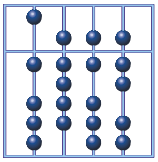


Lehrstuhl für Datenbanksysteme
Fakultät für Informatik
Technische Universität München

Metadata Management and Context-based Personalization in Distributed Information Systems

Dipl.-Inf. Univ.
Markus Keidl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Florian Matthes

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph. D.
2. Univ.-Prof. Dr.-Ing. Klemens Böhm
   Universität Karlsruhe (TH)

Die Dissertation wurde am 23.06.2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.09.2004 angenommen.

Abstract

Nowadays, the Internet provides a large number of resources such as data sources, computing power, or applications that can be utilized by data integration systems. For an efficient usage and management of these resources, data integration systems require an efficient access to extensive metadata. Within the scope of the ObjectGlobe system, our open and distributed query processing system for data processing services on the Internet, we therefore developed the MDV system, a distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier. It implements a novel, DBMS-based publish & subscribe algorithm to keep replicas up-to-date and initiate the replication of new and relevant information.

Many companies are not only interested in data integration but in application integration for cost-cutting reasons. In recent years, Web services have become a new and promising technology for application integration on the Internet. Within the scope of the ServiceGlobe system, our open and distributed Web service platform, we developed techniques that facilitate the development of Web services with regard to flexibility, reliability, and personalization. Dynamic service selection offers the possibility of selecting and invoking services at runtime based on a technical specification of the desired service. Thus, it provides a layer of abstraction from the actual services. Our context framework facilitates the development and deployment of context-aware adaptable Web services. Web services are provided with context information about clients that may be utilized to provide personalized behavior. Context is extensible with new types of information at any time without any changes to the underlying infrastructure.

Acknowledgments

First of all, I would like to thank my parents for their support and encouragement. They gave me the opportunity to study, and they never had any doubt that I would succeed.

My doctoral thesis was done in the context of the ObjectGlobe and the ServiceGlobe project. Many people contributed to these projects, and it is impossible to list them all here. In particular, I like to thank my colleagues Reinhard Braumandl, Konrad Stocker, Christian Wiesner, and Bernhard Stegmaier. All of them contributed various important parts to these projects. Without them, my work would not have been possible.

I also thank the following students for their work on the implementation of the ObjectGlobe and ServiceGlobe system as part of their diploma theses and programming courses: Alexander Kreutz, Franz Häuslschmid, Christof König, and Michael Denk.

For helpful criticism and advice on my doctoral thesis, I express my thanks to Stefan Seltzsam, Christian Wiesner, and Bernhard Stegmaier. I appreciate all their valuable suggestions.

I would also like to thank Natalija Krivokapić and Stefan Seltzsam. Natalija was the advisor for my master thesis, and I learned a lot from her insight and experience in doing research work. Stefan and I shared an office for several years. We always had a great and inspiring working atmosphere.

Special thanks go to the rest of the coffee gang. Drinking my - as you always said - huge cups of coffee would not have been half the fun without you all.

Last, but not least, I thank my supervisor Professor Alfons Kemper.

München, November 2004 Markus Keidl

Contents

List of Figures

Chapter 1

Introduction

The emergence of the Internet imposed new challenges to computer systems in the area of information systems and databases. The World Wide Web, for example, has made it very easy for people and organizations all over the world to publish their data as Web pages, as documents, or by establishing interfaces to the databases containing the data. Thus, data integration efforts started soon in order to provide a unified view over the various data sources and to achieve consistency across these data sources [Sto99]. In addition to these data sources, the Internet provides further resources such as computing power or applications that can be utilized by data integration systems. For example, the computing power of computers in the Internet can be used for data processing, similar to the SETI@home project [ACK+02] that utilizes a large number of Internet-connected computers to analyze radio telescope signals. Because of the increasing number of applications available on the Internet, many companies are nowadays not only interested in data integration but in application integration for cost-cutting reasons.

In recent years, Web services have become a new and promising technology for application integration on the Internet. A key feature of Web services is interoperability, that is, Web services are usable without any knowledge about their underlying operating system or the programming language used for their development. Together with the world-wide interconnection of computer systems, Web services technology makes enterprise application integration feasible. Although Web services offer solutions to numerous integration problems, application integration is still a cumbersome task as these applications were not designed for interoperability.

One of the problems in application integration is flexibility and reliability of Web services, which are important in a dynamic environment like the Internet. Without precautions, Web services could easily fail because a Web service on which they depend is unavailable. Another problem is the large number of heterogeneous consumer groups using Web services. Today, consumers want to use several ways to access information systems on the Internet, for instance, browsers on desktop computers, PDAs, or cell phones. As the trend to an increasing number of ubiquitous, connected devices—called pervasive computing—continues to grow, the heterogeneity of client capabilities and the number of methods for accessing Web services increases continually. Consequently, Web services are expected

to respect the needs, the preferences, and the current environment of their consumers to provide them with customized and personalized behavior.

1.1 Purpose of this Thesis

In this thesis, we address the challenges for data integration systems imposed by the large number of resources available on the Internet. We also introduce techniques that facilitate application integration based on Web services. These techniques support the development of Web services with regard to flexibility, reliability, and personalization.

The ObjectGlobe system [BKK+01a] is an example of an open and distributed query processing system for data processing services on the Internet. The goal of the ObjectGlobe project is to create an infrastructure that makes it as easy to distribute query processing capabilities, including those found in traditional database management systems (DBMSs), across the Internet as it is to publish data and documents on the Web today. The idea is to create an open marketplace for three kinds of suppliers: data providers which supply data, function providers which offer query operators to process the data, and cycle providers which are contracted to execute query operators.

Distributed information systems like ObjectGlobe require an efficient, distributed management of the available resources, that is, they require extensive metadata for the description, discovery, and administration of these resources. For example, ObjectGlobe has a metadata repository that registers all data, function, and cycle providers on which queries can be executed. Every time a new provider joins or leaves the ObjectGlobe federation, the corresponding metadata is added to or removed from the respective metadata repository. The metadata repository is used by the ObjectGlobe optimizer in order to discover relevant resources for a query. Due to the nature of the Internet, this metadata changes rapidly. Furthermore, such information must be available for a large number of consumers and Web services, and copies of pieces of information should be stored near the consumers that need this particular information.

In this thesis, we present the MDV ystem, a distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that queries can be evaluated locally. Thus, no expensive communication across the Internet is necessary. Users and applications specify the information they want to be replicated using a specialized subscription language. This reduces the amount of data that has to be queried locally resulting in a better query execution performance. MDV implements a novel, scalable, DBMS-based publish & subscribe algorithm to keep replicas up-to-date and initiate the replication of new and relevant information.

We further address the usage of the Web services technology as a new and promising technology for enterprise application integration on the Internet. As application integration is still a difficult task, because most applications were not designed with interoperability in mind, there are open problems regarding the development of flexible, reliable, and personalizable Web services.

Therefore, we developed a technique called dynamic service selection, which offers the

possibility of selecting and invoking services at runtime based on a technical specification of the desired service. Thus, it provides a layer of abstraction from the actual services. Constraints enable users and Web services to influence dynamic service selection, for example, services can be selected based on the relevant metadata.

The heterogeneous consumer groups on the Internet as well as the trend towards pervasive computing requires Web services that understand and respect the needs, the preferences, and the current environment of their consumers. Generally, all this information about a consumer is called context. More precisely, in our work context constitutes information about consumers and their environment that may be used by Web services to provide consumers with a customized and personalized behavior. Web services should use such context information to adjust their internal control flow as well as the content and format of their replies.

We developed a context framework that facilitates the development and deployment of context-aware adaptable Web services. The framework consists of two main parts: a distributed infrastructure, which transmits context between clients and Web services and which manages the context processing, and the context types, which are the supported types of context information and which are extensible at any time. The actual context processing is done by three components: Web services themselves, context plugins, and context services. Context plugins and context services are provided by the context framework, and they pre- and postprocess Web service messages according to the available context information. Both components are essential for automatic context processing, that is, for processing context without the support of Web services, and for automatic adaption of Web services to new context types. We also provide means for controlling the way context is processed. Context processing instructions can be used to specify hosts to which context is transmitted and at which hosts and by which components it is actually processed.

We present dynamic service selection and our context framework within the scope of the ServiceGlobe system [KSK03a, KSK02], our open and distributed Web service platform. ServiceGlobe provides a platform on which services can be implemented, stored, published, discovered, and deployed. The ServiceGlobe system is fully implemented in Java and based on standards like XML, SOAP, UDDI, and WSDL. ServiceGlobe supports mobile code, that is, Web services can be distributed on demand and instantiated during runtime at arbitrary Internet servers participating in the ServiceGlobe federation. Also, it offers all standard functionality of a service platform like SOAP communication, a transaction system, and a security system [SBK01].

The ServiceGlobe project is a successor of our ObjectGlobe project. It transfers ObjectGlobe's mobile query processing capabilities into the area of Web services. Whereas ObjectGlobe provides an infrastructure to distribute query processing operators across the Internet to execute them close to the data they process, the goal of ServiceGlobe is to create an infrastructure to distribute Web services across the Internet to execute them close to data or other Web services which they require during their execution. As additional optimization, Web services can be instantiated on machines having the optimal execution environment, e.g., a fast processor, huge memory, or a high-speed network connection.

1.2 Outline of this Work

The remainder of this thesis is organized as follows:

- Chapter 2 gives an overview of the ObjectGlobe system, our open and distributed query processing system for data processing services on the Internet. The MDV system was developed as part of the ObjectGlobe system. Therefore, we use ObjectGlobe as an example client of MDV.
- Chapter 3 presents the MDV system, its architecture, and core components. We also describe our publish & subscribe algorithm and several performance experiments conducted using our prototype implementation.
- Chapter 4 describes the deployment of MDV within the ObjectGlobe system. The MDV system is used as a lookup service for resources and as (distributed) data storage for security-related data by the security system of ObjectGlobe.
- Chapter 5 presents the ServiceGlobe system, our open and distributed Web service platform. We also give a short introduction into Web service standards that are important in our work.
- Chapter 6 describes dynamic service selection, a technique that we realized within the ServiceGlobe system. It provides a layer of abstraction for service invocation offering Web services the possibility of selecting and invoking services at runtime based on a technical specification of the desired service.
- Chapter 7 presents our context framework, which we also implemented within the ServiceGlobe system. It facilitates the development and deployment of context-aware adaptable Web services.
- Chapter 8 summarizes this thesis and gives an outline of possible future research.

Chapter 2

ObjectGlobe - Open and Distributed Query Processing

In this chapter, we present the design of ObjectGlobe, our open and distributed query processor for Internet data sources. The goal of the ObjectGlobe project is to distribute powerful query processing capabilities (including those found in traditional database management systems) across the Internet. The idea is to create an open marketplace for three kinds of suppliers: *data providers* which supply data, *function providers* which offer query operators to process the data, and *cycle providers* which are contracted to execute query operators. Of course, a single site (even a single machine) can comprise all three services, i.e., act as data, function, and cycle provider. In fact, we expect that most data and function providers will also act as cycle providers. ObjectGlobe enables applications to execute complex queries involving the execution of operators from multiple function providers at different sites (cycle providers) and the retrieval of data and documents from multiple data sources. A detailed description of the ObjectGlobe system is given in [BKK+01a, BKK+01b, BKK+00, BKK+99]. The HyperQuery project, an extension of ObjectGlobe for implementing scalable electronic marketplaces, is described in [KW04, KW01].

The remainder of this chapter is structured as follows: In Section 2.1, we outline how queries are processed in ObjectGlobe. We give an example and present the basic features of our system. The lookup service of ObjectGlobe and its relationship with the query optimizer are described in Section 2.2. Finally, we outline ObjectGlobe's quality of service management in Section 2.3 and discuss the security requirements of ObjectGlobe in Section 2.4.

2.1 Query Processing in ObjectGlobe

Processing a query in ObjectGlobe involves four major steps, as shown in Figure 2.1:

- **Lookup:** In this phase, the ObjectGlobe lookup service is queried to find relevant data sources, cycle providers, and query operators that might be useful executing

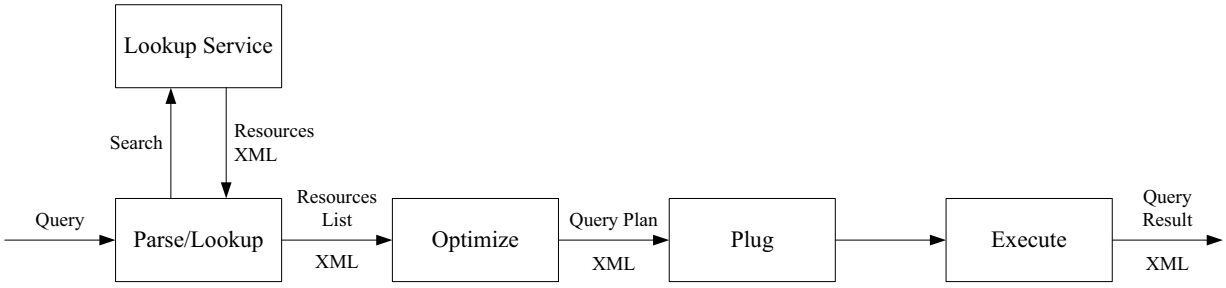


Figure 2.1: Processing a Query in ObjectGlobe

the query. In addition, the lookup service provides the authorization data—mirrored and integrated from the individual providers—to determine what resources may be accessed by the user who initiated the query and what other restrictions apply for processing the query.

- **Optimize:** The information obtained from the lookup service is used by a quality-aware query optimizer to compile a valid (as far as user privileges are concerned) query execution plan believed to fulfill the users' quality constraints. This plan is annotated with site information indicating on which cycle provider each operator is executed and from which function provider the external query operators involved in the plan are loaded.

- **Plug:** The generated plan is distributed to the cycle providers, and the external query operators are loaded and instantiated at each cycle provider. Furthermore, the communication paths (sockets) are established.

- **Execute:** The plan is executed following an iterator model [Gra93]. In addition to the *external* query operators provided by function providers, ObjectGlobe has *built-in* query operators for selection, projection, join, union, nesting, unnesting, sending data, and receiving data. If necessary, communication is encrypted and authenticated. Furthermore, the execution of the plan is monitored in order to detect failures, look for alternatives, and possibly halt the execution of a plan.

The whole system is written in Java for two reasons. First, Java is portable so that ObjectGlobe can be installed with very little effort on various platforms; in particular, cycle providers, which need to install the ObjectGlobe core functionality, can very easily join an ObjectGlobe system. The only requirement is that a site runs the ObjectGlobe server on a Java virtual machine. Second, Java provides secure extensibility. Like ObjectGlobe itself, external query operators are written in Java: They are loaded on demand (from function providers), and they are executed at cycle providers in their own Java "sandbox". Just like data and cycle providers, function providers and their external query operators must be registered in the lookup service before they can be used.

ObjectGlobe supports a nested relational data model in order for relational, object-relational, and XML data sources to be easily integrated. Other data formats, e.g., HTML,

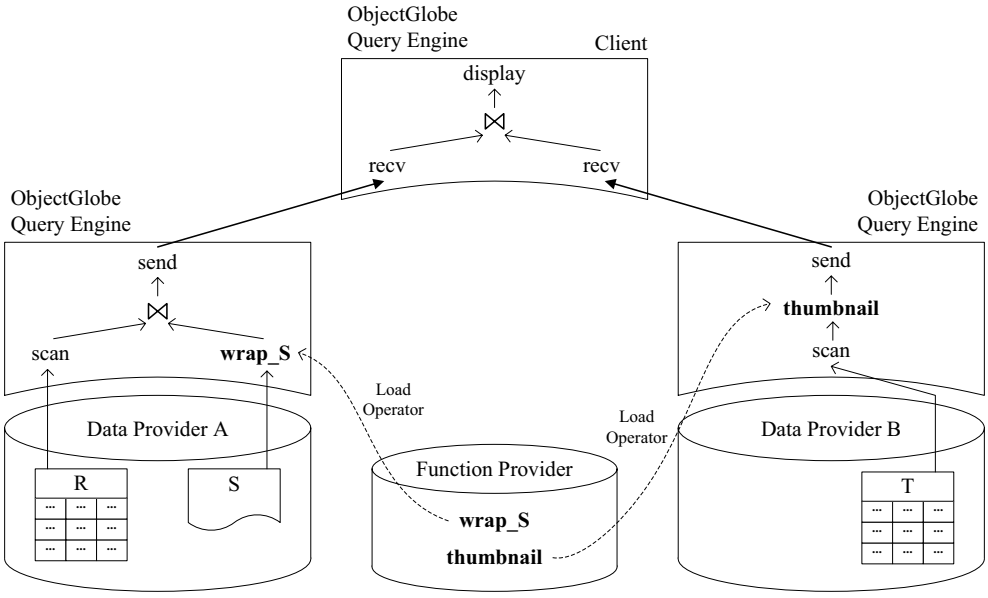


Figure 2.2: Distributed Query Processing with ObjectGlobe

however, can be integrated by the use of wrappers that transform the data into the required nested relational format. Wrappers are treated by the system as external query operators. As shown in Figure 2.1, XML is used as data exchange format between the individual ObjectGlobe components. Part of the ObjectGlobe philosophy is that the individual ObjectGlobe components can be used separately. XML is used so that the output of every component can be easily visualized and modified. For example, users can browse through the lookup service in order to find interesting functions that they might want to use in the query. Furthermore, a user can look at and change the plan generated by the optimizer.

To illustrate query processing in ObjectGlobe, let us consider the example shown in Figure 2.2. In this example, there are two data providers, A and B, and one function provider. We assume that the data providers also operate as cycle providers so that the ObjectGlobe system is installed on the machines of A and B. Furthermore, the client can act as a cycle provider in this example. Data provider A supplies two data collections: a relational table R and some other collection S which needs to be transformed (wrapped) for query processing. Data provider B has a (nested) relational table T. The function provider supplies two relevant query operators: a wrapper (*wrap_S*) to transform S into nested relational format and a compression algorithm (*thumbnail*) to apply on an image attribute of T.

2.2 Lookup Service and Optimization

The lookup service in ObjectGlobe plays the same role as the catalog or metadata management of a traditional query processor. Providers are registered before they can participate

in ObjectGlobe. In this way, the information about available services is incrementally extended as necessary. A similar approach for integrating various business services in business-to-business (B2B) e-commerce has been proposed in the UDDI standardization effort [UDD00].

The main challenge of the lookup service is to provide global access to the metadata of all registered providers, but also to allow a selective grouping of somehow related metadata in order to reduce the effort to query the metadata. Thus, the ObjectGlobe system uses the metadata management system MDV as its lookup service. MDV uses a distributed 3-tier architecture. Metadata, e.g., about providers and their services, must be specified as RDF documents [LS99]. A detailed description of the MDV system is given in Chapter 3. Information about the deployment of the MDV system within ObjectGlobe is presented in Chapter 4.

The metadata in the providers' RDF documents must conform to our meta-schema which defines the structures of the service descriptions of each kind of provider. These descriptions are rather detailed since they are the only information about services the optimizer gets in order to construct a valid and quality-aware query evaluation plan. For example, for each external query operator offered by a function provider, metadata about its name, category, signature, and cost model must be available. The optimizer also needs performance characteristics of cycle providers as well as schema information and statistics for data collections of data providers. Furthermore, authorization data for the services is used to construct compatibility matrices during the optimization process which represent the information about legal combinations of the services possibly involved in the query execution at a specific position in the query evaluation plan. Due to authorization constraints, our optimizer might not be able to find a query evaluation plan although necessary services could be retrieved from the lookup service.

We expect the registration of providers' services to become a similar market as the market for the providers themselves. So, someone interested in using a service will register this service; service providers themselves need not necessarily do this on their own. For example, wrapper developers are of course interested in registering data sources for which they have written the corresponding wrappers. Such an incremental schema enhancement by an authorized user is possible in the ObjectGlobe lookup service just as in any other database system. This means that an ObjectGlobe system normally is not tailored for a specific data integration problem, but can be extended dynamically with new data, cycle, and function providers by augmenting the metadata of its lookup service.

The ObjectGlobe optimizer consults the lookup service in order to find relevant resources to execute a query and obtain statistics. It enumerates alternative query evaluation plans using a System-R style dynamic programming algorithm, that is, the optimizer builds plans in a bottom-up way: First, so-called access plans are constructed that specify how each collection is read (i.e., at which cycle provider and with which scan or wrapper operator). After that, join plans are constructed from these access plans and (later) from simpler join plans. Evidently, the search space can become too large for full dynamic programming to work for complex ObjectGlobe queries. To deal with such queries, we developed another extension that we call iterative dynamic programming (IDP for short).

IDP is adaptive; it starts like dynamic programming, and if the query is simple enough, then IDP behaves exactly like dynamic programming. If the query turns out to be too complex, then IDP applies heuristics in order to find an acceptable plan. Details and a complete analysis of IDP are given in [KS00].

2.3 Quality of Service (QoS)

Although the example in Section 2.1 is rather small (in order to be illustrative), we expect ObjectGlobe systems to comprise a large number of cycle providers and data providers. For example, think of an ObjectGlobe federation that incorporates the online databases of several real estate brokers. A traditional optimizer would produce a plan for a query in this federation that reads all the relevant data (i.e., considers all real-estate data providers). Therefore, the plan produced by a traditional optimizer will consume much more time and money than an ObjectGlobe user is willing to spend. In such an open query processing system, it is essential that a user can specify quality constraints on the execution itself. These constraints can be separated into three different dimensions:

- **Result:** Users may want to restrict the size of the result sets returned by their queries in the form of lower or upper bounds (an upper bound corresponds to a stop after query [CK98]). Constraints on the amount of data used for answering the query (e.g., at least 50% of the data registered for the theme *Real Estate* should be used for a specific query) and its freshness (e.g., the last update should have happened within the last day) can be used to get results that are based on a current and sufficiently large subset of the available data.

- **Cost:** Since providers can charge for their services in our scenario, a user should be able to specify an upper bound for the respective consumption by a query.

- **Time:** The response time is another important quality parameter of an interactive query execution. A user can be interested in a fast production of the first answer tuples or in a fast overall execution of the query. A fast production of the first tuples can be important so that the user can look at these tuples while the remainder is computed in the background.

In many cases, not all quality parameters will be interesting. Just like in real-time systems, some constraints could be strict (or hard) and others could be soft and handled in a relaxed way. A detailed description of the QoS management in ObjectGlobe as well as a comparison with existing approaches are given in [BKK03, Bra01a].

The starting point for query processing in our system is a description of the query itself, the QoS constraints for it, and statistics about the resources (providers and communication links). QoS constraints will be treated during all phases of query processing. First, the optimizer generates a query evaluation plan whose estimated quality parameters are believed to fulfill the user-specified quality constraints of the query. For every sub-plan, the

optimizer states the minimum quality constraints it must obey in order to fulfill the overall quality estimations of the chosen plan and the resource requirements deemed necessary to produce these quality constraints. If, during the plug phase, the resource requirements cannot be satisfied with the available resources, the plan is adapted or aborted. The QoS management reacts in the same way if, during query execution, the monitoring component forecasts an eventual violation of the QoS constraints.

2.4 Security and Privacy Issues

Obviously, security is crucial to the success of an open and distributed system like ObjectGlobe. Dependent on the point of view, different security interests are important. On the one hand, cycle and data providers need a powerful security system to protect their resources against unauthorized access and attacks of malicious external operators. Apart from that, cycle and data providers might have a legitimate interest in the identity of users for authorization issues. Users of ObjectGlobe on the other hand want to feel certain about the semantics of external operators to rely upon the results of a query. For that purpose, it is also necessary to protect communication channels against tampering. Another interest of users is privacy, i.e., other parties must not be able to read confidential data. Furthermore, users normally want to stay anonymous as far as possible. Below we sketch our conception of the security system of ObjectGlobe. The security measures are classified by the time of application.

2.4.1 Preventive Measures

Preventive measures take place before an operator is actually used for queries and include checking of the results produced by the operator in test runs, stress testing, and validation of the cost model. These checkups are done by a trustworthy third party which generates a digitally signed document containing the diagnosis for the tested operator. To support the checkups, we developed a validation server that semi-automatically generates test data, runs the operator, and compares the results generated by the operator with results acquired from an executable formal specification or a reference implementation of the operator. Additionally, the validation server ensures that execution costs are within the limits given by the cost model of the operator.

Preventive measures should increase the trust in the non-malicious behavior of external operators. They are optional in ObjectGlobe, but users with a high demand of security will exclusively use certified external operators to ensure that all operators will calculate the result of the query according to the given semantics.

2.4.2 Checks during Plan Distribution

Three security related actions take place during plan distribution: setup of secure communication channels, authentication, and authorization.

ObjectGlobe is using the well-established secure communication standards SSL (Secure Sockets Layer) [FKK96] and/or TLS (Transport Layer Security) [DA99] for encrypting and authenticating (digitally signing) messages. Both protocols can carry out the authentication of ObjectGlobe communication partners via X.509 certificates [HFPS99]. If users digitally sign plans, such certificates are used for authentication of users, too. Additionally, ObjectGlobe supports the embedding of encrypted passwords into query plans which can be used by wrappers to access legacy systems using password-based authentication. Of course, users can stay anonymous when they use publicly available resources.

