Scalable Software Framework for Real-Time Data Processing in the Railway Environment

ABSTRACT
To this day, railway actors obtain information by actively hunting for relevant data in various places. Despite the availability of a variety of travel-related data sources, accurate delivery of relevant, timely information to these railway actors is still inadequate. In what follows we present a solution in the form of a scalable software framework that is able to interface with almost any type of (open) data. The framework aggregates a variety of data sources to create tailor-made knowledge, personalised to the dynamic profiles of railway users. Core functionality, including predefined non-functional support, such as load balancing strategies, is implemented in the generic base layer, on top of which a use case specific layer – that is able to cope with the specifics of the railway environment – is built. Data entering the framework is intelligently processed and the results are made available to railway vehicles and personal mobile devices through REST endpoints.

Keywords: Scalable, Reliable, Load Balancing, Data Processing, Railway, Framework, Travel Information, Passenger Information, Personalised, Profiles, Real-Time, Data Aggregation, REST, OSGi, Reasoning, Semantics

INTRODUCTION
Today more than ever, public transportation operators are aware of the importance of investing in their passengers. Moreover, meeting passengers’ public transportation needs, in addition to important environmental aspects (Hua, 2016, Pålsson & Kovács, 2014, Guerra et. al, 2016) is acknowledged as a central goal in the European Commission’s transport strategy roadmap. (European Commission, 2011) In this roadmap, it is shown that acting on issues important to passengers, such as reducing noise in a train or providing wireless connectivity, enriches the customer experience. At the same time, due to the current information-centric nature of society, passengers expect public transportation to be more and more augmented/personalised with information from various sources (e.g. social networks, multimodal travel information) (Sierpiński, 2017, van Lier et. al, 2014). Recent ICT developments present opportunities to meet passengers’ rising expectations. As a result, the amount of available mobile travel applications offering travel information to passengers has grown exponentially (Gardner, Haeusler & Tomitsch, 2010). Contemporary mobile applications, such as the travel information apps provided by European railway companies e.g. NMBS, NS, Deutsche Bahn, National Rail and SNCF, mostly offer Real Time Train Information (RTTI) about arrival and departure times as well as mobile ticketing services. Many mobile RTTI applications are mostly context specific, single-purpose applications that provide a solution to a particular problem or requirement. However, according to ORR, the (British) Office of Rail and Road, passengers want to receive live information and they want it at their fingertips (Office of Rail Regulation, 2012). Since the initial publication of this report, where train operators are required to provide appropriate, accurate and timely information to enable (prospective) passengers to plan and make their journeys with a reasonable degree of assurance, including in times of disruption, the ORR has elaborated on this key requirement. The ORR requires from a train operating company to publish a code on practice setting out how it will ensure compliance with this directive. In 2014, further studies and surveys were conducted to see whether passengers had noticed tangible improvements. The conclusions highlighted some improvements, but also raised a number of areas where special attention was needed. As a result of this, the ORR published a new regulatory guidance, in collaboration with the industry (Rail Delivery Group, 2016) which issued 50 recommendations, in 2016 (Office of Rail and Road, 2016).
According to the Danish Rail operator DSB for instance, a delay is often not experienced as being problematic, as long as passengers are assisted and know how long they will have to wait, how they could move on from the next station, whether there is still time to grab a coffee, etc. Access to RTTI positively changes passengers’ perception and experience of the quality of the public transportation service. Unfortunately, RTTI as it is currently offered to passengers is still mostly passive (passengers have to actively search for the information they need), based on a single source (i.e. the database of the train operator) and largely not tailored to the (dynamic) personal needs of the individual passenger, making it difficult to find relevant information when needed. As such, passengers expect travel information to become more context-aware and more personalised. Another fundamental, yet often unapparent, requirement of RTTI is its reliability in order for the public to be able to confidently use the information available. The full potential of (open) data for railway travel has clearly not been materialized yet.

The problems related to RTTI discussed above, together with changing passenger expectations form an interesting gap in the passenger experience that the software framework presented in this chapter aims to tackle. This software framework was developed in the Flemish iMinds ICON research project TraPIST (acronym for Train Passenger Interfaces for Smart Travel).

The approach taken by this framework differs from existing travel applications in many ways. First of all, passengers do not need to actively search for the information they need. Instead, personalised up-to-date information is sent to them during the course of their travels. Second, the information that they need is offered to them when they actually need it (e.g. information about alternative connecting trains only become relevant at the moment a passenger is likely to miss his/her intended connecting train). Third, the way the information is offered is highly personal and adapted to the context of the passenger (e.g. his/her travel goal, destination, activities or company). Another way in which the proposed solution differs from existing travel applications is the fact that it is built to easily and reliably connect to a multitude of data sources. It is not solely practical travel information (e.g. from the train operator’s database) that is relevant for passengers, but instead a combination of several (existing) sources of data to offer passengers the most interesting and relevant information (e.g. tourist event information for leisure travellers, weather information, safety travel tips for the visited area), including publicly available (open) data sources.

Capturing the requirements and needs from the end-users has been executed in a multi-disciplinary approach, where the needs were extracted using novel user experience techniques. Although not the main topic of this chapter, we refer the interested reader to (Slegers et al., 2015).

The remainder of this chapter is structured as follows. The next section presents the background motivation for the research and development of the LimeDS framework, as well as introducing a number of relevant related work, both in the context of provisioning platforms as well as in the context of Semantic Web technologies. Subsequently, the challenges and issues motivating the design of the framework, exemplary railway related use cases and corresponding requirements are detailed, followed by an extensive presentation of the architectural design and internal details. Two Proof-of-Concept demonstrators are illustrated in the second-to-last part of this chapter. Finally, to conclude, a number of performance metrics were evaluated and are presented together with the envisaged future research directions.

**BACKGROUND**

Given these basic first requirements, in addition to the more elaborate listing later in this chapter, it is clear that a highly scalable and customisable (micro service) framework imposes itself, allowing to tie together different data sources, apply application logic to this data and deal with non-functional issues such as reliability and robustness (highly important in the case of the harsh railway environment), performance, caching, etc.

The framework allows application developers to focus on improving the Quality of Life (QoL) of passengers instead of having to focus on the technical aspects. A visual approach to the construction and adaptation of services is an added nice to have and aids in rapid construction of new applications and realisation of new ideas. In other words, the technology should help developers realising their ideas w.r.t. innovative data-driven applications, rather than having to focus on the low-level technical iterative wiring.
Next to the traditional non-functional support such as caching and load balancing, the framework presented in this chapter also allows semantic processing of and reasoning on the data passing through, allowing e.g. highly personalised data relationships to be extracted.

In the following subsections, an overview is presented of the two most important aspects in the proposed approach, namely both on the facilitating platform as well as on the support for semantic technologies. The presented LimeDS framework has historically grown out of research into provisioning systems for the railways. Therefore, the example used in this chapter are mainly examples in a transportation setting. It has to be noted however, that the framework is broader applicable to other domains as well. The challenges defined by means of railway use cases have provided input for the definition of a number of general requirements.

**Related Work on Service Provisioning Platforms.**

The presented platform, LimeDS (Lightweight modular environment for Data-oriented Services) is an open source toolkit for building JVM-based (Web) applications with a strong focus on developer productivity and interoperability with other services and devices. It allows developers to immediately focus on the use-case specific implementation by drastically reducing typical boiler-plate code for setting up HTTP endpoints, managing dependencies, configuring services, etc. This is made possible thanks to a visual editor used to connect (micro) service components (= Segments) together to form modular logical units (= Slices), that then can be deployed to any LimeDS runtime. This is illustrated in Figure 1 below. The individual blocks represent the Segments. All such Segment instances together form a Slice. For more advanced users, a Java API is provided that allows more complex modules to be created. Below we compare LimeDS with a number of related platforms and technologies.

