Principles and Applications of Distributed Event-Based Systems

Annika M. Hinze
University of Waikato, New Zealand

Alejandro Buchmann
Technische Universität Darmstadt, Germany

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE

Hershey · New York

Director of Editorial Content: Kristin Klinger
Director of Book Publications: Julia Mosemann
Acquisitions Editor: Lindsay Johnston
Development Editor: Julia Mosemann
Publishing Assistant: Keith Glazewski
Typesetter: Michael Brehm
Production Editor: Jamie Snavely
Cover Design: Lisa Tosheff
Printed at: Yurchak Printing Inc.

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 12
Event Processing in Web Service Runtime Environments

Anton Michlmayr
Vienna University of Technology, Austria

Philipp Leitner
Vienna University of Technology, Austria

Florian Rosenberg
CSIRO ICT Centre, Canberra, Australia

Schahram Dustdar
Vienna University of Technology, Austria

ABSTRACT

Service-oriented Architectures (SOA) and Web services have received a lot of attention from both industry and academia. Services as the core entities of every SOA are changing regularly based on various reasons. This poses a clear problem in distributed environments since service providers and consumers are generally loosely coupled. Using the publish/subscribe style of communication service consumers can be notified when such changes occur. In this chapter, we present an approach that leverages event processing mechanisms for Web service runtime environments based on a rich event model and different event visibilities. Our approach covers the full service lifecycle, including runtime information concerning service discovery and service invocation, as well as Quality of Service attributes. Furthermore, besides subscribing to events of interest, users can also search in historical event data. We show how this event notification support was integrated into our service runtime environment VRESCo and give some usage examples in an application context.

INTRODUCTION

Following the Service-oriented Architecture (SOA) paradigm shown in Figure 1, service providers register services and corresponding descriptions in registries. Service consumers can then find services in the registry, bind to the services that best fit their needs, and finally execute them. Web services (Weerawarana, Curbera, Leymann, Storey, & Ferguson, 2005) are one widely adopted realization of SOA that build upon the main stan-

DOI: 10.4018/978-1-60566-697-6.ch012

dards SOAP (communication protocol), WSDL (service description) and UDDI (service registry). Over the years, a complete Web service stack has emerged that provides rich support for multiple higher level functionalities (e.g., business process execution, transactions, metadata exchange etc.).

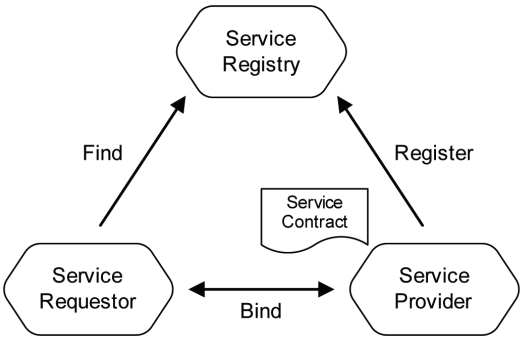
Practice, however, has revealed some problems of the SOA paradigm in general and Web services in particular. The idea of public registries did not succeed which is highlighted by the fact that Microsoft, SAP and IBM have shut down their public registries in the end of 2006. Moreover, there are still a number of open issues in SOA research and practice (Papazoglou, Traverso, Dustdar, & Leymann, 2007), such as dynamic binding and invocation, dynamic service composition, and service metadata.

One reason for these issues stems from the fact that service interfaces, service metadata and Quality of Service (QoS) attributes change regularly. Furthermore, new services are published, existing ones might be modified, and old services are finally deleted from the registry. This is problematic since service providers and consumers are usually loosely coupled in SOA. Thus, service consumers are not aware of such changes and, as a result, might not be able to access changed services any more. In this regard, the lack of appropriate event notification mechanisms limits flexibility because service consumers cannot automatically react to service and environment changes.

The current service registry standards UDDI (OASIS International Standards Consortium, 2005a) and ebXML (OASIS International Standards Consortium, 2005b) introduce basic support for event notifications. Both standards have in common that users are enabled to track newly created, updated and deleted entries in the registry. However, additional runtime information concerning service binding and invocation as well as QoS attributes are not taken into consideration by these approaches.

We argue that receiving notifications about such runtime information is equally important

Figure 1. Service-oriented architecture (Michlmayr, Rosenberg, Platzer, Treiber, & Dustdar, 2007)



and should, therefore, be provided by SOA runtime environments. Furthermore, complex event processing mechanisms supporting event patterns, and search in historical event data are needed for keeping track of vast numbers of events. In this chapter, we focus on such runtime event notification support. Our contribution is threefold: firstly, we present the background of this work and describe the motivation based on a case study from the telecommunications domain. Secondly, we introduce the VRESCo runtime environment (Michlmayr, Rosenberg, Platzer, Treiber, & Dustdar, 2007) and describe its notification support in detail. This includes event types, participants, ranking, correlation, subscription, and notification mechanisms, as well as event persistence, event search, and event visibility. Finally, we show some usage examples and point to further application scenarios enabled by our work.

BACKGROUND

This section consists of two main parts. In the first part, we summarize several research approaches that are related to our work. In the second part, we briefly introduce the open source event processing engine Esper which we use as technical background for our prototype.

Related Work

Event-based systems in general, and the publish/subscribe pattern in particular have been the focus of research within the last years. This research has led to different event-based architecture definition languages, for instance Rapide (Luckham & Vera, 1995), and QoS-aware event dissemination middleware prototypes (Mahambre, Kumar, & Bellur, 2007). Moreover, data and event stream processing has also been addressed in various prototypes, such as STREAM (Arasu, et al., 2008) or Esper (EsperTech, 2008).

Approaches to integrate publish/subscribe and the SOA model resulted in the two specifications WS-Notification (Oasis International Standards Consortium, 2006) and WS-Eventing (World Wide Web Consortium, 2006). While WS-Eventing uses content-based publish/subscribe, WS-Notification provides topics (WS-Topics) as a means to classify events. In both specifications, publishers, subscribers, and the event infrastructure are implemented as Web services. However, event processing mechanisms besides topic- and content-based filtering of events are not addressed by these specifications. The combination of SOA and event-driven architectures is further addressed by Enterprise Service Bus (ESB) implementations (e.g., Apache Servicemix[1]). In contrast to our work, ESBs mainly focus on connecting various legacy applications by using a common bus that performs message routing, transformation and correlation.

Cugola and Di Nitto (Cugola & di Nitto, 2008) give a detailed overview of other research approaches combining SOA and publish/subscribe. Furthermore, they introduce a system that aims at adopting content-based routing (CBR) in SOA. Their approach is built on the CBR middleware REDS (Cugola & Picco, 2006), and provides notifications following WS-Notification. Service discovery is implemented according to the query-advertise style using UDDI inquiry messages. In this work CBR is mainly used to perform service discovery, while we focus on event processing and notifications in service runtime environments. Additionally, we also provide support for dynamic binding and invocation, as well as QoS attributes and service metadata.

Service registries (e.g., UDDI, ebXML) represent one part of the SOA triangle that is responsible for maintaining a service repository including publishing and querying functionality. Both UDDI and ebXML provide subscription mechanisms to get notified if certain events occur within the service registry. However, these notifications are limited to the service data stored in the registry and do not include service runtime information. Notifications are sent per email or by invoking listener Web services. Other approaches such as Active Web Service Registries (Treiber & Dustdar, 2007) use news feeds such as Atom (Sayre, 2005) for dissemination of changes in the service repository content. News feeds enable to seamlessly federate multiple registries, yet, in contrast to our approach do not provide fine-grained control on the received notifications since they follow the topic-based subscription style. Furthermore, similar to UDDI and ebXML, these approaches do not include service runtime information.

There are several approaches that address search in historical events. Rozsnyai et al. (Rozsnyai, Vecera, Schiefer, & Schatten, 2007) introduce the Event Cloud system aiming at search capabilities for business events. Their approach uses indexing and correlation of events by using different ranking algorithms. In contrast to our approach, the focus of this work is on building an efficient index for searching in vast numbers of events whereas subscribing to events and getting notified about their occurrence is not addressed.

Li et al. (Li, et al., 2007) present a data access method which is integrated into the distributed content-based publish/subscribe system PADRES. The system enables to subscribe to events published in both the future and the past. In contrast to our work, the focus is on building a large-scale distributed publish/subscribe system that provides routing of subscriptions and queries.

Jobst and Preissler (Jobst & Preissler, 2006) present an approach for business process management and business activity monitoring using event processing. The authors distinguish between SOA events regarding violation of QoS parameters and service lifecycle, and business/process events building upon the Business Process Execution Language (BPEL). These events are fired by receive and invoke activities within BPEL processes. Unlike our approach, the focus is on search and visualization of business events whereas subscribing to events is not addressed. Furthermore, the different SOA events are not described in detail.

