Advanced Event Processing and Notifications in Service Runtime Environments

Anton Michlmayr, Florian Rosenberg, Philipp Leitner, Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Wien, Austria
lastname@infosys.tuwien.ac.at

ABSTRACT

Service-oriented Architecture (SOA) and Web services have become widely adopted for building cross-organizational and flexible applications. Yet, there is one issue inherent to this paradigm: services are changing regularly. Using the publish/subscribe style, subscribers can be notified when such changes occur. In current service registry standards, however, notifications are mainly used to inform about changes in the registry data, which does not include service runtime information. In this paper, we present an approach that leverages event processing mechanisms for Web services based on a rich event model that supports 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.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.2.4 [**Computer-Communication Networks**]: Distributed Systems — Distributed Applications

Keywords

Service-oriented Architecture, Publish/Subscribe, Event Processing

1. INTRODUCTION

Following the Service-oriented architecture (SOA) paradigm, service providers register services and corresponding descriptions in registries. Service consumers can then find services in a registry, bind to the services which best fit their needs, and finally execute them. Web services [24] are one widely adopted realization of SOA and build upon the main standards SOAP, WSDL and UDDI. Over the years, a complete Web service stack has emerged that provides rich support for multiple higher level functionality (e.g., business process execution, transactions, metadata exchange, etc.).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS '08, July 1-4, 2008, Rome, Italy
Copyright 2008 ACM 978-1-60558-090-6/08/07 ...$5.00.

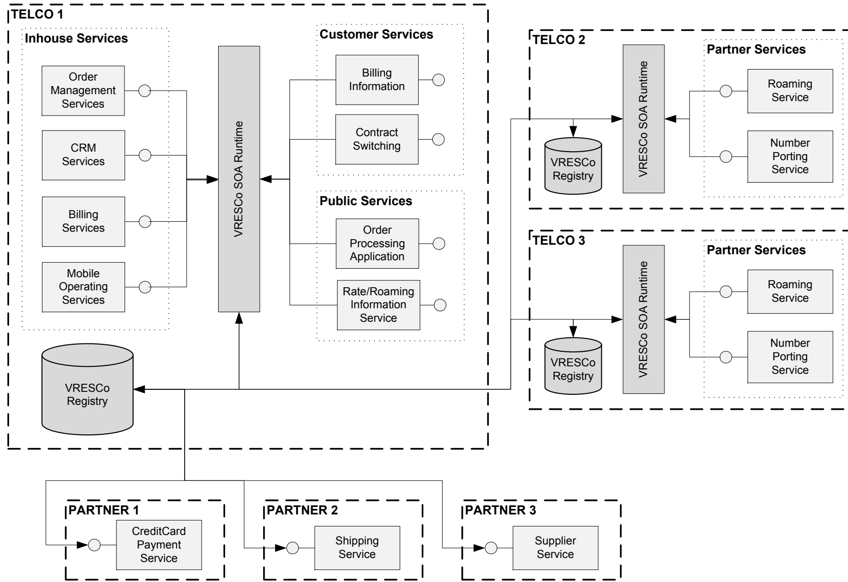
Practice, however, has revealed some problems of the SOA paradigm in general and Web services in particular. The idea of public registries where everybody can publish services did not succeed. This is highlighted by the fact that Microsoft, SAP and IBM have shut down their public registries in the end of 2006. Moreover, there are still a number of open issues in SOA research [16] and practice, such as dynamic binding and invocation, dynamic service composition, and service metadata.

One reason for these issues is represented by the fact that services and associated metadata and Quality of Service (QoS) attributes change regularly. However, service consumers are not aware of these changes. In this regard, the lack of appropriate event notification mechanisms limits flexibility because service consumers cannot automatically react to service and environment changes.

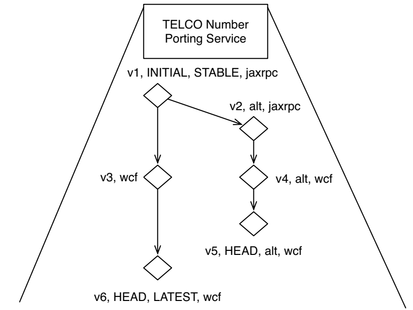
Notifying subscribers when events of interest occur represents the focus of the publish/subscribe style [4] in general, and event-based systems [12] in particular. Cugola and Di Nitto [2] give a detailed overview of approaches that combine publish/subscribe and SOA. The most popular examples are WS-Notification [15] and WS-Eventing [23]. 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.

Additionally, the service registry standards UDDI [14] and ebXML [13] introduce limited support for event notifications. Both standards have in common that users are enabled to track 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.

