A Notification Broker for Service-Oriented Grid Environments

Timo Elverkemper
ens04ter@cs.umu.se

July 17, 2008
Master's Thesis in Computing Science, 15 ECTS credits
Supervisor at CS-UmU: P-O Östberg
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Web Services are used as the technological basis for Service-Oriented Architectures, an architectural paradigm that defines distributed systems as a composition of communicating, loosely coupled services. This architecture is widely adapted for Grids, large scale distributed and heterogeneous systems concerned with resource sharing and utilization, often in order to solve complex computational problems. The communication of Web Services commonly uses a request-response message exchange pattern. As an alternative, a publish-subscribe model allows Web Services to notify subscribed entities of certain events. This basic notification mechanism does not scale well in the context of large distributed systems however. To overcome the problem of scalability, publishers may delegate the task of distributing notifications and managing subscriptions to a notification broker service that acts as a mediator between publishers and subscribers.

This thesis presents the Notification Service, an implementation of a notification broker that is based on the Globus Toolkit, a software toolkit for building Grid applications, and its implementation of the WS-Notification specification. Additional to Web Service notifications, propagation of notifications through additional protocols such as email and instant messaging is supported. The Notification Service is designed to facilitate a loose coupling between services and thus to integrate well with Service-Oriented Architectures and Grid environments.

Contents

List of Figures

Chapter 1

Introduction

Large scale scientific computations are computationally demanding and therefore require large amounts of resources. Grids are a form of distributed systems concerned with resource sharing at a large scale, where resources can be distributed geographically across several sites and participating institutions. Because of the possible heterogeneity of systems that constitute a Grid, Service-Oriented Architectures and Web Service technologies are a common technological and architectural basis used to facilitate the interoperability of different systems. Within a Service-Oriented Architecture, systems are composed of loosely coupled communicating services that use message-based protocols for communication. In this context, the Globus Toolkit provides a Web Services-based middleware and components that are commonly required in Grids to support the construction of service-oriented Grid systems and applications.

Commonly, a request-response pattern is used to interact with Web Services. A client requests an operation to be performed and the Web Service replies with the results. Another interaction pattern allows Web Services to notify clients of events. This is accomplished by a publish-subscribe mechanism analogous to the Observer design pattern used in object-oriented applications. Clients subscribe to events that are published by a Web Service and are thereafter notified of events that occur. This basic form of Web Service notification is defined by the WS-Notification specification and implemented by the Globus Toolkit. Scalability is however a problem of this basic mechanism in the context of large distributed systems with numerous subscribing clients that need to be served by notification producing services. To enhance scalability and to decouple subscribers from notification producers, a notification broker can be used as a mediator between subscribers and producers. Such a notification broker takes on the responsibility of managing subscriptions and distributing notifications on behalf of producing services, thus reducing their workload.

In general, notifications may contain arbitrary data. But even though Web Services are concerned with machine-to-machine interactions, some notifications may even be of direct interest to a human end user, e.g. a critical system failure or the completion of a computation. The Web Service notification mechanism has however no possibility to send messages by using other protocols.

The present thesis work introduces the Notification Service, an implementation of a notification broker based on the Globus Toolkit and the basic Web Service notification mechanism described in the WS-Notification specification. Along with the handling and propagation of Web Service notifications, the Notification Service is capable of for-

warding notifications as emails or instant messages to human end users. Furthermore, the Notification Service might act as a filter to discard notifications that are uninteresting to subscribers. Facilitating dynamic discovery of services in a Service-Oriented Architecture, the Notification Service integrates with the Globus Information Services which allow possible subscribers to dynamically discover Notification Services according to desired service parameters.

The remainder of the present thesis report is structured as follows. Chapter 2 gives a problem description and defines the goals of this thesis project. An in-depth study of Web Services, their basic technologies and Service-Oriented Architectures along with details on the components of the Globus Toolkit is given in Chapter 3. Details of the implemented Notification Service are given in Chapter 4 and Chapter 5 concludes the present thesis with a short evaluation of the work.

Chapter 2

Problem Description

This chapter gives a more detailed description of the purpose and goals of the present work. The first part introduces the environment and preconditions for the project while the second part discusses the problems that are to be solved by the implementation. Finally, the third part specifies the goals of the present work.

2.1 Environment and Preconditions

Web Services provide an interface that consists of a set of operations which are made available remotely to Web Service consumers. Consumers which can be client programs or other Web Services communicate with Web Services by exchanging messages. The communication is normally initiated by Web Service consumers requesting an operation, but Web Services may also initiate a message exchange in order to notify consumers of events that occurred. Web Services are able to publish events according to a specific topic to which clients may subscribe. This publish-subscribe mechanism, also known as the object-oriented Observer design pattern, allows consumers to be notified of events that occur within Web Services. Therefore, consumers do not need to poll Web Services frequently in order to monitor if a certain event occurred. Web Services will instead notify consumers.

This publish-subscribe mechanism in the context of Web Services is defined by the WS-Notification specification. A partial implementation of this specification is available as a part of the Globus Toolkit, a software toolkit for building Grid applications that is largely based on Web Services. The Globus Toolkit serves as the basis for the implementation of the present project.

2.2 Problem Statement

The basic publish-subscribe mechanism defined by the WS-BaseNotification specification introduces additional workload to Web Services that produce notifications. Services are responsible for managing subscriptions, producing notification messages and distributing these messages to subscribed consumers. In the context of Grids, a possibly very large number of consumers might subscribe to a single notification producer and reduce the producer's ability to efficiently perform its actual tasks such as performing a calculation. The basic publish-subscribe mechanism is therefore not scalable.

Web Services generally send all notifications for a particular topic to a client that has subscribed to that particular topic. The consumer might however only be interested in a small subset of the received notifications. Because of this, Web Service consumers have to implement more fine-grained notification filtering mechanisms themselves. The unwanted notification messages introduce unnecessary resource consumption for both network transport and message processing at notification consumers and producers.

Web Services can only distribute notifications as messages to other Web Services that use the same notification mechanisms which are defined by the WS-Notification specification. In particular, it is not possible to distribute notifications by other methods than Web Service message exchange.

2.3 Goals

Based on the preconditions and the identified problems, the goal of the present thesis project is to develop a separate Web Service responsible for handling notifications. This Notification Service acts as a mediator or notification broker between Web Services that produce and consume notifications. Notification producers are able to delegate some of the work for managing subscriptions and serving consumers to the Notification Service. This reduces the workload on the Web Services that originally produce the notifications since consumers subscribe to the Notification Service directly instead of subscribing to the producing Web Services.

Consumers are able to specify to the Notification Service which of the notifications for a particular topic are interesting to them. According to this, the Notification Service will then perform filtering and forward only notifications that are interesting to the consumer.

The Notification Service can also handle subscriptions of consumers that cannot receive notifications as messages sent by Web Services. Notifications may therefore be propagated through other communication channels such as email or instant messaging, so that it becomes possible to subscribe an email address to a certain type of event.

In order to suit a Service-Oriented Architecture, the Notification Service should be as generic as possible to accommodate different Web Services that produce notifications. Because of this, the Notification Service minimizes the knowledge of different notification producers in order to facilitate a loose coupling of services. The Notification Service uses standard notification mechanisms as defined by the WS-Notification specification and is therefore easily integrated into existing architectures.

Chapter 3

Technological and Architectural Background

3.1 Introduction

The present chapter introduces the basic concepts and technologies that serve as the basis for development of the Notification Service which concerns the practical part of the present thesis project. Since this software is targeted towards Grid computing environments, the chapter starts with a general introduction of Grid computing. Subsequent sections describe Web Services, the underlying technologies and the connected architectural paradigm of Service-Oriented Architectures. Finally, the chapter introduces the Globus Toolkit, a service-oriented middleware toolkit for Grid computing environments that is used to implement the Notification Service.

3.2 Grid Computing

A commonly used real world analogy to a Grid is the electrical power grid [1], which provides access to electricity on demand through wall sockets. For end users, the details on where and how the electricity is generated is of no importance. In a similar way, the ultimate goal of Grid computing is to provide access to computational resources on demand, without requiring detailed knowledge of how and where these resources can be utilized.

There are several definitions that define the notion of a Grid. The specific problem that the Grid addresses has been defined as "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations" [6]. This definition identifies resource sharing as a central aspect in Grid computing, while a resource can denote a wide variety of entities, e.g. computational capacity, storage, software programs or at a larger scale even clusters of computers. The Grid allows the sharing of these resources among virtual organizations which are composed of individuals, organizations and other resources in order to solve complex problems. The constituent resources of a Grid are possibly geographically distributed and are commonly under the control of different institutions. This lack of centralized control consequently introduces challenges of managing authentication, authorization, resource access and resource discovery in distributed systems.

Due to the wide variety of heterogeneous resources, interoperability among participants of the Grid is required to ensure that resources can be utilized among participants. Because of this, the concept of Service-Oriented Architectures and Web Services as described in the subsequent sections is a commonly used paradigm.

3.3 Web Services

The World Wide Web Consortium (W3C) defines a Web Service in its definition of the Web Service Architecture [12] as follows:

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Although this definition makes concrete references to underlying technologies that are most commonly used for the implementation of Web Services, those technologies are not necessarily the only options for implementing Web Services.

In contrast to applications that are designed for interaction with human end users, such as standard desktop applications, Web Services are explicitly designed for machine-to-machine interaction. End users do not interact with Web Services directly, while of course applications can interact with a Web Service on behalf of end users.

Web Services expose only their interfaces to the public. Such an interface can be completely described using a Web Service Description Language (WSDL, section 3.3.2) document that characterizes the Web Service interface in terms of operations that the Web Service provides, messages that are exchanged to do so and data types that are used to construct those messages. Since the WSDL document is written in XML, it is easily processable by software. In fact, it is a common approach to automatically generate client classes for interacting with a Web Service from its WSDL document.

An important characteristic of Web Services is that implementation details of the systems are hidden behind the interface. No internal details of Web Services are exposed and these details are of no importance to a possible consumer of a Web Service. The consumer is aware that a certain functionality is provided, but the internals on how this is done are abstracted behind the Web Service interface. Consequently, this involves that the actual functionality could be implemented in an arbitrary programming language and running on a arbitrary platform.

This platform independence is possible since the actual communication mechanisms are based on widely-adopted open standards that are commonly available among numerous platforms and programming languages. Especially XML is of importance in this context since it provides the basis for both WSDL and SOAP (section 3.3.1). While WSDL describes the Web Service interface in an XML document, SOAP is the message-based application-level protocol that is used for interaction between Web Services and consumers including the exchange of data and the invocation of operations.

The following sections cover the basic technologies used in conjunction with Web Services in greater detail.

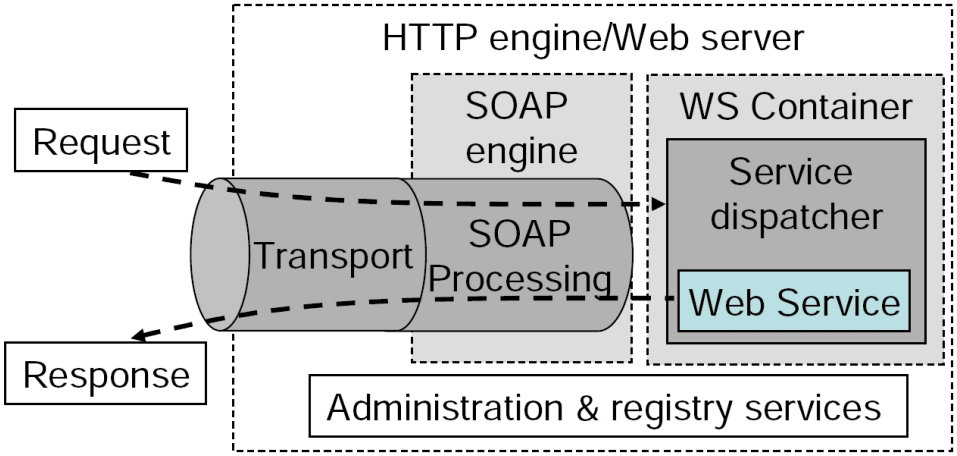

Figure 3.1: Processing of SOAP Messages [4]

3.3.1 SOAP

SOAP [13] is an application-level protocol based on XML used for data exchange and remote procedure call in distributed applications, usually for accessing Web Services. Formerly, the protocol was named "Simple Object Access Protocol". Due to its XML-based design, SOAP is platform and programming language independent. SOAP messages are transmitted embedded into or on top of other application-level protocols such as HTTP, SMTP or JMS.

3.3.1.1 SOAP Processing Model

The SOAP specification [13] defines that SOAP messages originate at a sending SOAP node and arrive at the ultimate receiving SOAP node. On the message path between the sender and the ultimate receiver, SOAP messages may pass through an arbitrary number of intermediary SOAP nodes. Each SOAP node is uniquely identified by a "Uniform Resource Identifier (URI)".