Based on the identity of a user, a provider can autonomously decide whether a user is authorized to, e.g., execute operators, access data, or load external operators. Thus, providers can (but need not) constrain the access or use of their resources to particular user groups. Additionally, they can constrain the information (respectively function code) flow to ensure that only trusted cycle providers are used during query execution. In order to generate valid query execution plans and avoid authorization failures at execution time, the authorization constraints are integrated into the lookup service of ObjectGlobe. For a more detailed description of authentication and authorization in ObjectGlobe, see Section 4.2.

2.4.3 Runtime Measures

To prevent malicious actions of external operators, ObjectGlobe is based on Java's security infrastructure to isolate external operators by executing them in protected areas, so-called "sandboxes". As a result, cycle providers can prohibit external operators from accessing crucial resources, e.g., the filesystem or network sockets. External operators are also prevented from leaking confidential data through, for instance, network connections. Additionally, a runtime monitoring component can react on denial of service attacks. Therefore, the monitoring component evaluates cost models of operators and supervises resource consumption (e.g., memory usage and processor cycles). When an operator uses more resources than the cost model predicted, it is aborted. A detailed description of these issues is given in [SBK01].

Chapter 3

MDV - Distributed Metadata Management

In this chapter, we present the MDV system, our distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that queries can be evaluated locally. Users and applications specify the information they want to be replicated using a specialized subscription language. In order to keep replicas up-to-date and initiate the replication of new and relevant information, MDV implements a novel, scalable publish & subscribe algorithm. We describe this algorithm in detail, show how it can be implemented using a standard relational database management system, and present the results of performance experiments conducted using our prototype implementation. Parts of this chapter have already been presented in [KKKK02b, KKKK02a, KKKK01].

The remainder of this chapter is structured as follows: Section 3.1 motivates the necessity of distributed metadata management for services and applications on the Internet. Section 3.2 presents the MDV system, its architecture, and core components. Section 3.3 describes our publish & subscribe algorithm, particularly the filter algorithm. Performance experiments conducted using our prototype implementation are presented in Section 3.4. Finally, Section 3.5 discusses related work.

3.1 Motivation

Nowadays, the Web is one of the main driving forces behind the development of new and innovative applications. The emergence of electronic marketplaces and other electronic services and applications on the Internet is creating a growing demand for effective management of resources. Dynamic composition of such services requires extensive metadata for the description, administration, and discovery of these services. Due to the nature of the Internet, such information changes rapidly. Furthermore, such information must be available for a large number of users and applications, and copies of pieces of information should be stored near the users that need this particular information. Thus, metadata

about such resources and services is a key to the success of these services.

In this chapter, we present the architecture of the MDV system, a distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that queries can be evaluated locally. Thus, no expensive communication across the Internet is necessary. This supports fast discovery of metadata which is, e.g., necessary in Web service composition or query optimization. Users and applications specify the information they want to be replicated using a specialized subscription language. This reduces the amount of data that has to be queried locally resulting in a better query execution performance.

MDV implements a novel, scalable publish & subscribe algorithm to keep replicas up-to-date and initiate the replication of new and relevant information. We describe this algorithm in detail, especially how it deals with the possibly huge set of subscription rules. We show how the algorithm can be implemented using a standard relational database management system (RDBMS) thereby taking advantage of their matured storing, indexing, and querying abilities. Although the algorithm is described in the context of RDF [LS99] and MDV's subscription language, it is also applicable to XML [BPSM+04] and the XQuery language [BCF+03]. The MDV system was developed as part of our ObjectGlobe project, an open and distributed query processing system for data processing services on the Internet. We therefore use ObjectGlobe as an example client of MDV.

3.2 Overview of the MDV System

In this section, we describe the architecture of the MDV metadata management system and give a general overview of its components. The main features of our system are a 3-tier architecture that eases the adjustment to varying workloads and activity hot spots, efficient metadata access using caching, and a publish & subscribe mechanism for preserving cache consistency.

The MDV system is intended for managing metadata about resources on the Internet. There are already established Internet standards, developed by the W3C, for describing and interchanging such metadata: RDF (Resource Description Framework) [LS99] to represent metadata about Web resources and RDF Schema [BG99] to define the schema the RDF metadata must conform to. Consequently, we adopted these standards. As RDF was specifically developed for describing metadata, it has some advantages over XML in this area [Bra01b]. For the MDV system, advantages are, e.g, that every resource has an unique URI (Uniform Resource Identifier) specified by the attribute `rdf:ID`, that resources can reference one another, and that RDF provides a graph-based data model including an XML-based syntax for serializing RDF metadata.

3.2.1 Example

Figure 3.1 shows an excerpt from an RDF document `doc.rdf`. The excerpt defines two resources: a `CycleProvider` and a `ServerInformation`. The former is a server on the In-

```
<CycleProvider rdf:ID="host">
  <serverHost>pirates.uni-passau.de</serverHost>
  <serverPort>5874</serverPort>
  <serverInformation>
    <ServerInformation rdf:ID="info" memory="92" cpu="600"/>
  </serverInformation>
</CycleProvider>
```

Figure 3.1: Excerpt from an MDV RDF Document

ternet capable of executing arbitrary ObjectGlobe services, the latter contains information about the computer running the provider. The `rdf:ID` property defines a local identifier for each resource, `host` and `info` in the example. A unique identifier, called *URI reference*, is constructed by combining the local identifier of a resource with the (globally unique) URI associated with an RDF document. The CycleProvider resource contains three further properties: `serverHost` which contains the server's DNS name, `serverPort` which contains the provider's port, and `serverInformation` which is a reference to the ServerInformation resource storing data about the computer running the provider. It contains the size of the computer's main memory in MB (property `memory`) and the speed of its CPU in MHz (property `cpu`). Properties (like serverInformation) always reference resources using their URI reference, i.e., RDF does not distinguish between nested and referenced resources. So it is irrelevant if resources are defined as nested elements (as in Figure 3.1) or somewhere else in the same or even in another document.

3.2.2 Architecture Overview

Figure 3.2 depicts the MDV system's 3-tier architecture, consisting of metadata providers, local metadata repositories, and MDV clients.

Metadata providers (MDPs), referred to as *(MDV) backbone*, are distributed all over the Internet to provide a uniform access regarding network latency and metadata content. MDPs accomplish the latter by sharing the same schema and consistently replicating metadata among each other. Basically, the backbone is an extension of a distributed DBMS with a flat hierarchy, full synchronization, and replication.[1] All metadata stored at MDPs is regarded as global and publicly available.

Local metadata repositories (LMRs) are the components of the MDV system that do the actual metadata query processing. For efficiency reasons, i.e., to avoid communication across the Internet, LMRs cache global metadata and use only locally available metadata for query processing. Consequently, LMRs should be running close to the applications querying the metadata, e.g., in the same LAN. The cache of an LMR should contain

[1] A more sophisticated distributed architecture regarding partitioning and replication is possible in the backbone. But this is not the focus of our work. Work on partitioning and replication for distributed database management systems is, e.g., described in [ÖV99].

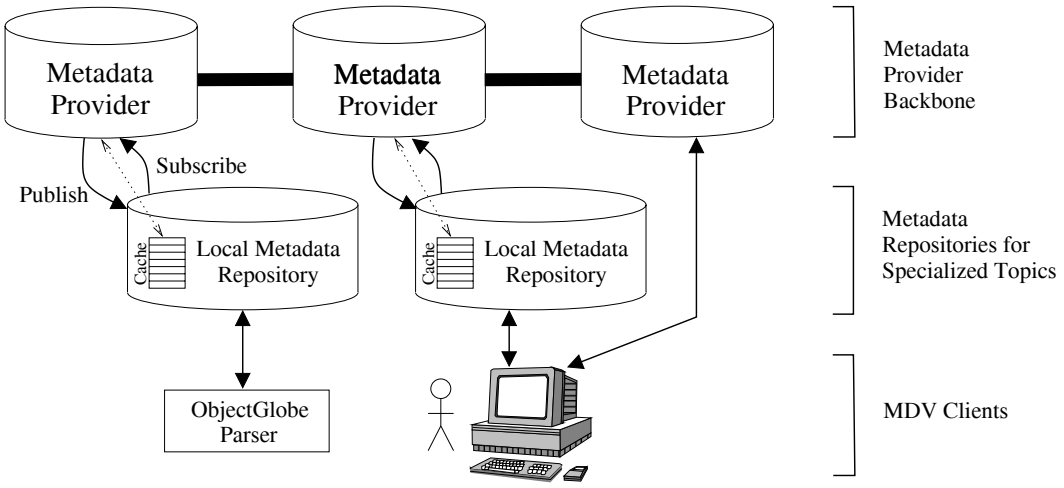


Figure 3.2: Overview of MDV's Architecture

relevant metadata, appropriate to the users or applications using it. LMRs use a publish & subscribe mechanism to fetch relevant metadata from an MDP and to receive updates to their data, that is, to keep their caches consistent. When *subscribing* to an MDP, an LMR registers a set of subscription rules that define the parts of the MDP's metadata it wants to cache. MDPs use subscription rules to *publish* updates, insertions, or deletions in the metadata to LMRs. In addition to global metadata, LMRs store local metadata that should not be accessible to the public and therefore is not forwarded to the backbone. Local metadata must be explicitly marked as such at registration time.

Applications and users accessing the MDV system are referred to as *MDV clients*. MDV clients can query metadata at an LMR using MDV's (declarative) query language, which is quite similar to the rule language that is explained in Section 3.2.3. Basically, the only difference is that the keyword `select` is used instead of the keyword `register`. Real users can also browse metadata at an MDP (as depicted in Figure 3.2) and select it for caching. Their LMR will generate appropriate rules and update its set of subscription rules.

Metadata administration, i.e., registering new metadata and updating or deleting old metadata, is done at MDPs. New metadata must be registered within a valid RDF document; updating metadata essentially means re-registering a modified version of an already registered document. Deleting can be done either by removing parts from a document and updating it or by removing the complete document with all its content. This is the only way to add, update, or delete metadata. MDV's query language does not provide any update or delete functionality.

MDV is implemented in Java so that it is portable which allows installation with very little effort, and it uses a relational database management system as basic data storage. RDF documents are mapped to tables as described in [FK99]. Search requests are translated into SQL join queries. This translation is not one-to-one as MDV hides the details of how the metadata is stored. Translating search requests into SQL queries is quite com-

plicated (albeit straightforward) and describing all the details is beyond the scope of this thesis.

3.2.3 Rule System

Subscription rules are specified by users browsing and selecting metadata or by administrators of LMRs. Rules must describe the kind of metadata that local MDV clients are interested in because only metadata matching these rules is cached locally and used for metadata query evaluation. Currently, MDV uses a proprietary rule language which supports path expressions and joins. A rule is (informally) defined using the following SQL-like syntax:

`search` $Extension_1\ e_1, Extension_2\ e_2, \ldots, Extension_n\ e_n$
`register` e_i
`where` $Predicates(e_1, e_2, \ldots, e_n)$

This rule matches or registers all resources e_i that are an element of extension $Extension_i$ and that satisfy the rule's `where` part ($i \in \{1, \ldots, n\}$). Every extension $Extension_k$ for $k = 1, \ldots, n$ is either some class defined in the schema or another subscription rule. *Predicates* is a conjunction of elementary predicates of the form $X \circ Y$ with X and Y either constants or valid path expressions (according to the schema) and $\circ \in \{$`=`, `!=`, `<`, `<=`, `>`, `>=`, `contains`$\}$. MDV provides a special any operator "?" that can be applied to set-valued properties.[2] Currently, our implementation does not support an `or` operator, but rules containing this operator can be easily transformed into rules without it using the algebra of logic and negated operators. *Predicates* can also contain join predicates, i.e., predicates that join two extensions by referring to different extensions in X and Y. For an example, see the subscription rule in Section 3.3.3.

The following rule subscribes to all resources that are an instance of class CycleProvider, which must be defined in the schema, and that have a property serverHost that contains 'uni-passsau.de' and a serverInformation property that references a ServerInformation resource with a memory property value greater than 64:

```
search CycleProvider c
register c
where c.serverHost contains 'uni-passau.de' and
      c.serverInformation.memory > 64
```

So, this rule subscribes to all cycle providers that run on computers in the 'uni-passau.de' domain with more than 64 MB of main memory. For example, the CycleProvider resource defined in the document excerpt of Figure 3.1 matches this rule.

[2]With set-valued properties, all set elements must have the same type even though RDF does not formally enforce this.

3.2.4 References

The previous example shows one problem: The rule, applied to the metadata of Figure 3.1, will register the CycleProvider resource with reference `host` and transmit it to an LMR. But obviously, the referenced ServerInformation resource must be transmitted, too. Otherwise, the CycleProvider resource will contain a dangling reference. There are three possible solutions to deal with referenced resources: a) never transmit them with a resource, b) follow all references until no new references are found (i.e., calculating the closure), or c) do something in between. The first two solutions both have major drawbacks, ranging from dangling references to a possible transmission of the complete database. Therefore, we introduced *strong* and *weak* references. Resources referenced by strong references are always transmitted together with the referencing resource whereas resources referenced by weak references are never transmitted. MDV augments RDF schema with the necessary RDF properties to allow the definition of strong and weak references.

Currently, the designer of the metadata schema is responsible for defining strong and weak references as the decision which references are strong and which are weak is part of the schema design. In the following, we sketch a solution that determines strong and weak references automatically by analyzing the queries stated at the LMRs. For example, assume the following query:

```
search CycleProvider c
select c
where c.serverHost contains 'uni-passau.de' and c.serverInformation.memory > 64
```

At an LMR, this query can only be answered if its cache contains all necessary CycleProvider and ServerInformation resources. Notice that the ServerInformation resources are necessary because of the serverInformation property restriction. All necessary resources are in the LMR's cache either if they are registered directly by appropriate subscription rules or if the necessary CycleProvider resources are registered directly and the serverInformation property is referenced using a strong reference. In conclusion, a query cannot be answered at an LMR if it contains a predicate with a path expression that contains a weak reference and if there is no subscription rule that registers the resources referenced by this weak reference directly. Obviously, such a weak reference is a candidate for changing it into a strong reference.

Queries at LMRs can be analyzed to detect this kind of weak references. If the same weak reference is a strong reference candidate at several LMRs, the metadata schema should be adjusted automatically by changing this weak reference into a strong reference. Otherwise, it is sufficient to create appropriate subscription rules at the corresponding LMRs. This approach can also be extended to determine obsolete strong references that should be changed into weak references or even resource classes that are used in queries but not yet cached locally.

With strong references, an LMR can receive resources for which there is no corresponding rule. An LMR must take care in deleting such resources if the resource that caused their transmission is deleted, e.g., because the according rule is changed or removed. MDV uses

a garbage collector (based on reference counting) to detect such resources and to remove them, if necessary.

3.3 Publish & Subscribe Algorithm

In this section, we describe our publish & subscribe algorithm, particularly one of its core components, the filter algorithm. One of the main challenges in publishing data is the evaluation of subscription rules. The evaluation is necessary to obtain all subscribers that have to be notified of new, updated, or deleted data. To avoid the evaluation of the possibly huge set of all subscription rules, our filter determines a (small) subset of subscription rules that are at most affected by the modification of the data. Additionally, our filter takes advantage of rule/predicate redundancy and evaluates affected rules incrementally, i.e., using only the modified data as far as possible.

Our filter algorithm is solely based on standard relational database technology, mainly for the advantages in robustness, scalability, and query abilities. We implemented a prototype based on the MDV system, its rule language, and the RDF data model.

3.3.1 Overview of the Approach

Consider the following rule that registers all cycle providers (their resources) that are running in the domain 'uni-passau.de':

`search` CycleProvider c `register` c `where` c.serverHost `contains` 'uni-passau.de'

This rule must be evaluated when a resource of class CycleProvider is registered, updated, or deleted. The following rule, which registers all cycle providers that are running on a computer with more than 64 MB of memory, shows that it is not that simple:

`search` CycleProvider c `register` c `where` c.serverInformation.memory > 64

This rule must be evaluated not only when a CycleProvider resource is registered, updated, or deleted, but also when the referenced ServerInformation resource is updated. For example, if the ServerInformation resource's memory property is updated from 32 to 128, all CycleProvider resources referencing the updated resource are then matching the above rule.

Figure 3.3 illustrates the basic idea of our filter algorithm. Both, documents and subscription rules are decomposed into so-called *atoms*, i.e., basically tuples of a table. For RDF documents, an atom is basically an RDF statement (or triple, as described in [LS99]). For subscription rules, an atom is composed of the rule parts that refer to a single class. The filter algorithm joins the document atoms with the rule atoms obtained from the subscription rule base to determine all rules that may register resources of the document and, therefore, have to be evaluated.

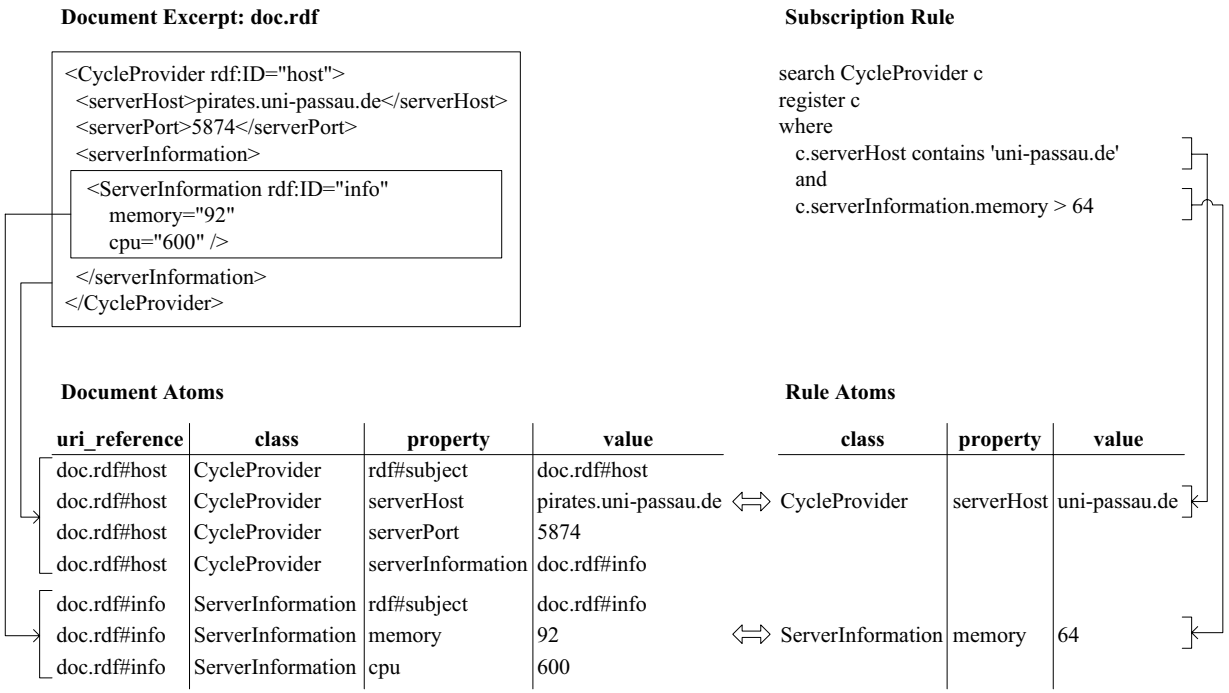


Figure 3.3: Basic Idea of the Filter Algorithm

In summary, our publish & subscribe algorithm proceeds in three steps: First, newly registered documents are decomposed. Second, newly registered rules are decomposed. Third, document atoms and rule atoms are matched, and rules that may register new resources are evaluated incrementally. We describe each of these steps in the following sections.

3.3.2 Decomposition of Documents

Any newly registered RDF document is decomposed into its atoms, i.e., RDF statements, as described in [LS99]. These statements and some additional information (necessary for filter execution) are inserted into the table *FilterData* (see Figure 3.4 for an example). Additionally, for each resource a tuple is inserted containing the URI reference and the class name (with the property attribute set to `rdf#subject` and the value attribute set to the resource's URI reference). Thus, rules are able to register a single resource using its URI reference.

3.3.3 Decomposition of Rules

To obtain the rule atoms, a new subscription rule is processed in three steps: First, the rule is normalized, basically to ease decomposition. Afterwards, the normalized rule is

FilterData			
uri_reference	**class**	**property**	**value**
doc.rdf#host	CycleProvider	rdf#subject	doc.rdf#host
doc.rdf#host	CycleProvider	serverHost	pirates.uni-passau.de
doc.rdf#host	CycleProvider	serverPort	5874
doc.rdf#host	CycleProvider	serverInformation	doc.rdf#info
doc.rdf#info	ServerInformation	rdf#subject	doc.rdf#info
doc.rdf#info	ServerInformation	memory	92
doc.rdf#info	ServerInformation	cpu	600

Figure 3.4: Table *FilterData* based on the RDF document of Figure 3.1

decomposed into so-called *atomic rules*. Finally, a dependency tree is created based on information obtained from the decomposition step and merged with a global dependency graph.

We distinguish two types of atomic rules: A *triggering rule* refers to a single class; it requires no results of other atomic rules and contains no path expressions, i.e., it contains only accesses to properties. A *join rule* represents a join of two extensions with a join predicate. It contains no other predicates, and it always depends on two other atomic rules.

The normalization of rules is not crucial for the correctness of rule decomposition, but it eases its explanation and implementation. We call a rule normalized if its `search` part contains all classes that are used in its `where` part, not only directly using a variable but also in path expressions. Path expressions are not allowed in normalized rules, only accesses to properties (including the "?" operator), and they are split up therefore. As an example, we present the normalized form of the rule from Section 3.2.3:

```
search CycleProvider c, ServerInformation s
register c
where c.serverHost contains 'uni-passau.de' and c.serverInformation = s
      and s.memory > 64
```

3.3.3.1 Rule Decomposition

The decomposition of a subscription rule into atomic rules is done based on the predicates used in it: In a first step, all predicates with a constant are removed from the original rule, and for each of them a triggering rule is created with the predicate as `where` part, i.e., the triggering rule matches all resources that satisfy the predicate. If there are classes in the `search` clause without such a predicate, a triggering rule without `where` clause is created. After this, the original rule is adjusted to use the results of the triggering rules as input instead of evaluating the removed predicates. As an example, consider the (normalized) rule

```
search CycleProvider c, ServerInformation s
register c
where c.serverHost contains 'uni-passau.de' and c.serverInformation = s
      and s.memory > 64 and s.cpu > 500
```

All predicates with constants are considered and appropriate triggering rules are generated:

```
search ServerInformation s register s where s.memory > 64          (RuleA)

search ServerInformation s register s where s.cpu > 500            (RuleB)

search CycleProvider c
register c
where c.serverHost contains 'uni-passau.de'                        (RuleC)
```

The original rule is adjusted afterwards:

```
search RuleA a, RuleB b, RuleC c
register c
where a = b and c.serverInformation = a                            (RuleD)
```

Notice that a rule's type is the type of the resources it registers, e.g., the type of RuleD is CycleProvider. All remaining predicates in the original rule are join predicates. Now, subsequently each such predicate is removed, and a join rule is created with the removed predicate in its `where` part. The original rule is again adjusted. This is done until the original rule is itself a join rule. In our example, two join rules are generated:

```
search RuleA a, RuleB b register a where a = b                     (RuleE)

search RuleE a, RuleC c register c where c.serverInformation = a   (RuleF)
```

The subscription rule is now decomposed into the atomic rules RuleA, RuleB, RuleC, RuleE, and RuleF.

3.3.3.2 Creation of the Dependency Graph

The decomposition always creates non-cyclic dependencies between the generated atomic rules. These dependencies are represented in a dependency tree in which nodes represent atomic rules and directed edges represent dependencies. The tree contains an end rule (an atomic rule that produces the result of the subscription rule) as root node, one or more triggering rules as leave nodes, and join rules as inner nodes. Figure 3.5 depicts a dependency tree that is based on the atomic rules in Section 3.3.3.1.

After decomposition, the generated atomic rules are merged with already existing atomic rules (stemming from previously registered rules). This is equivalent to merging the dependency tree of the newly registered rule with the *global dependency graph*, which is a directed, acyclic graph that consists of the merged dependency trees of previously registered rules. By merging dependency trees, the filter algorithm takes advantage of rule and predicate redundancy and, as a consequence, evaluates equivalent rules and atomic rules only once.

