A Conceptual Architecture for an Event-based Information Aggregation Engine in Smart Logistics

Anne Baumgrass[1], Cristina Cabanillas[2], and Claudio Di Ciccio[2]

Abstract: The field of Smart Logistics is attracting interest in several areas of research, including Business Process Management. A wide range of research works are carried out to enhance the capability of monitoring the execution of ongoing logistics processes and predict their likely evolvement. In order to do this, it is crucial to have in place an IT infrastructure that provides the capability of automatically intercepting the digitalised transportation-related events stemming from widespread sources, along with their elaboration, interpretation and dispatching. In this context, we present here the service-oriented software architecture of such an event-based information engine. In particular, we describe the requisites that it must meet. Thereafter, we present the interfaces and subsequently the service-oriented components that are in charge of realising them. The outlined architecture is being utilised as the reference model for an ongoing European research project on Smart Logistics, namely GET Service.

Keywords: Smart Logistics; Service-oriented Architectures; Complex Event Processing

1 Introduction

GET Service[3] is a European FP7 research project aiming at the realisation of a distributed service-oriented platform for the planning, execution and monitoring of smart transportation processes. The devised platform is meant to be adopted by Logistics Service Providers (LSPs) Europe-wide, in order to take advantage of a powerful infrastructure that allows the improvement of their core business processes, in terms of reduced CO_2 emissions, better time scheduling, more precise service time estimates and thus, reduced costs. Against this goal, we notice that such a platform must build upon the regular synchronisation of its real-world context-awareness. For instance, it is vital that the position of involved transportation means is kept under control during the shipment of goods in order to assist its run-time monitoring. Such information can be gathered by the interception, analysis and interpretation of so-called *events*.

Events are known to be detected by different sensors and reported by several sources. Due to the dynamic nature of the context domain, such information is intrinsically meant to change over time. Therefore, the GET Service core module that is in charge of extracting relevant information on the current development of transportation processes, deals with concurrent event streams stemming from various originators. The information coming from the collection and comparison of the events in the flow of updates has to be

[1] Hasso-Plattner-Institut, University of Potsdam, Germany, `anne.baumgrass@hpi.de`
[2] Vienna University of Economics and Business, `name.[particle.]surname@wu.ac.at`
[3] `http://www.getservice-project.eu/`

interpreted to detect and possibly foresee the development of the transportation process, given its execution history and the context within which it is carried out. This paper aims at defining the architecture of the information aggregation and provisioning engine in the context of smart logistics; in particular, in the scope of the GET Service software infrastructure, which is under development at the time of writing and is henceforth referred to as *the platform*.

The remainder of this paper is structured as follows. Section 2 presents background on event processing. In particular, Section 2.1 introduces the fundamental concepts of event processing networks (EPN), processing agents (EPA), source, consumer, object, and channel. Section 2.2 explains how such concepts come into play in the context of event processing. Furthermore, it outlines how aggregation and correlation patterns contribute to the gathering of knowledge regarding the evolution of transportation processes, out of event streams. Then, Section 3 delves into the details of the functionalities that the information aggregation services must offer in the GET Service platform. The discussion is promoted to Section 4, where the architecture of the component offering those services is detailed, in conformance with the aforementioned criteria. Section 5 concludes this paper.

2 Background

This section summarises the background on event processing as well as the requirements that are necessary to design the event aggregation engine.

2.1 Event Processing Infrastructure

Events that are of importance in our context are the *transportation-related events*. They serve three main purposes: *(i)* tracing how a specific transportation process is executed, *(ii)* coordinating the different parties involved, and *(iii)* making appropriate decisions in relation to re-planning and rescheduling. Typically, events are produced and collected by different kind of systems spanning an *event processing network* (EPN) in which *event processing agents* (EPA) are linked by *event channels* to exchange events [EN10], (cf. Fig. 1). Each EPA may act as an *event consumer* to receive *event objects*, and as an *event source* in case it observes *events* and publishes them in a machine-readable form as *event objects*. In this way, an EPA reacts to its input by processing events and outputs events that can be fed to other EPAs over event channels [Lu01].

In the context of GET Service, the GET Service Platform should act as an event consumer to gather events from several event sources (e.g. driver's mobile devices and weather stations) and process them to generate transportation-related events, which might be provided to several consumers, e.g., Logistics Service Providers (LSPs) [Ba13b].

