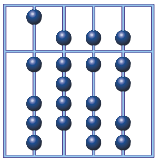


Lehrstuhl für Datenbanksysteme
Fakultät für Informatik
Technische Universität München

Security, Caching, and Self-Management in Distributed Information Systems

Diplom-Informatiker Univ.
Stefan Seltzsam

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. Helmut Krcmar

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph. D.
2. Univ.-Prof. Dr. Erhard Rahm, Universität Leipzig

Die Dissertation wurde am 17.06.2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.11.2004 angenommen.

Acknowledgments

First of all, I would like to thank my advisor, Prof. Alfons Kemper, for the opportunity to participate in ambitious and visionary projects. His advices, many helpful discussions, and comments provided invaluable guidance for my work.

Natalija Krivocapić was the advisor for my master thesis and introduced me to the topic of security in a distributed system of autonomous objects. She did a great job and I learned a lot from her insight and experience in doing research and project work.

My doctoral thesis was done in the context of the projects ObjectGlobe, ServiceGlobe, and AutonomicGlobe. Since so many people contributed to these projects, it is impossible to list all of them here. In particular I like to thank Reinhard Braumandl, Markus Keidl, Bernhard Stegmaier, and Christian Wiesner, who contributed various important parts to these projects. A big "Thank you!" to all other project members!

I wish to thank Stephan Börzsönyi, Tobias Brandl, Roland Holzhauser, and Christof König, whose master thesis I adviced, for their excellent work. A special thank to Tobias Brandl and Stefan Krompaß for the implementation of the autonomic computing concept and the simulation system. Another special thank to Roland Holzhauser and Christof König for the implementation of the SSPLC prototype system. They all worked with me for a long time and did an excellent job. Thank you all for being such a great team!

I wish to express my gratitude to all my colleagues at the University of Passau and afterwards at the Technical University Munich for many helpful discussions and the pleasant working environment: Reinhard Braumandl, Markus Keidl, Bernhard Stegmaier, Christian Wiesner, Bernhard Zeller, and my newest colleagues Daniel Gmach, Richard Kuntschke, and Martin Wimmer. Alexandra Schmidt and Evi Kollmann provided support in all kinds of administrative and non-technical tasks. Markus Keidl and I shared an office for several years. We always had a great and inspiring working atmosphere.

For helpful criticism, proof-reading and/or advice on my doctoral thesis, I express my thanks to Laura Alvarey, Markus Keidl, Roland Holzhauser, Martin Wimmer, Bernhild Ellmann, Reinhard Braumandl, Natalija Krivocapić, Thomas Sturm, Andreas Seidl, and Richard Kuntschke. I appreciate all their valuable suggestions. A very special thank to Laura Alvarey who did a very thorough and fast job in proof-reading. As native speaker she even found grammatical subtleties and helped me fixing them.

I thank Wolfgang Becker, Ingo Bohn, and Thorsten Dräger of SAP's Adaptive Computing Infrastructure group for their cooperation.

Last, but not least, many thanks to my parents, my brother, Susanne Koerber, and my friends for their support and encouragement throughout the years and for "always being there".

Garching, January 2005,

Stefan Seltzsam

Abstract

In this thesis, we investigate three different aspects of distributed information systems: security, caching, and self-management.

We describe our concept of a security system for distributed and open systems using our query processing system ObjectGlobe as an example. One part of this concept is our OperatorCheck server, which validates the semantics of an operator and analyzes its quality before the operator is actually used in queries. This is done semi-automatically using an oracle-based approach to compare a formal specification of an operator against its implementation. Further security measures are integrated into the query processing engine: secure communication channels are established, authentication and authorization are performed, and overload situations are avoided by admission control. Operators are guarded using Java's security model to prevent unauthorized resource access and leakage of data. The resource consumption of operators is monitored and limited to avoid resource monopolization.

We present a semantic caching scheme suitable for caching responses from Web services on the SOAP protocol level. Web services are typically described using WSDL documents. For semantic caching we developed an XML-based declarative language to annotate WSDL documents with information about the caching-relevant semantics of requests and responses. Using this information, our semantic cache answers requests based on the responses of similar previously executed requests. Performance experiments—based on the scenarios of TPC-W and TPC-W Version 2—conducted using our prototype implementation demonstrate the effectiveness of the proposed semantic caching scheme.

We present a novel autonomic computing concept which is hiding the ever increasing complexity of managing IT infrastructures. For this purpose, we virtualize, pool, and monitor hardware to provide a dynamic computing infrastructure. A fuzzy-logic-based controller supervises all services running on this virtual platform. Higher-level services such as business applications profit from running on this platform. For example, failed services are restarted automatically. A service overload is detected and remedied by either starting additional service instances or by moving the service to a more powerful server. The capabilities and constraints of the services and the hardware environment are specified in a declarative XML language.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the last decade, we have seen a substantial growth of the Internet with respect to several dimensions: the number of computers connected to the Internet, the number of users, and the number of content providers all have increased dramatically. Initially, only a very limited and static content was available on the Internet. Nowadays, however, a vast amount of static and dynamic information is accessible. With the increasing flood of information, the desire to be able to efficiently query this data and correlate data from different sources has grown. Thus, more and more complex data integration systems have been developed with the goal of realizing the vision of the Internet as a global database management system [LKK+97].

Data integration systems evolved from centralized middleware systems [Wie93] to globally operating data integration systems like ObjectGlobe [BKK+01a] which can potentially cover all the appropriate data sources on the Internet. ObjectGlobe is not a monolithic architecture. Instead, it is a distributed and open query processor for Internet data sources in which operators can be integrated in the form of user-defined mobile code in a seamless and effortless manner to, for example, add user-defined data transformations or predicates. Thus, ObjectGlobe satisfies the emerging need for distribution and quick adaptation to new requirements stemming from, for example, virtual enterprises. However, usage of mobile code introduces specific security concerns. ObjectGlobe and all other distributed and open architectures need sophisticated security systems to check the semantics and quality of mobile code before the code is actually executed. Additionally, they must provide secure runtime environments for mobile code to protect the executing systems from unauthorized resource access and overload situations. An additional issue is the prevention of data leakage.

Currently, a second wave of integration is rolling through the Internet. This time the focus of the integration efforts is on applications rather than on data. Service-oriented architectures based on Web services are already emerging as the predominant application type on the Internet. This development is primarily driven by the desire of companies for a global application integration platform. Companies bargain for cost-cuttings by automated flexible workflows for business-to-business (B2B) e-commerce, like fully-automated supply

chain management, as well as for business-to-consumer (B2C) e-commerce. Companies and technology providers are both interested in interoperability. Therefore, they are working together on several standards for Web services [RV02], for example, XML [BPSM+04], SOAP [BEK+00], UDDI [UDD00], and WSDL [CCMW01] (to name the most important ones). Currently, there are several Web service platforms available from different vendors, for example, IBM WebSphere [IBMb], Microsoft .NET [NET], Sun ONE [Sunb], and ServiceGlobe [KSK03a], our own research platform. All these platforms implement the standards mentioned above and can therefore seamlessly interact with one another. These Web service platforms can be used for inter-company as well as intra-company application integration.

Some of the most urgent problems of globally accessible Web services are performance and scalability. These problems are common in distributed systems on the Internet and, thus, there are solutions for different application areas. For example, distributed database management systems and traditional Web servers rely heavily on different caching techniques [RV02, INST02] to reduce the load of their servers and to speed up processing, for example, proxy caches, content distribution networks (CDNs), or edge server caches. Currently, there are no sophisticated caching schemes available in the area of Web services. It is obvious that such caching schemes will be essential in the future as the number of users of Web services grows steadily.

Another purpose of Web services is, as already mentioned, the intra-company application integration. Here again, the predominant aim of companies is cutting costs. Using Web service technology, the linking of applications becomes much easier, less complex and, therefore, cheaper. Nevertheless, complexity and consequently administration costs of IT infrastructures are ever increasing. IBM coined the term autonomic computing [Hor01] for solutions which overcome this trend. This term refers to some kind of self-management of hardware and software. Comprehensive self-management capabilities for systems include self-configuration, self-optimization, self-healing, and self-protection. Several global players conduct research in this area, and they have already integrated some aspects of self-management into their hardware and software products. While several technologies and products are already available, most of them are only able to handle problems of isolated components of IT infrastructures, for example, a failed processor of a multi-processor system. Additionally, they depend heavily on vendor-specific hardware features. A solution for a self-managing, vendor-independent IT infrastructure that supervises and controls itself is still missing.

1.1 Purpose of this Thesis

In this thesis, we investigate the three different aspects of distributed information systems addressed above. Our focus is on security, caching, and self-management.

As mentioned before, usage of mobile user-defined code introduces specific security concerns. We present a comprehensive security architecture for distributed and open systems

using ObjectGlobe as an example. In ObjectGlobe, users can provide operators in the form of mobile code. Before such an operator is actually used in queries, an OperatorCheck server validates its semantics and analyzes its quality. This is done semi-automatically using an oracle-based approach to compare a formal specification of an operator against its implementation. Further security measures are integrated into the query processing engine: during plan distribution, secure communication channels are established, authentication and authorization are performed, and overload situations are avoided by admission control. During plan execution, operators are guarded using Java's security model to prevent unauthorized resource access and leakage of data. The resource consumption of operators is monitored and limited to avoid resource monopolization. We show that the presented security system is capable of executing arbitrary operators without risk to the executing host or the privacy and integrity of the processed data.

Caching is an approved solution for performance and scalability issues of distributed systems. We present a semantic caching scheme suitable for caching responses from Web services on the SOAP protocol level. Existing semantic caching schemes for database systems or Web sources cannot be applied directly because there is no semantic knowledge available about the requests to and responses from Web services. Web services are typically described using WSDL (Web Service Description Language) documents. To enable semantic caching we developed an XML-based declarative language to annotate WSDL documents with information about the caching-relevant semantics of requests and responses. Using this information, our semantic cache answers requests based on the responses of similar previously executed requests. Performance experiments—based on the scenarios of TPC-W and TPC-W Version 2—conducted using our prototype implementation demonstrate the effectiveness of the proposed semantic caching scheme.

The third challenge of distributed systems we investigate in this thesis is the self-management of complex IT systems. We present a novel autonomic computing concept which is hiding the ever increasing complexity of managing IT infrastructures. For this purpose, we virtualize, pool, and monitor hardware to provide a dynamic computing infrastructure. A fuzzy-logic-based controller supervises all services running on this virtual platform. According to the vision of autonomic computing, this infrastructure is a step towards a self-managing, self-optimizing, and self-healing virtual platform for services. Higher-level services such as business applications benefit from running on this supervised virtual platform. For example, failed services are restarted automatically and a service overload is detected and remedied by starting additional service instances or by moving the service to a more powerful server. Available resources are shared between all services as appropriate for a particular situation. Thus, by dynamically allocating the services, we improve the average utilization of the available hardware and minimize idle time. Thereby, total cost of ownership (TCO) is reduced either because more users can be handled using the existing hardware or because less hardware is required to begin with. The capabilities and constraints of the services and the hardware environment are specified using a declarative XML language. We used our prototype implementation, AutonomicGlobe, for first tests managing a blade server configuration and for comprehensive simulation studies which demonstrate the effectiveness of our proposed autonomic computing concept.

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 distributed and open query processing system for data processing services on the Internet.
- Chapter 3 presents a security system for distributed and open architectures using ObjectGlobe as an example system. Additionally, a technique concerning quality assurance for external operators is presented.
- Chapter 4 introduces the ServiceGlobe system—our distributed and open Web service platform. We also give a brief introduction to Web service standards that are important in our work.
- Chapter 5 investigates caching techniques for Web services. We present a semantic caching scheme suitable for caching responses from Web services on the SOAP protocol level. Additionally, results of comprehensive performance experiments are given.
- Chapter 6 describes an autonomic computing concept for application services using an enterprise resource planning (ERP) software installation as an example. This work is based on the ServiceGlobe system and adds a controller framework for autonomic decisions to ServiceGlobe. A fuzzy controller which can be plugged into this framework is presented. Comprehensive simulation studies using this fuzzy controller are described.
- Chapter 7 concludes the thesis.

Chapter 2

ObjectGlobe - A Distributed and Open Query Processing System

In this chapter, we present the design of ObjectGlobe, our distributed and open 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 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 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, Bra01, BKK+00, BKK+99]. The HyperQuery project presented in [KW01, KW04] is an extension of the ObjectGlobe system offering a platform for scalable electronic marketplaces on the Internet.

This chapter is organized as follows: In Section 2.1 we outline how queries are processed in ObjectGlobe. Section 2.2 gives a concrete example and introduces the external Skyline operator. The lookup service of ObjectGlobe is presented in Section 2.3. In Section 2.4 we give a survey of the quality of service management of ObjectGlobe.

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 in executing the query. In addition, the lookup service provides the authorization data—mirrored and integrated from the individual providers—to determine what resources may be

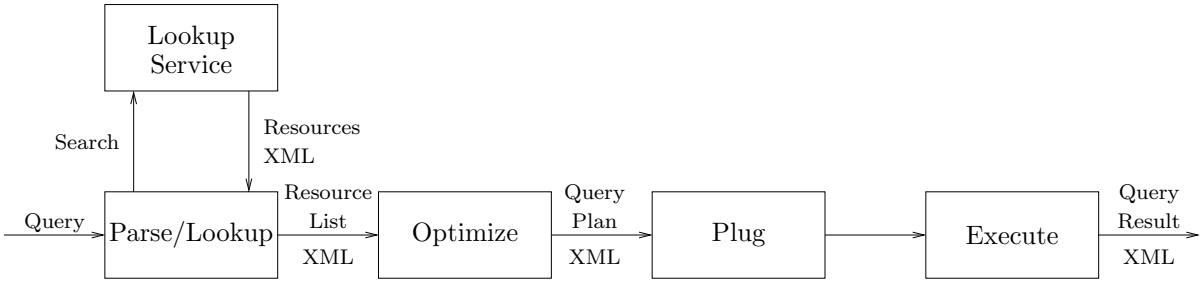


Figure 2.1: Processing a Query in ObjectGlobe

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*, i.e., user-defined, 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" (described in Chapter 3). 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),

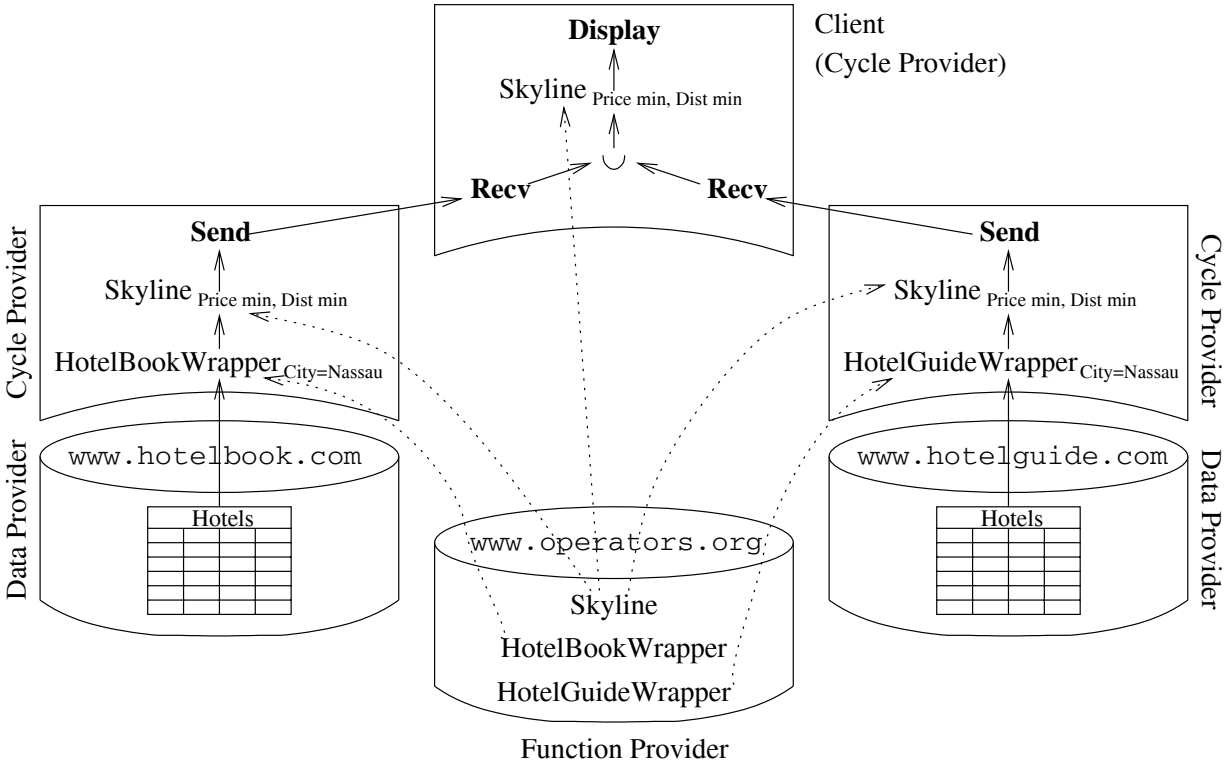


Figure 2.2: Distributed Query Processing with ObjectGlobe

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 a 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 which they might want to use in the query. Furthermore, a user can look at and change the plan generated by the optimizer.

2.2 Example Query

Figure 2.2 shows an example of distributed query processing with ObjectGlobe. Suppose one is going on holiday to Nassau, Bahamas, and is looking for a hotel that is cheap and close to the beach. This task is known as the "maximum vector problem" [PS85]. Actually, we are searching for the minimal vectors, but these can be found analogously. Formulated precisely, we are looking for all hotels which are not dominated by other ones. A hotel dominates another one if it is cheaper *and* closer to the beach. Dominance imposes a partial ordering on the hotels.

A naive solution for this problem is to compare each hotel to every other and delete dominated ones. This simple algorithm yields quadratic runtime. However, a more sophisticated algorithm with a lower complexity has been developed by [KLP75]. [BKS01] have investigated this algorithm in the context of databases and adapted it to constrained main memory. They called the operator "Skyline".

The resources used to find the desired cheap hotel near the beach are as follows: two Web sites, `www.hotelbook.com` and `www.hotelguide.com`, supply hotel data and all external operators are provided by the function provider `www.operators.org`. Two wrappers, HotelBookWrapper and HotelGuideWrapper, are responsible for querying the two Web sites and transforming the data into ObjectGlobe's internal format. They are executed at cycle providers located near the data sources in order to minimize transfer time. As the following equation holds for the Skyline operator, it can be applied to each data source directly in order to further reduce data shipping costs:

$$Skyline(Skyline(A) \cup Skyline(B)) = Skyline(A \cup B)$$

Thus, only the best hotels are passed to the client. The send/receive iterator pairs performing the transmission of data are installed automatically during the plug phase. The client calculates the Skyline of the union of both data sources, and the user can choose a hotel from the result.

2.3 Lookup Service

The lookup service plays the same role in ObjectGlobe 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 detailed description of the lookup service of ObjectGlobe is given in [KKKK02].