We argue that receiving notifications about such runtime information is equally important 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 paper, we focus on such runtime event notification support. Our contribution is threefold: firstly, we introduce event notification support in SOA runtime environments, including event types, participants, representations, as well as ranking, correlation, subscription, and notification mechanisms. Secondly, we show how this was integrated into the VRESCo runtime [6, 11]. Finally, we present a case study which is used for evaluation, and point to further application scenarios which are enabled by such runtime notification support.



(a) TELCO Environment



(b) Number Porting Service Revisions

Figure 1: TELCO Case Study

The remainder of this paper is organized as follows. Section 2 presents a motivating example for this work. Section 3 introduces the VRESCo SOA runtime and gives an architectural overview of the event notification engine. Section 4 introduces the necessary background of this work while Section 5 presents in detail how event notifications are supported in VRESCo. Section 6 gives an evaluation of our work, while Section 7 presents related work in this area. Finally, Section 8 concludes the paper.

2. MOTIVATING EXAMPLE

The case study shown in Figure 1(a) is borrowed from [11] 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., roaming service). Furthermore, service providers maintain multiple revisions of their services. Figure 1(b) illustrates the revision graph of TELCO2's number porting service. In this figure, TELCO2 provides two branches of this service which are built on different Web service platforms (e.g., JAX-RPC) and may provide different interfaces.

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 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 the 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 build in a flexible manner. For instance, this includes SLAs and service pricing models as well as provenance-aware applications, which are discussed in Section 6.

3. VRESCO APPROACH

The event notification approach presented in this paper was implemented as part of the VRESCo (Vienna Runtime Environment for Service-Oriented Computing) project. Before going into the details of our eventing approach, we give a short overview of this project.

3.1 Introduction

The VRESCo runtime environment introduced in [11] aims at addressing some of the current challenges in Service-oriented Computing research [16] 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, an-

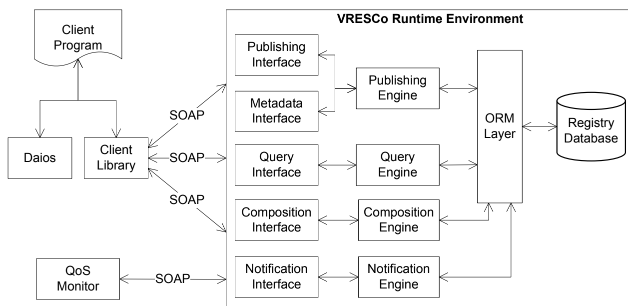


Figure 2: VRESCo Overview

other goal is to facilitate engineering of service-oriented applications by reconciling some of these topics and abstracting from protocol-related issues. The basic architecture of VRESCo is shown in Figure 2. To be interoperable and platform-independent, the VRESCo services are provided as Web services which can be accessed either directly using the SOAP protocol, or via the client library that provides a simple API for accessing these services.

Services and associated 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 as described in [18], which continuously monitors the QoS values of services, and keeps the QoS information in the registry up to date. Furthermore, the composition engine aims at providing support for QoS-aware service composition which is part of our ongoing work. The event notification engine which is the focus of this paper is responsible for notifying subscribers when certain events of interest occur.

To carry out the actual Web service invocations the Daios dynamic Web service invocation framework [7] has been integrated into the VRESCo client-side 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 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 [11]. The aim is to dynamically bind to services offering the same functionality. This is done using so called rebinding strategies which are integrated into the service proxies. The rebinding can either be QoS-based (using queries on the measures QoS attributes) or content-based (using unique identifiers within different service categories).

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 (see Figure 1(b) for an example) that define successor-predecessor relationships between different revisions of a service and support multiple branches [6]. Revision tags are used to distinguish the different service revisions. Service consumers are then enabled to decide which service to use. For instance, they can always invoke specific revisions of a service, or even switch between service revisions at runtime (e.g., always invoke the newest revision).

3.2 Eventing Architecture

This section gives a high-level overview of the VRESCo notification support, while the details are described in Section 5. 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 get notified about the occurrence of these events.

Figure 3 depicts the architecture of the notification engine which represents one component of the VRESCo runtime shown in Figure 2 and is, therefore, also implemented in C# on the .NET platform. The event processing functionality is based on NEsper, which is a port of the event processing engine Esper[1]. 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. This is done using the ORM layer. Finally, the eventing service feeds incoming events into the Esper engine.