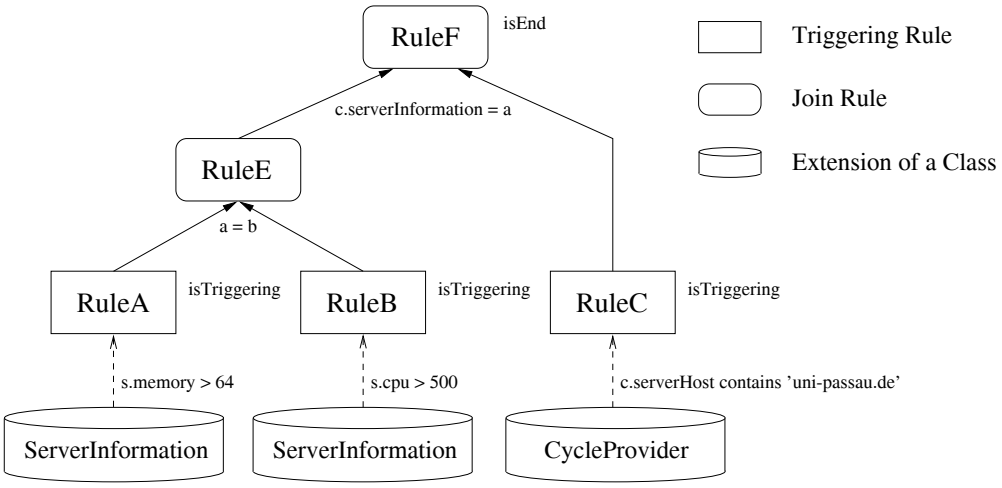


Figure 3.5: Dependency Tree of the Example Rule in Section 3.3.3.1

3.3.3.3 Rule Groups

Although the decomposition algorithm already considers redundancies, there remain similar atomic rules. Consider the following example:

`search` CycleProvider c `register` c `where` c.serverInformation.memory > 64

`search` CycleProvider c `register` c `where` c.serverInformation.cpu > 500

Decomposition results in the following atomic rules (notice that RuleA is already shared):

`search` CycleProvider `register` c `where` c (RuleA)

`search` ServerInformation s `register` s `where` s.memory > 64 (RuleB1)

`search` RuleA c, RuleB1 s `register` c `where` c.serverInformation = s (RuleC1)

`search` ServerInformation s `register` s `where` s.cpu > 500 (RuleB2)

`search` RuleA c, RuleB2 s `register` c `where` c.serverInformation = s (RuleC2)

Comparing RuleC1 and RuleC2 reveals that both atomic rules have equal `register` and `where` parts and that even the classes that the variables are bound to are equal.[3] But the resources used to evaluate the rules are different. To avoid individual evaluation of such join rules, *rule groups* are introduced. A rule group combines join rules that have an equal `where` part and where the corresponding variables are bound to the same classes. All grouped join rules are evaluated at once by combining their input data, evaluating the shared `where` part, and splitting up the result afterwards. Figure 3.6 depicts this for the above example.

[3]Notice that variable names need not be equal.

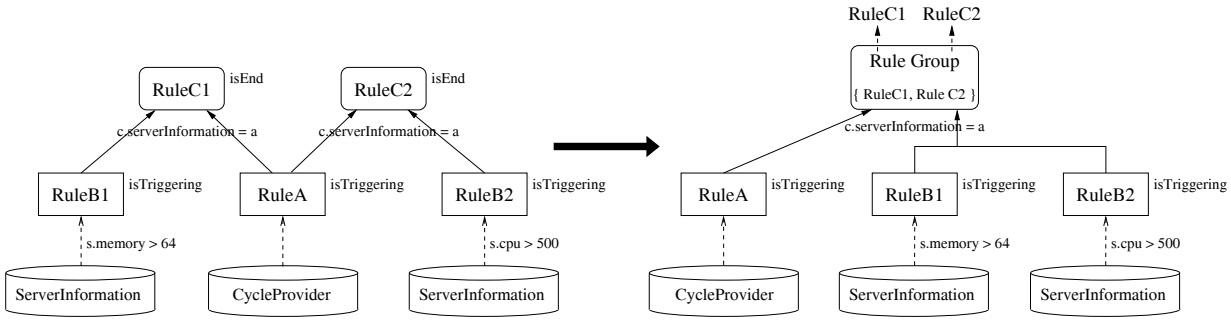


Figure 3.6: Generation of Rule Groups

AtomicRules		
rule_id	**text**	**group**
1	`search` ServerInformation s `register` s `where` s.memory > 64	
2	`search` ServerInformation s `register` s `where` s.cpu > 500	
3	`search` CycleProvider c `register` c `where` c.serverHost `contains` 'uni-passau.de'	
4	`search` view(1, ServerInformation) a, view(2, ServerInformation) b `register` a `where` a = b	1
5	`search` view(4, ServerInformation) a, view(3, CycleProvider) c `register` c `where` c.serverInformation = a	2

Figure 3.7: Table *AtomicRules* based on the Example in Section 3.3.3.1

3.3.3.4 Implementation Issues

We now describe the most important tables used by the filter algorithm. For brevity reasons, we omit tables that are not directly related to it, e.g., tables that store the subscription rules of LMRs. A key concept to an efficient filter implementation is the physical database design. First, the filter tables are used as indexes to all triggering rules affected by newly registered metadata. Given some metadata, the tables allow an efficient determination of all triggering rules that have a `where` part that evaluates to true given the new metadata. Second, the tables themselves are created with indexes supporting an efficient access on the database level.

Table *AtomicRules*[4] stores all atomic rules (see Figure 3.7 for an example). There are no duplicates, i.e., no rules having the same rule text, but different values in their rule_id attribute. *RuleDependencies* stores the global dependency graph (see Figure 3.8).

[4]We use view(rule_id, class) to refer to another atomic rule (instead of, e.g., RuleA).

RuleDependencies			
source	**target**	**param**	**group**
1	4	1	1
2	4	2	1
4	5	1	2
3	5	2	2

Figure 3.8: Table *RuleDependencies* based on the Example in Section 3.3.3.1

RuleGroups	
group	**text**
1	`search` group(ServerInformation) a, group(ServerInformation) b `register` a `where` a = b
2	`search` group(ServerInformation) a, group(CycleProvider) c `register` c `where` c.serverInformation = a

Figure 3.9: Table *RuleGroups* based on the Example in Section 3.3.3.1

A reference to the rule group of an atomic rule is stored in its *AtomicRules* tuple and, for efficiency reasons, in *RuleDependencies*, in the tuples where the atomic rule is the target. The rule groups themselves are stored in *RuleGroups* (see Figure 3.9).

Triggering rules are additionally inserted into one of the tables *FilterRulesOP* or *FilterRules* depending on the operator used in their `where` part. Our current implementation supports comparisons with operators `<`, `<=`, `>`, and `>=` only on numerical constants. To avoid the creation of an own *FilterRulesOP* table with an appropriate type of the value attribute for all numerical types, constants are stored as strings and re-converted when joining the tables with the *FilterData* table. Figure 3.10 shows an example extension of the *FilterRules/FilterRulesOP* tables based on the triggering rules from Section 3.3.3.1.

3.3.4 Filter Algorithm: Matching Documents and Rules

The filter algorithm is started after a new document has been registered and decomposed. It consists of two steps: First, all triggering rules are determined that are affected by the registration of new metadata. Subsequently, all join rules depending on the affected triggering rules are evaluated incrementally, as defined by the global dependency graph.

3.3.4.1 Determination of Affected Triggering Rules

It is crucial that a triggering rule refers to a single class. Its `where` part is either empty or a comparison with a constant. If it is empty, the atomic rule matches any newly registered

FilterRulesGT			
rule_id	**class**	**property**	**value**
1	ServerInformation	memory	64
2	ServerInformation	cpu	500

FilterRulesCON			
rule_id	**class**	**property**	**value**
3	CycleProvider	serverHost	uni-passau.de

Figure 3.10: Triggering Rules of Example 3.3.3.1

resource that is an instance of the class the rule refers to. If it is a comparison with a constant, the rule's predicate has to be evaluated based on the atoms of the document. One matching atom is sufficient for a triggering rule to be affected. Our prototype implementation starts with joining the table *FilterData* with *FilterRules* and all *FilterRulesOP* tables using a join predicate depending on the actual *FilterRules/FilterRulesOP* table. The table *ResultObjects* always contains the result of a filter step. The left table of Figure 3.11 shows the result of this step based on the *FilterData* table in Figure 3.4 and the *FilterRules/FilterRulesOP* tables in Figure 3.10.

3.3.4.2 Evaluation of Join Rules

Now, all join rules depending on affected triggering rules are evaluated. With join rules, complete incremental evaluation is not possible, so the results of atomic rules which join rules depend on are materialized.

The evaluation consists of several iterations. In each iteration, all atomic rules depending on the atomic rules currently stored in *ResultObjects* are determined using the table *RuleDependencies*. Then, the rule groups of these atomic rules are evaluated using the resources currently stored in *ResultObjects* and, if necessary, materialized data as input. The result of this evaluation, i.e., the matching resources and the atomic rules they match, are again stored in *ResultObjects* and used as input of the next iteration. Any resources matching an end rule are saved for later. The algorithm terminates if there are no more dependent join rules. Termination is guaranteed because the dependency graph is an acyclic, directed graph, so there is a longest path from a triggering rule leaf to an end rule node, which has no dependent join rules. The length of this path is the maximum number of iterations executed by the filter algorithm. Figure 3.11 shows an example filter run based on the tables presented in Section 3.3.3.4. The filter terminates with the resource doc.rdf#host as result.

After the filter terminated, all resources produced by end rules are transmitted to the appropriate LMRs.

Initial Iteration	
uri_reference	**rule_id**
doc.rdf#info	1
doc.rdf#info	2
doc.rdf#host	3

Iteration 1	
uri_reference	**rule_id**
doc.rdf#info	4

Iteration 2	
uri_reference	**rule_id**
doc.rdf#host	5

Figure 3.11: Table *ResultObjects* for an Example Execution of the Filter

3.3.5 Updates and Deletions

Updated and deleted resources can be determined by comparing the original RDF document with the updated, re-registered one. A resource is updated if it is contained in both documents, but at least one property is changed, added, or removed. A resource is deleted if it was contained in the original document, but it is then no longer contained in the updated one. If a complete document is deleted, all contained resources are deleted.

One execution of the MDV filter is not sufficient if updates and deletions are allowed. If a resource is updated, three situations can be distinguished:

- The resource is no longer matched by a rule that matched the resource before. It must be removed from an LMR's cache if this was the only rule the resource matched. If the resource still matches other rules, it must stay in the cache.
- The resource is matched by a rule that did not match the resource before. This situation is handled properly by the filter.
- The resource is still matched by all rules that matched it before. All LMRs that cache this resource must update their cache to contain the modified resource.

Furthermore, resources referencing an updated or deleted resource must be considered. Assume the following subscription rule:

`search` CycleProvider c `register` c `where` c.serverInformation.memory > 64

If a ServerInformation resource is updated, i.e., its memory property is set to 128, there can be CycleProvider resources that now match this rule. Whereas, if the ServerInformation resource's memory property is set to 32 or if the resource is deleted, there can be cached CycleProvider resources that now must be removed from the cache because they no longer match the rule, but only if there are no other rules that the resources match. Notice that resources that are cached because of strong references are removed by the garbage collector, if necessary.

Our approach to solve all of this is to execute the filter multiple times, each time with different input data. First, the filter is executed with the original version of updated and deleted resources as input. The result is a set of so-called candidate resources. Each of these resources no longer matches at least one rule. We call them candidates because

OID: `search` CycleProvider c `register` c `where` c = *URI*

COMP: `search` CycleProvider c `register` c `where` c.synthValue > *INT*

PATH:
```
search CycleProvider c
register c
where c.serverInformation.memory = INT
```

JOIN:
```
search CycleProvider c
register c
where c.serverHost contains 'uni-passau.de' and
      c.serverInformation.cpu = 600 and c.serverInformation.memory = INT
```

Figure 3.12: Benchmark Rule Types

there can be other rules they still match or new rules they now match (after an update). Second, the modified metadata is written into the database, and the filter is executed a second time, with the candidate resources as input. The result of the execution is a set of wrong candidate resources, that is, resources that must not be deleted. All true candidate resources (i.e., the set of resources determined in the first iteration excluding those obtained in the second iteration) can be deleted from LMRs' caches. Finally, the filter is executed a third time, now with the modified metadata as input. The last execution corresponds to the single filter execution that would suffice if no updates and deletions were allowed.

Alternatives to executing the filter multiple times are, for example, to store a list of the LMRs which cache a resource (for each resource); or to use periodical cache invalidation based on a time-to-live approach, resulting in resources dropping out of an LMR cache if they are not reinserted periodically.

3.4 Performance Experiments

Now, we present some performance experiments conducted using our prototype implementation of the filter algorithm. The results are important to decide if the filter should be started either when a new document is registered or periodically, to process several documents in one batch. All benchmarks were conducted on a Sun Enterprise 450 with 4GB memory running Solaris 2.7 and using Sun's Java JDK 1.2.2. As a back-end, we used a major commercial RDBMS.

We registered RDF documents similar to the document of Figure 3.1, each containing two resources: one of class CycleProvider and one of class ServerInformation. As rule base, we used the rule types shown in Figure 3.12. For a single measurement, documents and rules of type OID, PATH, and JOIN were created in such a way that the CycleProvider resource in a document was matched by exactly one rule and that each rule matched exactly one resource (stored in one document) of all newly registered resources. COMP rules and corresponding documents were created in a way that a CycleProvider resource

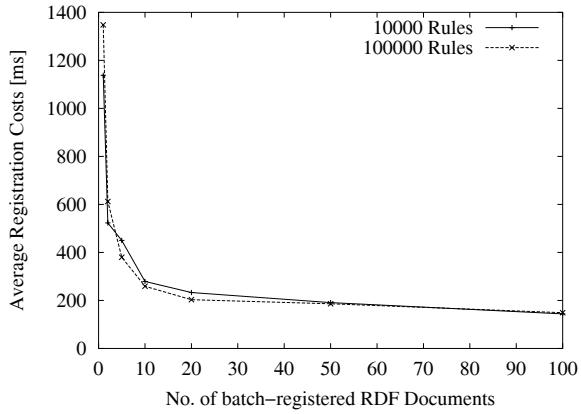


Figure 3.13: OID Rules

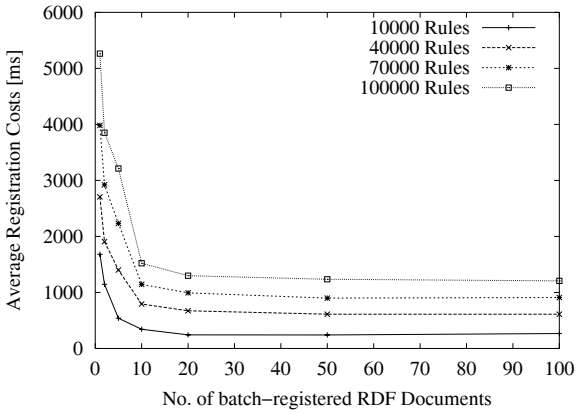


Figure 3.14: PATH Rules

was matched by a certain percentage of the rules stored in the rule base, e.g., 10%. OID rules, which register a single CycleProvider resource with a given URI reference *URI*, and COMP rules, which register all CycleProvider resources with a synthValue property greater than the parameter *INT*, were both triggering rules. No decomposition was necessary, and the filter algorithm did not need to evaluate any join rules. On the other hand, PATH and JOIN rules contained accesses to properties of referenced resources, so decomposition was necessary and join rules were created. Therefore, the complete filter algorithm was used to evaluate them.

We conducted the measurements with various batch sizes, an increasing rule base size, and different rule types. In a single measurement, we first created a rule base consisting of rules of the same type. Then, we registered a number of RDF documents and measured the overall runtime of the filter algorithm to process them. Depending on the rule type, the algorithm terminated after the evaluation of all triggering rules (OID, COMP) or after the evaluation of all dependent join rules (PATH, JOIN). The average registration time of a single RDF document was calculated by dividing the overall runtime by the batch size. From the filter's point of view, registering several small documents and registering one large document is the same. So, these measurements also illustrate the behavior of the filter algorithm regarding different document sizes.

Figures 3.13, 3.14, 3.15, and 3.16 show the dependency of the average registration costs from the batch size, i.e., the number of documents registered in one batch, for different rule types and rule base sizes.[5] For OID, PATH, and JOIN rules the behavior is basically the same. Registration of a small number of documents is more expensive than the registration of a lot of documents in one batch. From a certain batch size on (dependent on the rule type), the average registration costs are nearly constant. For COMP rules this is different; although the registrations costs are again nearly constant from a certain batch size on, registering few documents in one batch is preferable.

For simple OID rules, the rule base size does not influence the runtime of the algorithm

[5]Minor variations of the graphs are a consequence of the timing based on Java.

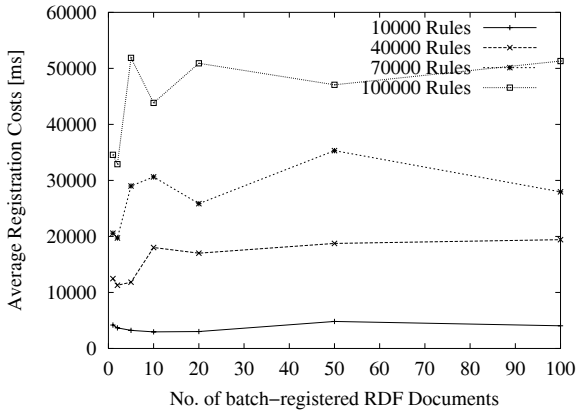


Figure 3.15: COMP Rules (10% of Rule Base)

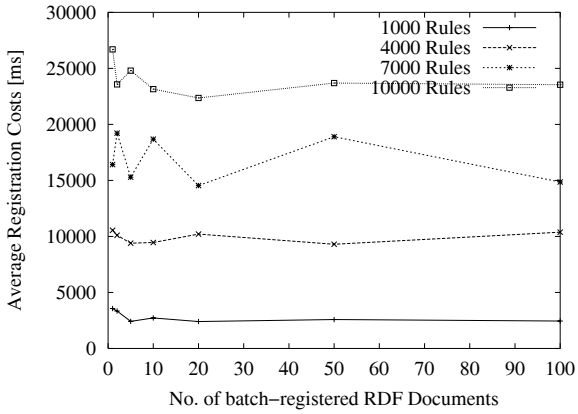
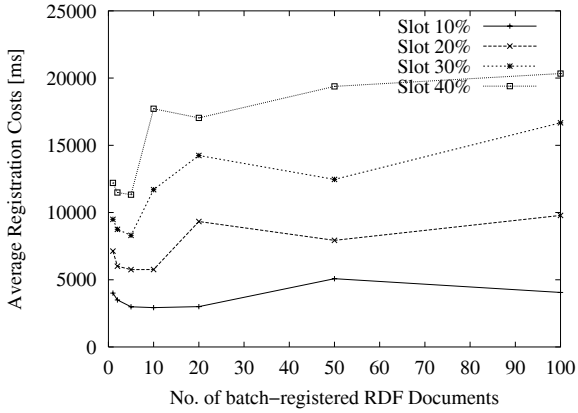


Figure 3.16: JOIN Rules



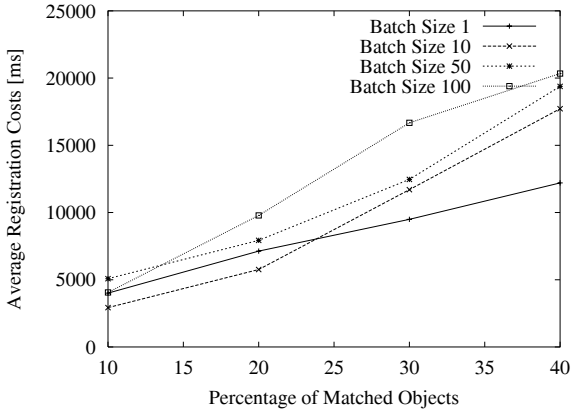


Figure 3.17: 10,000 COMP Rules - Varying Batch Sizes and Triggered Rule Base Percentage

as the curves for 10,000 and 100,000 are almost identical. This is different for PATH, JOIN, and COMP rules where the registration costs do—as expected—depend on the rule base size, as Figures 3.14, 3.15, and 3.16 show.

Figure 3.17 shows the influence of the percentage of rules that match resources from the registered documents on the average registration costs. Not surprisingly, a higher rule percentage results in higher registration costs independent of the batch size.

3.5 Related Work

Metadata management systems, data repositories, and catalogs are used in DBMSs since many years to store metadata about tables, indexes, and other data structures [WDH+82]. With the emergence of the Internet, new metadata management systems with new chal-

lenges arose. UDDI [UDD00] is a framework for storing and searching services provided by companies on the Internet. Contrary to MDV, UDDI defines a fixed metadata schema and no automatic notification on data changes is provided. WebSemantics [MRT98] searches the Internet for HTML pages that contain metadata about data sources and integrates them into a catalog. The middleware system MOCHA [RMR00] uses a (centralized) metadata repository to store Java classes and documentation about these classes in RDF. The Secure Service Discovery Service [CZH+99] stores metadata about network services in XML format. Lookup services like Jini [Wal01], UPnP [UPN], and SLP [GPVD99] allow the discovery of plug-and-play services in networks but do not support large quantities of data or a query language.

Our filter algorithm uses triggering rules as an index to all subscription rules that are affected by new metadata. A similar approach is used in the publish & subscribe system Le Subscribe [PFLS00, FJL+01]. It uses the predicates of subscription queries as index but within the scope of a main memory algorithm. The Gryphon system [ASS+99] creates a tree from queries composed of simple predicates where each node represents a comparison. In [HCKW90], an Interval Binary Search Tree is introduced that stores intervals and allows it to find all intervals that contain some given value efficiently. Whereas most publish & subscribe systems assume a distinguished data format, the information dissemination system SIFT [YGM99] allows arbitrary (text) documents to be published. [AF00] presents a filter algorithm that uses XQL queries to select specific XML documents from an incoming stream of documents. To our knowledge none of these systems allows references between the information that is published, i.e., between different documents, as MDV does.

The NiagaraCQ [CDTW00] system evaluates queries continuously against a database. Queries are decomposed into partial queries so that common partial queries are evaluated only once. A similar concept is implemented in OpenCQ [LPT99] where queries are decomposed into events connected by the operators of the original query. Events are triggered by changes of the underlying data. Both systems and MDV can handle the insertion, update, and deletion of data. [TGNO92] introduced the concept of continuous queries but is restricted to append-only databases.

The cache of an LMR can be viewed as a set of materialized views with the corresponding rules as view definitions. [GMS93, BLT86, LHM+86] present some algorithms for updating materialized views. In [Han87], different strategies to update materialized views are investigated with respect to their performance. There are also similarities with semantic caching as described in [DFJ+96].

Chapter 4

Deployment of MDV within ObjectGlobe

In this chapter, we describe the deployment of the MDV system within the ObjectGlobe system. In particular, in Section 4.1 we describe the MDV lookup service, which provides global access to the metadata of all registered providers. It is also consulted by the ObjectGlobe query optimizer in order to find relevant resources to execute a query and to obtain statistics. In Section 4.2, we describe the MDV security provider, which is a central part of ObjectGlobe's security system. The MDV security provider is responsible for authentication and authorization of users and their query plans, and it deploys the MDV system to store and administrate security information. We also show how MDV's publish & subscribe mechanism is used to distribute authorization data throughout the ObjectGlobe federation so that it is available to all query optimizers, which need this data in order to generate valid query plans.

4.1 MDV Lookup Service

The ObjectGlobe system deploys the metadata management system MDV as its lookup service. The MDV lookup service is used to store the metadata of all registered data, function, and cycle providers. It is also used by the ObjectGlobe query optimizer in order to find relevant resources for query optimization and to obtain statistics and authorization data. Furthermore, end users can browse through the metadata of the MDV lookup service and search for available query capabilities and data sources for their applications. The MDV lookup service has already been presented in [BKK+01a, BKK+01b, BKK+00, BKK+99].

4.1.1 ObjectGlobe's Metadata

Within the ObjectGlobe system, the MDV lookup service records the following information:

- **Data Provider Metadata:** Each collection of objects stored by a data provider and the attributes of each collection are recorded by the MDV lookup service. A col-

lection is either a materialized partition conforming to ObjectGlobe's internal nested relational format or a virtual collection, i.e., an Internet data source transformed into the collection's recorded schema by a wrapper. Collections are associated to a specific theme. A theme describes a special concept with a set of terms, called attributes. A theme's attributes can be viewed as the union of all attributes meaningful for the theme. Queries are formulated over the themes and their attributes.

Integration of a new data source is achieved by registering it as a new collection and associating it to a theme. So collections can be seen as horizontal (possibly overlapping) partitions. The attributes provided by the new collection must be a subset of the attributes defined by the associated theme. Currently, ObjectGlobe uses a non-hierarchical set of themes, but more complex ontologies [BCV99] could be added on top of our flat theme structure. As an example, `www.hotelbook.com` and `www.hotelguide.com` provide different collections which are associated to the theme *Hotel*.

Furthermore, the MDV lookup service stores binding patterns of a collection, statistics about a collection like histograms for estimating the selectivity of simple (non-external) predicates, and information about replicas (mirrors) of a collection that could be provided by some other data provider.

- **Cycle Provider Metadata:** The CPU power, the size of the main memory, and the temporary disk space of each cycle provider are recorded. The load on the cycle provider regarding CPU power and available main memory is stored as a function of time, and likewise we store the latency and bandwidth information for the network links between cycle providers.

- **Function Provider Metadata:** The name and signature of each query operator are recorded. Furthermore, formulas to estimate the consumption of CPU cycles, main memory, disk space, and the selectivity for each query operator are kept by the MDV lookup service. These formulas use a set of parameters that describe the characteristics of the executing cycle provider (e.g., the available CPU power/main memory) and the input data for a specific application of this operator.