During the optimization of every query in an ObjectGlobe federation, the lookup service is queried for descriptions of useful services for the respective query. Therefore, the main challenge of the lookup service is to provide global access to the metadata of all registered services without becoming the bottleneck of the whole system. Since the metadata structures in an open and extensible system are naturally quite complex, the lookup service offers a sophisticated special-purpose query language, which also allows for the expression of joins over metadata collections. In addition to the network and storage devices, the computing power of a lookup service machine can limit the throughput of metadata queries. Thus, our lookup service uses a three-tier architecture as depicted in Figure 2.3. The purpose of this architecture is to scale in the number of users of the lookup service (both real users who browse the metadata and optimizers which search for specific services) by adding new local metadata repositories at the hot spots of user activity.

The information at metadata providers is regarded as globally and publicly available and therefore it is consistently replicated by all metadata providers which appear in the metadata provider backbone. For the efficiency reasons stated above, metadata providers

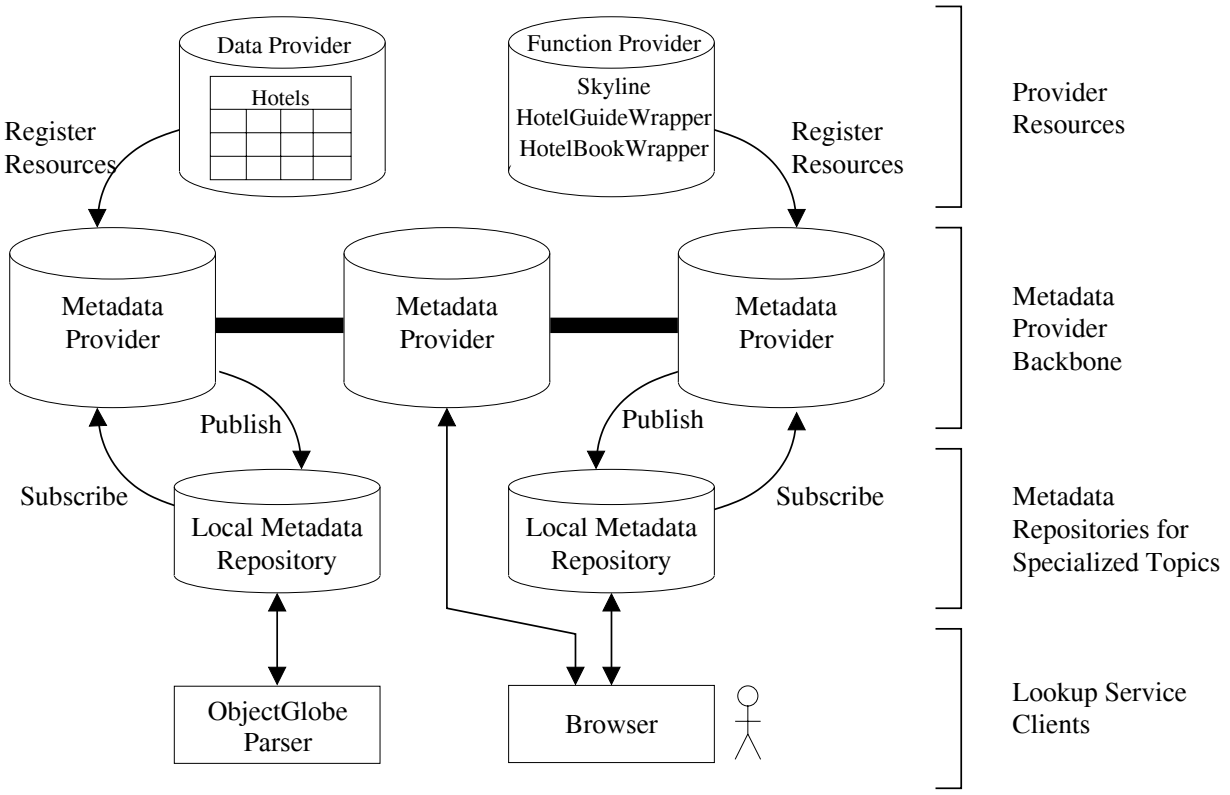


Figure 2.3: The Architecture of the Lookup Service

themselves cannot be queried; they only can be browsed in order to detect metadata which should also be available at a specific local metadata repository. Only these local metadata repositories can be queried for locally cached metadata. They use a *publish/subscribe* mechanism to fetch relevant data from a metadata provider. In the case of updates, insertions, or deletions of metadata, a metadata provider evaluates the possibly huge set of subscription rules with the help of a sophisticated prefilter algorithm and forwards the appropriate changes to the corresponding local metadata repositories.

2.4 Quality of Service (QoS)

Query execution in ObjectGlobe can involve a large number of different function, cycle and data providers. Therefore, a plan produced by a traditional optimizer might 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:** There are several important properties of a query result a user should be able to specify. For example, a user may want to restrict the size of the result set

returned by a query in the form of a lower or an upper bound (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 "hotels" 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 which are based on a current and sufficiently large subset of the available data.

- **Cost:** Since providers can charge for their service, 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 to other existing system architectures are given in [BKK03, Bra01].

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. In case the resource requirements cannot be satisfied with the available resources during the plug phase, 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.

Chapter 3

Security and Privacy Issues in Distributed and Open Systems

Security is crucial to the success of distributed and open systems like ObjectGlobe because usage of mobile code introduces specific security concerns. In this chapter, we describe our concept of the security system of ObjectGlobe. The security measures are classified by the time of application. Before an operator is actually used in queries, our OperatorCheck server validates its semantics and analyzes its quality. This is done semi-automatically using an oracle-based[1] approach to compare a formal specification of an operator against its implementation. Further security measures are integrated into the query processing engine: during plan distribution, secure communication channels are established, authentication and authorization are performed, and overload situations are avoided by admission control. During plan execution, operators are guarded using Java's security model to prevent unauthorized resource access and leakage of data. The resource consumption of operators is monitored and limited to avoid resource monopolization. We show that the presented security system is capable of executing arbitrary operators without risks to the executing host or the privacy and integrity of the processed data. Parts of this chapter have already been presented in [SBK01].

This chapter is organized as follows: Section 3.1 motivates the importance of security systems for distributed and open systems and Section 3.2 elaborates on the security requirements of such systems. Our security system is based on the security model of Java, which is outlined in Section 3.3. Sections 3.4 and 3.5 give a survey of the security measures during plan distribution and outline runtime guarding and monitoring measures used to detect malicious or defective operators. Section 3.6 discusses some issues on the correctness of the runtime security system. After that, Section 3.7 describes in detail preventive measures appropriate to detect the majority of low quality and malicious external operators before actually executing them in queries. Section 3.8 discusses security concerns in different scenarios and demonstrates the usage of our security system adapted to the specific needs of the scenarios. Related work is addressed in Section 3.9, and Section 3.10 concludes this chapter.

[1] We think of an oracle in the true sense of the word, not of the commercial DBMS.

3.1 Motivation

The recent trend towards distributed and open systems demands new sophisticated security systems to meet the challenges of mobile user-defined code. Examples for such systems are Web browsers executing Applets, Web application servers executing Servlets or Java Server Pages, and extensible database management systems implementing, e.g., the SQL99 standard [SQL99] for user-defined functions. Nowadays more and more wireless devices execute code in the form of WML scripts [GS01], e.g., to interact with a mobile e-commerce system. In this chapter, we use ObjectGlobe as an example for such a system.

The openness of a system like ObjectGlobe creates new demands on a security system. The source, the programmer, and the code of external user-defined operators are normally unknown. Thus, users of such operators are unsure whether the operator is calculating the correct result or may crash or manipulate data given to it. For this reason quality assurance is necessary because users of external operators want to feel confident about the semantics and functioning of operators in order to rely upon the results of a query. As in every distributed system, it is necessary to protect communication channels against tampering and eavesdropping. Cycle providers execute arbitrary external operators. Thus, they need a security architecture which prevents external operators from accessing resources like the file system of the cycle provider, monopolizing resources like memory or CPU time, or manipulating vital components of the ObjectGlobe system. Additionally, cycle providers need an authentication framework to be able to determine the identity of a user.

The goal of this work is to provide a security architecture for ObjectGlobe which addresses all mentioned challenges and in general is appropriate for distributed and open systems. In order to achieve these goals we have to rely upon some basic assumptions about the environment of ObjectGlobe. First, we assume that the operating system which is running ObjectGlobe is secure and that the administrators of the cycle providers are trustworthy, because we cannot protect the system against the operating system. Second, we assume that the code and the Java virtual machine (JVM) used to run ObjectGlobe are unmodified. These requirements are enforced in ObjectGlobe by giving the user the possibility to restrict the cycle providers to a set of trusted cycle providers. The last and most serious assumption is that the security system of Java 2 [Oak98] works as designed. There have been some security-related bugs and implementation flaws of Java, but it seems that there are no elementary flaws in the design of the security model.

3.2 Security Requirements

There are several security requirements of users and cycle providers. First, we will concentrate on security issues introduced by external operators: cycle providers want to be able to execute arbitrary external operators in a safe way, i.e., they want to be sure that operators do not monopolize resources, access resources like the file system unauthorized, or manipulate vital components of the system.

We now demonstrate some attacks and the implications that arise without an effective security system using modified versions of the Skyline operator (see Section 2.2). The example below shows a code snippet which monopolizes the memory by continuously generating and storing new `Object` instances in a `LinkedList`. The execution of this operator results in a denial of service because there will be insufficient memory for other operators. Of course, an external operator could just as well monopolize CPU time, secondary memory, or any other limited shared resource. It would also be possible for an operator to, e.g., access the file system to modify arbitrary files or to steal confidential data.

Example *Resource Monopolization*

```
public class Skyline extends IteratorClass {
  public TypeSpec open() throws CommandFailedException,IOException {
    List l = new LinkedList();
    while(true)
      l.add(new Object());
    ...
  }
  ...
}
```

Along with security requirements of providers there are requirements of users. They want to feel confident about the semantics of external operators in order to rely upon the results of a query even when they use external operators. Recall the query using a Skyline operator to find the hotels which are cheap and near the beach (see Section 2.2). Now assume a modified Skyline operator that filters all hotels of the Sheraton hotel chain. The result of a query using this modified Skyline would be all cheap hotels near the beach not being a Sheraton hotel, which is not the result the user requested. Privacy is another severe requirement endangered by external operators: an operator could calculate the result as required, but it could also send a copy of the processed data to an arbitrary host on the Internet or leak data to a concurrent query, possibly compromising confidentiality of data. For that reason, it is necessary that the security system can guarantee that data given to an operator can only flow using communication channels which are obvious to and authorized by the user.

In addition to the security requirements induced by external query operators there are some which are common to many distributed systems. First, using our terminology, cycle and data providers may have a legitimate interest in obtaining the identity of users for authorization purposes. It must be considered that users normally want to stay anonymous as far as possible, therefore it must not be mandatory to give authentication data to cycle providers. Second, the communication channels between different collaborating hosts must be protected against tampering to avoid unnoticed modifications of the data. Additionally, it must be possible to encrypt confidential data to prevent other parties from eavesdropping. Third, cycle providers need an admission control system to guard themselves against overload situations.

To meet these requirements, we use a multilevel security architecture combining preventive measures, security measures during plan distribution, and a runtime security system. Preventive measures take place before an operator is actually used in queries. They are used to validate the semantics and analyze the quality of the operator. Based upon the validation results, ObjectGlobe could renounce runtime security measures. Because preventive measures are optional, untested operators are regarded as possibly malicious and all security measures apply. During plan distribution, common security measures of distributed systems take place which include admission control. The remaining security requirements, e.g., protection of cycle providers, are met by the runtime architecture. We give a brief overview of Java's security model and the mandatory security levels of our security architecture, i.e., security measures during plan distribution and the runtime security architecture, in the next three sections. Thereafter, we present the preventive measures in detail and point out the advantages of validated operators.

3.3 Java's Security Model

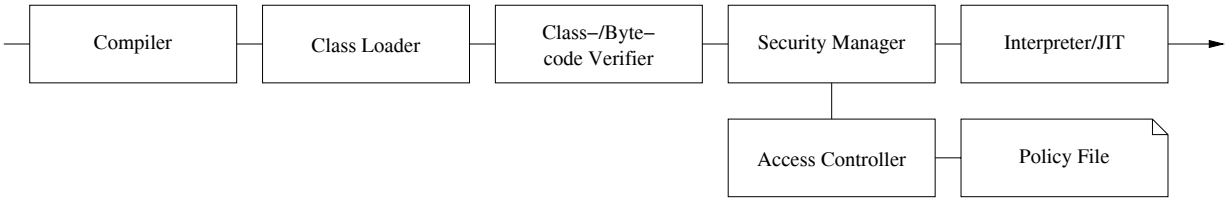


Figure 3.1: Java's Five-Layer Security Model

Figure 3.1 outlines the five layers of Java's security model [Oak98]. Java is a strongly typed object-oriented programming language with information hiding. The adherence to typing and information hiding rules is verified by the compiler and, because code could be generated by an evil compiler, again by the class/bytecode-verifier before a class object is generated from the bytecode. The class loader's tasks are to load the bytecode of a class into memory, monitor the loaded code's origin (i.e., its URL), and to verify the signature of digitally signed code. The security manager controls the access to safety critical system resources, such as the file system, network sockets, peripherals, etc. The security manager is used to create a so-called *sandbox* in which untrusted code is executed. Most well-known are the restrictive sandboxes in which Web browsers execute mobile code (i.e., Applets) loaded from Web servers. The ObjectGlobe system is based on Java Release 2, in which the security manager interfaces with the access controller. The access controller verifies whether an access to a safety-critical resource is legitimate according to a configurable policy. Privileges can be granted based on the origin of the code and whether or not it is digitally signed (i.e., authenticated) code. Additionally, the access controller offers the facility to temporarily give classes the ability to perform an action on behalf of a class

that normally might not have that ability by marking code as *privileged*. This feature is essential, for example, for granting access to temporary files via a secure interface (called *TmpFile* in ObjectGlobe). Finally, the Java program is executed by the interpreter which is responsible for runtime enforcement of security by, e.g., checking array bounds and object casts. From a security perspective, it is irrelevant whether or not parts of the code are compiled by a just-in-time (JIT) compiler to increase performance.

3.4 Security Measures during Plan Distribution

The common security requirements of distributed systems, as enumerated above, are met during plan distribution where four security-related actions take place: setup of secure communication channels, authentication, authorization, and admission control.

Privacy and integrity of data and function code that is transmitted between ObjectGlobe servers is protected against unauthorized access and manipulation by using the well-established secure communication standards SSL (Secure Sockets Layer) [FKK96] and/or TLS (Transport Layer Security) [DA99] for encrypting and authenticating (digitally signed) messages. Both protocols can carry out the authentication of ObjectGlobe communication partners via X.509 certificates [HFPS99], thus ensuring communication with the desired ObjectGlobe server. The security level of network connections can be chosen depending on the processed data.

If authentication is required for authorization or accounting purposes of providers, ObjectGlobe can authenticate users using one of the two possibilities described below. 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 (e.g., a time stamp to avoid reusage of signed plans).

- 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.

Of course, usage of X.509 certificates is preferred, but until certificates are commonly used, password-based authentication is supported as an alternative. The signature of a query arriving at a provider is verified using the user's X.509 certificate or the user's password. After that, the originator and the integrity of the query is known reliably. Providers can use this knowledge to enforce their local authorization policy autonomously. Of course, users and applications accessing only free and publicly available resources can stay anonymous and no authentication is required. If a user wants to access a resource that charges and accepts electronic payment, the user can remain anonymous as well (if the electronic payment system supports it) and the electronic payment is shipped as part of the plug phase.

The last security-related action during plan distribution, which is needed by all systems offering services to users, is admission control. ObjectGlobe's admission control component determines whether or not the estimated resource requirements of a query—calculated using the cost models of the operators (see Section 3.5)—can be satisfied by the executing host. This proceeding is advantageous insofar as queries can be aborted as early as possible if any cycle provider executing a part of the query cannot satisfy the resource requirements of the query. Queries whose resource requirements can be fulfilled are scheduled using a FCFS (First Come First Served) scheduler considering only main memory usage of the operators. We assume that a sufficient amount of all other resources such as secondary memory is available. This scheduling approach works well, because every query is restricted to using only a small fraction of the main memory of the server. Thus, a certain degree of parallel execution is guaranteed.

3.5 Architecture of the Runtime Security System

After plan distribution, all involved cycle providers execute the operators assigned to them to calculate the result of the query. Therefore, they must be protected from damage by malicious or low quality operators as outlined above. To satisfy the security interests of users, the security system must also be able to guarantee that data given to an operator can only flow using communication channels which are obvious to and authorized by the user. These security requirements are met using two techniques: guarding and monitoring.

The guarding mechanisms are realized using Java's security architecture, i.e., security manager and class loaders, to control and restrict access to resources of the cycle provider and components of the ObjectGlobe system.

The class loader's tasks are to load the bytecode of a class into memory, monitor the loaded code's origin (i.e., its URL), and verify the signatures of digitally signed code. Additionally, every class loader generates its own name space. Normally, class loaders are able to load further classes from the code's origin at runtime. But to prevent external operators from abusing a connection to a function provider as a covert communication channel by requesting classes with data coded into the names of the classes, all (non built-in) classes required by an external operator must be combined into a JAR[2] file. This file is loaded and cached by the class loader during the plug phase. Thus, there is no connection to the function provider during plan execution. Furthermore, queries running concurrently are separated from each other to prevent them from exchanging information with each other via, e.g., static class variables. This is done by using a new class loader instance (called *OGClassLoader*) for each query which implicitly separates the name spaces. In this way, external operators are isolated and leakage of data is prevented, because they are only able to communicate with their children and parent operators.

The security manager is used to create a sandbox in which untrusted code is executed. It controls the access to safety-critical system resources such as the file system, network

[2]JAR (Java ARchive) is a platform-independent file format that aggregates many files (compressed) into one (like ZIP) and is supported by the Java runtime environment.