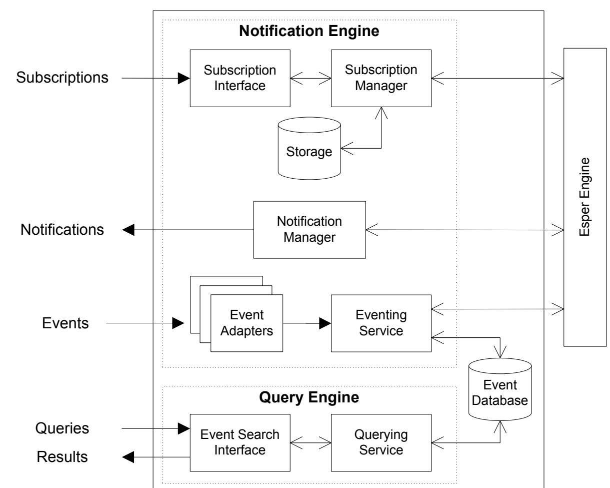


Figure 3: VRESCo Eventing Architecture

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

[1] `http://esper.codehaus.org`

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. 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.

4. ESPER

The VRESCo notification support is based on the open source event processing engine Esper. To understand how Esper is integrated into our runtime and how events, subscriptions and notifications are handled in VRESCo, we briefly introduce Esper before going into more detail.

The Esper engine 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]*) whereas *mapped* properties represent keyed collections of values (e.g., *user['firstname']*). Finally, *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 which is invoked when the subscription matches incoming events. The queries use the Esper Query Language (EQL) which is similar to the Structured Query Language (SQL). The main difference is that EQL is formulated on event streams whereas SQL uses database tables. The `select` clause specifies the event properties to retrieve, the `from` clause defines the event streams to use, and the `where` clause specifies 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.

EQL provides a powerful mechanism to integrate temporal relations of events using *sliding event windows*. These operators allow to 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 of a given service during the last 6 hours. Finally, EQL supports subqueries, output frequency, and event patterns. The latter allows to define relations between subsequent events (e.g., `->` representing a 'followed by' relation). Section 6 gives some examples for EQL queries. More information on Esper can be found in [3].

5. EVENTS IN VRESCO

After this brief introduction of the VRESCo project and the Esper engine, this section presents in detail how events are supported in our SOA runtime. This includes event types, participants, representation, ranking, and correlation, as well as subscription and notification mechanisms.

5.1 Event Types

The first step in developing such event notification mechanism is to define all event types which are supported by the engine. In the context of our work there are several events which can be captured at runtime. We have identified the basic event types shown in Figure 4.

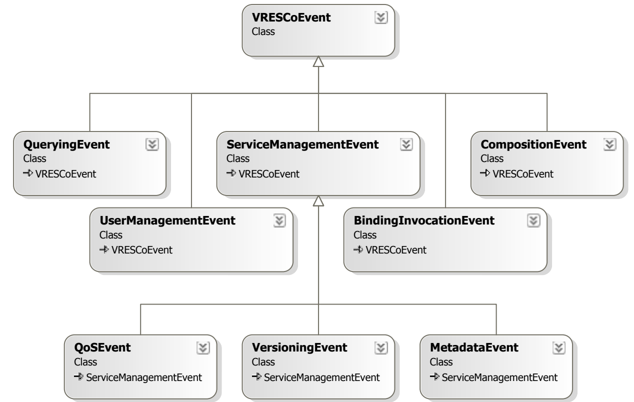


Figure 4: VRESCo Event Type Hierarchy

The event types form an event type hierarchy following the concepts of class hierarchies (i.e., events inherit the properties of their parent event type). The biggest group in this tree is represented by the service management events which 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. As shown in Figure 4, all events inherit from the base type *VRESCo-Event* which provides a unique event sequence number and a timestamp measured during event publication.

Table 1 gives a detailed description of all events considered by the VRESCo notification engine, where the events are grouped according to their event type. The event condition in the right column describes the situations when the event occurs. Besides composition events, all events are fully implemented in VRESCo. The composition engine which will also make use of event notifications is part of our ongoing work and, therefore, not addressed in this paper.

The current prototype provides a simple service metadata model consisting of categories of services that perform the same task, but we are currently working on an enhanced metadata model that supports richer semantic descriptions of services (e.g., including pre- and post-conditions). The metadata events will then be adapted to reflect this model.

5.2 Event Participants

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

Table 1: **VRESCo Events**

Event Type	Event Name	Event Condition
UserManagementEvent	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	ServicePublishedEvent	New service is published into the runtime
	ServiceModifiedEvent	Service is updated in the runtime (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	RevisionPublishedEvent	New service 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	ServiceCategoryAddedEvent	Service category is added to the runtime
	ServiceCategoryModifiedEvent	Service category is modified in the runtime
	ServiceCategoryDeletedEvent	Service category is deleted from the runtime
QoSEvent	QoSEvent	Current QoS value of some service revision is published
	RevisionGetsUnavailableEvent	Service revision gets unavailable
	RevisionGetsAvailableEvent	Service revision gets available again
BindingInvocationEvent	ServiceInvokedEvent	Specific service is invoked
	ServiceInvocationFailedEvent	Service invocation failed
	ProxyRebindingEvent	Service proxy is (re-)bound to a specific service
QueryingEvent	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

Event Producers

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* which are produced within the SOA runtime and *external events* which are published from 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 servixe 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 feed binding and invocation events into the runtime. These client events represent external events which are then transformed into the internal event format by the runtime.