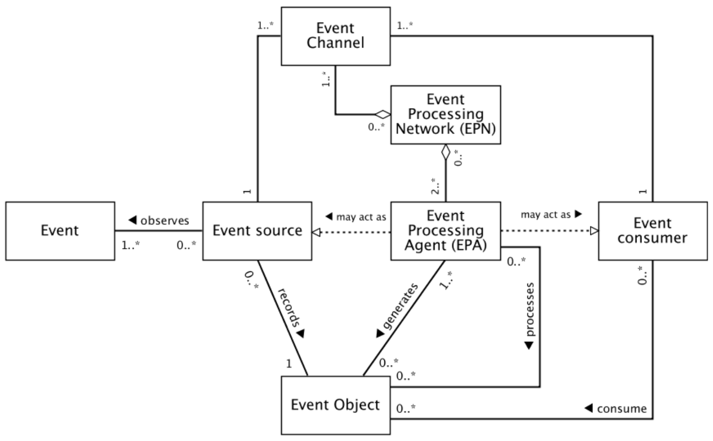

Fig. 1: Event processing infrastructure.

2.2 Event Processing Mechanism

In a smart logistics context like the one of GET Service, it is of utmost importance that all events related to the transportation of goods and corresponding transportation plans can be observed. Furthermore, such events have to be processed to derive transportation-related information, and be published to all interested consumers. To this extent, event processing is in charge of computing operations on events, including reading, creating, transforming, or discarding [EN10]. Specifically, an EPA carries out these operations.

First, an EPA contains an event adapter, which transforms events into event objects [Lu01], e.g., for importing weather forecasts in XML format. The EPA receives the events from an event channel as input in order to process them. Adapters are used to identify event objects published by several event sources, in possibly different formats. Data can be indeed encoded according to standards such as EDIFACT (United Nations/Electronic Data Interchange For Administration, Commerce and Transport [Be94]), MXML (Mining eXtensible Markup Language [vD05]), XES (eXtensible Event Stream, [GV14]), but also more general-purpose ones like CSV (Comma-Separated Values), Excel and XML (eXtensible Markup Language).

Second, events are related to each other according to event relationships. Typically, events are related by time, causality, aggregation [Lu01] or correlation [ROS11]. Event patterns are used to specify these relationships and identify them in an event stream considered by an event processing system. For example, only if `Container mounted` happened before `Goods loaded` in a certain time window and both event objects refer to the same container, they are related by a correlation relationship and can be aggregated to an event `Goods ready for transportation` (cf. Tab. 1). In this way, a correlation is specified through defining correlation attributes (e.g., time and container in the example) between event object types. Furthermore, event aggregation patterns can be used for recognising or

	Goods loaded	Container mounted	Goods ready for transportation
Description	New goods are loaded in Container1.	Container1 is mounted onto Truck1.	Truck1 is ready to start its transportation.
Occurrence time	January 2nd 2014 07:30	January 1st 2014 16:30	January 2nd 2014 07:30
Occurrence location	Harbour of Rotterdam	Harbour of Rotterdam	Port of Rotterdam
Originator(s)	Container1	Truck1, Container1	GET Service Platform
Impact	Truck may start the transportation.	Goods can be loaded in the container.	Truck may start driving and transport the loaded goods to their destination.
Target(s)	Container1	Truck1	TransportationOrder1

Tab. 1: Example events in logistics. For the sake of simplicity, the example is kept simple and abstracted from the real-world. For example, in order to have a truck ready for transportation other events might be relevant as well (e.g., documentation on board).

detecting a significant group of events from among a set of events, and creating a single event that summarises their significance in its data.

Third, event patterns are also used to forward events to interested consumers. For this purpose, an event consumer subscribes to an event processing system with a defined event pattern. Events that match that pattern are sent as notifications over event channels to the consumer. A notification contains data describing an event and may additionally carry information describing the circumstances of the event. In [MFP06], the event processing infrastructure only represents the theoretical components and disregards the issues to be dealt with when implementing this infrastructure in practice, e.g., access control or messaging formats.

We aim to use the transportation context to identify events from several event sources, process them into transportation-related events, and forward them to interested parties.

3 Design of the Information Aggregation Engine

The Information Aggregation Engine is a main component of the platform, which is responsible for collecting events from different sources and processing them in order to offer a unified interface to clients, planners, information providers, and other stakeholders. Thereby, the engine supports, among others, the use cases of track&trace, vessel arrivals and capacity visualisation. The specific functionalities that this service requires in transportation are described in the following sections. Such functionalities derived from a preliminary requirements elicitation phase and thorough analysis of the typical use cases scenario in the context of the GET Service project [Tr13].

3.1 Import and Export of Event Data

It is crucial that the interfaces of the platform adhere to existing messaging standards and interchange formats of all services that are used by the involved stakeholders. For this purpose, the messaging standards of logistics were investigated in [Ve13]. Four common message types where identified: *(i)* EDIFACT, used by shipping lines, terminal operators, or customs; *(ii)* EDIFACT XML (UN/CEFACT working groups); *(iii)* Business Logistics XML, used by larger Logistics Service Providers (LSPs); and *(iv)* Excel uploads and downloads, used by smaller Transportation Service Providers (TSPs). Furthermore, applications might use JSON as exchange format, which has less markup overhead in comparison to XML. To unify the communication in logistics for all stakeholders, the e-Freight project[4] developed a standard framework that also needs to be considered in the platform. It is a standard for freight information exchange covering all transport modes and stakeholders. Each of the above mentioned message types can be transported over different channels using different protocols and services, for example through SOAP web services, HTTP protocols, RPC, or FTP file transfers.