  ObjectGlobe differentiates between iterators like *join* or *display* and transformers such as *thumbnail*. (In addition, ObjectGlobe also has special categories for predicates and aggregate functions.) Any kind of function, however, will automatically be wrapped by ObjectGlobe into an iterator so that we ignore these distinctions in this thesis and use the words function and query operator interchangeably for the general concept.

Figure 4.1 shows an example RDF document that can be used by a data provider to register a *Hotel* collection.[1] The relevant information about a data provider can be

[1] We shortened and simplified the example RDF fragments in this thesis to a reasonable degree and removed all namespaces from the presented documents for better readability and a more concise presentation.

found enclosed in the `DataProvider` element. It contains information about the name of the provider and a URL with which the data provider can be contacted. The `Partition` element contains information about the collection that the data provider makes available.

At the beginning of the collection description, we can find the data provider of the collection, a plain-text description of the content of the collection, the theme (*Hotel*) this collection is associated with, etc. The element `wrapper` specifies a reference for the wrapper which performs the necessary transformations to integrate the collection into the ObjectGlobe system. More interesting is the content of the `attributes` element. It contains the description of the type of the tuples given by the collection. In our case, the type contains three attributes, and for each attribute the name and the type of the attribute are specified. It is possible to insert additional information about attributes which is omitted for brevity.

The metadata kept in the MDV lookup service can be outdated or incomplete. It is possible, for instance, that a function provider removes a query operator from its repository without notifying the MDV lookup service. As a result, the execution of a query might fail due to a missing operator which is detected at execution time. ObjectGlobe relies on data, function, and cycle providers to notify the MDV lookup service if important metadata changes. If a plan fails due to stale metadata in the MDV lookup service, all relevant metadata is invalidated so that providers that do not update their metadata are eventually excluded from the ObjectGlobe federation. As an alternative, [CZH+99] proposes to use a time-to-live scheme; in that scheme, providers must periodically contact the lookup service if they want to continue to remain in the federation.

4.1.2 Using the MDV Lookup Service

As mentioned before, data, function, and cycle providers are registered by generating RDF documents describing their services. We use RDF because it is very flexible and an Internet standard for describing resources [LS99]. Typical collections such as relational or XML data sources can very easily be described using RDF; it is also possible to automatically produce large fractions of an RDF description from, say, an XML DTD or a relational schema. An RDF document is also used to update the metadata if a provider changes or extends its services, and the URI reference of an RDF object is used to unregister (delete) services.

To find relevant resources and obtain statistics and authorization data, the MDV lookup service provides a declarative query language. As an example, the following query shows how to ask the MDV lookup service for all collections that supply data for the *Hotel* theme:

```
search Partition d
register d.uniqueId, d.attributes.*
where d.theme.name='Hotel' and d.attritutes.?.topic='city' and
      d.attritutes.?.topic='address' and d.attritutes.?.topic='price'
```

More specifically, the above query asks for *Hotel* collections that have city, address, and price attributes, and the query asks for the uniqueId of the collection (used to identify replicas) and information about all attributes. (The "?" in the query is an *any* operator.)

```
<RDF>
  <DataProvider rdf:ID="HotelBook">
    <dataProviderName>HotelBook</dataProviderName>
    <dataProviderUrl>http://www.hotelbook.com</dataProviderUrl>
  </DataProvider>
  <Partition rdf:ID="HotelBookPartition">
    <dataProvider rdf:resource="#HotelBook"/>
    <partitionDescription>
      Description of hotels worldwide
    </partitionDescription>
    <theme rdf:resource="http://example.org/Themes#Hotel"/>
    <localName>hotelBookPartition</localName>
    <wrapper rdf:resource="http://example.org/Operators#HotelBookWrapper"/>
    <uniqueID>4711</uniqueID>
    <cardinality>30000</cardinality>
    <attributes>
      <rdf:Bag>
        <rdf:li>
          <Attribute>
            <topic rdf:resource="http://example.org/Themes#cityTopic"/>
            <domain rdf:resource="http://example.org/Themes#StringDomain"/>
          </Attribute>
        </rdf:li>
        <rdf:li>
          <Attribute>
           <topic rdf:resource="http://example.org/Themes#addressTopic"/>
           <domain rdf:resource="http://example.org/Themes#StringDomain"/>
          </Attribute>
        </rdf:li>
        <rdf:li>
          <Attribute>
            <topic rdf:resource="http://example.org/Themes#priceTopic"/>
            <domain rdf:resource="http://example.org/Themes#IntegerDomain"/>
          </Attribute>
        </rdf:li>
      </rdf:Bag>
    </attributes>
  </Partition>
</RDF>
```

Figure 4.1: RDF Registration Code for a Data Collection

The result of this query is shown in Figure 4.2. Here, we show the results for the *Hotel* collection specified in the RDF document of Figure 4.1.

```
<collection>
  <uniqueId>4711</uniqueId>
  <attribute topic="city" domain="String" />
  <attribute topic="price" domain="Integer" />
  <attribute topic="address" domain="String" />
</collection>
```

Figure 4.2: Example Search Result

The MDV lookup service also allows the definition of views. These views can be materialized. Such materialized views are very helpful to support sessions in which search results are iteratively refined. For example, it is possible to first ask for all cycle providers that are allowed to process objects of a specific collection and then, in a separate search request, ask which of *these* cycle providers are capable to execute a specific query operator.[2] This feature is important for parsing and optimization and for users who interactively browse the metadata.

4.2 MDV Security Provider

In this section, we describe the MDV security provider, which is a central part of ObjectGlobe's security system. A general conception of the security system as well as a description of its security measures were already given in Section 2.4. Within this security system, the MDV security provider is responsible for authentication and authorization as well as the distribution of authorization data, which is needed throughout query optimization to generate valid query plans. Parts of the MDV security provider have already been presented in [KKKK01, BKK+01a].

In ObjectGlobe, a cycle, function, or data provider is autonomous regarding authentication and authorization, that is, the provider is free to choose its own authentication and authorization policy and it is also free to choose the actual security system which enforces this policy and to which every decision regarding authentication and authorization is delegated. A cycle, function, or data provider has two choices: If available, it can use its own internal security system (for instance, Web servers or DBMSs have their own built-in security system). If such a security system is not available, the cycle, function, or data provider can use the ObjectGlobe security system. Within ObjectGlobe's security system, the MDV security provider (not to be mixed up with common providers which are cycle, data, or function providers[3]) is responsible for authentication and authorization, i.e., every decision regarding authentication and authorization is delegated to it.

In the following, we first present the MDV security provider. Then, we describe the

[2] Of course, these cycle providers could also be found in a single search request.

[3] In the following, the term provider always means cycle, data, or function provider. We never use it to refer to an MDV security provider.

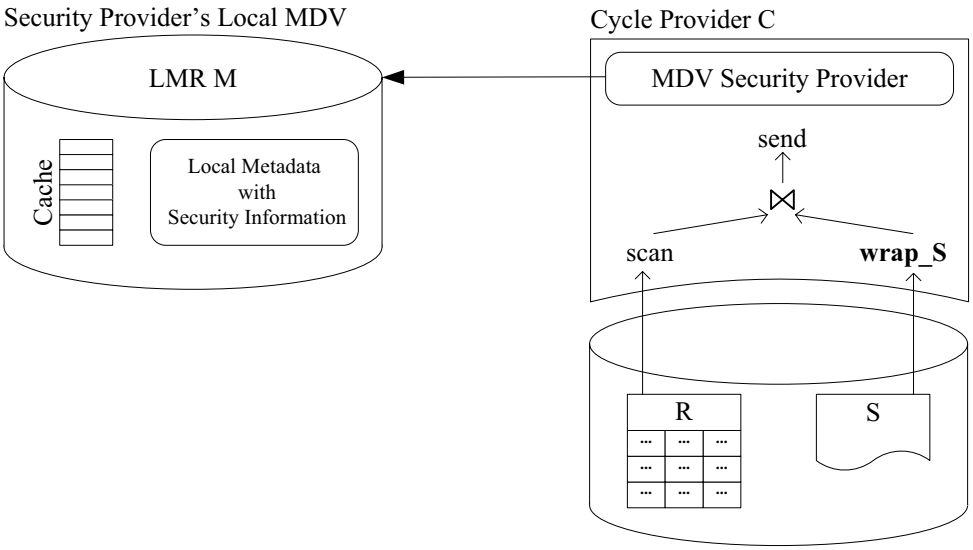


Figure 4.3: Architecture of the MDV Security Provider

distribution of authorization data in order to support query optimization. Finally, we show how the security systems of (cycle, function, or data) providers that use their own internal security system are integrated.

4.2.1 Architecture of the MDV Security Provider

The architecture of the MDV security provider is shown in Figure 4.3. Every provider that uses the ObjectGlobe security system contains an MDV security provider. Every decision of the provider regarding authentication and authorization is delegated to its MDV security provider. The MDV security provider stores all its security information in a local metadata repository (LMR) of the MDV system. The security information is stored as *local metadata*, independent of any cached global metadata.[4] We refer to the LMR in which an MDV security provider stores its data as its *local MDV*.

Using an LMR for storing security information has the advantage that MDV's public & subscribe mechanism can be used to distribute parts of the stored security information to the local MDVs of other MDV security providers, which is necessary for generating valid query plans in query optimization, see Section 4.2.2. A further advantage is that related providers, e.g., all providers of a company, are able to share the same security information even if the providers are distributed widely throughout the Internet. They can, e.g., share the same group of users.

Authentication and authorization are carried out by the individual cycle, data, or function providers when a query plan is received, i.e., during the plug phase. Prior to the query

[4]Basically, other systems than the MDV system could be used as well, e.g., RDBMSs or LDAP repositories [WHK97].

plan's instantiation, a provider asks its MDV security provider if access should be granted, i.e., if the query plan should be executed locally. The MDV security provider requests all necessary information from its local MDV to authenticate the user and to verify if the user that initiated the query has the proper access rights. If so, the access is granted; otherwise, the query plan is rejected.

4.2.1.1 Authentication

ObjectGlobe supports a flexible authentication policy. Users and applications that only access free and publicly available resources can be anonymous and no authentication is required. Authentication is only required for authorization or accounting purposes of providers. Cycle providers can also require authenticated external operators to restrict the function providers, e.g., to execute only code originating from trusted sources within the same company or Intranet. A detailed description of this issue is given in [SBK01].

Two possible authentication schemes are supported. In both schemes, the authentication data is inserted into the query plan generated from the user's query:

- A user can provide a password. The password is used to generate a secret key (using the PKCS #5 password-based encryption standard [PKC99]) which is afterwards used to calculate a MAC (Message Authentication Code) of the query plan and some additional data.

- The user possesses a valid X.509 certificate [HFPS99, PKI]. The private key corresponding to the certificate is used to calculate a digital signature of the query plan and some additional data.

If a data provider does not support one of these schemes, i.e., requires the password in plain text, the password is inserted (as authentication data) into the query plan. The wrapper accessing the data provider extracts the password and passes it to the data provider. To keep the password secure, it is encrypted with the public key of the cycle provider that executes the wrapper. So no other cycle provider is able to access the plain password.

Locally at a provider, the authentication data contained within the query plan is compared with authentication data generated based on the received query plan and the user's authentication data stored in the local MDV (the users identity is contained within the query plan). If they are equal, the user is successfully authenticated; otherwise, authentication failed and the query plan is rejected.

4.2.1.2 Authorization

Contrary to the Internet, where most of the information is available for everyone, providers will constrain access to their resources, e. g. machine cycles, relations and operators. Authorization guarantees that only users having the proper access rights can access the resources of the providers.

ObjectGlobe and the MDV security provider use a role-based access control (RBAC) model [SCFY96] to specify authorization rules. RBAC distinguishes between three sets of entities:

- **Users:** This set contains the legal users of the system. Users in the ObjectGlobe system are the human beings who are allowed to use the system.
- **Roles:** Roles are used to model the functions that can be occupied by the users, for instance, within a company or an organization. Thus, roles depend to a high degree on the scope in which they are used, i.e., they depend on an actual company or organization. Role inheritance allows modeling of relationships and hierarchies between roles. As a result, a role hierarchy is created in which a more powerful (or derived) role inherits permissions from a less powerful (or base) role.
- **Permissions:** Permissions represent the privileges available within the system. Just as roles, they depend highly on the scope in which they are used.

  ObjectGlobe provides the following permissions:

  - **Read Permissions:** These permissions allow the access to a data collection on a data provider (e.g., executing a wrapper or using a *scan* operator). If a data collection is accessed using a wrapper, the permission to access the data collection implicitly includes the permission to execute the corresponding wrapper.
  - **Execute Permissions:** Execute permissions allow users to execute a given query operator at a cycle provider, e.g., the *thumbnail* operator at cycle provider C in the example query plan from Figure 2.2.
  - **Load Permissions:** These permissions allow users to load a query operator from a function provider.

Relations exist between the sets users and roles and between roles and permissions. Every user can be assigned to several roles, depending on the function the user occupies within a company or an organization. Likewise, a role can be assigned to several users. Also, a role can be assigned to one or more permissions, just as a permission can be assigned to several roles. So in summary, RBAC distinguishes between users, roles that are assigned to users, and permissions that are assigned to roles.

When a provider receives a query plan, it uses the local parts of the query plan to determine a list of permissions that the user requires to execute the query plan at the provider. The security provider compares these list of required permissions with the permissions that are stored within its local MDV for the corresponding user. If a required permission is missing, the query plan is rejected.

For example, to execute the query plan in Figure 4.3 (which is a sub-plan of the example query plan in Figure 2.2), a user requires the following permissions on cycle provider C: the permission to execute the query operator *scan* (*send* is an internal operator for which no permission is required), the permission to access data collection R (with the *scan* operator),

```
<User rdf:ID="keidl">
  <userName>Markus Keidl</userName>
  <authenticationType>password</authenticationType>
  <authenticationData>qZ8uiXFEePpGu</authenticationData>
  <assignedRoles>
    <rdf:Bag>
      <rdf:li rdf:resource="example-role" />
      <rdf:li rdf:resource="other-role" />
    </rdf:Bag>
  </assignedRoles>
</User>
```

Figure 4.4: User Security Information in RDF Format

the permission to load the wrapper *wrap_S* from a function provider, and the permission to access data collection S using the wrapper *wrap_S*. As said above, the last permission implicitly includes the permission to execute the wrapper *wrap_S*.

4.2.1.3 The MDV Security Provider's Metadata

A MDV security provider stores all its security information within an LMR of the MDV system as local metadata. It records the following security information as RDF metadata:

- **User Security Information:** User security information constitutes data about a user's identity, data for the user's authentication (password or certificate), and the roles that the user is assigned to. Figure 4.4 shows an RDF fragment with security information about the user *Markus Keidl*.[5] The user is authenticated by a password and assigned to two roles. The roles are specified by their URI reference.

- **Role Security Information:** The security information about a role encompasses its name and the permissions that are assigned to it. As roles can be derived from one another, a base role can also be specified. The derived role then inherits all permissions of its base role. Currently, the MDV security provider only supports single inheritance. Figure 4.5 shows an RDF fragment that defines the role *ExampleRole*. The role has four permissions assigned to it specified by their URI reference, and it is not derived from another role.

- **Permission Security Information:** The MDV security provider supports three different types of permissions: read, execute, and load permissions. The permissions stored within an LMR are only valid at the providers that use this LMR as their local MDV. For example, in Figure 4.3 an execute permission for the *thumbnail* operator

[5]Passwords (the content of the property `authenticationData`) are stored in encrypted format, of course.

```
<Role rdf:ID="example-role" roleName="ExampleRole">
  <assignedPermissions>
    <rdf:Bag>
      <rdf:li rdf:resource="execute-scan" />
      <rdf:li rdf:resource="access-R" />
      <rdf:li rdf:resource="access-S" />
      <rdf:li rdf:resource="load-wrap-S" />
    </rdf:Bag>
  </assignedPermissions>
</Role>
```

Figure 4.5: Role Security Information in RDF format

```
<ExecutePermission rdf:ID="execute-scan" operatorName="scan" />
<ReadPermission rdf:ID="access-R">
  <partitionName>R</partitionName>
</ReadPermission>
<ReadPermission rdf:ID="access-S" wrapperName="wrap_S">
  <partitionName>S</partitionName>
</ReadPermission>
<LoadPermission rdf:ID="load-wrap-S" operatorName="wrap_S" />
```

Figure 4.6: Permission Security Information in RDF format

stored within the LMR M is valid at all cycle providers that use the LMR M as their local MDV, e.g., it is valid at cycle provider C.

All permissions require some additional data, which depends on the actual type of the permission:

- For a read permission, the name of the data collection and the name of its data provider (at which the data collection is located) must be specified. If the data provider is omitted, the data collection can be accessed at any data provider using the LMR with this security information. If the data collection is accessed using a wrapper, the wrapper's class name must also be specified.
- An execute permission requires the query operator's class name as additional data.
- For a load permission, the query operator's name and (optionally) the function provider from which the operator can be loaded must be specified.

To execute the query plan in Figure 4.3 on cycle provider C, a user requires the permissions shown in Figure 4.6. Notice that the `ReadPermission` for data collection S implicitly includes an `ExecutePermission` for the execution of wrapper *wrap_S*.

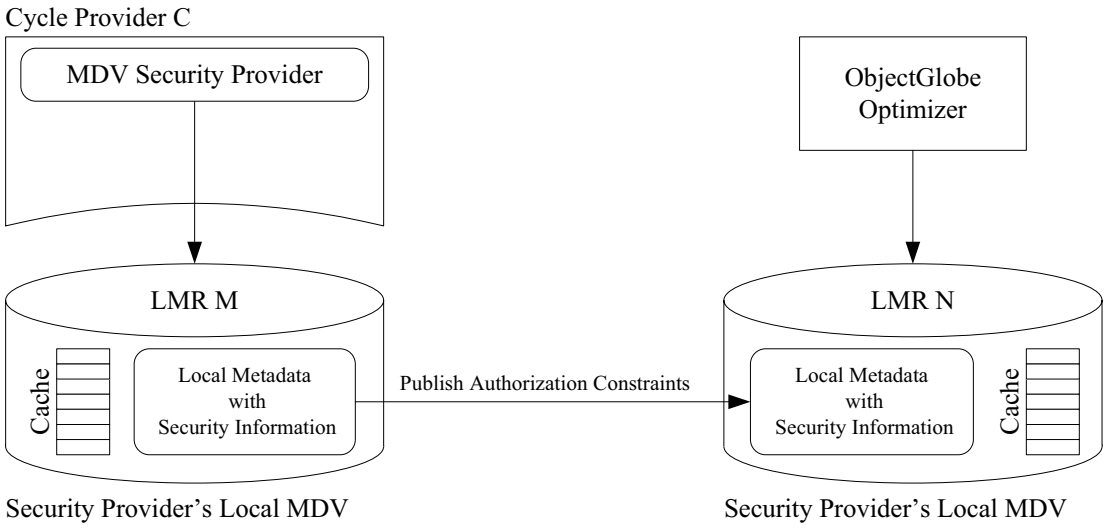


Figure 4.7: Distribution of Authorization Constraints

An MDV security provider obtains the security information for authentication and authorization of a user and the user's query plan using MDV's query language. For example, the following query determines if the user *Markus Keidl* has the necessary permissions to access data collection S using the wrapper *wrap_S*:

```
search User u, Role r, ReadPermission p
select p
where u.userName='Markus Keidl' and u.assignedRoles.?=r and
      r.assignedPermissions.?=p and
      p.partitionName='S' and p.wrapperName='wrap_S'
```

4.2.2 Distribution of Authorization Constraints

Providers constrain the access or use of their resources to particular user groups. In order to generate valid query execution plans and avoid failures at execution time, an ObjectGlobe query optimizer must know about these authorization constraints, i.e., information about the data sources a user is allowed to access, about the cycle providers the user is allowed to execute code on, and the query operators the user is allowed to load. For query optimization, these constraints are required from all providers that should be considered throughout plan optimization.

ObjectGlobe uses the MDV system's publish & subscribe mechanism to distribute these authorization constraints. An example situation is illustrated in Figure 4.7. On the left, cycle provider C and its MDV security provider are shown. The security provider uses the LMR M as its local MDV to store and administrate security information. Parts of this security information are distributed, as authorization constraints, to LMR N on the right

using MDV's publish & subscribe mechanism. LMR N acts as local MDV for the illustrated ObjectGlobe optimizer. Consequently, the query optimizer considers the authorization data of cycle provider C and its MDV security provider throughout query optimization.

In the above scenario, the LMR M builds its own backbone. The subscribers to this backbone are other local MDVs, which are, for example, used by ObjectGlobe query optimizers. As authorization data and therefore authorization constraints are critical information, the subscription to this data is handled differently than the subscription to public metadata: The subscription rules for authorization data are not specified by the subscribers themselves as it is done when subscribing to public metadata. Instead, these subscription rules are given by the administrator of the MDV security provider (and its associated local MDV) which stores and administrates the corresponding security information. The actual authorization data a subscriber is allowed to subscribe to must be negotiated between the suppliers of the MDV security provider and the query optimizer, respectively.

The following subscription rules subscribe to all authorization data of user *Markus Keidl* (including the permissions):

```
search User u
register u
where u.userName = 'Markus Keidl'

search Permission p, Role r, User u
register p
where u.userName='Markus Keidl' and u.assignedRoles.?=r and
      r.assignedPermissions.?=p
```

Updates to the authorization data of this user, e.g., updates to the user's permissions, are automatically propagated as authorization constraints to all subscribers using MDV's publish & subscribe mechanism.

4.2.3 Internal Security Systems of Providers

As said above, providers may have their own internal security system to which they delegate every decision regarding authentication and authorization. An example is a Web server that allows to download query operators. The Web server participates in the ObjectGlobe federation as function provider, but it does not use the MDV security provider.

ObjectGlobe's security system supports such providers in two ways. First, it allows to delegate authentication and authorization to the provider's own internal security system. Second, it enables the extraction of authorization data from the internal security system to consider it in query optimization, similar as it is done with authorization data stored in local MDVs.

4.2.3.1 Delegation of Authentication and Authorization

For data providers that use their own internal security system, like, e.g., DBMSs or Web servers, ObjectGlobe supports the handover of authentication data contained within query

plans to the provider's internal security system. Authentication and authorization of users and their local query plans are then performed solely by the provider's internal security system.

The handover is done by passing the authentication data to the wrapper that is used to access the data provider. The wrapper is responsible for forwarding this data to the internal security system in an appropriate way. While contained within the query plan, the authentication data is encrypted using public key cryptography. Only the receiving cycle provider, which executes the corresponding wrapper, is able to decrypt it.

4.2.3.2 Authorization Constraints of Internal Security Systems

Initially, the authorization data of internal security systems is not contained within any local MDV (which are used by ObjectGlobe query optimizers to find relevant resources and obtain statistics). So, this data would not be known and considered throughout query optimization. But in order to generate valid query execution plans and avoid failures at execution time, ObjectGlobe must know about this authorization data, which means that it must be incorporated into the local MDVs. Further problems are updates and modifications of this data, which are not performed within the local MDVs but within the internal security systems.

ObjectGlobe supports the integration of such authorization data as authorization constraints by the use of *extractors*. They provide the necessary functionality to extract authorization data from a provider's internal security system. Every provider that wants to allow ObjectGlobe the extraction of this data must provide a suitable extractor. This is similar to the concept of wrappers used to integrate data sources.

When accessing a provider's data collection for the first time, the corresponding extractor (if available) is used to extract all available authorization data from the internal security system and to store it in the security provider's local MDV. MDV's publish & subscribe mechanism distributes the data to other local MDVs the same way it distributes the authorization data of MDV security providers.

Updates and modifications of internal authorization data are determined in the following ways:

- The query of a wrapper is rejected by a provider's internal security system. Obviously, the query plan was generated based on obsolete authorization data. The wrapper notifies its cycle provider of the failure. The cycle provider at first cancels the query. Then, it uses the appropriate extractor to re-extract any updated or modified authorization data from the internal security system. Finally, it updates the local metadata of its security provider's local MDV.

- The administrator of a provider's internal security system notifies all cycle providers that are allowed to access the provider (using a wrapper) of modifications. The cycle providers will extract the modified authorization data and update their security providers' local MDVs accordingly.

- Using the extractor, cycle providers check in periodical intervals if the authorization data in internal security systems of providers and therefore authorization constraints have changed. If so, they update their local MDVs accordingly.

Chapter 5

ServiceGlobe - Open and Distributed Web Services

Web services are a new technology for the development of distributed applications on the Internet. A Web service (also called service) is an autonomous software component that is uniquely identified by a URI and that can be accessed by using XML and standard Internet protocols like SOAP or HTTP [RV02]. A service may combine several applications that a user needs such as the different pieces of a supply chain architecture. For a client, however, the entire infrastructure will appear as a single application. Due to its potential of changing the Internet to a platform of application collaboration and integration, Web service technology gains more and more attention in research and industry; products like Sun ONE, Microsoft .NET, or IBM WebSphere show this development. All these frameworks implement respectively use Web service standards published by the World Wide Web Consortium (W3C) and other consortia, for example, SOAP, WSDL, and UDDI.