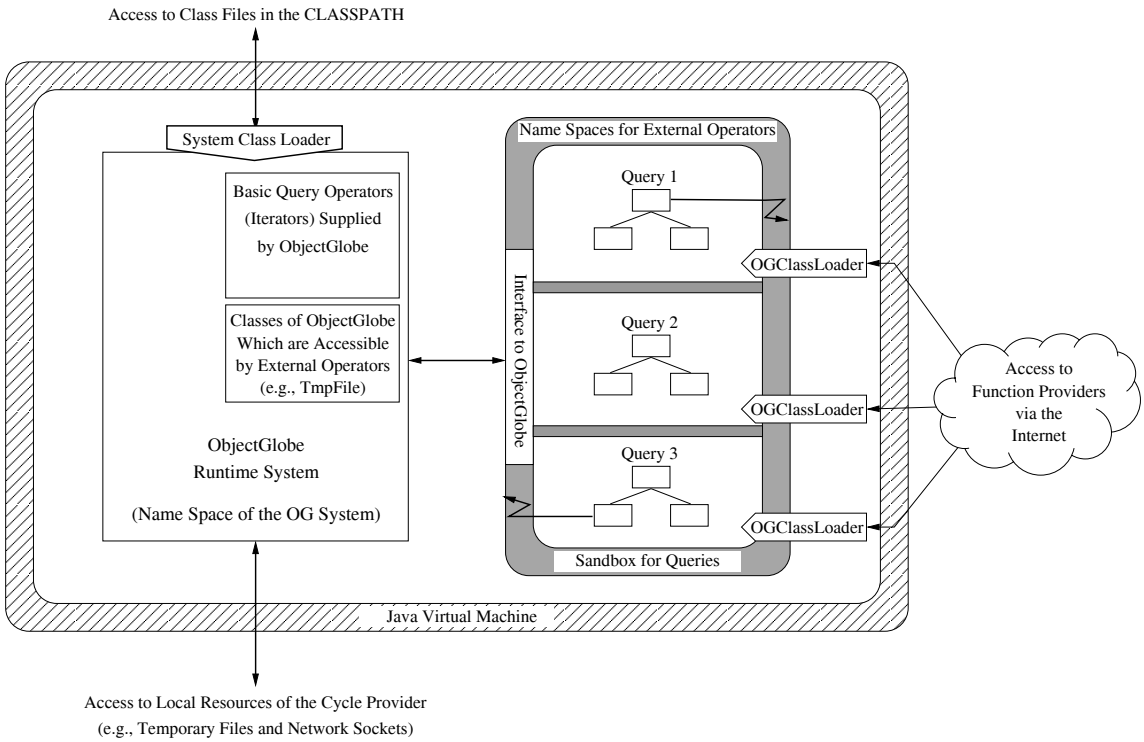


Figure 3.2: Protection of the Resources of Cycle Providers

sockets, peripherals, etc. Privileges can be granted to code based on its origin and whether or not it is digitally signed. Of course, it would be unreasonable to grant unprotected access to system resources to unknown code. Therefore, all user-defined operators are normally executed in a "tight" sandbox. The sandbox, the name space separation, and the class loaders are illustrated schematically in Figure 3.2.

Additionally, only selected classes of the name space of the ObjectGlobe system are accessible to external operators. Access to other classes of the ObjectGlobe system is prevented by the class loaders of external operators. Thus, vital components of the system are protected. One of the classes available to operators is *TmpFile*. This class implements a secure interface to the file system to enable operators to use temporary files. Figure 3.3 sketches the usage of temporary files by external operators.

Monitoring measures are necessary to avoid resource monopolization. We use our own (platform-dependent) resource accounting library which supervises CPU and main memory usage of external operators[3], because Java does not offer such functionality. Accounting of secondary memory and data volume produced by an operator is done using pure Java. External operators must be endowed with (worst-case) cost models written in MathML [CIMP03] for their CPU usage, consumption of main memory and secondary memory, number of temporary files simultaneously in use, and the number and size of tuples they produce. The last two cost models are necessary to prevent operators from

[3]Using our library, accounting results in overheads between 5% and 10%.

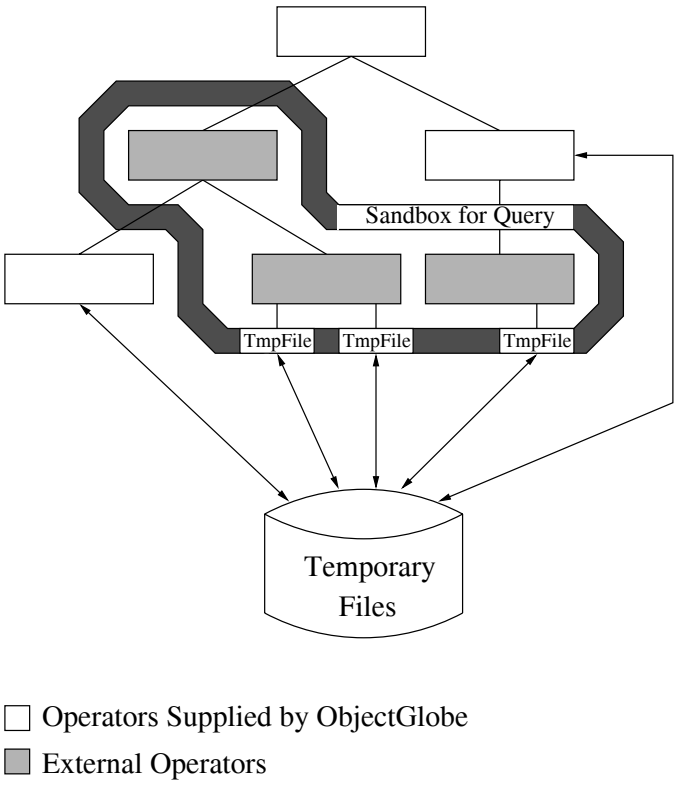


Figure 3.3: Extending Privileged Access Rights to User-Defined Operators

blocking the network and from flooding their parent operators. If an external operator is not equipped with its own cost models, ObjectGlobe uses default cost models, which of course might not be very appropriate.

Figure 3.4 illustrates the monitoring for an external operator with two children. In order to keep track of resource consumption, every external operator is executed by a separate thread and is disconnected from other operators using buffers, each managed by a send/receive iterator pair. The resource consumption of operators is traced by several collaborating components: an *RMAccount* object is used to store the current resource consumption and limits thereof, cost models, and information about number and size of tuples delivered to the operator. The *ResourceMonitor* is used to interact with our resource monitoring library to periodically determine the CPU and main memory usage of the thread running the external operator. The send operator above the external operator updates the number and size of the tuples produced by the operator, while the receive operators beneath the operator measure the number and maximal size of tuples and the total data volume delivered to the operator. TmpFiles (not shown in the illustration) register themselves at the corresponding RMAccount and permanently report their current sizes. Thus, an RMAccount is able to track the number and overall size of the operator's temporary files.

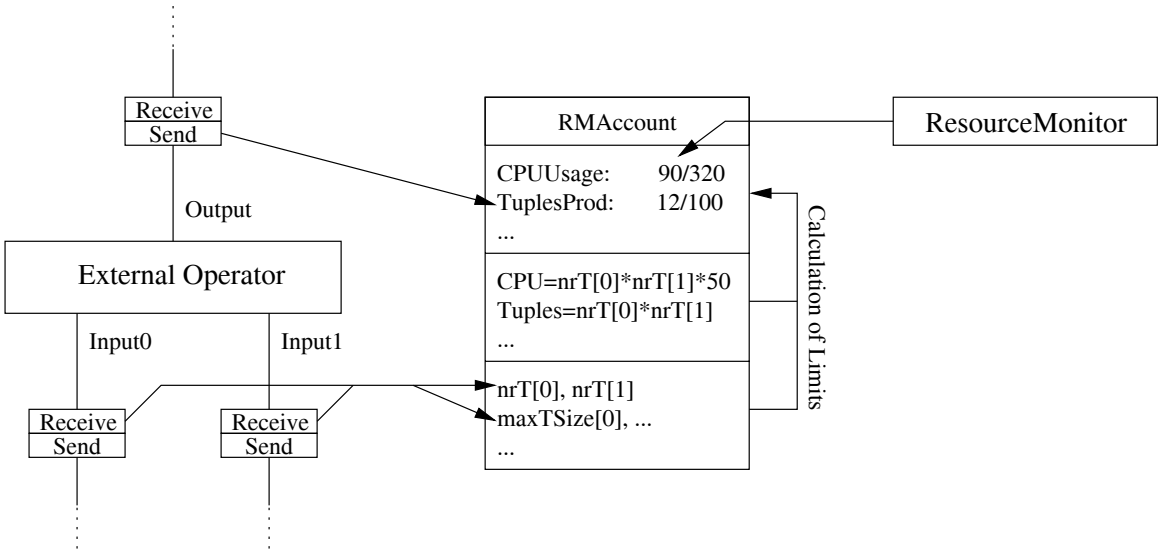


Figure 3.4: Architecture of the Resource Monitoring Component

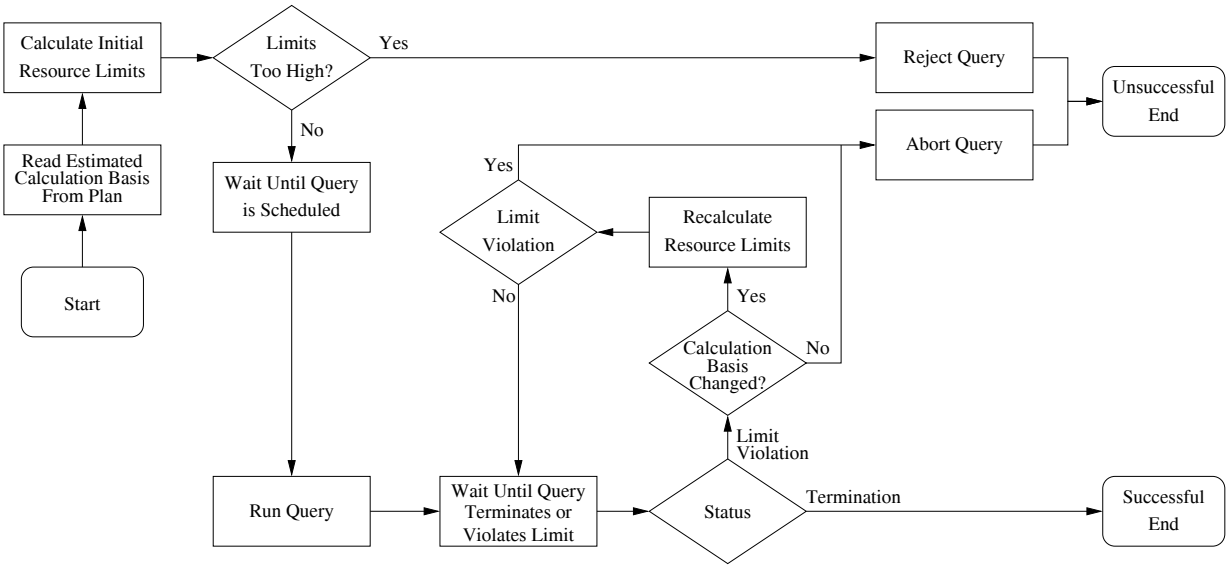


Figure 3.5: Flow Chart of Supervised Plan Execution

Figure 3.5 illustrates the workflow which is executed when a user-defined operator is executed. The optimizer annotates query plans with estimated values for the number and size of the tuples that the different operators will produce. Using these values and the cost models, a cycle provider is able to calculate the initial resource limits for an external operator. If there is any limit violation, the monitoring component verifies if the limits have to be adapted dynamically, i.e., if changes to the initial calculation basis of the cost models has occurred. For example, if the optimizer estimated that the underlying operator will produce 100 tuples but it has produced 110 tuples so far, the limits are adapted dynamically. If the newly calculated resource requirements of the operator exceed the upper resource limits set by the cycle provider, the plan is aborted. The plan is also aborted if the newly calculated resource limits are still too low to satisfy the current resource demand of the operator. Otherwise, the operator is allowed to resume until there is another limit violation or the operator terminates normally.

3.6 Correctness Issues of the Runtime Security System

As already mentioned, users of the ObjectGlobe system are interested in the privacy and integrity of data processed during query execution. Cycle providers could always manipulate the system in order to access or modify data without authorization, so users can restrict the cycle providers used to execute a plan to a set of trusted cycle providers.

3.6.1 Integrity of Data

There are two different situations in which the integrity of data is endangered: during the transport of data from data providers via cycle providers to the client and during the processing of data by operators at cycle providers.

Data integrity during transportation is warranted by secure communication channels using MACs. These channels can be used to construct a virtual private network, e.g., to always protect data in the Intranet of an enterprise. Another possibility is that the user annotates a plan to force application of MACs. This way, it is possible for both providers and users to be concerned with data integrity.

Integrity of data during processing of data by built-in operators is no problem because these operators are tested and work as expected. If arbitrary user-defined operators are used, integrity of data becomes a concern because ObjectGlobe cannot guarantee that these operators do not modify data unintentionally. Therefore, the user can specify a set of trusted operators by specification of their names, signatures, and/or function providers. A signature can, for example, confirm that an operator is tested by the OperatorCheck server (see Section 3.7). This way the user can minimize the risk of using malicious operators.

3.6.2 Privacy of Data

In ObjectGlobe, privacy of data means that data can only flow in a predefined way from data providers to users. There must not be any holes where an attacker or a malicious user-defined operator can leak the data, or a copy of it, and make it available to someone else. Unlike data integrity, ObjectGlobe can guarantee privacy of data even in case of the usage of arbitrary user-defined operators. To justify this statement, it is necessary to analyze in detail how a query is processed in ObjectGlobe.

Query processing in ObjectGlobe is composed of three stages: generation of the query plan (parse/lookup and optimization), distribution of the plan, and execution of the plan. In order to ensure privacy it is necessary to provide security during these three steps.

3.6.2.1 Plan Generation

Guarantee 1 *The plan generation stage provides a plan which cannot be modified without notice and which can only be executed once.*

A plan is generated by a user of the ObjectGlobe system using an SQL-like language. Assuming the parser and optimizer are working correctly, the user receives an XML representation of the plan. The user can verify the plan which later can be annotated with authentication information for, e.g., wrappers (where required). Additionally, the plan is signed using the user's private key (or a password), assuring integrity of the plan in the remaining query processing stages. To prevent reuse of (wire trapped) signed plans, every plan has a unique ID assigned containing a time stamp among other things. A signed plan can only be executed for a specified amount of time, e.g., for 60 minutes. Cycle providers store the IDs of processed queries until they are outdated.

3.6.2.2 Plan Distribution

Guarantee 2 *The plan distribution stage ensures that the query is instantiated as specified in the plan.*

Query plans are distributed in a straightforward way using the *host* annotations of the operators in the plan. Every cycle provider loads the code of external operators with a specialized ObjectGlobe class loader (OGClassLoader); the URL of the code is given in the *codeBase* annotation. If the plan or the cycle provider requires that the code be digitally signed, the OGClassLoader will check the signature of the code. Furthermore, all communication paths (including the paths for sending the plan) are established by built-in send and receive operators. If desired (i.e., specified in the annotations of the plan or required by the cycle provider), an SSL (Secure Sockets Layer) connection is established.

By using SSL, ObjectGlobe achieves not only the privacy and integrity of the plan during network communication, but also authenticates both communication partners of connections, assuring that a plan is distributed to the correct cycle providers. Cycle providers can authenticate function providers and check the digital signatures of operators to ensure that they receive and instantiate the expected operators.

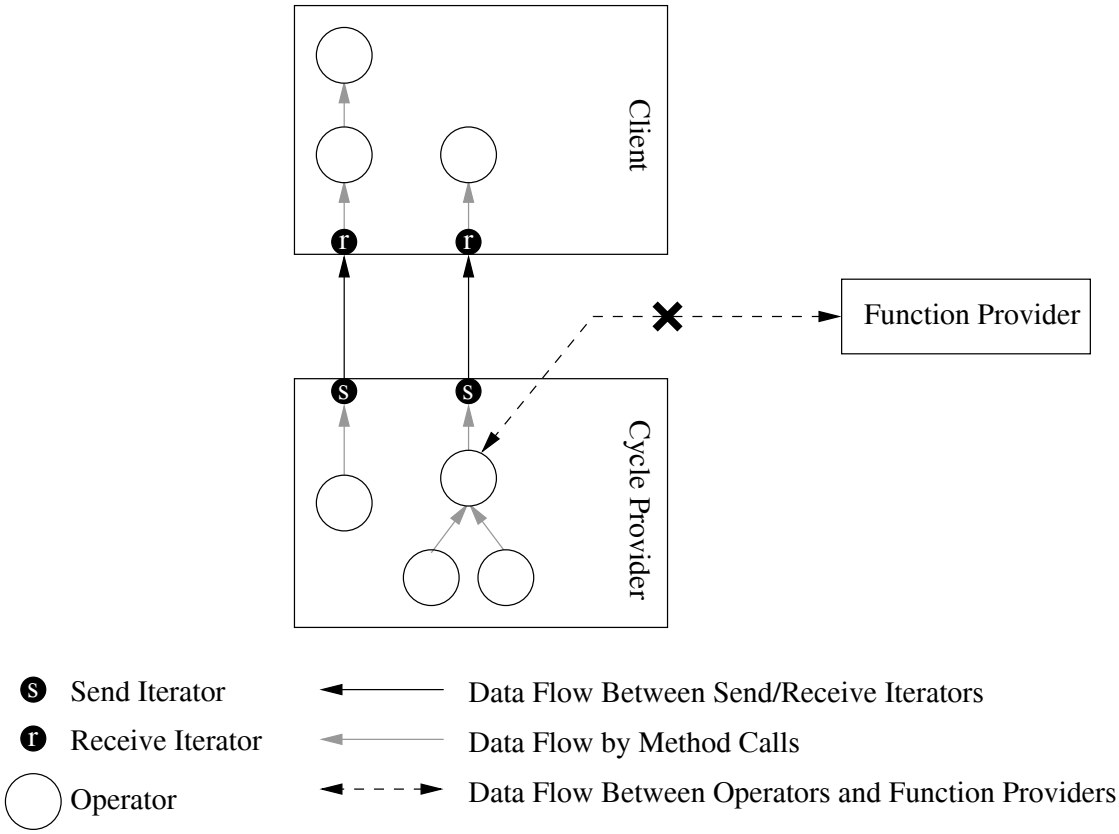


Figure 3.6: Overview of the Communication Channels During Plan Execution

3.6.2.3 Plan Execution

Privacy during plan execution is achieved by granting user-defined operators access only to controlled communication channels. These channels are presented in Figure 3.6, showing two queries running at several cycle providers (the client is a cycle provider, too). The arrows indicate the direction of data flow. As explained above, there is always a pair of built-in send/receive operators between host boundaries, which is created during plan distribution. Local communication between operators belonging to the same query is done by calling the open, next, and close methods from the succeeding operators.

Guarantee 3 *During plan execution there are no other connections except those shown in Figure 3.6.*

The communication channels shown in the figure are provided by ObjectGlobe core functionality. Other channels are prevented by the security system of ObjectGlobe. User-defined operators cannot access arbitrary classes of the ObjectGlobe core system because they reside in a different name space. Furthermore, user-defined operators are not able to load classes into packages belonging to ObjectGlobe or the Java API preventing them from

superseding classes or achieving widened access rights based on the fact that the classes belong to the same package. Thus, user-defined classes can neither manipulate the classes of Java and ObjectGlobe nor can they access arbitrary methods of the ObjectGlobe core system.

User-defined operators have no direct access to local resources like network sockets due to the security manager. Even network connections from OGClassLoaders to function providers are not available during plan execution any more. The classes of user-defined operators must be combined into a single JAR file. During query distribution, the ObjectGlobe system requests the JAR file for the operator specified in the query plan. This file is loaded and cached. Subsequent requests of a user-defined operator to dynamically load classes are satisfied using the JAR file. If the requested class is not available there it cannot be loaded at all. Thus, user-defined operators cannot abuse class loaders as covered channels by requesting classes (which are actually not needed and do not exist) with information about the processed data coded into the class name. Altogether, user-defined operators are prevented from sending data to other hosts, restricting communication to the local host.