Thus, to extract events from all exchanged messages and to publish transportation-related events in the aggregation engine, it requires four generic interfaces for communication:

1. An interface to import messages of events from different sources (e.g. from client devices of LSPs) provided in different formats. Based on the aforementioned message formats, the engine must be able to call external web services, connect to message queuing services, generate HTTP requests, and download files from FTP. Additionally, it has to offer an interface, to which clients can push events contained in messages.

2. An interface for identifying the event information in these kinds of messages. By implementing adapters the aggregation engine defines where and how to extract events from all the imported messages types. Thus, it must be possible to import events using EDIFACT, XML, Excel, JSON, and the e-Freight format.

3. An interface for submitting event patterns to be notified of the occurrence(s) of events that the stakeholders are interested in. For this purpose, the aggregation engine must enable all stakeholders to specify these event patterns in a well-chosen language, such as Esper[5]. Furthermore, the aggregation engine must be extendable to implement the functionality of deriving event patterns from transportation plans, logistic process models, and route descriptions.

4. An interface to forward events to interested targets. Thus, the aggregation engine must itself provide functionalities to publish events and provide them to the stakeholders involved in transportation. Community systems or other platforms might act as intermediate event distributors. Thus, the engine needs to implement a message queuing service to distribute events and also forward notifications containing information on a subset of events. This forwarding may be implemented as HTTP responses or as API, but may also be realised through emails to be shown on the mobile client devices. The format for the notifications depends on the client devices but should at least adhere to the message standards mentioned above, including EDIFACT, XML, Excel, JSON, and the e-Freight format.

[4] `http://www.efreightproject.eu/`
[5] `http://esper.codehaus.org/`

Each of these interfaces must provide capabilities to access and modify the functionalities in order to adapt to a changing environment. Therefore, the interfaces should provide methods to support the standard operations of Create, Read, Update, and Delete (CRUD). For example, partners interacting with the platform should be able to adapt their own aggregation rules to changing plans, or set up new event sources, but also be able to delete rules that are no longer required.

3.2 Normalisation of Events

Because events are collected from different sources that can have different formats, events need to be normalised into a common unified format for further event processing. The normalisation needs to take place in the aggregation engine in order to process events. The normalisation includes the definition of the format of the normalised events, as well as the stored event properties that are available for purposes ranging from information extraction to correlation of events based on values. The different formats and their differing structure imply that the target event format needs to be extensible and general enough to allow for incorporation of structured or unstructured information from all different sources.

The transformation into the unified format can be specified by corresponding adapters. An adapter refers to a component that formats heterogeneous event data into a suitable input format. For example, an event stream in XML format can be processed by an XML parser and events can be extracted based on conversion rules, which can include mappings for different formats of dates and timestamps to the internal format. The mapping rules should be extensible and reusable, such that the task of connecting new sources can be conveniently performed.

3.3 Integration of Event Processing

Once the events are made available to the aggregation engine in a normalised format, the actual event processing has to be performed in form of aggregation and correlation. Thus, the functional requirements for the event processing engine is to support the above mentioned relations between events, i.e. to detect relations based on time, causality, aggregation and correlation. These relations are stored as rules that allow to relate and to aggregate several events.

Furthermore, the aggregation engine is expected to capture a large amount of events and needs to be able to process them within a complex environment where many actors subscribe for their respective events. The actual Complex Event Processing (CEP) system that is used is therefore required to be *scalable*.

