The OM4SPACE Activity Service

A semantically well-defined cloud-based event notification middleware

Marc Schaaf*, Irina Astrova†, Arne Koschel‡, and Stella G. Grivas*

*University of Applied Sciences and
Arts Northwestern Switzerland
Olten, Switzerland
Email: {marc.schaaf|stella.gatziu-grivas}@fhnw.ch
†Institute of Cybernetics
Tallinn University of Technology
Tallinn, Estonia
Email: irina@cs.ioc.ee
‡Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
Email: akoschel@acm.org

***Abstract*—OM4SPACE provides a cloud-based event notification middleware. This middleware delivers a foundation for the development of scalable complex event processing applications. The middleware decouples the event notification from the applications themselves, by encapsulating this functionality into a component called Activity Service. This paper presents the architecture of the Activity Service, its application scenario, its semantic parameters, the implemented prototype of the Activity Service and the preliminary results of the performance evaluation of the prototype. The contribution of the paper is threefold: (1) we identified a new use case for the application scenario; (2) we extended a list of semantic parameters; and (3) we presented an implemented prototype of the Activity Service.**

Keywords–OM4SPACE; Activity Service; Cloud computing; Complex event processing (CEP); Active database management systems (ADBMSs); Smart grids.

I. Introduction

This paper provides an in-depth overview of the Activity Service, including details of its implementation and performance evaluation. The Activity Service was developed as part of the OM4SPACE (Open Mediation for Service-oriented architecture and Peer-to-peer Active Cloud Components) project [1], [2], [3], [4], [5], [6], [7], [8], which was started as a joint project of the University of Applied Sciences and Arts Hannover Germany and the University of Applied Sciences and Arts Northwestern Switzerland.

The idea behind the OM4SPACE project was to merge an event-driven architecture (EDA), a service-oriented architecture (SOA), complex event processing (CEP) and cloud computing together to provide a semantically well-defined cloud-based event notification middleware for decoupled communication between CEP application components on all the layers of a cloud stack, including IaaS (Infrastructure-as-a-Service), PaaS (Platform-as-a-Service) and SaaS (Software-as-a-Service). By decoupled, we mean that events are posted to the middleware without knowing if and how they are processed later.

The remainder of this paper is organized as follows. Section II gives an overview of a possible application scenario for the Activity Service. Section III presents the motivation for the Activity Service. Section IV describes the architecture of the Activity Service; it is followed by a discussion of its semantic foundation (Section V). Section VI gives an overview of transport technologies supported by the Activity Service. Section VII presents the implementation of the Activity Service. Section VIII evaluates the performance of the (implemented) Activity Service. Section IX gives an overview of the related work. Finally, Section X draws the conclusion, whereas Section XI discusses the future work.

II. Application Scenario

The OM4SPACE project defines an application scenario for CEP in a cloud environment. Such a scenario is placed in the domain of smart grids, whose main goal is to reduce peak energy consumption and energy wastage. This should be enabled, by dynamically controlling energy generation and consumption using active components. The application scenario covers the definition of actors, who participate in a smart grid, and event-based communication between them. Also, it details use cases, which constitute the background for defining semantic parameters later.

The Activity Service suits the requirements of smart grids very well because, on the one hand, smart grids provide active components producing and consuming events and, on the other hand, smart grids are targeted towards heterogeneous distributed cloud environments involving diverse transport technologies. Therefore, the application scenario for the Activity Service was settled into the domain of smart grids.

A smart grid is an electricity network that can intelligently integrate the actions of all users connected to it – generators,

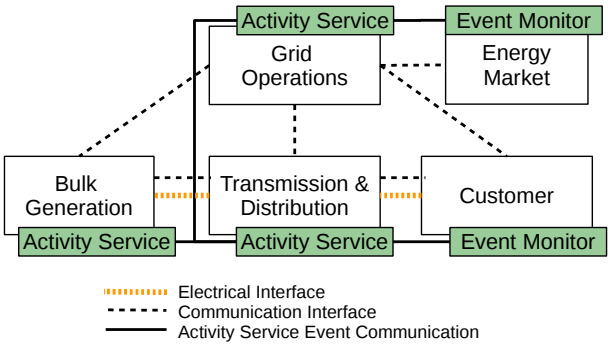

Figure 1. Smart grid actors and relationships among them (adapted from [9]).

consumers and those that do both – in order to efficiently deliver sustainable, economic and secure electricity supplies. Smart grids try to provide all these features with basically three approaches. First, they add dynamics to a power grid. This is done, by replacing passive components with active ones. Second, they add 'intelligence' to the power grid, meaning that the active components are able to communicate with each other and react to changes in the demand of energy. Third, they make the energy-related infrastructure flexible. As a result, new energy sources can easily be added to the power grid, whereas the transmission enables flexible routing.

Today's energy grids are based on mid-20th technologies and concepts. With the lasting gain of energy consumption, the existing infrastructures do not meet future requirements of energy generation, transmission, regulation and availability. Smart grids are 'intelligent' power grids as they are in development by power suppliers and public agencies. With green aspects in mind, smart grids aim to reduce the overall energy consumption and energy wastage.

Basically, there are two parties with interest in smart grids. On the one hand, there are utility providers, such as power authorities. They get enabled to provide stable power grids, get more control over the grids and reduce costs, by avoiding wasting the energy. Providers aim to produce a steady amount of energy, which means no peak or low levels in energy production. Further, the level of energy produced should dynamically fit the current demand of energy. Next to this, the transmission of energy should be minimized. On the other hand, there are customers who can benefit from reliable power grids, energy monitoring and saving, flexible pay scales and convenience features, like remote controlling of electric devices.

In the next subsections, the application scenario will be defined by a set of actors, relationships and use cases.

A. Actors

Actors have certain tasks and interests. In the context of the application scenario, we distinguish four actors: **Bulk Generation**, **Customers**, **Transmission and Distribution**, **Operations** and **Energy Markets** [9].

Bulk Generation simply produces energy. It can be, e.g., generating plants, wind power stations and solar energy plants. In a smart grid, **Bulk Generation** acts as one virtual power plant.

Customers are the consumers of energy. **Customers** may also be part of a virtual power plant as they can generate energy themselves (e.g., using photovoltaic cells). On the **Customers** side, a smart meter is of major importance. Smart meters can be considered as 'intelligent' electricity readers, which are capable of monitoring the energy consumption of attached devices in real time. Furthermore, they provide a history and evaluation of the recorded data. Smart meters are able to control attached and prepared electric devices, e.g., by switching on a washing machine.

Transmission and Distribution is the infrastructure for the grid's power transmission. It is directly connected to **Bulk Generators** and **Customers**.

Operations are a higher controlling and monitoring actor. **Operations** watch the smart grid's state continuously and provide information to the other actors. Furthermore, **Operations** are enabled to actively control the other actors in reaction to certain states of the smart grid.

Energy Markets are commodity markets that deal specifically with the trade and supply of energy (e.g., electricity).

B. Relationships

Basically, in a smart grid, there can be two different kinds of relationships between actors. These are energy transmission and information communication. But the application scenario focuses on the latter only, since this is where events are actually generated. Figure 1 shows all the actors along with the relationships between them. The actor **Operations** represents a smart grid to external partners. Therefore, it is connected to all other actors, thereby enabling to monitor and control the entire smart grid. **Transmission and Distribution** is the connector between **Bulk Generation** and **Customers** for both energy transmission and data transfer.