Esper

The open source engine Esper (EsperTech, 2008) provides event processing functionality and is available for both Java and C#. Esper supports several ways for representing events. Firstly, any Java/C# object may be used as an event as long as it provides getter methods to access the event properties. Event objects should be immutable since events represent state changes that occurred in the past and should therefore not be changed. Secondly, events can be represented by objects that implement the interface java.util.Map. The event properties are those values that can be obtained using the map getter. Finally, events may be instances of org.w3c.dom.Node that are XML events. In that case, XPath expressions are used as event properties.

Additionally, Esper provides different types of properties that can be obtained from events:

- Simple properties represent simple values (e.g., name, time).
- Indexed properties are ordered collections of values (e.g., user[4])
- Mapped properties represent keyed collections of values (e.g., user['firstname'])
- Nested properties live within another property of an event (e.g., Service.QoS)

In Esper, subscriptions are done by attaching listeners to the Esper engine, where each listener contains a query defining the actual subscriptions. These listeners implement a specific interface that is invoked when the subscription matches incoming events. The queries use the Esper Event Processing Language (EPL) which is similar to the Structured Query Language (SQL). The main difference is that EPL is formulated on event streams whereas SQL uses database tables: select clauses specify the event properties to retrieve, from clauses define the event streams to use, and where clauses specify constraints. Furthermore, similar to SQL there are aggregate functions (e.g., sum, avg, etc.), grouping functions (group by), and ordering structures (order by). Multiple event streams can be merged using the insert clause, or combined using joins. In addition to that, event streams can be joined with relational data using SQL statements on database connections. To give a simple example, the following EPL query triggers when a new service is published by 'TELCO1'.

```
select * from ServicePublishedEvent where
Service.Owner.Company = 'TELCO1'
```

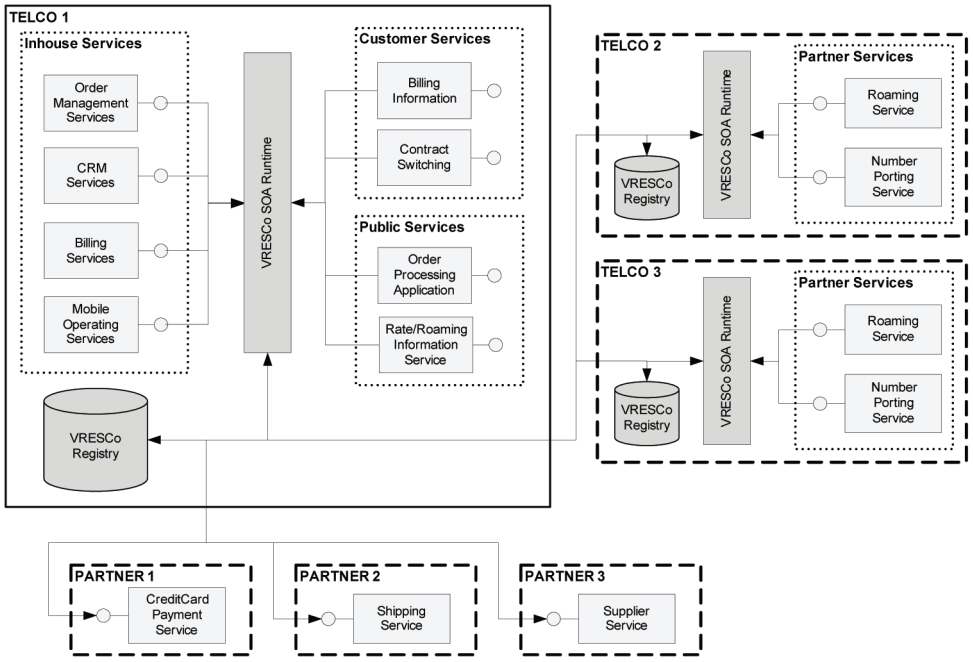
EPL provides a powerful mechanism to integrate temporal relations of events using sliding event windows. These operators define queries for a given period of time. For instance, if QoS events regularly publish the QoS values of services, then subscriptions can be defined on the average response time during the last 6 hours as shown in the following simplified example.

```
select * from QoSEvent win:time(6 hours).
stat:uni('ResponseTime')

where average > 300
```

Finally, EPL supports subqueries, output frequency, and event patterns. The latter are used to define relations between subsequent events (e.g., representing 'followed by' relations). For

Figure 2. TELCO case study (Michlmayr, Rosenberg, Leitner, & Dustdar, 2008)



more information on Esper and EPL we refer to (EsperTech, 2008).

VIENNA RUNTIME ENVIRONMENT FOR SERVICE-ORIENTED COMPUTING

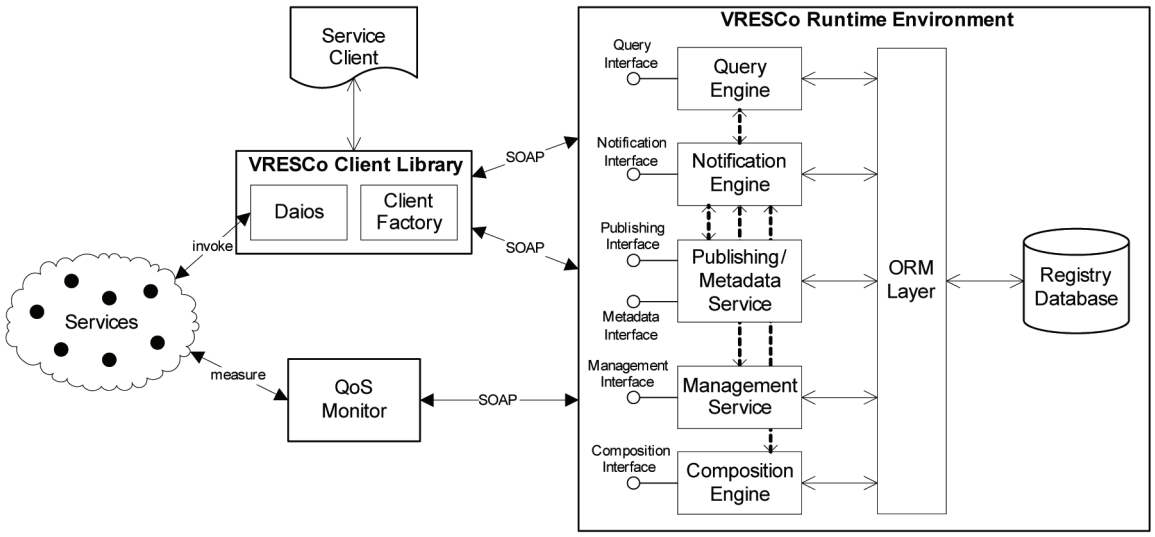
This section describes the VRESCo project[2] (Vienna Runtime Environment for Service-Oriented Computing) and its event notification support. Before we go into the details of this event notification support we give a motivating example for our work, followed by a brief introduction of the overall runtime architecture and the service metadata model of VRESCo.

Motivating Example

The case study shown in Figure 2 is adapted from (Michlmayr, Rosenberg, Platzer, Treiber, & Dustdar, 2007) and will be used for illustration purposes. In this case study, a telecommunication company (TELCO) consists of multiple departments that provide different services to different service consumers. *Inhouse services* are shared among the different departments (e.g., CRM services). *Customer services* are only used by the TELCO customers (e.g., view billing information) whereas *public services* can be accessed by everyone (e.g., get phone/roaming charges). Additionally, the TELCO consumes *partner services* (e.g., credit card service) as well as *competitor services* from other TELCOs (e.g., number porting service). Furthermore, service providers maintain multiple revisions of their services.

This case study shows several scenarios where notifications are useful. Consider for example that TELCO1 wants to get notified if new shipping services get available or if new revisions of TELCO2's number porting service are published. Furthermore, it is also important to know if services get unavailable or are removed from the registry (e.g., in order to automatically switch to another service). Besides these basic event notifications another concern for TELCO1 is to observe QoS

Figure 3. VRESCo overview



attributes. For instance, TELCO1 wants to react if the response time of a service falls beyond a given threshold. This implies that the environment considers runtime information of its services. To go one step further, TELCO1 also wants to get notified if the average response time of TELCO2's number porting service (measured within a time frame of 6 hours) falls beyond a given threshold since this might violate their Service Level Agreement (SLA).

In addition to subscribing to certain events of interest, TELCOs also want to search in the vast amount of historical events. In that way, stakeholders are enabled to observe the history of a given service or service provider within a given period of time, when deciding about the integration of external services into their own business processes.

In these scenarios notifications have clear advantages over traditional approaches using runtime exceptions, since service consumers can instantly react to failures or QoS changes. The power of events additionally opens up new perspectives and applications scenarios that can be built in a flexible manner. For instance, this includes SLAs and service pricing models as well as provenance-aware applications, which are discussed later.