3.4 Predictive Functionalities in Cooperation with Discriminative Classifiers

The ability not only to monitor but also to interpret the context information can be seen as one of the main objectives of the event processing component. Indeed, streams of transportation-related events represent a temporal snapshot over the current development of transportation processes. As an example, the consecutive coordinates and altitude levels of an aeroplane trace its movements. The events may rise from different sources (e.g., weather conditions along the route, traffic information in the arrival airport, etc.) and can altogether concur in the creation of a dangerous situation for the regular advancement of the transportation. Therefore, it is of considerable relevance to distinguish the sequences of events that lead to a disruption from those that are safe. Evaluating queries over event streams is a basic approach to this extent. Such queries would weigh the combination of events over time in order to determine whether the current evolvement of facts is likely to end up in a risky situation, or not. However, it would be impractical to predefine all such queries a priori. This is due both to the quantity of possible concurrent causes to check, and to the unfeasibility of foreseeing any possible anomalous sequence of events. To this extent, classifiers from the field of Machine Learning [Mi97] can be of significant help. For instance, Support-Vector Machines (SVMs [CV95]) are supervised learning models for linear classifications, i.e., able to identify a hyperplane in the space of features that separates numeric representations of input objects in two different categories. The hyperplane is determined on the basis of a learning process made on labelled historical data. In the context of transportation-related events, e.g., labelled historical data can represent the reported trajectories of aeroplanes, divided into those that were known to have landed in time and in the expected airport, and those, which were known to have been delayed or diverted. Once trained on such data, the classifier (e.g., SVM) can analyse current flights and predict whether they show an anomalous behaviour, or not. The input as events can be provided by a CEP system, as long as the transportation process specifies the information to be extracted from events to this extent. The learning systems can be used indeed to correlate available data, in order to detect anomalies based on previous knowledge. The selection of independent and dependent variables for the decision functions is thought to be determined a priori, since they are strongly domain-related. For instance, the SVM can recognise a possible diversion of flights on the basis of features such as gained distance from the departure airport, velocity and altitude of the aeroplane. However, the input sources (e.g., flight monitoring services) as far as the information aggregation and features extraction (e.g., from positional data to distance, velocity and altitude) are meant to be predefined.

On the basis of the prediction made by the classifier, a new event raising an alert can be generated in case of anomaly detection. Therefore, it is required for the event stream to be restructured in a way that makes it readable from an external classifier, on one hand. On the other hand, the classification returned as a result has to be treated and transformed into a new event. It is worthwhile to recall here that a framework for controlling the safe execution of tasks and signalling possible misbehaviours at runtime has already been outlined in [Ca14b], and preliminary results are already applied in the context of flight diversion detections [Ca14a].

3.5 Correlation of Events to Processes

An additional function within the aggregation engine is the (semi-)automatic derivation of correlation rules on the basis of process data and transportation plans. The intention is to analyse process models, route descriptions, or transport execution plan as input that is analysed to derive correlation and aggregation rules. For this purpose, the components outside the aggregation engine need to provide these documents in a way they can be parsed and event patterns can be derived. In case of processes modelled with the Business Process Model and Notation (BPMN) 2.0[6] format [Du13], the approach of [Ba13a] can be implemented to correlate events to process instances and identify whether this instance of a process model was executed successfully.

3.6 Notification Mechanism

The aim of the platform is to offer services to many clients and hence, it needs to adhere to common notification paradigms. The publish/subscribe paradigm is very common in distributed systems [MFP06] and needs to be supported by the information aggregation engine. Using this paradigm allows clients and planners to subscribe to certain types of events or aggregated events. For example, a planner might subscribe to all events that are correlated to the respective transportation plans that the planner has created. Then, if events occur during execution, the planner is notified about their occurrence and can react accordingly.

Besides the publish/subscribe paradigm, regular access to events is required in the platform. That is, information providers need the option to add new events directly into the platform via the appropriate interface (push). And additionally, the option to query for recent events from the event history should be made available in that interface (pull).

3.7 Summary

The above sections point out that we aim to design appropriate filtering mechanisms at early stages, to reduce the burden on the correlation and prediction activities. Furthermore, the derivation of correlation rules based on processes range from very simplistic approaches (e.g., correlating by container id), to more sophisticated, control flow, location, and time-aware correlation mechanisms. To provide a brief overview of the requirements of the aggregation engine, a tabular representation is given in Tab. 2.

4 Architecture of the Information Aggregation Engine

This section presents the architecture of the information aggregation engine, in the light of the requirements previously explained. UML component diagrams are used to visualise the logical interconnection of its internal components.

[6] `http://www.bpmn.org/`

Requirement	Description
R1. Heterogeneous Sources	Connection to different kinds of event sources
R2. Heterogeneous Formats	Collect events from different message formats
R3. Normalisation of Events	Store events of different formats in same normalised format
R4. Event Storage	Store normalised events in a central database
R5. Event Processing	
R5.1 Event Aggregation	Provide functionality to aggregate events of finer granularity to single events
R5.2 Query Subscription	Register queries to be informed of events of interest
R5.3 Domain-Specific Query Subscription	Register queries to be informed of transportation-related events of interest
R5.4 Domain-Specific Event Correlation	Automatically correlate events to transportation processes
R6. Notification Mechanism	Notify subscribed clients of respective events
R7. Event Classifiers	Determine criticality of an event for transportation

Tab. 2: Summary of required capabilities of the event-based information aggregation engine.

4.1 Design of External Interfaces

The information aggregation engine offers four interfaces to be used either by external event sources (e.g., driver, weather stations) or event consumers (e.g., planner, driver). Furthermore, it implements an interface to access the information store to request static information. All five interfaces are required to provide the functionalities described in Section 3. These are shown in the component diagram in Fig. 2 and summarised as follows.