The security manager also prevents communication between operators belonging to different queries, e.g., via a local network connection or a file. The remaining way of communication between different queries is the use of method calls. In order to do so, an operator needs a reference to another operator, but these references are retained by ObjectGlobe. Therefore, operators could only communicate using static members (class members) or static methods. But every query is instantiated by a separate OGClassLoader instance, implicitly separating the name spaces of queries. Thus, if several queries are using the same operator, there are several independent instances of a class residing in different name spaces. As consequence there are several sets of static members, restricting communication to operators of the same query.

There is one possible communication channel left, namely the only available access to local resources: temporary files. Access to temporary files is controlled by ObjectGlobe. When a user-defined operator claims a temporary file, ObjectGlobe generates a new empty file. The implementation of file access in Java prevents reading of data that was previously stored at the location of the new file (just as it prevents reading from uninitialized variables). Thereafter, the operator does not have direct access to the file but only to streams to write into and read from this file. Since user-defined operators do not have direct access to the local file system, only the operator which created such a file can access it.

Guarantee 4 *The connections shown in Figure 3.6 create no risk for privacy.*

Method calls can only be used to transport data within the same query. The implementation of send and receive operators ensures that user-defined operators can only call methods which are uncritical. The network connection between these operators is established in the query distribution stage and cannot be changed afterwards. This connection can be encrypted to ensure privacy during transportation of the data.

3.7 Quality Assurance for External Operators

Using the security measures presented so far, we are able to execute arbitrary external operators without risk to cycle providers and privacy of data. Nevertheless, it would be advantageous if the system could in advance verify the semantics of new external operators, examine their behavior under heavy load, and compare their resource consumption with given cost models. If an operator is well-behaved, ObjectGlobe could renounce security measures and execute the code at full speed or it could relax the sandbox of an operator. Several methods of software verification and testing have been developed so far (see [Mye79] for an overview), but it has also been shown that in general the correctness of arbitrary code cannot be proved [HH76].

3.7.1 Goal of Testing

Testing is a verification technique used to find bugs by executing a program. The testing process consists of designing test cases, preparing test data, running the program with this test data, and comparing the results to the correct results. An oracle is consulted for the correct result of certain test data. This could be a human, a reference implementation, or an interpreter of a (formal) specification of the program. While the design of good test cases requires some ingenuity, test data can sometimes be derived automatically. For automated testing, a test driver is necessary to feed test data to the function and to receive and record the results.

Methods for deriving test cases can be divided into two classes—white-box and black-box testing—depending on whether or not the source code is available. [Mye79] provides a detailed description of the most important techniques. We are focusing on black-box testing.

3.7.2 Methods of Formal Specification

As the correctness of a program depends on what it is specifically supposed to do, a complete and consistent specification is necessary. If testing should be processed automatically, a formal specification is required so that an interpreter can determine whether or not a calculated result is correct. There are two classes of formal specifications: *operational techniques* describe a way how the result can be calculated. Their advantage is that the correct result can be determined in advance and compared to the result of the program. However, they will not choose the most efficient way and hence are not a viable alternative to the real program. In contrast to that, *descriptive techniques* specify what the result should look like. Although the correct result cannot be calculated, the result of the program can be checked against them. Moreover, they usually are even more concise than operational specifications.

We have investigated several methods of formal specification, e.g., SQL, Haskell, Prolog, and mathematical formulae. Table 3.1 shows these specification methods for the Skyline

operator. For our purpose, the best choice is to use a purely functional language like Haskell [Bir98] because coding is quite straight-forward and efficient.

Especially in the database context, not only is the correctness of the result important but also the efficiency of its computation. Therefore, the specification of an external operator is augmented with several cost models. For each supervised resource the user can specify a worst-case cost model, which is a function of the extent of the input relations, namely their number of tuples, their maximum tuple size, and their total data size. If any resource is overconsumed, the operator is considered faulty and aborted immediately in a real application. Nevertheless, the cost models chosen should not be too generous, because then a cycle provider might refuse to instantiate the operator.

3.7.3 User-Directed Test Data Generation

As stated before, the design of good test cases requires some ingenuity. Thus, in our implementation, it is possible to direct the generation of test data so that they fulfill the preconditions of operators as well as meet the testers' strategies. Testers may want to specify single attribute values, enforce functional dependencies between attributes, establish relationships between relations, and control the order of the tuples. Therefore, the generation is done in three steps. First, the relation is created and the attribute types and number of tuples are specified. Second, all attribute rows are filled by random values or by referencing other relations. Third, the relation can be sorted or permuted some other way.

Possible domains for attributes are Boolean, Integer, Real, and String. Boolean values are $true$ and $false$. Integers are taken from a set $\{min, \ldots, max\}$, Reals from an interval $[min, max]$. For Strings, only the minimal and maximal string lengths are defined. The tester may also specify a set $\{x_1, \ldots, x_n\}$ from which the attribute values are drawn. *Null* values are not supported yet, because Haskell cannot deal with them.

For a single attribute column the values can be generated randomly or deterministically. The latter means that the values are taken one after the other in increasing order. If there are more tuples than different values, the procedure is started cyclically again. Random values can be taken uniformly from their possible values. In order to simulate functional dependencies and primary keys, it is important that unique values can be generated. [DeW93] presents an algorithm that produces random numbers with this property. For Real values, other distributions are possible such as normal distribution or exponential distribution. Foreign key relationships can also be simulated. For 1:1 relationships, the attributes of the other relation can be copied one after the other or be referenced unique-randomly. For 1:N relationships, a uniform random reference should be applied. Occasionally the order of the tuples matters. Thus, the relation can be sorted by the values of an attribute. Moreover, a shuffle operation has been implemented that permutes the tuples of a relation. This is useful to create a slight disorder. A factor between 0.0 (identity) and 1.0 (completely random shuffle) describes how far a tuple can move relative to the cardinality of the relation.

	Skyline (S,$\succeq$)	Explanation
Formula	$\{s \mid s \in S \wedge \neg\exists t \in S : t \neq s \wedge t \succeq s\}$	This formula can be derived directly from the definition: "The Skyline of a set S consists of all tuples s that are in S and for which no tuple t exists in S that is different from s and dominates s."
Conditions	Pre $\equiv$ **true** Post $\equiv \forall s \in \text{Skyline}(S) :$ $(s \in S \wedge \neg\exists t \in S : t \neq s \wedge t \succeq s)$ $\wedge\ \forall s \in S \setminus \text{Skyline}(S) :$ $(\exists t \in S : t \neq s \wedge t \succeq s)$	There is no precondition, i.e., the operator can be applied to any set on which a partial ordering relation is defined. The postcondition describes which tuples may be in the Skyline (cf. the formula above) and which must not be left out, defining exactly the result.

SQL	`SELECT *`
	`FROM S s`
	`WHERE NOT EXISTS (`
	`  SELECT * FROM S t`
	`  WHERE t≠s AND t≻s);`
		This is the naive approach to calculate the Skyline. Each tuple is compared to every other and is only selected if it is not dominated by any other tuple. $\neq$ and $\succeq$ must be adapted to the specific scenario.

Prolog	`skyline(S,R) :- skyline'(S,S,R).`
	`skyline'([],T,[]).`
	`skyline'([X|S],T,R) :-`
	`  dominated(X,T),`
	`  skyline'(S,T,R).`
	`skyline'([X|S],T,[X|R]) :-`
	`  not(dominated(X,T)),`
	`  skyline'(S,T,R).`
	`dominated(X,[Y|T]) :-`
	`  dominance(Y,X).`
	`dominated(X,[Y|T]) :-`
	`  dominated(X,T).`
	`dominance(Y,X) :- Y≠X, Y≻X.`
		The Skyline of a list S is R, if the result of a function `skyline'` that filters S with itself, i.e., compares each tuple with each tuple and deletes dominated tuples, is also R. If the empty list `[]` is filtered, the result is also empty.
		Now consider a list that contains at least one element X. If X is dominated by any tuple of the filter T, the result consists only of the rest of the list still to be filtered by T. Otherwise, X is taken over into the result.
		X is dominated by a non-empty list if it is dominated by the first element or by the rest. If the list is empty, X is not considered dominated (closed world assumption).

Haskell	`skyline :: [α] → [α]`
	`skyline ss = skyline' ss ss`
	`skyline' [] ts = []`
	`skyline' (s:ss) ts =`
	`  if dominated s ts`
	`  then skyline' ss ts`
	`  else s:skyline' ss ts`
	`dominated s [] = False`
	`dominated s (t:ts) =`
	`  dominance t s || dominated s ts`
	`dominance t s = (t≠s && t≻s)`
		`skyline` is a function that takes a list of elements of some type α and returns a list of elements of the same type.
		Like in Prolog, the Skyline of a list ss is the result of a function `skyline'` that filters ss with itself.
		Again, there is a distinction between the empty list `[]` and a list containing at least one element. This element is only taken over into the result if it is not dominated by any element of the filter ts.

Table 3.1: Specification Methods for Database Operators (Skyline)

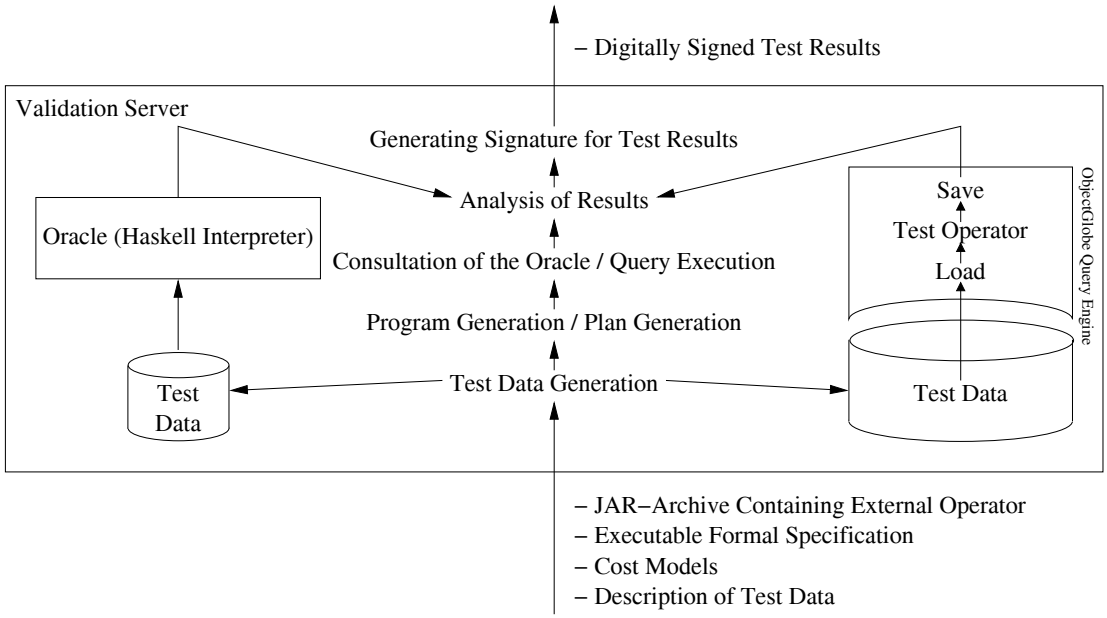


Figure 3.7: Architecture of the Operator Check Server

3.7.4 The OperatorCheck Server

We have implemented a server that checks external operators in ObjectGlobe by performing some tests on them. This server can be used by developers during development to test the implementation of operators. Additionally, trustworthy independent associations can use the server to check external operators and to generate digitally signed test reports.

Figure 3.7 shows the architecture of the server. The tester provides a JAR archive containing the external operator to be tested, a formal specification of the operator in Haskell (for a *correctness test*) or cost models (for a *benchmark test*), and some directives on how the test data should look like. For a correctness test, the server generates test data based on the directives and stores them on the hard disk. A Haskell program is built from the formal specification and the ObjectGlobe query execution plan is assembled. Now the test is performed: the Haskell interpreter and the ObjectGlobe system calculate their results. Afterwards, the results are loaded and compared, and the user receives the results of the test. For comparison, the semantics of the result must be taken into account. If the resulting relation is a list, the order of the tuples, as well as their count, is important. A multiset is a set where an element may occur several times. The order of the elements, however, is arbitrary. In a set, neither order nor count of elements matters.

It is also possible to perform a *reference test*. Instead of providing a Haskell specification, the user can also provide an ObjectGlobe query execution plan or an operator that serves as the oracle. The testing process works in an analogous way.

In a benchmark test no oracle is consulted, but the test operator is executed several times using different sizes of input data. Instead of a formal specification, the user provides cost models for several resources. The consumption of these resources is measured and

compared to the cost models. The test result shows the resources actually consumed and the maximum consumption allowed by the cost models. Using large input sizes, a stress test can be carried out that examines the behavior of the operator under heavy load and checks whether its performance degenerates or is still in accordance with the cost models.

3.7.5 Limitations of Testing

E. W. Dijkstra noted that "program testing can be used to show the presence of bugs, but never to show their absence" [DDH72]. Nevertheless, testing provides a practicable and promising way to find bugs in a piece of code, thus improving confidence in it. Several sophisticated methods of deriving "good" test cases have been developed. Under the hypothesis that the tested program behaves the same way for all test data of an equivalence class, the correctness can even be guaranteed by successful tests with one representative of each class. Malicious operators, however, intentionally destroy the uniformity hypothesis. This can only be detected by a white-box test which inspects the source code. Therefore, it is still necessary to take further measures for absolute security.

3.8 Usage Scenarios and their Security Implications

As we have seen, ObjectGlobe offers a very powerful security system which can easily be adapted to different scenarios. Thus, the amount of work to be done by the security system can be kept as small as possible depending on the hostility of the environment. The applications of an ObjectGlobe system can, e.g., be distinguished according to the openness of the underlying network. In the following sections we describe three different scenarios with varying levels of openness and the resulting security requirements.

3.8.1 Intranet

An Intranet is a controlled network within an organization and therefore access is restricted to a limited group of authorized users, i.e., the employees of the company. ObjectGlobe's cycle, data, and function providers are located within the Intranet and all query operators are implemented by employees of the company or obtained from trustworthy third party suppliers. Therefore, it is not necessary to monitor the resource consumption of these operators and they can be executed in privileged mode, e.g., these operators are granted privileges to access the disk or establish network connections if necessary. To avoid that operators are manipulated, they should be digitally signed (authenticated) by a responsible security administrator of the ObjectGlobe system. Execution can then be restricted to these digitally signed operators. If there is a need for secure communication (e.g., if there are outposts), ObjectGlobe can establish secure communication channels itself or it can rely on underlying network layers (e.g., hard- or software transparently enabling a virtual private network).

3.8.2 Extranet

An Extranet is a network that is used by different companies, e.g., by a company and its suppliers, forming a virtual enterprise. An important example of an Extranet is an electronic marketplace. There are many different scenarios in which virtual marketplaces can be run, but we assume in this example that the core cycle and function providers of the marketplace are operated by a trusted third party which is also responsible for the digitally signing of external operators. Within the Extranet these authenticated operators can be executed with additional privileges. Every participant of the marketplace operates at least one data provider to supply its product catalog and offers operators to access it, but it can operate additional cycle providers, too. The task of such cycle providers could be to execute external operators developed by the participants themselves, either because the marketplace does not trust the operators or because the participants do not want others to execute their operators to prevent, e.g., decompilation of the operators. As in the Intranet scenario there are several built-in possibilities to achieve secure communication.

3.8.3 Internet

The (global) Internet is the most challenging environment and it requires the full featured security system of ObjectGlobe. As mentioned in Section 3.2, protecting the sensitive resources of cycle providers is necessary because external operators could contain hostile code. There are many external operators which are not signed or which are signed by unknown third parties and, thus, cannot be trusted. With its effective security system, ObjectGlobe is able to execute such operators in a restricted sandbox, thereby guaranteeing security and availability of the system. Furthermore cycle providers are protected against denial of service attacks by the resource monitoring component of ObjectGlobe.

3.9 Related Work

There are a lot of extensible database systems allowing the implementation of user-defined functions as predicates or general functions/operators in C, C++, or Java. Examples for such systems include Postgres [SR86], Iris [WLH90], Starburst [HCL+90], and our distributed system of autonomous objects called AutO [KSKK99], but there are also several commercially available systems like Informix, Oracle, and DB2. The AutO system was also developed at the University of Passau and we adopted some fundamental results from the security system of AutO for ObjectGlobe. The systems mentioned above are all more or less exposed to the same security risks as ObjectGlobe, even if they do not load untrusted code dynamically from function providers like ObjectGlobe does. The security measures of most systems are not appropriate to guard the database system against attacks by such code. Thus, only administrators are allowed to augment the functionality and they must take care that the extensions are well-behaved and non-malicious. For example, DB2 implements the SQL99 standard for user-defined functions and provides the possibility to specify a function as `FENCED` to execute it in its own process. In this way, the internal

structures of the database system are protected, but denial of service attacks are still possible. Recently, with the development of Java as a secure programming language, some new considerations have been taken into account. [GMSvE98] have compared the efficiency of several designs using the Predator database server: the naive approach of putting user-defined functions directly into the server process, running them in a separate process and communicating with the server process via shared memory, and accessing Java user-defined functions via the Java Native Interface (JNI). Their conclusion is that Java is a bit slower on average while being a viable and secure alternative, although it still faces the problem of denial of service attacks. [CMSvE98] recommend to use a resource accounting system like JRes to guard against denial of service attacks, bill users, and obtain feedback that can be used for adaptive query optimization. To neutralize resource-unstable functions, they restrict CPU usage, number of threads, and memory usage to a fixed limit which is not appropriate for complex operators.

Beside database systems, there are, e.g., Java operating systems which must ensure security because enforcing resource limits has been a responsibility of operating systems for a long time. Another task of operating systems is to separate applications to avoid interference. Using our security system, a Java operating system could limit the amount of damage that a compromised application could do to the system and other applications on the system as described in [DC01]. This paper proposes the usage of Trusted Linux as secure platform for e-services application hosting because it adequately protects the host platform as well as other applications if an application is attacked and compromised.

The OperatorCheck approach is used to validate the semantics and to analyze the quality of operators. Thus, the quality of service of validated operators is higher than that of untested operators. This leads to a more reliable query execution, continuously available cycle providers, and better result quality. A more detailed motivation for the importance of these aspects can be found in [Wei01]. Obviously, for upcoming service platforms like the Sun ONE framework [Sunb], IBM WebSphere [IBMb], and Microsoft .NET [NET], those quality considerations will also play a very important role.

3.10 Conclusions

We presented an effective security framework for distributed and open systems and used ObjectGlobe as an example. We focused on security requirements of cycle providers and users. The security requirements of users are satisfied by the OperatorCheck server which is used to rate the quality of external operators and test their semantics. Privacy of data is guaranteed by isolating external operators and by using secure communication channels. Cycle providers are protected using a monitoring component which tracks resource consumption of external operators to prevent them from resource monopolization and an admission control system to guard providers against overload situations. A security manager and class loaders are used to protect cycle providers from unauthorized resource accesses and to shield the ObjectGlobe system from external operators. Additionally, we presented

the authentication framework of ObjectGlobe which can be used by cycle providers to determine the identity of users in a reliable way.