In the application scenario, two different mechanisms for information communication are defined: events and messages. Events are data that are actively provided by the actors to the smart grid. Events have no dedicated receiver, but are available to other actors as long as they are interested in certain events (this is a simplified view, with no respect to security aspects). Events themselves do not affect the behavior of single actors or smart grids. In fact, events can be collected and processed, which may result in reaction to an event (or a certain set of events). In the application scenario, events are used to add an activity to the actors. With events, actors broadcast information, which indicates their state. Examples of events are the current energy consumption by **Customers**, the current workload of an energy transmission route or the current percentage of 'green' energy produced by **Bulk Generation**.

Messages are data, which are sent from one distinct actor (a sender) to one or more other distinct actors (receivers). Messages are used to directly send commands from one actor to another. The sending of a message may be a reaction to the interpretation of events that currently happened. Examples of messages are **Operations** instructions to **Bulk Generation** to start **Customers** electric devices due to a low energy price or to reduce the energy production due to a lower demand of energy in the smart grid.

In addition to relationships between actors within a single smart grid, the application scenario defines relationships between smart grids. An example of such a relationship is a

situation where one smart grid offers its excessive energy for sale or purchase to another smart grid.

C. Use Cases

Tables I and II illustrate uses cases of various complexity to demonstrate the event-driven behavior of smart grids. A common precondition for all the use cases is that all the actors are completely "smart-grid-ready," meaning that the mandatory infrastructure, such as smart meters and remotely controllable devices, is available.

III. Motivation

Cloud computing is a new paradigm, which quickly finds its way into many IT areas. It offers vast resources and highly dynamic scalability while reducing the required upfront investment costs to a minimum [10]. However, the deployment of CEP applications into the cloud is still a hard task, especially when the applications should benefit from specialized cloud services. This is, however, often required in order to allow for high scalability as for example in case of the event-based communication among the various application components. Many cloud services exist from the various cloud providers as for example Amazon SNS. These services are highly optimized for the cloud as they are providing automatic scaling mechanisms and tight integration with the other provider services. However, their usage typically comes with the price of a vendor lock-in, because they have their own proprietary APIs. As such, they are highly provider-specific.

Current standardization approaches as for example OpenStack are mostly focused on infrastructure aspects and do not cover any standardized support for cloud-based CEP applications. This hampers with interoperability between cloud providers and the outside world because these approaches do not feature well-defined semantics [2], [3].

One simple example from the application scenario, which illustrates the need for well-defined semantics, is an attribute *time* in the following event definition:

```
Event:
    source: s_298
    time: 04:37:21
    temperature: 30
    humidity: 45
```

A time *04:37:21* given in the example above as part of the event definition is problematic, when considered that the time depends on the location where it was made. Thus, a precise time specification needs additional information, such as *according to UTC*. Furthermore, additional information is needed for the correct processing of the time as for example if the given time value represents the point in time when the reported event *occurred* or when it was *detected*. In case where an attribute *temperature* represents some form of an accumulated value, like the average temperature over ten minutes, does the *time* represent the start, the end or some other point in the time during the accumulation period? Does the *time* indicate that the event has already happened or it will happen soon? For example, the *temperature* could have a warning character. If the measuring sensor is approaching a certain threshold, it might send an event upfront to notify about an expected change.

The *time* is only one attribute of an event. But as the example above illustrated, since the semantics of event payload were not explicitly described in the event, further knowledge was required for the correct processing of the event. Unfortunately, cloud services being used to build CEP applications today lack the capability of explicitly specifying such semantics. As a result, when building a CEP application based on those services, the application becomes tightly linked to the provider-specific semantics, which results in a high risk of a vendor lock-in. In many cases, this is also true if measures are taken to hide the service-specific API, because quick abstraction approaches typically do not cover a specification of the overall semantic parameters, which are implicitly provided by the underlying transport technology.

IV. Architecture

Resulting from the motivation and application scenario requirements, we generally aimed at helping application developers to overcome the following challenges when building cloud-based CEP applications:

- Making CEP applications scalable, while minimizing changes to be done to the application design when deploying the applications into the cloud.
- Reducing the risk of vendor lock-in caused by the usage of provider-specific service offerings.
- Dealing with further complexity when crossing the border of a single cloud provider.
- Compensating for lack of semantics.

Within the OM4SPACE project, we developed the Activity Service as the combination of an event notification system and an event processing system to allow for easy event-based communications as well as for event based-rule evaluation and action triggering. In particular, the Activity Service monitors events. After the detection of an event, it notifies the component responsible for executing the corresponding rules (event signaling), which in turn triggers (or fires) these rules into execution. Rule execution incorporates condition evaluation as a first step and, if successful, action execution as the second step. Rules can temporarily be enabled or disabled.

In general, events occur within transactions, whereas rules are executed within transactions. A variety of execution models exist for the coupling of the transactions that raise events, evaluate conditions and execute actions, some giving rise to quite complex behavior. The adaption of such coupling mechanisms will be one of the major challenges in the later phases of the OM4SPACE project.

The general architecture concept follows the unbundling approach introduced in [11]. As such, we divided the Activity Service into three separate components: Event Service, Rule Execution Service and Event Monitor (EM). Each of these components provides one or more well-defined interfaces with a clear definition of their semantics. Thereby the concrete implementation of the different components is interchangeable. The communication between all the components in the architecture is done through events, where an event is any kind of information sent as a notification from one component to another.

Figure 2 illustrates the components of the Activity Service in relation to cloud-based event sources (also called event producers) and event consumers.

TABLE I. Use Case 1 overview

Use Case 1:	Dynamically adjust the bulk energy generation
Actors:	**Bulk Generation**, **Transmission and Distribution**, **Customer**
Preconditions:	*none*
Outcome:	The amount of produced bulk energy is adjusted to the actual energy demand
Trigger:	Calculated changes of the *RouteWorkload* triggered by *EnergyConsumption* events
Description:	To allow for an efficient operation of the whole smart grid, the overall production should be adapted to the actual consumption to reduce overproduction and losses. Modern bulk energy generation is often capable of quickly adjusting its energy production (e.g., gas turbine power plants). Smart meters can provide measurements on their energy consumption patterns allowing for a better estimation of the overall energy need. As such, the measurements generated by the smart meters shall be used to calculate the overall power consumption so that the bulk energy generation can adapt its production schedule.
Procedure:	1. Smart meters installed at the **Customer** premises produce *EnergyConsumption* events based on the accumulated actual energy consumption. 2. The **Transmission and Distribution** provider consumes the *EnergyConsumption* events, calculates the overall route workload for a given time interval and publishes it as an event. 3. The **Bulk Generation** consumes the *RouteWorkload* event and decides if adjustments in its production schedule are required and executes them if needed.

TABLE II. Use Case 2 overview

Use Case 2:	Intelligent energy production / purchasing
Actors:	**Operations**, **Bulk Generation**, **Energy Market**
Preconditions:	*none*
Outcome:	The cost for providing the required energy in the near future is optimized
Trigger:	The energy price on the **Energy Market** falls below a given threshold
Description:	The **Operations** use available energy from the **Energy Market** to optimize its energy cost during periods of high energy consumption.
Procedure:	1. The **Energy Market** raises an *EnergyPrice* event based on the current market situation. 2. The **Operations** receive the *EnergyPrice* event and evaluate it against a given threshold. If the price is below, the **Operations** use the most recent information received from the weather forecast provider as well as the most recent route workload (Use Case 1) to make a prediction for the near future energy consumption and publishes this as an *EnergyConsumtionForecast* event. 3. The **Bulk Generation** consumes the *EnergyConsumptionForecast* event, calculates its own energy production cost for a given time interval and publishes the result as an *EnergyProductionCostEstimate* event. 4. The **Operations** receive the *EnergyProductionCostEstimate* event and correlate the cost with the prices available on the **Energy Market**. If the prices are lower, the **Energy Market** buys the required energy and issues an *ExternalEnergyFeed* event. 5. The **Bulk Generation** consumes the *ExternalEnergyFeed* event and reduces its production schedule accordingly.