In this chapter, we introduce the ServiceGlobe system, our open and distributed Web service platform. It is fully implemented in Java and based on standards like XML, SOAP, UDDI, and WSDL. ServiceGlobe supports mobile code, i.e., Web services can be distributed on demand and instantiated at runtime at arbitrary Internet servers participating in the ServiceGlobe federation. It offers all standard functionality of a service platform like SOAP communication, a transaction system, and a security system [SBK01]. These areas are well covered by existing technologies and are therefore not the focus of our work. We also assume that appropriate standards will be developed and incorporated into service platforms. Parts of this chapter have already been presented in [KSK03a, KSKK03, KSK03b, KSK02]. A demo of the ServiceGlobe system was given at the VLDB'02 conference [KSSK02].

The remainder of this chapter is structured as follows: Section 5.1 presents a motivating scenario that is used as an example in the following sections. In Section 5.2, we present a short introduction into Web service standards that are important in our work. In Section 5.3, our ServiceGlobe system is described. Finally, Section 5.4 presents related work.

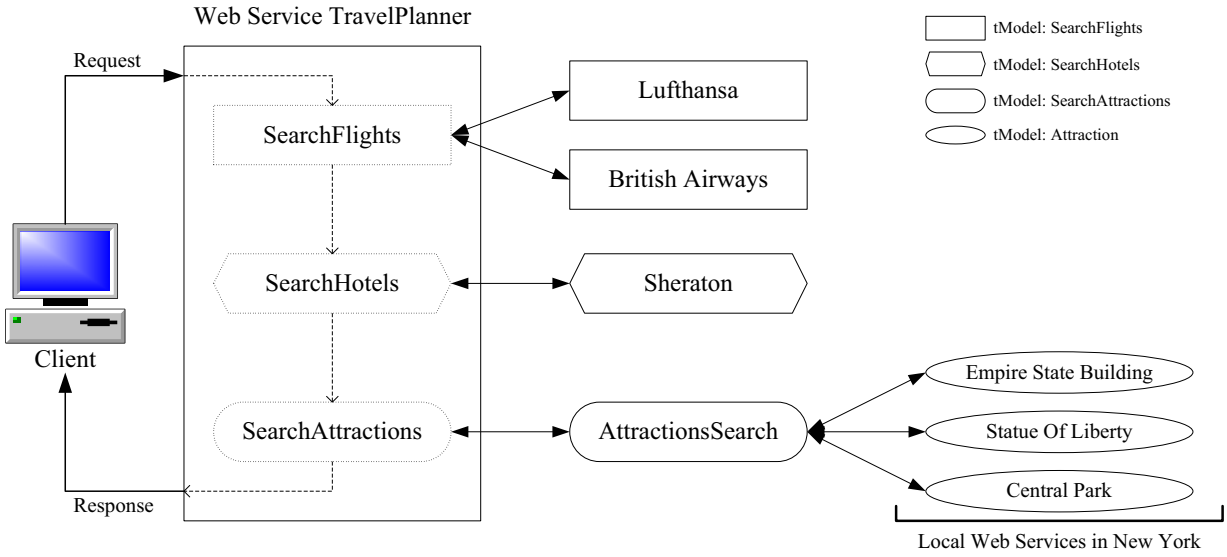


Figure 5.1: Motivating Scenario: A Travel Agency Portal

5.1 Motivating Scenario

In the following, we use an information service scenario from the travel agency field as a motivating example. Currently, services in this area are mostly available as plain Web pages or HTML forms. In the future, the Internet will provide a lot of information services in this area, e.g., Web services for searching for flights, hotels, attractions, and so on. These Web services can help, e.g., travel agencies, to plan and carry out journeys.[1]

The motivating scenario is depicted in Figure 5.1. We assume a portal for travel agencies that provides Web services for travel agencies as well as for consumers to plan and book trips, including flights, hotels, attractions and so on. Information about the available Web services of different providers is stored within a UDDI repository. In this repository, the services are assigned to so-called *tModels* (short for "technical model"). A tModel provides a semantic classification of a Web service's functionality and a formal description of its interfaces. All Web services assigned to the same tModel must provide the same functionality. More information on UDDI and tModels is given in Section 5.2.2.

Besides Web services of providers, the travel agency portal also provides own Web services with extended functionality that combine several of the providers' Web services. The Web service TravelPlanner is one of these services. It allows one to plan a holiday or business trip. Figure 5.1 shows an example execution of this service in order to plan a trip to New York. To begin, TravelPlanner searches for suitable flights. For this purpose, it obtains all Web services assigned to the tModel `SearchFlights`, which describes Web services for searching for flight information, and invokes one or more of them. In the example, two services are found (Lufthansa and British Airways) and invoked. Next,

[1] Travel portals like MapQuest.com are a first step in this direction.

TravelPlanner searches for suitable hotels by invoking Web services assigned to the tModel `SearchHotels`. Finally, TravelPlanner searches for local attractions at the holiday resort. Therefore, it invokes the Web service AttractionsSearch to retrieve a list of suitable attractions. AttractionsSearch itself uses UDDI to find Web services close to New York and to obtain detailed and up-to-date information about, e.g., fees, opening hours, or calendars of events.

5.2 Web Services Fundamentals

There exist a variety of XML-based standards concerning Web services. We will briefly survey the most important ones needed to understand our work.

5.2.1 The SOAP Standard

SOAP [Mit03] is an XML-based communication protocol for distributed applications. SOAP is designed to exchange messages containing structured and typed data and can be used on top of several different transfer protocols like HTTP, SMTP (Simple Mail Transfer Protocol), or FTP (File Transfer Protocol). The use of SOAP over HTTP is the de-facto standard in the current landscape of Web services.

SOAP itself does not define any application semantics and therefore can be used in a broad range of applications. It can be used to simply deliver a single message or for complex tasks like request/response message exchange or RPC (Remote Procedure Call). The following XML document shows the basic structure of a SOAP message, consisting of three parts: an envelope, an optional header, and a mandatory body.

```
<Envelope encodingStyle="...">
  <Header>
    <!-- The header is optional -->
  </Header>
  <Body>
    <!-- Serialized object data -->
  </Body>
</Envelope>
```

The `Envelope` element is the root element of a message and contains the other two elements `Header` and `Body`. The `Header` element of a message offers a generic mechanism for extending the SOAP protocol in a decentralized manner. This is used for extensions like Web Service Security [ADLH+02]. We defined a SOAP header extension to transmit Web service context within SOAP messages, see Chapter 7. The `Body` element of the message contains the payload of the message.

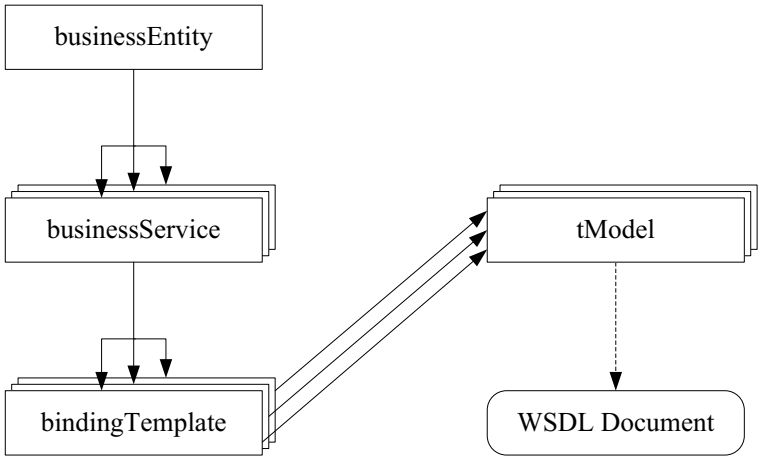


Figure 5.2: UDDI Data Structures

SOAP offers a standard *encoding style*[2], i.e., serialization mechanism, to convert arbitrary graphs of objects to an XML-based representation, but user-defined serialization schemes can be used as well.

5.2.2 The UDDI Standard

UDDI (Universal Description, Discovery and Integration) is designed to "provide a platform-independent way of describing services, discovering businesses, and integrating business services using the Internet" [UDD00]. Four main data structures constitute the basic schema: businessEntity, businessService, bindingTemplate, and tModel. While the first three data structures form a hierarchy, the tModel can be seen as an independent structure providing concepts, ideas, and technical fingerprints of services (see Figure 5.2).

- **businessEntity:** This data structure contains data about an entire company or party which offers a family of services. For example, a travel agency can register its company name, address information, and contact persons. The concept of categories allows for the classification of businesses in several dimensions, e.g., industry codes or geographic locations. User-defined dimensions are also possible. Normally, a businessEntity registers several services.

- **businessService:** This structure contains information about a particular service offered by a businessEntity. For example, a travel agency may have search and booking services. It also contains one or more bindingTemplates specifying binding information for this service.

[2]The standard serialization can be referenced by the URL `http://schemas.xmlsoap.org/soap/encoding/`.

- **bindingTemplate:** The most important component of this structure is the access point of a service, i.e., the actual URL, phone number, etc., by which a service can be invoked. In ServiceGlobe, each service host is specified by a bindingTemplate with its URL as an access point. A bindingTemplate may have several references to tModels.

- **tModel:** As a technical fingerprint, tModels describe various concepts and classifications. In ServiceGlobe, for example, tModels are used as functionality descriptions for services like searching attractions or service hosting. A tModel may contain a link to a WSDL document that specifies the signature of the service in detail [CER02]. Besides these service-classification-oriented tModels, concept-oriented tModels like geographical locations or industry codes are possible as well.

Invoking a service requires knowledge of the signature and the access point of the service. The signature of the service provides the structure of the SOAP document to communicate with the service (input parameters, output parameters, and data types). This signature is defined in the WSDL document referenced by the tModel of the service. The access point, which is stored in the bindingTemplate structure, references an actual implementation of a service.

5.2.3 The WSDL Standard

WSDL (Web Service Description Language) [CCMW01] is an XML-based language for describing the technical specifications of a Web service. In particular, it describes the operations offered by a Web service, the syntax of the input and output documents, and the communication protocol to use for communication with the service. The exact structure of a WSDL document is complex and beyond the scope of this thesis, but we will give a brief overview of the WSDL standard. First, a service in WSDL is described on an abstract level and then bound to a specific protocol, network address (normally a URL), and message format.

On the abstract level, port types are defined. A port type is a set of operations. Every operation is associated with a number of input and output messages, defining the order and type of the messages sent to/received from the operation. There are four message exchange patterns defined within the WSDL specification: one-way, request/response, solicit/response, and notification. An operation expects one message as input and generates one output message. The messages themselves are assembled from several typed parts. The types are defined using XML Schema [Fal01].

On the non-abstract level, port types are bound to concrete communication protocols and concrete formats of the messages using so-called bindings. The most commonly used binding today is the SOAP 1.1 binding defined in the WSDL 1.1 specification. Messages are serialized according to a set of rules defined by an encoding style. Finally, a service in WSDL is defined as a set of ports, i.e., bindings with associated network addresses.

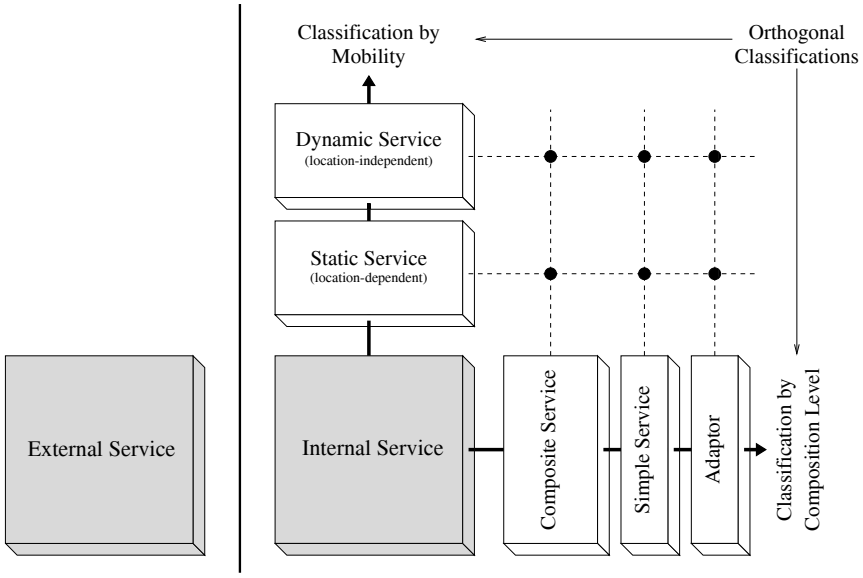


Figure 5.3: Classification of Services

5.3 Architecture of ServiceGlobe

The ServiceGlobe system provides a lightweight infrastructure for an open, distributed, and extensible service platform. It is completely implemented in Java and based on standards like XML, SOAP, UDDI, and WSDL. In this section, we present the basic components of the ServiceGlobe infrastructure. First of all, we distinguish two different types of services: external and internal services (see Figure 5.3).

External services are services currently deployed on the Internet that are not provided by ServiceGlobe itself. Such services are normally stationary, i.e., running only on a dedicated host, may be realized on arbitrary systems on the Internet, and may have arbitrary interfaces for their invocation. Since we want to integrate these services independent of their actual invocation interface, e.g., RPC, we use *adaptors* to transpose internal requests to the external interface (and vice versa). This way we are also able to access arbitrary applications, e.g., ERP applications. Thus, external services can be used like internal services and, from now on, we consider only internal services.

Internal services are native ServiceGlobe services. They are implemented in Java using the Service API provided by the ServiceGlobe system. ServiceGlobe services use SOAP to communicate with other services. Services receive a single XML document as input parameter and generate a single XML document as a result. There are two kinds of internal services, namely *dynamic services* and *static services*. Static services are location-*dependent*, i.e., they cannot be executed dynamically on arbitrary ServiceGlobe servers. Such services may require access to certain local resources, e.g., a local DBMS to store data, or require certain permissions, e.g., access to the file system, that are only available on dedicated servers. These restrictions prevent the execution of static services on arbitrary

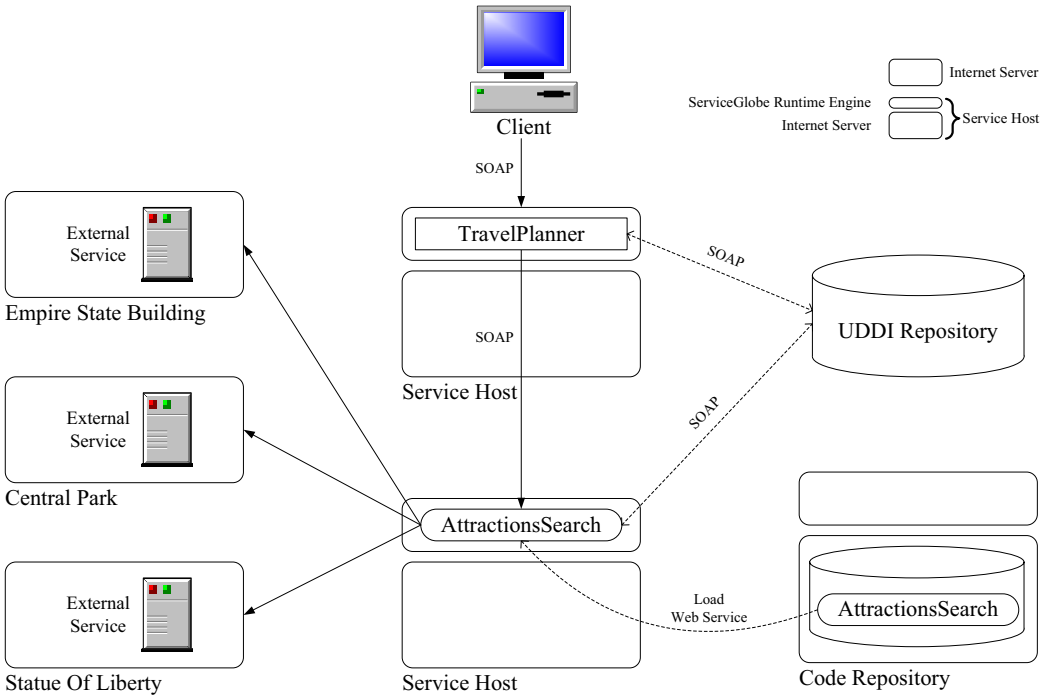


Figure 5.4: Architecture of the ServiceGlobe System

ServiceGlobe servers. In contrast, dynamic services are location-*independent*. They are state-less, i.e., the internal state of such a service is discarded after a request was processed, and do not require special resources or permissions. Therefore, they can be executed on arbitrary ServiceGlobe servers.

There is an orthogonal categorization for internal services: *adaptors*, *simple services*, and *composite services*. We have already defined adaptors. Simple services are internal services not using any other service. Composite services are higher-value services assembled from other internal services. These services are, in this context, called *basis services* because the composite service is based on them. Notice that a composite service can also be used as a basis service for another higher-value composite service. Of course it is feasible to use a specialized programming language, e.g., XL [FK01], or a GUI tool to draw a representation (similar to a workflow graph) of a service, but that is not the focus of our work.

Internal services are executed on *service hosts*, i.e., hosts connected to the Internet that are running the ServiceGlobe runtime engine. ServiceGlobe's internal services are mobile code, therefore their executables are loaded on demand from *code repositories* onto service hosts or, more precisely, into the service hosts' runtime engines. A UDDI server is used to find an appropriate code repository storing a certain service. Service hosts offer a SOAP service (called *runtime service loading*) to execute dynamic services. Thus, the set of available services is not fixed and can be extended at runtime by everyone participating in the ServiceGlobe federation. If internal services have the appropriate permissions, they

can also use resources of service hosts, e.g., databases. These permissions are part of the security system of ServiceGlobe, which is based on [SBK01]. The permissions are managed autonomously by the administrators of the service hosts. This security system also deals with the security issues of mobile code introduced by runtime service loading. Thus, service hosts are protected against malicious services.

Runtime service loading allows *service distribution* of dynamic services to arbitrary service hosts, opening a great optimization potential: Several instances of a dynamic service can be executed on different hosts for load balancing and parallelization purposes. Dynamic services can be instantiated on service hosts having the optimal execution environment, e.g., a fast processor, huge memory, or a high-speed network connection to other services. Of course, this feature also contributes to reliable service execution because unavailable service hosts can be replaced dynamically by available service hosts. Together with runtime service loading this provides a large flexibility in order to consider load balancing or optimization issues.

Figure 5.4 gives an overview of the basic components of the ServiceGlobe system and their mutual interaction (based on the travel agency scenario in Section 5.1). At first, a client sends a SOAP request to a service host to invoke the static service TravelPlanner. This Web service deploys the dynamic service AttractionsSearch as basis service during its execution. Therefore, a suitable service host is located by UDDI requests and the AttractionsSearch service is loaded from a code repository (if not already cached). Then, it is instantiated and executed on the service host on behalf of the TravelPlanner service. The AttractionsSearch service deploys three external Web services (adaptors are omitted in the figure) to obtain detailed information about attractions.

5.4 Related Work

The success of Web services results in a large number of commercial service platforms and products, e.g., the Sun ONE framework [Sunb] that is based on J2EE [J2E], Microsoft .NET [NET], and IBM WebSphere [IBM]. All these products and platforms rely on the well-known standards XML, SOAP, UDDI, and WSDL. They all provide tools for fast and straightforward deployment of existing applications as Web services. Furthermore, there are research platforms like ServiceGlobe and SELV-SERV [BDSN02] which focus on certain aspects in the Web service area. SELV-SERV, for example, focuses on composing Web services using state charts.

Although service composition languages are not the focus of our work, we are aware of work in this area. IBM's WSFL (Web Services Flow Language) [Ley01], Microsoft's XLang [Tha01], and HP's WSCL (Web Services Conversation Language) [BBB+02] are languages for describing how to compose existing services, i.e., to describe some kind of conversation. Compaq's Web Language [Mar99] (formerly WebL) specializes in fetching, parsing, and generating HTML and XML content. Besides service composition, the XL language [FK01] additionally offers very powerful statements for easy and efficient programming of services. HP's eFlow [CS01] is similar but more workflow oriented and

based on a graphical notation. Regarding services and their composition, there is also the ebXML [ebX] standardization effort that defines a standard for global electronic business.

Chapter 6

Dynamic Service Selection

In this chapter, we present dynamic service selection, a novel technique for flexible and reliable execution and deployment of Web services in dynamic environments, which can be integrated into existing service platforms. It offers the possibility of selecting and invoking services at runtime based on a technical specification of the desired service. Therewith, it provides a layer of abstraction from the actual services. Constraints enable Web services to influence dynamic service selection. Web services can be selected based on the relevant metadata, or replies may be checked for defined properties and discarded, if necessary. Constraints also allow the specification of the number of services that should be invoked and how they should be invoked. Constraints may be specified directly when invoking Web services, but they may also be stored in a service's context. In the latter case, they are extracted and used automatically for dynamic service selection by the service platform. We implemented dynamic service selection within the ServiceGlobe system, our open and distributed Web service platform. Parts of this chapter have already been presented in [KSK03a, KSKK03, KSK03b, KSK02].

The remainder of this chapter is structured as follows: Section 6.1 presents an overview of dynamic service selection. Next, Section 6.2 describes the constraints that can influence dynamic service selection, and Section 6.3 explains how these constraints can be combined to form complex constraints. Section 6.4 shows how constraints are evaluated throughout dynamic service selection. Finally, Section 6.5 presents related work.

6.1 Overview of the Approach

In general, Web services invoke other Web services by passing the Web service's URL or access point to the service platform. In contrast, *dynamic service selection* (DSS) enables Web services to state a technical specification of the services that should be invoked. It is the service platform's task to select suitable Web services, possibly utilizing UDDI. Additionally to the technical specification, different kinds of constraints can be passed over to influence dynamic service selection. The approach of dynamic service selection offers three main advantages:

- An important goal in distributed systems is a high reliability rate and fault tolerance. When using other Web services, it can occur that some of them are not reachable, e.g., due to network partitioning, unavailable service hosts, or the unexpected crash of a basis service.[1] For this reason, hard-coded access points within a Web service are not desirable. Dynamic service selection provides a solution for this problem.

- Using constraints to influence dynamic service selection allows developing generic Web services. As will be shown, it is not necessary to code special properties of the invoked Web services into a Web service itself. Instead, these properties are specified as (declarative) constraints and passed to the Web service at runtime. As new kinds of constraints become available, they can be used without modifying or re-compiling the Web service. Constraints can, for example, specify that only free Web services are used or that services of a given company should be preferred, if possible.

- Dynamic services are instantiated at runtime. Combining this with dynamic service selection offers potential for various optimizations. For example, constraints allow to specify that all dynamic services must be instantiated within a LAN or on a given set of hosts.

In the following, a more detailed description of dynamic service selection will be given. In the description, UDDI is used as Web service repository. Basically, because it is the de-facto standard for such a kind of repository and because it provides the necessary functionality to use it in conjunction with dynamic service selection.

In UDDI, every service is assigned to a tModel[2] which provides a semantic classification of a service's functionality and a formal description of its interfaces. So, a service can be called an *implementation* or an *instance* of its tModel. With dynamic service selection, instead of explicitly stating an actual access point in a service, it is also possible to reference or "call" a tModel. Thus, one defines the functionality of the service that should be called rather than its actual implementation. Without DSS, the selection of services from UDDI based on a search criteria like a tModel has to be done manually by a programmer when implementing a Web service. Furthermore, the search criteria available in UDDI are less general, and there are no criteria for influencing service invocation or for filtering service replies.

As an example of DSS, see Figure 6.1: Three services are assigned to tModel T: Service A, B, and C. Assume, that a programmer wants to implement a new Web service that should invoke a service assigned to tModel T. Without DSS, the programmer would search UDDI for an appropriate service, e.g., Service A, and use its access point in the new Service N1. With DSS, the programmer will instead develop Service N2. This service does not contain any hard-coded access points, instead it contains a call to the tModel T. At runtime, the service will query UDDI for an appropriate Web service and invoke it. If an

[1] Related problems, although in the context of Web scripting languages, have been studied in [CD99a].

[2] In fact, a service in UDDI can be assigned to several tModels. DSS could be adjusted to allow calling services which implement several tModels. As there is no essential difference to calling a single tModel, this will not be considered in the following.

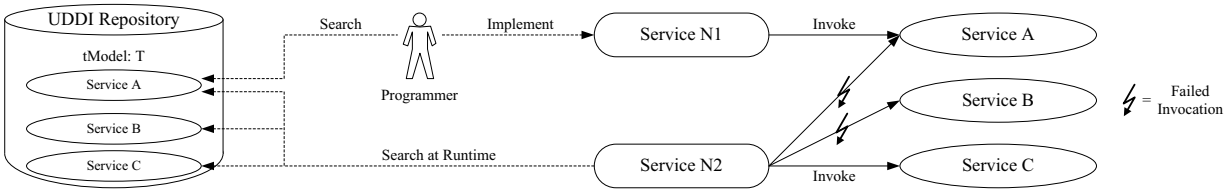


Figure 6.1: Example of Dynamic Service Selection

invocation fails, alternative services are tried until an invocation succeeds (as depicted in Figure 6.1) or no more alternative services are available.

As already mentioned, DSS is implemented within ServiceGlobe. The ServiceGlobe API provides methods for Web services to invoke tModels and to optionally specify constraints and/or use constraints contained in the service's context.