The security system can easily be adapted to other applications, e.g., Web application servers using server-side Java components such as Servlets, Java Server Pages, or Enterprise Java Beans to generate dynamic Web content. Nowadays it is common, to outsource one's own Web server to specialized suppliers. Using our security system, suppliers of such services can set resource limits to, e.g., Java Server Pages. Of course, there are some necessary adaptions to the security system. For example, server-side components usually do not have real cost models. In most of these cases, however, it is sufficient to use fixed resource limits. Additionally, the resource monitoring component can be used to establish a "per-resource" instead of, for example, a flat-rate tariff structure.

Chapter 4

ServiceGlobe - A Distributed and Open Web Service Platform

Web services are a new technology for the development of distributed applications on the Internet. By a Web service (also called service or e-service), we understand 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]. Web services are running within Web service platforms, also called service oriented architectures (SOAs). 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 IBM WebSphere, Microsoft .NET, or Sun ONE show this development. All these frameworks implement–respectively use–Web service standards published by the World Wide Web Consortium (W3C) or other consortia, e.g., SOAP, WSDL, and UDDI.

Parts of this chapter have already been presented in [KSK03a, KSK03b, KSK02]. A demo of the ServiceGlobe system was given at the VLDB'02 conference [KSSK02].

This chapter is organized as follows: In Section 4.1 we give a short introduction to Web service standards that are important for our work. In Section 4.2, we present the architecture of our Web service platform ServiceGlobe. ServiceGlobe's load balancing and service replication framework is presented in Section 4.3. Finally, Section 4.4 presents related work.

4.1 Web Services Fundamentals

There are several XML-based standards in the area of Web services. We will give a brief survey of the most important standards needed to understand this work.

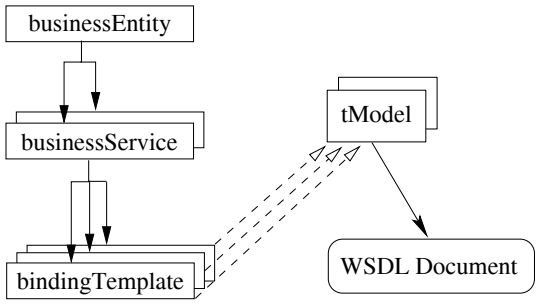


Figure 4.1: UDDI Data Structures

4.1.1 Web Service Registry UDDI

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 can be identified which set-up the basic schema, as shown in Figure 4.1: *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.

- **businessEntity:** This data structure gathers information about an entire company or party which offers a family of services. For example, a dealer 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 dealer may have product information and selling services. It also contains one or more bindingTemplates specifying binding information for this service.

- **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, i.e., host connected to the Internet which is running the ServiceGlobe runtime engine, is specified by a bindingTemplate with its URL as an access point. A bindingTemplate may have several references to tModels.

- **tModel:** tModels describe as a technical fingerprint various concepts and classifications. In ServiceGlobe, for example, tModels are used as functionality descriptions for services, like retailing or service hosting. The tModel may contain a link to a WSDL document (see Section 4.1.3) which specifies the signature of the service in de-

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

Figure 4.2: Basic Structure of a SOAP Message

tail [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 documents to communicate with the service (input parameters, output parameters, 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.

4.1.2 Communication Protocol SOAP

SOAP (Simple Object Access Protocol) [BEK+00, 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 (Hypertext Transfer Protocol), SMTP (Simple Mail Transfer Protocol), and FTP (File Transfer Protocol). The usage of SOAP over HTTP is the default 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 more complex tasks like request/response message exchange or even RPC (Remote Procedure Call).

Figure 4.2 shows the basic structure of a SOAP message consisting of three parts: an envelope, an optional header, and a mandatory body. The root element of a SOAP message is an `Envelope` element containing an optional `Header` element for SOAP extensions and a `Body` element for the payload. 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].

SOAP offers a standard *encoding style*[1], 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.

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

4.1.3 Web Service Description Language WSDL

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 work, but we will give a brief overview of the WSDL standard. At 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. 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*. Messages are serialized according to a set of rules defined by an encoding style. At last, a service in WSDL is defined as a set of *ports*, i.e., bindings with associated network addresses (normally URLs).

4.2 Architecture of ServiceGlobe

The ServiceGlobe system is a distributed and open service platform. It is fully implemented in Java Release 2 and is based on standards like XML, SOAP, UDDI, and WSDL. Additionally, the system supports mobile code, i.e., services can be distributed and instantiated on demand during runtime at arbitrary Internet servers participating in the ServiceGlobe federation. Of course, ServiceGlobe offers all the standard functionality of a service platform like SOAP communication and a transaction system. These areas are well covered by existing technologies and are therefore not the focus of this work. In this section, we present the basic components of the ServiceGlobe infrastructure. First of all, we distinguish between external and internal services (see Figure 4.3).

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

Internal services are native ServiceGlobe services implemented in Java. They are using the service API provided by the ServiceGlobe system. ServiceGlobe services use SOAP to communicate with each other. Services receive a single XML document as input and gen-

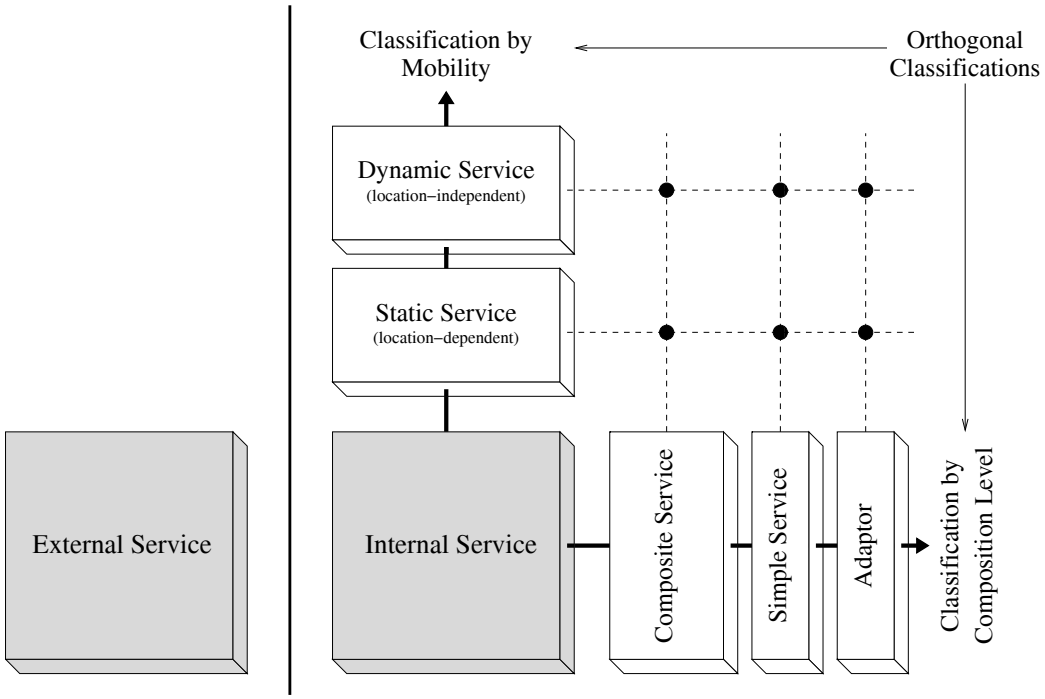


Figure 4.3: Classification of Services

erate 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 because they, for example, require access to certain local resources like a DBMS. 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. 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-based tool to draw a representation (similar to a workflow graph) of a composite service, but that is not the focus of our work.

Internal services are executed on *service hosts*, i.e., hosts connected to the Internet which are running the ServiceGlobe runtime engine. ServiceGlobe's internal services are mobile code. Therefore, their executables can be loaded on demand from *code repositories* into a service host's runtime engine (this feature is called *runtime service loading*). A UDDI server is used to find an appropriate code repository storing a certain service. 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 the security system described in Chapter 3. 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 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, large 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 the flexibility needed for load balancing or optimization issues.

[KSK03a] describes a sophisticated technique called *dynamic service selection* which is now an integral part of ServiceGlobe. It provides a layer of abstraction for service invocation offering Web services the possibility of selecting and invoking Web services at runtime based on a technical specification of the desired service. The selection can be influenced by using different types of constraints. [KK04b, KK04a, KSKK03] present a context framework that facilitates the development and deployment of context-aware adaptable Web services in ServiceGlobe.

4.3 Basic Load Balancing and Service Replication Framework

For large-scale, mission-critical applications such as an enterprise resource planning system like SAP with thousands of users working concurrently, a single service host is not sufficient to provide low response times. Even worse, if there are any problems with the service or the service host, the service will be completely unavailable. Such downtime can generate high costs even if a service host is only down for some minutes. Therefore, it is necessary to run several instances of a service on multiple service hosts for fault tolerance reasons. Moreover, a load balancing component is needed to avoid load skew. A server blade architecture (see Section 6.1) is very beneficial for this purpose because scale-out of computing power can be done on demand by adding additional server blades. Of course, a traditional cluster of service hosts connected by a LAN can be used as well but with higher total cost of ownership (TCO) and normally slower network connections.

Since it is very expensive and error-prone to integrate the functionality for the cooperation of the service instances directly into every new service, we propose a generic solution to this problem: a modular *dispatcher service* which can act as a proxy for arbitrary services. Using this dispatcher service it is possible to enhance many existing services or develop new services with load balancing and high availability features without having to consider

these features during their development. All kinds of services are supported as long as concurrency control mechanisms are used, e.g., by using a database as back-end (as many real-world services do). The concurrency control mechanisms ensure a consistent view and consistent modifications of the data shared between all service instances. Of course, if there is no data sharing between different instances of a service, the dispatcher can be used as well. An additional feature of our dispatcher is called *automatic service replication* and it enables the dispatcher to install new instances of static services on demand.

4.3.1 Architecture of the Dispatcher

Our dispatcher is a software-based layer-7 switch[2]. Such switches perform load balancing (or load sharing) using several servers on the back-end with identically mirrored content. They use a dispatching strategy like round robin or more complex strategies which are using load information about the back-end servers. Our solution is a pure software solution and—in contrast to existing layer-7 switches—is realized as a regular service. Thus, our dispatcher is more flexible, extensible, and seamlessly integrated into the platform.

Figure 4.4 shows our dispatcher monitoring three service hosts which are running two instances of service S (both connected to the same DBMS). The database server is monitored as well, using a stand-alone monitoring application. Using information from monitoring services and monitoring applications, the dispatcher generates the dispatcher's local view of the load situation of the service hosts. Upon receiving a message (in this case for service S), the dispatcher looks for the service instance running on the least loaded service host and forwards the message to it. As already mentioned, our dispatcher is modular, as shown in Figure 4.5. There are four types of modules:

- **Operation Switch Module:** This module controls the operation mode of the dispatcher on a per-service level. In our implementation, the standard operation mode is *forward*. Other modes are *buffer* or *reject*. The latter two modes are set to prevent the more expensive execution of the dispatch module when there are no suitable service hosts.

- **Dispatch Module:** This module implements the actual dispatching strategy. It can access the load situation of service hosts and of other resources for the assignment of requests to service instances. Possible results of a dispatch strategy are an assignment of a request to a service instance, a command to initiate a service replication (see below), a reject command, or a buffer command. We implemented a strategy which assigns requests to the service instance on the least loaded service host based on the CPU load. We additionally implemented a more sophisticated strategy which handles the load of CPU and main memory on different types of resources (e.g., service hosts and database management systems) needed for the execution of a service. This strategy prevents overload situations not only on service hosts but also on other resources like DBMSs.

[2]This kind of switch is also used in the context of Web servers [CCCY02].

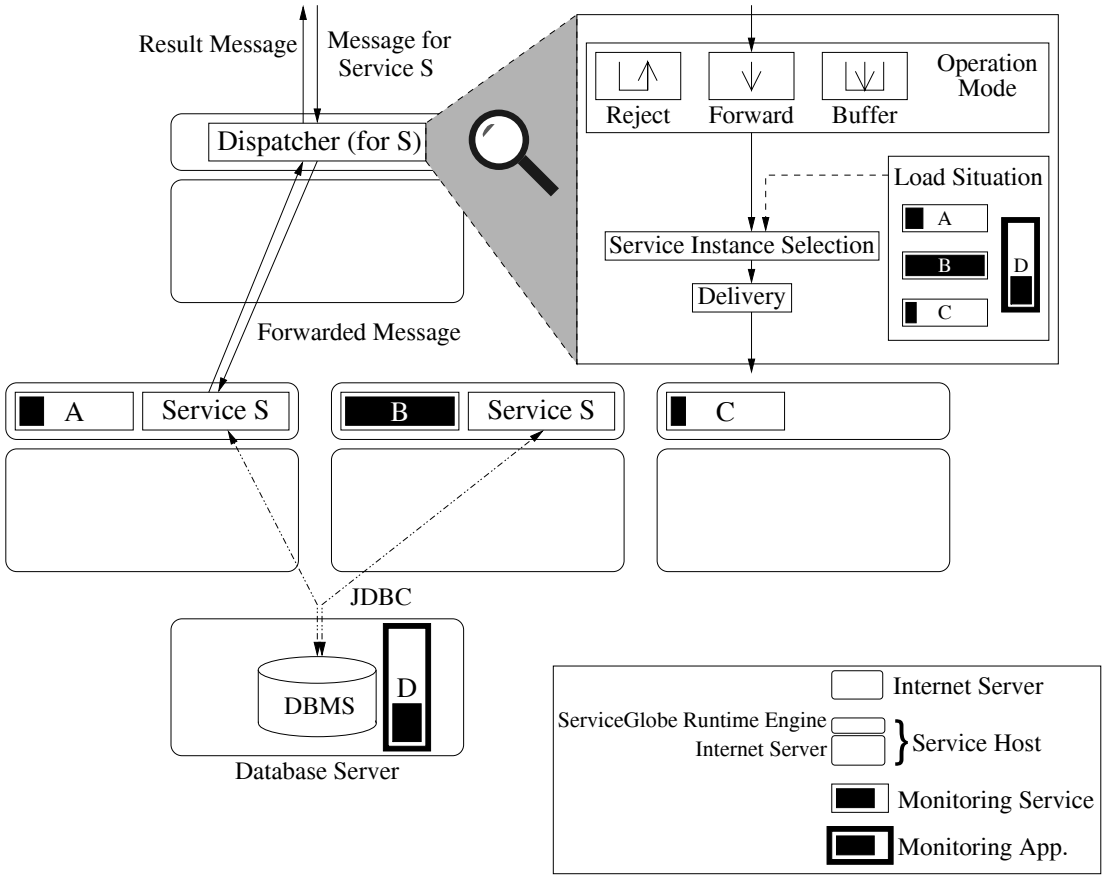


Figure 4.4: Survey of the Load Balancing System

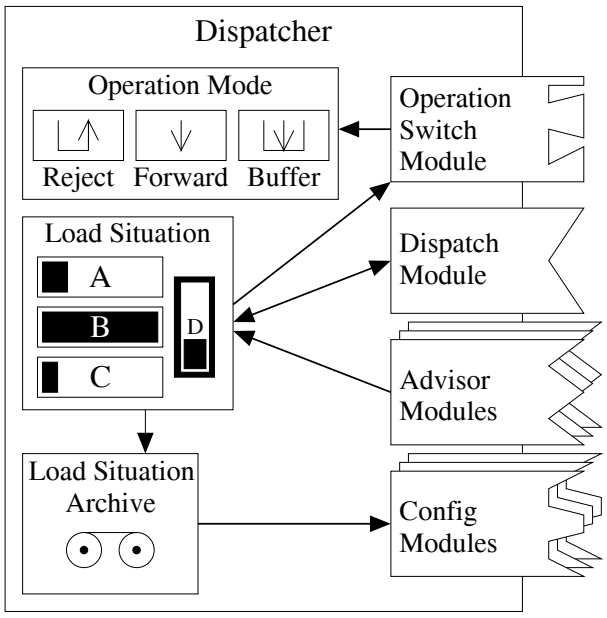


Figure 4.5: Dispatcher's Architecture

- **Advisor Modules:** Advisor modules are used to collect data for the dispatcher's view of the load situation of all relevant resources. We implemented advisor modules to measure the average CPU and memory load on service hosts (using the monitoring services) and on hosts running database management systems (using the monitoring applications). There are lots of different reasonable advisor modules. The simplest kind of advisor module only knows two conditions of a resource: available or unavailable. For service hosts, this could be achieved by a simple *ping* on the host running the ServiceGlobe system. More complex advisors can provide more detailed information like CPU or main memory load of a service host, or the load of a database management system depending on CPU, memory, disc I/O, and others.

- **Config Modules:** The configuration modules are used to generate the configuration for new service instances (see Section 4.3.3). The modules can access the load situation archive which stores aggregated historic load information. This is very beneficial if there are, e.g., several instances of a database system working on replicated data. Using historic load information, a new service instance can be advised to connect to the instance of the DBMS which had the lowest average load in the past.

To turn an existing service into a highly available and load balanced one, a properly configured dispatcher service must be started. Additionally, the dispatcher must be registered at the UDDI repository. Already existing UDDI entries of the service instances and service hosts have to be modified so that all service instances and all service hosts can be found by the dispatcher. After that the service instances are no longer contacted directly, but are accessible via the dispatcher service controlling the forwarding of the messages. A cluster of service hosts can easily be supplemented with new service hosts. The administrators of these service hosts only have to install the ServiceGlobe system and register them at the UDDI repository using the appropriate tModel, e.g., ServiceHostClusterZ, indicating that these service hosts are members of cluster Z. The dispatcher will automatically use these service hosts as soon as it notices the changes to the UDDI repository.

4.3.2 Load Measurement

The dispatcher's view of the load situation is updated at intervals of several seconds to prevent overloading the network. Thus, this view is constant between two updates. Therefore, a service host SH will still be considered to have low load even if several requests have been assigned to it after the last load update. Without precautions the dispatcher might overload SH for this reason. To avoid these overload situations, the dispatcher adds "penalties" to its view of the load once a request is assigned. Figure 4.6 illustrates the load of SH, the load reported to the dispatcher (load without penalties), and the load with penalties.