EventAdministrationInterface. The EventAdministrationInterface receives the structural description of an event type and offers further administrative tasks related to events. The communication through this interface has to be implemented in two ways. It should be either initiated by any event source sending the event type description of the events it publishes (push) or it can be configured inside the corresponding event source adapter (cf. EventSourceAdapter, Fig. 4). The implementation is meant to be realised by means of a web service to which the event source can push the event type description.

EventSourceAdapterInterface. Through the EventSourceAdapterInterface the aggregation engine is able to receive events (resp. implementing R1 in Tab. 2). Each event source is intended to be connected through a specific adapter. This adapter then offers an interface of its own that can be used by the event source. Each adapter has to internally use the EventImportInterface (cf. Fig. 4), i.e., the interface through which the AggregationService can take as input and process new events.

EventSubscriptionInterface The EventSubscriptionInterface is used to register subscriptions to the aggregation engine. These subscriptions can be arbitrarily complex, i.e., they may be composed of specific event processing queries. The subscriptions should be pushed to the aggregation engine. Therefore, the aggregation engine provides an implementation of a request-response pattern to register subscriptions in the platform. The events being imported via the EventImportInterface are forwarded to the event consumers by the aggregation engine based on registered subscriptions.

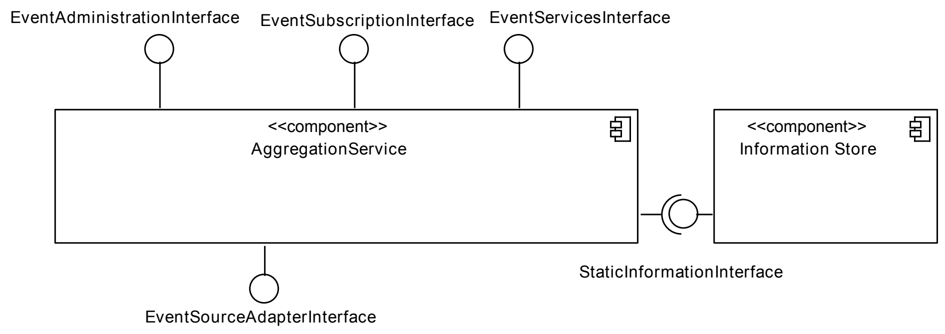
EventAdministrationInterface
EventSubscriptionInterface
EventServicesInterface
<<component>>
AggregationService
<<component>>
Information Store
StaticInformationInterface
EventSourceAdapterInterface

Fig. 2: Interfaces of the information aggregation engine.

EventServicesInterface. The EventServicesInterface combines all services of the aggregation engine that are visible to the external consumers. For example, a consumer may submit routes to the aggregation engine, which can be used in subscriptions later on.

StaticInformationInterface. The StaticInformationInterface offers access to information to enrich events. The aggregation engine uses it to receive all types of information, e.g., about transportation plans and schedules.

Tab. 3 summarises the interfaces with a short description, their inputs and outputs, the interaction pattern realised, and how errors should be handled.

4.2 Structure and Functionality of the Information Aggregation Engine

In this section the three components realising the aggregation engine are described in detail: they are EventHandler, EventProcessing, and EventServices. Fig. 3 shows the three components of the aggregation engine, derived from Fig. 2. The interfaces that they implement are also depicted, along with the interconnecting associations. In the middle, the EventProcessing component handles event transformations and querying. Thus, it includes the functionality of event processing and implements the requirements R5.1 and R5.2 and provides the functionality to implement R5.3 and R5.4 shown in Tab. 2.

4.2.1 The EventHandler

The EventHandler is meant to be implemented to collect, receive, and handle events from different kinds of systems in different formats. This means, it implements the requirements R1, R2, R3, and R4. For that purpose, it provides the EventAdministratorInterface and the EventSourceAdapterInterface. The internal structure of the EventHandler is represented by the following four components (see also Fig. 4).

Interface ID	**EventAdministrationInterface**
Description	Push event type definitions to the platform, necessary in order to import events of this type, and conduct administrative tasks on events.
Input	Structural description of an event type (e.g. as XSD) or task execution
Output	Confirmation
Interaction Patterns	Synchronous request/response
Error Handling	Synchronous confirmation

Interface ID	**EventSourceAdapterInterface**
Description	Events are pushed by event source, pulled from event sources or received by a subscription to event sources.
Input	Events including a reference to its event type (e.g., XML)
Output	None
Interaction Patterns	Synchronous push or pull, or publish/subscribe (always depends on the adapter)
Error Handling	No error handling

Interface ID	**SubscriptionInterface**
Description	Subscribe for events by queries (or other criteria)
Input	event processing query (e.g., String or EPL) or other event criteria
Output	ID of an event channel from which the events are pushed, events
Interaction Patterns	Synchronous request/response, publish/subscribe
Error Handling	Synchronous response or exception, retransmission on publish/subscribe communication