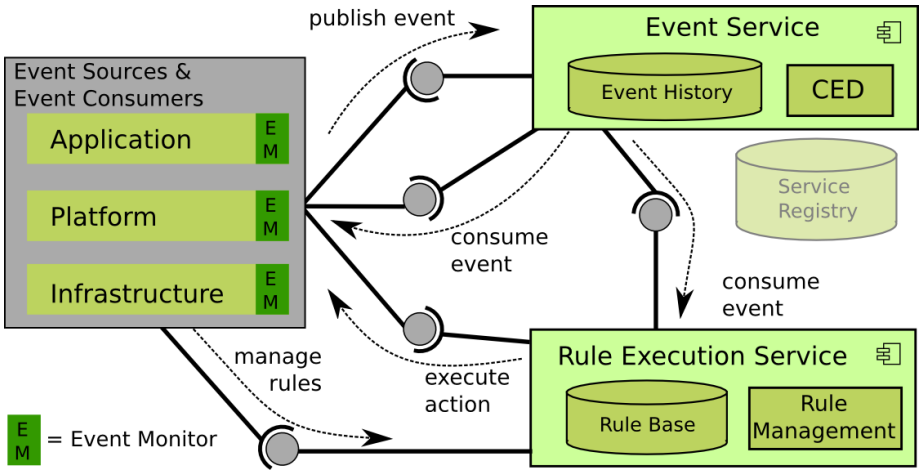

Figure 2. High-level architecture of the Activity Service [8].

A. Event Service

An Event Service represents the central component for the event-based communication. It provides all means necessary to be informed about the occurrence of events from the different event sources and to deliver the events to the subscribed receivers. To receive events from the Event Service, an event consumer has to implement an appropriate Event Handler Service, which needs to be published to the Service Registry. The Event Service discovers those services through the Service Registry and considers them as event subscribers. This mechanism stands in contrast to the commonly used subscriber mechanism where an event consumer needs to know the address of an event source or the event/message broker to directly register the subscription to them.

In the Activity Service, such a direct subscription is not required. Instead we follow the Whiteboard pattern [12] where the Service Registry acts as a 'whiteboard' to allow event consumers to advertise their interest in particular events. We consider this mechanism to be particularly suitable for a dynamic environment, such as the cloud, where the number of instantiated components can change frequently due to automatic scaling activities, making it a hard task to keep all available components up-to-date.

The Event Service further provides the capability to mediate among heterogeneous transport technologies. As part of this mediation process, it interprets semantic parameters to ensure that all participating transport technologies follow the specified behavior, e.g., they all use the requested level of encryption.

In addition to its mediation responsibilities, the Event Service also provides the CEP capabilities. The incoming events are stored into an Event History to support the monitoring of complex events. Its subcomponent Complex Event Detector (CED) evaluates the events and derives new complex events, which are fed back into the processing mechanism so that they can be delivered to registered subscribers or used again as input for the CED. The technical details of the complex event detection process are hidden from the event producers and consumers and can, thus, easily be changed without impacting the rest of the system.

B. Rule Execution Service

A Rule Execution Service extends the CEP capabilities of the Event Service by allowing for more complex rules, which are allowed to access additional background knowledge as part of their processing. The outcome of these extended rules is not required to be a new complex event, but can also represent the execution of an external action, like a remote service invocation. In detail, the rules result in the execution of action handlers. Such an action handler needs to be implemented by each of the components that are to be called from within rules.

The Rule Execution Service receives events from the Event Service to evaluate them against sophisticated rules. Therefore, it acts as an event consumer of the Event Service when registering an appropriate Event Handler Service. The evaluated rules are stored in a Rule Base, which can be managed by a special Rule Management Service.

C. Event Monitors

The Activity Service needs to obtain and process events from heterogeneous event sources that might even be spread across a single cloud. On the one hand, we consider active event sources, such as sources supporting triggers and callback mechanisms typically found in active database management systems (ADBMSs) or sources with internal triggers. On the other hand, we consider passive event sources, such as protocoled sources, which write all their actions into log files. For example, a smart meter usually realizes both types of event sources.

With the requirement to support different types of heterogeneous and highly distributed event sources, we designed a subcomponent called Capsule that hides from the Event Service all the details of a raw event source producing the event payload in a provider-specific format. In particular, the Capsule is responsible for converting the provider-specific format to the format used by the Event Service (e.g., web service calls) and annotating events with semantic parameters.

Figure 3 gives an overview of the architecture of a Capsule. This figure illustrates the raw event source together with the matching Capsule as part of the event producer (each producer has exactly one Capsule). The event producer can reside on any layer of the cloud stack (e.g., IaaS, PaaS and SaaS). Not shown in the figure is the unique capability of the Capsule to utilize event sources outside of the cloud as for example in case of an incoming shipment into a warehouse.

V. Semantic Parameters

The OM4SPACE project aimed at bringing the Activity Service into a cloud environment. Clouds are highly heterogeneous distributed environments, in which multiple different transport technologies can be used. Furthermore, clouds are likely to contain a plenty of heterogeneous event producers and event consumers. Thus, the Activity Service running in a cloud should be able to deal with any kind of events from any kind of event source (both active and passive). It should also provide the CEP capabilities across multiple proprietary and non-proprietary cloud environments.

Beside the issues concerning the transportation of events, the events themselves have to be more meaningful to be processed properly. Multiple different event producers and event consumers may have different requirements on how to interpret the meaning or context of events. Therefore, the events need to be enriched with semantic parameters.

The general concepts of event-based rule evaluation and action triggering have been established in the context of ADBMSs [13]. Therefore, we defined semantic parameters based on those typically found in ADBMSs. In particular, semantic parameters for the Activity Service $ASSP = ESP \cup TSP \cup DSP$ fall into three categories:

1) **Event semantic parameters** (ESP)
   Event semantic parameters describe the interpretation of events and their payload from a non-domain specific perspective. They describe general aspects of an event as for example the exact semantics of a given event timestamp or the lifetime of the event. In the Activity Service, these parameters are heavily influenced by the semantic parameters known from ADBMSs.
2) **Transport semantic parameters** (TSP)
   Transport semantic parameters describe how data are transferred within the Activity Service and enable

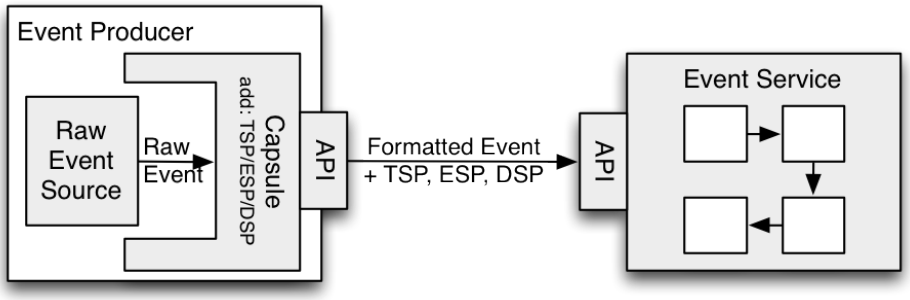

Figure 3. The Capsule annotating events with semantic parameters [3].