The grey, thick line represents the load $L_{\mathrm{SH}}(t)$ of the service host SH. The dashed line represents the dispatcher's view $D'_{\mathrm{SH}}(t)$ of the load of SH, which is the average load of SH over the last update interval of length I_u. This average load is calculated by SH and sent

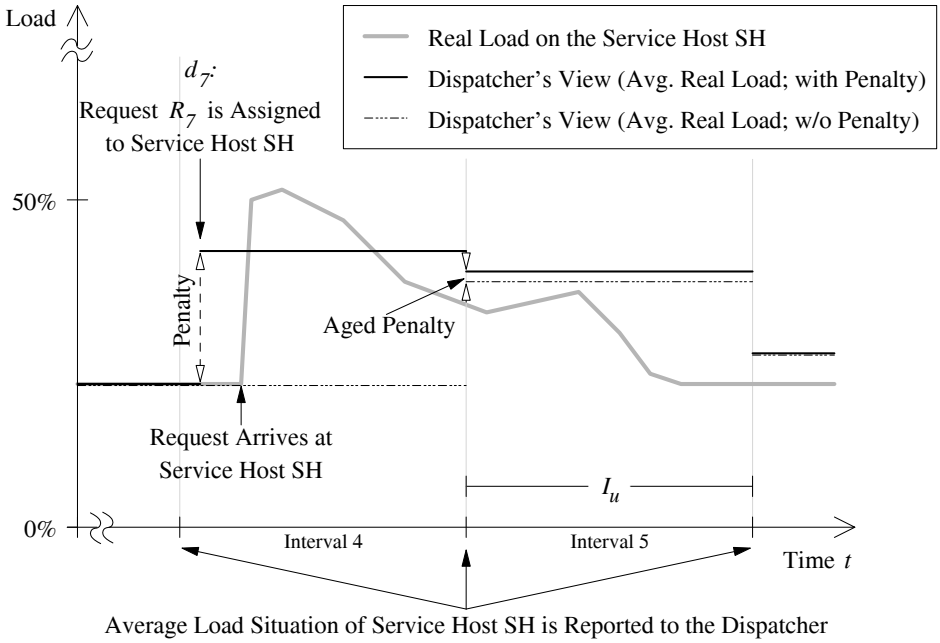


Figure 4.6: Different Views of the Load Situation during Request Dispatching

to the dispatcher at regular intervals. The function $\text{int}(t)$ calculates the number of the interval containing a given time t:

$$\text{int}(t) := \lfloor t / I_u \rfloor$$

The dispatcher's view can now be written as follows:

$$D'_{\text{SH}}(t) := \text{avg}\left\{ L_{\text{SH}}(t') \,\middle|\, \text{int}(t') = \text{int}(t) - 1 \right\}$$

The black, solid line shown in Figure 4.6 represents the dispatcher's view including penalties $D_{\text{SH}}(t)$. The initial (maximum) value of a penalty (represented by $P^m_{\text{SH,S}}$ in the equations) depends on the service S and the performance of the service host SH and is configurable. This way, every assignment of a request R_i, i.e., every dispatch operation (represented by $d_i, i \in \mathbb{N}$; d_7 in the figure), has an immediate effect on the dispatcher's view of the load situation. If there is a load update from SH shortly after an assignment of a request R_i but before SH started to process R_i, the associated penalty is lost if the dispatcher replaces its view with the reported load. This is due to the fact that the load does not yet include the load caused by R_i. Thus, the load reported by the load monitors and the dispatcher's view of the load situation are remerged using aging penalties: the penalties decrease over time and are added to further load values reported by the service host until the values of penalties reach zero. The time I_p until a penalty is zero is configurable and normally shorter than shown in the picture, e.g., twice the time a request R_i needs to arrive at SH plus the time SH needs to start processing R_i. After I_p, we assume that a request R_i arrived at SH and that the load caused by R_i is already included in the reported load so that there is no need for the dispatcher to add penalties for R_i any longer. Using our notation

and defining time(d_i) to indicate the time of the assignment d_i, host(d_i) to indicate the destination host of the assignment d_i, and service(d_i) to indicate the destination service of the assignment d_i, the view with penalties $D_{\mathrm{SH}}(t)$ can be calculated as follows: the penalty P_{d_i} for the assignment d_i is zero before the assignment. After I_p, it is zero again. In between this interval the penalty is calculated using a linear function $f_{d_i}(t)$ with the following constraints: $f_{d_i}(0) = P^m_{\mathrm{host}(d_i),\mathrm{service}(d_i)}$ and $f_{d_i}(I_p) = 0$.

$$P_{d_i}(t) := \begin{cases} 0 & \text{if} \quad t < \mathrm{time}(d_i) \vee t > \mathrm{time}(d_i) + I_p \\ f_{d_i}(t - \mathrm{time}(d_i)) & \text{else} \end{cases}$$

When receiving load updates from the service host SH, i.e., $t = x * I_u$ for $x \in \mathbb{N}$, the load including penalties is calculated by adding all aged penalties of assignments to SH to the reported value:

$$Ass_{\mathrm{SH}} := \left\{ a \in \mathbb{N} \,\middle|\, \mathrm{host}(d_a) = \mathrm{SH} \right\}$$

$$D_{\mathrm{SH}}(t) := D'_{\mathrm{SH}}(t) + \sum_{i \in Ass_{\mathrm{SH}}} P_{d_i}(t) \quad \text{if} \quad \exists x \in \mathbb{N} : t = x * I_u$$

Within an update interval, penalties of new assignments to SH, i.e., assignments done within the current update interval, are added to this load as soon as they occur:

$$NewAss_{\mathrm{SH}}(t) := \left\{ a \in Ass_{\mathrm{SH}} \,\middle|\, \mathrm{int}(\mathrm{time}(d_a)) = \mathrm{int}(t) \wedge t > \mathrm{time}(d_a) \right\}$$

$$D_{\mathrm{SH}}(t) := D_{\mathrm{SH}}\left(\mathrm{int}(t) * I_u\right) + \sum_{i \in NewAss_{\mathrm{SH}}(t)} P^m_{\mathrm{SH},\mathrm{service}(d_i)} \quad \text{if} \quad \forall x \in \mathbb{N} : t \neq x * I_u$$

4.3.3 Automatic Service Replication

If all available service instances of a static service[3] are running on heavily loaded service hosts and there are service hosts available which have a low workload, the dispatcher can decide to generate a new service instance using a feature called automatic service replication. Figure 4.7 demonstrates this feature: service hosts A and B are heavily loaded and host C currently has no instance of service S running. Thus, the dispatcher sends a message to service host C to create a new instance of service S. The configuration of the new service S is generated using the appropriate configuration module. If no service hosts with low workload are available, the dispatcher can buffer incoming messages (until the buffer is full) or reject them depending on the configuration of the dispatcher instance and the modules.

[3]Dynamic services can be executed on arbitrary service hosts and need not be installed, anyway.

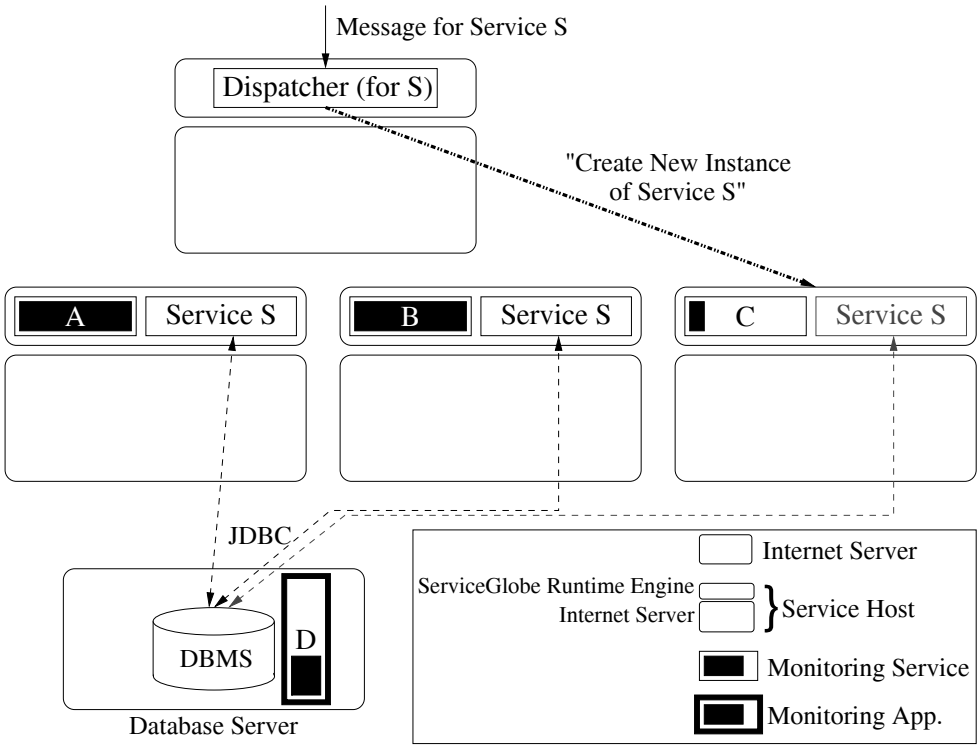


Figure 4.7: Automatic Replication of Service S

4.3.4 High Availability / Single Point of Failure

Using several instances of a service greatly increases its availability and decreases the average response time. Just to give an impression of the high level of availability, we want to sketch this very simple analytical investigation. Assuming that the server running the dispatcher itself and the database server (in our example the database server is needed for service S) are highly available, the availability of the entire system only depends on the availability α of the service hosts. The availability of a pool of service hosts can be calculated as follows:

$$\alpha = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \quad (1) \qquad \alpha_{pool} = \sum_{i=1}^{N} \alpha^i (1-\alpha)^{(N-i)} = 1 - (1-\alpha)^N \quad (2)$$

Equation 1 calculates the availability of a single service host based on its MTBF (mean time between failures) and MTTR (mean time to repair). The availability of a pool of N service hosts can be calculated using Equation 2. Even assuming very unreliable service hosts with MTBF = 48h and MTTR = 12h a pool with 8 members will only be unavailable about 1.5 minutes a year.

Because database management systems are very often mission critical for companies, there are different approved solutions [Bre98, HD91] for highly available database management systems. Thus, the remaining single point of failure is the dispatcher service. There

are several options for reducing the risk of a failure of the dispatcher. A pure software solution is to run two identical dispatcher services on two different hosts. Only one of these dispatchers is registered at the UDDI server. The second dispatcher is the spare dispatcher and it monitors the other one ("watchdog mechanism"). If the first dispatcher fails, the spare dispatcher modifies the UDDI repository to point to the spare dispatcher. If the clients of the dispatcher call services according to the UDDI service invocation pattern, any failed service invocation will lead to a check for service relocation. Thus, failures of the first dispatcher will lead to an additional UDDI query and an additional SOAP message to the second dispatcher. Of course, there are many other possible solutions which are adaptable for a highly available dispatcher service known from the fields of database systems [Bre98, HD91] and Web servers [CCCY02], including solutions based on redundant hardware. These solutions are outside the scope of this work.

4.4 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] and IBM WebSphere [IBMb], which are both based on J2EE [J2E], and Microsoft .NET [NET]. 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 [KSSK02, KSK02] and SELV-SERV [BDSN02] which focus on certain aspects in the Web service area. In SELV-SERV services with equal interfaces are grouped together into service communities. This project focuses on composing Web services using state charts. The main difference of ServiceGlobe is that it offers mobile services which can be executed on every service host.

A lot of work has been done in the area of load balancing, e.g., load balancing for Web servers [CCCY02] and load balancing in the context of Grid computing [Glo]. Grid computing is focused on distributed computing in wide area networks involving large amounts of data and/or computing power, using computers managed by multiple organizations. Our dispatcher focuses on distributing load between hosts inside a LAN. In contrast to dispatchers for Web servers [CCCY02], dispatchers for service platforms cannot assume that all requests to services produce the same amount of load because the computational demands of different services might be very different. There are also commercial products available, e.g., DataSynapse [Dat] which offers a self-managing distributed computing solution. One of the key differences of this system is that it works in a pull-based manner, i.e., hosts request work, instead of using a dispatcher to push work to the hosts. Additionally, DataSynapse requires an individual integration of every application, which is not necessarily an easy task for arbitrary applications.

Chapter 5

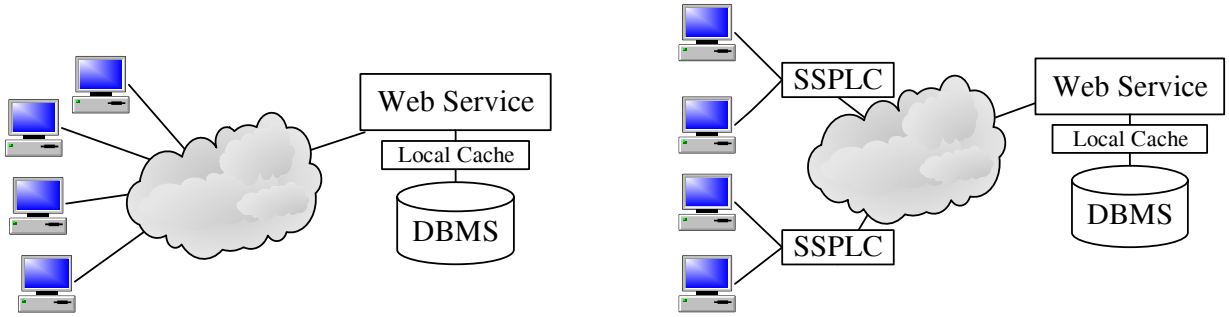
Semantic Caching for Web Services

In this chapter, we present a semantic caching scheme suitable for caching responses from Web services on the SOAP protocol level. Existing semantic caching schemes for database systems or Web sources cannot be applied directly because there is no semantic knowledge available about the requests to and responses from Web services. Web services are typically described using WSDL documents. For semantic caching we developed an XML-based declarative language to annotate WSDL documents with information about the caching-relevant semantics of requests and responses. Using this information, our semantic cache answers requests based on the responses of similar previously executed requests. Performance experiments—based on the scenarios of TPC-W and TPC-W Version 2—conducted using our prototype implementation demonstrate the effectiveness of the proposed semantic caching scheme.

This chapter is organized as follows: In Section 5.1 we describe how Web services are deployed today and motivate the usage of Web service caches. In Section 5.2 we present background information for semantic caching. Additionally, we introduce an example Web service based on the TPC-W scenario. This service is used to explain our semantic caching scheme. Several basic design decisions are described in Section 5.3. A detailed description of our Web service cache SSPLC, the embedded control instructions of service providers, and some sophisticated features of the SSPLC are presented in Section 5.4. Experimental results follow in Section 5.5. Section 5.6 surveys related work and Section 5.7 presents our conclusions.

5.1 Motivation

Service-oriented architectures (SOAs) based on Web services are emerging as the dominant application on the Internet. Mission critical services like business-to-business (B2B) or business-to-consumer (B2C) services often require more performance, scalability, and availability than a single server can provide. Server side caching [YFIV00] and some kind of cluster architecture alleviate some of these problems. Figure 5.1a) shows this central architecture. The computers on the left-hand side represent the clients, the cloud repre-



a) b)

Figure 5.1: Web Service Architecture Without (a) and With Distributed Caching (b) in a Highly Accessed System

sents the Internet. On the right-hand side, there is a Web service (possibly running on a cluster) using a database as back-end like many real-world services do. The local cache shown in the figure can be, e.g., a cache for the DBMS and/or a cache for XML fragments. A major drawback of this architecture is that all clients must still access the Web service directly over the Internet, possibly resulting in high latency, high bandwidth consumption, and high server load.

One solution to these scalability problems appears to be distributing Web service instances across strategic locations on the Internet, i.e., edge servers. A similar approach is already known in the context of traditional Web servers where static content like images, text, or videos is replicated on servers around the world using content distribution networks (CDNs) [INST02] like, e.g., Akamai [Aka]. This approach works well with traditional Web content assembled from a composite HTML page and other resources like images, referenced via URLs in the HTML page. Thus, static resources can easily be moved from the origin server to a CDN. However, this approach is not particularly suitable for Web services because their results are typically monolithic XML documents without links to other documents. Thus, the distinction between static and dynamic content is more difficult and the data is not available in predetermined fragments like images and HTML pages. Furthermore, applying this approach to Web services including their back-end databases requires replication of the application logic as well as utilization of some kind of distributed DBMS or local database cache for the service instances [GDN+03]. This must be done individually for every service and is very time-consuming and costly. Therewith, this is one of the main disadvantages of this approach.

There are many Web services characterized by many requests corresponding to read-only queries on their back-end databases and only a small fraction of requests actually initiating updates on the databases. One important category of services showing this kind of access pattern is business services (B2C and B2B) offering query-like interfaces to, e.g., access product catalogues. Such services are also used in standard benchmarks for B2C and B2B environments, e.g., TPC-W [TPPC02] and TPC-W Version 2 [TPPC03]. Users normally send many read-only *query-style* requests to find the products they are interested in before sending a few (generally not cachable) *transaction-style* requests to order the se-

lected products. Another important category of Web services includes information services like news services, weather services, etc., which typically offer read-only access. There are Web services with different access patterns but since the Web service categories described above are very common and important, this work focuses on them.

Our generic approach to achieving higher performance and scalability is called *Semantic SOAP Protocol Level Cache* (SSPLC). The performance increase is based on semantic caching of responses from Web services in request/response message exchange patterns on the SOAP [BEK+00, Mit03] protocol level. The resulting Web service architecture is shown in Figure 5.1b). Clients are not directly accessing the origin service anymore; instead they are accessing instances of SSPLC. As long as requests can be answered based on cached data, the origin server hosting the Web service is not involved anymore. Therefore, the load at the origin server is reduced, bandwidth consumption is diminished, and latency is reduced. The advantage of a semantic cache is that it reuses the responses to prior requests to answer similar requests, not only the exact same requests. Thus, if request R_1 retrieves all books written by "Kemper" and afterwards a request R_2 retrieves all books written by "Alfons Kemper", a semantic cache reuses the response to R_1 to answer the more selective request R_2.

Our proposed cache can be used like traditional HTTP proxies, i.e., SSPLC instances need not be hosted by service providers themselves, but can easily be run by, e.g., companies and universities, just like HTTP proxies nowadays. However, SSPLC can also be used as client cache, reverse-proxy cache, or edge server cache [RV02, RS01]. Because of synergy effects, there are major savings when the cache is used by a large number of clients, i.e., is not used as client cache. Additionally, if used as a reverse-proxy or an edge cache, server-driven cache consistency techniques are applicable.

Our approach relies on service provider cooperation. All instructions to control the SSPLC are embedded by the provider of a service in SOAP result documents and in the WSDL description of a service. The SOAP results are augmented with information about cache consistency. This is the only modification to a Web service required for the use of SSPLC. The effort necessary to generate these annotations depends on the consistency strategy and the complexity of the application logic and is subject to further investigations. Simple annotations, e.g., TTL values, can be inserted by the SOAP-engine in a post-processing step without modifications of the Web service. More complex annotations demand some coding effort. Additionally, the WSDL document of the service is annotated with information about the caching-relevant semantics of a service. This is done manually using an XML-based declarative language because automatic reasoning about the semantics normally results in a very conservative caching behavior. Developers of a Web service should not have problems writing these annotations because they already have the required knowledge. Using our declarative semantic annotations it is possible to formalize a considerable amount of application domain knowledge and knowledge about the semantics of the requests and responses of Web services to achieve effective caching behavior. Currently, we are describing semantics of individual operations without considering semantic correlations between different operations. Thus, SSPLC is internally organized as a set of *virtual caches* such that every operation has its own private virtual cache instance.