The processing of SOAP messages at a SOAP node and the invocation of Web Services is commonly accomplished by the architecture depicted in figure 3.1.

A hosting environment such as a HTTP web server provides the basic transport functionality for SOAP messages. Deployed into the web server, a SOAP engine is responsible for processing SOAP messages and invoking the service dispatcher which finally invokes the appropriate Web Service.

3.3.1.2 SOAP Messages

The XML messages exchanged with SOAP consist of the following parts.

Envelope. The SOAP envelope is the "outermost syntactic construct" [13] of a message. Looking at the XML representation, the SOAP envelope is the root element of the XML-document that defines the SOAP message. All other parts of the message are enclosed within the SOAP envelope.

Block. Information carried in the SOAP header and body is structured into units of information called SOAP blocks. Concerning the XML representation, these blocks are the immediate children of the SOAP header or body element.

Header. The SOAP header is an optional part of SOAP messages and may be used to add metadata to the actual data that is contained within the SOAP message body. It consists of a collection of zero or more SOAP blocks which may be addressed to a specific SOAP intermediary within the message path.

The specification of SOAP does not define any of the contents that might be carried in the SOAP header. This makes the SOAP header a flexible mechanism for extending SOAP messages in a modular way. Common usage scenarios of the SOAP header include authentication information, transaction management or encryption information.

Each SOAP header block might carry one of the following attributes that control the way in which the SOAP header is process by either an intermediary SOAP node or the ultimate receiver of the SOAP message.

- `mustUnderstand` - The value "true" denotes that the SOAP node must fully understand and process this SOAP block or fail and not process it at all.
- `relay` - This attribute governs if the header block is to be passed on if it has not been processed by an intermediary.
- `role` - This attribute targets a header block to a specific intermediary in the message path.
- `encondingStyle` - This attributes defines encoding rules used for serialization.

Body. The SOAP body contains the actual information of the message that is intended for the ultimate receiver of the SOAP message. It contains the actual data that denotes the request to the Web Service or the response from the Web Service to the consumer.

Structurally, the SOAP body may consist of an arbitrary number of SOAP blocks. The SOAP specification does not define how the information is to be carried in the SOAP body. However, the specification defines that the SOAP body must be present in every SOAP message. The SOAP body may also contain error information encapsulated as SOAP faults.

Faults. A SOAP fault is a part of the SOAP message that is carried inside the SOAP body element and encapsulates error information. The SOAP body must not include any additional elements when a SOAP fault is included, otherwise the message is not recognized as being an error message. SOAP faults are structured into five sub-elements:

- `Code` - An error code as predefined by the SOAP specification.
- `Reason` - A Human-readable error description.
- `Node` - Specifies the SOAP node in the message path where the error occurred.
- `Role` - Specifies the role of the SOAP node that caused the error, either the ultimate receiver of the message or an intermediary SOAP node.

– `Detail` - Contains application specific error information, for example a Java stack-trace.

Faults are similar to exceptions that are used for error handling in many object-oriented programming languages such as Java and C++, but are not tied to a specific programming language. Possible faults that a Web Service can communicate to clients in case of an occurring error are defined in the description of the Web Service's interface. Faults are therefore a platform-neutral mechanism for communicating errors to clients.

3.3.1.3 SOAP Binding

The usage of SOAP in conjunction with another protocol for transporting the actual messages is referred to as binding. The SOAP specification defines only the binding for the HTTP protocol. For other protocols, the specification defines the "SOAP Protocol Binding Framework" which allows the construction of additional bindings by conforming to general rules of the framework.

As mentioned earlier, the SOAP envelope is the root element of the XML markup that defines a SOAP message. Because of this, a SOAP binding has at least the responsibility to specify how the SOAP envelope is transported using the underlying protocol.

When using SOAP over HTTP, the SOAP message is simply transported in the body of a HTTP request or response while the standard HTTP header is used.

3.3.2 WSDL

The Web Service Description Language (WSDL) provides the possibility to completely describe a Web Service interface through the use of an XML document that conforms to an XML Schema as defined by the WSDL specification [15] of the W3C.

Since Web Services are primarily intended for machine-to-machine interaction, WSDL provides machine-processable information on how to interact with a given Web Service to a Web Service consumer application. Thus, Web Services are also referred to as "self describing software elements" [11]. Since the Web Service is fully described by the WSDL document, it is possible to generate client code for interaction with a given Web Service by using the definitions given in the WSDL document.

A WSDL document is composed of several parts which are now described in more detail. The description will cover the latest specification available at the time of writing, WSDL 2.0 [15].

3.3.2.1 Abstract Part

The abstract components of a WSDL document define the Web Service interface in a deployment-independent way in terms of data types, operations and messages that are used to interact with the Web Service and are composed types, interfaces, operations, and faults.

Types. The type component defines the message and fault message types that are used when interacting with the Web Service. The types and data structures are commonly defined using an XML Schema. The schema definition can be embedded into the types component or imported from an external XML Schema file.

The operations of the interface component as described in the next paragraph refer to these message types, which in turn define how the data contained in the messages is to be structured for interaction with the Web Service.

Interface. The interface component of the WSDL document defines a set of operations that the Web Service exposes to possible Web Service consumers. These definitions are made independently from any underlying transport mechanism.

WSDL supports inheritance of interface definitions. An interface can therefore extend another interface and inherits all operations of that particular interface along with any directly or indirectly inherited operations.

Operations. An operation is an interaction between a Web Service consumer and the Web Service and therefore includes a set of messages that are exchanged between the Web Service and the Web Service consumer. Theses messages can be both regular and fault messages.

The WSDL specification includes a set of Message Exchange Patterns that define the sequencing and cardinality of messages sent during an operation.

Faults. A fault is an error event that may occur during the execution of an operation. This condition leads to the termination of the normal message exchange between Web Services and consumers. Fault messages are usually used to communicate error information and may include more detailed information.

Faults are defined directly inside the interface component, which allows several operations to reuse the fault definition. Operation definitions can refer to defined faults through the unique name of the interface component and the fault itself.

3.3.2.2 Concrete Part

The concrete components of the WSDL document defines how and where Web Service consumers can access a Web Service by means of transport protocols, message format and endpoint addresses.

Binding. The interface component defines the messages used for interacting with Web Services at an abstract level. It describes messages used and data contained in these messages.

The purpose of the binding component on the other hand is to define how these previously defined messages are exchanged between Web Services and consumers. Therefore the binding component associates the abstract interface definition to a concrete message format and transport protocol. These implementation details are necessary in order to access a Web Service and are used to define an endpoint in the WSDL service component at which a Web Service is physically available.

Since there exist several possibilities of how messages can be formatted and transmitted, the core WSDL specification [13] does not define any bindings. There exists however a related W3C specification [14] that defines SOAP and HTTP bindings.

Service. The service component of a WSDL document defines a set of concrete endpoints at which a Web Service interface is provided. Such an interface is given through the abstract interface definition of the interface component inside the WSDL document, while a single service component refers to exactly one abstract interface definition.

Endpoint. An endpoint defines the address at which a service is available. This address is given through a Uniform Resource Identifier (URI). The endpoint definition

refers to a concrete binding definition given in the WSDL document which defines the message format and transmission protocol.

3.3.3 UDDI

The Universal Description Discovery & Integration specification [8] defines a Web Service registry that allows possible Web Service consumers to dynamically discover Web Services that provide a certain service. While WSDL describes the Web Service interface, UDDI allows the discovery of the Web Service interface by clients.

The UDDI registry is actually a Web Service itself and makes use of WSDL to describe its interface. The UDDI specification includes a set of WSDL documents that completely describe the interface of the UDDI registry and thereby defines how clients can access the registry for inquiry and publication. The WSDL documents also include a binding that specifies how to interact with the UDDI registry using SOAP over HTTP.

As mentioned, the main purpose of the UDDI registry is to allow client applications to dynamically discover Web Services that provide a required service. As an example, an e-commerce application might require to authorize a credit card payment. Instead of tying the application to a single credit card authorization service, UDDI allows the application to discover several services that provide the desired functionality and choose the one that suits best, possibly in favor of the best response time or according to some other criteria. This loose coupling also provides advantages in case of service failure. The use of UDDI allows the client applications to discover a replacement for failed Web Services.

3.3.3.1 Data Structure

UDDI has been designed as a registry for business services which is reflected by its three level data structure:

1. At the top level the registry describes businesses, organizations and other entities providing Web Services. An entry at this level is referred to as a Business Entity by the UDDI specification. Commonly, the set of entries at this level is referred to as White Pages although the specification does not use this term.

   The data included in a Business Entity is mainly textual information describing the business or organization in more detail. A categorization of the business according to geographical region or provided services is also possible.

2. The data at the second level is referred to as a Business Service and is a logical group of Web Services which are provided by a single Business Entity. These logical groups are used to categorize Web Services in business terms according to the provided services. This categorization is also referred to as Yellow Pages.

3. The two previous levels in the UDDI data structure do not contain any technical information about Web Services. At the third level, the UDDI registry contains the technical information that is needed for consumers to interact with Web Services. This data could for example be a URL of the Web Service's WSDL document that specifies its interface. The UDDI specification refers to such an entry as a Binding Template, while the registry at this level is also referred to as Green Pages.

It is to note that the UDDI registry does not contain the WSDL documents of the Web Services itself, but only contains references to locations of these documents.

3.3.3.2 Programming APIs

The UDDI specification defines a set of APIs that govern how clients can interact with the registry. Each API groups a set of operations according to their purposes. The description of the API is given in great detail by the specification which also defines all operations and the used data structures in a corresponding XML Schema and WSDL document. Because of this, only a brief overview of the APIs and their purpose will be given. More detailed information is provided by the specification document [8].

Inquiry API. Operations for for locating and retrieving information on the registry's entries is defined by the Inquiry API, which describes three possible forms of querying:

Browse Pattern. This pattern describes a broad large-scale search over the whole registry. Such a search can for example involve searching all businesses located within a particular geographic region.

Drill-Down Pattern. After a broad search according to the browsing pattern, more detailed information can be retrieved using a drill-down operation. Each entry in the registry's core data structures is uniquely identified by a key value. The browse operations list entries along with their respective key values. Such a key value can then be used by a drill-down operation to retrieve the complete information for that particular entry.

Invocation Pattern. In contrast to the two previous patterns that describe methods of searching data within the UDDI registry, the invocation pattern describes how client applications retrieve the technical information and start interacting with discovered Web Services.

The Binding Template contains the technical information that defines the interaction with a given Web Service. Commonly, this information is given in form of a reference to a WSDL document.

Client applications start by retrieving this technical information and caches it for further interaction with the Web Service. In case the interaction fails at some point, the client application refreshes the information from the UDDI registry and retries to interact with the Web Service. Because of this, it is easy to replace faulty Web Services and redirect consumers to new Web Services by changing the information in the UDDI registry.

Publication API. The Publication API specifies the interface for publishing, updating and deleting information in the UDDI registry. Each entry published in the registry is uniquely identified by a key value. Such a key can either be assigned by the UDDI registry itself or proposed by the client that submits information to the registry. The rules how keys are assigned to registry entries are given in detail in the specification [8].

Optional APIs. Additional to the two previously described main APIs of UDDI, Publication and Inquiry, the specification outlines four additional APIs which concern security mechanisms, ownership of UDDI entries, subscriptions to changes in the registry, and validation.

3.4 Service-Oriented Architecture

After discussing Web Services as components that provide a certain capability in the previous section, we now introduce the concept of Service-Oriented Architectures which defines an architectural style for building distributed systems.

The "Reference Model for Service-Oriented Architecture" [9] which has been published by the Oasis organization defines the term in a very general way:

"Service-Oriented Architecture" (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.

More specifically, SOA is an architectural paradigm for designing and building distributed system on the basis of loosely coupled, interacting services. Since SOA is concerned with the architecture of a distributed system, it is neither tied to nor targeted towards a specific problem domain.

3.4.1 Services

The central concept in a Service-Oriented Architecture is the service, which defines a mechanism that enables access to capabilities through a predefined interface [9]. Services in the context of Service-Oriented Architectures share a set of common characteristics [12]:

3.4.1.1 Logical View

A service is an abstracted, logical view of an entity that provides arbitrary functionality to service consumers. This includes for example actual programs, databases, or at a higher level even business processes. The focus lies on the functionality that services provide, not on how their operations are implemented.

3.4.1.2 Message Orientation

The service is formally defined by the messages that are exchanged during interactions with service consumers. Internal properties of services or service consumers are not exposed.

3.4.1.3 Description Orientation

A service is described by machine-processable metadata that fully describes the service's interface. The description only specifies what is required to interact with the service and does not describe any internals of the service.