![Figure 1: Segments and Slices in LimeDS](image)

Java API for Restful Web Services (JAX-RS) (Burke, 2009, Burke, 2013) is the Java standard specification for a Java API that allows implementing REST-based applications. It relies on annotations added as metadata on classes to transform these into REST resources that are exposed as HTTP-endpoints. Various
implementations of this standard exist ranging from Jersey (Oracle Corporation, 2015), to Resteasy (RedHat, 2015) and Apache Wink (Apache Software Foundation, 2015). LimeDS uses Apache Wink as a base layer and thus offers support for JAX-RS resources. On top of this, the Data Flow mechanism allows developers to build Web API with integrated support for reliability, scalability and security with a minimal amount of code that can easily be modified at-runtime.

Dropwizard (Dallas, 2014) is an open-source Java project that integrates a number of mature libraries into a light-weight ecosystem targeted at developing REST-based applications. This is done by allowing JAX-RS resources to be wired with various support services, ranging from logging, to storage and authentication / authorisation. The goal of Dropwizard is similar to that of LimeDS, but the focus and scope differs. While Dropwizard is a very powerful framework to build high-performance REST-based applications that are easy to monitor and maintain, LimeDS provides a more dynamic environment where changes can be made on-the-fly while the application is running. Dropwizard also focuses on a single-node setup, while LimeDS can be configured to run as a cluster with load balancing.

OSGi (OSGi Alliance, 2003) is a proven standard that specifies a modular middleware framework for Java. OSGi implementations such as Apache Felix or Eclipse Equinox (The Eclipse Foundation, 2015) provide an execution environment where software modules (bundles) can be installed at-runtime and communicate safely in highly dynamic circumstances. While a lot of applications could benefit from the (at-runtime) modularity and loose coupling of the OSGi framework, a lot of developers struggle to unlock its true potential because of the steep learning curve. LimeDS is built on top of OSGi and can offer developers OSGi features such as dynamic reconfiguration of dependencies and at-runtime addition of new functionalities in a much easier to use package - albeit for a more restricted set of REST-based applications. The Data Flow mechanism of LimeDS is loosely based on principles of Business Process Management (BPM) solutions, such as JBoss BPM Suite (RedHat, 2015), where interactions between components can also be modelled as a visual process. The difference is in the scale of the application domain. While BPM is typically used to implement the complex orchestration of distributed software systems, LimeDS will model interaction patterns of micro-services which are primarily found on the same system, although LimeDS Data Flow components can depend on external remote data sources and can be distributed over multiple systems, e.g. for load balancing purposes. A BPM instantiation could however integrate a LimeDS-based service in order to implement a certain process on a higher level. As a result LimeDS can be a much more lightweight option in comparison with traditional BPM solutions, depending on the scope of the problem.

Vert.x (Clement Escoffier, 2015) is a Java-based framework that allows developers to build 'reactive' applications by setting up modular components called Verticles and by routing data between these components using a distributed event bus. The 'reactive' keyword refers to the framework using event-driven, non-blocking calls and the fact that it is designed from the ground up to support high availability. Some of the additional services provided by Vert.x are: (i) the ability to use multiple programming languages (Java, JavaScript, Groovy and Ruby), (ii) integration of storage solutions (MongoDB, SQL and Redis), (iii) support for clustering, advanced capturing of metrics on a per-component basis and (iv) built-in support for authentication and authorisation. Vert.x and LimeDS overlap in some features, but the latter focuses on HTTP/JSON-based applications with a clear choice to build upon the services model of OSGi, while Vert.x offers a more generic approach. When developing those type of applications, LimeDS has an edge as there is no need for boilerplate code in setting up the HTTP endpoints.

Additionally, LimeDS has built-in support for semantic reasoning. Arguably the most important concept in the Semantic Web is the use of ontologies. They describe in a formal and well-defined way the concepts and relationships between these concepts in a particular system, often referred to as the domain. Apart from the fact that OWL is a well-defined vocabulary for describing a domain, because of its foundation in Description Logics, the model described in this formalism can be used as input for a reasoner to check consistency of the model and infer extra knowledge out of the model. Relevant related work on this last topic is presented separately in the next section.

Related Work on a Number of Relevant Semantic Web Technologies
A short, but comprehensive definition of an ontology, based on the definition by Gruber in (Gruber, 1993) is: “An ontology is a formal specification of an agreed conceptualisation of a domain in the context of knowledge description.” Accordingly, an ontology describes in a formal manner the concepts and relationships, existing in a particular domain or system and using a machine-processable common vocabulary within a computerised system. It can also contain classification rules. This standardised representation of the semantics of a domain can then be used to exchange data and its attached domain model. This way an ontology encourages re-use, communication, collaboration and integration (Fernández-López, Gómez-Pérez & Juristo, 1997). By managing the data about the current context in an ontology, intelligent algorithms can be more easily defined that take advantage of this information to optimize and personalize applications.

OWL has different levels of expressive power, each of them varying in their trade-off between expressiveness and inferential complexity. They are, in order of increasing expressiveness: (i) OWL Lite: supports classification hierarchies and simple constraint features, (ii) OWL DL: OWL Description Logics, a subset providing great expressiveness without losing computational completeness and decidability and (iii) OWL Full: supports maximum expressiveness and syntactic freedom, however without computational guarantees. Using one of the three sub-language flavours of OWL, one can easily adapt to the required expressiveness. Arguably, the most interesting sublanguage for many application domains is OWL DL, balancing great expressiveness with inferential efficiency. Due to its foundation in description logics, OWL DL is also very flexible and computationally complete. The constraints imposed by the adoption of the DL formalism assure that a decidable reasoning implementation can be achieved. OWL can be seen as the evolution of several previous WorldWide Web Consortium (W3C) recommendations, being XML, XML Schema, Resource Description Framework (RDF) and RDF Schema. In this view, each step introduces more functionality, i.e. XML introduces a common syntax, XML Schema introduces data types and structure, RDF focuses on meta-data, allowing to say anything about anything, RDF Schema introduces RDF resource types and finally OWL specifically supports the construction of vocabularies and shared meanings.

Ontologies are considered as dynamic and evolving in time and are also tailored towards the distributed nature of the Web. Some of the serialisation formats of OWL are based on eXtensible Markup Language (XML), namely RDF/XML (Beckett and McBride, 2004) or OWL/XML (Hori et. al, 2004), others being Turtle (Beckett et. al. 2008) or Manchester Syntax (Horridge et. al. 2006). Serialisation is the process of converting an in-memory object or data structure into a format that can be persisted on disk or transmitted by some communication medium. Each of those potential serialisations serve different purposes. An elaborate discussion can be found in (Hitzler et. al, 2012).

In recent years, research and standardisation efforts have taken OWL into a next level, namely OWL 2. In contrast to the earlier decomposition of the description logics in OWL Light, OWL DL and OWL Full, OWL 2 (World Wide Web Consortium, 2012) specifies three new types of sublanguages, called profiles, which have favourable computational properties and are easier to implement, namely OWL 2 EL (Existential quantification Logic): useful in applications employing ontologies that contain very large numbers of properties and/or classes, OWL 2 QL (Query Language): aimed at applications using very large volumes of instance data, and where query answering is the most important reasoning task and OWL 2 RL (Rule Language): used for applications which require scalable reasoning without sacrificing too much expressive power. The profiles are designed for increased efficiency of reasoning for specific types of applications.

JSON-LD (World Wide Web Consortium, 2014) is a JSON-based format to serialise and message data captured in an ontology. The syntax is designed to easily integrate into deployed systems that already use JSON, and provides a smooth upgrade path from JSON to JSON-LD. OWL can be easily integrated with various rule platforms. This is demonstrated by a.o. the Semantic Web Rule Language (SWRL) W3C Submission (Horrocks, et al. 2004).

