCES

Tanenbaum, A. S., & Van Steen, M. (2002). Distributed systems (Vol. 2). Prentice Hall.

Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A. M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, *35*(2), 114-131.

Richardson, L., & Ruby, S. (2008). *RESTful web services*. O'Reilly Media.

Pautasso, C., Zimmermann, O., & Leymann, F. (2008, April). Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web* (pp. 805-814). ACM.

Membrey, P., Plugge, E., & Hawkins, T. (2010). *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress.

Ramakrishnan, R., & Gehrke, J. (2000). *Database management systems*. Osborne/McGraw-Hill.

Vinoski, S. (2006). Advanced message queuing protocol. *Internet Computing, IEEE*, *10*(6), 87-89.