3.4.1.4 Granularity

Services generally provide few coarse-grained operations with rather complex messages. An interface that is composed of few operations facilitates loose coupling because a fine grained interface with numerous operations requires more knowledge to use the service. Since services are commonly invoked remotely, an interaction involving many small messages leads also to an increased overhead for network communication and message processing.

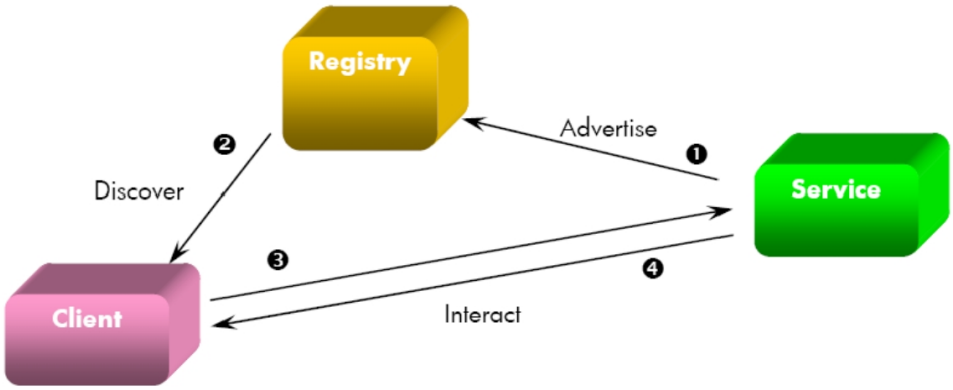

Figure 3.2: The Publish, Find, Bind Pattern

3.4.1.5 Network Orientation

Services commonly provide their capabilities to remote service consumers over a network.

3.4.1.6 Platform Independence

Messages exchanged during interactions between services and consumers are sent in a platform neutral and standardized format, usually XML. This results in interoperability between different platforms.

Although the term service and its characteristics might indicate a strong connection of Web Services as described in section 3.3 and SOA, this is not necessarily the case. Service-Oriented Architectures can be built using Web Services and it is currently a common method to do so, but SOAs do not rely explicitly on Web Services. Other options are possible in this context and are likely to appear in the future.

3.4.2 Loose Coupling

Another important concept in the context of Service-Oriented Architectures is loose coupling of services. Generally, this involves services having only minimal knowledge of each other built into their implementations.

Services are discovered dynamically when needed, rather than being hardcoded into service consumers. Dynamic discovery of services is realized through use of the so-called "publish, find, bind" pattern as shown in figure 3.2:

- **Publish.** A service publishes its presence along with its capabilities into a service registry.
- **Find.** A service consumers finds the service by querying the service registry for a certain capability that is required.
- **Bind.** Since the service consumer is now aware of the service's presence, it can interact with the service through the use of its service description which fully describes the service's interface.

This mechanism makes it easy to dynamically add, remove, replace or relocate services as needed without further modification of service consumers. For example adding extra services for load balancing or replacing faulty services are scenarios where this can be of importance.

Another aspect of loose coupling is the use of open standards for protocols, message formats and service descriptions. Widely available and open standards allow for the interoperability of different applications since the standards are available for a variety of hardware- and software platforms and programming languages. This concerns of course only the service interface and the network communication between services, how the service is implemented is of no importance since it is not exposed beyond the service interface.

As an example, an architecture including two services interacting through Java RMI is not easily integrated with another system that runs on Microsoft's .NET platform since proprietary protocols and data exchange formats are used. On the other hand, if services were all based on open standards that are available on both platforms, integrating these service would become a much easier task.

3.4.3 Benefits of Using Service-Oriented Architectures

A Service-Oriented Architecture is especially useful in the context of large-scale distributed systems in highly heterogeneous environments where parts of the system are built on different platforms.

Without using standard interfaces and open protocols, the system components would have to interact with each other through proprietary pairwise interfaces which leads effectively to a tight coupling between components. Growth of a tightly coupled system is hard to manage since adding new components might require the introduction of new proprietary interfaces to a number of already existing systems. This problem of system integration can be significantly simplified if components of the system are structured as loosely coupled services that communicate with each other through open protocols and standard interfaces that are generally available on different platforms.

In general, a Service-Oriented Architecture is much more agile and responsive to changes in the systems architecture than a system that consists of tightly coupled components.

Especially in the context of Grid computing, where resource sharing between arbitrary participants in a large distributed and heterogeneous environment is a major concern, the use of Service-Oriented Architectures with open standards and loosely coupled services is an important technique to facilitate interoperability between participants.

3.5 Globus Toolkit

The development of distributed computing applications and Grid infrastructures often involves solving a set of common problems. Naturally, each application has different requirements that are unique to a particular problem domain, but the distributed nature of those applications results in a number of common requirements among them.

The Globus Toolkit is a middleware toolkit that provides tools and components to ease the development of Grid applications. Basic issues that are addresses by the Globus Toolkit include resource management, resource discovery, and security. The included components are structured as Web Services and outline a Service-Oriented Architecture of communicating services. This facilitates the development of service-oriented applications by enabling developers to build applications by adding services that utilize other services of the toolkit. Development tools for building new Web Services in different programming languages are also included.

The subsequent sections describe the architecture and different components that constitute the Globus Toolkit.

3.5.1 Architecture

The architecture of the Globus Toolkit consists of three major parts that involve both the server and the client side of an application.

3.5.1.1 Service Implementations

The Globus Toolkit includes a set of infrastructure services that provide functionality common to distributed computing applications such as resource management, resource discovery, security, and data movement. Most of the services are written in Java and exposed as Web Services to facilitate a Service-Oriented Architecture.

3.5.1.2 Containers

Custom Web Services that are to be used with the Globus Toolkit run within a Web Service container. Such a container implements SOAP over HTTP as the message transport protocol and a number of other Web Service specifications such as the Web Service Resource Framework (WSRF) and Web Service notifications (WS-Notification). Both specifications are described in more detail in section 3.5.3 and 3.5.4 respectively. Containers are available for services written in Java, C or Python. The Java container utilizes the Apache Axis SOAP engine [10] and also allows to host Web Services that are part of the Globus Toolkit and written in Java. Each container has a registry interface that allows clients to determine which services are running inside a specific container.

3.5.1.3 Client APIs and Tools

Services that are part of the Globus Toolkit include client APIs in different languages that allow a simplified interaction with the provided services. Additionally, command line programs for interaction with some of the provided services are included.

3.5.2 Components

Numerous components are included in the Globus Toolkit in order to facilitate the development of distributed computing applications. This section introduces these components

and briefly describes their functionality.

3.5.2.1 Security

The Grid Security Infrastructure (GSI) of the Globus Toolkit is the basic security infrastructure that is used for Web Service and pre-Web Service authentication and authorization features. GSI is largely based on standard public key cryptography.

Each user and resource on the Grid is assumed to have a certificate that is encoded using the X.509 certificate format. Such a certificate binds a name that identifies the owner to a public key. It is necessary that a trusted certificate authority (CA) digitally signs the certificate in order to establish that the public key really belongs to that particular owner. Using such certificates, two entities that trust the same CA may establish each others identity using a method known as mutual authentication.

Another important concept in the context of GSI are Proxy Certificates [2] that enable single-sign on and credential delegation on the Grid. An entity that has established its identity through a certificate creates a proxy by generating a new private key and certificate. Instead of having a CA sign the newly generated certificate, the generating entity signs the new certificate itself and thus establishes a path of trust from the CA to the newly created proxy certificate. Since the proxy has a very short lifetime, its private key does not have to be kept as secure as the original private key of the owner. The proxies private key can be transferred over secure communication channels and used to identify the entity throughout the proxies lifetime, thus allowing single sign-on and delegation through the use of the proxy.

Web Service Authentication and Authorization. This component consists of two separate sub-components. The first component addresses security concerns through message- and transport level security.

The Globus Toolkit uses SOAP over HTTP for interacting with its Web Service components. Since SOAP does not define any security mechanisms itself, additional security mechanism are required.

Message level security is applied on a per-message basis, without any pre-existing context between the sender and receiver of a message. Using this security model, SOAP messages are protected by encrypting and/or signing the message contents, thus addressing integrity and confidentiality concerns.

Transport level security is based on providing a secure channel through which messages are exchanged instead of securing every message separately. This is realized by using HTTP in conjunction with Transport Layer Security (TLS) to provide for encrypted secure communication.

The second sub-component is the authorization framework of the Globus Toolkit and concerns authorization that can be performed at the level of resources, services or containers. Chains of authorization modules can be associated with such an entity and all modules in the chain are used to evaluate a certain request until a deny or permit decision is reached.

Community Authorization. Users and groups in a virtual organization can reside across multiple sites. Sites maintain their own authorization policies and are not responsible for maintaining policies for every user of other communities.

Because of this, the provider of a resource does not have to specify policies for users separately, but specifies a policy for a community as a whole.

Since this mechanism only allows to define a very coarse-grained policy, the Community Authorization Service (CAS) is used to enable a more fine-grained access control for every user belonging to a community. The CAS is responsible for maintaining a list of community users and for handling these users separately.

Delegation. The Delegation Service is used to delegate user credentials to a remote component, so that a component can request other services on behalf of the user by applying the user's credentials. Such a delegated credential can then be used to execute multiple invocations of remote services.

The delegation is done by handing a proxy certificate to the delegation service, which in turn keeps this proxy in a WS-Resource (section 3.5.3) for later retrieval whenever the proxy credentials are required. Proxies are usually short-lived entities that expire after a certain amount of time. Because of this, the Delegation Service also offers an interface for refreshing a delegated credential.

Credential Management. This component is divided into two sub-components. The SimpleCA sub-component is a wrapper around the certificate authority (CA) functionality provided by the OpenSSL library. It can be used to to issue credentials to Globus users and resources when a real certificate authority is not available.

The other sub-component, MyProxy, stores proxy credentials into an online repository. It has been developed to allow credential delegation in conjunction with web-based Grid Portals where the standard Delegation Service is not applicable because of the security protocols used between the user's web browser and web server [7].

3.5.2.2 Data Management

GridFTP. Since Grid applications possibly deal with huge amounts of data that may be distributed across several sites, a robust and efficient way method for transferring data between storage systems is required.

The GridFTP protocol is based on the traditional File Transfer Protocol (FTP), but adds extensions to the protocol targeted towards Grid environments [3]. It incorporates the security mechanisms provided by the Grid Security Infrastructure (GSI) for securing data transfers.

Due to the large amount of data, performance is an important aspect of the GridFTP protocol. For example, striped transfers allow for multiple network endpoints to be utilized in a single data transfer and parallel transfers allow a single data transfer to be distributed across multiple TCP streams between two endpoints.

Third party transfers allow for the mediation of data transfers between servers. A client can initiate the transfer between the two parties rather than participating self in the transfer.

Reliable Transfer Service. The Reliable Transfer Service (RFT) is a Web Service that is concerned with the management of reliable file transfers and builds therefore on the basic data transfer capabilities of GridFTP. It is capable of orchestrating several GridFTP file transfers in a transactional manner, that is all files are either transferred successfully or none is transferred at all.

The service uses a WS-Resource representation that includes a list of URL pairs for describing the files that need to be transferred. For monitoring purposes, information of the file transfer as a whole is exposed by the WS-Resource.

Replica Location Service. The management of file replicas is supported by the Globus Toolkit through the Replication Location Service (RLS). Essentially, the service is a possibly distributed registry that is used to keep track of one or more replicas of files in a Grid environment. That is, the registry keeps information on where replicas can be found on physical storage systems. This information is kept as mappings between logical file names and physical files. The logical file name serves as a unique identifier for physical replicas that might be situated at different sites throughout the Grid.

Services or users that make use of this registry can query the registry for a logical file name and receive a list of the physical replicas. The reverse method is also possible, that is, clients may ask for the logical file name of a given physical file in order to find other replicas.

3.5.2.3 Execution Management

WS-GRAM. The Grid Resource Allocation Management (GRAM) service allows for the execution of arbitrary user-defined programs on remote computational resources despite possible heterogeneity of remote systems.

In a common usage scenario, the GRAM service sets up the environment in which the computation is to be executed, stages input files to this environment and executes the user-defined program. Furthermore, the GRAM service is capable of monitoring the execution and notifying clients of state changes. The output produced by the remote program can later be copied from the environment to a user-defined location. The parameters of the execution are defined in a Resource Specification Language (RSL) XML-file. These parameters define among other parameters the program to be executed and the files to be staged to and from the execution environment.

The GRAM service interfaces with other services of the Globus Toolkit for various tasks. Files are transferred using GridFTP or the Reliable File Transfer Service and resources may be discovered using the Monitoring and Discovery System (MDS).