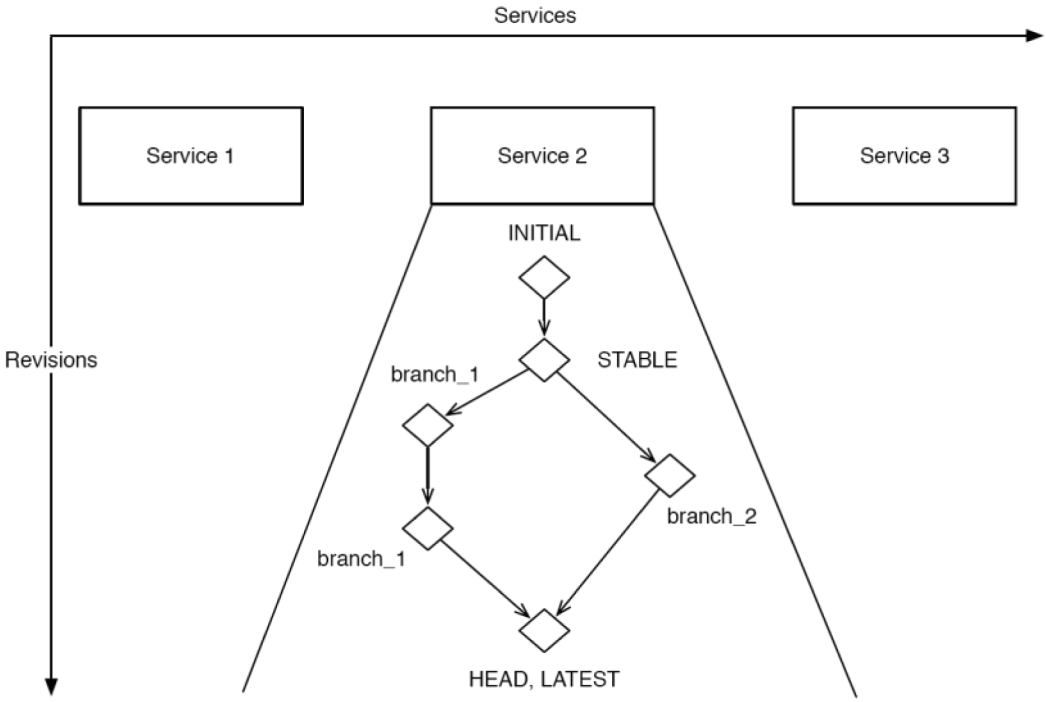
VRESCo Overview

The event notification approach presented in this chapter was implemented as part of the VRESCo runtime introduced in (Michlmayr, Rosenberg, Platzer, Treiber, & Dustdar, 2007). Before going into the details of our eventing approach, we give a short overview of this project.

The VRESCo runtime environment aims at addressing some of the current challenges in Service-oriented Computing research (Papazoglou, Traverso, Dustdar, & Leymann, 2007) and practice. Among others, this includes topics related to service discovery and metadata, dynamic binding and invocation, service versioning and QoS-aware service composition. Besides this, another goal is to facilitate engineering of service-oriented applications by reconciling some of these topics and abstracting from protocol-related issues.

The architecture of VRESCo is shown in Figure 3. To be interoperable and platform-independent, the VRESCo services which are implemented in C#/.NET are provided as Web services. These services can be accessed either directly using the SOAP protocol, or via the client library that provides a simple API. Services and associated

Figure 4. Service revision graph (Leitner, Michlmayr, Rosenberg, & Dustdar, 2008)



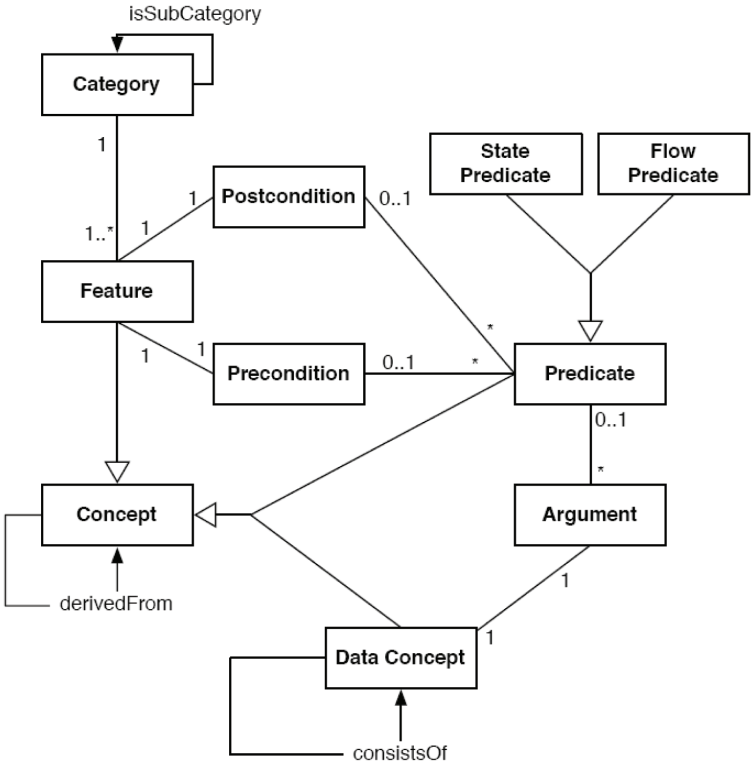
metadata are stored in the registry database that is accessed using the object-relational mapping (ORM) layer. The services are published and found in the registry using the publishing and querying engine, respectively. The VRESCo runtime uses a QoS monitor (Rosenberg, Platzer, & Dustdar, 2006) which continuously monitors the QoS, and keeps the QoS information in the registry up to date. Furthermore, the composition engine provides support for QoS-aware service composition (Rosenberg, Celikovic, Michlmayr, Leitner, & Dustdar, 2009). Finally, the event notification engine is responsible for notifying subscribers when events of interest occur.

Versioning, Dynamic Binding and Invocation

Web services evolve over time, which raises the need to maintain multiple service revisions concurrently. VRESCo supports service versioning by introducing the notion of service revision graphs (Figure 4), which define successor-predecessor relationships between different revisions of a service and support multiple parallel branches of the same service (Leitner, Michlmayr, Rosenberg, & Dustdar, 2008). Revision tags (e.g., INITIAL, STABLE, LATEST) are used to distinguish the different service revisions. Service consumers make use of versioning strategies to specify which revision of a service should be invoked (e.g., always invoke the newest revision, always invoke a specific revision, etc.).

To carry out the actual Web service invocations the Daios dynamic Web service invocation framework (Leitner, Rosenberg, & Dustdar, 2009) has been integrated into the VRESCo client library. Daios decouples clients from the services to be invoked by abstracting from service implementation issues such as encoding styles, operations or endpoints. Therefore, clients only need to know the address of the WSDL interface

Figure 5. Metadata model (Rosenberg, Leitner, Michlmayr, & Dustdar, 2008)



describing the target service, and the corresponding input message; all other details of the target service implementation are handled transparently. Besides dynamic invocation, VRESCo also supports dynamic binding of Web services. The aim is to dynamically bind to services offering the same functionality. The rebinding can either be QoS-based (using queries on QoS attributes) or content-based (using unique identifiers within different service categories). Rebinding strategies are used to define when the current binding of the service proxy should be evaluated (e.g., periodic, on demand, on invocation, etc.). We give an example for service invocations in VRESCo in Listing 1 below (see Section *Usage Examples*).

VRESCo Service Metadata Model

The VRESCo runtime provides a rich service metadata model capable of storing additional service information in the registry. This is needed to capture the purpose of services to enable querying and mediating between similar services that perform the same task. The VRESCo metadata model presented in (Rosenberg, Leitner, Michlmayr, & Dustdar, 2008) is depicted in Figure 5. The main building blocks of this model are *concepts* that represent the definition of an entity in the domain model. We distinguish between three different types of concepts:

- *Features* represent concrete actions in the domain (e.g. PortNumber).
- *Data concepts* represent concrete entities in the domain (e.g., customers) which are defined using other data concepts and atomic elements such as strings or numbers.
- *Predicates* represent domain-specific statements that either return *true* or *false*. Each predicate can have a number of *arguments*.

For example, a predicate for a *feature* PortNumber could be Portability_Status_Ok(PhoneNumber), expressing the portability status of a given phone number.

Concepts have a well-defined meaning specific to a certain domain. For example, the data Concept Customer in one domain is clearly different to the concept Customer in another. Furthermore, concepts may be derived from other concepts (e.g., PremiumCustomer is a special variant of the more general concept Customer).