TABLE III. The defined event semantic parameters

Signaling point	$sp \in \{pre, post, instead\}$
Granularity	$g \in \{instanceoriented, setoriented\}$
Net effect	$ne \in \{yes, no\}$
Life Span	$ls \in \{explicit, implicit\}$
Consumption policy	$cp \in \{recent, chronicle, continuous, cumulative\}$
Coupling mode	$cp \in \{coupled, decoupled, immediate, deferred\}$
Strategy	$s \in \{parallel, arbitrary, priority, static, dynamic\}$

the usage of heterogeneous transport technologies as they appear in cloud environments due to the various provider-specific services. In other words, transport semantic parameters define in a technology-independent way all the information that must be provided by the underlying transport technology, like the need for confidential event communication or the guarantee that events are delivered exactly once.

3) **Domain-specific parameters** (DSP)
Domain-specific parameters describe the meaning of all relevant information in the application domain that uses the Activity Service. Although being defined separately, these parameters allow for the delivery of domain-specific information as part of the event signaling.

A. Event Semantic Parameters

We took event semantic parameters from ADBMSs and adapted them to the Activity Service. These parameters are defined as $ESP = \{sp, g, ne, ls, cp, cm, s\}$ where each parameter represents the following aspects (Table III):

The **signaling point** describes if an event was raised before the triggering state change happens (*pre*), after the state has already taken place (*post*) or the event replaces the actual state change by giving this notification only (*instead*). We consider all these values as valid options with the change that for the signaling point *pre*, we cannot guarantee that given rules are actually triggered by such an event before the triggering activity has been executed.

The **granularity** indicates the granularity of an event, viz., it can represent a simple singular state change or an aggregation of multiple initial events or state changes, thereby having another granularity. From the perspective of the Activity Service, we support the same granularities as they are typically defined in ADBMSs: *instance-oriented* where each single state change is considered as a single event and *set-oriented* where multiple state changes are considered as one event.

The **net effect** indicates if the event triggering activity had any actual effect and is strongly motivated by transaction handling in ADBMSs. As we are currently not supporting transaction handling, the Activity Service does not handle this parameter yet.

The **life span** defines how long an event is valid for processing. We consider two values (*implicit, explicit*) for the specification of the life span.

The **consumption policy** describes the order how events are processed. In the context of ADBMSs, four policies are defined: *chronicle* where events should be processed by the order of the event creation; *recent* where the last received event should be processed only; *continuous* where the order of receiving is the order of processing (FIFO); and *cumulative* where events are processed as one whole group. We consider the same values but have a special focus on the details of ordered handling as it requires some effort in a distributed system where events are prone to arrive unordered.

The **coupling mode** is also one parameter for transactional behaviors in ADBMSs. It indicates if an event happened within the transaction (*coupled*) or not (*decoupled*). It defines also if an event is thrown immediately or at the end of the transaction (*deferred*). Currently, the Activity Service does not cover transaction handling explicitly and thus, consider only the deferred decoupled value.

The **strategy** defines how the rule execution is triggered if multiple rules would be triggered by an event. The ADBMS semantic definition considers the following values: *parallel* where all matching rules are fired in an unpredictable order; *arbitrary* where one matching rule is picked randomly; *priority* where the rules have priorities and the rule with the highest priority is fired; *static* where a static order is given by an administrator; *dynamic* where the order is generated at runtime. In general, we aim to support all of the available parameters but with one important difference. As for smart grids it is usually the case that multiple rule execution components exist, a global ordering of the rule executions would be hard to achieve. Thus, we consider the given attributes per processing component and not on a global scope. So on a global scope, we only support

TABLE IV. The defined transport semantic parameters

Priority	$p \in \{low, normal, high\}$
Order of delivery	$od \in \{ordered, unordered\}$
Transport reliability	$tr \in \{bestEffort, atLeastOnce, exactlyOnce\}$
Confidentiality	$c \in \{yes, no\}$
Integrity	$i \in \{yes, no\}$
Authentication	$ae \in \{yes, no\}$
Authorization	$ao \in \{yes, no\}$
Transport technology specific	$ts \in \{...\}$

the parallel strategy and allow for a detailed specification per component.

B. Transport Semantic Parameters

The original ADBMS model does not cover distribution. Therefore, to reflect the distributed nature of cloud-based CEP applications, we invented transport semantic parameters. These parameters are defined as $TSP = \{p, od, tr, c, i, ae, ao, ts\}$ where each parameter represents the following aspects (Table IV):

The **priority** allows for the specification of the event importance with regard to its transport. Events with higher priority will be transported more quickly compared to lower priorities if the Activity Service is under heavy load. The current model supports a fixed set of three priority levels *low, normal, high*, which allows for easy mapping to various transport technologies.

The **order of delivery** specifies how events are to be delivered relative to their occurrence. If the ordered delivery is requested, the order in which the events have been published by the corresponding event producer will be kept. The ordering is, however, only guaranteed within the scope of each of the event producers separately. Ordering of the event publications between multiple event producers is not provided. The alternative unordered mode makes no guarantees for the event ordering.

The **transport reliability** parameter enables to specify the need for level of reliability for the event transport level. In general, two categories can be defined: no reliability (*bestEffort*) where no guarantees are given that an event will be correctly transported and guaranteed delivery. The guaranteed delivery is further divided into the categories: *atLeastOnce* where events are guaranteed to be delivered but might be delivered multiple times and *exactlyOnce* where events are guaranteed to be delivered only once.

As cloud-based CEP applications often need to integrate external data sources potentially via an unsecured network like the Internet, the specification of basic security mechanisms is part of transport semantic parameters. In particular, the **confidentiality** parameter enables to specify that events shall be transmitted in such a way that a third party is not able to eavesdrop on them. The **integrity** parameter enables to specify that transmitted events shall be protected from unnoticed modification by a third party. Typically both parameters require some form of authentication and authorization mechanism to be active. The **authentication** parameter enables to specify that an authentication of the communicating parties is required. Based on this, the **authorization** parameter can be used to request that communicating parties are authorized for accessing the transmitted events. As the authorization is only possible once an authentication has been done, it implies that the authentication is active.

In addition to the generally defined parameters, the **transport technology specific** parameter allows for the specification of parameters that are specific to a certain transport technology and thus, understood only by this technology. This parameter can be used, e.g., to optimize a transport technology for lower latency if fast communication is required.

C. Domain-specific Parameters

Domain-specific parameters give the meaning to domain-specific attributes. As an example, consider an event *RouteWorkload* with an attribute *workload* from the application scenario. This attribute has a self-explaining name, which is easy to understand by a human, but not by a software system. Thus, the attribute could be misunderstood by the other components of the system. The attribute needs a domain-specific parameter as for example *measure* with a value of *percent*, thereby explaining that the value of the attribute is given in percent so that all the actors in a smart grid can know the meaning of that attribute.

VI. Transport Technologies

The general communication between the Activity Service components is realized based on the concept of SOA. However, the actual method of message transportation is provided by technology-specific extensions as for example the usage of web services trough an enterprise service bus (ESB) or the use of messaging-based communication trough Amazons SNS or a common message-oriented middleware, like Apache ActiveMQ. This way the Activity Service enables the transparent use of different transport technologies.

The Activity Service is focused on providing a semantically well-defined abstraction from diverse transport technologies to reduce the risk of a vendor lock-in. Consequently, one of the main advantages of the Activity Service is its independence of service providers, such as WebLogic, Amazon and Google. In detail, once an event producer has sent events to the channel, the Activity Service located in the cloud will forward the events to the channel of an event consumer that is subscribed for those events. A decision on which channel to use for sending events is left solely to the event producer. Similarly, a decision on which channel to use for receiving events is left solely to the event consumer. For example, the event producer can select a JMS topic because it is not chargeable, whereas the event consumer can select an SQS queue because it is highly available (i.e., the availability of an SQS queue is not affected if the cloud instance fails). To be used in such scenarios, the Activity Service provides a generalized API along with semantic parameters. The actual transport technology is integrated into the Activity Service as an extension (plug-in), which has the responsibility to map the requested semantic parameters to its technology-specific configuration.