3.5.2.4 Information Services - MDS

Dynamic discovery and monitoring of services is provided by the Monitoring and Discovery System (MDS). MDS is concerned with monitoring and discovery of resources. Monitoring involves the detection and diagnosis of possible problems and allows corrective measures to be taken. Discovery on the other hand aims to identify resources or services with desired properties among the possibly large number of resources available throughout different sites of a Grid.

Architecture. From an architectural point of view, the MDS Web Services are based on the so called Aggregator Framework which is responsible for retrieving information through its Aggregator Sources and publishing information to Aggregator Services.

MDS includes three aggregator sources that are able to gather information in different ways. The Query Aggregator Source allows interaction with Web Services that are based on the Web Service Resource Framework (section 3.5.3) through the use of standard resource property queries, while the Notification Aggregator Source employs the Web Service notification mechanism as described in section 3.5.4 for information gathering. Additionally, information can be collected by executing external programs through the Execution Aggregator Source.

These aggregator sources deliver the gathered information to Aggregator Sink components, which are used by the actual Web Services that expose the functionality of the

MDS. These services are also referred to as Aggregator Services and are described in the subsequent paragraphs.

Index Service. The index service aggregates information from different sources and exposes the registered information as WS-ResourceProperties. It acts as a registry of resources and caches values from those resources. These values are updated periodically at user-defined intervals. As a consequence, information contained in the index service is recent in respect to the update interval but not absolutely up-to-date.

Since the data captured in Index Services is exposed as WS-ResourceProperties, clients can interact with Index Services through standard resource property queries. A common way of querying the Index Service is to execute an XPath query against the resource properties document (section 3.5.3) of the Index Service. The XML representation of the Index Service's data through the resource property document serves as the basis for WebMDS, a browser-based interface to the Index Service that employs XSLT to transform the resource properties document into HTML.

Due to the decentralized nature of the Grid, the presence of a single global Index Service that aggregates data from all existing resources is extremely unlikely. Because of this, Index Services may be organized hierarchical so that Index Services may aggregate data from other Index Services at lower levels. In this way, the information flows upwards from resources to Index Services at higher levels. Consequently, this hierarchical organization leads to the fact that Index Services which aggregate data directly from resources have more recent information but only knowledge of a limited number of resources. Index Services at higher levels keep older information but keep data of more resources.

Trigger Service. The Trigger Service evaluates data gathered by the Aggregator Sources against a set of pre-configured conditions. If such a monitoring criteria is met, the Trigger Service executes a pre-configured action. For example, a warning message may be sent if gathered data indicates a problem with a certain resource.

3.5.2.5 Common Runtime

The common runtime components provide libraries and tools to support the development of new services to be used with the Globus Toolkit. Such services include Web Services and other services.

Concerning Web Services, the WS-Core components implement the Web Service Resource Framework (WSRF) and WS-Notifications to be utilized by the Web Services that are distributed with the Globus Toolkit as well as custom developed Web Services. These components also include the Web Service containers described earlier. The WS-Core exists for the three programming languages Java, C and Python.

For other services, the common runtime components provide C libraries that include abstraction layers on top of low-level system calls and data types in order to facilitate portability.

3.5.3 Web Service Resource Framework

The Web Service Resource Framework (WSRF) introduces state handling capabilities in a standardized way to normally inherently stateless Web Services. These capabilities are introduced through stateful resources that are associated to Web Services.

3.5.3.1 Web Services and State

During interaction with clients, Web Services do not keep any information from previous invocations. Thus, Web Services itself have no notion of state. Any stateful interaction that is to be realized when using Web Services has to be implemented on top of Web Services. That is, stateful information needs to be provided inside the request message or managed by other components that interact with Web Services [5].

The representation of state in conjunction with Web Services is therefore dependent on the implementation of state handling. Since interoperability between Web Services is a central aspect of the Web Services Architecture [12], introducing implementation dependence is not desirable. It is argued that a standardized representation of state in the context of Web Services "enhances service interoperability, simplifies the definition of new service interfaces, and enables more powerful discovery, management, and development tools" [5].

3.5.3.2 Stateful Resources

In the context of Web Services and the WSRF, state is modeled by stateful resources that are associated with a Web Service. The combination of a Web Service and a stateful resource is referred to as a WS-Resource.

A stateful resource has a specific set of state data that can be represented in the form of an XML document which is referred to as the Resource Property Document. Although the state data is representable as an XML document, it does not imply that the actual implementation keeps state data in this manner.

Identity. Each stateful resource has an identity that is unique to the WS-Resource, while network wide uniqueness is not a requirement. For example, it is common practice to identify a stateful resource by a randomly generated unique identifier. The identity of a stateful resource is kept in the Resource Property Document along with other state data.

Addressing. The identity is not only used from the perspective of the Web Service that is associated with the stateful resource, but can also be used to construct a network wide pointer to the resource. This is done by extending an endpoint reference containing the address of the Web Service with the identity of the stateful resource. Such an endpoint reference that effectively references a WS-Resource is said to be WS-Resource-qualified.

Implied Resource Pattern. In the context of addressing and resource identity, the implied resource pattern [5] concerns Web Service message exchanges when dealing with stateful resources. Stateful resources are implicitly associated to a message exchange through the use of WS-Resource-qualified endpoint references. Consequently, the association between a WS-Resource and a given message exchange is not done by passing explicit resource identifiers as message parameters. Instead, this association is already established through the mechanism for addressing a WS-Resource.

Lifecycle. Additional to their generally stateless nature, Web Services have an unbounded lifetime and are therefore persistent entities. Stateful resources associated to

Web Services therefore require a defined lifecycle in order to be able to dynamically create and destroy stateful resources.

The creation of stateful resources is facilitated by the use of a pattern that is referred to as WS-Resource factory. A factory is responsible for creating stateful resources, assigning identifiers and associating the resources to Web Services. Resources might be associated with a Web Service statically at deployment time of the service or dynamically through message exchange.

Stateful resources may only be in use for a short period of time and can therefore be destroyed in several ways, e.g. to free system resources. Destruction of a resource is possible through immediate and scheduled destruction. Immediate destruction is initiated by a Web Service consumer by sending a destroy message to the WS-Resource that contains the stateful resource to be destroyed. Scheduled destruction is carried out on the basis of a scheduled termination time. When the termination time expires, the resource self-destructs without further interaction. Web service consumers might update the termination time to prolong or shorten the lifetime of the resource.

3.5.3.3 Resource Properties

The stateful resources that have been discussed so far are composed of data elements that are referred to as resource properties which hold the actual state data. Not taking into account the details on how the resource properties are actually implemented, the Resource Properties Document is an XML document that defines a projection of the resource's state held in the resource properties.

The Resource Properties Document is described through the use of an XML Schema that fully describes the resource properties including their data types. This XML Schema is imported or inlined into the WSDL definition of the Web Service that is associated to the stateful resource.

Resource properties can be accessed through standard interfaces that are defined by the WSRF. After obtaining a qualified endpoint reference of a WS-Resource, Web Service consumers are able to query and modify resource properties of the stateful resource. It is possible to query multiple resource properties with a single message exchange. Since the WS-Resource properties document defines an XML document as a view of the current state captured in the resource properties, Web Service consumers might execute an XPath query against this document in order to query resource properties. Additionally, WSRF allows the manipulation of resource properties through insert, update, and delete operations.

3.5.3.4 Additional Specifications

Along with the details of stateful resources and the resource properties described so far, the Web Service resource framework includes a set of additional specifications which are now briefly described.

Renewable References. References to stateful resources may at some point in time become invalid. WS-RenewableReferences defines the mechanisms for renewing such an invalid endpoint reference to a resource.

Service Groups. Collections of Web Services or WS-Resources can be organized as service groups in order to perform collective operations on the group's members.

Base Faults. The WS-BaseFaults specification defines a basic fault type in order to allow operations that are associated with stateful resources to return consistent faults. More specific faults can thus be described by extending the basic fault type.

3.5.4 Web Service Notifications

In object-oriented design, the Observer pattern is a common design pattern that allows an object to observe changes in another object and to be notified of these changes. This pattern is also commonly referred to as a publish-subscribe mechanism. Although the pattern is commonly utilized at the object level of object-oriented applications, it is adaptable to the Web Services environment.

The main purpose of Web Service notifications is to allow Web Services to notify other interested entities of events that have occurred inside the Web Service. In the basic form of the publish-subscribe interaction pattern, a Web Service publishes a topic of events to which other interested entities may subscribe. Occurrence of an event inside the publishing Web Service then triggers the notification of subscribed entities. Additional to this basic form, Web Service notifications may also involve an intermediary Web Service, called a notification broker, that may introduce additional features and enhanced scalability to the Web Service notification architecture. The Notification Service described in the present thesis is an implementation of such a notification broker, but does not fully implement the WS-BrokeredNotification specification described in section 3.5.4.3.

The exact interfaces and message exchanges involved in Web Service notifications are defined by the three specifications WS-BaseNotifcation, WS-BrokeredNotification, and WS-Topics. Starting with WS-Topics, these specifications will be described in more detail in the following section along with common usage scenarios.

3.5.4.1 WS-Topics

In the context of Web Service notifications and WS-Notification specification, topics categorize items of interest to which entities can subscribe. More specific, topics group events of a particular type as for example the completion of a computation.

The WS-Topics specification defines in detail how topics are structured and how subscribers can express which topics are of interest. Along with definition of the concepts of topics, WS-Topics provides XML Schema definitions for representing topics in the form of XML documents.

Topics. Notification producers create notification messages and associate them with one or more topics, thus categorizing the events. Subscribers that have subscribed to at least one of the topics that are associated with an event will possibly receive a notification message from the notification producer.

Topics can be organized hierarchical into so-called Topic Trees, which structure several topics into a root topic and several child topics. This organization defines that the root topic includes all topics further down in the tree structure. Generally, a topic anywhere in the tree structure includes all child topics of that particular topic. Because of this, a subscription to a root topic implicitly includes a subscription to all other child topics in the topic tree. This mechanism introduces a convenient way of subscribing to several related topics at once.

The tree structure of topics is also easily modeled in an XML document, and WS-Topics includes XML Schema definitions to do so. Topics are always assigned an XML-namespace, and all topics assigned to a particular namespace form a Topic Space, which is a collection of Topic Trees. Topic names do not need to be unique within a topic space, but it is required that root topics are named uniquely.

Topic Expressions. In order to select a topic of interest from one or more Topic Spaces, an interested subscriber employs Topic Expressions. Returning to the XML-representation, such expressions allow to navigate Topic Spaces and the included Topic Trees in order to select a relevant topic.

Topic expressions can be stated in three different dialects which provide different capabilities for selecting topics:

- `SimpleTopic` expressions allow the selection of root topics only.
- `ConcreteTopicPath` expressions are able to address all topics in a Topic Space by specifying the path to a topic.
- `FullTopicPath` expressions employ XPath to address topics.

3.5.4.2 WS-BaseNotification

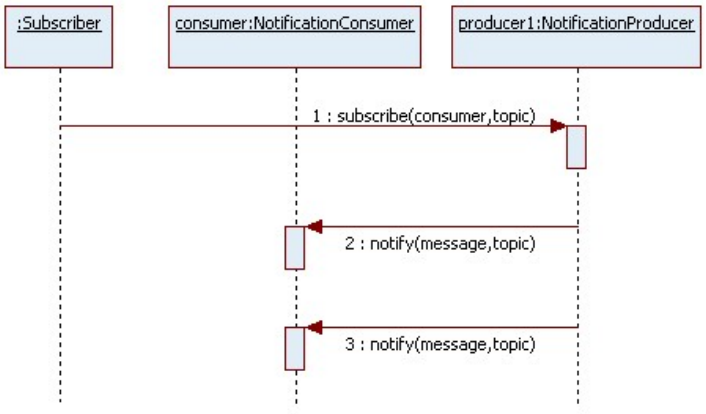

Figure 3.3: Basic interaction according to WS-BaseNotification

The WS-BaseNotification specification defines the basic publish-subscribe mechanism for Web Services. This involves that the notification consumer that is interested in a topic subscribes or is subscribed to a notification producer. Occurring events that are classified with the topic to which the consumer subscribed are then propagated using a notification message from the notification producer to the consumer.

Figure 3.3 shows the message exchange used for this notification mechanism. For simplicity, the notification consumer and the subscriber are the same entity in this example, but they might in fact be different entities so that an external subscriber subscribes the notification consumer to the producer.

The specification defines the complete message exchange and interfaces of the involved parties by providing a WSDL document. More specifically, it defines the basic NotificationProducer, NotificationConsumer, and SubscriptionManager interfaces which are implemented by service providers to provide the notification mechanism. For example the Globus Toolkit implements this specification.