Each feature in the metadata model is associated with one *category* expressing the purpose of a service (e.g., PhoneNumberPorting). Each category can have additional subcategories following the semantics of multiple inheritance to allow a more fine-grained differentiation. Features have *preconditions* and *postconditions* expressing logical statements that have to hold before and after the execution of a feature. Both types of conditions are composed of multiple predicates, each having a number of optional arguments that refer to a concept in the domain model. There are two different types of predicates: *Flow predicates* describe the data flow (i.e., the data required or produced by a feature) while *state predicates* express some global behavior that is valid either before or after invoking a feature.

Services in VRESCo can be mapped to this metadata model (e.g., services map to categories, service operations map to features, operation parameters map to data concepts, etc.). As a result, services that perform the same task but have different interfaces can be dynamically replaced at runtime. More information can be found in (Rosenberg, Leitner, Michlmayr, & Dustdar, 2008).

VRESCo Eventing Engine

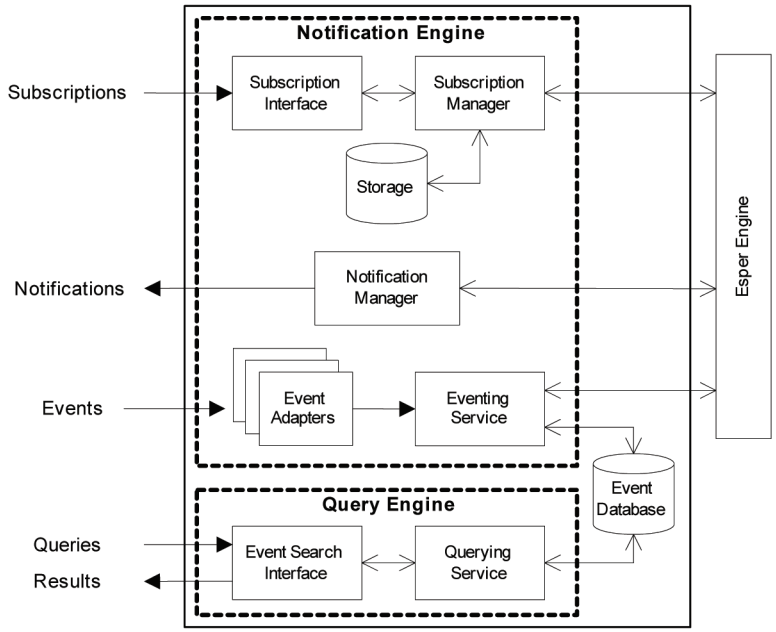
This section presents the VRESCo notification support that was introduced in (Michlmayr, Rosenberg, Leitner, & Dustdar, 2008). The basic idea can be summarized as follows: notifications are published within the runtime if certain events occur (e.g., service is added, user is deleted, etc.). In contrast to current Web service registries, this also includes events concerning service binding and invocation, changing QoS attributes, and runtime information. Service consumers are then enabled to subscribe to these events.

Figure 6 depicts the architecture of the notification engine which represents one component of the VRESCo runtime shown in Figure 3. The event processing functionality is based on NEsper, which is a.NET port of Esper. Within the notification engine, events are published using the eventing service. Most events are directly produced by the corresponding VRESCo services (e.g., service management events are fired by the publishing service while querying events are fired by the querying service). In contrast to this, events related to binding and invocation are produced by the service proxies located in the client library. Event adapters are thereby used to transform incoming events into the internal event format which can be processed efficiently. The eventing service then forwards these events to the event persistence component that is responsible for storing events in the event database. Finally, the eventing service feeds incoming events into the Esper engine.

The subscription interface is used for subscribing to events of interest according to the methods proposed in the WS-Eventing specification. The subscription manager is responsible for managing subscriptions which are put into the subscription storage. In addition, subscriptions are translated for further processing. This is done by converting the WS-Eventing subscriptions into Esper listeners which are attached to the Esper engine.

The Esper engine performs the actual event processing and is, therefore, responsible for matching incoming events received from the eventing service to listeners attached by the subscription manager. On a successful match, the registered listener informs the notification manager that is responsible for notifying interested subscribers.

Figure 6. Eventing architecture (Michlmayr, Rosenberg, Leitner, & Dustdar, 2008)



Depending on the listener type, the notification manager knows which notification type to use (e.g., email, listener Web service).

Finally, the search interface is used to search for historical events. The event database is implemented using a relational database and accessed via the ORM layer. The querying service returns a list of events that match the given query.

Event Types

The first step in developing such notification mechanism is to define all events supported by the engine. In the context of our work there are several events that can be captured at runtime. We have identified the events shown in Table 1 where events are grouped according to their event type. The event condition in the right column describes the situations when the event occurs. These event types form an event type hierarchy following the concept of class hierarchies (i.e., events inherit the properties of their parent event type) which is illustrated using colons.

The biggest group in this hierarchy is represented by the service management events that are triggered when services or service revisions and their associated metadata or QoS values change. Other event types include runtime information concerning binding and invocation, querying information and user information. All events inherit from the base type *VRESCoEvent* which provides a unique event sequence number and a timestamp measured during event publication.

Event Participants

Event-based systems usually consist of two types of participants that pose different requirements to the system, namely event producers and event consumers.

In general, events are produced by VRESCo components. However, different components are responsible for firing different kinds of events. These components, which mainly differ in their location, are described in this section. In this regard, we distinguish between *internal events* that are produced within the SOA runtime and

Table 1. VRESCo events

Event Type	Event Name	Event Condition
UserManagementEvent : **VRESCoEvent**	*UserAddedEvent*	User is added to the runtime
	UserModifiedEvent	User is modified in the runtime
	UserDeletedEvent	User is deleted from the runtime
	UserLoginEvent	User logs in using the GUI
	UserLogoutEvent	User logs out using the GUI
ServiceManagementEvent : **VRESCoEvent**	*ServicePublishedEvent*	New service is published into the runtime
	ServiceModifiedEvent	Service is updated (no new revision)
	ServiceDeletedEvent	Service is deleted from the runtime
	ServiceActivatedEvent	Service is activated in the runtime
	ServiceDeactivatedEvent	Service is deactivated in the runtime
VersioningEvent : **ServiceManagementEvent**	*RevisionPublishedEvent*	New revision is published into the runtime
	RevisionActivatedEvent	Service revision is activated in the runtime
	RevisionDeactivatedEvent	Service revision is deactivated in the runtime
	RevisionTagAddedEvent	Service revision tag is added by the owner
	RevisionTagRemovedEvent	Service revision tag is removed by the owner
MetadataEvent : **ServiceManagementEvent**	*ServiceCategoryAddedEvent*	Service category is added to the runtime
	ServiceCategoryModifiedEvent	Service category is modified in the runtime
	ServiceCategoryDeletedEvent	Service category is deleted from the runtime
	FeatureAddedEvent	Feature is added to a service category
	FeatureModifiedEvent	Feature is modified in a service category
	FeatureDeletedEvent	Feature is deleted from a service category
	MappingEvent	Service is mapped to a feature
QoSEvent : **ServiceManagementEvent**	*QoSRevisionEvent*	QoS value of service revision is published
	QoSOperationEvent	QoS value of service operation is published
	RevisionGetsUnavailableEvent	Service revision gets unavailable
	RevisionGetsAvailableEvent	Service revision gets available again
BindingInvocationEvent : **VRESCoEvent**	*ServiceInvokedEvent*	Specific service is invoked
	ServiceInvocationFailedEvent	Service invocation failed
	ProxyRebindingEvent	Service proxy is (re-)bound to a specific service
QueryingEvent : **VRESCoEvent**	*RegistryQueriedEvent*	Registry is queried using a specific query string
	ServiceFoundEvent	Specific service is found by a query
	NoServiceFoundEvent	No services are found by a query

external events that are published by components outside the runtime. Most events are directly produced by the corresponding VRESCo services. For instance, service management events (e.g., *ServicePublishedEvent*) are fired by the publishing service. The same is true for versioning and metadata events. According to this, user management events are published by the user management service while querying events are produced by the querying service. All these event types have in common that they are produced as part of the VRESCo services and therefore represent internal events.

The application logic inherent to binding and invocation of services is located in the service proxies provided by the client library. As a result, the events concerning binding and invocation (e.g., *ServiceInvokedEvent*) are fired by this component. Therefore, VRESCo provides a notification interface in order to allow clients to publish binding and invocation events into the runtime. These client events represent external events that are then transformed into the internal event format by the runtime. Finally, the QoS monitor that regularly measures the QoS values of services is responsible for firing QoS events. Similar to the client library, the QoS monitor uses the notification interface to publish external events into the runtime.

Similar to event producers, we distinguish between *internal* and *external consumers*. Internal

consumers reside within the runtime and register listeners at the Esper engine that are invoked when subscriptions match incoming events. External consumers outside the runtime are notified depending on the notification delivery mode defined in the subscription request.