Interface ID	**EventServicesInterface**
Description	Additional services are offered in relation to events, e.g. process model monitoring or route handling.
Input	Process models (e.g., BPMN), transport orders (e.g., XML), or routes (e.g., JSON)
Output	ID of an event channel from which the events are pushed, events
Interaction Patterns	Synchronous request/response
Error Handling	Synchronous response or exception

Interface ID	**StaticInformationInterface**
Description	Request/response interface to access information, e.g., about route information and timetables.
Input	Database queries or function calls to databases
Output	Route, timetable, transportation plan
Interaction Patterns	Synchronous request/response
Error Handling	Synchronous response or exception

Tab. 3: Overview of the Interfaces of the information aggregation engine.

EventSourceAdapter Each kind of event sources requires an EventSourceAdapter, which is able to retrieve events from any kind of event source (over the EventSourceAdapterInterface). Event sources differ in the mechanism they use to provide events, e.g., downloads of event information from a FTP server or offering a web service to request events. Thus, all mechanisms to request events from event sources are considered by implementing a corresponding event source adapter through which requirement R1 is met (cf. Tab. 2).

EventReceiver The EventReceiver is responsible for converting the events of an event source into event objects that the aggregation engine can process. For example, one EventSourceAdapter receives events in form of an XML document and another adapter in the JSON format (cf. Section 2.1 and R2 in Tab. 2). Thus, the EventReceiver normalises

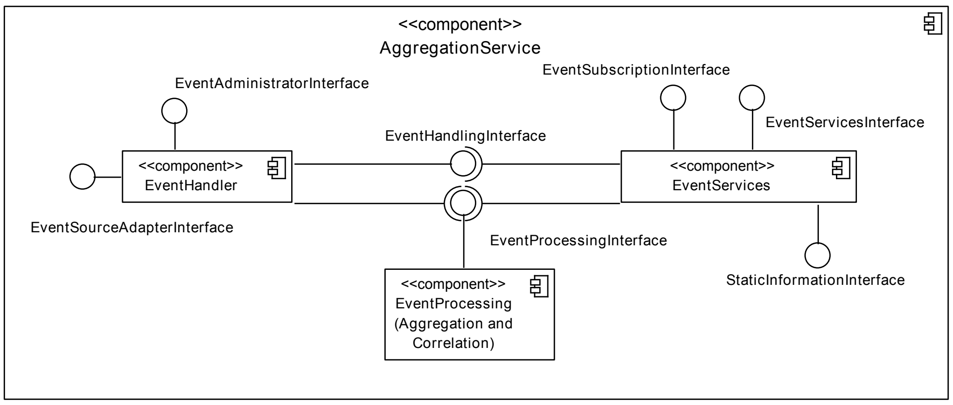

Fig. 3: Architecture Overview of the information aggregation engine.

events in different formats and converts them into the internal structure for processing, i.e., implementing requirement R3 shown in Tab. 2.

EventStore Events are stored in the EventStore, which realises requirement R4 in Tab. 2 of the aggregation engine.

EventManager The EventManager handles all operations on events. This component is the connection between the Event Receiver, the stores and the EventProcessing component via the EventProcessingInterface. In the same way, the connection to the EventServices component is established via the EventHandlingInterface. Thus, the EventManager is responsible to both save and load events and event types from the stores and thereby enables a synchronized access to events and event types.

In summary, the EventHandler is the central component of the Information Aggregation Engine.

4.2.2 The EventServices

The **EventServices** component handles the associations of events to information stored in the event store and handles the communication to event consumers. To this extent, it includes the EventSubscriptionInterface and the EventServicesInterface to external consumers as well as the EventProcessingInterface and the StaticInformationInterface. For internal communication to the EventHandler also the EventHandlingInterface is required, e.g., to reference a specific event type within a subscription. The following three main components are required for its realisation (cf. Fig. 5).