In description logic terminology, the T-Box contains the axioms defining the concepts and relations in an ontology, while the A-Box contains the assertions about the individuals in that domain (Carroll et. al, 2004). OWL, however, does not explicitly make this distinction because model and data can be mixed in the same
description, but for clarification purposes it is still beneficial to make this distinction. An example to illustrative the difference between T-Box and A-Box concerns a simplified description of my Personal Computer (PC). The T-Box description of a Dell Latitude E5400 laptop (concept) states that this type of computer has one serial number and an Intel(R) Core(TM)2 DUO P8600 Central Processing Unit (CPU) (concept). The processor concept also has one serial number. A corresponding A-Box instantiation could that the Dell Latitude E5400 with serial number 53-373-312-19 has the specific Intel(R) Core(TM)2 DUO P8600, with serial number 6.1.7600.16385.

The main reason for creating an ontology of all the concepts within a domain, such as for example the previous computer example, is that logical connections and relationships can be described between the concepts in the domain. This allows for inference to be applied on the ontology. The concepts in the domain need to be described, as well as the relationships and constraints that define the concepts. Once the ontology is constructed, inference rules can be declared about the concepts and their properties within the ontology. A semantic reasoner is a piece of generic software, which is able to infer logical consequences, i.e., new knowledge, out of the information captured in an ontology. A wide range of mature reasoners exist today, e.g., Pellet (Sirin, et. al., 2007), Hermit (Glimm, et al., 2014) and Fact++ (Tsarkov & Horrocks, 2006). SPARQL (Prud'hommeaux & Seaborne, 2008) can be used to query data captured by an ontology. The leading language for encoding ontologies is OWL (McGuinness & Van Harmelen, 2004).

Three generally accepted conceptual phases can be distinguished in an inference process. These are: (i) Consistency checking and satisfiability: This process analyses the ontology model to find any contradicting axioms. For satisfiability, this conforms to determining whether a concept can have individuals. In case a concept is unsatisfiable but does have individuals defined, the ontology is inconsistent; (ii) Classification: Depending on the definitions of all named concepts in the ontology, this process completes the already existing asserted concept hierarchy with the inferred one; and (iii) Realisation: Since it is possible for a given individual to belong to more than one concept, this phase calculates the set of concepts to which each of the individuals can belong, taking into account the complete inferred concept hierarchy.

**Main Focus of the Chapter**

**Challenges**

**Interoperability**

Interoperability can be defined as the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties. The LimeDS framework requires support for interoperability with both a wide range of different data sources or services and with various passenger applications that will depend on the framework in order to receive filtered, enriched and up-to-date information.

Nowadays, every passenger has his/her own device, but it does not stop there, there are a lot of different public displays both in stations and within the trains themselves, all with different form factors and capabilities, each conveying different information. This means LimeDS has to be designed with this variety in mind. The public transport scene is a very dynamic environment. It is dependent of the time of day, passenger needs can vary enormously, timetables can change dynamically, and passengers must be informed as soon as possible at all times. The framework has to make sure that it can withstand the quickly changing requirements that such an environment implies. To illustrate this with an example: new

![Figure 2: Interoperability is a key feature of the LimeDS framework](image-url)
public data may become available, that can enhance an already delivered service, and we want the framework to help in quickly integrating that, without much effort from a developer perspective, that is.

**Extensibility**

Extensibility can be defined as the ability of a system to incorporate future growth on a functional level. Extensible systems are designed to support the addition of new features or modifications while minimizing the impact of these changes on existing parts of the system. A high level of extensibility is required in order to be successful:

1. The more data sources that can be supported, the more added-value information can potentially be provided to the end-user.
2. When more data sources become available over time, it is only natural that the reasoning logic that processes this data will need to be extended as well.
3. For some aspects, the framework will have to rely on modules developed by third party developers. The framework will have to provide so called service hooks where this applies, in order to give those third party developers the tools to extend upon existing functionality without compromising the overall system operations.

**Scalability**

Scalability is the ability of a system to either handle increases in load with little to no impact on the performance of the system, or the ability to be readily enlarged. The framework must be able to support a large amount of data sources while still being able to process this input in a relatively small period of time. There is also the implicit user load that the framework needs to take into account. While we expect the framework to serve a whole range of information consumer applications or services, the potential user base for each of these consumers can be very large and the framework must support large numbers of users concurrently requesting information through the information consumers. On the other hand, this also suggests the framework must be able to withstand certain peeks of traffic, since use of the service is likely to fluctuate greatly throughout the day.

**Usability**

The usability of a system is determined by how easy it is for users to interact with it in order to achieve the desired goals. While the focus of the framework is on interoperability, extensibility and scalability, easy-of-use for applications developers has been taken into account.

**Intermittent connectivity**

Another challenge is the harsh environment in which public transport sometimes operates. Software has to be robust, since maintenance crews cannot always intervene in a timely manner. Communication to and from the train is not always guaranteed. The on-board systems need to be operational even when no new information from the wayside can be received. So for instance: A train driving through a large barren area, without data-coverage, should still be able to display and show information on-board for its passengers. The same holds true for areas which do have data coverage.

LimeDS looks at every connection as being unreliable and tries to cope with this, e.g. if a train enters a long tunnel through a mountain, the framework should somehow detect this, and fall back to a local mode, so that it can still inform the passengers as much and as accurate as possible. This can be done by storing data ahead of time, or falling back to other algorithms (for situating the train for instance) while the main service cannot be reached.
Framework Use Cases

The goal of the software framework is to provide an end-to-end solution for processing data from multiple sources into useful information that can be provided to a wide range of passenger applications. Figure 4 shows an overview of how this framework is envisioned conceptually.

High volumes of heterogeneous data are transformed by the framework into (digested) information by passing the data through a number of reasoning / processing components involved with near real-time data consolidation and the dynamic profiling of passengers. In the following subsections we describe in more detail a number of use cases for the software framework that are essential in order to achieve this. For the sake of clarity and ease of reference, the use cases are split up according to the involved actors: content providers, developers, system administrators, devices running the LimeDS framework and finally information consumers. Note that the presented use cases are not intended as a complete definition of features supported by the LimeDS framework, but rather as a set of scenarios by which the resulting Proof-of-Concept can be evaluated.

Content Provider

A content provider can be any instance that provides data to the system, either actively or passively. This can be an external instance (e.g. the framework using the news feed of a newspaper) or an internal instance (e.g. a train equipped with the framework posting its GPS data to all participating devices). Two of the use cases apply to a content provider.

1. **Provide access to data**: a content provider makes a data source available to the framework and supplies (developers with the necessary information to implement) a module that allows the framework to extract data from this source.
2. **Push new data**: a content provider actively pushes new data into the system using the framework API.

Developer and System Administrator

In the context of LimeDS, developers are employed to expand upon the original system, offering additional functionality in packaged modules to the framework. The administrator role extends from the typical responsibilities such as managing the users and devices, to deciding which of the provided modules are actively deployed in the framework. This level of control is required as third party developers can also
provide modules, but their contributions need to be validated before they are put into use. For a developer, we define the following use cases:

1. **Add input controllers**: developers can implement additional input controller modules which allow the framework to extract data from new data source types. This use case is closely related with “Provide access to data” in the Section ‘Content Provider’.

2. **Add reasoning components**: developers can implement additional reasoning component modules or incorporate new knowledge descriptions (e.g. in the form of rules) which enable the framework to make new conclusions based on the input data in order to extract new information.

3. **Add information consumers**: developers can implement information consumers which can be distributed and deployed by the framework. Information consumers can be any software component that uses the framework to enhance the experience of the passengers, crew, etc. Typically, this will be a mobile application or an application driving public displays on board a train or in stations.