In general, there are two main groups of external consumers: *humans* and *services*. Clearly, notification delivery mechanisms and the notification payload differ for these two groups. Humans are mainly interested in notifications sent per email, SMS or news feeds. In some scenarios, it might also be suitable to log the occurrence of events in log files that are regularly checked by the system administrator. In any case, notifications for humans might be less explicit since humans can interpret incomplete information. In contrast to this, service notifications can be sent using the Web service notifications standards WS-Eventing and WS-Notification. For our current prototype implementation, we have made use of the WS-Eventing specification since it represents a light-weight approach supporting content-based subscriptions.

Moreover, another distinction can be made between service providers and consumers that may be interested in different types of events. For instance, service consumers might not be interested in user management events or might not even be allowed to receive them. We introduce different event visibilities later.

Event Ranking

The importance and relevance of different events can be estimated by ranking them according to some fitness function. This is of particular interest when dealing with vast numbers of events. The following list describes several ways we have identified for ranking events:

- *Priority-based*: Event priority properties (e.g., 1 to 10 or 'high' to 'low') can be predefined according to the event model, or defined by the event producer when publishing the event. In the latter case, one problem might be that event producers do not know the importance of particular events related to others.
- *Hierarchically*: Events are ordered in a tree structure where the root represents the most important event while the leaves are less important.
- *Type-based*: All events are ranked based on the event type. That means each event has a specific type (possibly supporting type inheritance) that is used to define the ranking. However, the importance of some event might not always depend only on its type – sometimes the event properties will make the difference.
- *Content-based*: Events can be ranked based on keywords in the notification payload (e.g., the keyword 'exception' might be more important than the keyword 'warning' or 'info').
- *Probability-based*: In general, the event frequency depends on environmental factors. In this regard, one can assume that frequent events (e.g., *RegistryQueriedEvent*) might be less important than infrequent ones (e.g., *RevisionGetsUnavailableEvent*).
- *Event Patterns*: Finally, some events often occur as part of event patterns (e.g., proxy is bound to a specific service, followed by service is invoked using this proxy). The ranking mechanism could consider such event patterns.

VRESCo supports hierarchically, priority-, typed-, and content-based ranking. Probability-based ranking could be integrated by using the univariate statistic function provided by Esper. This mechanism calculates statistics over the occurrence of different events. In general, however, it should be noted that event ranking has one inherent problem: while some events can be critical for one subscriber, they might be only

Table 2. Event correlation sets

Event Correlation Set	Events	Correlation Identifier
User Management	Create, update & delete users	UserId
Service Lifecycle	Create, update, delete, bind, invoke & query services	ServiceId
Service Revision Lifecycle	Create, update, delete, bind, invoke, query & tag revisions	ServiceRevisionId
QoS	Correlate QoS measurements of one service revision	ServiceRevisionId
Service Category	Correlate events of services within one service category	ServiceCategoryId
Feature	Correlate events of services that provide one feature	FeatureId

minor for others. Yet, introducing event ranking mechanisms provides different ways to express the importance of events.

Event Correlation

Event-based systems usually deal with vast numbers of events that have to be managed accordingly. Event correlation techniques are used to avoid losing track of all events and their relationship. For instance, the work in (Rozsnyai, Vecera, Schiefer, & Schatten, 2007) describes the Event Cloud that provides different correlation mechanisms. Basically, the idea is to use event properties that have the same value as correlation identifier. For instance, two events (e.g., *ServicePublishedEvent* and *ServiceDeletedEvent*) having the same event attribute *ServiceId* are correlated since they both refer to the same service.

In the context of our work, we have identified a number of correlation sets summarized in Table 2, which shows the name of the correlation set, the events that are subsumed in this correlation, and the correlation identifier. The correlation sets cover three different aspects: user management using the *UserId* as correlation identifier, service (and service revision) lifecycle and *QoS* using *ServiceId* and *ServiceRevisionId*, and metadata information using *ServiceCategoryId* and *FeatureId*.

Besides correlating events using identifiers (e.g., the same *ServiceId*), we also consider temporal correlation of events. This is important since events that occur at the same time might be related. Furthermore, users are often interested in all events that occurred within a given timeframe. To accomplish temporal correlation of events, every event has a timestamp that is set during event publication. This timestamp can then be used to group events that happened within a given period of time (e.g., within the same hour, day, week, etc.).

The difference between event correlation sets and event types can be summarized as follows: while event types represent groups of events that occur in the same situations or indicate the same state change (e.g., some service is published), event correlation sets correlate all events that are related due to some event attribute (e.g., service revision *X* is published, deactivated, invoked, or the QoS value changes, etc.).

Subscription and Notification Mechanism

In general, event consumers can be enabled to subscribe to their events of interest in several ways (Eugster, Felber, Guerraoui, & Kermarrec, 2003). The most basic way is following the topic-based style that uses topics to classify events. Event consumers subscribe to receive notifications about that topic. Similar to topic-based subscriptions, the type-based style uses event types for classification. Even though these two styles are simple, they do not provide fine-grained control over the events of interest. Therefore, the content-based

style can be used to express subscriptions based on the actual notification payload.

Since the VRESCo runtime is provided using Web service interfaces, the subscription interface should also be using Web services. WS-Eventing represents a light-weight specification that defines such an interface by providing five operations: *Subscribe* and *Unsubscribe* are used for subscribing and unsubscribing. The *GetStatus* operation returns the current status of a subscription, while *Renew* is used to renew existing subscriptions. Each subscription has a given duration specified by the *Expires* attribute. Finally, *Subscription End* is used if an event source terminates a subscription unexpectedly.

For implementing the event processing mechanism of the VRESCo runtime, we build upon an existing WS-Eventing implementation[3] that was extended for our purpose. WS-Eventing normally uses XPath message filters as subscription language that are used for matching incoming XML messages to stored subscriptions. The specification defines an extension point to use other filter dialects which we used to introduce the *EPLDialect* for using EPL queries as subscription language. The actual EPL query is then attached to the subscription message by introducing a new message attribute *subscriptionQuery*.

WS-Eventing distinguishes between subscriber (the entity that defines a subscription) and event sink (the entity that receives the notifications) that are both implemented using Web services. VRESCo additionally supports notifications sent per email and written to log files. Therefore, in addition to the default delivery mode *PushDeliveryMode* using Web services, we introduced *EmailDeliveryMode* and *LogDeliveryMode* which are attached to the subscription messages.

The subscription process is illustrated in Figure 7. When the subscription manager receives requests from subscribers, it first extracts the subscription and puts it into the subscription storage to be able to retrieve it at a later time. Then it extracts the EPL subscription query and the delivery mode from the request and creates a corresponding Esper listener. This listener is finally attached to the Esper engine to be matched against incoming events. Furthermore, the subscription manager is responsible for keeping the subscriptions in the storage and the listeners attached to Esper synchronized. That means, when subscriptions are renewed or expire, the subscription manager re-attaches the corresponding listener or removes them, respectively.

Sending notifications can be done in several ways.

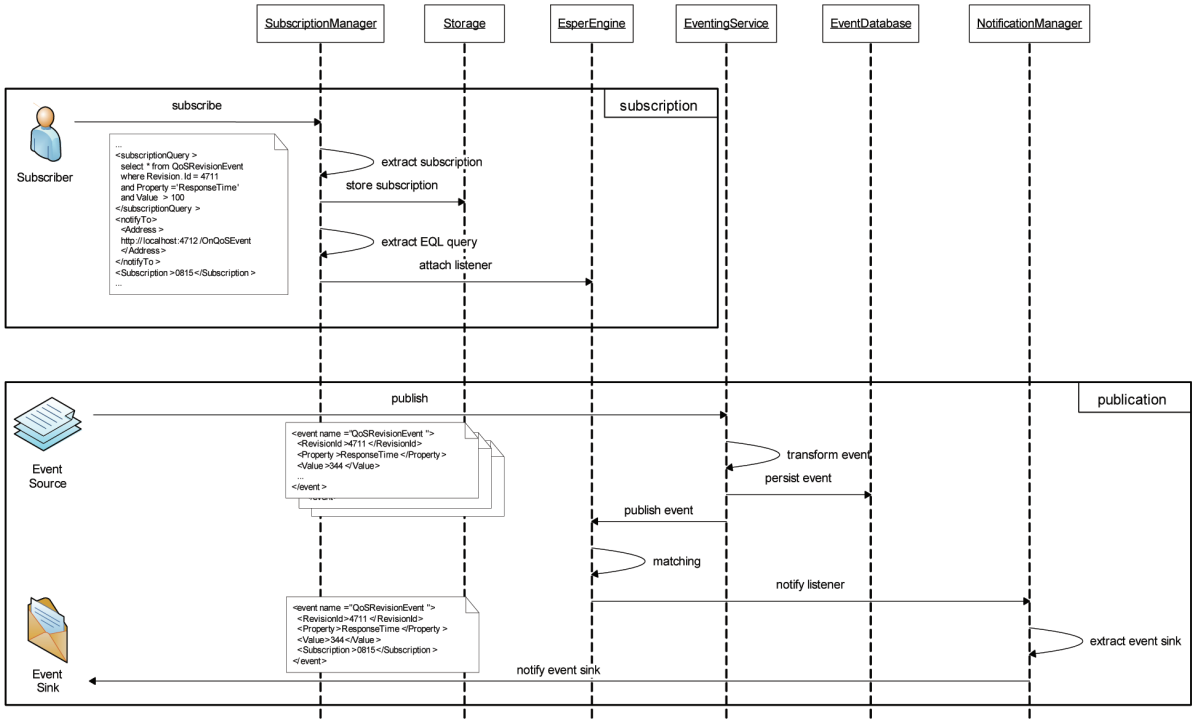
In the best-effort model, notifications are lost in case of communication errors. To prevent such loss, subscribers can send acknowledgements when receiving notifications. Besides pushing notifications towards subscribers, pull-style notifications enable subscribers to retrieve pending notifications from the event engine.