6.2 Constraints

Constraints are used to influence DSS. They can be passed to a service platform within a service's context or by specifying them directly when calling a tModel. The term context refers to information about clients that may be utilized to provide personalized behavior. In the ServiceGlobe system, context is transmitted in the header of the SOAP messages that services send and receive. The integration of constraints into context information enables not only the invoked services to take advantage of them but also further services invoked by these services as the context information of a service is (automatically) included into SOAP messages sent by it. More information about the use of context within the ServiceGlobe system is given in Chapter 7.

Constraints can be differentiated into *preferences* and *conditions*.[3] Conditions must be fulfilled whereas preferences should be fulfilled. When considering preferences in DSS, a service platform at first invokes services that fulfill these preferences. If there is an insufficient number of such services, additional services are invoked that do not fulfill all preferences (but, of course, they must fulfill all conditions).

Orthogonally, there are five different types of constraints: metadata, location, mode, reply, and result constraints. Every constraint type influences a certain phase of dynamic service selection, as depicted in Figure 6.2. Location constraints additionally influence the selection of service hosts at which dynamic services are instantiated. For each type, there are preferences and conditions; though, for mode and result constraints, preferences are useless.

[3] A similar classification of conditions of SQL statements in hard and soft constraints is described in [Kie02].

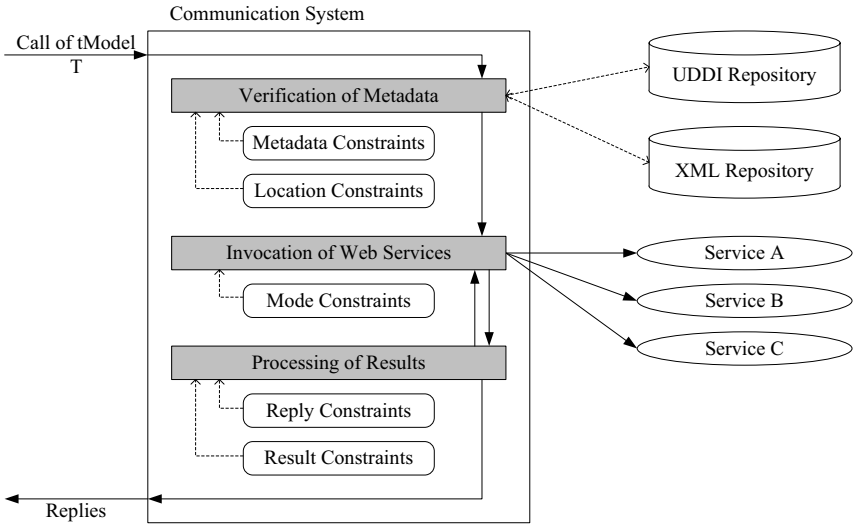


Figure 6.2: Phases of Dynamic Service Selection

Metadata Constraints

Prior to the invocation of services, when the service platform requests all services assigned to a tModel, metadata constraints are applied as filter on all services returned by UDDI (as depicted in Figure 6.2). Metadata constraints are basically XPath [CD99b] queries that are applied to the metadata of a service. Metadata about a service includes primarily its UDDI data. Also, additional metadata that is stored in other metadata repositories like the MDV system, which is described in Chapter 3, and that cannot be found in UDDI may be contained.The following example shows a metadata preference that favors services assigned to a businessEntity with name `MyTravelAgency`:

```
<metadataPreference>
  /businessEntity/name="MyTravelAgency"
</metadataPreference>
```

Location Constraints

Location constraints are used to specify the place of execution of a Web service, i.e., the service host. For static services, this allows their selection based on their location. For dynamic services, this ensures that they are instantiated and executed preferably (preference) or strictly (condition) at the given location. The information about the location of services and service hosts is retrieved from the UDDI repository. The location can be specified by, e. g., a host's network address or geographically based on GPS coordinates or ISO 3166 codes. For example, the following location constraint specifies that selected services must be located or instantiated in an area around the city Passau (ISO 3166 code DE-BY-PAS),

within a radius of 50 kilometers:

```
<locationCondition addressType="Geographical">
  <center>DE-BY-PAS</center>
  <maxDistance>50km</maxDistance>
</locationCondition>
```

Mode Constraints

DSS is not limited to invoke only one instance of a given tModel; it is also possible to invoke several instances. With a mode constraint, the number of services that should be invoked can be specified. As Figure 6.2 shows, mode constraints are central for the invocation of services and the processing of their replies. When processing the reply of a service, the service platform decides based on a mode constraint if an alternative service must be invoked, if the invocation is finished and the replies must be returned as result, or if it is necessary to wait for further replies.

There are three modes available:[4] Using the *one mode*, only one instance out of all tModel instances is called. In case of a failure, e.g., unavailability of a service, an alternative service is tried. Using the *some mode*, a subset of all services returned by UDDI is called in parallel.[5] The number of services is specified as an absolute value or as a percentage. Services that fail are replaced with alternative services. Using the *all mode*, all returned tModel instances are called. Obviously, no alternative services can be called if failures occur. The following example shows a mode constraint that specifies that five percent of the available services should be invoked:

```
<modeCondition modeType="Some" number="5" numberType="Percentage" />
```

Reply Constraints

Reply constraints are evaluated after receiving of a reply of an invoked service. All replies not fulfilling all relevant reply constraints are discarded. There are two kinds of reply constraints. *Selection constraints* are XPath queries that are applied to the reply of a service, including its SOAP parts. With *property constraints*, replies can be selected based on a set of properties in the reply. Properties must be provided either by the service platform or by the invoked service. A service accomplishes this by including corresponding XML elements in its reply. ServiceGlobe itself supports properties for encryption, signature, and age of data. Using the first two properties, it is possible to verify if a reply is encrypted or signed, respectively, and by whom it is signed. The third property can be used to check the age of the returned data.

[4] These modes are similar to unicast, multicast, and broadcast communication on networks.

[5] It should be noticed that the one and the all mode are obviously special cases of the some mode.

```
<andGroup>
  <metadataCondition>
    /businessEntity/name="MyTravelAgency"
  </metadataCondition>
  <locationCondition addressType="Geographical">
    <center>DE-BY-PAS</center>
    <maxDistance>50km</maxDistance>
  </locationCondition>
</andGroup>
```

Figure 6.3: Example of the Combination of Constraints

Result Constraints

Result constraints refer to all replies received so far. There are two kinds of result constraints. With a *timeout constraint*, a maximal waiting time for replies of invoked services can be set. After its expiry, all pending services are aborted, and all replies received so far are returned to the calling service. The following constraint is an example of a timeout constraint:

```
<timeoutCondition value="100" valueUnit="Seconds" />
```

With *first-n constraints*, the call to a tModel can be ended after a predetermined number of replies has been received. The calling service gets only these replies as result of its call. Services that have not responded until this moment are aborted. The number of replies to wait for can be set by an absolute number or by a percentage depending on the number of services invoked initially. The following constraint would end a call after ten percent of all replies having been received:

```
<firstNCondition number="10" numberType="Percentage" />
```

6.3 Combination of Constraints

Constraints can be combined using the operators AND and OR. Figure 6.3 shows an example of an AND combination of two constraints (`andGroup` represents the AND operator). The (combined) constraint specifies that only Web services should be selected and invoked that are assigned to a businessEntity with name `MyTravelAgency` and that are located close to the city Passau (ISO 3166 code DE-BY-PAS).

By the combination of constraints, conflicts can be created that may prevent the fulfillment of all given constraints. As a consequence, only a subset of the given constraints

may be fulfillable, as the following example shows (`orGroup` represents the OR operator):

```
<orGroup>
  <metadataCondition>
    /businessEntity/name="MyTravelAgency"
  </metadataCondition>
  <timeoutCondition value="100" valueUnit="Seconds" />
</orGroup>
```

Initially, the service platform has two choices: On the one hand, it can invoke *only services of the company* `MyTravelAgency` and wait for their replies (therewith fulfilling only the first constraint). On the other hand, it can invoke *all services* assigned to the tModel. But if a timeout occurs, the service platform faces the situation that it either must return all replies received so far immediately (therewith fulfilling only the second constraint) or that it must ignore the timeout and wait for all replies (therewith fulfilling only the first constraint). In the latter case, though, it invoked too many services initially. So, in general, the service platform is unable to fulfill both constraints at the same time.

6.4 Evaluation of Constraints

This section explains how a tModel call is actually executed and how constraints are evaluated in this process. At first, constraints from all different sources are combined conjunctively into one single combined constraint, called *main constraint*, using the AND operator. This constraint is passed as an input to the tModel call. Its evaluation consists of two phases: First, it is transformed into disjunctive normal form (DNF), and conflicts are resolved. Second, UDDI is queried for services assigned to the given tModel, and the services are invoked considering the main constraint.

6.4.1 Preprocessing of Constraints

First, the main constraint is transformed into DNF. Notice that the same constraint can now be present multiple times in the transformed constraint. Afterwards, all constraints of an *AND term*, i.e., a term only containing AND operators, are sorted according to their time of evaluation. The order is: metadata, location, mode, reply, and result constraints.

Then, the main constraint is checked for conflicts. Only conflicts within a single AND term are resolved in this phase, conflicts between different AND terms are resolved later during the invocation phase. Within an AND term, a conflict occurs if it either contains more than one mode constraint or more than one result constraint. For mode constraints, this is obvious. For result constraints, there are some rare situations where several result constraints would make sense. But, as we see no real benefit, two or more result constraints per AND term are prohibited.[6] Of course, conflicts between metadata, location, or reply

[6]The implementation would be straightforward although requiring many even though simple case discriminations.

constraints are possible in principle, e.g., an AND term that contains metadata constraints with contradictory XPath queries. Detecting this type of conflict would require a detailed investigation of the XPath queries.

Conflicts are resolved by keeping only the constraint with the maximum priority and removing all other conflicting constraints. Priorities range from 0 (minimum) to ∞ (maximum) and they can be assigned to a term by its creator, e.g., the consumer or a Web service. An additional, implicit prioritization is given by the sequence of the terms in their XML representation. The later a term is defined there, the less its priority is. If two terms have the same explicit priority, their implicit priority decides which one has the higher priority.

Finally, identical mode and result constraints that are contained in several AND terms because of the transformation into DNF are merged.[7] The resulting terms are called *merged AND terms*. Without merging, a service platform would evaluate identical mode and result constraints multiple times which would result in a different result. Only mode and result constraints are considered for merging because, unlike the other constraint types, they are restrictions on sets of services respectively replies, not on single services or replies. Therefore, the result of the main constraint is only modified by duplicating them when transforming the main constraint into DNF.

6.4.2 Invocation of Web Services

After the main constraint has been preprocessed, UDDI is queried for all information about services assigned to the given tModel. These services as well as their metadata are stored in a *services list*. Initially, there is one such services list for every merged AND term, i.e., the initial list of Web services is duplicated as many times as there are merged AND terms.[8] At first, all these lists are identical. But while the constraints are processed, the lists start to get different because different constraints are applied to them in different merged AND terms. In the following, services which do not fulfill a condition are removed from a services list. Preferences are used to sort this list.

Now, metadata constraints are applied to the services list of their merged AND term, followed by location constraints. For the evaluation of location constraints for dynamic services, all available service hosts are retrieved from UDDI first. Then, the location constraints are used to filter and sort this list of service hosts (similar to services lists). Notice that the location constraints in the merged AND terms are probably different so that the service hosts lists will probably be different, too. For each merged AND term, the corresponding service hosts list is assigned to all dynamic services of this term.

Next, all mode constraints of the main constraint are evaluated in parallel, i.e., Web services are invoked as specified by the mode constraints considering all relevant services lists. As a consequence of the merging of identical mode constraints, services lists from

[7] Basically, merging means factoring out identical mode and result constraints.

[8] In our implementation, the services lists are not duplicated for efficiency reasons. Instead, only one list is used in which all necessary data is stored, separated by the merged AND term the data belongs to.

more than one merged AND term may have to be considered. For each invocation of Web services based on a single mode constraint, the corresponding services list is processed sequentially, starting with the service at the top (which has the highest priority). Thereby static services are invoked only once, dynamic services can be invoked as often as there are service hosts in their service hosts list (service host are chosen according to their priority).

Every time the reply of a Web service is received, all relevant reply constraints are applied to it. Notice that the Web service may be contained in several services lists, so there can be more than one merged AND term with relevant reply constraints. The service platform must also check whether the invocation phase must be ended. This is the case if the result constraint with the highest priority is fulfilled. After the invocation phase has ended, all outstanding requests are aborted and all replies are returned to the calling Web service.

6.5 Related Work

The eFlow system [CS01] models composite services as business processes, specified in eFlow's own composition language and provides techniques similar to DSS. With dynamic service discovery, a composite service searches for services based on available metadata, its own internal state, and a rating function. Multiservice nodes allow several services to be invoked in parallel similar to DSS mode and result constraints though with different termination criteria. In contrast to eFlow, DSS allows the combination of all these different constraints in a flexible way. In addition, eFlow does not utilize standards like UDDI or WSDL for its adaptive techniques.

In [MS03], an agent-based architecture is presented that provides service selection based on a rating system for Web services. Ratings are gathered by monitoring the usage of Web services by clients, fetching rating information from other agents, and a feedback mechanism. The CB-SeC framework [MM03] follows a similar approach though ratings are calculated using so-called context of interest functions based on context information about consumers and services. In Jini [Wal01], clients utilize a lookup service to discover services based on the Java interfaces they implement and service attributes. The lookup service's attribute search is limited to searching only for exact matches [MJ01]. Extensions have been proposed to support, e.g., attributes providing context information about services [LH03] or more sophisticated match types [MJ01]. Based on WSDL, WSIF [WSI03] allows a Web service to select a specific port of a service it wants to invoke, i.e., its actual access point and the communication protocol and message format to use, at runtime. The selection is limited to the information provided by WSDL documents as no service repositories like UDDI are considered.

Chapter 7

Context-Aware Adaptable Services

In this chapter, we present our context framework that facilitates the development and deployment of context-aware adaptable Web services. Web services are provided with context information about clients that may be utilized to provide personalized behavior. Context is extensible with new types of information at any time without any changes to the underlying infrastructure. Context processing is done by Web services, context plugins, and context services. Context plugins and context services pre- and postprocess Web service messages based on the available context information. Both are essential for automatic context processing and automatic adaption of Web services to new context types without the necessity to adjust the Web services themselves. We implemented the context framework within the ServiceGlobe system, our open and distributed Web service platform. Parts of this chapter have already been presented in [KK04b, KSKK03]. A demo of the our context framework was given at the EDBT'04 conference [KK04a].

The remainder of this chapter is structured as follows: In Section 7.1, we motivate the usage of context-aware adaptable Web services. Section 7.2 presents an adjusted version of the motivating scenario, which is used as an example in this chapter. In Section 7.3, our context framework is described. Several types of context information available in our framework are presented in Section 7.4. Finally, Section 7.5 presents related work.

7.1 Motivation

Today, consumers want to use several ways to access information services on the Internet, e.g., browsers on desktop computers, PDAs, or cell phones. As the trend to an increasing number of ubiquitous, connected devices—called pervasive computing—continues to grow, the heterogeneity of client capabilities and the number of methods for accessing information services also increases. Nevertheless, consumers expect Web services to be accessible from all of these devices in a similar fashion. They also expect that Web services are aware of their current environment, e.g., the type of device they are using, their preferences, or their location. Generally, this kind of information is called *context*.

More precisely, in our work context constitutes information about clients and their

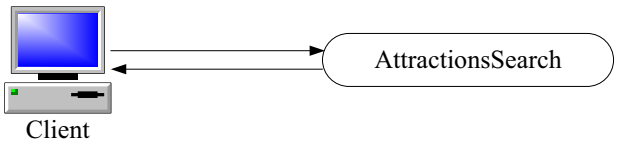


Figure 7.1: Motivating Scenario: No Context Processing

environment that may be used by Web services to provide clients with customized and personalized behavior. So, context contains, e.g., a consumer's name, address, and current location, the type of client device (hard- and software) the consumer is using, or all kinds of preferences regarding the communication, the format of the Web services' replies, or—in case of information services—the maximum amount of data that should be returned. Web services use such context information to adjust their internal control flow as well as content and format of their replies.

In this chapter, we present a context framework that facilitates the development and deployment of context-aware adaptable Web services. The framework consists of two main parts: a distributed infrastructure, which transmits context between clients and Web services and manages the context processing, and the context types, which are the supported types of context information and which are extensible at any time.

The actual context processing is done by three components: Web services themselves, context plugins, and context services. Context plugins and context services are provided by the context framework, and they pre- and postprocess Web service messages based on available context information. Both components are essential for automatic context processing, i.e., for processing context without the support of the original Web services, and automatic adaption of Web services to new context types.

Context plugins are basically Java objects implementing a dedicated interface. They are loaded by the service platform during startup, and they support locally executed Web services, i.e., a context plugin cannot be used if it is not locally available. Context services, on the other hand, are Web services (implementing a special interface defined using the WSDL standard), and they might be available anywhere on the Internet. They provide similar functionality as context plugins but need not be locally available.

In our context framework, the set of context types is extensible at any time without any changes to the underlying infrastructure. By adding appropriate context services or context plugins, new context types are used instantly and automatically. If using these new context types is achieved by pre- and postprocessing Web service messages, the implementations of Web services need not be adjusted. This way Web services may even utilize context information that was unknown at the time of their development.

7.2 Motivating Scenario

For the following sections, we extend the travel agency scenario that we presented in Section 5.1. Assume that one of the travel agency's providers, e.g., the provider of the

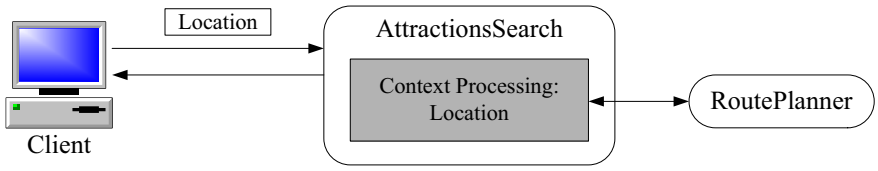


Figure 7.2: Motivating Scenario: Internal Context Processing

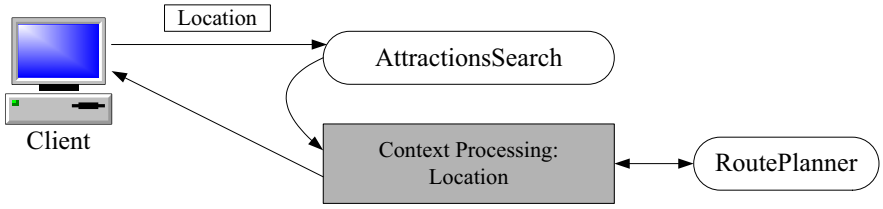


Figure 7.3: Motivating Scenario: External Context Processing

AttractionsSearch Web service (depicted in Figure 7.1), starts to allow the use of context information with its services. The provider extends its AttractionsSearch Web service with a component that uses a consumer's location to include driving directions into the Web service's reply (the necessary data is retrieved from an appropriate Web service, e.g., the Web service RoutePlanner). Therefore, the implementation of the AttractionsSearch Web service has to be changed (as depicted in Figure 7.2).

Consumers are accepting this new feature. So, the provider of another Web service, e.g., HotelsSearch, also wants to enable context processing for its service. Thus, the provider must adjust the implementation of its Web service to utilize location context information. Although the Web services AttractionsSearch and HotelsSearch share the same functionality, both implement their own version of it. So, both Web service implementations have to be changed. Furthermore, if one of these providers wants to extend its Web service again, e.g., to use context information about the consumers' clients, it must adjust the Web service's implementation a second time.

The functionality to process and use context information should not be (deeply) integrated into the Web services themselves. Instead, the different functional duties should be isolated, and they should be implemented in separate components. These components should be provided externally, as depicted in Figure 7.3. Their usage must be transparent for Web services, and they must be used automatically if a Web service's request contains context information. Additionally, they must provide a generic solution, i.e., the same component must be usable for a variety of Web services, e.g., the AttractionsSearch Web service as well as the HotelsSearch Web service.

Our context framework presents a solution for the problems outlined above as it has precisely the desired properties: It is transparent for Web services, context processing components are automatically deployed, and these components can be used generically (of

course, they must be implemented properly). In Section 7.4, we describe how the above scenario can be implemented with our context framework.

7.3 Context for Web Services

In the literature, there are a number of different definitions and uses of the term context [MM03, DSA99, SAT+99, HBS02, PLdH03, IRRH03, RB03]. In our work, context encompasses all information about the client of a Web service which may be utilized by the Web service to adjust execution and output to provide the client with customized and personalized behavior.

Context is different from the parameters of a Web service: First of all, the same context information is interesting for a number of Web services whereas parameters are only used by the exact Web service they belong to. As a consequence, context can often be evaluated automatically, e.g., by the service platform. This simplifies the development of Web services as the evaluation of such context does not need to be integrated into the Web services themselves. A further difference is that context information is optional whereas parameters are mandatory. Context information does not need to be passed to a Web service and if it is, the Web service does not necessarily need to understand and process it.

7.3.1 Context Infrastructure

In our framework, context is transmitted as a SOAP header block within the SOAP messages that Web services receive and send (see Figures 7.4 and 7.5 for an example). Legacy Web service platforms that do not support context information may ignore this specific header block (in conformity with the SOAP standard).[1] As context information is optional, Web services executed on legacy platforms are nevertheless able to process such requests, but they lose the benefit of the context information.

Analogous to a SOAP header, context consists of several *context blocks*. Each context block is associated to one *context type* which precisely defines the type of context information the context block is allowed to contain. At most one context block is allowed for a specific context type, i.e., no two context blocks can be associated to the same context type. The context in Figure 7.4 contains two context blocks: one associated to the context type Location (with information about a consumer's current location) and another one of context type Client (with information about the client's hard- and software). More information about the context types supported by our context framework is given in Section 7.4.

Every context type has a unique *context type identifier*. This identifier is equal to the qualified name of the XML element that represents corresponding context blocks, i.e., identifiers must be valid qualified names. The qualified name of an XML element is composed of its namespace and element name. For example, in Figure 7.5 there is a context

[1] If the attribute `mustUnderstand` is set in a context header block, Web service platforms must process the context or fail processing the message [Mit03].

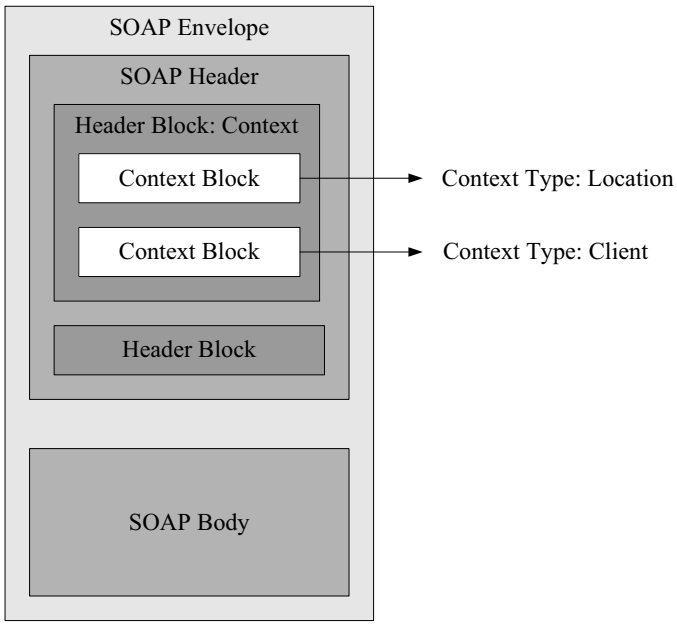


Figure 7.4: Context within a SOAP Message

block element `Client` with namespace `http://sg.fmi.uni-passau.de/context`. Consequently, the associated context type is `http://sg.fmi.uni-passau.de/context:Client`. We omit the namespace part in the following and reference context blocks (and context types) only by the corresponding element names.

Context types are basically used to distinguish context blocks. For example, if a Web service wants to access information of its context, it specifies the type of context information it wants to retrieve, i.e., a context type identifier. The context infrastructure determines the corresponding context block using this identifier and returns it. For the infrastructure, the knowledge of a context type identifier is sufficient for allowing access to the context type and for guaranteeing that a context contains at most one context block of any context type.

Though the infrastructure does not require the validation of a context block's content against the schema of its context type, it does provide the possibility for it, especially to free Web services themselves from this task. For this purpose, a context type has to be published in a UDDI repository as a tModel.[2] In the tModel, the identifier of the context type must be specified. Also, if content validation should be possible, an XML schema document must be referenced that defines the schema to which corresponding context blocks have to conform to. If the validation of a context block fails, the context block is marked as incorrect and not used further on.

Figure 7.6 gives an example of a tModel which defines the context type Location. The

[2]These tModels are also used by other parts of the context framework, see Section 7.3.4 for further details.