3.5.4.3 WS-BrokeredNotification

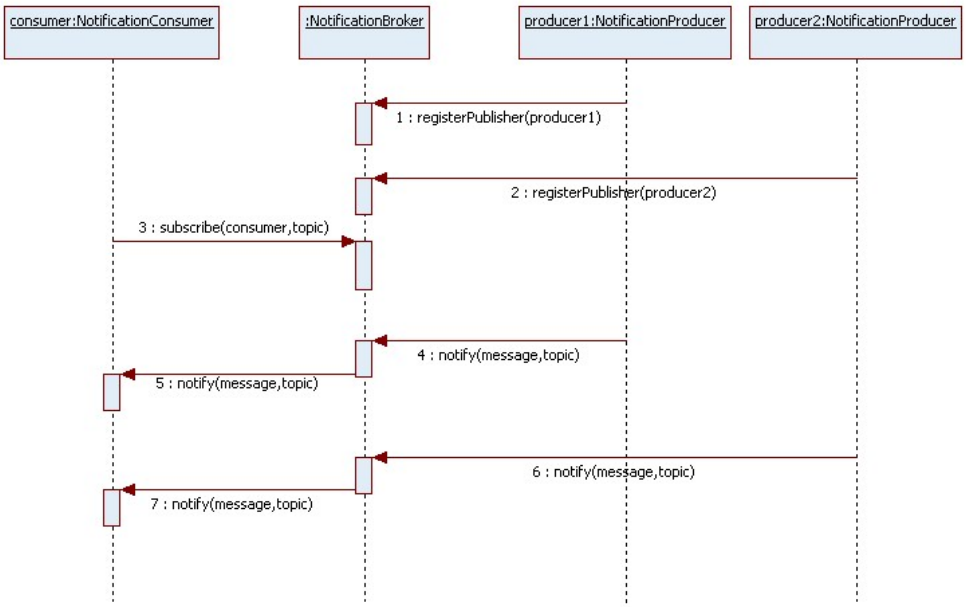

Figure 3.4: Interaction involving a NotificationBroker

In contrast to the WS-BaseNotification specification which defines point-to-point notifications, the WS-BrokeredNotification defines notifications that are propagated through the use of an intermediary Web Service called a notification broker. The intermediary broker effectively decouples notification consumers and producers and can relieve the notification producer from tasks such as subscription management and the propagation of notification messages to several consumers. Since the notification broker acts as both a notification consumer and producer, it implements both interfaces defined by the WS-BaseNotification specification.

Figure 3.4 shows a common usage scenario that involves two notification producers that publish events of the same topic, a notification broker acting as the intermediary and a notification consumer that is interested in events of that particular topic. Notification producers start by registering themselves as publishers to the broker and start sending notifications to the broker thereafter. Depending on the configuration of the broker, this registration might not be necessary. The notification consumer then subscribes to the broker, which in turn is able to propagate the notifications from the producers to the single consumer. Since the consumer subscribes to the topic that both producers publish to, it ultimately receives messages from both producers.

Utilizing a notification broker to distribute notification messages assists the notification producer with the task of managing subscriptions itself since notification consumers can subscribe directly to the broker instead of the notification producer. Additionally, the brokered approach provides greater scalability since the workload of distributing notifications to consumers is shifted to the notification broker. The notification producer only has to transmit a single notification message, even if a large number of consumers are subscribed to the notification broker. Extending this idea, notification brokers may even be chained in order to further distribute the workload of transmitting notification messages to consumers.

The decoupling of notification consumers and producers allows notifications to be distributed from producers to consumers anonymously so that both entities have no knowledge about each other. This is especially important in the context of loose coupling and Service-Oriented Architectures which are described in section 3.4.

Chapter 4

Results

The present chapter gives a detailed description of the software that has been developed during the thesis project. The presented software is a notification broker service that builds on the basic Web Service notification capabilities, which are described in section 3.5.4.2 as the WS-BaseNotification specification. This Notification Service adds additional possibilities to the standard notification capabilities.

At first, the Notification Service is described at a high level, thus showing the overall architecture and integration into a service-oriented Grid environment. After this overview, the design and architecture of the Notification Service is presented including details of the actual implementation.

4.1 Architectural Overview

Because of its intended use as a notification broker, the Notification Service is used as a mediator that decouples notification producers and notification consumers. This involves consequently that producers and consumers alike can be unaware of each other which facilitates the concept of loose coupling as described in section 3.4.2. Another aspect concerning Service-Oriented Architectures and loose coupling in the context of the Notification Service is the use of a discovery service to enable consumers and producers to find and utilize the Notification Service. Since the Notification Service is build on top of the Globus Toolkit (section 3.5), the Globus Monitoring and Discovery System (MDS) is utilized for dynamic discovery.

Figure 4.1 shows a high level overview of the entities that are involved in utilizing the Notification Service. Notification producers register or are registered with the Notification Service and publish their notifications to the Notification Service thereafter. Having discovered the appropriate Notification Service by querying the MDS Index Service, consumers subscribe to the Notification Service and notifications from producers are then propagated to the subscribed consumers in different ways.

Facilitating a loose coupling of services, the Notification Service is not aware of any implementation specific details of notification consumers and publishers beyond the mechanisms that are used for basic Web Service notifications as defined by the WS-BaseNotification specification. Although fully functional without producer specific knowledge, the Notification Service provides a plugin mechanism that may introduce additional notification processing features for specific notification producers. However, this introduces a tighter coupling between the Notification Service and producers.

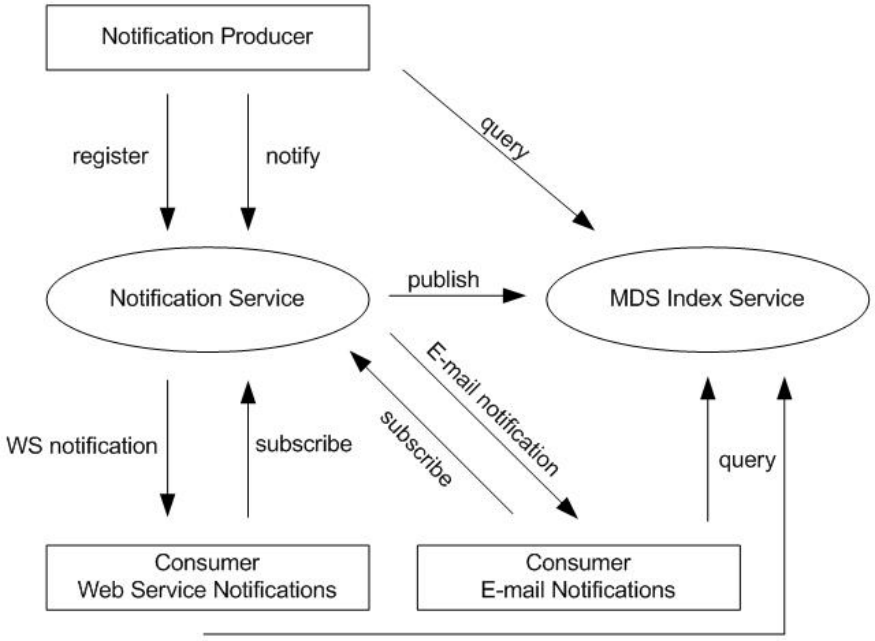

Figure 4.1: Notification Service as a Mediator between Producer and Consumer

4.2 Usage Scenarios

The use of a Notification Service for distributing notification messages can be initiated by both consumers and producers.

A notification producer may decide to delegate the distribution of notifications and subscription handling to the Notification Service. In order to employ the Notification Service for this task, the producer first discovers a suitable Notification Service through the MDS. After retrieval of the Notification Service's address, the producer registers with the Notification Service which in turn subscribes to the notification producer and advertises itself in the MDS as the Notification Service in charge of managing notifications for that particular notification producer. Possible consumers find the Notification Service by querying the MDS and subscribe to notifications from the producer. This producer-initiated scenario is depicted in figure 4.2.

Another possibility for using a Notification Service is the consumer-initiated scenario as shown in figure 4.3. In this scenario, the notification consumer is responsible for integrating the Notification Service for notification message distribution. To achieve this, the notification consumer registers the notification producer with the Notification Service instead of the publisher registering itself. Again, the dynamic lookup using the MDS allows the notification consumer to discover a suitable Notification Service. After the consumer has registered the producer and subscribed itself to the Notification Service, notification messages can be propagated from the producer through the Notification Service to the consumer. As in the producer initiated scenario, the Notification Service advertises its association to a particular notification producer in the MDS. This also

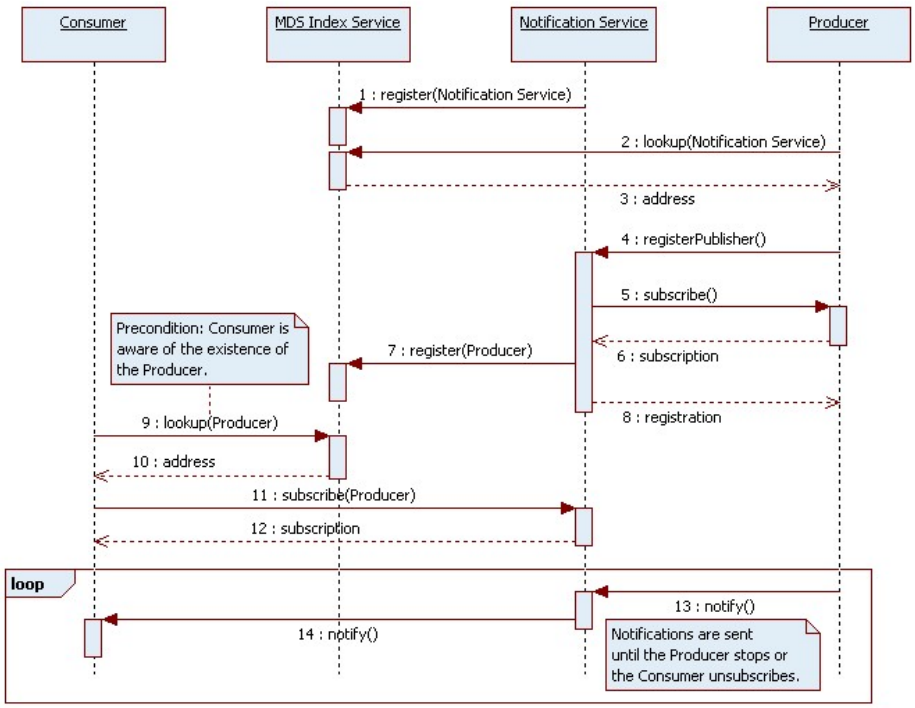

Figure 4.2: Producer-Initiated Usage Scenario

allows other consumers interested in the same producer to subscribe to the Notification Service directly since the producer has already delegated the message distribution to the Notification Service on behalf of the first consumer.

4.3 Notification Service Design

After introducing the architecture and usage of the Notification Service from a general point of view, the present section advances to more technical details of the Notification Service and the basic building blocks used for implementation.

4.3.1 Environment

Targeted towards Grid environments, the Notification Service is based on the Globus Toolkit middleware as described in section 3.5. Services and implementations of the Globus Toolkit utilized by the Notification Service include:

- An implementation of the Web Service Resource Framework (WSRF) for stateful Web Services as discussed in section 3.5.3.
- An implementation of Web Service notifications according to WS-BaseNotification and WS-Topics.

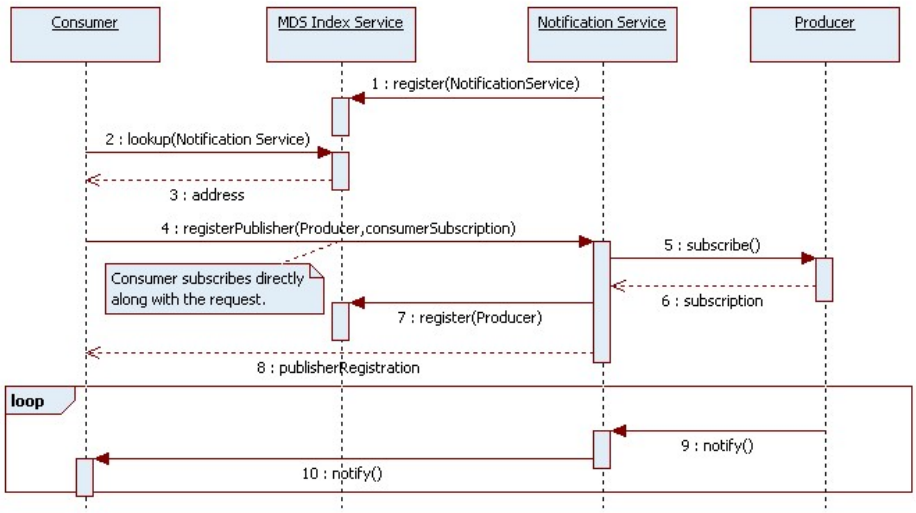

Figure 4.3: Consumer-Initiated Usage Scenario

– Java WS-Core Web Service container providing a runtime environment for implementing user developed Web Services in Java.