VRESCo notifications are sent push-style using emails or listener Web services. As shown in Figure 7, the notification manager knows which notification type to use depending on the listener attached to the Esper engine. On a successful match the notification manager first extracts this information from the listener. If the event sink prefers email notifications, the notification manager connects to an SMTP server. In case of Web service listeners, the notification manager invokes the corresponding listener Web service provided by the event sink. If the event sink cannot be notified, these pending notifications are stored in the event database and can be retrieved by the subscribers in pull-style.

Event Persistence and Event Search

Event notifications are often used when subscribers want to quickly react on state changes. Additionally, in many situations it is also important to search in historical event data. For instance, users might want to get notified if a new service revision is published into the registry while they

Figure 7. Subscription and event publication sequence



also want to search for the five previous service revisions.

To support such functionality, the VRESCo notification engine stores all events and provides an appropriate search interface for it. As illustrated in Figure 7, when events are published by an event source (e.g., QoS monitor), the eventing service first transform the events into the internal event format and then persists them into the event database. These events can be queried using the event search interface that is part of the querying interface which is used to search for services in the registry database. Data access in VRESCo is done via an ORM layer using NHibernate[4]. Therefore, the event search builds on the Hibernate Query Language (HQL).

Since event-based systems often deal with vast numbers of events, in some situations using relational databases might not be efficient enough. In such cases, building highly targeted and efficient index structures might be preferred. In this regard, we envision using the Vector space model in addition to a traditional relational event database. Following this model, documents (events) are represented by n-dimensional vectors where each dimension represents one keyword. The similarity of two vectors then indicates the similarity of the two corresponding documents (events) using these keywords. The advantage of the Vector space model compared to traditional database search is that the search returns a list of fuzzy matches together with a similarity rating. Furthermore, the search queries can be easily executed on multiple distributed vector spaces.

Event Visibility

In our first prototype, events were visible to all users within the runtime. However, this can be problematic in business scenarios. For instance,

considering our TELCO case study shown in Figure 2, TELCO1 might agree that PARTNER1 can see events concerning service management and versioning, but might restrict that events related to binding and invocation are only visible for its own employees.

Mühl et. al. (Mühl, Fiege, & Pietzuch, 2006) discuss security issues in event-based systems by introducing different access control techniques such as access control lists (ACL), capabilities, and role-based access control (RBAC). ACLs, on the one hand, define the permissions of different users (principals) for specific security objects. Capabilities, on the other hand, define the permissions of a specific user for different security objects. The difference is that ACLs are stored for every security object while capabilities are stored for every user. Finally, RBAC extends capabilities by allowing users to have several roles that represent abstractions between users and permissions. Users can have one or more roles while permissions are directly granted to the different roles.

In the VRESCo notification engine, we have integrated an access control mechanism following RBAC (Michlmayr, Rosenberg, Leitner, & Dustdar, 2009). Therefore, VRESCo users are divided into different user groups. Event visibility can then be defined according to the event visibilities shown in Table 3.

It is interesting to note that in our work the event publisher is enabled to define the visibility of her events. While one publisher might not want that other users can see events ("PUBLISHER"), another might not define any restrictions ("ALL"). Additionally, user access to events can be granted only to specific users (e.g., "anton"). Finally, RBAC is introduced by either defining visibility for all users of a specific group (e.g., ":admins"), or all users within the same group as the publisher ("GROUP").

Besides defining event visibilities for different users and groups, more fine-grained access control is provided by allowing users to specify event visibilities for specific event types. Clearly,

Table 3. Event visibilities

Event Visibility	Description
ALL	Events are visible to all users
GROUP	Events are visible to all users within the same group of the publisher
PUBLISHER	Events are visible to the publisher only
<:GroupName>	Events are visible to all users within a specific group
<Username>	Events are visible to a specific user only

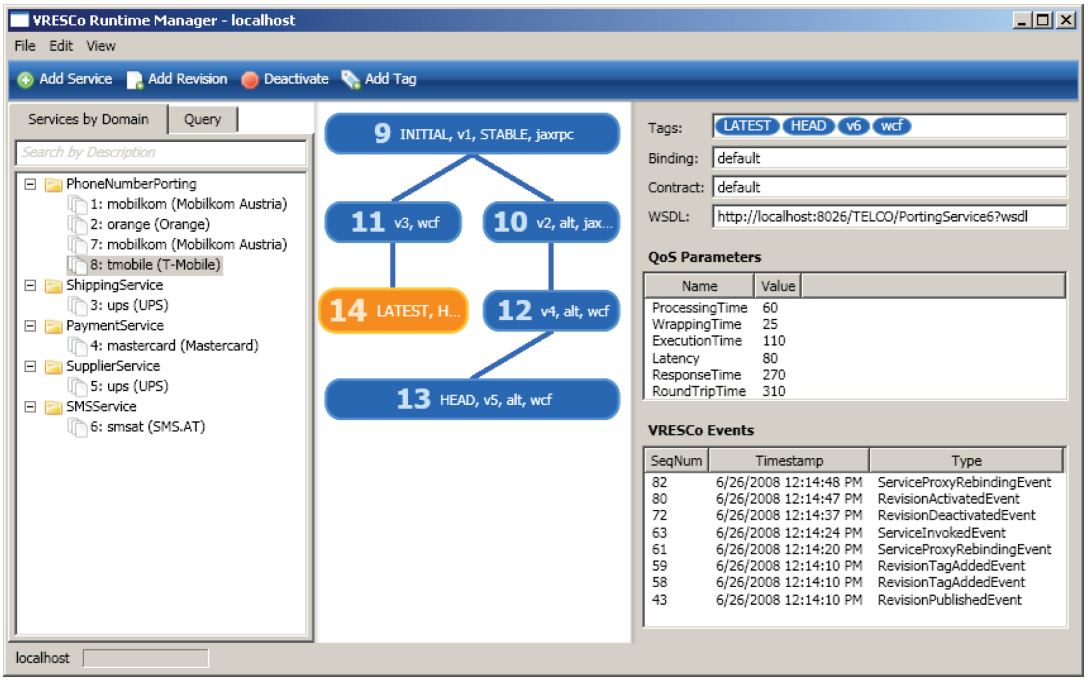
these definitions take the event type hierarchy into consideration: If no event visibility is defined for a specific event type, the engine takes the visibility of the parent type. If there is no visibility for any type the default visibility is chosen (i.e., ALL for type *VRESCoEvent*).

The access control mechanism is enforced by the eventing service and the notification manager shown in Figure 6. On the one side, the eventing service attaches both event visibility and name of the publisher to the event before feeding it into the Esper engine. While the name of the publisher can be directly extracted from the request message of the invoked VRESCo service (e.g., *QueryingService*), the event visibility of the publisher is queried from the registry database.

On the other side, when events match subscriptions the notification manager gets name and user group of the subscriber from the subscription storage and extracts publisher name and event visibility from the notification payload. Based on this information, the notification manager can verify if the current event is visible to the subscriber. If the event is visible the subscriber is notified, otherwise no notification is sent. Furthermore, the event search also follows the same principle: if events are not visible to the requester, they are removed from the search result.

In our approach, publishers are able to specify which subscribers can see which events by using event visibilities. Therefore, event access is mainly controlled by the publishers. Apart from that, however, subscribers are able to specify which

Figure 8. VRESCo runtime manager



event producers they are interested in. This is done by specifying the event attribute *publisher* in the EPL subscription queries.