Finally, the QoS monitor which 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 feed external events into the runtime.

Event Consumers

Similar to event producers, we distinguish between *internal* and *external consumers*. Internal consumers reside within the runtime and register listeners at the Esper engine which 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 other technolgies such as news feeds (e.g., RSS [20], Atom [21]). In some scenarios, it might also be suitable to log the occurrence of events in log files which 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, services notification can be sent using the Web service notifications standards WS-Eventing and WS-Notification. For our current prototype implementation, we have enhanced the WS-Eventing specification since it represents a light-weight approach supporting content-based subscriptions. The integration of WS-Eventing will be described later.

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

Table 2: VRESCo 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 all QoS measurements of one service revision	ServiceRevisionId
Service Category	Correlate all events of services within one service category	ServiceCategoryId

5.3 Event Representation

The notification payload differs for humans and services. While services need exact information about the event type or the context in which an event occurred, the notification payload for humans might be less verbose. In addition, notifications for humans do not necessarily have to adhere to standardized formats or rules.

To guarantee efficient processing of a huge number of events, VRESCo uses its own event format implemented as hierarchy of C# classes instead of using XML events. Clearly, external events and notifications to Web service listeners are sent using XML. The event classes consist of simple name-value pairs which are often used in event-based systems since they support efficient content-based filtering of events. According to the type-based approach the events classes are part of an event hierarchy as introduced above. Furthermore, in addition to simple name-value pairs the event classes may also include non-primitive data types.

5.4 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 pre-defined 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 other events.

- *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) which 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., if the payload contains the keyword 'exception' it be more important than events with keyword 'warning' or 'info').

- *Probability-based*: In general, the frequency of different events 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 univariant statistic function provided by Esper. This mechanism calculates statistics over the occurrence of different events (see Section 6 for an example).

In general, however, it should be noted that event ranking has one inherent problem: while one specific event can be critical for one subscriber, it is only minor for others (e.g., QoS value changes, old service revision is deleted, etc.). Yet, introducing event ranking mechanisms provides different ways to express the importance of events.

5.5 Event Correlation

Event-based systems usually deal with vast numbers of events which have to be managed accordingly. Event correlation techniques are used to avoid losing track of all events and their relationship. For instance, Rozsnyai et al. [19] describe the *Event Cloud* that provides different correlation mechanisms. Basically, the idea is to use event properties which 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 which 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 service category information using the *ServiceCategoryId*.

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.). The correlation sets enable users to track all important events which are related.

5.6 Subscription Mechanisms

In general, event consumers can be enabled to subscribe to their events of interest in several ways [4]. The most basic way is following the topic-based style which uses topics to classify events. Event consumers subscribe to receive noti-