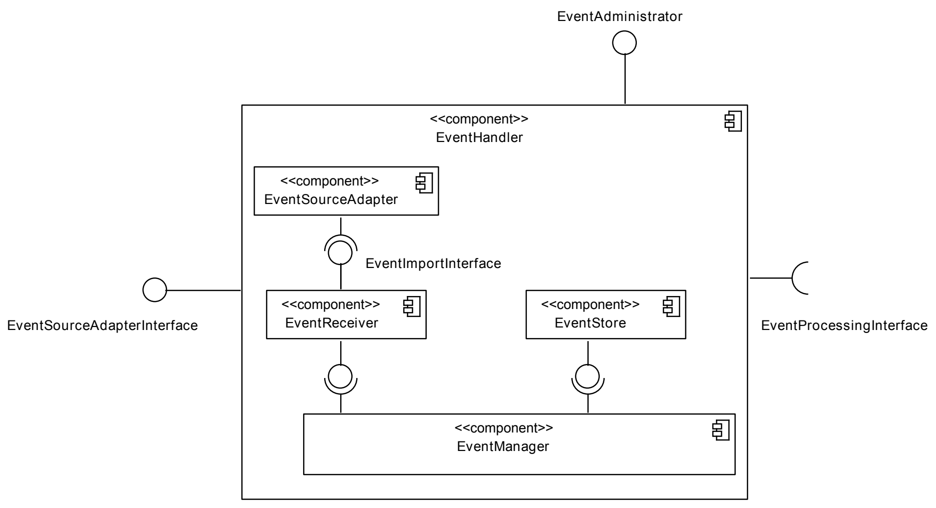

Fig. 4: Structure of the EventHandler component.

SubscriptionManager The SubscriptionManager handles the publication of events to the event consumers based on subscriptions they provided and which are stored in the subscription store. For this purpose, each subscription must include an address to which the events are pushed.

SubscriptionStore All subscriptions are administered in the SubscriptionStore. It thus mainly gathers the requests for receiving updates on events of interests, and serves as a repository where the targets for dispatching events are recorded.

ServiceUnits The ServiceUnits component is a placeholder for all upcoming functionalities that enrich events with external knowledge For example, the coordinates given by an event may be used to identify the city in which the event occurs. However, this requires that an external knowledge source to be accessible, where the boundaries of cities are given. A first idea of such enhanced event processing is published in [Me13].

Furthermore, predicting algorithms should be developed in this component, to implement the functionality discussed in Section 3.4. In particular, ServiceUnits are meant to be used to meet requirement R7.

In summary, the purpose of the EventServices component is to correlate events to logistics processes but also to external knowledge sources. It is therefore used to extend the platform and realise the requirements R5.3, R5.4, and R7 shown in Tab. 2. Furthermore, it is meant to be used to allow the subscription to events, thus implementing requirements R5.2 and R6.

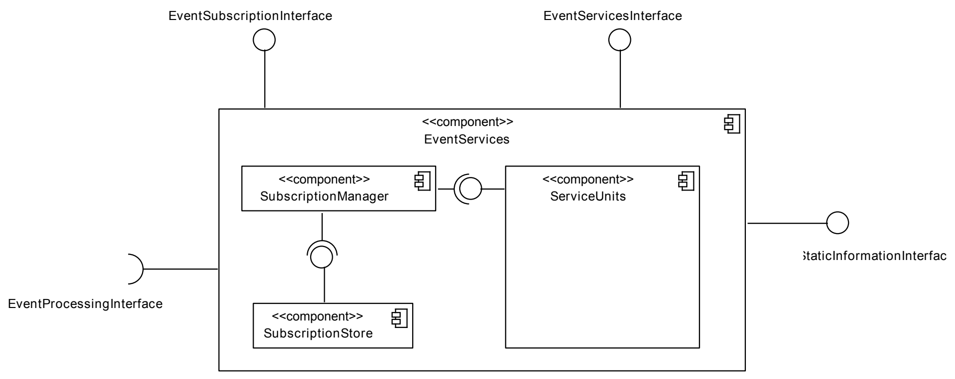

Fig. 5: Structure of the EventServices component.

5 Conclusion

Throughout this paper, the architecture of an event-based information aggregation engine in the context of smart logistics has been described. In particular, the requirements that the information aggregation engine must fulfil have been detailed. They serve as the basis according to which the architecture of the component is designed. Indeed, this paper ends with a thorough analysis of the interfaces offered by the event processing module, along with the description of its internal components and the functionalities offered.

Although all functional requirements are given, challenges may be faced during the implementation. This is due to the dynamic nature of the development process. These dynamics might occur during the implementation of the single components of the aggregation engine and their interaction. More integration effort and dynamics are expected by the integration of the aggregation service in the core GET Service platform. Challenges may also arise from technical requirements (hard- or software) or from necessary event sources that are not publicly available. Furthermore, the complexities of data integration for unifying data, messages, information, and events have to be faced.

Future work will be dedicated to the implementation of the described software components, with a particular focus on the enhancement of their interoperability and extendibility. Efforts will be also put in the devising of the automated process-model-to-queries task for monitoring and processing events, and on the realisation of prediction modules that foresee plausible delays or disruptions during the run-time execution of the transportation activities.

Acknowledgement

The presented research work has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement 318275 (GET Service).

References

[Ba13a] Backmann, Michael; Baumgrass, Anne; Herzberg, Nico; Meyer, Andreas; Weske, Mathias: Model-Driven Event Query Generation for Business Process Monitoring. In: ICSOC Workshops. S. 406–418, 2013.