– Monitoring and Discovery System (MDS) Web Services for dynamic discovery of services as described in section 3.5.2.4.

At the core, the Globus Toolkit employs the Apache Axis [10] SOAP engine and the service interface and types are described using WSDL (section 3.3.2) and XML Schema. The Globus Toolkit also includes tools that are capable of generating client classes, so-called stubs, from WSDL documents for interaction with Web Services.

4.3.2 Resource Representation

The Notification Service requires stateful data for managing notification producer registrations. Stateful resources as defined by the WSRF are a standard solution to incorporate state into Web Services.

Basing a Web Service on stateful resources has an impact on the internal design of the Web Service since it requires certain classes to be in place along with structuring the stateful data into a single stateful resource type since the Web Service Resource Framework only allows a single stateful resource type for a Web Service. Using several resource types would consequently result in the need to split the service into two cooperating services.

A notification broker as it is defined by the WS-BrokeredNotification specification needs to keep track of consumer subscriptions and producer registrations. Modeling this with stateful resources would consequently lead to the requirement of two distinct types of stateful resources representing consumer subscriptions and producer registrations, thus requiring two cooperating Web Services.

Concerning the Notification Service, a design decision was made to use an explicit resource representation for producer registrations only. The handling of consumer subscriptions is therefore left to the subscription handling mechanisms of the Globus Toolkit, which are provided through its implementation of the WS-BaseNotification specification. Subscription and notification handling capabilities provided by the Globus Toolkit concern only Web Service notifications.

Looking at the problem description for the present project, it should be possible to propagate notifications by other mechanisms such as email or instant messaging. To allow for this, the publisher registration resource keeps a list of subscribed addresses that are notified by other means than Web Service notifications. The publisher registration resource also keeps information that is used for filtering notifications. This filtering informations is provided in two forms, a list of values that are interesting to the notification consumers and a regular expression that is evaluated against the string representation of the notification message. It is possible to use both filtering methods combined, each method separately, or none at all.

In order to keep a history of notification messages, the stateful resource holds a predefined number of messages. This is included because of notification mechanisms that need to present a history of messages when new notifications arrive.

4.3.2.1 Resource Properties

Since data kept in stateful resources is divided into resource properties that are defined using an XML Schema included into the WSDL of the Notification Service, the definition of the publisher registration resource is now presented as its XML Schema definition of the resource properties. The complete XML Schema and WSDL files can be found in the appendix.

```
<schema targetNamespace="http://elverkemper.com/NotificationService"
[...]
  <!-- Resource Property Elements -->
  <element name="Message"            type="anyType" />
  <element name="MessageLog"         type="anyType" />
  <element name="RegistrationName"   type="string"  />
  <element name="PublisherReference" type="wsa:EndpointReferenceType"/>
  <element name="NonWsSubscriptions"
           type="notif:NonWsSubscriptionType"/>
  <element name="FilterRegExp"       type="string"  />
  <element name="InterestingValues"  type="anyType"  />
  <element name="PublisherTopic"     type="wsnt:TopicExpressionType"/>

<!-- Resource Property Document -->
  <element name="NotificationRpDoc">
    <complexType>
      <sequence>
        <element ref="notif:Message"
                 minOccurs="0" maxOccurs="1"/>
        <element ref="notif:MessageLog"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element ref="notif:RegistrationName"
                 minOccurs="0" maxOccurs="1"/>
```

```
        <element ref="notif:PublisherReference"
                 minOccurs="1" maxOccurs="1"/>
        <element ref="notif:NonWsSubscriptions"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element ref="notif:FilterRegExp"
                 minOccurs="0" maxOccurs="1"/>
        <element ref="notif:InterestingValues"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element ref="notif:PublisherTopic"
                 minOccurs="1" maxOccurs="1"/>
      </sequence>
    </complexType>
  </element>
[...]
</schema>
```

The Notification Service processes incoming notifications according to the values of these resource properties. Figure 4.4 illustrates the usage of the resource properties by the notification handler which is responsible for processing and forwarding notifications.

Message. This resource property exposes incoming notification messages that have been sent by the publisher. A predefined number of messages is kept as a history. In order to utilize the notification mechanism implemented by the Globus Toolkit which allows subscribers to be notified of changes of values of resource properties, this resource property is exposed as a topic to which notification consumers can subscribe. The topic under which this resource property is exposed is determined dynamically when a publisher registration resource is created and is identical to the topic under which the notification producer publishes the notification messages.

MessageLog. For logging purposes and in particular for Notifiers that publish a history of past events (such as the RSS Notifier), the Notification Service keeps a number of past notification messages in this resource property. Messages are implicitly moved into this resource property when a new notification message is saved to the `Message` resource property. As for the current implementation, this logging functionality keeps 20 messages at the maximum. After reaching this limit, the oldest message will be discarded upon insertion of a new message.

RegistrationName. Web service interactions and notification message exchanges concern machine-to-machine interaction. Although notification messages in XML format are readable by humans to some extent, it might be hard to determine the origin of the notification message by looking at the reference that identifies the notification producer. To increase the usability of notifications that are sent to human end users by email or instant messaging, publisher registrations include the `RegistrationName` resource property which defines a simple text label to identify publishers. This label can be set arbitrarily when registering a publisher with the Notification Service.

PublisherReference. Notification producers are in effect resources and identified by unique references to the resources. These references are kept in this resource property and are also published in the MDS. Incoming notifications include this unique identifier

and the Notification Service uses it to match incoming messages to the appropriate publisher registration resource. Since this identifier is also published into the MDS system, interested consumers can dynamically discover the resource and subscribe to the Notification Service in order to receive notification messages that originate from the publisher.

NonWsSubscription. Subscriptions of consumers that receive notifications through mechanisms other than Web Service notifications as for example email or instant messaging are captured by this resource property in form of addresses. An address in this context must be composed of a protocol identifier and the actual address. This type construct is defined in the XML Schema of the WSDL document and consists of two separate strings. As a restriction, the XML Schema defines the possible values for the protocol identifier according to protocols supported for notification distribution by the Notification Service. These protocols are served by Notifier components which are described in section 4.3.4.3.

FilterRegExp. This resource property defines a regular expression which is evaluated against the string representation of the value of the notification message. The notification message is propagated only if the regular expression matches the current notification, otherwise it is discarded.

InterestingValues. Notification consumers might only be interested in a subset of notifications. To support this, consumers might specify exactly which values of notification messages are interesting to them. These values are then matched to incoming notification messages to decide if the notification is to be propagated or discarded.

PublisherTopic. When a notification producer is registered with the Notification Service, the topic under which the producer publishes notification messages is kept in this resource property. This topic is also published in the MDS Index Service, so that possibly interested notification consumers can dynamically discover the publisher registration resource.

4.3.3 Notification Service Interface

The Web Service interface of the Notification Service is composed of several parts, some of which are provided by facilities of the Globus Toolkit. The complete interface definition in terms of types and messages is given by the WSDL document provided in the appendix.

The interface of the Notification Service follows the principle that read access to resource properties is provided by the standard resource properties interface provided by the Globus Toolkit's implementation of the Web Service Resource Framework as described in section 3.5.3.

Write access to resource properties and the creation of the stateful publisher registration resource on the other hand is explicitly provided by the implementation of the Notification Service. The factory operation `registerPublisher` creates a new stateful resource for a notification producer and initializes the values of the resource properties with values that might have been provided in the request message of the operation. Additionally, a notification consumer that registers a publisher by using this operation may

supply subscription information in order to be subscribed to the publisher registration instantly after its creation.

Furthermore, the Notification Service provides methods for manipulating the filtering behavior for a given publisher along with subscription management for non-Web Service consumers. Notification filtering is controlled by the `setFilter` operation which interacts with multiple resource properties that concern filtering, `InterestingValues` and `FilterRegExp`. The operation is able to set a single or even both resource properties at once. Calling this operation without providing values effectively disables filtering.

Subscription management for non-Web Service consumers is provided by the operations `subscribeNonWs` and `unsubscribeNonWs`. Both operations expect a list of addresses that are to be subscribed or unsubscribed respectively. Both operations of the interface manipulate the `NonWsSubscriptions` resource property which keeps a list of subscriptions that are served by the Notifier components.

Another part of the Notification Service's interface is concerned with handling subscriptions of notification consumers that receive notification via the standard method of Web Service calls as it is described by the WS-BaseNotification specification. Since the Globus Toolkit provides an implementation of this specification, it consequently includes a subscription handler component that is utilized by the Notification Service to delegate the task of handling subscriptions of notification consumers. In order to use the provided subscription handling capabilities, the port type of the Notification Service inherits operations from the pre-defined Notification Producer port type. The `subscribe` operation of this port type is used by notification consumers to subscribe to notification messages that are sent by the notification producer to the Notification Service. Additionally the `getCurrentMessage` operation allows notification consumers to query the Notification Service for the last notification message that has been received from the notification producer. This mechanism allows notification consumers that were not subscribed when the latest notification message arrived to retrieve the latest message.

The previously described operations of the port type are defined through the port type definition which is included in the WSDL document. For the purpose of structuring the WSDL document, the XML Schema types that are used inside the messages along with the resource properties are defined in an external XML Schema file. Both files can be found in the appendix.

In general, the type definitions used in the XML Schema are as restrictive as possible to avoid unsupported data being handed to the Notification Service. As an example, the resource property holding subscriptions of consumers that are notified by other means than Web Service calls consists of a protocol part and an actual address. In this case, the protocol part is restricted to protocols supported by the Notification Service. Such data validation partially removes the need to programmatically validate request data in the Notification Service itself, but also publishes the expected data format by means of the WSDL document.

4.3.4 Internal Service Design

Along with the stateful resource and its resource properties that model a publisher registration, the Notification Service contains behavioral components that provide functionality and act upon the stateful resources. The previous section describes the interface to the Notification Service including the factory method which creates the publisher registration resources. In this section the components for handling and redistributing incoming notification messages are explained in detail.

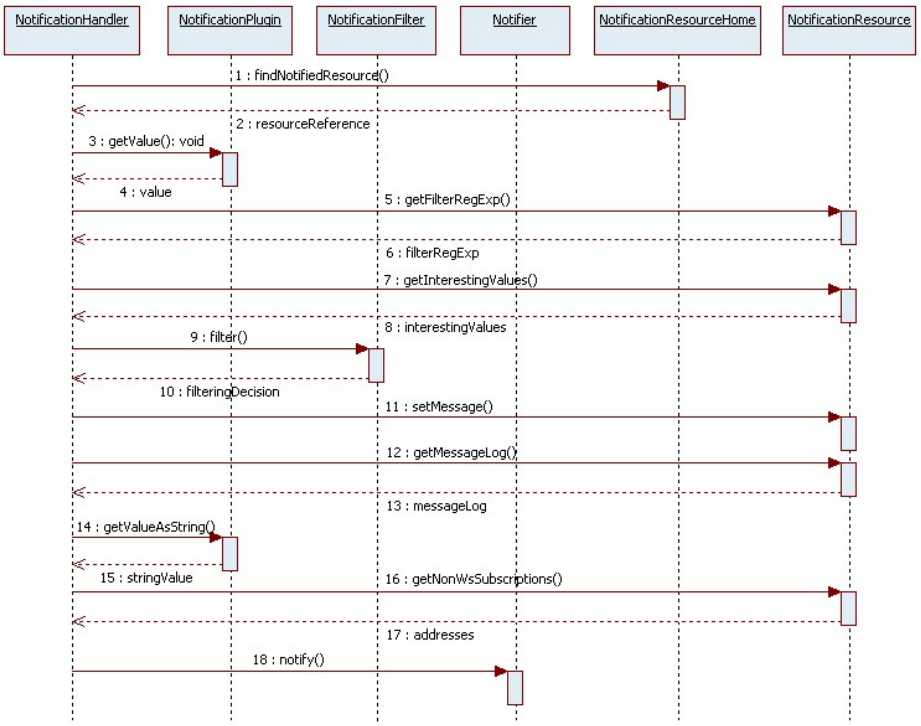

Figure 4.4: Message Processing by the Notification Handler

4.3.4.1 Notification Handler

The notification handler component is invoked when notification messages arrive at the Notification Service. In order to allow the notification handler to process incoming notifications appropriately, a valid publisher registration and thus an existing resource for the publisher is required. Messages that cannot be matched to a valid resource are discarded by this component.

The notification handler is realized as an inner class of the Notification Service implementation class and its `deliver` method is invoked every time a notification message arrives. This invocation is initiated by the basic Globus components that implement the WS-BaseNotification mechanism. Thus the Notification Service can rely on having its notification handler activated when notification messages arrive.