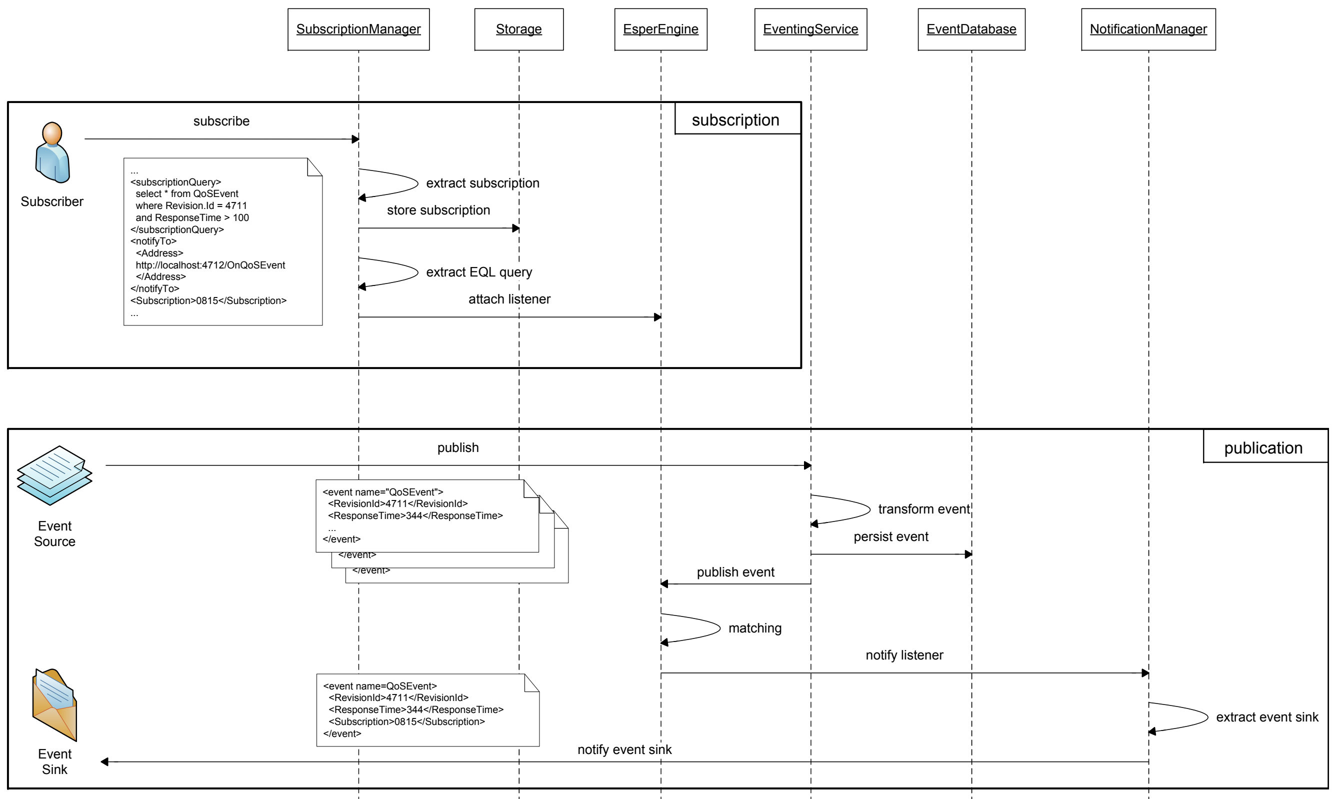


Figure 5: VRESCo Subscription and Event Publication

fications 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 [23] 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[2] which was extended for our purpose. WS-Eventing normally uses XPath message filters as subscription language which 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 introduced the *EQLDialect* for using EQL queries as subscription language. The actual EQL query is then attached to the subscription message by introducing a new message attribute *subscriptionQuery*.

[2] `http://www.codeproject.com/KB/WCF/WSEventing.aspx`

WS-Eventing distinguishes between subscriber (the entity that defines a subscription) and event sink (the entity that receives the notifications) which are both implemented using Web services. VRESCo 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 5. When the subscription manager receives subscription requests from subscribers, it first extracts the subscription and puts it in the subscription storage to be able to retrieve it at a later time. Then it extracts the EQL 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.

5.7 Notification Mechanisms

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 might send acknowledgements on receiving notifications. Besides pushing notifications towards the interested subscribers, pull-style notifications enables subscribers to retrieve pending notifications from the event engine.

VRESCo notifications are sent push-style using email or listener Web services defined in the subscription. As shown in Figure 5, 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 notifies 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.

5.8 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 also want to search for the five previous service revisions.

To support such functionality, the VRESCo notification engine persists all events in an event database and provides an appropriate search interface for it. As illustrated in Figure 5, 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 the events into the event database. The events can be queried by using the event search interface which is part of the querying interface used to search for services in registry database. Data access in VRESCo is done via an ORM layer using NHibernate[3]. 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 plan to use the Vector space engine described in [17] in addition to a traditional relational event database. Following the Vector space 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.

6. EVALUATION

In this section we evaluate the VRESCo event notification support threefold. Firstly, we show the expressiveness of the subscription language by using scenarios from the motivating example in Section 2. Secondly, we present performance results of the VRESCo event engine, and thirdly, we discuss further application scenarios enabled by this work.

6.1 Subscription Expressiveness

Considering our TELCO case study, assume the system administrator of TELCO1 wants to get notified as soon as some of their service revisions get deactivated. This can be easily expressed using the following subscription.

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

[3]http://www.nhibernate.org

Another example is to notify about new services. Consider that a service consumer wants to get notified if a new revision of service 11 is published. This can be written as

```
select * from RevisionPublishedEvent
where Service.Id = 11
```

The first two example are intentionally basic. Besides the fact that UDDI does not provide versioning support, these examples could also be implemented using existing web service registry standards.