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
    <Context xmlns="http://sg.fmi.uni-passau.de/context">
      <Location>
        <address useType="Office">
          <addressLine keyName="Street" keyValue="60">
            Innstrasse 33
          </addressLine>
          <addressLine keyName="City" keyValue="40">
            D-94032 Passau
          </addressLine>
        </address>
      </Location>
      <Client>
        <DeviceDefaults>
          http://example.org/context/device/PDA
        </DeviceDefaults>
        <Hardware>
          <ScreenSize>320x200</ScreenSize>
          <IsColorCapable>Yes</IsColorCapable>
        </Hardware>
      </Client>
    </Context>
  </env:Header>
  <env:Body>
    <!-- serialized object data -->
  </env:Body>
</env:Envelope>
```

Figure 7.5: SOAP Message with a Context Header Block

context type's identifier `http://sg.fmi.uni-passau.de/context:Location` is specified as a keyedReference in the tModel's identifierBag. In the categoryBag, it is specified that the tModel is derived from the tModel `ContextType`, which serves as base tModel. The overviewDoc entry contains a URL that links to the XSL schema document which defines the context type's schema.

7.3.2 Life-Cycle of Context Information

The life-cycle of a Web service's context, illustrated in Figure 7.7, starts at a client's site: First, the client gathers all relevant context information and inserts it into the SOAP request as a context header block. Then, the request is sent to the host executing the Web service.

```
<tModel>
  <name>Location Context Type</name>
  <overviewDoc>
    <overviewURL useType="xmlSchema">
      http://sg.fmi.uni-passau.de/context/context-location.xsd
    </overviewURL>
  </overviewDoc>
  <identifierBag>
    <keyedReference keyName="ContextTypeID"
         keyValue="http://sg.fmi.uni-passau.de/context:Location"
         tModelKey="uddi:serviceglobe:identifier:contexttype"/>
  </identifierBag>
  <categoryBag>
    <keyedReference keyName="derivedFrom:ContextType"
         keyValue="uddi:serviceglobe:categorization:contexttypes"
         tModelKey="uddi:uddi.org:categorization:derivedFrom"/>
    <keyedReference keyName="uddi-org:types"
         keyValue="categorization"
         tModelKey="uddi:uddi.org:categorization:types"/>
  </categoryBag>
</tModel>
```

Figure 7.6: tModel for the Location Context Type

After the request has been received by a Web service platform, the context is extracted by the context framework and provided to the invoked Web service as its *current context*. During its execution, the Web service can access and modify this current context using the Context API provided by the framework. For example, in Figure 7.7 the first context block is modified (illustrated by the color change to gray) and a new context block (the third, black rectangle) is inserted.

When the Web service invokes another service during its execution, its current context is automatically inserted into the outgoing request. The response to such a request may also contain context information. In this case, the Web service can extract the interesting parts of the context data from the response and insert them into its current context.

After the Web service's termination, its (possibly modified) current context is automatically inserted into its response and sent back to the invoker. If the invoker is a client, as in Figure 7.7, it may integrate portions of the returned context into its local context (for use in future requests). Furthermore, the returned context may be utilized by the client to adjust the Web service's response.

In the entire context life-cycle, potential privacy and security issues have to be considered. For example, clients should be able to specify what modifications a Web service is allowed to perform on the context and also what parts of the context the Web service is

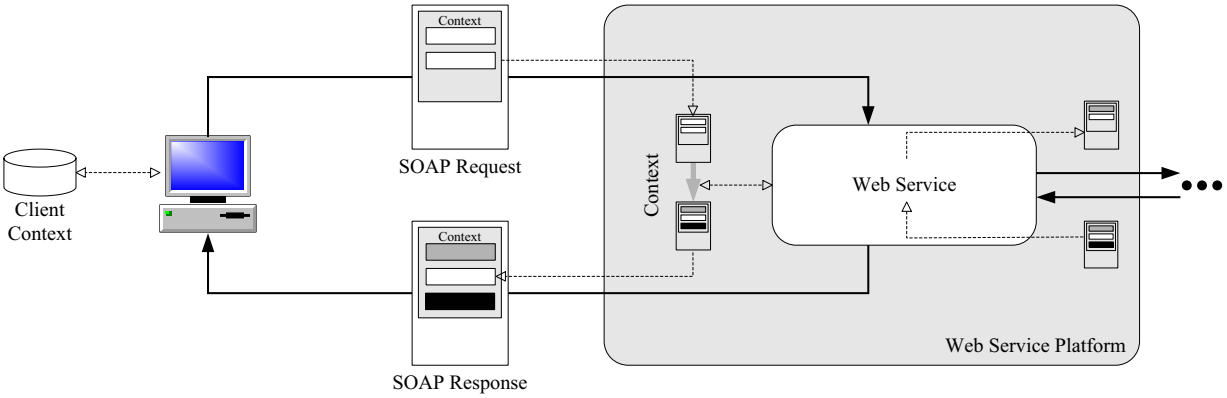


Figure 7.7: Context Life-Cycle

allowed to insert into requests to other Web services. Furthermore, the policies must state if and how a client is allowed to modify its local context. These issues are partly considered in Section 7.3.4. Elaborate privacy and security policies are beyond the scope of this thesis.

7.3.3 Context Processing

In our context framework, we distinguish two types of context processing: explicit processing by Web services or clients and automatic processing by the context framework.

Explicit Context Processing

Explicit processing means that Web services or clients directly access the context contained in a SOAP message using the framework's Context API and, consequently, that the context processing functionality is part of their code. Thus, there is a tight coupling between such Web services and clients and the context types they are able to process. That is, such Web services and clients can only utilize context types that were known and integrated at the time of their development. A further disadvantage is the additional coding effort as the same or at least similar context processing functionality is basically contained in many Web services and clients. A strict separation of concerns is also missing. An advantage of this type of context processing is that Web services and clients have full control over how the context information influences their control flow and their replies. Additionally, they can access the context information to modify it.

An example of explicit processing is the client we implemented. It processes the returned context information to finally adjust the response to its device capabilities, e.g., by using stylesheet information inserted into the returned context.

Automatic Context Processing

Automatic context processing means that SOAP messages are pre- and/or postprocessed (from a Web service's point of view) based on the context information they contain. Automatic context processing is done by the context framework, i.e., Web services are not involved in it. As a consequence, the context processing task is moved from the Web services to the service platform, and the coding effort for Web services is reduced. A disadvantage is, of course, that only a Web service's requests and responses can be modified; its internal process flow cannot be adjusted by this means.

There are four different points in time at which context is processed automatically (see Figure 7.8): First, the incoming SOAP request of an invoked Web service is preprocessed (1) based on the context in the request. Furthermore, whenever the Web service invokes other services (using the invocation manager), outgoing messages, i.e., requests to other services, are postprocessed before they are actually sent (2) and incoming messages, i.e., responses to outgoing requests, are preprocessed before they are returned to the Web service (3). Finally, the outgoing response of the invoked Web service is postprocessed (4) based on the service's current context (which might be a modified version of the received one). To sum up, this means that all messages to and from an invoked Web service can be modified based on context information.[3]

Of course, modification of messages also implies that the messages' content can be modified, e.g., the content of a Web service's reply. For example, in our demonstration at the EDBT'04 conference [KK04a], we used automatic context processing to convert the price information within a Web service's reply content into the currency of the consumer's location.

In the following, we refer to the procedures when a Web service's message is pre- or postprocessed as *context operations*, and we call them *PreprocessRequest* (1), *PostprocessMessageRequest* (2), *PreprocessMessageResponse* (3), and *PostprocessResponse* (4), respectively.

In every context operation, automatic context processing is done by processing the context blocks of the SOAP message's context in arbitrary order. Consequently, during the processing of a context block, no assumptions can be made on the processing state of any other context block, i.e., if some other context block has already been processed or not.

After selecting an arbitrary, not yet processed context block, the context framework determines its context type. For every context type, the context manager (see Figure 7.8) manages a list of components capable of processing context information of the associated type. The actual processing of the selected context block is delegated to these components, which are described below. There are several ways to configure which of these components should actually be used for processing and in what order. Details are given in Section 7.3.4.

[3] In our work, we assume a request-response message exchange pattern as, e.g., used for RPC. If a different exchange pattern is used, e.g., if a Web service does not return a response, the corresponding processing steps are omitted.

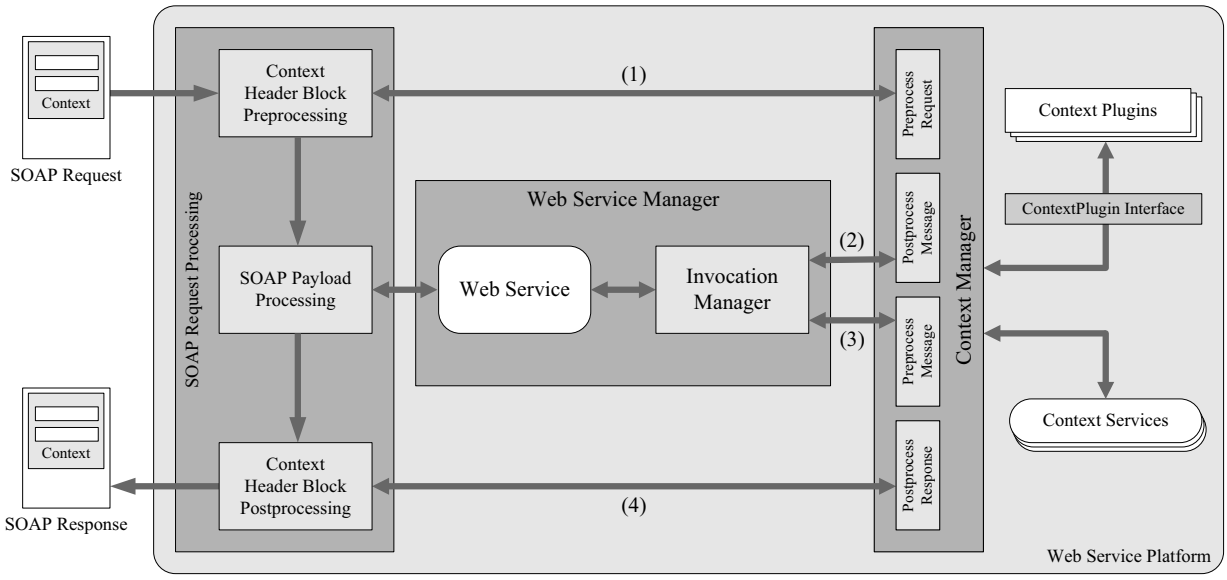


Figure 7.8: Components for Context Processing

Components for Automatic Context Processing

The context framework delegates automatic context processing to two types of components, as shown in Figure 7.8: context plugins and context services. *Context plugins* are basically Java objects implementing a special Java interface. They must be installed locally at a host, and they are loaded by the service platform during startup. *Context services* are Web services that implement the `ContextService` interface. This interface, defined using the WSDL standard, describes the four context operations a context service should implement. A UDDI tModel `ContextService` is provided that links to the WSDL document with the interface description. Context services should refer to this tModel when published in UDDI. In their bindingTemplate entries, context services may also specify which context operations they actually support.

Every component, i.e., every context plugin and every context service, is associated with one context type, and it is used to process context information of this type only. When invoked, a component requires two parameters: The first one is a context block of the component's associated context type. The second one depends on the context operation that is invoked on the component: It is the request to a Web service, the service's response, an outgoing request of the Web service (to invoke another Web service), or an incoming response of such an invoked service.

Context services are very similar to context plugins as they basically implement the same interface. Both enable automatic processing of context information and are essential for the easy extensibility of context. On the other hand, context services are Web services. They need not be installed locally, as context plugins must be, but can be available anywhere on the Internet. If implemented as dynamic services, as it is possible in the Ser-

viceGlobe system, context services may be loaded dynamically and on demand, and they may be executed on the local host.

Besides the reduction of the coding effort, context plugins and context services have the advantage that they constitute a generic solution. A context plugin or service for a specific context type can be used for a variety of Web services without any need for specific adjustments. Even more important, Web services can now utilize context types they do not support themselves. This is also beneficial for legacy Web services which cannot be modified.

7.3.4 Context Processing Instructions

In Section 7.3.3, several components for context processing were introduced. A context block in a SOAP request is possibly processed by context plugins, context services, and the invoked Web service itself. Therefore, rules of precedence are required and also information about which components should actually be used for processing. Additionally, the same context block is probably not only processed at a Web service's local host but also at hosts on which the Web service invokes other services, as context is inserted into outgoing messages.

Precautions have to be taken to prevent these problems and ambiguities. In our framework, *context processing instructions* are used for this purpose.[4] If no context processing instructions are specified, defaults are used: Context plugins are invoked as configured locally at the service platform (default here is alphabetical order according to the class name). Context services are not used by default. Invoked Web services themselves can always process their context information.

Context processing instructions are specified within a `ContextProcessingInstructions` element, as depicted in the example in Figure 7.9 (for the moment, we ignore the enclosing UDDI elements). For every context type, they can contain at most one `ContextType` child element. Within this child element, instructions for the corresponding context type are specified. Currently, component instructions and processing guidelines can be specified. But we are still investigating these issues and are going to consider them in more detail in future work.

Component Instructions

With component instructions, context plugins and context services that should be used for processing context information of the enclosing context type and their execution order are specified. Context plugin instructions must be defined using `ContextPlugin` elements, context service instructions using `ContextService` elements. If several context plugins and services are specified, they are executed in the same order as they are specified.

[4] Apart from their name, context processing instructions and XML processing instructions are very different. Context processing instructions are ordinary context information just like location or client context information.

```
<tModelInstanceInfo
    tModelKey="uddi:serviceglobe:context:processing-instructions">
  <instanceDetails>
    <instanceParms><![CDATA[
      <?xml version="1.0" encoding="utf-8" ?>
      <pi:ContextProcessingInstructions
          xmlns:pi="urn:serviceglobe:context">
        <pi:ContextType ID="http://sg.fmi.uni-passau.de/context:Location">
          <pi:ContextService>
            <pi:AccessPoint useType="http">
              http://example.org/services/CurrencyConverter
            </pi:AccessPoint>
            <pi:ContextOperations>post</pi:ContextOperations>
          </pi:ContextService>
          <pi:ProcessingGuideline>
            <pi:ServiceHost>Next</pi:ServiceHost>
            <pi:ComponentTypes>
              ContextPlugin+ContextService
            </pi:ComponentTypes>
          </pi:ProcessingGuideline>
        </pi:ContextType>
        <pi:ContextType ID="http://sg.fmi.uni-passau.de/context:Client">
          <pi:ContextPlugin>
            serviceglobe.context.plugins.StylesheetFinder
          </pi:ContextPlugin>
        </pi:ContextType>
      </pi:ContextProcessingInstructions>]]>
    </instanceParms>
  </instanceDetails>
</tModelInstanceInfo>
```

Figure 7.9: Context Processing Instructions in the tModelInstanceInfo entry of a bindingTemplate

In the example in Figure 7.9, a context service with access point `http://example.org/services/CurrencyConverter` is used for processing Location context. The context framework supports several types of access points: SOAP-HTTP URLs (as in the example), ServiceGlobe URLs, or bindingTemplate UUIDs referencing context services published in a UDDI repository. The instructions in the example also state that only the context operation *PostprocessResponse* (keyword `post`) should be invoked. Other keywords are `pre` for *PreprocessRequest*, `postmessage` for *PostprocessMessageRequest*, and `premessage` for *PreprocessMessageResponse*. Furthermore, the context plugin StylesheetFinder (specified

by its class name) is used for processing Client context (for a description of this plugin, see Section 7.4).

Processing Guidelines

With processing guidelines, the types of components that should be used to process a certain context block are specified as well as the hosts at which the context block should be processed. A processing guideline is defined using the `ProcessingGuideline` element (which must be a child element of the `ContextType` element for which the guideline is specified). Figure 7.9 shows an example. The child element `ServiceHost` specifies the host at which corresponding context blocks should be processed, and the child element `ComponentTypes` specifies the actual component types that should be used for processing.

Possible values of the `ServiceHost` element are `Next` and `All`. (The meaning of this element is similar to the `role` attribute that can be used in SOAP header blocks [Mit03].) If `Next` is specified, only the next host should receive and process a context block. For that reason, the context block is not included into outgoing requests of the invoked Web service. When using `All`, the corresponding context block is inserted into outgoing requests, and all hosts that receive it may also process it. Possible values that can be used within the `ComponentTypes` element are `ContextPlugin` and `ContextService`. Their meaning is obvious. Both values can be combined using the `+` operator. In this case, both components are used sequentially for processing. In the example of Figure 7.9, context plugins are applied first. Then, context services are invoked. Without any processing guidelines, defaults are used: `Next` for `ServiceHost` and `ContextPlugin` for `ComponentTypes`.

Web services themselves can always access and process any context block passed to them even if a context block was already processed by a context plugin or context service. Obviously, it would be possible to remove a processed context block from the context to prevent Web services from processing it a second time. But in this case the context block would only be used to modify the Web services' messages (due to the constraints of context plugins and context services).

Providing Context Processing Instructions

There are several possibilities to make context processing instructions available to the context framework. The first two possibilities are especially useful if it should be enforced that only particular context plugins and context services are used to process context blocks of certain context types.

First of all, the *context itself* can contain context processing instructions. For this purpose, the instructions, e.g., the `ContextProcessingInstructions` element of the example in Figure 7.9, are inserted into the context as a self-contained context block. The context block is then processed by a special context plugin provided by the context framework.

Second, a *Web service's UDDI metadata* may be annotated with context processing instructions. Therefore, the bindingTemplate entry of the Web service must contain a tModelInstanceInfo entry that specifies the instructions. An example is shown in Figure 7.9.

```
<businessService>
  <name>CurrencyConverterContextService</name>
  <categoryBag>
    <keyedReference keyName="ContextService"
         keyValue="true"
         tModelKey="uddi:serviceglobe:interfaces:contextservice"/>
    <keyedReference keyName="ContextType"
         keyValue="http://sg.fmi.uni-passau.de/context:Location"
         tModelKey="uddi:serviceglobe:categorization:contexttypes"/>
  </categoryBag>
</businessService>
```

Figure 7.10: UDDI Metadata of a Context Service

The context processing instructions are contained within the `instanceParms` element (as a string, according to the UDDI standard[5]). The format of the context processing instructions is the same as if used within the context of a SOAP request.

Providers or developers of Web services can use this second option to specify context services that should be used for processing certain context blocks. Even operators of hosts executing Web services may utilize UDDI this way to force the use of certain context plugins or context services with Web services executed at their hosts.

The third possibility is different to the preceding ones: Instead of relying on explicitly specified context processing instructions, the context framework uses available UDDI metadata to automatically determine available context services for context processing. Context plugin instructions and processing guidelines cannot be determined this way.

Just as ordinary Web services, context services may be published in UDDI. Every context service that should be found by the context framework when searching for appropriate context services must be associated to two tModels: the tModel `ContextService`, which marks a Web service as context service, and the tModel `ContextType`. The `ContextType`'s association contains a parameter that specifies the context type the context service is able to process. For an example, see Figure 7.10. When processing a context block, the framework queries UDDI for all services associated to both tModels and having the correct parameters. As all of these services provide semantically equivalent functionality, one of them is chosen randomly. For the future, we are going to consider the utilization of ServiceGlobe's dynamic service selection in this process.

7.4 Context Types

In this section, several context types provided by our context framework are explained as well as some of the context services and context plugins we implemented. Our framework

[5] According to [UDD00], the content of an `instanceParms` element must be of type string. The suggested format is a namespace-qualified XML document.

```
<tModelInstanceInfo
   tModelKey="uddi:serviceglobe:contexttype:client:stylesheets">
  <instanceDetails>
    <instanceParms><![CDATA[
     <?xml version="1.0" encoding="utf-8" ?>
     <UDDIinstanceParmsContainer
         xmlns="urn:uddi-org:policy_v3_instanceParms">
       <Stylesheet deviceType="http://example.org/context/device/PDA">
         http://example.org/service-stylesheet-pda.xsl
       </Stylesheet>
       <Stylesheet deviceType="http://example.org/context/device/Desktop">
         http://example.org/service-stylesheet-desktop.xsl
       </Stylesheet>
      </UDDIinstanceParmsContainer>]]>
    </instanceParms>
  </instanceDetails>
</tModelInstanceInfo>
```

Figure 7.11: UDDI Metadata: Stylesheets for a Web Service's Reply