For an administrator we define the following use cases:

1. **Manage devices**: administrators are responsible for setting up new framework devices and to review the status of existing ones. Registered devices can run the reasoning modules available in the framework. The framework must facilitate administrators in performing these tasks.

2. **Manage modules**: administrators must authorise modules before they can be deployed and installed. They also need to be able to revoke this authorisation at a later time.

3. **Manage users**: administrators are responsible for the management of users and roles and the assignment of the required user rights.

**System Device and Information Consumer**

A system device in this context is defined as any device that is equipped with the LimeDS framework. An information consumer is a piece of software that is dependent on the LimeDS framework; this can include Android or iPhone apps running on the passenger’s phone or native applications that drive the content being displayed on the train screens. Figure 5 shows the use cases that apply to a system device and how these are related to those of the information consumer.

![Figure 5: System Device and Information Consumer](image)

For the system device, we define the following use cases:

1. **Accept input**: each system device must be able to accept input data in a uniform way, either through an API or by using input modules that can actively fetch data from a source.

2. **Host reasoning modules**: additional modules added by developers (such as reasoning components and on-board passenger service applications) and greenlighted by an administrator can be hosted on each of the system devices.
3. **Provide information**: each system device must support a mechanism allowing clients to retrieve information. Clients can be passenger applications, but can also be other system devices that require information from their peers.

For information consumers we define the following use cases:

1. **Receive information**: an information consumer must be able to receive/retrieve the required information in a uniform way. The application or services are dependent on a connected system device to support this use case, and LimeDS must support developers in interfacing with the device API.
2. **Generate data**: an information consumer can also produce data that can be relevant to the operation of the framework, such as user preferences and user feedback. This data can enter the framework through any of the available system devices. Note that the inner workings and logic performed by an information consumer application or service is out of the scope of the framework.

**Framework Requirements**

Based on the use cases identified in the previous section we identified a number of requirements, which can be split into two categories: functional requirements and non-functional requirements. The latter are also commonly referred to as the so called quality attributes of the system.

**Generic Data Input Module**

The framework will interface with a wide range of data sources, some of these will be periodically queried for new information by the framework itself, while other sources will want to actively push new information to the framework using its API. A generic data input module is thus required that offers an elegant solution to both handle taking input from any potential data source as well as converting this data into an internal format the framework can easily work with. This is commonly referred to as the “Data Integration” challenge. Since we want to expose rich information to the passengers, it is important to be able to quickly and easily integrate different kinds of data sources that are available. Easy integration of web-based data sources via the framework is equally important (especially given that open data initiatives are more and more gaining in momentum worldwide). Not all data is available as an accessible structured service however, more than often the data source is a regular file or webpage. Therefore, a flexible interface integration mechanism is to be provided and considered to be a key aspect of the framework.

These requirements are very important in our design, since devices and applications that use the LimeDS framework should be able to present accurate and clear data to the passengers at all times. In general, the richer the data we want to show, the more (heterogeneous) data sources will have to be used. For instance: a geographical map provider, location-based advertising, location-based weather information, upcoming public transportation connections, etc.
Modular Reasoning Engine

A large number of datasets will be available to the framework at any time, so some kind of reasoning is required to extract valuable information from this data. However, as reasoning will be performed for topics that can vary in scope, the decision was made that multiple modules, each performing a certain aspect of reasoning, will be needed. Some of these reasoning modules will make decisions in near real-time based on up-to-date data that is being streamed to the framework, while other modules will periodically generate new information based on historical data. This modular approach also allows us to easily add new functionality when the need arises. A modular reasoning engine thus enables us to partition the reasoning aspect of the framework and will coordinate routing the data between the various modules. Finally, the engine must support running third party reasoning components without putting the framework’s core operations at risk, e.g. ensuring that the failure of one component can be contained so that other applications are not influenced by this failure, or can fall-back on a degraded mode of operation.

Information Service

Data is made available to the system and several reasoning modules will be in place to extract valuable information from this data. After all, data are the individual facts where information can be derived from. As such, for data to become information context should be included as well. It is essential that the framework can expose this generated information to interested parties (e.g. passenger applications) in different ways:

1. Querying model for historic information and on a need to know basis, i.e. just in time. In this case interested services or passenger applications will look for the required information themselves by actively querying our framework API.
2. Notification/pub-sub mechanism for near real-time information. A mechanism is required that allows for services or passenger applications to express the topics they are interested in to the framework. This allows the framework to notify the necessary services or applications whenever new information becomes available that matches their field-of-interest.

Architectural Design

Having defined the scope and general technical requirements for the framework, the architecture of the system can be elaborated.

Framework Layers
The high-level architecture of the LimeDS system is shown in Figure 9. First, an abstraction layer is built on top of the operating system of the device on which the framework will be running. The goal of this module is to hide all the operating system specific details from the other framework modules, allowing these to run unmodified on all devices for which the abstraction layer can be implemented. As the framework is implemented in the Java programming language, system-independent development is achieved by using the Java Virtual Machine as abstraction layer.

![LimeDS architecture diagram](image)

*Figure 9: The high-level architecture of the LimeDS ecosystem*

On top of the abstraction layer the core platform is built, consisting of a number of utility components that facilitate the development of the actual LimeDS modules, namely the input adapters and functional components. Input adapters are needed that can convert data from multiple sources into a format that is compatible with the internal representation used by the framework. A distinction can be made between data sources that will autonomously push new data into the framework (usually real-time data, e.g. GPS coordinates posted by a moving train) and data sources that will be queried by the framework itself (e.g. a Web Service that can be used to retrieve route information). The functional components are features responsible for implementing the actual logic that will result in the end-user services / applications targeted at providing a better travelling experience. Each such feature can be represented by a functional component that uses various input adapters, core platform components or even other functional components in order to transform unstructured, loose bits of data into useful information for the passenger. The functional components can also deploy endpoints to a REST API using the service layer, allowing applications (on mobile devices, public screens, etc.) to communicate with the framework.

**Detailed Architecture Overview**

From the requirements discussed in the previous sections, it is clear that the framework architecture should support a wide range of use cases. The framework can only succeed in such a context when its design incorporates the aspects described in the subsections below.

*Data Representation*
The data representation used by the framework must be highly flexible in order to support a wide range of use cases, while retaining good developer usability and interoperability with existing solutions that produce data. The corresponding data model should not be fixed to a predefined schema, as this limits the capability to be extended in an intuitive manner. In its role as native object model for the dynamic scripting language JavaScript, JSON (Crockford, 2006) matches the requirement for a highly flexible representation. Additionally, thanks to its wide adoption in web technologies, JSON has good compatibility with all major programming languages. To model the JSON format internally LimeDS uses the Jackson JSON processor library (FasterXML, 2009).

**Dynamic Module System**

As the framework has been designed for modifiability, it should support changes to be made at runtime, in order to avoid any unnecessary downtime. For this reason, a dynamic module system has been introduced that can be used in combination with the service layer to implement a highly adaptable system. The dynamic module system for LimeDS has been based on the existing standard specification OSGi. This modular system design has other advantages as well. All of these modules conform to a certain contract (an interface). This means that new modules can be added in or swapped in, as long as they just adhere to the specifications of the contract. This leads to easy extensibility of the framework. This also leads to re-usability, since after a while an elaborate collection of modules becomes available. Since communication between modules is restricted to the contracts, these contracts are actually an explicit description of the dependencies of a module. This allows the framework to define what should happen when a dependency is not available (for each module). Thanks to this, instead of crashing, the system will allow a module to execute an alternative scenario whenever the dependencies change at runtime.