Furthermore, the VRESCo runtime also considers QoS attributes of services which are measured by the QoS monitor. Although not natively supported by UDDI, this could be implemented by storing QoS attributes in corresponding *tModels* of the UDDI registry as for example illustrated in [25]. To give a concrete example, assume a service consumer wants to get notified, as soon as the response time of service revision 17 is greater than 500 milliseconds which is expressed by the following subscription.

```
select * from QoSEvent
where Revision.Id = 17
and QoS.ResponseTime > 500
```

The notification features of current Web service registry standards mainly provide support for subscribing to static registry data. The VRESCo eventing engine goes one step further and also includes runtime information such as binding and invocation of services. In that way, service providers are enable to get notified if some service has been invoked. Furthermore, subscribers are interested in events within a given period of time which is supported by the sliding window operator. For instance, getting univariate statistics (e.g., sum, average, variance, etc.) of property $Timestamp$ of the last ten subsequent $ServiceInvokedEvents$ concerning service 9 can be expressed as easily as follows.

```
select * from ServiceInvokedEvent(Service.Id=9)
.win:time(10).stat:uni('Timestamp')
```

Similar to this, the sliding window can also be defined on the actual time when the events occur. Additionally, the *where* clause can be used to express constraints on the statistical function. For example, the following subscription fires if the average $ResponseTime$ of $QoSEvents$ concerning service revision 47 that occurred within the last six hours is greater than 350 milliseconds.

```
select * from QoSEvent(Revision.Id=47)
win:time(6 hours).stat:uni('QoS.ResponseTime')
where average > 350
```

Finally, event patterns may be considered which enables subscribers to define temporal relations between different events. For instance, the following subscription fires, if a new service revision is invoked within ten days after its publication. `A -> B` means that event A happens before event B, the *every* operator defines which events trigger the pattern to be fired.

```
select * from pattern
[every publish=RevisionPublishedEvent
  -> every invoke=ServiceInvokedEvent
    (publish.Revision.Id=invoke.Revision.Id)
where timer:within(10 days)]
```

To summarize, the subscription language used in our approach enables to define complex subscriptions using event patterns, sliding window operators, and statistical functions on event streams, which cannot be defined using traditional Web service registry notification mechanisms.

6.2 Performance Results

The Esper documentation [3] gives an overview of the engine's performance characteristics. According to that information, Esper exceeds 500.000 events per second and exhibits linear scalability. This section presents initial performance results of the VRESCo event engine which were measured by publishing QoS events and varying the number of interested subscribers to show the scalability of our approach. The simulation was executed on a standard laptop with 2Ghz Core 2 Duo CPU, 2GB memory and a 5400rpm hard disk using .NET 3.5 on Windows XP and self-hosting WCF services. The results represent the average throughput taken from 10 subsequent runs of 15 seconds each.

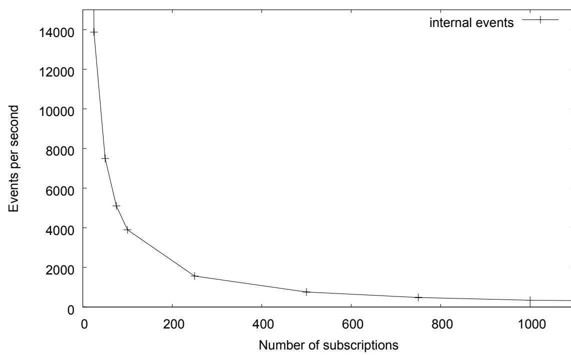


Figure 6: Internal Throughput

Figure 6 depicts the throughput of internals events depending on the number of subscriptions. The graphs illustrates that the throughput is high for a small number of matching subscriptions (around 150.000 events without subscribers – not shown in the figure) and decreases quickly with the number of subscriptions. Figure 7 shows that providing event persistence by storing events into the event database significantly reduces the throughput to hundreds of events per second, which mainly is a result of using relational databases instead of more efficient indexes. We plan to alleviate this performance decrease by implementing a scheduler that saves events in the database in batch mode.

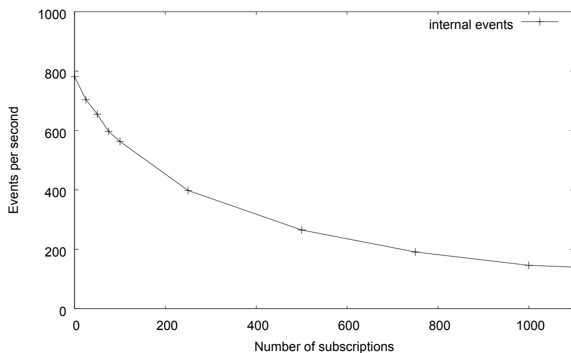


Figure 7: Internal Throughput with Persistence

The performance of the internal events is adequate for our system since the typical setting consists of a small to medium number of Web services which minimizes the number of produced events. Furthermore, most events are triggered by invocations of the VRESCo Web service operations (e.g., Publishing Service, User Management Service, etc.) and, therefore, the maximum throughput of internal events is usually not reached in practical deployments.