VRESCo Runtime Manager

Figure 8 shows a screenshot of the VRESCo Runtime Manager GUI (displaying the implementation of the TELCO case study). The service categories and their services are listed in the left part of the GUI that also provides a search interface for querying services within the registry database. The service revision graph of the selected service is illustrated in the middle, showing identifier and tags of the different service revisions. The initial revision is always placed on the top of the graph and the edges define the predecessor-successor relationship. The details of the selected service revision are shown in the right part including revision tags, URL of the WSDL document and current QoS parameters (e.g., response time, latency, etc.). The table in the bottom right corner depicts the service revision lifecycle represented by all events related to this service revision (i.e., correlated using the same revision identifier) that are visible to the current user. The table shows sequence number, timestamp and type of the events.

Usage Examples

In this section, we use our motivating example to show how the VRESCo runtime is used to invoke services, as well as how the event notification support works in practice. The performance of the event engine and further examples illustrating the expressiveness of the subscription language can be found in (Michlmayr, 2010).

Listing 1 illustrates how service proxies are generated in VRESCo and how the Daios framework is used to invoke services. In lines 2-3, the VRESCo client factory is used to create a proxy for the querying service running on port number 8001 on localhost. Service proxies in VRESCo are defined using a search query that is constructed in lines 6-8. In this example, we want to access the latest service revision of category PhoneNum-

berPorting from TELCO2 having a response time of less than 500 milliseconds. In lines 12-13, the service proxy is created using this selection while the rebinding strategy PeriodicRebinding(10000) means that the binding should be evaluated every 10 seconds. For instance, if a new revision is published that better matches the selection, the service proxy should automatically rebind to this revision. In lines 16-22, the input message is built while the service is actually invoked using the request/response pattern in line 25. It should be noted that Daios also supports one-way and asynchronous invocation patterns. Finally, the number porting confirmation is extracted from the output message that is returned by the service (line 28).

Listing 1. Dynamic Binding and Invocation

```
01  // create proxy for the querying service
02  IVRESCoQuerier querier =
03      VRESCoClientFactory.
        CreateQuerier("localhost", 8001);
04
05  // select service and new provider
06  string selection = "Service.Category.Name
    like 'PhoneNumberPorting' and"
07      + " Service.Owner.Name like
        'TELCO2' and"
08      +"Tag.Name like 'LATEST' and QoS.
        ResponseTime < 500";
09  Provider prov = new Provider("TELCO1",
    "Main Street 1, A-1234 Vienna");
10
11  // create service proxy with periodic
    rebinding
12  DaiosProxy proxy = querier.
    CreateRebindingProxy(
13      selection, new PeriodicRebinding
        (10000));
14
15  // create input message
16  DaiosMessage inMsg=new DaiosMessage();
17  inMsg.SetString("NumberToPort", nr);
18  DaiosMessage provider = new
    DaiosMessage();
19  provider.SetString("address", prov.
    Address);
20  provider.SetString("name", prov.name);
21  provider.SetLong("id", prov.Id);
22  inMsg.SetComplex("NewProvider",
    provider);
23
24  // invoke service using request/response
    pattern
25  DaiosMessage outMsg = proxy.
    RequestResponse(inMsg);
26
27  // get response from output message
28  DaiosMessage conf = outMsg.
    getComplex("Confirmation");
```

The subscription procedure is shown in Listing 2. Again, it starts by creating a proxy for the corresponding VRESCo service; this time it is the subscription service running on localhost using port number 11111.

The code listing shows three subscription examples. The first one in lines 5-9 uses email notifications to root@localhost when the EPL query in line 6 is matched (i.e., every time a new revision for service 17 is published). The last argument represents the duration of the subscription in seconds (i.e., in this case, the subscription is valid for 30 minutes).

The second example in lines 12-18 declares interest if the availability of revision 23 is greater than 99 percent which is valid until 31.12.2009. In that case, a Web service notification should be sent to net.tcp://localhost:8006/OnVRESCoEvents. The second parameter in line 15 defines where *subscriptionEnd* messages should be sent.

The third subscription in lines 21-27 demonstrates statistical functions over event streams and the sliding window operator which are supported by Esper. In this example, the property ResponseTime of *QoSOperationEvents* regarding service operation 33 of service revision 47 is inspected within a time frame of 12 hours. If the average response time is greater than 500 milliseconds

any time before 20.09.2009 at 20:09, notifications should be sent per email.

In all three examples, the subscription identifier sid is returned by the subscription service. This identifier can be used to get the status, renew, or unsubscribe from this subscription. Furthermore, the notification payload also contains this identifier so that event consumers can correlate notifications to subscriptions.

Listing 2. Subscription Examples

```
01  IVRESCoSubscriber subscriber =
02      VRESCoClientFactory.
        CreateSubscriber("localhost", 11111);
03
04  // subscribe using email notifications
05  Identifier sid = subscriber.
    SubscribePerEmail(
06      "select * from RevisionPublishedEvent
        where Service.Id = 17",
07      "root@localhost",
08      60 * 30
09  );
10
11  // subscribe using Web service notifications
12  sid = subscriber.SubscribePerWS(
13      "select * from QoSRevisionEvent "+
14      "where Revision.Id = 23 and
        Property='Availability' and Value >
        0.99",
15      "net.tcp://localhost:8005/
        SubscriptionEndTo",
16      "net.tcp://localhost:8006/
        OnVRESCoEvents",
17      new DateTime(2009, 12, 31)
18  );
19
20  // use sliding window and statistics
21  sid = subscriber.SubscribePerEmail(
22      "select * from QoSOperationEvent"+
23      "(Revision.Id=47 and Operation.Id=33
        and Property='ResponseTime') "+
24      ".win:time(12 hours).stat:uni('Value')
        where average > 500",
25      "root@localhost",
26      new DateTime(2009, 9, 20, 20, 9, 0)
27  );
```

Finally, Listing 3 illustrates a concrete use case for the notification support demonstrating notification-based rebinding as opposed to the periodic rebinding exemplified in Listing 1. Using notifications the rebinding of service proxies can now be forced as soon as the given subscription matches (line 5). In VRESCo, event consumers have to implement the interface IEventNotification (line 1) that defines the event handler method Notify (lines 3-6). This handler method provides access to the subscription identifier and to the actual events.

Listing 3. Notification-based Rebinding

```
01  public class EventSink: IEventNotification
    {
02
03      public void Notify(VRESCoEvent[]
        newEvents, VRESCoEvent[]
        oldEvents,
04      string subscriptionId) {
05      proxy.ForceRebinding();
06      }
07
08  }
```

FUTURE TRENDS

In this chapter, we have presented the foundational work on event notification support in the VRESCo runtime environment. There are several application scenarios and research directions that are enabled by this work:

- **Provenance-aware Applications:** Provenance is an important issue that enables (especially

in service-oriented systems) assertions on who did what in applications or business processes. Based on the availability of event data, provenance information can be gathered and used to proof compliance with certain regulations (e.g., laws, standardized processes, etc.) which is addressed by (Curbera, Doganata, Martens, Mukhi, & Slominski, 2008). Complementary to this work, we have introduced the notion of service provenance which defines provenance information of services (Michlmayr, Rosenberg, Leitner, & Dustdar, 2009).

- **SLAs and Service Pricing:** Service pricing models receive increasing attention as more and more services become available. In this regard, service usage can be automatically billed to the user account according to the agreed pricing model. The pricing is also influenced by the SLA defined between the interacting partners, possibly resulting in penalties if providers cannot meet the SLAs. Using event information stored in the event database, the billing information can be easily aggregated for given time periods by issuing queries over the event database. This allows flexible derivation of pricing models based on dynamically negotiated SLAs.
- **Event-based Composition:** The aim of SOA often is to achieve higher level business goals by composing multiple services possibly considering QoS attributes (Zeng, Benatallah, Ngu, Dumas, Kalagnanam, & Chang, 2004). Ideally, this composition should be dynamic in order to allow replacing services if there are alternative services performing the same task. The metadata model described earlier allows defining the differences between similar services that can then be used to mediate between services at runtime. In this regard, events may trigger the composition process. For instance, if the response time of some service operation goes beyond a given threshold (which is highlighted by QoS events) the composition engine should restructure the composition using an alternative service providing the same operation. A complementary service composition approach using content-based publish/subscribe to automatically detect the compatibility of services is presented in (Hu, Muthusamy, Li, & Jacobsen, 2008).

CONCLUSION

In typical SOA environments, functional and non-functional properties of services change regularly. Since service providers and consumers are usually loosely coupled, the latter are not informed about such changes and may not be able to access changed services any more. Current registry standards provide basic support for event notifications when registry data changes. However, this does not include QoS attributes and runtime information concerning binding and invocation of services.