The Event Service as the central component for the event communication is designed to act as a mediator between different transport technologies. This allows the Activity Service to bridge the gap between multiple different provider-specific environments. Due to the explicit definition of the semantic parameters, application developers can rely on the

specified behavior even in complex setups where the event communication needs to be handled with multiple different transport technologies. Figure 4 illustrates such a scenario where event data are received from event producers outside a cloud through two transport technologies: Web Services Eventing (WSE) and web services (WSs). The event data are received by an Event Service, which is also located outside of the cloud that mediates between the aforementioned technologies and JMS-based communications link to another Event Service, which is located in the cloud. In its turn, this second Event Service mediates between JMS and the cloud internal communication service, Amazon SNS/SQS, which is used by the event consumers in the cloud.

In the current version, the Activity Service supports the following transport technologies: JMS and Amazon SNS/SQS.

A. Java Message Service

Java Messaging Service (JMS) provides an API that contains the abstraction of interfaces and classes, which are to be implemented by channel service provider on the basis of a Service Provider Interface (SPI) Adapter. The basic idea behind JMS is that an application can communicate over the JMS API through message-oriented middleware with any other (including non-Java) applications.

We selected JMS as an example of a topic channel from a "native" provider. As a native provider, Apache ActiveMQ was used. JMS was chosen also because the prototype of the Activity Service was implemented in Java.

B. Amazon SNS/SQS

Amazon provides a Simple Queue Service (SQS) and a Simple Notification Service (SNS). SQS is a web service for Amazon Elastic Compute Cloud (Amazon EC2) to decouple applications with message passing. It provides a distributed queue. Messages sent to an SQS queue are stored there until they are received and deleted by the consumer. SNS is another web service that lets endpoints subscribe for a topic and publish messages to that topic. SNS supports different endpoints, including SQS.

There are advantages of coupling SQS with SNS. SQS is a distributed queuing system, where messages are polled by consumers. Polling inherently introduces some latency in the delivery of messages in SQS unlike SNS where messages are immediately pushed to consumers. By coupling SNS with SQS, this latency can be avoided because SNS enables to send messages via an SQS queue to more than one consumer at the same time.

We selected SNS/SQS as an example of a queue channel from a foreign provider. Because of the decision to support SNS/SQS, EC2 was used as the cloud.

C. JMS vs. Amazon SNS/SQS

Table V summarizes our comparison of JMS and SNS/SQS.

JMS is a component of Java Enterprise Edition (JEE), whereas Amazon SNS/SQS abstracts the JEE-specific details of JMS. The main advantage of JMS over SNS/SQS is its independence of a channel service provider. But this also means that there has to be an administrator responsible for setting up the whole infrastructure. On the other hand, the main advantage of SNS/SQS over JMS is the high availability of a channel, which is not affected if a particular Amazon EC2 instance becomes unavailable. Messages waiting in queues for their delivery are stored redundantly on multiple servers and in multiple datacenters. Another big advantage of SNS/SQS over JMS is that there is no limit on the number of messages or the size of a particular queue. One message body can be up to 64 KB of text in any format (default is 8 KB). Large messages can be stored somewhere else reliably (e.g., in Amazon S3) and passed around a reference to the storage location instead of passing the message itself.

However, because of the dependency of Amazon, SNS/SQS is chargeable. There is a free usage tier for up to 100,000 requests per month. Beyond that, Amazon adds $0.01 per 10,000 requests to the bill. In addition, there is a need to pay for the data transfer. Data are transferred free of charge between SNS/SQS and an EC2 instance but within a single region only. Moreover, an SQS queue is distributed. Due to this fact, there is no guarantee that messages are delivered in the same order as they were sent. A sequence number can be added to every message in order to recover the original order of the messages. However, since SNS/SQS saves copies of messages at different servers of the queue, it might happen that in case of a server breakdown, single copies cannot be deleted and are sent twice to the consumer. Therefore, the consumer needs to be implemented in a way that it can handle these redundant messages. On retrieve-message-request, SNS/SQS delivers messages to some of the servers only. This means that it might happen that not all the messages in the queue are delivered or even no messages are delivered at all, if the number of messages is too low. But if the command is executed often enough, all the messages will be delivered step-by-step.

VII. Prototype

Based on the architecture of the Activity Service, we implemented its prototype in Java. The prototype was mostly focused on overcoming the technological gaps between different environments and cloud providers, by providing support for two different transport technologies (viz., JMS and SNS/SQS) and implementing the capability to mediate between them.

For the actual CEP, both the Event Service and the Rule Execution Service utilized the Esper CEP engine. The action handling was implemented based on web service calls against the defined action handler interface. Furthermore, the architecture was extended with a Registration Service, which provides a discovery mechanism, and a Mediation Layer, which provides the flexibility for different transport technologies.

A. Registration Service

To provide the required service discovery functionality for available event producers and consumers as well as the Event Service instances, a Registration Service was implemented that acts as the central service repository (Figure 5). The Registration Service offers its API based on a web service. Each Event Service registers itself via the offered API to announce its presence. Furthermore, all event consumers and producers register themselves via the API to announce either the events they offer or the events they want to receive.

Once the Registration Service has been informed about new event consumers and producers, or about changes in their registrations, it informs the available Event Services about those registrations. The communication with the Event Service

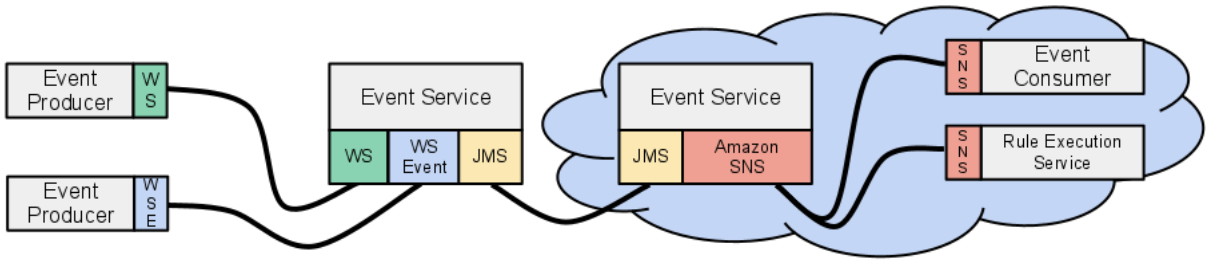

Figure 4. The Activity Service mediating among different transport technologies.

TABLE V. JMS vs. SNS/SQS

Feature	JMS	SNS/SQS
Max queue size	Limit depends on JVM heap and persistence store	Unlimited
Best Quality-of-Service (QoS)	Exactly once	At least once
Channel	Both topics and queues	Queues
Configurable retries	Supported	No
Persistence	Optional	Always
Scalability	Yes, depending of Message Broker	Inherent
Availability	Yes, depending of Message Broker	Inherent
Message order	Supported	Not guaranteed
Auto acknowledge	Supported	No
Message expiry	1 ms to unlimited	1 h to 14 d
Max message size	Unlimited	64 KB (default 8 KB)
Compression / Encryption	Yes	No
Language binding	Java	Java, PHP, Perl and C#

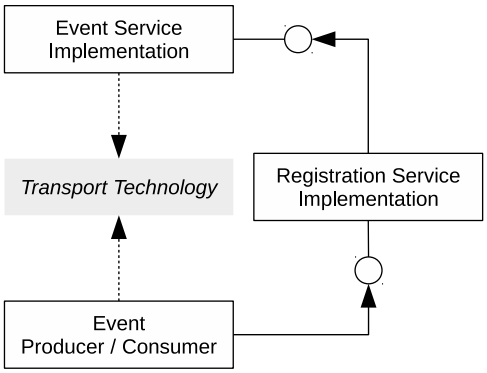

Figure 5. The Registration Service provides service discovery functionality.