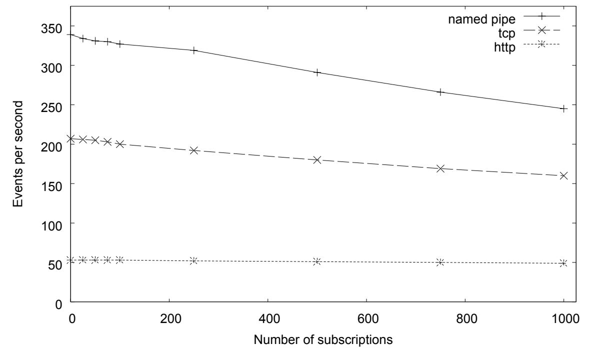


Figure 8: External Throughput

Figure 8 depicts the throughput of external events (e.g, events fired by the QoS monitor) that are published using the VRESCo notification interface. Since the throughput using the HTTP binding is only around 50 events per second, we used additional bindings such as TCP and named pipes which are also illustrated in this figure. Figure 9 demonstrates the throughput decrease when persisting the events into the database.

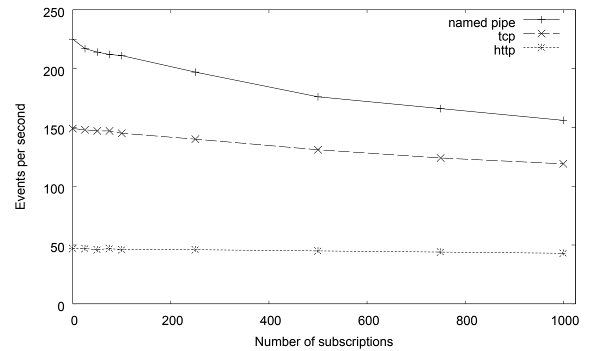


Figure 9: External Throughput with Persistence

Both figures depict that the throughput using HTTP is nearly constant, which demonstrates the limit of the underlying Web service platform. The performance of external events (e.g., *QoSEvents*) is also acceptable for VRESCo because these events are less frequent than internal events. Nevertheless, we are working on a more efficient external notification interface, for instance, by writing a custom binding based on TCP.

6.3 Application Scenarios

Based on the foundational work presented in this paper there are number of application scenarios that can be built in a flexible manner when enabling events in SOA environments. We present two such scenarios below.

- *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 that is defined between the 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 store. This allows flexible derivation of pricing models based on dynamically negotiated SLAs.

- *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 (possibly including human interaction). 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.).

The main benefit of using events to develop such scenarios is the flexibility that is gained by retrieving the desired information from the event database. It does not require building new components into the existing infrastructure because the required information already exists in form of events which can be accessed by leveraging the querying service to get the relevant information. We plan to implement some of these scenarios based on the work presented in this paper.

7. RELATED WORK

Event-based systems [12] in general, and the publish/-subscribe pattern [4] in particular have been the focus of research within the last years. This research has led to different event-based architecture definition languages (e.g., Rapide [9]) and QoS-aware event dissemination middleware prototypes [10]. Moreover, data and event stream processing has also been addressed which lead to different prototypes (e.g., STREAM [1], Esper [3]).

Approaches to integrate publish/subscribe and the SOA model led to WS-Notification [15] and WS-Eventing [23]. While WS-Eventing uses content-based publish/subscribe, WS-Notification provides topics (WS-Topics) as a means to classify events. The combination of SOA and event-driven architectures is further addressed by Enterprise Service Bus (ESB) implementations (e.g., Apache Servicemix[4]). 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 [2] 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, and provides notifications following WS-Notification. Service discovery is implemented according to the query-advertise style using UDDI inquiry messages. The aim of this work is to use CBR to perform service discovery, while we focus on event processing and notifications in service runtime environments, and, besides service discovery, also provide support for dynamic binding and invocation, and QoS attributes.

Service registries (e.g., UDDI [14], ebXML [13]) 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 registry approaches, such as AWSR [22] and XMethods[5], use news feeds (e.g., RSS and Atom) for dissemination of changing 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 [5, 8, 19]. Rozsnyai et al. [19] introduce the *Event Cloud* system which aims at searching for business events. Their approach uses indexing and correlation of events by using different ranking algorithms. The implementation uses the open source text search engine Apache Lucene[6]. 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. [8] 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 [5] 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 a BPEL process. In contrast to our work, the focus is on search and visualization of business events whereas subscribing to events is not addressed. Furthermore, the different SOA events and how they are handled is not described in detail.