**Service System**

The OSGi specification consists of a service model that provides a very good match with the envisioned LimeDS service layer. The service layer is one of the most important systems of the framework as it allows modules to communicate while retaining loose coupling, enabling functionality to be extended or replaced at runtime. It also provides the foundation for the dataflow system that is explained in the next section.

**Dataflow System**

The dataflow system is an abstraction on top of the service layer that facilitates developers in implementing data oriented processes, which are at the core of the various use cases that must be supported. Figure 10 shows the three interfaces that define the components from which entire dataflow processes can be built:

1. A producer specifies the generic interface for components that can make data available. It represents a pull-mechanism of communication in which the producer is queried whenever the data it provides is required in some part of the data flow.
2. A consumer specifies the generic interface for components that can consume data. It represents a push-mechanism of communication in which the consumer can be actively called whenever up-to-date content is available.
3. Connectors are the links between producers and consumers in the data flow model. Each connector defines which producer and consumer instances will participate in the connection and provides the logic that allows the data to be captured from the producer-side and transformed into appropriate output that can be delivered to the target instances on the consumer-side.
Components of the connector-type can be activated periodically to perform certain tasks, e.g., a connector that enriches all stored places of interest with up-to-date weather information can be activated hourly. To support these connectors, the framework must integrate a scheduler. We use the Quartz library (Cavaness, 2006) to actually trigger the recurring tasks while a Connector Scheduler is responsible for the synchronization of schedulable connectors available from the service registry with the triggers that are enabled within the Quartz scheduler.

Once a dataflow component is defined and all its properties are set, the framework needs to take this definition and translate it to an active instance of the component that can participate in the various dataflow chains. This process is called the materialization of the component, i.e., the necessary components need to be deployed in the framework and corresponding interface wiring needs to be properly activated.

To illustrate the above theoretical presentation, an example is detailed below. The example concerns the tracking of a journey by a user and providing personalized travel information to that user.
The entry-point for this flow is a function `startJourney()` that will initialise a new travel route. We use a MongoDB database to persist the journey data. The MongoDB instance is integrated with LimeDS, by providing three functions: `store()`, `delete()` and `query()`. We connect the `startJourney()` block to the store function to store the incoming requested travel route.

To be able to generate updates for the chosen journey, we introduce a trigger function that will be scheduled to execute periodically. Each time this function is executed, we use the database query function to retrieve the relevant route information. This data is then sent to an `updateJourney()` function that will check for route updates. This can be done by using three data sources:

- An RSS news feed that lists traffic updates.
- A Web Service that provides real-time information concerning train departures.
- A CSV file that contains parking information for which a new dump is generated every 15 minutes.

To integrate these three data sources in the dataflow, a function is made for each source that can connect to the original data source and transform the data into a compatible format. Once all the data is processed, the `updateJourney()` function can send an update to the user (if necessary) using the `sendNotification()` function, which uses the Google Cloud Messaging service to send the update to the phone of the user.

The dataflow can be completed by adding a `stopJourney()` function that enables the user’s travel assistant app to stop or cancel the journey. To do this, we rely on the database `delete()` function. LimeDS allows any function to be exposed over HTTP as a Web Service. We do this for the `startJourney()` and `stopJourney()` functions, so these operations can be called from the personal travel assistant mobile application. In addition we also expose the `getParkingInfo()` function.

Features which are inherently provided by the framework are data validation for any function (checking whether the data is consistent with a specific model or within specific constraints), e.g. it allows validating the journey information that is sent by the mobile client before initializing the update process.

The framework also allows to add caching to any function. In the dataflow example above a remote call is made each time the traffic RSS feed and real-time train info Web Services are called. This can take a while to complete, and the number of allowed requests for these external sources could be limited. However, by adding a cache, we can circumvent these problems without writing any additional lines of code. Now the results will be cached locally and the amount of remote calls will be greatly reduced (based on tuneable cache parameters).

**Storage System**

The framework storage system fully embraces the data representation as discussed in the SubSection ‘Data Representation’ by using subtypes of JsonNode as the main argument and return types. For the Proof-of-Concept, to be detailed later in the chapter, we chose to implement the storage service using MongoDB (Chodorow, 2013) as this technology uses a binary variant of JSON, both for storing data and as a query format for retrieving data. However, this implementation can easily be replaced by other NoSQL type of databases as the JSON representation can be mapped to any hierarchical key-value store.

**Communication System**

The LimeDS framework supports the standard specification JAX-RS (Hadley & Sandoz, 2009), allowing automated generation of REST Web Services based on annotated tools that are available in the framework. The principle is very similar to how data flows can be specified: JAX-RS annotated services are added to the service registry. The annotations contain information about the REST service path and the available HTTP methods. A JAX-RS Bridge module connects with the service registry and is continuously scanning for new annotated services. The JAX-RS Bridge then calls Apache Wink (Apache Software Foundation, 2015) for each discovered service. Wink is a library that translates the annotated resources to HTTP servlets, which can be hosted in a servlet container.
To facilitate calling the services provided in this way (or external REST services), a client utility library is provided by the framework that can be used to easily call REST services using a minimal amount of code. The client implementation is registered as a service with the service registry, allowing it to be accessed from any module. The client API allows expressing REST calls in a rather natural way, e.g. in order to post a JSON object to an example service, one can write the following statement:

```java
JsonNode jsonObject =
    Json.fromJson("{
      "example": "Hello world!"
    }");
client.target("http://example.com/API/example")
    .post(jsonObject).returnNoResult();
```

With both a way to easily expose REST services (server side) and a way to fluently call and make use of these services, the framework provides all the features that are required of a communication system.

**Load Manager System**

The framework further provides a load manager system which facilitates dataflow component developers in adding robustness and scalability to their services, allowing use cases to be implemented without introducing a lot of development overhead. For each registered service, the load manager will generate a proxy service that transparently captures all requests sent to the original service and redirects it to the service instance (running locally or remote) that is currently best suited to answer the request. This decision can be based on the number of requests for each instance, the mean response times, etc.

![Figure 12: LimeDS functions are designed to be robust](image)

The framework is able to cope with situations that are a result of unreliability by providing a system that makes it possible to monitor the availability of external systems. Under normal circumstances, a train that loses its network connectivity would not be able to refresh its data. When employing LimeDS however, we see that train-deployed applications will autonomously query the availability of the external system / service / data source on the wayside. If this system is periodically inaccessible, LimeDS will notice this unavailability and act by transparently redirecting data calls to local storage / cached data until the external services are reachable again, at which point the original dataflow is restored.

**Scalable Services**
The final aspect to be presented is the scalability of the system. Because of the highly dynamic transportation context and the potentially high (and fluctuating) number of clients, the framework must be able to scale based on the number of users and connected devices. Two cases in particular are of importance. In the first case, imagine a large amount of requests happening at about the same time. All these requests will start remote connections to external data sources via the framework. This will obviously generate a lot of remote requests, which may get blocked by the external services in question, because their allowed request quota was reached. This first case can be solved by using a caching mechanism. The framework allows to easily utilise caches that will store the results of certain requests within the configured timeframe. This will not only greatly reduce the load on the external services and thus attempt to stay below the allowed limits. It will also greatly reduce response times, since no more remote requests are needed if there is a cached answer available in the LimeDS framework.

Secondly, imagine if all requests would be tunnelled through a single LimeDS instance. This would put a lot of load on this server. The framework is however inherently able to balance the load between multiple instances that are setup in a cluster, each employing their own caches. This approach is based on the mechanism proposed in (Burns, et al, 2016).