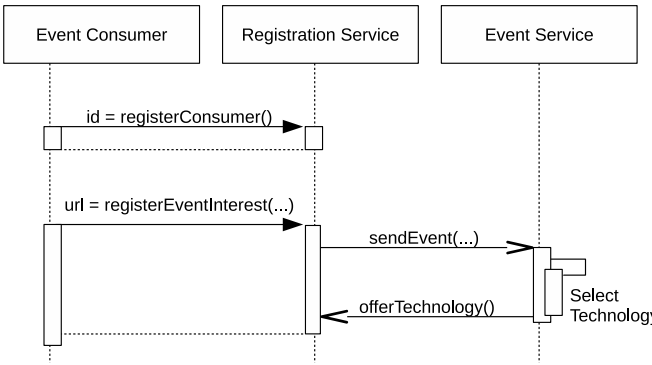

Figure 6. Registration of new event consumer.

is based on special management events. Figure 6 illustrates the registration process of a new event consumer. (The registration process of an event producer is done in a similar way.)

1) A new event consumer first requests a unique id from the Registration Service in order to label all further interactions.
2) Once the id has been assigned, the consumer informs the Registration Service about the type of events it is interested in via a *registerEventInterest()* call. In addition to the *ID* and the *event type*, the call contains a *prioritized list of transport technologies* that are accepted by the consumer for that event subscription.
3) The Registration Service forwards this information to an Event Service that selects suitable transport technologies supported on its side for the requested events.
4) The Event Service then generates the intersection of the list of transport technologies supported by the consumer and the list generated by itself, and selects from the 'common' transport technologies the one with the highest priority.
5) The Event Service then offers a transport technology specific URL, which can be used by the event consumer to receive events from the Event Service. Similarly to the initial call, this information is relayed by the Registration Service to the calling event consumer.

Based on the assigned id, the event consumer could later change or terminate its registration.

In the prototype, the Registration Service has to be started as the first component in order for the Event Service instances to discover and connect to the event consumers and producers.

Also, the Registration Service needs to be implemented as a highly available subsystem, because it is the backbone of the Activity Service dynamic behavior with regard to new or changed event subscriptions.

B. Mediation Layer

One of the main goals of the Activity Service is to bridge the gap between the various communication middleware systems around. Therefore, we designed and implemented a Mediation Layer, which is used by all the components of the prototype and hides the details of the underlying transport technology. In the prototype, the Mediation Layer supports the usage of both JMS and SNS/SQS-based messaging. However, the implementation of additional transport technologies especially from other cloud providers is also planned in the future. To ease such additions, we designed the Mediation Layer as a pluggable system that can easily be extended. Figure 7 gives an overview of the Mediation Layer, which consists of ReceiveMediator and SendMediator.

A ReceiveMediator handles different transport technologies for receiving events from heterogeneous event producers. Therefore, it is expandable for different transport technologies by different plug-ins. In addition, the ReceiveMediator transforms the messages it receives from the event producers to a generic format for the Event Service. This is necessary if the transport technologies keep the messages in different formats in their channels (either topics or queues). As an example, consider the smart meter in a private household that sends its real-time consumption to the distribution network via JMS. The ReceiveMediator receives JMS messages, but the Event Service might know only SNS/SQS because the instance is deployed in Amazon EC2. Therefore, the ReceiveMediator transforms the JMS messages to the XML structures that can then be forwarded to the Event Service via SNS/SQS for further processing.

After the processing, the Event Service sends the (new complex) events to a SendMediator, which distributes the events to heterogeneous event consumers. The SendMediator is a complement of the ReceiveMediator. Like the ReceiveMediator, the SendMediator is expandable for different transport technologies by different plug-ins. In addition, the SendMediator transforms the events it receives from the Event Service to a specific format for the event consumers (e.g., SNS/SQS messages).

The Activity Service does not have to know which of the event producers send events to it. But the Activity Service has to know which of the event consumers want to receive events from it. Therefore, the SendMediator needs an Event Consumer Repository. With such a repository, there is the possibility to store and query information about the event consumers. In the simplest case, this repository could be a database table with one entry for each event consumer. In particular, the Event Consumer Repository has to store the following information:

- The supported transport technologies for each event consumer.
- The values for each transport semantic parameter per event consumer.
- For each event consumer, the event types it is interested in. Not every event consumer wants to receive every kind of events. There are only certain types of events of interest. These types have to be stored in the repository. For example, the distribution network is interested in the real-time consumption of the connected households only.

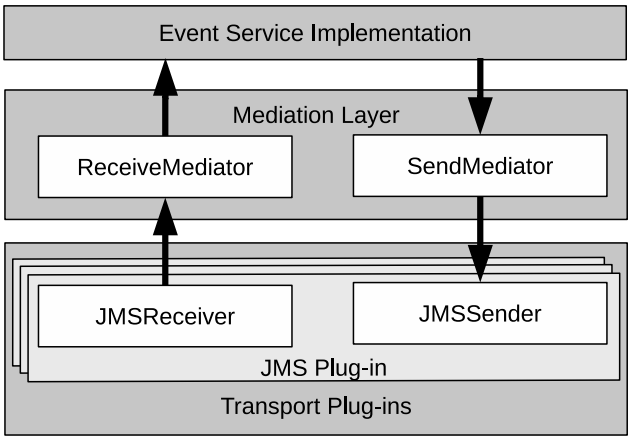

Figure 7. High-level architecture of the Mediation Layer.

With this information, the SendMediator knows if and how it should forward the events to the (registered) event consumers.

For the registration, an event consumer has to send a registration event to the SendMediator. This event must contain information like the used transport technologies and subscribed events. It must also contain the information about the channels of each event consumer so that the SendMediator knows where it has to forward the events later.

C. Capsule

A Capsule acts as the bridge between foreign (i.e., non-Java) applications and the Activity Service, by providing the functionality to forward events from a non-Java application to the Event Service as well as to forward events from the Event Service to the application. As such, the Capsule is implemented as a Java library that can directly be used by the event producing or consuming software. The Capsule also uses the Mediation Layer for supporting multiple transport technologies, while hiding the details from the application that uses the Capsule. The required configuration of the transport technology is possible via a separate XML configuration file and thus, independent of the application:

```
<Event transportType="jms">
  <tsp>
    <key name="choosingPrio">90</key>
    <key name="confidentiality">true</key>
    <key name="integrity">true</key>
    <key name="authorization">true</key>
    ...
  </tsp>
</Event>
```

However, the provided configuration does not contain the values for typical configuration parameters, like the address of the JMS message broker, because these parameters are dependent of the actually used Event Service. Thus, such connection-specific configurations are provided as part of the

registration process, which results in a URL describing the actual endpoint that is to be used (Section VII-A).

In general, the Capsule implementation can support three different modes of operation:

- *Event Consumer Capsule* where the Capsule is used to receive events from one or more Event Services.
- *Active Event Source Capsule* where the event source itself actively notifies the Capsule about new events that shall be transmitted by the Capsule to one or more Event Services.
- *Passive Event Source Capsule* where the Capsule has to detect that new events occurred in the event source, which is not capable to actively notify the Capsule itself.

In the current implementation, we only support the Event Consumer Capsule and the Active Event Source Capsule.