[Ba13b] Baumgrass, Anne; Cabanillas, Cristina; Di Ciccio, Claudio; Meyer, Andreas; Schmiele, Jürgen: GET Service D6.1: Taxonomy of transportation-related events. `http://getservice-project.eu/en/project/public-deliverables`, 2013.

[Be94] Berge, John: The EDIFACT standards. Blackwell Publishers, Inc., 1994.

[Ca14a] Cabanillas, Cristina; Campara, Enver; Di Ciccio, Claudio; Koziel, Bartholomäus; Mendling, Jan; Paulitschke, Johannes; Prescher, Johannes: Towards a Prediction Engine for Flight Delays based on Weather Delay Analysis. In: EMoV. Jgg. 1185 in CEUR Workshop Proceedings. CEUR-WS.org, S. 49–51, March 2014.

[Ca14b] Cabanillas, Cristina; Di Ciccio, Claudio; Mendling, Jan; Baumgrass, Anne: Predictive Task Monitoring for Business Processes. In: BPM. Jgg. 8659 in Lecture Notes in Computer Science. Springer, S. 424–432, September 2014.

[CV95] Cortes, Corinna; Vapnik, Vladimir: Support-Vector Networks. Machine Learning, 20(3):273–297, 1995.

[Du13] Dumas, Marlon; La Rosa, Marcello; Mendling, Jan; Reijers, Hajo A.: Fundamentals of Business Process Management. Springer, 2013.

[EN10] Etzion, Opher; Niblett, Peter: Event Processing in Action. Manning Publications Co., Greenwich, CT, USA, 1st. Auflage, 2010.

[GV14] Günther, Christian W.; Verbeek, Eric: XES Standard Definition, 2014.

[Lu01] Luckham, David C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Me13] Metzke, Tobias; Rogge-Solti, Andreas; Baumgrass, Anne; Mendling, Jan; Weske, Mathias: Enabling Semantic Complex Event Processing in the Domain of Logistics. In: ICSOC Workshops. S. 419–431, 2013.

[MFP06] Mühl, Gero; Fiege, Ludger; Pietzuch, Peter R.: Distributed Event-based Systems. Springer, 2006.

[Mi97] Mitchell, Thomas M.: Machine Learning. McGraw Hill series in computer science. McGraw-Hill, Inc., New York, NY, USA, 1. Auflage, 1997.

[ROS11] Rozsnyai, Szabolcs; Obweger, Hannes; Schiefer, Josef: Event Access Expressions: A Business User Language for Analyzing Event Streams. In: AINA. IEEE Computer Society, S. 191–199, 2011.

[Tr13] Treitl, Stefan; Rogetzer, Patricia; Hrušovský, Martin; Burkart, Christian; Bellovoda, Bruno; Jammernegg, Werner et al.: GET Service D1.2: Use Cases, Success Criteria and Usage Scenarios, 2013.

[vD05] van Dongen, Boudewijn: The MXML standard. `http://www.processmining.org/WorkflowLog.xsd`, 2005.

[Ve13] van der Velde, Marten; Rook, Hans; Saraber, Paul; Grefen, Paul; Ernst, Albert Charrel: GET Service D2.1: Report Message Standards. `http://getservice-project.eu/en/project/public-deliverables`, 2013.

References

Baumgrass, Anne, Cristina Cabanillas, and Claudio Di Ciccio (2015). "A Conceptual Architecture for an Event-based Information Aggregation Engine in Smart Logitics". In: *EMISA*. Ed. by Jens Kolb, Henrik Leopold, and Jan Mendling. Vol. 248. Lecture Notes in Informatics (LNI). GI, pp. 109–123. ISBN: 978-3-88579-642-8. URL: http://dbis.eprints.uni-ulm.de/1293/.

BibTeX

```
@InProceedings{   Baumgrass.etal/EMISA2015:ConceptualArchitectureEvent,
  author        = {Baumgrass, Anne and Cabanillas, Cristina and Di Ciccio,
                  Claudio},
  title         = {A Conceptual Architecture for an Event-based Information
                  Aggregation Engine in Smart Logitics},
  booktitle     = {EMISA},
  year          = {2015},
  pages         = {109--123},
  publisher     = {GI},
  crossref      = {EMISA2015}
}
@Proceedings{     EMISA2015,
  title         = {Enterprise Modelling and Information Systems
                  Architectures, Proceedings of the 6th Int. Workshop on
                  Enterprise Modelling and Information Systems Architectures,
                  {EMISA} 2015, Innsbruck, Austria, September 3-4, 2015},
  year          = {2015},
  editor        = {Jens Kolb and Henrik Leopold and Jan Mendling},
  publisher     = {Gesellschaft f{\"{u}}r Informatik ({GI})},
  series        = {Lecture Notes in Informatics ({LNI})},
  volume        = {248},
  isbn          = {978-3-88579-642-8},
  url           = {http://dbis.eprints.uni-ulm.de/1293/}
}
```