![Figure 13: Scalability as a native, inherent built-in feature](image)

**SOLUTIONS AND RECOMMENDATIONS**

Two Proof-of-Concept demonstrator implementations, using the LimeDS framework as described in the previous section, have been successfully deployed. We succeeded in building a robust and modifiable framework, facilitating the addition of new functionality at any time (i.e. even at runtime when the system is in operation) and allowing various layers of robustness to be added to the system through the introduction of the concept of service dynamics. This implies that components are no longer statically bound, but rather rely on independent services executing the defined behaviour in case of service failure (e.g. in the event of system or network failures.

**On-board Passenger Information Display**
Figure 14 shows a screenshot of an on-board display application powered by the LimeDS framework. The application depicts context-aware travel connections for passengers on board the train, i.e. relevant information about multi-modal connecting services at the next calling point of the train. In order to obtain this information, a dataflow was implemented on the LimeDS framework. A conceptual overview of this dataflow is drawn in Figure 15.

The dataflow is triggered when the client application (i.e., the on-board display application) sends a REST request to the framework URL of the `getConnections()` component. The next stop of the train and its Expected Time of Arrival (ETA) at the next stop are passed as arguments. This activates the input adapters developed to access real-time train (`getTrainDepartures()`) and bus (`getBusDepartures()`) information as well as information from a bike-sharing scheme (`getVilloInfo()`) through REST calls. Train and bus connections are next filtered on due time (`dueTime-connector`) and travel distance towards the platform of the connection in order to mark services that will already have left (by the time passengers from the requesting train arrive) as not feasible. Similarly, only operational bike-sharing stands in the vicinity of the next stop are passed through the availability connector. The profile filter further categorises the connections based on two profiles, namely, passengers with or without mobility impairment where the filter takes into account longer walking distances for the former. As such, the resulting departing services are marked as:
1. **Feasible**: when there is sufficient time to board the service taking into account the real-time arrival time of the requesting train and the travel distance towards the platform of this service.
2. **Tight**: when the service is feasible but the margin is less than 60 seconds.
3. **Not feasible**: when the service will no longer be available when the passenger arrives.

The on-board public display application extracts the resulting information from the REST response and properly displays it to train passengers as demonstrated in Figure 15. As such, train passengers are provided with timely, relevant information that is the result of combining multiple journey-related data sources.

**Personalised Travel Connections**

The second PoC concerns the personalisation of feasible connections at interchange stations in an off-board situation, enabled through a Semantic Web approach. The steps executed to create such an application using the LimeDS framework are presented in this section.

In general, we will use an ontology both to describe public transport related concepts as well as to support classification using a combination of OWL DL and SWRL based reasoning. The LimeDS framework can then be used to compose the data flows for specific applications, allowing to connect a variety of data producers (e.g. railway timetable information providers) with reasoning modules drawing conclusions based on the currently available information and context, and exposing these reasoning results as REST endpoints for visualisation by (mobile) client applications.

An important aspect of a satisfactory travel experience for all travellers is the fact that transportation information should be correct and accurate at all times, and preferably tuned for the specific situation or context of the person at hand. Let us clarify this with an example: a connecting public transportation option may not be feasible to catch for every type of person if only five minutes are scheduled between arriving at a station and boarding that scheduled connection leaving from a different platform, as there is a need to disembark, orient yourself and find/make your way to the other platform and finally board the connecting transport. In other words, the time a traveller requires to transfer between platforms can be dependent on the context (e.g. a first time visit versus daily use/knowledge of the station layout) and physical limitations of that person (e.g. people with physical disabilities or in wheelchairs, elderly, people using a child carrier or heavy travelling luggage). In order to come up with a true personalised travel guidance system, this contextual information needs to be captured and taken into account before presenting travelling guidance to the user. Another example of taking context into account when presenting personalised travel information is when delays are occurring: public transport connections that under normal circumstances are not considered because there is too little platform transfer time according to the regular (non-delayed) timetable, may become a valid transportation option in case the connection is delayed, and can therefore exceptionally be presented to the user as a viable alternative, allowing that person to potentially reach their destination quicker.

**Ontology Engineering**

In support of this scenario an ontology has been created extending two existing ontologies:

2. **Weather** (Gajderowicz, 2011): Based on the ontology created by Aaron Elkiss (Elkiss, 2011).

Starting from the **Transit** and **Weather** ontologies an extension has been modelled, representing the railway timetable from the Belgian National Railway Operator (NMBS/SNCB) at a number of Belgian railway stations. This represents the railway schedule as it should be if everything runs according to plan. The main new OWL class has been named **Connection**. Individuals of these **Connections** represent the actual running of that service on a specific date and are used at runtime to determine which of the **Connections** at any given railway station can be caught for the given traveller's profile. To perform the classification, a combination
of OWL DL as well as SWRL reasoning is adopted. A number of illustrative axioms are presented in the following paragraphs.
For a Connection to be classified as suitable for travellers with a mobility impairment, the OWL definition is given below, based on the terms available in the Transit vocabulary. It specifies that the Connection should be on a route which has a pre-arrangement for mobility impaired passenger, this arrangement should have been confirmed by the operator and the Connection should depart from a platform at ground level:

\[
\text{MobilityImpairmentSuitableConnection} \\
\iff \\
\text{(route some owl:Thing)} \\
\quad \land \quad \text{(hasAccessArrangement value prearranged)} \\
\quad \land \quad \text{(platform some xsd:int[> "-1"^^xsd:int])} \\
\quad \land \quad \text{(isArrangementOK value true)}
\]

It should be clear that for other situations or other transport modes, such as a bus, taxi or cycle hire, other definitions can be specified in the domain ontology, used for that specific operator deployment. However, thanks to the generic notion of a MobilityImpairmentSuitableConnection, the UI visualising this information does not need to be aware of the specific definitions in place for that specific situation. The reasoner performs the job of filtering and classification. At first, the use of as much DL-based axioms as possible has been pursued, for the reason of genericness and to support the ability to exchange multiple OWL DL enabled reasoner implementations. However, sometimes it might not be possible to purely rely on OWL DL. To facilitate those more complex situations, support for SWRL rules has been included as well. An example of using SWRL to support complex rules and to take the real-time running information of the Connection into account, is given below:

\[
\text{MobilityImpairmentSuitableConnection(?connection),} \\
\text{currentTime(Now, ?ct),} \\
\text{departureTime(?connection, ?departuretime),} \\
\text{hasDelay(?connection, ?delay),} \\
\text{add(?earliestdeparturetime, ?ct, 900000),} \\
\text{add(?realdeparturetime, ?departuretime, ?delay),} \\
\text{greaterThan(?realdeparturetime, ?earliestdeparturetime)} \\
\rightarrow \\
\text{MobilityImpairmentTimedSuitableConnection(?connection)}
\]

This SWRL rule specifies that, for a Connection to be classified as a MobilityImpairmentTimedSuitableConnection, it should already have been classified as a MobilityImpairmentSuitableConnection and in addition its actual departure time should be at least 15 minutes in the future.
Other definitions have been included for other categories of passengers as well, such as HearingImpairmentSuitableConnection or VisualImpairmentSuitableConnection in case of Connections with specific requirements for an on-board Passenger Information System or taking into account the earlier introduced Weather ontology. Examples include a FrostSuitableConnection, SnowSuitableConnection or BicycleSuitableConnection.

**LimeDS Dataflow**

Once the ontology and its defining axioms have been agreed upon, the implementation within the LimeDS framework can start. For this, the integrated flow builder can be used. For the scenario presented in this section, the dataflow as modelled using the LimeDS Flow Builder is illustrated in Figure 16.
Figure 16: Data Flow to classify the personalised connections

The main Flow Function is illustrated in the middle of the figure. This represents the configuration of the reasoner. In addition, the configuration also allows to specify the endpoint which should be exposed and used by client applications to trigger this data flow, as well as non-functional properties for caching, load-balancing and authentication/authorisation characteristics, and this in a user friendly manner without the need to write any boilerplate code.