To support the semantic parameters defined by the Activity Service, the Capsule implements the event enrichment process, when it acts as an active event source. In this mode, it annotates each forwarded event with semantic parameters that can be specified in a separate XML configuration file. In particular, the Capsule receives the "raw" events, detects their type and enriches them with the correct values for event semantic and domain-specific parameters. However, not all of the event semantic parameters are set within the Capsule. In particular, event semantic parameters such as for example **consumption policy** and **strategy** cover parts of the rule execution and, thus, require knowledge about the used rules. This knowledge should not be placed into the Capsule because of the maintenance reasons. Therefore, these parameters are set within the Activity Service instead.

Furthermore, the Event Handler Service is responsible for the enrichment of the events with the correct values for transport semantic parameters. This task is not performed by the Capsule either because the transport technologies should be independent of a specific Capsule and, thus, the decisions how to transport the events are done later on within the Event Handler Service.

VIII. Performance Evaluation

As the Activity Service introduces an additional layer of abstraction, we suggested that the Activity Service is likely to have an impact on the overall communication performance. To check if our suggestion is true and to determine if a significant performance impact exists, we measured the Activity Service communication performance and compared the results against measurements taken by direct usage of the underlying transport technology. Each measurement was done with a different number of events (viz., 100, 500 and 1000) to see how the event number affects the performance. Next, we give an overview of these results, which were initially published in [1].

Figure 8 summarizes the test results for the direct usage of a JMS topic (Figure 9.A) and the usage of the Activity Service as a mediator between two JMS topics (Figure 9.B). As expected, the communication via the Activity Service was slower than the direct communication. But JMS still demonstrated a very good performance in all the tests even being interconnected with the Activity Service. For example, sending and receiving 100 events via JMS interconnected with the Activity Service took only 1286 msecs.

We expected that the time would increase with an increase of the number of events. Indeed, for sending and receiving 500 events, JMS interconnected with the Activity Service needed 3184 msecs more than for sending and receiving 100 events. However, of peculiar interest is the fact that for sending and receiving 1000 events, JMS interconnected with the Activity Service needed only 305 msecs more than for sending and receiving 500 events. In both cases, the average time was about 4500 msecs. Therefore, we suggested that extra time needed for sending and receiving 100 events was the time that the Activity Service needed for initialization.

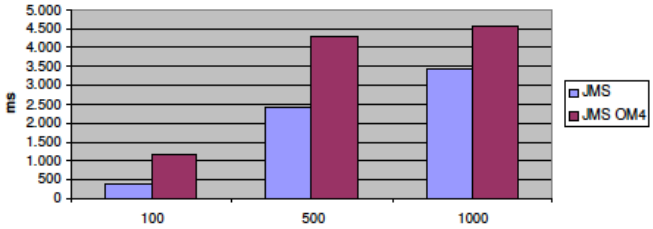

Figure 8. Comparison of event throughput via direct JMS communication and JMS communication through the Activity Service [1].

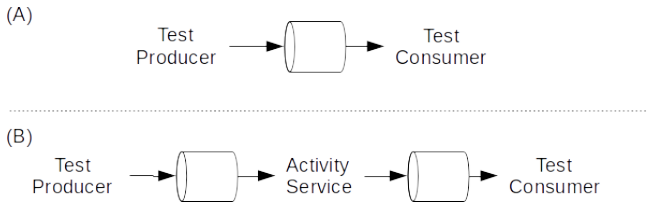

Figure 9. Test set-up for comparison of direct JMS to the Activity Service based JMS communication.

We conducted the same tests with SNS/SQS as the other currently supported transport technology. As expected, due to the distributed nature of an SQS queue, SNS/SQS alone was much slower than JMS alone. Furthermore, due to higher network delays caused by the usage of the cloud service and a less efficient implementation of the event consumer in the prototype, it resulted in a drastic reduction of the relative overhead of the Activity Service.

In general, the test results proved that the additional abstraction layer introduced by the Activity Service also introduces additional performance overhead. This certainly poses a problem for high performance/high throughput application scenarios such as smart grids that need to address the challenges related to the constantly increasing number of events and near real-time reaction on those events. However, the impact becomes less severe once the communication takes place over the across the border of a single cloud or network due to the added latency.

IX. Related Work

The foundation for the Activity Service can be found in the previous work on ADBMSs [14], [15], [16], [17]. An ADBMS provides a rule and execution model with well-defined proven semantics. It also provides a rule definition

language to specify event types, conditions, actions, and their assembly into rules. Much work has been done in the ADBMS context regarding the detection of so called complex events [18], [19]. Complex events are expressions of an event algebra, which are formulated over primitive or complex event types by means of algebraic operators (e.g., say E1 and E2 are event instances, conjunction (E1 ˆ E2) means that E1 and E2 occurred independently of their sequence. A number of technologies for the discovery of complex events are used, such as finite state automates, Petri nets and event trees [14].

In addition to the common database functionality, an ADBMS offers the capability to react to predefined events, by executing the appropriate rules. It provides connectors for event detection, condition evaluation and action execution; all of these are exposed as an overall functionality of an active mechanism. In ADBMSs, the active mechanism is usually closely tied to the systems as a whole. (This is due to the monolithic architecture of ADBMSs.) Therefore, one step beyond the work in ADBMSs go approaches to unbundling the active mechanism from ADBMSs and making it usable in other contexts [11]. For example, in [20], it was prosed to integrate the active mechanism into a rule service for CORBA-based distributed environments. In fact, this work formed a solid starting point for the development of the Activity Service.

Furthermore, we used the original ADBMS model for the definition of semantic parameters for the Activity Service (viz., event semantic parameters). However, the ADBMS set of semantic parameters does not yet cover central aspects that arise from the distributed nature of cloud-based CEP applications. Moreover, they do not cover domain-specific parameters, which can greatly ease the development of cloud-based CEP applications. Therefore, we greatly extended the ADBMS set of semantic parameters with transport semantic parameters and domain-specific parameters for the Activity Service.

Being an important part of the Activity Service, distributed event monitoring systems [19], [21] are an excellent instrument for (distributed) monitoring systems and can contribute to the general monitoring principles of the Activity Service. However, such systems mainly focus on primitive (mostly pure) event sources, like operating system level signals. The Activity Service, on the other hand, has to deal with event sources that are typically found in heterogeneous cloud environments. Event modeling aspects and semantics often lack precision [19] when compared to ADBMSs. Nevertheless, general work on the design of monitoring services for distributed event monitoring systems is valuable for a transfer into the cloud.

Some event monitoring and propagation within the cloud in conjunction with CEP are discussed in [22]. However, rule processing with precisely defined semantics was not the focus there either. On the other hand, several approaches to bringing the CEP technology into the cloud computing domain exist as for example [23], [24]. However, in contrast to the Activity Service, they are application-specific. The first attempt to rule processing within the cloud in conjunction with CEP has been done in [24]. At least some combination of EDA and SOA for the cloud was discussed there. However, the work remains quite high-level and mainly focuses on policy-driven CEP in the cloud. It does not really adapt the active mechanism of an ADBMS to the cloud, in particular, not with well-defined ADBMS semantics.

Web service development standards, such as the business process execution language WSBPEL [25], usually operate on a higher level than the Activity Service. However, they are an excellent example of web services-based systems, which can generate events as for example ”a process or an activity has started or ended.” Such events can then be monitored and acted upon, by using the Activity Service across the whole (heterogeneous) cloud.