8. CONCLUSION

Since services change regularly, service consumers want to get notified about such changes of interest. Event notifications in service registries have been addressed in both UDDI and ebXML. However, both approaches only consider changes in registry data and do not include runtime information concerning binding and invocation of services. In this paper, we presented an approach for event notifications in service runtime environments that is capable of including such runtime information, together with QoS attributes. Furthermore, temporal relation between events can be considered by using the sliding window operator and event patterns. Our approach was integrated into the VRESCo runtime that supports dynamic binding and invocation of ser-

[4]`http://servicemix.apache.org/`

[5]`http://www.xmethods.net`

[6]`http://lucene.apache.org/`

vices, service versioning, and provides a registry database including publishing and querying services.

The case study used for evaluation shows the expressiveness of the subscription language. Furthermore, the performance results indicate that the event notification support can deal with the expected number of events and subscribers. Finally, we sketched two application scenarios that can be built based on the approach introduced in this paper.

There are some issues which are not addressed by our current approach. First of all, we plan to integrate different event visibilities to define which event consumers are allowed to receive which events. Furthermore, future work includes dynamic monitoring of SLAs based on service runtime events, and other application scenarios as described above. Finally, data mining methods may be used to retrieve further information from the event database (e.g., service lifecycle, user behavior, event patterns).

9. ACKNOWLEDGEMENTS

We would like to thank Ivona Brandic, for her many helpful comments, and her insightful perusal of our first draft.

10. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. In M. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2008.

[2] G. Cugola and E. Di Nitto. On adopting content-based routing in service-oriented architectures. *Information and Software Technology*, 50(1–2):22–35, Jan. 2008.

[3] EsperTech. *Esper Reference Documentation*, 2008. `http://esper.codehaus.org/`.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Survey*, 35(2):114–131, 2003.

[5] D. Jobst and G. Preissler. 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)*, pages 230–236. ACM, 2006.

[6] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. End-to-End Versioning Support for Web Services. In *Proceedings of the International Conference on Services Computing (SCC 2008)*. IEEE Computer Society, July 2008.

[7] P. Leitner, F. Rosenberg, and S. Dustdar. DAIOS - Efficient Dynamic Web Service Invocation. Technical Report TUV-1841-2007-01, Vienna University of Technology, 2007. `http://www.vitalab.tuwien.ac.at/~florian/papers/TUV-1841-2007-01.pdf`.

[8] G. Li, A. Cheung, S. Hou, S. Hu, V. Muthusamy, R. Sherafat, A. Wun, H.-A. Jacobsen, and S. Manovski. Historic Data Access in Publish/Subscribe. In *Proceedings of the Inaugural International Conference on Distributed Event-Based Systems (DEBS'07)*, pages 80–84. ACM, 2007.

[9] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[10] S. P. Mahambre, M. K. S.D, and U. Bellur. A Taxonomy of QoS-Aware, Adaptive Event-Dissemination Middleware. *IEEE Internet Computing*, 11(4):35–44, 2007.

[11] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar. Towards Recovering the Broken SOA Trianlge – A Software Engineering Perspective. In *Proceedings of the Second International Workshop on Service Oriented Software Engineering (IW-SOSWE'07)*, pages 22–28, Sept. 2007.

[12] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[13] OASIS International Standards Consortium. *ebXML Registry Services and Protocols*, 2005. `http://oasis-open.org/committees/regrep`.

[14] OASIS International Standards Consortium. *Universal Description, Discovery and Integration (UDDI)*, 2005. `http://oasis-open.org/committees/uddi-spec/`.

[15] OASIS International Standards Consortium. *Web Services Notification (WS-Notification)*, 2006. `http://oasis-open.org/committees/wsn/`.

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

[17] C. Platzer and S. Dustdar. A Vector Space Search Engine for Web Services. In *Proceedings of the 3rd European IEEE Conference on Web Services (ECOWS'05)*, 2005.

[18] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, Sept. 2006.

[19] S. Rozsnyai, R. Vecera, J. Schiefer, and A. Schatten. 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 2007)*, pages 409–420. IEEE Computer Society, 2007.

[20] RSS Advisory Board. *Really Simple Syndication (RSS)*, 2007. `http://www.rssboard.org/rss-specification`.

[21] R. Sayre. Atom: The Standard in Syndication. *IEEE Internet Computing*, 9(4):71–78, 2005.

[22] M. Treiber and S. Dustdar. Active Web Service Registries. *IEEE Internet Computing*, 11(5):66–71, 2007.

[23] W3C. *Web Services Eventing (WS-Eventing)*, 2006. `http://www.w3.org/Submission/WS-Eventing/`.

[24] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[25] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.