The behavior of the notification handler when being activated due to an incoming notification message is depicted in figure 4.4. The notification message is first dispatched to the appropriate plugin for extracting the value of the notification message. The notification handler recognizes which plugin to use by examining the namespace of the incoming notification message and uses this namespace as a key for finding the appropriate plugin. Using the reference of the publisher contained in the notification message, the notification handler is able to retrieve a reference to the stateful resource that represents the registration of the publisher. This is important since the resource includes information that is needed for further processing of the notification message.

At first, filtering information is retrieved from the resource and matched against the notification message by a filter component which may discard the notification message and stop further processing.

If the message is accepted for further processing, the next step involves distributing the message to subscribers that are notified by mechanisms other than Web Service notifications such as email and instant messaging. These other mechanisms are served by components called Notifiers which are described in a subsequent section. For notification distribution by Notifiers, the resource contains a list of addresses to which the notification is to be forwarded. Addresses in this context include an identifier that specifies the notification mechanism for which the address is valid.

For the final processing step which concerns forwarding the notification message to subscribers which receive notifications via Web Service notifications, the notification handler relies on the facilities of the Globus Toolkit for Web Service notifications. That is, subscribers may be notified of changes in a resource property which is exposed as a topic. The `Message` resource property contained in the publisher registration resource is exposed in such a way, thus allowing consumers to subscribe. Concerning the notification message that is processed by the notification handler, forwarding to Web Service subscribers is then achieved by writing the value of the incoming notification message to this particular resource property. Since the resource property is exposed as a topic, changing its value triggers the creation of a new notification message which is sent to the consumers. Thus the original notification message has been forwarded by the Notification Service.

After describing the internal behavior of the notification handler, the subsequent sections cover the internal workings of the components that interact with the notification handler.

4.3.4.2 Plugins

The Notification Service makes use of plugins to adapt to different types of notification producers. More specific, the functionality that is provided by plugins is concerned with extracting the values of the notification messages and converting these values into their correct Java types for further usage.

Plugins also provide a method that converts the value of a notification message into a textual representation. This value is then used in conjunction with Notifier components for sending notifications to human end users.

The appropriate plugin for an incoming notification message is selected according to the XML namespace that is used by the notification message. Since different notification producers use different namespaces, this can be used to identify the producer and select the correct plugin accordingly.

Plugins are defined in a configuration file that specifies the name of the Java class to be loaded along with the namespace value that is used by the notification handler to find the correct plugin. The configuration mechanism is described in more detail in section 4.3.4.4.

4.3.4.3 Notifiers

Notifications may be propagated by the Notification Service through mechanisms other than Web Service notifications. That is, interested parties may subscribe to a publisher registration with, e.g., an email or instant messaging address.

To serve such subscribers, the Notification Service includes Notifier components that allow notification messages to be forwarded as for example email messages. Generally, Notifiers are used for all notifications that are to be transported or otherwise processed by means other than standard Web Service notifications.

Subscriptions of notification consumers that need to make use of Notifiers are kept in the "NonWsSubscriptions" resource property of a publisher registration resource. As mentioned earlier, such a subscription includes an identifier for the Notifier component that is to be used along with an address that is valid in the context of this notifier. Using this identifier, the notification handler selects the appropriate Notifier for a given subscription. As an example, a subscription including the identifier "email" and the address value of "ens04ter@cs.umu.se" would cause the notification handler to employ a Notifier with the identifier "email" to forward the notification message to the given email address.

The behavior of a Notifier is not generally defined. It is therefore possible to interface a wide variety of different methods. The Notification Service includes three Notifiers that cover different methods for processing notification messages. All implemented Notifiers have in common that notification messages are forwarded to a human end user, while the Web Service notification mechanism is mainly used for machine-to-machine communication. Notifiers may however implement arbitrary processing and forwarding capabilities.

Initially, the implementation of a Notifier for the Short Message Service (SMS) in order to distribute notifications to mobile phones was planned. Unfortunately, a SMS provider was unavailable during development. Integrating SMS as a Notifier however would only require to develop a new Notifier component that can be integrated into the Notification Service.

Email. This Notifier allows email addresses to be subscribed to a publisher registration resource and thus having incoming notification messages forwarded as emails to the subscribed addresses.

From a more technical point of view, the email Notifier is implemented on top of the Java Mail API, which provides a convenient way to interact with mail servers using the Simple Mail Transfer Protocol (SMTP).

Jabber Instant Messenger. Another option for subscribing to notifications from a notification producer is to use the Jabber Notifier which uses the Jabber open-source instant messaging protocol. Subscribing a Jabber address to a notification producer resource will therefore cause the Notifier to forward notification messages as Jabber instant messages.

RSS. In contrast to the two previous Notifier which use a message based protocol for forwarding notifications, this Notifier produces a RSS feed of notification messages. Since RSS is an XML format that can be interpreted by end user client programs such RSS aggregators and traditional web browsers, the Notifier formats the incoming notification messages into an XML document that conforms to the RSS 2.0 specification [16]. The RSS feed can be seen as a log of notification messages that have been processed by the Notification Service.

Since the Globus Toolkit does not provide a web server to host a RSS feed, the RSS Notifier creates the XML document into a file and copies the file to a remote web

server using Secure Copy (SCP). The remote web server along with login credentials are defined in the configuration file of the Notification Service.

4.3.4.4 Configuration

The Plugin and Notifier components are dynamically loaded by the Notification Service by the use of an XML-based configuration file. This configuration file defines the fully qualified class names of the components to be loaded along with a list of parameter names and their corresponding values.

After extracting the information from the XML file through the use of standard XPath queries, the Notification Service loads the defined classes dynamically by utilizing the fully qualified class name. The parameter names and corresponding values defined for each component are placed into a map data structure and passed to the constructor when instantiating the component's class. In this way, the component gains access to the parameters that were defined in the configuration file.

The dynamic classloading mechanism used for loading and configuring Plugins and Notifiers allows to integrate additional components without changes in the implementation of the Notification Service.

4.3.4.5 Notification Filtering

The filtering of notifications is handled by a single component by matching pre-defined filtering rules against the incoming notification message. The filtering is applied at the level of publisher registrations, that is all messages that arrive at the Notification Service from a single publisher are matched against the same filtering expressions which are held in two resources properties. This consequently introduces the limitation that that notification consumers which subscribe to the Notification Service cannot specify an individual filtering expression in the current implementation of the Notification Service.

The notification filtering mechanism of the Notification Service allows two different methods for content-based filtering that defines which notifications should be discarded or accepted. The first method which is connected to the resource property `InterestingValues` and allows to enumerate values of the notification message that are to be accepted by the Notification Service. An alternate method connected to the resource property `FilterRegExp` allows the definition of a regular expression that is matched against the string value of the notification message. Since the Notification Service is implemented in Java, regular expressions follow the syntax of regular expressions of Java. This introduces of course a certain coupling to the Java platform and might be replaced by a more platform independent approach in the future.

4.3.4.6 MDS Integration

Recalling from the previous chapter describing the Globus Toolkit, the Monitoring and Discovery System (MDS) is used to dynamically discover resources and services. In order to allow possible users, notification consumers and producers to find and utilize the Notification Service dynamically, the Notification Service registers itself with the Index Service of the MDS. The data that is published into the index is based on the publisher registration resources that represent a registration of a notification producer that publishes notifications to the Notification Service.

The information that is published to the Index Service allows possible other notification consumers that are aware of the identifier of the notification producer to find

the responsible Notification Service and subscribe to it without involving the producer itself. On the other hand, notification producers that are interested in delegating the distribution of notification messages to the Notification Service may discover its address by querying the Index Service.

From a technical point of view, the registration of a resource is performed by the resource home, directly upon its creation. The registration involves the configuration of an MDS aggregator source that periodically queries the resource properties of the registered resource and publishes this data in the index.

Five resource properties of a publisher registration resource are aggregated into the index.

1. `PublisherReference`
2. `RegistrationName`
3. `PublisherTopic`
4. `FilterRegExp`
5. `InterestingValues`

This information includes the identity of the notification producer, the topic under which notifications are published and the current status of the notification filtering expressions

4.3.4.7 Client API

The development of clients that interact with the Notification Service is simplified by the provision of a client API written in Java. The client API consists of two classes that resemble the operations available in the Web Service interface of the Notification Service and hide lower level communication details from the client. This introduces the additional advantage that client programs are isolated from minor changes in the interface of the Notification Service.

The Notification Service uses stateful resources to represent publisher registrations. Such a resource is created when the `registerPublisher` factory operation of the Web Service interface is invoked.

Similar to the behavior of the Web Service and the `registerPublisher` factory method, the client class `NotificationServiceClientFactory` creates instances of the `NotificationServiceClient`, which effectively invokes the corresponding operation of the Web Service. Therefore, an instance of the `NotificationServiceClient` class is associated to a stateful resource of the Web Service and provides methods to manipulate the remote stateful resource.

For dynamic discovery of publisher registration resources and the Notification Service itself, the client API provides methods that query a given MDS Index Service and find available publisher registrations and notification services that are registered in the index.

4.4 Discussion

The detailed design of the Notification Service is the result of a number of design decisions that have been made during design and development of the Notification Service. This section discusses some aspects of the final design of the Notification Service, evaluates advantages and limitations, and introduces possible alternatives to the current design.

In general, design decisions have been made to provide an extensible system that integrates well with the architectural and technological environment of Web Services, Service-Oriented Architectures, Grids, and the Globus Toolkit. Design decisions made are however also strongly influenced by external factors such as the limited time available for the project, previous inexperience with the technology base and poorly documented software components.

4.4.1 Resource Representation and Subscription Management

Stateful resources of the Notification Service represent publisher registrations to which notification consumers can subscribe. A different approach could model the stateful resources as consumer subscriptions to which publishers register. The latter approach has not been followed since the Globus Toolkit already provides subscription management that the Notification Service was able to utilize, thus decreasing development efforts. This simplification comes however at the expense of not being able to apply content-based filtering for every subscription to the Notification Service separately since the implementation of subscription management provided by the Globus Toolkit does not support filtering as provided by the Notification Service. Future development efforts might develop a Notification Service that provides its own implementation of subscription management rather than relying on the provided implementation. Such an implementation would deal with two separate types of stateful resources: publisher registrations and consumer subscriptions. Since WSRF allows only one resource type for a single Web Service, this would require the Notification Service to be split into two communicating services.

4.4.2 Notification Handling

The notification handler processes incoming messages from notification producers. The Globus Toolkit provides an abstraction of the basic notification mechanism that delivers the contents of a message together with other information as parameters to a callback method. Since the Notification Service is an infrastructure service which would need to handle messages at a lower level than other services which build on top of the provided notification mechanisms, it was unnecessarily difficult to simply redistribute a message without changing it. The Notification Service should be able to access the SOAP notification message or at least its Java representation directly, but this was not possible because of the abstractions provided by the Globus Toolkit. The API of the Globus Toolkit might allow for this, but due to the poor documentation at the API-level, a solution was not found within the limited time of the project. Further development efforts should aim to break the dependency on the toolkit's abstractions of the notification mechanisms and implement the interface for notification consumers and producers directly.

4.4.3 Filtering

The Notification Service only supports content-based filtering of notifications and does not explicitly deal with topic-based filtering. The topics under which a notification producer publishes its notifications are propagated by the Notification Service so that it publishes these notifications under the same topic. More advanced topic filtering mechanisms, especially the combination of hierarchical Topic Trees and XPath expressions provide an interesting technique for topic-based filtering in future development.

4.4.4 Extensibility

The approach of using dynamically loaded components such as Notifiers and plugins was taken to provide an effective mechanism for extending the Notification Service with support for other notification mechanisms. Further development could refactor Notifiers into separate Web Services in order to allow other services to use the additional communication mechanisms as well.

Chapter 5

Conclusions

The developed Notification Service provides notification brokering capabilities and features like notification filtering and additional messaging mechanisms that enhance the capabilities of the standard Web Service notification mechanism. Its open technology base and the dynamic discoverability facilitate the integration into the Service-Oriented Architectures of Grid infrastructures.

Although it was possible to implement a notification broker using the Globus Toolkit, development of such an infrastructure service was difficult because of the implementations of the basic notification mechanisms provided by the toolkit. These implementations abstract the underlying notification infrastructure in order to ease the standard use of notifications. A notification broker is however concerned with notification message handling at a lower level so the possibility to do so would have simplified development to a great extent. On the other hand, the development of the Notification Service was able to benefit greatly from the provided implementations of Web Service subscription management. Unfortunately, the Globus Toolkit turned out to be poorly documented at the API-level, thus significantly slowing development efforts and often requiring a closer examination of the toolkit's source code.

5.1 Limitations

The Notification Service has two limitations that should be considered. Since the service is based on the subscription manager that is provided by the Globus Toolkit, it is currently not possible to provide filtering on a per-subscription basis. The filtering settings are therefore global in the scope of a single publisher registration. It is however possible to register the same notification producer twice to the Notification Service with different filtering rules.