X. Conclusion

In smart grids, diverse transport technologies are often involved and a large number of events occur on different layers of the cloud (e.g., IaaS, PaaS and SaaS). This paper provided an in-depth overview of the Activity Service starting from its application scenario, motivation and high-level architecture, and ending with its implementation and performance evaluation. Continuing our successful previous work [1], [2], [3], [4], [5], [6], [7], [8], this paper made additional contributions to the Activity Service. These are a new use case for the application scenario, an extended list of semantic parameters and an implemented prototype.

The Activity Service was developed as a transport technology independent event notification middleware to reduce the risk of a vendor lock-in. It offers an approach to managing events from heterogeneous event sources, processing these events in near real time and triggering appropriate actions on the events. In addition, the Activity Service can be used for monitoring events occurred in the cloud and for scaling CEP applications deployed in the cloud (e.g., starting new virtual machine instances when a certain threshold for the CPU load has been exceeded). A particular highlight of the Activity Service compared to the other work in that area is that the Activity Service is based upon a semantically well-defined rule and execution model. This model is a significant extension of the work originating from the ADBMS area into nowadays distributed, heterogeneous, cloud-based world.

XI. Future Work

In the future, the Activity Service seeks to support more transport technologies, including Web Services Notification, Web Services Eventing and Google App Engine.

A. Web Services Notification and Web Services Eventing

Both Web Services Notification (WSN) and Web Services Eventing (WSE) define a standard web service approach to exchanging notification messages. Both are based on an event-driven or notification-based architecture and use a topic-based publish-subscribe pattern. The difference between the two is that WSN is an OASIS5 standard, whereas WSE is a W3C6 standard. In other words, they are competing specifications with exactly the same idea. However, for the Activity Service, WSN could be a better choice because it supports small devices (with a restricted set of mandatory features) and enables direct and brokered notifications. Also, it offers transformation and aggregation of topics. Furthermore, there are semantic parameters (e.g., available subscription types and broker federations) that are important for high scalability.

B. Google App Engine

Google App Engine's API allows for a persistent connection between a client (HTML or JavaScript) and a server (Python, Java or Go). New information for clients will be pushed through a channel. The clients can subscribe to that channel in order to receive the new information from servers.

A server creates a unique channel for each client and sends a unique token to each client. The server side also receives update messages of the clients and sends those updates to the clients via their channels. A client is primarily responsible for the connection with the channel over the received token. It listens to the channel for updates, makes use of the data and sends the updates to the server. The client id identifies each client on the server. The tokens are responsible for allowing the client to connect and listen to the channel, which is the one-way communication path for the server to send updates to the client.

XII. Acknowledgements

We would like to thank all team members of the OM4SPACE project for their work.

Irina Astrova's work was supported by the Estonian Centre of Excellence in Computer Science (EXCS) funded mainly by the European Regional Development Fund (ERDF). Irina Astrova's work was also supported by the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12.

References

[1] I. Astrova, A. Koschel, A. Olbricht, M., Popp, and M. Schaaf, "Performance evaluation of OM4SPACE's Activity Service," In Proceedings of the 6th International Conferences on Advanced Service Computing, IARIA, pp. 58-61, 2014.

[2] R. Sauter, A. Stratz, S. Grivas, M. Schaaf, and A. Koschel, "Defining events as a foundation of an event notification middleware for the cloud ecosystem," In Proceedings of the 15th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, LNCS, vol. 6882, pp. 275-284, 2011.

[3] M. Schaaf, A. Koschel, and S. Grivas, "Towards a semantic definition for a cloud-based event notification service," In Proceedings of the 3rd International Conference on Cloud Computing and Services Science, pp. 345-349, 2013.

[4] M. Schaaf, A. Koschel, S. Gatziu, and I. Astrova, "An ADBMS-style Activity Service for cloud environments," In Proceedings of the 1st International Conference on Cloud Computing, GRIDs, and Virtualization, IARIA, pp. 80–85, 2010.

[5] I. Astrova, A. Koschel, S. Grivas, M. Schaaf, I. Hellwich, S. Kasten, N. Vaizovic, and C. Wiens, "Active mechanisms for cloud environments," In Proceedings of the Sixth International Conference on Digital Society, IARIA, pp. 109–114, 2012.

[6] I. Astrova, A. Koschel, L. Renners, T. Rossow, and M. Schaaf, "Integrating structured peer-to-peer networks into OM4SPACE project," In Proceedings of the 27th International Conference on Advanced Information Networking and Applications Workshops, pp. 1211–1216, IEEE, 2013.

[7] M. Schaaf, A. Koschel, and S. Grivas, "Event processing in the cloud environment with well-defined semantics," In Proceedings of the 1st International Conference on Cloud Computing and Services Science, pp. 176–179, 2011.

[8] A. Koschel, A. Hödicke, M. Schaaf, and S. Grivas, "Supporting smart grids with a cloud-enabled Activity Service," In Proceedings of the 27th International Conference on Environmental Informatics for Environmental Protection, Sustainable Development and Risk Management, Berichte aus der Umweltinformatik, pp. 205–213, 2013.

[9] NIST Framework and Roadmap for Smart Grid Interoperability Standards. Release 2.0. NIST Special Publication 1108R2. Available: http://www.nist.gov/smartgrid/upload/NIST_Framework_Release_2-0_corr.pdf Last accessed: November 2014.

[10] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," Technical report, EECS Department, University of California, Berkeley, 2009.

[11] S. Gatziu, A. Koschel, G. Bültzingsloewen, and H. Fritschi, "Unbundling active functionality," ACM SIGMOD Record, vol. 27, no. 1, ACM, pp. 35-40, 1998.

[12] OSGi Alliance, "The Whiteboard pattern," Technical Whitepaper, Available: http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf Last accessed: November 2014.

[13] D. McCarthy and U. Dayal, "The architecture of an active database management system," In Proceedings of the ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, pp. 215–224, 1989.

[14] K. Dittrich and S. Gatziu, "Aktive Datenbanksysteme, Konzepte und Mechanismen," Int. Thomson Publishing GmbH, Bonn, Albany, Attkirchen, 1996.

[15] N. Paton (editor), "Active rules for databases," Springer, New York. 1999.

[16] The ACT-NET Consortium, "The active database management system manifesto: a rulebase of ADBMS features," In ACM SIGMOD Record, vol. 25, no. 3, ACM, pp. 414-471, 1996.

[17] J. Widom and S. Ceri (editors), "Active database systems: triggers and rules for advanced database processing," Morgan Kaufmann Publishers, Inc., San Francisco, California, U.S.A. 1996.

[18] S. Gatziu and K. Dittrich, "An event definition language for the active object-oriented database system SAMOS," In Proceedings of the Conference on Datenbanksysteme in Büro, Technik und Wissenschaft, Braunschweig, Germany, 1993.

[19] S. Schwiderski, "Monitoring the behaviour of distributed systems," PhD thesis, Selwyn College, University of Cambridge, UK, 1996.

[20] A. Koschel, "Distributed events in active database systems: Letting the genie out of the bottle," In Data Knowledge Engineering, vol. 25, no. 1-2, 1998.

[21] B. Schroeder, "On-line monitoring: a tutorial," IEEE Computer, vol. 28, no. 6, pp. 72–80, 1995.

[22] D. Luckham, "The power of events," Addison-Wesley, Boston, MA, 2002.

[23] G. Wishnie and H. Saiedian, "A complex event routing Infrastructure for distributed systems," In Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, pp. 92-95, 2009.

[24] P. Goyal and R. Mikkilineni, "Policy-based event-driven services-oriented architecture for cloud services operation and management," In Proceedings of the IEEE International Conference on Cloud Computing, pp. 135-138, 2009.

[25] OASIS WSBPEL TC. Web Services Business Process Execution Language ver. 2.0, Oasis standard, OASIS, 2007.