5.2 Background and Running Example

5.2.1 Fundamentals of Semantic Caching

Semantic caching is a client-side caching technique introduced in the mid 90s for DBMSs to exploit the semantic locality of queries, e.g., [CB00, DFJ+96, LC99, LC01, LKRPM01]. A semantic cache is managed as a collection of *semantic regions*. Semantic regions group together semantically related objects and are composed of *region descriptor* and *region content*. The descriptor basically contains a region predicate (like 'author = "Alfons Kemper" ') describing the region content. The region content stores the objects related to a region descriptor. Access history is maintained and cache replacement is performed at the granularity of semantic regions.

Every query sent to a semantic cache is split into two disjoint parts: a *probe query* and a *remainder query*. The probe query extracts the relevant portion of the result already available in the cache while the remainder query is sent to the origin server to fetch the missing, i.e., not cached, part of the result. If the remainder query is empty, the cache does not interact with the origin server. If a client wants to pose, e.g., a query $\mathcal{A}$ and the cache already contains the result for the query $\mathcal{A} \wedge \mathcal{B}$, it sends a remainder query $\mathcal{A} \wedge \neg\mathcal{B}$ to retrieve the missing results. In the context of DBMSs or Web sources, all participating components have been full-fledged DBMSs. Since Web services normally have a more constrained query interface, semantic caching must be adapted to these limitations (see Section 5.4).

5.2.2 Running Example

In this section, we will present an example Web service based on the TPC-W scenario. This service is used to explain our semantic caching scheme. Amazon offers a SOAP-based Web service interface [Ama] which is very similar to their broadly known HTTP interface. Since Amazon is in fact a "real-world implementation" of the TPC-W benchmark, we use parts of their interface for our example and the TPC-W benchmark scenario as basis for performance experiments conducted using our prototype implementation.[1] Our example service is called *Book Store Light* and is a slim version of Amazon. The relevant operation of this service is a search for books written by certain authors (*author search*). The XML documents used by Amazon are too large to be presented entirely in this work. We shortened and simplified them to a reasonable degree and removed all namespaces from the presented documents for better readability and a more concise presentation.

Figure 5.3 shows an example SOAP response corresponding to the request shown in Figure 5.2. As already mentioned in Section 4.1.2, SOAP offers a standard encoding style, 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. Since the techniques presented in this work are applicable independent of the concrete serialization

[1] We also used some other Web services listed by the online directory XMethods [XMe], e.g., Google and a recipe service, to verify the capabilities of our caching approach.

```
<Envelope encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <Body>
  <AuthorSearchRequest>
   <AuthorSearchRequest type="AuthorRequest">
    <author type="string">Alfons Kemper</author>
    <levelOfDetail type="string">lite</levelOfDetail>
   </AuthorSearchRequest>
  </AuthorSearchRequest>
 </Body>
</Envelope>
```

Figure 5.2: Example SOAP Request for Book Store Light

```
<Envelope encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <Body>
  <AuthorSearchRequestResponse>
   <return type="ProductInfo">
    <TotalResults type="int">4</TotalResults>
    <DetailsArray arrayType="Details[4]" type="Array">
     <Details type="Details">
      <Title type="string">Object-Oriented Database Management</Title>
      <Authors arrayType="string[2]" type="Array">
       <Author type="string">Alfons Kemper</Author>
       <Author type="string">Guido Moerkotte</Author>
      </Authors>
     </Details>
     <!-- ...three more Details elements... -->
    </DetailsArray>
   </return>
  </AuthorSearchRequestResponse>
 </Body>
</Envelope>
```

Figure 5.3: Example SOAP Response from Book Store Light

```
<message name="AuthorSearchRequest">
  <part name="AuthorSearchRequest" type="AuthorRequest" />
</message>
<message name="AuthorSearchResponse">
  <part name="return" type="ProductInfo" />
</message>

<portType name="BookStoreLightPort">
  <operation name="AuthorSearchRequest">
    <input message="AuthorSearchRequest" />
    <output message="AuthorSearchResponse" />
  </operation>
</portType>
```

Figure 5.4: Messages and Port Types (Book Store Light)

method (as long as the cache understands the encoding), we use the standard serialization throughout this chapter.

As stated in Section 4.1.3, there are four message exchange patterns defined within the WSDL specification: one-way, request/response, solicit/response, and notification. Our SSPLC handles the most commonly used request/response message exchange pattern. In fact, this is also the only message exchange pattern qualifying for caching. Such an operation expects one message as input and generates one output message. Our prototype implementation currently supports the SOAP 1.1 binding defined in the WSDL 1.1 specification which is the most commonly used binding today.[2] Of course, the prototype can be enhanced to support other bindings.

We will now present parts of the WSDL document of our Book Store Light service. Since SSPLC is currently mainly based on annotations at the abstract level we will focus on this level. Figure 5.4 shows a fragment of a WSDL document defining the port type of the Book Store Light service (`BookStoreLightPort`) having one operation (`AuthorSearchRequest`). This operation expects an `AuthorSearchRequest` message as input and produces an `AuthorSearchResponse` message as an output document. These messages are defined just above the `portType` element. Messages are composed of several `part` elements. As shown in the figure, the request message has one part of type `AuthorRequest` and the response message has one part of type `ProductInfo`. These types are defined using XML Schema [Fal01] in another fragment of the WSDL document, shown in Figure 5.5. An element of type `AuthorRequest` has the elements `author` and `levelOfDetail`, both of type `string`, in its content. In our example, `levelOfDetail` can be "heavy" or "lite" and influences the level of detail of the result. Figure 5.2 shows an example SOAP message requesting the most important information about books written by "Alfons Kemper".

An element of type `ProductInfo` contains the two subelements `TotalResults` and `DetailsArray`. The former is of type `int`, whereas `DetailsArray` is, in short, an array of

[2] An example for a SOAP 1.1 binding is presented in Section 5.4.

```
<types>
 <schema>
  <complexType name="AuthorRequest">
   <all>
    <element name="author" type="string" />
    <element name="levelOfDetail" type="string" />
   </all>
  </complexType>
  <complexType name="ProductInfo">
   <all>
    <element name="TotalResults" type="int" />
    <element name="DetailsArray" type="DetailsArray" />
   </all>
  </complexType>
  <complexType name="DetailsArray">
   <complexContent>
    <restriction base="Array">
     <attribute ref="arrayType" arrayType="Details[]" />
    </restriction>
   </complexContent>
  </complexType>
  <complexType name="Details">
   <all>
    <element name="Asin" type="string" />
    <element name="Title" type="string" />
    <element name="Authors" type="AuthorArray" />
   </all>
  </complexType>
  <complexType name="AuthorArray">
   <complexContent>
    <restriction base="Array">
     <attribute ref="arrayType" arrayType="string[]" />
    </restriction>
   </complexContent>
  </complexType>
 </schema>
</types>
```

Figure 5.5: Type Definitions (Book Store Light)

`Details` elements. `Details` is another type defined inside the WSDL document, having the three subelements `Asin`, `Title`, and `Authors`. The first two subelements are of type `string`, the last one is of type `AuthorArray` which is an array of `string`s representing the authors of the book. For our example, we assume that `Asin` is only present in a result if `levelOfDetail` was "heavy". Figure 5.3 shows an example SOAP response corresponding to the request shown in Figure 5.2. Only one of the books (i.e., `Details` elements) is shown.

5.3 Basics of the Web Service Cache SSPLC

The SSPLC features protocol level semantic caching, not application level caching. Thus, the SSPLC a priori has no implicit knowledge about the applications, i.e., Web services. It is therefore necessary to instruct the cache what to cache, how to cache, and how long to cache. In our approach this information is specified by the provider of a service (see Section 5.4). Of course, protocol level caching in general cannot be as efficient as an application level cache, but added generic usability and good applicability to a wide range of existing Web services is compensating for that.

We will now discuss our design decisions on caching aspects like replacement policy and cache consistency strategy. These concerns are not the main focus of our work so we used existing solutions as far as possible and adapted existing work where necessary.

5.3.1 Replacement Policy

Since cache memory is a limited resource, the cache may have to discard some regions to free memory for new regions. There are several well known replacement strategies available, e.g., FIFO (First In First Out), LRU (Least Recently Used), LRU-K [OOW93], and a low overhead approximation to LRU-2 called 2Q [JS94]. After experimenting with FIFO and 2Q, we decided to implement our own modified version of the 2Q strategy. Empirically, standard 2Q is a smart choice because of good replacement decisions and low CPU overhead, but this algorithm is designed to handle objects of uniform size.

The 2Q strategy (more precisely the *simplified 2Q strategy*) is based on two queues which share the cache memory. The first queue (A_1) is organized using a FIFO strategy. Every object which is requested for the first time is inserted into this queue. If an object is requested for a second time while it is still contained in A_1, the object is considered a hot spot and is moved to the other queue A_m which is organized using the LRU strategy. Every time an object contained in A_m is requested, the corresponding entry is moved to the top of the queue. Objects reaching the tail of A_1 or A_m are removed if memory is required for new objects. Which queue is selected for deletion depends on a tunable threshold for the size of A_1. In our implementation, A_1 and A_m are of the same size.

As semantic regions can be of different size, it is obvious that purging a region from the cache should not only depend on its usage but also on its size. Thus, we introduce a simple but efficient *cost-to-size ratio*. This is done by dividing the queues into slots. Thus,

large regions allocate multiple slots of the queues A_1 or A_m. Of course, every region exists only once but is referenced from multiple slots. How many slots a region r uses is defined by the tuning parameter *slot_size*: slots(r) = $\lceil$size(r)/slot_size$\rceil$.

Now, a large region r must be requested for slots(r) times before it is completely moved from A_1 to A_m. The queue A_m is still organized using LRU. After r has been completely moved to A_m, every time r is requested, the lowest slot of r contained in A_m is moved to the top of A_m. If one of the slots allocated by r reaches the bottom of A_1 or A_m, r is purged if memory is required, as described above. Thus, the larger a region r is, the more often it has to be requested to preserve it from being purged.

5.3.2 Distribution Control and Cache Consistency

The replacement strategy determines which data to cache for how long. There are also other cache consistency and legal issues affecting this decision [Ber02]. The SSPLC is transparent to users and other Web services. Thus, for clients responses from the cache look like responses generated by the origin Web service itself. Since Web services are often used in business environments, ambiguity about who is liable for a response is not tolerable. There are a lot of other problems related to caches, e.g., is it allowed to cache the response of a pay-per-use service, or is it allowed to cache responses from a service at all? Therefore, SSPLC gives providers exclusive control over distribution and cache consistency using a SOAP header extension. As long as there is no consistency information, SSPLC won't cache a response as postulated by [Not01] and [Ber02]. Web services can also explicitly forbid caching.

There are several techniques described in the literature offering weak or strong cache consistency guarantees. The most commonly used weak consistency techniques are client-driven and easy to handle: time-to-live (TTL) and expiry-time [Cz02]. Strong consistency techniques, e.g., server invalidation or lease-based techniques [CAL+02, Cz02, INST02, NKS+02], are typically server-driven and are more complex. As already mentioned, these techniques can only be applied in a reasonable way if SSPLC is used as a reverse-proxy or an edge cache. Using these techniques, SSPLC can even handle highly dynamic Web services. Since cache consistency mechanisms are not the focus of this work, we assume service-specific TTL in the following discussion.

If a provider allows caching, it must explicitly state some cache consistency information. For example, the following `CacheControlHeader` element allows caching of the message and states that the response is *fresh* for at least the given duration (12 hours). After this duration, the cached version of the response must be removed from the cache.

```
<CacheControlHeader>
  <CacheConsistency>
    <TTL>P0Y0M0DT12H00M00S</TTL>
  </CacheConsistency>
</CacheControlHeader>
```

5.3.3 Physical Storage of Semantic Regions

Using a cache requires a large amount of memory to be able to serve lots of clients based on a reasonably large number of semantic regions. Since disks are considerably larger and cheaper than main memory, it is obviously a good idea to use them for the storage of semantic regions. The simplest way is to keep only region descriptors in main memory and to store region content on disk, e.g., by using a DBMS (a rudimentary one should be sufficient) or a flat file system. Preferably, the replacement strategy should be aware of disks in order to distinguish between "purge a semantic region" and "move a semantic region to disk", thus allowing "hot" regions to be kept in main memory while storing "cooler" regions on disk.

Since it is orthogonal to the issues discussed in this work whether the cache is based on main memory, disk, or both, we assume for the rest of the chapter that the cache is only based on main memory. Our prototype system is main memory-based as well.

5.4 Semantic Caching in the Web Service Cache SSPLC

Basically, semantic caching in SSPLC is done by annotating WSDL documents with information about the caching-relevant semantics of services using the language presented in the next section. This information is used for mapping SOAP requests to predicates, for fragmenting responses, and for reassembling responses. Thus, adapted semantic caching algorithms can be applied.

5.4.1 WSDL Annotations

Our language is designed both to cover common capabilities of existing Web service interfaces and to preserve efficient solvability of the *query containment problem* [GSW96, Ull89], which is intrinsic to the semantic caching approach.

5.4.1.1 Fragmentation and Reassembling

Since Web services deliver monolithic XML documents rather than tuple-oriented responses, SSPLC needs some information about how to fragment such documents to obtain fine-granular response units comparable to tuples in database caching. These units are called *fragments*. We use an XPath-expression [CD99] to specify the fragmentation. Additionally, SSPLC needs further instructions regarding the generation of a complete response document based on fragments of prior requests. This information is specified using the XQuery language [BCF+03]. Both the XPath-expression and the XQuery, are provided using an additional element (`OperationCacheControl`) inside the `binding` element of the WSDL document of a service because it depends on the actual coding of the messages.

```
<binding name="BSLBinding" type="BookStoreLightPort">
 <binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
 <operation name="AuthorSearchRequest">
   <operation soapAction="BookStoreLight" />
   <input>
     <!-- ...describes how the input message is mapped to XML... -->
   </input>
   <output>
     <!-- ...describes how the output message is mapped to XML... -->
   </output>
   <OperationCacheControl>
    <fragmentationXPath>
     /Envelope/Body/AuthorSearchRequestResponse/return/DetailsArray/Details
    </fragmentationXPath>
    <reassemblingXQuery>
     <!CDATA[
      let $details := ##RESULT_FRAGMENTS##
      return
      <Envelope encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
       <Body>
        <AuthorSearchRequestResponse>
         <return type="ProductInfo">
          <TotalResults type="int">##COUNT_RESULT_FRAGMENTS##</TotalResults>
          <DetailsArray arrayType="Details[##COUNT_RESULT_FRAGMENTS##]"
                        type="Array">
           {$details}
          </DetailsArray>
         </return>
        </AuthorSearchRequestResponse>
       </Body>
      </Envelope>
     ]]>
    </reassemblingXQuery>
   </OperationCacheControl>
 </operation>
</binding>
```

Figure 5.6: Annotation of the AuthorSearchRequest Operation

Figure 5.6 gives an example for our Book Store Light. The marked region depicts the annotated information for the SSPLC while the rest of the document constitutes a standard SOAP binding. Referring to our book store example, we are interested in the individual books, i.e., `Details` elements, contained in a response document of our example service. The XPath-expression shown inside the `fragmentationXPath` element in Figure 5.6 can be used to fragment a response document accordingly. This XPath-expression can be figured out by examination of the type definition part of the service's WSDL document (see Figure 5.5) and of an example response of the service (see Figure 5.3). The XQuery to reassemble a response is shown in the figure inside the `reassemblingXQuery` element. The macros `##RESULT_FRAGMENTS##` and `##COUNT_RESULT_FRAGMENTS##` are expanded by the SSPLC before evaluating the XQuery and represent exactly the fragments (respectively their number) which should be reassembled to a complete response document. Since an introduction to XQuery lies outside the scope of this work, we will not explain the XQuery shown in the figure. It should be obvious that the result of the XQuery is a SOAP response like the one shown in Figure 5.3.

5.4.1.2 Predicate Mapping

We need predicates in region descriptors to describe the fragments stored in the region contents. Thus, we need some information about the semantics of requests. Moreover, we want to be able to filter semantic regions, e.g., if we are looking for all books written by "Alfons Kemper" in a region storing all books written by "Kemper". Therefore, we need to know how to access the individual "attributes" (elements) of a tuple (fragment). This information is annotated to the type definitions of requests in WSDL documents.

We will explain the annotations using our Book Store Light example. The original type definition of `AuthorRequest`, which is the request type of our service, is shown in Figure 5.5. Currently, we assume that if there are several parameters defined in a request, i.e., `levelOfDetail` and `author`, they are combined by an AND operator. Thus, the request shown in Figure 5.2 means that we are looking for all books written by "Alfons Kemper" and we are only interested in the most important facts of the books. Additionally, we assume that if there are several elements inside an array, the elements are logically ANDed together, too. This is also true for responses (see the `Author` elements inside the `Authors` element shown in Figure 5.3). The annotated version of the `AuthorRequest` type is shown in Figure 5.7.

We annotate every parameter of the request using one or more `CacheControl` elements. It is necessary to specify some context information because a message can be used for several operations having different semantics. Also, if another binding is used, the coding of the message might be different, requiring some modifications inside the `CacheControl` element. Thus, the context information given by the attributes of `CacheControl` defines when to use the information inside the `CacheControl` element. The information shown in Figure 5.7 can only be used to analyze an input message for the `AuthorSearchRequest` operation using the `BSLBinding`. A `StringParameter` element defines that the parameter is of type string. The content of this element gives more detailed information about how

```
<complexType name="AuthorRequest">
 <all>
  <element name="author" type="string">
   <annotation>
    <appinfo>
     <CacheControl context="AuthorSearchRequest"
                   bindingContext="BSLBinding">
      <StringParameter>
       <required>true</required>
       <fragmentXPath>
        Authors/Author/text()
       </fragmentXPath>
       <implicitOperator>contains_wwo</implicitOperator>
       <caseSensitive>false</caseSensitive>
       <operators>
        <and> </and>
        <and>,</and>
       </operators>
      </StringParameter>
     </CacheControl>
    </appinfo>
   </annotation>
  </element>
  <element name="levelOfDetail" type="string">
   <annotation>
    <appinfo>
     <CacheControl context="AuthorSearchRequest"
                   bindingContext="BSLBinding">
      <StringParameter>
       <required>true</required>
       <implicitOperator>equals</implicitOperator>
       <caseSensitive>true</caseSensitive>
      </StringParameter>
     </CacheControl>
    </appinfo>
   </annotation>
  </element>
 </all>
</complexType>
```

Figure 5.7: Annotated WSDL Type Definition

to handle this string parameter. We also defined elements for other parameter types, e.g., an `IntegerParameter` element. Each of these elements contains further information (e.g., operators) depending on the parameter type.

Looking at the example in Figure 5.7, we observe that the author parameter is mandatory (`required` element). If a parameter is optional, a default value of the parameter that is used in case of absence of the parameter in a request must be specified using a `default` element (not available in the example document). The `fragmentXPath` element specifies how to extract the information from result fragments that correspond to this parameter (compare Figure 5.3). For example, if we ask for books written by an author, the `fragmentXPath` can be used to find the authors in the result fragments. If, as in our example, an XPath is specified, the cache can inspect the fragments to look up the actual author(s) of a book. This information can be used to filter all fragments contained in a semantic region. If there is no XPath specified, the cache is not able to do such filtering because it is constrained to the information obtained from the request.