In this chapter, we have presented an event notification mechanism for service runtime environments that supports such information. Furthermore, temporal relation between events can be considered using sliding window operators and event patterns. Subscribers can be notified about events using emails or Web service notifications following WS-Eventing. Our approach was integrated into the VRESCo runtime which supports dynamic binding and invocation of services, service versioning, service metadata, and a registry database including publishing and querying services. Additionally, we have shown how the core VRESCo features and the notification support are used in practice. Finally, we have sketched different application scenarios and future research directions that are enabled by our approach.

REFERENCES

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., & Ito, K. (2008). STREAM: The Stanford Data Stream Management System. In Garofalakis, M., Gehrke, J., & Rastogi, R. (Eds.), *Data Stream Management: Processing High-Speed Data Streams*. Berlin, Germany: Springer.

Cugola, G., & di Nitto, E. (2008). On adopting Content-Based Routing in service-oriented architectures. *Information and Software Technology*, *50*(1-2), 22–35. doi:10.1016/j.infsof.2007.10.004

Cugola, G., & Picco, G. P. (2006). REDS: a reconfigurable dispatching system. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM'06)* (S. 9-16). Portland, OR: ACM Press.

Curbera, F., Doganata, Y. N., Martens, A., Mukhi, N., & Slominski, A. (2008). Business Provenance - A Technology to Increase Traceability of End-to-End Operations. In *Proceedings of the 16th International Conference on Cooperative Information Systems (CoopIS'08)* (S. 100-119). Monterrey, Mexico: Springer.

EsperTech. (2008). *Esper Reference Documentation*. Retrieved August 25, 2008, from http://esper.codehaus.org/

Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A.-M. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, *35*(2), 114–131. doi:10.1145/857076.857078

Hu, S., Muthusamy, V., Li, G., & Jacobsen, H.-A. (2008). Distributed Automatic Service Composition in Large-Scale Systems. *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)* (S. 233-244). Rome: ACM Press.

Jobst, D., & Preissler, G. (2006). Mapping clouds of SOA- and business-related events for an enterprise cockpit in a Java-based environment. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java (PPPJ'06)* (S. 230-236). Mannheim, Germany: ACM Press.

Leitner, P., Michlmayr, A., Rosenberg, F., & Dustdar, S. (2008). End-to-End Versioning Support for Web Services. In *Proceedings of the International Conference on Services Computing (SCC'08)*. Honolulu, HI: IEEE Computer Society.

Leitner, P., Rosenberg, F., & Dustdar, S. (2009). Daios - Efficient Dynamic Web Service Invocation. *IEEE Internet Computing*, *13*(3), 72–80. doi:10.1109/MIC.2009.57

Li, G., Cheung, A., Hou, S., Hu, S., Muthusamy, V., Sherafat, R., et al. (2007). Historic Data Access in Publish/Subscribe. In *Proceedings of the Inaugural International Conference on Distributed Event-Based Systems (DEBS'07)* (pp. 80-84). Toronto, Canada: ACM Press.

Luckham, D. C., & Vera, J. (1995). An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, *21*(9), 717–734. doi:10.1109/32.464548

Mahambre, S. P., Kumar, M. S., & Bellur, U. (2007). A Taxonomy of QoS-Aware, Adaptive Event-Dissemination Middleware. *IEEE Internet Computing*, *11*(4), 35–44. doi:10.1109/MIC.2007.77

Michlmayr, A. (2010). Event Processing in QoS-Aware Service Runtime Environments, PhD Thesis, Vienna University of Technology.

Michlmayr, A., Rosenberg, F., Leitner, P., & Dustdar, S. (2008). Advanced Event Processing and Notifications in Service Runtime Environments. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)* (pp. 115-125). Rome: ACM Press.

Michlmayr, A., Rosenberg, F., Leitner, P., & Dustdar, S. (2009). Service Provenance in QoS-Aware Web Service Runtimes. In *Proceedings of the 7th International Conference on Web Services (ICWS'09),* (S. 115-122). Los Angeles: IEEE Computer Society.

Michlmayr, A., Rosenberg, F., Platzer, C., Treiber, M., & Dustdar, S. (2007). Towards Recovering the Broken SOA Triangle - A Software Engineering Perspective. In *Proceedings of the Second International Workshop on Service Oriented Software Engineering (IW-SOSWE'07)* (pp. 22-28). Dubrovnik, Croatia: ACM Press.

Mühl, G., Fiege, L., & Pietzuch, P. (2006). *Distributed Event-Based Systems*. Berlin, Germany: Springer-Verlag.

OASIS International Standards Consortium. (2005a). *ebXML Registry Services and Protocols*. Retrieved August 22, 2008, from http://oasis-open.org/committees/regrep

OASIS International Standards Consortium. (2005b). *Universal Description, Discovery and Integration (UDDI)*. Retrieved August 22, 2008, from http://oasis-open.org/committees/uddi-spec/

OASIS International Standards Consortium. (2006). *Web Services Notification (WS-Notification)*. Retrieved August 22, 2008, from http://www.oasis-open.org/committees/wsn

Papazoglou, M. P., Traverso, P., Dustdar, S., & Leymann, F. (2007). Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, *40*(11), 38–45.

Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., & Dustdar, S. (2009). An End-to-End Approach for QoS-Aware Service Composition. In *Proceedings of the 13th IEEE International Enterprise Computing Conference (EDOC'09)*. Auckland, New Zealand: IEEE Computer Society.

Rosenberg, F., Leitner, P., Michlmayr, A., & Dustdar, S. (2008). Integrated Metadata Support for Web Service Runtimes. In *Proceedings of the Middleware for Web Services Workshop (MWS'08)* (S. 24-31). Munich, Germany: IEEE Computer Society.

Rosenberg, F., Platzer, C., & Dustdar, S. (2006). Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06),* (pp. 205-212). Chicago: IEEE Computer Society.

Rozsnyai, S., Vecera, R., Schiefer, J., & Schatten, A. (2007). Event Cloud - Searching for Correlated Business Events. In *Proceedings of the 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE'07)* (S. 409-420). Tokyo, Japan: IEEE Computer Society.

Sayre, R. (2005). Atom: The Standard in Syndication. *IEEE Internet Computing*, *9*(4), 71–78. doi:10.1109/MIC.2005.74

Treiber, M., & Dustdar, S. (2007). Active Web Service Registries. *IEEE Internet Computing*, *11*(5), 66–71. doi:10.1109/MIC.2007.99

Weerawarana, S., Curbera, F., Leymann, F., Storey, T., & Ferguson, D. F. (2005). *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing WS-BPEL, WS-Reliable Messaging, and More*. Upper Saddle River, NJ: Prentice Hall.

World Wide Web Consortium. (2006). *Web Services Eventing (WS-Eventing)*. Retrieved August 22, 2008, from http://www.w3.org/Submission/WS-Eventing/

Zeng, L., Benatallah, B., Ngu, A. H., Dumas, M., Kalagnanam, J., & Chang, H. (2004). QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, *30*(5), 311–327. doi:10.1109/TSE.2004.11

KEY TERMS AND DEFINITIONS

Service: Services are autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel ways. They perform functions that range from answering simple requests to executing sophisticated business processes requiring peer-to-peer relationships among multiple layers of service consumers and providers. Any piece of code and any application component deployed on a system can be reused and transformed into a network-available service.

Service Registry: Service registries provide repositories of services which contain service descriptions and additional service metadata. Services are published into registries by service providers, while service consumers query these repositories to find services of interest.

Service Provider: Services are provided and maintained by service providers which represent the owner of the service that define who is able to consume these services. Service providers may guarantee functional and non-functional Quality of Service (QoS) attributes which can be defined in Service Level Agreements (SLA).

Service Consumer: Services are invoked by service consumers in various ways. This can range from single invocations to invocations as part of a complex business processes. The technical service descriptions which are necessary to invoke services are found in service registries.

Event: Events represent situations, detectable conditions or state changes which trigger notifications (e.g., a service has changed in the registry).

Notification: Notifications are messages which are triggered by the occurrence of events. These notifications are sent to all event consumers that have previously subscribed to the corresponding events.

Subscription: Subscriptions are used to declare interest in different events. This can range from simple topic-based subscriptions where events are grouped into different topics, over type-based subscriptions where events are part of event type hierarchies, to content-based subscriptions which enable fine-grained control over event attributes.

Event Producer: Event producers (also called event sources or publishers) are those entities that detect and finally publish events.

Event Consumer: Event consumers (also called event sinks) are those entities that receive notifications when certain events of interest occur. It should be noted that subscribers (i.e., the entity that creates subscriptions) and event consumers can represent different entities.

Event Engine: The event engine (also called event-based infrastructure) is responsible for managing subscriptions and matching of incoming events to stored subscriptions. If subscriptions match incoming events, the corresponding event consumers are notified.

ENDNOTES

1 http://servicemix.apache.org
2 http://vresco.sourceforge.net
3 http://www.codeproject.com/KB/WCF/WSEventing.aspx
4 http://www.nhibernate.org