```
<ReplyProperties xmlns="urn:serviceglobe:context">
  <Stylesheet>
    http://example.org/service-stylesheet-desktop.xsl
  </Stylesheet>
</ReplyProperties>
```

Figure 7.12: Context Block of Context Type ReplyProperties

is not limited to this set of context types, context plugins, and context services. As it is extensible at any time, new context types can be added by inserting corresponding context information into the context and providing appropriate context services and/or context plugins for processing it. Neither the context framework nor Web services must be adjusted for this.

An important context type is *Location*. It contains information about the consumer's current location, e.g., the consumer's current address, GPS coordinates, country, or local time and time-zone. An example was shown in Figure 7.5. Location context may also include semantic location information, e.g., that a consumer is currently at work. We implemented, for example, a context service CurrencyConverter that converts price information included in a Web service's reply to the currency of the consumer's location.

Client context information comprises data about a client's device. It includes information about hardware, e.g., processor type or display resolution, as well as software, e.g, operating system or Web browser type and version. An example of such a context

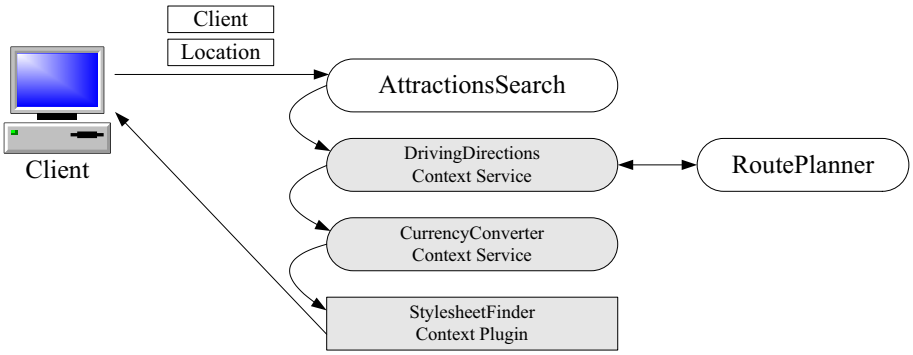


Figure 7.13: Motivating Scenario: Context Processing with the Context Framework

block was also shown in Figure 7.5. Two schemas are supported for this context type: a rather simple one as used in Figure 7.5 and an RDF-based one as defined in the CC/PP standard [KRW+04] of the W3C.

The main purpose of this context type is to allow Web services to adjust their output to the client device's properties. For example, Web services from the information systems area, which often query data from a DBMS, can use this context information to optimize their database queries and to query only data that can actually be displayed at the client. The Amazon Web service [Ama] is an example of such a Web service. In its replies, it includes several lengthy customer reviews. If viewing a reply of this Web service on a PDA or cell phone, the inclusion of these reviews is rather pointless as they require too much space. Optimally, the corresponding data should not be retrieved from the DBMS in the first place. But if this is not possible, e.g., because modifications of the Web service are impossible, context services can be used to adjust the reply of the Web service.

We also implemented a context plugin StylesheetFinder which uses the Client context to provide the client with a stylesheet that can be used to format the Web service reply. A Web service must specify XSL stylesheets that should be used for the various client types in its UDDI metadata. Figure 7.11 shows an example of such metadata. Based on this metadata and the Client context information, the plugin inserts a new context block of type *ReplyProperties*. This context block, see Figure 7.12 for an example, is processed by clients. They use the specified stylesheet to transform the reply's XML data into HTML.

The *Consumer* context type contains information about the consumer invoking the Web service, e.g., name, email address, and so on. Although it is very important, it can actually be used only by the Web services themselves in a sensible way.

The *Connection Preferences* context type allows to specify properties of the connections to Web services. It was added by implementing a context service ConnnectionPreferencesService. Based on the content of the corresponding context block, the context service compresses and decompresses Web service messages using gzip [Deu96] or the XML compressor XMill [LS00]. The context service could, e.g., be extended to support encryption, too.

With these context types, the motivating scenario in Section 7.2 can be implemented as depicted in Figure 7.13. Although the AttractionsSearch Web service was not modified, the reply that the client receives contains personalized, context-dependent information. Client and Location context information is processed automatically by two context services and one context plugin. After AttractionsSearch generated its reply, the context service DrivingDirections uses the RoutePlanner Web service to insert driving directions from the client's location into the reply. Then, the context service CurrencyConverter converts all price information included in the AttractionsSearch's reply to the currency of the consumer's location. CurrencyConverter determines the XML elements that contain such price information from the WSDL document of AttractionsSearch, which it retrieves from the UDDI repository. This document describes the type of AttractionsSearch's reply using XML schema. Finally, the context plugin StylesheetFinder chooses an XSL stylesheet that fits best to the current client device and inserts this information into the reply. After the client receives the reply, this stylesheet information is used to display the XML reply on the specified device in the appropriate way.

We also implemented clients for different types of client devices, e.g., Java-based clients for PDAs and cell phones. They are used to demonstrate the usefulness and the advantages of context information based on the example Web services of our motivating scenario. Furthermore, we implemented a Web-based client. With this client, the influence of various types of context information can be investigated in more detail. A demonstration of our context framework and these clients was presented at the EDBT'04 conference [KK04a].

7.5 Related Work

There are several technologies which are related to our context processing architecture, i.e., the automatic context processing of context by successively invoking context plugins and context services. The Chain of Responsibility design pattern [GHJV97], for example, describes how to decouple the receiver of a request from the sender by chaining the receiving objects and passing the request along the chain until an object handles it. On the other hand, in our framework several receivers, i.e., context plugins and services, can process the same request. Aspect-oriented programming (AOP) [KLM+97] allows the modification of applications with so-called aspects. Aspects are modular units of functionality that are used across the application's code. They are woven into an application's code at so-called pointcuts, thereby allowing to transparently extend, e.g., objects with new functionality. This is similar to the way Web services are extended with new context processing functionality using context plugins and services. In Java, AOP is supported, for example, by the AspectJ toolkit [KHH+01] or in the J2EE application server JBoss [JBo].

In CORBA, technologies like interceptors or smart proxies can be used to insert new functionality into existing applications. They are, for example, supported in IANA's Orbix [ION]. Interceptors and smart proxies have also been integrated into Java RMI [SMS02]. The Java Servlet specification [Suna] describes filters that could be used to intercept and modify messages. In the Axis framework [Axi], chains of handlers can be created. Re-

quests and responses are passed along these chains, and they may be modified by the chains' handlers.

Although our context framework shares similarities with the above approaches, there are also differences. In our framework, the context information contained within the request determines which context plugins and context services are actually invoked. Context plugins and services are not chained sequentially, and not all of them are invoked every time. With context services, our context framework is extensible at runtime, just by adding appropriate context information into requests. Furthermore, context plugins and services selection may depend on the consumer's preferences, which can be integrated into the client's context as context processing instructions. Thus, we facilitate fine-grained dynamic control over the context processing.

In the mobile computing area, context has been investigated for several years. Best known are the location-based services. The PLIM framework [PLdH03], for example, provides an infrastructure for the distribution and retrieval of location information of (Bluetooth-connected) mobile devices using a publish & subscribe mechanism. [IRRH03] presents a context model for pervasive systems based on the CC/PP standard and points out some limitations of this standard. [HHS+02] describes a system that builds a dynamic model of the environment where the locations of the environment's objects are updated using location sensors. Also, an event-based monitoring system is provided that allows applications to detect location changes and to query the relationship of objects regarding their location. In Jini [Wal01], a Java lookup service for services, extensions have been proposed to support search attributes that provide context information about services [LH03].

In the Web service area, there are several research projects that deal with context. The CB-SeC framework [MM03] is an agent-based architecture that provides service selection based on a rating system for Web services. The ratings are calculated using so-called context of interest functions and context information about consumers and services. Aura [SG02] is an architectural framework that models user tasks as coalitions of abstract services. Aura migrates such tasks from one environment to another one if the user's location changes. Also, tasks can be adjusted if the environment changes, e.g., if a provider for a currently used service disappears. In [DCMC01], the concept of dynamic bookmarks is presented. Dynamic bookmarks are descriptions of services that are bound to actual services based on a user's location. Clients use a dynamic bookmark service to update their dynamic bookmarks if their location changes. [DSA99] describes a distributed infrastructure to support context-aware applications based on context widgets. Context widgets gather context information from low-level sensors. They may also aggregate and interpret it. Applications use these widgets by subscribing to them. In [Men01], an architecture is presented that allows to develop applications where the application logic is decoupled of the UI based on an event-graph. With it, UIs for different client types can be developed independent of the implementation of the application itself.

Our focus in this thesis is on automatic and transparent processing of context and on easy extensibility of context types, not on how the context is stored locally at clients. A distributed storage system for context information is, e.g., presented in [RB03]. In this system, context data is pro-actively replicated and migrated on mobile devices, dependent

on the clients' behavior. In [EHL01], a context service is presented that is basically a storage for context information. Context sources deliver their information to this service. Applications access the context service to query for context information. [HBS02] discusses requirements for a representation format for context information and examines different existing formats. As result of the discussion, they present a novel RDF-based representation format. In [SAT+99], data about the environment is collected using collections of low-level sensors. This data is analyzed and the context—a set of two-dimensional vectors—is updated accordingly. Using a script language, actions can be defined based on context changes. Actions can, e.g., be commands to applications.

The need for Web service personalization also poses challenges to other areas of computer science. In the information systems area, preferences are gaining noticeable attention [Kie02, AW00, Cho02] as they are a way to support personalization of Web services, especially for information services that often use a DBMS as back-end. Preferences, also called soft constraints, are more like wishes: The result of a query should be a perfect match, but a best possible match is also acceptable. For example, [BKK03] shows how quality of service preferences can be considered in distributed query processing. Recently, approaches to integrate preferences into Web services have also been proposed [BKU03, KH02].

Chapter 8

Conclusions and Future Work

In this thesis, we addressed the challenges for data integration systems imposed by the large number of resources available on the Internet. We also introduced techniques that facilitate application integration based on Web services and support the development of Web services with regard to flexibility, reliability, and personalization.

We first presented the MDV system, our distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that information is stored near the users that need it and queries can be evaluated locally. By adding servers to (or removing servers from) the middle-tier as necessary, our system can be adjusted easily to varying workloads. In order to keep replicas up-to-date and initiate the replication of new and relevant information, MDV implements a scalable publish & subscribe algorithm. We described this algorithm in detail, showed how it can be implemented using a standard relational database management system, and presented the results of performance experiments conducted using our prototype implementation.

The MDV system was developed as part of the ObjectGlobe system, our open and distributed query processing system for data processing services on the Internet. We, therefore, used ObjectGlobe as an example client of MDV. The goal of ObjectGlobe is to establish an open marketplace in which data, function, and cycle providers can offer/sell their services. We gave details of ObjectGlobe's architecture, its lookup service, its QoS management, and its security system, including the different security measures to protect the providers' resources against unauthorized access and attacks of malicious external operators.

We also described the deployment of the MDV system within ObjectGlobe. In particular, we described the MDV lookup service, which provides global access to the metadata of registered providers and which is consulted by ObjectGlobe's query optimizer in order to find relevant resources to execute a query and to obtain statistics. We presented the MDV security provider, which is responsible for authentication and authorization of users and their query plans and which deploys the MDV system to store and administrate security information. We also showed how MDV's publish & subscribe mechanism is used to distribute authentication data throughout the ObjectGlobe federation so that it is available to all query optimizers, which need this information in order to generate valid query plans.

Next, we presented the ServiceGlobe system, our open and distributed Web service platform. It supports mobile code, that is, Web services can be distributed on demand and instantiated at runtime at arbitrary Internet servers participating in the ServiceGlobe federation. Also, it offers all standard functionality of a service platform like SOAP communication, a transaction system, and a security system.

We introduced dynamic service selection, which offers Web services the possibility to select and invoke services at runtime based on a technical specification of the desired service. We showed how constraints can be used to influence dynamic service selection. Using them, services can be selected based on the relevant metadata. After invocation, replies may be checked for defined properties and discarded, if necessary. Constraints also allow the specification of the number of services that should be invoked and how they should be invoked. Constraints may be specified directly when invoking Web services or in a Web service's context.

We presented our context framework, which facilitates the development and deployment of context-aware adaptable Web services. We introduced our context model and gave a detailed description of the framework's main parts: the distributed infrastructure, which transmits context between clients and Web services and which manages the context processing, and the context types, which are extensible at any time. We showed how the actual context processing is done by Web services themselves, context plugins, and context services. Context plugins and context services pre- and postprocess Web service messages based on available context information. Context processing instructions, specified within the UDDI metadata of Web services or within the consumers' context, are a means to specify the host to which context is transmitted and at which hosts and by which components it is actually processed. We also introduced a basic set of context types that are supported by our context framework.

For the future, the MDV system can be extended to support XML as data format and XPath/XQuery as rule language—particularly within the publish & subscribe algorithm. Also, the integration of and the support for Web service standards like UDDI and WSDL could be beneficial for MDV and its clients. For ServiceGlobe's context framework, additional context types can be investigated and context processing instructions can be studied in more detail. Further issues of interest in this area are elaborate privacy and security policies that enable clients to specify which Web services should receive their context and what operations these services are allowed to perform on it.

Bibliography

[ACK+02] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM (CACM)*, 45(11):56–61, 2002.

[ADLH+02] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B.LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra, J. Shewchuk, and D. Simon. Web Service Security (WS-Security). `http://www.ibm.com/developerworks/webservices/library/ws-secure`, April 2002.

[AF00] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 53–64, Cairo, Egypt, September 2000.

[Ama] Amazon.com. Amazon Web Services. `http://soap.amazon.com/`.

[ASS+99] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–61, Atlanta, GA, USA, May 1999.

[AW00] R. Agrawal and E. L. Wimmers. A Framework for Expressing and Combining Preferences. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 297–306, Dallas, TX, USA, May 2000.

[Axi] Axis Architecture Guide. `http://ws.apache.org/axis/java/architecture-guide.html`.

[BBB+02] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams. Web Services Conversation Language (WSCL). `http://www.w3.org/TR/2002/NOTE-wscl10-20020314`, March 2002. World Wide Web Consortium (W3C), W3C Note.

[BCF+03] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/2003/WD-xquery-20031112`, November 2003. World Wide Web Consortium (W3C), W3C Working Draft.

[BCV99] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *ACM SIGMOD Record*, 28(1):54–59, 1999.

[BDSN02] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 297–308, San Jose, CA, USA, February 2002.

[BG99] D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. `http://www.w3.org/TR/1999/PR-rdf-schema-19990303`, March 1999. World Wide Web Consortium (W3C), W3C Proposed Recommendation.

[BKK+99] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Pröls, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous Query Processing on the Internet. Technical Report MIP-9909, Universität Passau, Passau, Germany, 1999.

[BKK+00] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Pröls, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous Query Processing on the Internet. In *Proceedings of the International Workshop on Technologies for E-Services (TES)*, pages 247–268, Cairo, Egypt, September 2000.

[BKK+01a] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous Query Processing on the Internet. *The VLDB Journal: Special Issue on E-Services*, 10(1):48–71, 2001.

[BKK+01b] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, S. Seltzsam, and K. Stocker. ObjectGlobe: Open Distributed Query Processing Services on the Internet. *IEEE Data Engineering Bulletin*, 24(1):64–70, 2001.

[BKK03] R. Braumandl, A. Kemper, and D. Kossmann. Quality of Service in an Information Economy. *ACM Transactions on Internet Technology (TOIT)*, 3(4):291–333, 2003.

[BKU03] W.-T. Balke, W. Kießling, and C. Unbehend. Performance and Quality Evaluation of a Personalized Route Planning System. In *Proceedings of the Brazilian Symposium on Databases (SBBD)*, pages 328–340, Manaus, Brazil, October 2003.

[BLT86] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 61–71, Washington, DC, USA, May 1986.

[BPSM+04] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). `http://www.w3.org/TR/2004/REC-xml-20040204`, February 2004. World Wide Web Consortium (W3C), W3C Recommendation.

[Bra01a] R. Braumandl. *Quality of Service and Query Processing in an Information Economy.* PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, Passau, Germany, 2001.

[Bra01b] T. Bray. What is RDF? `http://www.xml.com/pub/a/2001/01/24/rdf.html`, January 2001. XML.com.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, March 2001. World Wide Web Consortium (W3C), W3C Note.

[CD99a] L. Cardelli and R. Davies. Service Combinators for Web Computing. *IEEE Transactions on Software Engineering (TSE)*, 25(3):309–316, 1999.

[CD99b] J. Clark and S. DeRose. XML Path Language (XPath) version 1.0. `http://www.w3.org/TR/1999/REC-xpath-19991116`, November 1999. World Wide Web Consortium (W3C), W3C Recommendation.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 379–390, Dallas, TX, USA, May 2000.

[CER02] F. Curbera, D. Ehnebuske, and D. Rogers. Using WSDL in a UDDI Registry, Version 1.07 - UDDI Best Practice. `http://www.uddi.org/pubs/wsdlbestpractices-V1.07-Open-20020521.pdf`, 2002.

[Cho02] J. Chomicki. Querying with Intrinsic Preferences. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, volume 2287 of *Lecture Notes in Computer Science (LNCS)*, pages 34–51, Prague, Czech Republic, March 2002.

[CK98] M. Carey and D. Kossmann. Reducing the Braking Distance of an SQL Query Engine. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 158–169, New York, NY, USA, August 1998.

[CS01] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.

[CZH+99] S. Czerwinsky, B. Zhao, T. Hodes, A. Joseph, and R. H. Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 24–35, Seattle, WA, USA, August 1999.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. `http://www.rfc-editor.org/rfc/rfc2246.txt`, January 1999. RFC 2246.

[DCMC01] S. Duri, A. Cole, J. Munson, and J. Christensen. An Approach to Providing a Seamless End-User Experience for Location-Aware Applications. In *Proceedings of the International Workshop on Mobile Commerce (WMC)*, pages 20–25, Rome, Italy, July 2001.

[Deu96] P. Deutsch. GZIP file format specification version 4.3. `http://www.rfc-editor.org/rfc/rfc1952.txt`, May 1996. RFC 1952.

[DFJ+96] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 330–341, Bombay, India, September 1996.

[DSA99] A. K. Dey, D. Salber, and G. D. Abowd. A Context-based Infrastructure for Smart Environments. In *Proceedings of the International Workshop on Managing Interactions in Smart Environments (MANSE)*, pages 114–128, Dublin, Ireland, December 1999.

[ebX] Electronic Business XML Initiative (ebXML). `http://www.ebxml.org/`.

[EHL01] M. Ebling, G. Hunt, and H. Lei. Issues for Context Services for Pervasive Computing. In *Proceedings of the Advanced Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, November 2001.

[Fal01] D. C. Fallside. XML Schema Part 0: Primer. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502`, May 2001. World Wide Web Consortium (W3C), W3C Recommendation.

[FJL+01] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 115–126, Santa Barbara, CA, USA, May 2001.

[FK99] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[FK01] D. Florescu and D. Kossmann. An XML Programming Language for Web Service Specification and Composition. *IEEE Data Engineering Bulletin*, 24(2):48–56, 2001.

[FKK96] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 Protocol. `http://home.netscape.com/eng/ssl3`, November 1996. Netscape Communications Corp.

[GHJV97] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1997.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 157–166, Washington, DC, USA, May 1993.

[GPVD99] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. `http://www.rfc-editor.org/rfc/rfc2608.txt`, June 1999. RFC 2608.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Han87] E. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 440–453, San Francisco, CA, USA, May 1987.

[HBS02] A. Held, S. Buchholz, and A. Schill. Modeling of Context Information for Pervasive Computing Applications. In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics (SCI)*, Orlando, FL, USA, July 2002.

[HCKW90] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 271–280, Atlantic City, NJ, USA, May 1990.

[HFPS99] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. `http://www.rfc-editor.org/rfc/rfc2459.txt`, January 1999. RFC 2459.

[HHS+02] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. *Wireless Networks*, 8(2-3):187–197, 2002.

[IBM] IBM Web Sphere. `http://www.ibm.com/websphere`.

[ION] IONA Technologies Inc. Orbix. `http://www.iona.com/products/orbix.htm`.

[IRRH03] J. Indulska, R. Robinson, A. Rakotonirainy, and K. Henricksen. Experiences in Using CC/PP in Context-Aware Systems. In *Proceedings of the International Conference on Mobile Data Management (MDM)*, volume 2574 of *Lecture Notes in Computer Science (LNCS)*, pages 247–261, Melbourne, Australia, January 2003.

[J2E] Java 2 Platform Enterprise Edition (J2EE). `http://java.sun.com/j2ee`.

[JBo] JBoss Aspect Oriented Programming. `http://www.jboss.org/developers/projects/jboss/aop`.

[KH02] W. Kießling and B. Hafenrichter. Optimizing Preference Queries for Personalized Web Services. In *Proceedings of the IASTED International Conference on Communications, Internet and Information Technology (CIIT)*, pages 461–466, St. Thomas, VI, USA, November 2002.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science (LNCS)*, pages 327–353, Budapest, Hungary, June 2001.

[Kie02] W. Kießling. Foundations of Preferences in Database Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 311–322, Hong Kong, China, August 2002.

[KK04a] M. Keidl and A. Kemper. A Framework for Context-Aware Adaptable Web Services (Demonstration). In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, volume 2992 of *Lecture Notes in Computer Science (LNCS)*, pages 826–829, Heraklion, Crete, Greece, March 2004.

[KK04b] M. Keidl and A. Kemper. Towards Context-Aware Adaptable Web Services. In *Proceedings of the International World Wide Web Conference (WWW)*, Manhattan, NY, USA, May 2004. Accepted for publication.

[KKKK01] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. Verteilte Metadatenverwaltung für die Anfragebearbeitung auf Internet-Datenquellen. In *Proceedings of the GI Conference on Database Systems for Business, Technology, and Web (BTW)*, Informatik Aktuell, pages 107–126, Oldenburg, Germany, March 2001.

[KKKK02a] M. Keidl, A. Kemper, D. Kossmann, and A. Kreutz. Verteilte Metadatenverwaltung und Anfragebearbeitung für Internet-Datenquellen. *Informatik - Forschung und Entwicklung*, 17(3):123–134, 2002.

[KKKK02b] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A Publish & Subscribe Architecture for Distributed Metadata Management. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 309–320, San Jose, CA, USA, February 2002.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science (LNCS)*, pages 220–242, Jyväskylä, Finland, June 1997.

[KRW+04] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. H. Butler, and L. Tran. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0. `http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115`, January 2004. World Wide Web Consortium (W3C), W3C Recommendation.

[KS00] D. Kossmann and K. Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Transactions on Database Systems (TODS)*, 25(1):43–82, 2000.

[KSK02] M. Keidl, S. Seltzsam, and A. Kemper. Flexible and Reliable Web Service Execution. In *Proceedings of the Workshop on Entwicklung von Anwendungen auf der Basis der XML Web-Service Technologie*, pages 17–30, Darmstadt, Germany, July 2002.

[KSK03a] M. Keidl, S. Seltzsam, and A. Kemper. Reliable Web Service Execution and Deployment in Dynamic Environments. In *Proceedings of the International Workshop on Technologies for E-Services (TES)*, volume 2819 of *Lecture Notes in Computer Science (LNCS)*, pages 104–118, Berlin, Germany, September 2003.

[KSK03b] M. Keidl, S. Seltzsam, and A. Kemper. ServiceGlobe: Flexible and Reliable Web Services on the Internet (Poster Presentation). In *Proceedings of the International World Wide Web Conference (WWW)*, Budapest, Hungary, May 2003.

[KSKK03] M. Keidl, S. Seltzsam, C. König, and A. Kemper. Kontext-basierte Personalisierung von Web Services. In *Proceedings of the GI Conference on Database Systems for Business, Technology, and Web (BTW)*, volume 26 of *Lecture Notes in Informatics (LNI)*, pages 344–363, Leipzig, Germany, February 2003.

[KSSK02] M. Keidl, S. Seltzsam, K. Stocker, and A. Kemper. ServiceGlobe: Distributing E-Services across the Internet (Demonstration). In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1047–1050, Hong Kong, China, August 2002.

[KW01] A. Kemper and C. Wiesner. Hyperqueries: Dynamic Distributed Query Processing on the Internet. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 551–560, Rome, Italy, September 2001.

[KW04] A. Kemper and C. Wiesner. Building Scalable Electronic Market Places using HyperQuery-Based Distributed Query Processing. *World Wide Web*, 2004. Accepted for publication.

[Ley01] F. Leymann. Web Services Flow Language (WSFL 1.0). `http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, May 2001. IBM Software Group.

[LH03] C. Lee and S. Helal. Context Attributes: An Approach to Enable Context-awareness for Service Discovery. In *Proceedings of the Symposium on Applications and the Internet (SAINT)*, pages 22–30, Orlando, FL, USA, January 2003.

[LHM+86] B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh, and P. F. Wilms. A Snapshot Differential Refresh Algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 53–60, Washington, DC,USA, June 1986.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 11(4):610–628, 1999.

[LS99] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`, February 1999. World Wide Web Consortium (W3C), W3C Recommendation.

[LS00] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 153–164, Dallas, TX, USA, May 2000.

[Mar99] H. Marais. Compaq's web language. `http://www.research.compaq.com/SRC/WebL/WebL.pdf`, 1999. Compaq Systems Research Center (SRC).

[Men01] G. Menkhaus. Architecture for Client-Independent Web-Based Applications. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 32–40, Zurich, Switzerland, March 2001.

[Mit03] N. Mitra. SOAP Version 1.2 Part 0: Primer. `http://www.w3.org/TR/2003/REC-soap12-part0-20030624`, June 2003. World Wide Web Consortium (W3C), W3C Recommendation.

[MJ01] M. B. Møller and B. N. Jørgensen. Enhancing Jini's Lookup Service Using XML-Based Service Templates. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 19–31, Zurich, Switzerland, March 2001.

[MM03] S. K. Mostéfaoui and G. K. Mostéfaoui. Towards A Contextualisation of Service Discovery and Composition for Pervasive Environments. In *Proceedings of the AAMAS Workshop on Web-services and Agent-based Engineering (WSABE)*, Melbourne, Australia, July 2003.

[MRT98] G. A. Mihaila, L. Raschid, and A. Tomasic. Equal Time for Data on the Internet with WebSemantics. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, volume 1377 of *Lecture Notes in Computer Science (LNCS)*, pages 87–101, Valencia, Spain, March 1998.

[MS03] E. M. Maximilien and M. P. Singh. Agent-based Architecture for Autonomic Web Service Selection. In *Proceedings of the AAMAS Workshop on Web-services and Agent-based Engineering (WSABE)*, Melbourne, Australia, July 2003.

[NET] Microsoft .NET. `http://www.microsoft.com/net`.

[ÖV99] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, NJ, USA, 1999.

[PFLS00] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-Based Publish/Subscribe Systems. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, volume 1901 of *Lecture Notes in Computer Science (LNCS)*, pages 162–173, Eilat, Israel, September 2000.

[PKC99] PKCS #5 v2.0: Password-Based Cryptography Standard. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf`, March 1999. RSA Laboratories.

[PKI] Public-Key Infrastructure (X.509) (PKIX). `http://www.ietf.org/html.charters/pkix-charter.html`. The Internet Engineering Task Force (IETF).

[PLdH03] A. J. H. Peddemors, M. M. Lankhorst, and J. de Heer. Presence, Location, and Instant Messaging in a Context-Aware Application Framework. In *Proceedings of the International Conference on Mobile Data Management (MDM)*, volume 2574 of *Lecture Notes in Computer Science (LNCS)*, pages 325–330, Melbourne, Australia, January 2003.

[RB03] S. Riché and G. Brebner. Storing and Accessing User Context. In *Proceedings of the International Conference on Mobile Data Management (MDM)*, volume 2574 of *Lecture Notes in Computer Science (LNCS)*, pages 1–12, Melbourne, Australia, January 2003.

[RMR00] M. Rodriguez-Martinez and N. Roussopoulos. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 213–224, Dallas, TX, USA, May 2000.

[RV02] E. Rahm and G. Vossen, editors. *Web & Datenbanken: Konzepte, Architekturen, Anwendungen*. dpunkt-Verlag, Heidelberg, Germany, 2002.

[SAT+99] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. Van Laerhoven, and W. Van de Velde. Advanced Interaction in Context. In *Proceedings of the International Symposium on Handheld and Ubiquitous Computing (HUC)*, volume 1707 of *Lecture Notes in Computer Science (LNCS)*, pages 89–101, Karlsruhe, Germany, September 1999.

[SBK01] S. Seltzsam, S. Börzsönyi, and A. Kemper. Security for Distributed E-Service Composition. In *Proceedings of the International Workshop on Technologies for E-Services (TES)*, volume 2193 of *Lecture Notes in Computer Science (LNCS)*, pages 147–162, Rome, Italy, September 2001.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[SG02] J. P. Sousa and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 29–43, Montréal, Québec, Canada, August 2002.

[SMS02] N. Santos, P. Marques, and L. Silva. A Framework for Smart Proxies and Interceptors in RMI. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems (ISCA PDCS)*, Louisville, KY, USA, September 2002.

[Sto99] M. Stonebraker. Integrating Islands of Information. *EAI Journal*, pages 1–5, September 1999. `http://www.bijonline.com/Article.asp?ArticleID=130`.

[Suna] Sun Microsystems Inc. The Java Servlet Specification 2.4. `http://java.sun.com`.

[Sunb] Sun Open Net Environment (Sun ONE). `http://www.sun.com/sunone`.

[TGNO92] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 321–330, San Diego, CA, USA, June 1992.

[Tha01] S. Thatte. XLANG: Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`, 2001. Microsoft Corp.

[UDD00] Universal Description, Discovery and Integration (UDDI) Technical White Paper. `http://www.uddi.org`, 2000. Ariba Inc., IBM Corp., and Microsoft Corp.

[UPN] Universal Plug and Play Device Architecture. `http://www.upnp.org`. Microsoft Corp.

[Wal01] J. Waldo. *The Jini Specifications*. Addison-Wesley, Reading, MA, USA, 2001.

[WDH+82] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An Overview of the Architecture, June 1982.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). `http://www.rfc-editor.org/rfc/rfc2251.txt`, December 1997. RFC 1997.

[WSI03] Web Services Invocation Framework (WSIF). `http://ws.apache.org/wsif`, 2003. The Apache Software Foundation.

[YGM99] T. W. Yan and H. Garcia-Molina. The SIFT Information Dissemination System. *ACM Transactions on Database Systems (TODS)*, 24(4):529–565, 1999.