Finally, on the left hand side in Figure 16, several input components (data producers) can be seen. These ensure that the correct information is (i) fetched from the online data sources (e.g. for retrieving the at-runtime information of the railway operations, the weather at a certain location and optionally geocoding that location), and (ii) converted into JSON-LD ready for processing by the reasoner component, illustrated in the middle of the figure. On the right hand side, components are linked to the reasoner for persistency functionality. As indicated in earlier, built-in support for persistency by means of MongoDB is provided by the framework.
Figure 17: Based on the Semantic Model and at-runtime information, connections are classified in 10 categories

Evaluation

The LimeDS platform was evaluated during the TraPIST project for different scenarios on a low-resource server instance (2 cores, Intel Xenon CPU E5-2630 @ 2.3 GHz, 2GB RAM). In a first scenario we subjected the host machine to an increasing number of requests per second. This was done for four different server implementations: a pure Jetty (8.1.14) servlet-based implementation (The Eclipse Foundation, 2011), a LimeDS (limeDS.0.1.0-pre-alpha) Java implementation, a LimeDS (limeDS.0.1.0-pre-alpha) Javascript implementation and a Node-RED (0.13.3) implementation (JS Foundation, 2015). It should be noted that the purpose of this test was to find a scaling breaking point (in part the reason why modest hardware resources were used). Also note that the Servlet (Jetty, version 8.1.14) measurement can be considered a performance baseline, but offers none of the high-level features of LimeDS. The test was executed until the server crashed or timeouts started to occur on the client. The Jetty implementation was able to maintain good response times before crashing at > 260 RPS. The LimeDS Java implementation can almost match the servlet implementation up until 200 RPS, before failing at > 220 RPS. The JavaScript variant performs slightly worse and fails earlier at > 200 RPS. Node-RED has, out of the four tested technologies, the worst overall response-times and stops working at > 180 RPS. The results are graphically presented in Figure 18.
In a second scenario, illustrated in Figure 19, we tested the data processing capabilities of LimeDS, opposed to an equivalent Jetty servlet implementation by sending data that contained extensive profile information of users. We see similar response times and a linear increase in processing time for all implementations. Large requests up to 30 MB can be processed under 2 seconds. 30 MB requests are actually huge, given the nature of typical Web Service requests. Note that a default configured Node-RED setup is incapable of processing requests this large and responds with an error message when passing an array of 100 user records.

The third scenario evaluates the impact of the dataflow size/complexity on the response time of the functions calls (implemented in JavaScript). The highest curve in Figure 20 shows the average response time when the dataflow is called for the first time, while the lowest curve shows the average response time for all subsequent calls. This warm-up phenomenon that can be witnessed is to do with the way our JavaScript engine is initialised.
Related to the previous scenario, we also tested the deployment and undeployment time required for increasingly larger/more complex dataflows. As expected, the deployment time increases for larger dataflows, but remains linear. Undeployment remains practically constant.

FUTURE RESEARCH DIRECTIONS

The presented framework rapidly grew from an OSGi abstraction layer and library in Java for internal use, to a full-featured development platform with a visual (JavaScript-based) editor. After a number of successful collaborative research projects with industrial partners, it was decided to share the platform with the open-source community. Future research will continue to expand the LimeDS feature set and improve upon the runtime engine to evolve the platform into a general purpose, reliable and easy-to-use service development framework. Ideally, it will allow developers to setup a cluster of LimeDS instances in the Cloud and immediately start designing logical data flows on top of these instances, by connecting readily
available modules. Aspects such as deployment and configuration are taken care of by the framework, while the uploaded flows are executed by our runtime engine. Future research directions include:

1. **Profiling support**: with the current release, it can be difficult to detect certain kinds of bugs. A LimeDS Aspect can easily be introduced that maintains statistics for each Segment and reports back issues to a central monitoring system. The real challenge is to integrate this information with the visual editor in order to allow quick troubleshooting.

2. **Enhanced processing engine**: although LimeDS is built with scalability in mind from the start, there is always room for improvement. LimeDS’s current thread per request model is eventually limited in terms of scalability by the maximum amount of allowed threads. We are evaluating various asynchronous programming tools to improve the framework even further in this aspect. The benefit of LimeDS is that we can update the processing engine transparently, without needing to change modules written for LimeDS.

3. **Support for popular technologies out-of-the-box**: the success of a framework like LimeDS relies on the presence of a rich eco-system. It is made relatively easy to create and distribute shareable modules and a number of adapters and connectors (that can be used straight from the visual editor) for popular technologies is made available, allowing users to dispose of a set building blocks for their applications. Future connectors include: Twitter & Slack integration, modules to interact with the underlying OS, Big Data and storage technologies, etc.

**CONCLUSION**

In this chapter, we presented a multi-source, data processing software framework suitable for the railway transport sector, allowing for agile development of data processing applications which rely on a multitude of heterogeneous open data and knowledge sources. A real-time data consolidation mechanism has been implemented on top of the service layer of the OSGi-based platform. This abstraction allows to provide robustness, easy configurability and extensibility. The LimeDS framework embraces HTTP(S) and other proven web technologies for transferring data between nodes running on different host machines as well as client applications that make use of the enriched information offered by these nodes. Two Proof-of-Concept demonstrator using the framework were developed and integrated with a passenger information system that visualises real-time information on on-board public train and station displays. These demonstrators have a focus on offering personalised and context-filtered information to the end-user, allowing that end-user to be presented with information tailored to his or her needs and (physical) disabilities. LimeDS is able to deal with sudden unavailability of select data sources and offers a developer-friendly way of reasoning over this data. Furthermore, it can (visually) aid with the construction of data processing/reasoning workflows and provides inherent and configurable support for scaling, resilience and fall-back scenarios.

**REFERENCES**


We thank the reviewer for his/her comments and appreciate that is was considered interesting and an important contribution to the book. Please find below how we have revised the manuscript to try to resolve the issues that were raised.

1. Literature review needs to be more comprehensive

We have updated the bibliographic references with a number of more up-to-date citations and elaborated the presentation where appropriate.

INTRODUCTION

Today more than ever, public transportation operators are aware of the importance of investing in their passengers. Moreover, meeting passengers' public transportation needs, in addition to important environmental aspects (Hua, 2016, Pålsson & Kovács, 2014, Guerra et al, 2016) is acknowledged as a central goal in the European Commission’s transport strategy roadmap. (European Commission, 2011) In this roadmap, it is shown that acting on issues important to passengers, such as reducing noise in a train or providing wireless connectivity, enriches the customer experience. At the same time, due to the current information-centric nature of society, passengers expect public transportation to be more and more augmented/personalised with information from various sources (e.g. social networks, multimodal travel information) (Sierpiński, 2017, van Lier et al, 2014). Recent ICT developments present opportunities to meet passengers' rising expectations. As a result, the amount of available mobile travel applications offering travel information to passengers has grown exponentially (Gardner, Haeusler & Tomitsch, 2010). Contemporary mobile applications, such as the travel information apps provided by European railway companies e.g. NMBS, NS, Deutsche Bahn, National Rail and SNCF, mostly offer Real Time Train Information (RTTI) about arrival and departure times as well as mobile ticketing services. Many mobile RTTI applications are mostly context specific, single-purpose applications that provide a solution to a particular problem or requirement. However, according to ORR, the (British) Office of Rail and Road, passengers want to receive live information and they want it at their fingertips (Office of Rail Regulation, 2012). Since the initial publication of this report, where train operators are required to provide appropriate, accurate and timely information to enable (prospective) passengers to plan and make their journeys with a reasonable degree of assurance, including in times of disruption, the ORR has elaborated on this key requirement. The ORR requires from a train operating company to publish a code on practice setting out how it will ensure compliance with this directive. In 2014, further studies and surveys were conducted to see whether passengers had noticed tangible improvements. The conclusions highlighted some improvements, but also raised a number of areas where special attention was needed. As a result of this, the ORR published a new regulatory guidance, in collaboration with the industry (Rail Delivery Group, 2016) which issued 50 recommendations, in 2016 (Office of Rail and Road, 2016).