The specification for Web Service notifications, WS-Notification, defines the interfaces of a notification broker in detail. The Notification Service provides a similar interface and set of features, but is not compliant with the WS-BrokeredNotification specification.

5.2 Future Work

Future enhancements of the Notification Service should aim to provide an implementation of subscription management that allows for distinct filtering rules to be applied for every subscription and to provide a single subscription interface for subscriptions of Web Service notification consumers and other mechanisms alike. From a standards perspective, it is also desirable to fully implement the WS-BrokeredNotification specification to ensure interoperability between different implementations of this specification.

References

[1] Bart Jacob et al. *Introduction to Grid Computing*. International Business Machines Corporation, 2005.

[2] Von Welch et al. X.509 Proxy Certificates for Dynamic Delegation. 2004.

[3] W. Allcock et al. GridFTP Protocol Specification (Global Grid Forum Recommendation GFD.20), 2003.

[4] Ian Foster. A GT4 Primer. Website, 2005. http://www.globus.org/primer/; visited on June 10th 2008.

[5] Ian Foster, Jeffrey Frey, Steve Graham, and Steve Tuecke. Modeling Stateful Resources with Web Services v. 1.1. 2004.

[6] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[7] Jason Novotny, Steven Tuecke, and Von Welch. An Online Credential Repository for the Grid: MyProxy. *hpdc*, 00:0104, 2001.

[8] OASIS. UDDI Version 3.0.2, UDDI Spec Technical Committee Draft. Website, 2004. http://uddi.org/pubs/uddi_v3.htm; visited on June 10th 2008.

[9] OASIS. Reference Model for Service Oriented Architecture 1.0. Website, 2006. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf; visited on June 10th 2008.

[10] Apache Web Services Project. Axis. Website, 2008. http://ws.apache.org/axis; visited on June 10th 2008.

[11] Eric Pulier and Hugh Taylor. *Understanding Enterprise SOA*. Manning Publications Co., 2006.

[12] W3C. Web Services Architecture. Website, 2004. http://www.w3.org/TR/ws-arch/; visited on June 10th 2008.

[13] World Wide Web Consortium (W3C). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Website, 2007. http://www.w3.org/TR/soap12-part1/; visited on June 10th 2008.

[14] World Wide Web Consortium (W3C). SOAP Version 1.2 Part 2: Adjuncts (Second Edition). Website, 2007. http://www.w3.org/TR/soap12-part2/; visited on June 10th 2008.

[15] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Website, 2007. http://www.w3.org/TR/2007/REC-wsdl20-20070626/; visited on June 10th 2008.

[16] Dave Winder. RSS 2.0 Specification. Website, 2003. Available online at http://cyber.law.harvard.edu/rss/rss.html; visited on June 10th 2008.

Appendix A

Source Code

This chapter lists the interface definition of the Notification Service. The first section gives the WSDL document defining messages and operations. The type definitions used by the WSDL document are given in the subsequent section for better readability.

A.1 WSDL Interface Specification

Contents of the file `elverkemper-notification_interface.wsdl`:

```
  <?xml version="1.0" encoding="UTF-8"?>
<definitions name="NotificationService"
  targetNamespace="http://elverkemper.com/NotificationService"
  xmlns:notif ="http://elverkemper.com/NotificationService"
  xmlns:wsrpw ="[....]/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
  xmlns:wsrlw ="[....]/wsrf-WS-ResourceLifetime-1.2-draft-01.wsdl"
  xmlns:wsntw ="[....]/wsn-WS-BaseNotification-1.2-draft-01.wsdl"
  xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

<!-- Imports -->
<!-- ************************************** -->
  <import namespace="[....]/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
      location="../wsrf/properties/WS-ResourceProperties.wsdl"/>

  <import namespace="[....]/wsn-WS-BaseNotification-1.2-draft-01.wsdl"
      location="../wsrf/notification/WS-BaseN.wsdl"/>


<!-- Type Definitions -->
<!-- ************************************** -->
  <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://elverkemper.com/NotificationService">

      <include schemaLocation="./elverkemper-notification.xsd"/>
```

```
    </schema>
  </types>


<!-- Messages -->
<!-- ************************************** -->
  <message name="RegisterPublisherMessage">
    <part name="parameters" element="notif:RegisterPublisher"/>
  </message>

  <message name="RegisterPublisherResponseMessage">
    <part name="parameters"
          element="notif:RegisterPublisherResponse"/>
  </message>

  <message name="PublisherRegistrationFailedFaultMessage">
    <part name="parameters"
          element="notif:PublisherRegistrationFailedFault"/>
  </message>

  <!-- Non-WS subscriptions -->
  <message name="SubscribeNonWsMessage">
    <part name="parameters" element="notif:SubscribeNonWs"/>
  </message>
  <message name="SubscribeNonWsResponseMessage">
    <part name="parameters" element="notif:SubscribeNonWsResponse"/>
  </message>
  <message name="UnsubscribeNonWsMessage">
    <part name="parameters"
          element="notif:UnsubscribeNonWs"/>
  </message>
  <message name="UnsubscribeNonWsResponseMessage">
    <part name="parameters"
          element="notif:UnsubscribeNonWsResponse"/>
  </message>

  <!-- Filtering -->
  <message name="SetFilterMessage">
    <part name="parameters" element="notif:SetFilter"/>
  </message>
  <message name="SetFilterResponseMessage">
    <part name="parameters" element="notif:SetFilterResponse"/>
  </message>



<!-- Porttypes -->
<!-- ************************************** -->
  <portType name="NotificationService"
```

```
        wsrpw:ResourceProperties="notif:RpDoc"
        wsdlpp:extends="wsrpw:GetResourceProperty
                  wsrpw:GetMultipleResourceProperties
                  wsrpw:QueryResourceProperties
                  wsntw:NotificationProducer
                  wsrlw:ScheduledResourceTermination
                  wsrpw:SetResourceProperties">

      <operation name="RegisterPublisher">
          <input  message="notif:RegisterPublisherMessage"/>
          <output message="notif:RegisterPublisherResponseMessage"/>
          <fault
              message="notif:PublisherRegistrationFailedFaultMessage"
              name="PublisherRegistrationFailed"/>
      </operation>

      <!-- Subscription management for non-ws subscriptions -->
      <operation name="SubscribeNonWs">
          <input  message="notif:SubscribeNonWsMessage"/>
          <output message="notif:SubscribeNonWsResponseMessage"/>
      </operation>
      <operation name="UnsubscribeNonWs">
          <input  message="notif:UnsubscribeNonWsMessage"/>
          <output message="notif:UnsubscribeNonWsResponseMessage"/>
      </operation>

      <!-- Filtering expressions -->
      <operation name="SetFilter">
          <input  message="notif:SetFilterMessage"/>
          <output message="notif:SetFilterResponseMessage"/>
      </operation>
  </portType>
</definitions>
```

A.2 XML Type Definitions

Contents of the file `elverkemper-notification.xsd`:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://elverkemper.com/NotificationService"
    xmlns:notif="http://elverkemper.com/NotificationService"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    xmlns:wsnt="[....]/wsn-WS-BaseNotification-1.2-draft-01.xsd"
    xmlns:wsrf-bf="[....]/wsrf-WS-BaseFaults-1.2-draft-01.xsd">

<!-- Message Types of NotificationService -->
<!-- ************************************** -->
<element name="RegisterPublisher">
```

```
  <complexType>
    <sequence>
      <element name="PublisherReference"
        ref="wsa:EndpointReference" minOccurs="0" maxOccurs="1" />
      <element name="Topic" ref="wsnt:Topic" minOccurs="0"
        maxOccurs="unbounded" />
      <element name="Subscribe" ref="wsnt:Subscribe"
        minOccurs="1" maxOccurs="unbounded" />
      <element name="NonWsSubscriptions"
               type="notif:NonWsSubscriptionType"
        minOccurs="0" maxOccurs="unbounded" />
      <element name="Filter" ref="notif:SetFilter"
        minOccurs="0" maxOccurs="1"/>
      <element name="RegistrationName" type="string"
          minOccurs="0" maxOccurs="1"/>
    </sequence>
  </complexType>
</element>

<element name="RegisterPublisherResponse">
  <complexType>
    <sequence>
      <element name="PublisherRegistrationReference"
        ref="wsa:EndpointReference" minOccurs="1" maxOccurs="1" />
      <element name="SubscriptionReference"
        ref="wsa:EndpointReference" minOccurs="0" maxOccurs="1" />
    </sequence>
  </complexType>
</element>

<complexType name="RegisterPublisherFailedFaultType">
  <complexContent>
    <extension base="wsrf-bf:BaseFaultType"/>
  </complexContent>
</complexType>
<element name="RegisterPublisherFailedFault"
         type="notif:RegisterPublisherFailedFaultType"/>

<!-- Non ws subscription message types -->

<element name="UnsubscribeNonWs">
  <complexType>
    <sequence>
      <element name="Subscriptions"
               type="notif:NonWsSubscriptionType"
               minOccurs="1" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>
```

```
<element name="UnsubscribeNonWsResponse">
  <complexType/>
</element>

<element name="SubscribeNonWs">
  <complexType>
    <sequence>
      <element name="Subscriptions"
               type="notif:NonWsSubscriptionType"
               minOccurs="1" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>

<element name="SubscribeNonWsResponse">
  <complexType/>
</element>

<!-- Message types for filter definitions -->
<element name="SetFilter">
  <complexType>
    <sequence>
      <element name="InterestingValues" type="anyType"
         minOccurs="0" maxOccurs="unbounded"/>
      <element name="FilterRegExp" type="string"
           minOccurs="0" maxOccurs="1"/>
    </sequence>
  </complexType>
</element>
<element name="SetFilterResponse">
  <complexType/>
</element>

<!-- Fault Types                                  -->
<!-- ******************************************** -->
<complexType name="PublisherRegistrationFailedFaultType">
  <complexContent>
    <extension base="wsrf-bf:BaseFaultType"/>
  </complexContent>
</complexType>
<element name="PublisherRegistrationFailedFault"
         type="notif:PublisherRegistrationFailedFaultType"/>


<complexType name="ParameterFaultType">
  <complexContent>
    <extension base="wsrf-bf:BaseFaultType"/>
  </complexContent>
```

```
</complexType>
<element name="ParameterFault"
         type="notif:ParameterFaultType"/>

<complexType name="ProtocolNotSupportedFaultType">
  <complexContent>
    <extension base="wsrf-bf:BaseFaultType"/>
  </complexContent>
</complexType>
<element name="ProtocolNotSupportedFault"
         type="notif:ProtocolNotSupportedFaultType"/>

<complexType name="SubscriptionNotFoundType">
  <complexContent>
    <extension base="wsrf-bf:BaseFaultType"/>
  </complexContent>
</complexType>
<element name="SubscriptionNotFound"
         type="notif:SubscriptionNotFoundType"/>

<!-- Resource Properties of PublisherResource  -->
<!-- ****************************************** -->

<!-- Possible notifier protocols, remember to change when
   adding notifier components -->
<simpleType name="NotifierProtocols">
  <restriction base="string">
    <enumeration value="jabber"/>
    <enumeration value="email"/>
    <enumeration value="rss"/>
  </restriction>
</simpleType>

<complexType name="NonWsSubscriptionType">
  <sequence>
    <element name="protocol" type="notif:NotifierProtocols"/>
    <element name="address"  type="string"/>
  </sequence>
</complexType>

<element name="Message"            type="anyType" />
<element name="MessageLog"         type="anyType" />
<element name="RegistrationName"   type="string"  />
<element name="PublisherReference" type="wsa:EndpointReferenceType"/>
<element name="NonWsSubscriptions" type="notif:NonWsSubscriptionType"/>
<element name="FilterRegExp"       type="string"  />
<element name="InterestingValues"  type="anyType"  />
<element name="PublisherTopic"     type="wsnt:TopicExpressionType" />
```

```
<element name="RpDoc">
  <complexType>
    <sequence>
      <element ref="notif:Message"
               minOccurs="0" maxOccurs="1"/>
      <element ref="notif:MessageLog"
               minOccurs="0" maxOccurs="unbounded"/>
      <element ref="notif:RegistrationName"
               minOccurs="0" maxOccurs="1"/>
      <element ref="notif:PublisherReference"
               minOccurs="1" maxOccurs="1"/>
      <element ref="notif:NonWsSubscriptions"
               minOccurs="0" maxOccurs="unbounded"/>
      <element ref="notif:FilterRegExp"
               minOccurs="0" maxOccurs="1"/>
      <element ref="notif:InterestingValues"
               minOccurs="0" maxOccurs="unbounded"/>
      <element ref="notif:PublisherTopic"
               minOccurs="1" maxOccurs="1"/>
    </sequence>
  </complexType>
</element>
</schema>
```