Additionally, the section “Related Work on a Number of Relevant Semantic Web Technologies”, has been reworked to be more comprehensive.

Related Work on a Number of Relevant Semantic Web Technologies

A short, but comprehensive definition of an ontology, based on the definition by Gruber in (Gruber, 1993) is: “An ontology is a formal specification of an agreed conceptualisation of a domain in the context of knowledge description.” Accordingly, an ontology describes in a formal manner the concepts and
relationships, existing in a particular domain or system and using a machine-processable common vocabulary within a computerised system. It can also contain classification rules. This standardised representation of the semantics of a domain can then be used to exchange data and its attached domain model. This way an ontology encourages re-use, communication, collaboration and integration (Fernández-López, Gómez-Pérez & Juristo, 1997). By managing the data about the current context in an ontology, intelligent algorithms can be more easily defined that take advantage of this information to optimize and personalize applications.

OWL has different levels of expressive power, each of them varying in their trade-off between expressiveness and inferential complexity. They are, in order of increasing expressiveness: (i) OWL Lite: supports classification hierarchies and simple constraint features, (ii) OWL DL: OWL Description Logics, a subset providing great expressiveness without losing computational completeness and decidability and (iii) OWL Full: supports maximum expressiveness and syntactic freedom, however without computational guarantees. Using one of the three sub-language flavours of OWL, one can easily adapt to the required expressiveness. Arguably, the most interesting sublanguage for many application domains is OWL DL, balancing great expressiveness with inferential efficiency. Due to its foundation in description logics, OWL DL is also very flexible and computationally complete. The constraints imposed by the adoption of the DL formalism assure that a decidable reasoning implementation can be achieved. OWL can be seen as the evolution of several previous WorldWide Web Consortium (W3C) recommendations, being XML, XML Schema, Resource Description Framework (RDF) and RDF Schema. In this view, each step introduces more functionality, i.e. XML introduces a common syntax, XML Schema introduces data types and structure, RDF focuses on meta-data, allowing to say anything about anything, RDF Schema introduces RDF resource types and finally OWL specifically supports the construction of vocabularies and shared meanings.

Ontologies are considered as dynamic and evolving in time and are also tailored towards the distributed nature of the Web. Some of the serialisation formats of OWL are based on eXtensible Markup Language (XML), namely RDF/XML (Beckett and McBride, 2004) or OWL/XML (Hori et. al, 2004), others being Turtle (Beckett et. al, 2008) or Manchester Syntax (Horridge et. al, 2006). Serialisation is the process of converting an in-memory object or data structure into a format that can be persisted on disk or transmitted by some communication medium. Each of those potential serialisations serve different purposes. An elaborate discussion can be found in (Hitzler et. al, 2012).

In recent years, research and standardisation efforts have taken OWL into a next level, namely OWL 2. In contrast to the earlier decomposition of the description logics in OWL Light, OWL DL and OWL Full, OWL 2 (World Wide Web Consortium, 2012) specifies three new types of sublanguages, called profiles, which have favourable computational properties and are easier to implement, namely OWL 2 EL (Existential quantification Logic): useful in applications employing ontologies that contain very large numbers of properties and/or classes, OWL 2 QL (Query Language): aimed at applications using very large volumes of instance data, and where query answering is the most important reasoning task and OWL 2 RL (Rule Language): used for applications which require scalable reasoning without sacrificing too much expressive power. The profiles are designed for increased efficiency of reasoning for specific types of applications.

JSON-LD (World Wide Web Consortium, 2014) is a JSON-based format to serialise and message data captured in an ontology. The syntax is designed to easily integrate into deployed systems that already use JSON, and provides a smooth upgrade path from JSON to JSON-LD. OWL can be easily integrated with various rule platforms. This is demonstrated by a.o. the Semantic Web Rule Language (SWRL) W3C Submission (Horrocks, et al. 2004).

In description logic terminology, the T-Box contains the axioms defining the concepts and relations in an ontology, while the A-Box contains the assertions about the individuals in that domain (Carroll et. al, 2004). OWL, however, does not explicitly make this distinction because model and data can be mixed in the same description, but for clarification purposes it is still beneficial to make this distinction. An example to illustrative the difference between T-Box and A-Box concerns a simplified description of my Personal Computer (PC). The T-Box description of a Dell Latitude E5400 laptop (concept) states that this type of
The computer has one serial number and an Intel(R) Core(TM)2 DUO P8600 Central Processing Unit (CPU) (concept). The processor concept also has one serial number. A corresponding A-Box instantiation could that the Dell Latitude E5400 with serial number 53-373-312-19 has the specific Intel(R) Core(TM)2 DUO P8600, with serial number 6.1.7600.16385.

The main reason for creating an ontology of all the concepts within a domain, such as for example the previous computer example, is that logical connections and relationships can be described between the concepts in the domain. This allows for inference to be applied on the ontology. The concepts in the domain need to be described, as well as the relationships and constraints that define the concepts. Once the ontology is constructed, inference rules can be declared about the concepts and their properties within the ontology.

A semantic reasoner is a piece of generic software, which is able to infer logical consequences, i.e., new knowledge, out of the information captured in an ontology. A wide range of mature reasoners exist today, e.g., Pellet (Sirin, et. al., 2007), Hermit (Glimm, et al., 2014) and Fact++ (Tsarkov & Horrocks, 2006). SPARQL (Prud'hommeaux & Seaborne, 2008) can be used to query data captured by an ontology. The leading language for encoding ontologies is OWL (McGuinness & Van Harmelen, 2004).

Three generally accepted conceptual phases can be distinguished in an inference process. These are: (i) Consistency checking and satisfiability: This process analyses the ontology model to find any contradicting axioms. For satisfiability, this conforms to determining whether a concept can have individuals. In case a concept is unsatisfiable but does have individuals defined, the ontology is inconsistent; (ii) Classification: Depending on the definitions of all named concepts in the ontology, this process completes the already existing asserted concept hierarchy with the inferred one; and (iii) Realisation: Since it is possible for a given individual to belong to more than one concept, this phase calculates the set of concepts to which each of the individuals can belong, taking into account the complete inferred concept hierarchy.

2. Detailed discussion on challenges and issues faced while designing a framework, and its implementation is missing. The chapter needs some more inputs with a focus on rail, issues, challenges and how the proposed framework will solve these challenges.

We understand that the flow of the text did not clearly present the challenges and issues in the railway domain that were considered as the starting point for the development of the LimeDS platform. Therefore, we have re-arranged and re-titled some of the rationale sections. More precisely, we have built up the rationale in the text as follows:

The remainder of this chapter is structured as follows. The next section presents the background motivation for the research and development of the LimeDS framework, as well as introducing a number of relevant related work, both in the context of provisioning platforms as well as in the context of Semantic Web technologies. Subsequently, the challenges and issues motivating the design of the framework, exemplary railway related use cases and corresponding requirements are detailed, followed by an extensive presentation of the architectural design and internal details. Two Proof-of-Concept demonstrators are illustrated in the second-to-last part of this chapter. Finally, to conclude, a number of performance metrics were evaluated and are presented together with the envisaged future research directions.

We would like to thank the reviewer once again for the feedback and we hope that with these enhancements, this chapter can be considered for publication in the book.

On behalf of all contributing authors,
Sincerely yours,

Dr. Stijn Verstichel
Dr. Bruno Volckaert
Ghent University