The element `implicitOperator` defines the operator of the parameter. Currently, we support the following operators (for appropriate parameter types): $>$, $\geq$, $<$, $\leq$, $=$ (or *equals*), *contains*, *contains_wwo*, *starts_with*, and *ends_with*. In our example, the operator is *contains_wwo* which is a contains operator that looks for "whole word only" occurrences of the given pattern in a string, i.e., "Alfons Kemper" does not contain_wwo "Kemp", but contains_wwo "Kemper". The comparison of strings is case insensitive as defined by the `caseSensitive` element.

Additionally, we support the logical operators AND and OR to support complex predicates. We also support parentheses for precedence control. Currently, we are not supporting the $\neg$ operator (logical NOT operator)[3] because there are virtually no Web services offering this operator and we are interested in keeping the query containment problem efficiently solvable. The `operators` element in Figure 5.7 defines two AND operators for the author parameter: a space character and a comma.

The second parameter is `levelOfDetail`. This is also a mandatory string parameter. The implicit operator is a case sensitive "equals". There is no `fragmentXPath` defined because in the response document of our Web service no explicit information about whether it is a "heavy" or a "lite" result is contained. As this information is contained in the request and therefore is stored as part of the region predicate, this information is not lost.

Using these annotations our SSPLC can figure out the semantics of a request and is able to extract interesting elements from fragments. Also, it is able to generate a region predicate from a request. For example, the request shown in Figure 5.2 is mapped to the following predicate:

$$\begin{gathered}\text{author contains\_wwo\_case\_insensitive ``Alfons''} \wedge \\ \text{author contains\_wwo\_case\_insensitive ``Kemper''} \wedge \\ \text{levelOfDetail equals\_case\_sensitive ``lite''}\end{gathered}$$

[3] and hence we are not supporting $\neq$, $\neg$*contains*,...

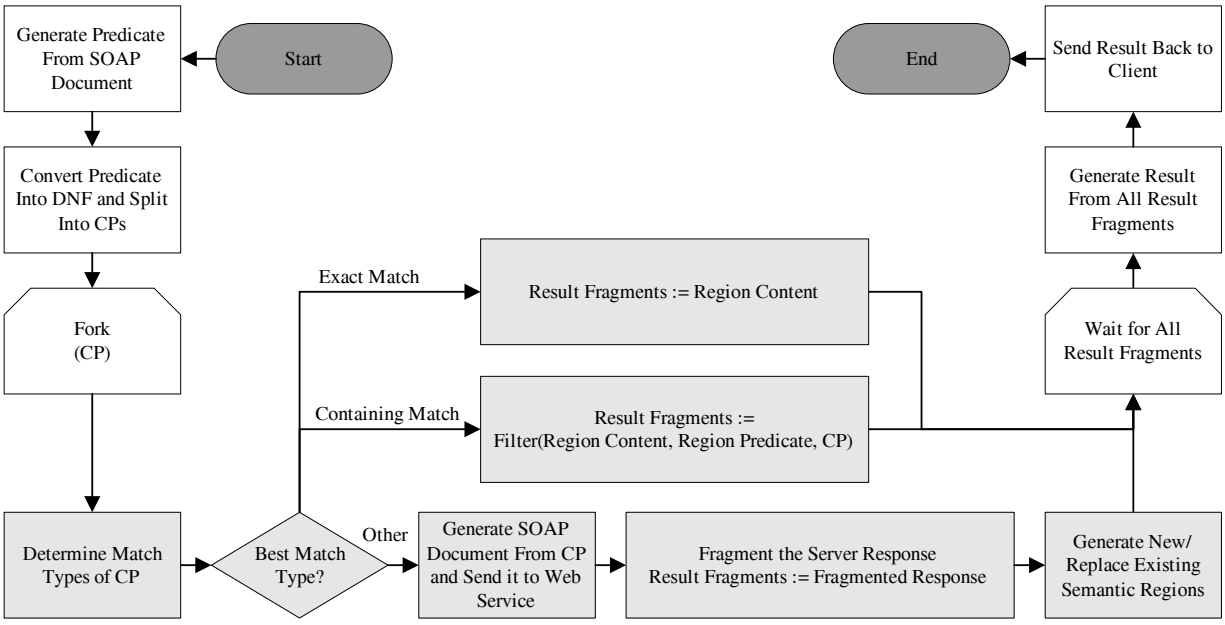


Figure 5.8: Flow Chart of the Caching Process

5.4.2 Matching and Control Flow

Using our annotations we are now able to understand the caching-relevant semantics of requests and responses. We will now describe how this information is used for caching. The control flow of our SSPLC is shown in Figure 5.8. First of all, a SOAP request R is mapped to a predicate P as described above. Although the Book Store Light does not offer a logical OR operator for the author parameter, we will use the following predicate P (operator names are shortened) for demonstration purposes throughout this section:

(author contains “Kemper” $\vee$ author contains “Moerkotte”) $\wedge$ levelOfDetail = “lite”

After the mapping, P is transformed into *disjunctive normal form* (DNF) and split into *conjunctive predicates* (CPs), i.e., predicates only containing simple predicates connected by logical AND operators. If there is no logical OR in a request, P is processed as is. The transformation of our example predicate P results in:

CP_1: author contains “Kemper” $\wedge$ levelOfDetail = “lite”
CP_2: author contains “Moerkotte” $\wedge$ levelOfDetail = “lite”

For every CP, the light gray actions shown in Figure 5.8 are executed in parallel. First, match types of a CP with all semantic regions are determined, i.e., the correlation between every semantic region S and the result of CP is determined. There are five different match types (compare [CB00, LC01]) as shown in Figure 5.9. The best match type for a CP and a semantic region S is, of course, the exact match. The next best match type is a containing match because we only have to filter S by eliminating all fragments fulfilling the region predicate but not CP to get the fragments for the response. The other three match

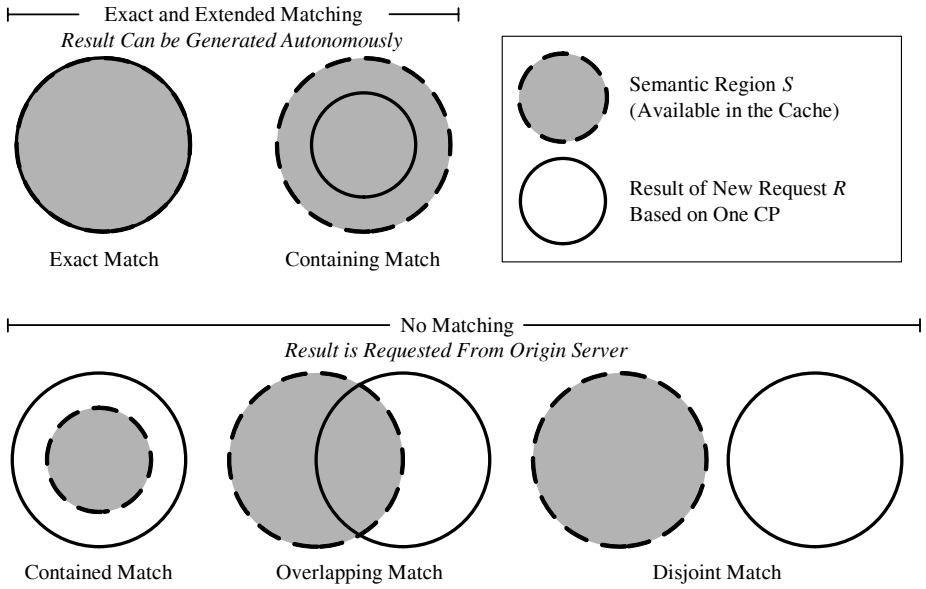


Figure 5.9: Match Types

types require server interaction because we do not have all fragments cached to answer the request. Since most Web services do not have adequate interfaces to be able to process complicated remainder requests, we are sending a request generated from the CP to the Web service even though there already might be some relevant fragments available in the cache. Even if a Web service can process complicated remainder requests, processing of such complex requests is likely to be costly. As one of the goals of SSPLC is to reduce processing demands of the central servers, usage of complex remainder requests could be counterproductive.

The response of the Web service is fragmented and afterwards stored in the cache. If there are already regions in the cache containing some of the fragments of the response (i.e., in the case of a contained match), these semantic regions are replaced with the new (larger) semantic region. In all other cases, the fragmented response is inserted as a new semantic region using CP as the region predicate.

After all CPs have been processed, SSPLC calculates the result of P as the union of the results of all CPs. By default, duplicates are eliminated, i.e., SSPLC implements the very common set semantics. Alternatively, SSPLC calculates the result without duplicate elimination. This behavior is controlled by an optional `distinct` element inside the `OperationCacheControl` element (not shown in the example document). The default behavior is to eliminate duplicates. Fragments are considered equal if their contents are equal or if keys are defined, their keys are equal. Keys can be defined vi a `key` element inside the `OperationCacheControl` element using the standard XML Schema syntax for keys. Usage of keys considerably speeds up duplicate elimination. We do not further investigate keys in the scope of this work. The result of P is (conceptually) written to an XML document D. After that, the `reassemblingXQuery` is evaluated with the macro `##RESULT_FRAGMENTS##` expanded to D. Finally, the response is sent back to the client.

We implemented some optimizations in our SSPLC, e.g., if P does not contain an OR operator, there is only one CP to be processed. If there is no matching region for CP, we send a request to the server. After that, we temporarily store its response and send it back to the client instead of regenerating it from the fragmented response.

5.4.3 Sorting and Generalization

Since the order of elements can be important in XML documents, we enhanced our SSPLC to be aware of it. XML documents are inherently ordered by the sequence of the elements (*document order*). As long as the document order generated by a Web service offers no real added value (e.g., lexicographical order by title), it does not matter in which order the fragments emerge in the response. Also, as long as we are using fragments of only one semantic region (filtered or not), order is abided and we can generate correctly ordered results as in the Book Store Light example.

If a Web service orders fragments using some information available in the response, there are two possibilities to establish the same order even if we are merging fragments of several semantic regions to generate the response. First, if the order is fixed, i.e., always the same, the `reassemblingXQuery` can be modified to do the sorting using the *order by* clause of XQuery. Second, if the order depends on a request parameter, we can annotate this parameter using a `SortParameter` element. This element contains a mapping from the service's sorting facilities to order by clauses of XQuery. For example, if a Web service has a parameter `sort` and the value "`+title`" means "sort by title", a mapping to XQuery could look like "`order by $fragment/Title`". The appropriate order by clause is inserted into `reassemblingXQuery` before evaluation.

The value of a sorting parameter is stored in the region descriptor because it is relevant for determining the match types. An exact or containing match is only usable if either the sorting is already as it should be or we are able to establish the correct order using an order by clause. If the response of the Web service does not contain an element that can be used to reestablish the sorting of fragments, the SSPLC has to send a SOAP document generated from the region predicate to the Web service for the correct order when fragments from more than one semantic region are needed for the response. If only fragments from one semantic region are needed, it must already have the required order, otherwise the service must be contacted as well. To avoid several semantic regions containing the same fragments in different order, SSPLC stores sorting vectors inside semantic regions to remember alternative sortings for future use.

Another enhancement of our semantic caching scheme is the usage of generalization for better decisions on the query containment/predicate subsumption problem. Our SSPLC supports two different types of generalization. First, tree-structured containment relations for values of parameters can be defined. For example, if there is a parameter defining whether we are interested in paperback, hardcover, or both, we are able to annotate this parameter to point out that "hardcover $\subseteq$ both" and "paperback $\subseteq$ both". This information is used during match type computation and for filtering of semantic regions.

The second type of generalization can be seen in our Book Store Light example. There is a parameter `levelOfDetail` that influences the level of detail of the response. Since "heavy" fragments simply contain some extra elements, it is possible to define an XQuery filter to transform "heavy fragments" to "lite fragments" by removing the surplus elements like the `Asin` elements in our example. This information is also used during match type computation and region filtering.

5.5 Performance Evaluation

We implemented a prototype of the Web Service Cache SSPLC for our service platform ServiceGlobe using Java. We conducted several performance experiments based on the scenarios of the TPC-W [TPPC02] and TPC-W Version 2 [TPPC03] benchmarks.

5.5.1 Benchmark Scenario 1 (TPC-W)

The first scenario is related to the online bookstore scenario of the TPC Web commerce benchmark (TPC-W). Because TPC-W does not aim at SOAP Web services and semantic caching, but instead at traditional Web servers and back-end servers, major modifications to TPC-W (system architecture as well as data generation) are necessary to adjust the benchmark to the context of our SSPLC in a reasonable way. Thus, we decided to model our benchmark scenario on the SOAP interface of Amazon, just as the scenario of TPC-W is modeled on the HTTP interface of Amazon. We chose to use Amazon's author search request for our benchmarks because this search functionality is also addressed in the TPC-W benchmark.

5.5.1.1 Experimental Setup

Our benchmark environment for the first scenario consists of two standard off-the-shelf computers with an Intel Pentium 4 CPU, 2.8 GHz, 1 GB main memory, and 100 MBit/s network adapter each. The operating system is Red Hat 9 and we use Java 2 SDK, Version 1.4.1 from Sun Microsystems. One of the computers is used for the simulation Web service (see below), the other one for the cache and the benchmark engine. We use the QIZX/open Version 0.2_01 [Fra] as XQuery processor and a recent version of Xalan [Xal] as XPath processor. A recent version of Xerces [Xer] is used as XML parser and Axis [Axi] is used as implementation of SOAP.

To show the effectiveness of our semantic cache, we implemented a simulation service rather than using Amazon directly because Amazon delivers its results page-wise (i.e., 10 books per SOAP response), which is an unusual behavior for Web services. The requests and responses of our simulation service are identical to those of the Amazon service despite the fact that our service delivers all results to a request in one response. For that purpose, we materialized some of the data of Amazon to be able to work with real data. Since our simulation service delivers these materialized results extremely fast, we are delaying results

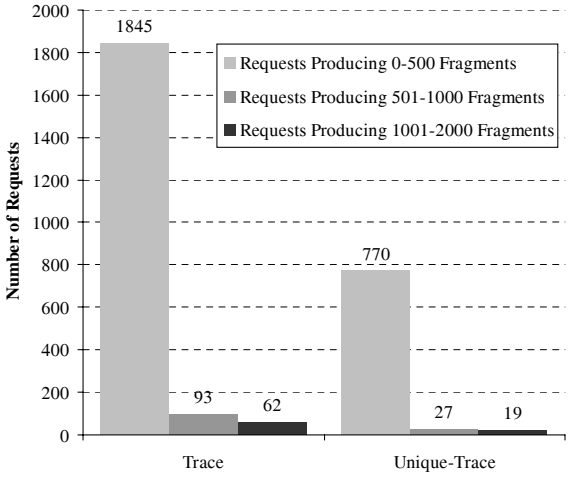


Figure 5.10: Request Distribution

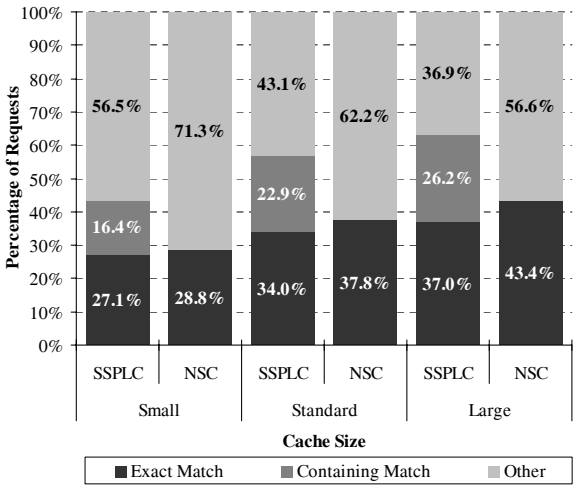


Figure 5.11: Match Distribution Varying Cache Size

to simulate processing time of a Web service. We conducted some experiments to assure that SSPLC is able to deliver its results as fast or faster on average than the origin Web service. Since these results depend heavily on the performance of the origin server and of the machine running SSPLC, we do not present quantitative results.

Our benchmark scenario is based on several top-300 bestseller lists (top selling science books, top selling sports books, ...) of Amazon. We used these different bestseller lists to generate different traces as described below and we always present the average of all performance experiments conducted using these different traces. If an author's book is present on the bestseller list, people will be interested in other books published by the same author, too. Thus, an author search request is more likely for authors whose books are ranked high on the bestseller list. Since studies [AH02] have found that access to data on the Internet often follows a Zipf distribution, we use a Zipf distribution ($\theta = 0.75$) on the top-300 bestseller lists to select books.[4] Using the names of the authors of a book, we generate a request for our simulation service. We randomly choose which names (surnames, first names) are used for the request. Every request contains at least one surname of an author. This is done to challenge semantic caching. We generated traces of 2000 requests each for the performance experiments.

Some of the requests produce very large response documents containing up to 32000 fragments. Since the size of such documents is about 40 MB, it is very likely that Web services do not generate such large responses. Rather, they generate a fault response informing the caller that there are too many results and that the request has to be refined. Thus, our simulation service sends fault messages for results containing more than 2000 fragments. SSPLC caches these fault messages because they are marked cachable in the SOAP header.

[4]We conducted a second series of experiments using a Zipf distribution with $\theta = 0.86$. These experiments confirm the results and correlations presented in this section.

Figure 5.10 shows the average distribution of the requests of our traces. The majority of the requests produces responses containing up to 500 fragments (or fault messages as described above). The term *unique-trace* refers to a trace where all duplicates are removed. The figure shows that there are, e.g., only 19 unique requests producing responses containing between 1001 and 2000 fragments. So, each of these requests is only contained about 3 times in the full trace. Thus, for our Book Store Light, caching of large responses is not very promising because they are requested very infrequently.

We conducted several performance experiments varying different parameters and we present the results in this section. For the experiments in this section, the TTL of responses was set to 30 minutes, if not explicitly stated differently. The data volume of all responses materialized by our simulation service was about 85.4 MB. Every execution of a trace lasted for about 100 minutes. We conducted some longer running experiments which demonstrated similar results. The maximum size for responses to be cached was set to about 1000 fragments (1.2 MB). Larger responses were fetched from the remote Web service and forwarded to the client without caching. One slot (see Section 5.3.1) had the size of about 1/4 of an average response, i.e., 40 KB. We conducted the experiments using three different cache sizes: small (10% of the data volume of the unique-trace), standard (20%), and large (30%). The cache was warmed up by running every trace twice and measuring the second one, although there are only minor differences between the two runs.

5.5.1.2 Experimental Results

The main goal of the SSPLC is to improve scalability of Web services. Figure 5.11 shows[5] that already the smallest semantic cache is able to answer 43.5% (exact matches + containing matches) of all requests using data stored in the cache, reducing processing demands on the central servers significantly. A traditional (non-semantic) cache (NSC) achieves much smaller hit rates (28.8%). The bigger the caches are, the better the hit rates become even though the increase rate is not linear with the cache size increment. This is due to the fact that already the standard cache size is large enough to cache most of the hot spot responses. The only advantage of a larger cache is that it is able to additionally store some of the less frequently requested responses. SSPLC benefits more from a larger cache than NSC because SSPLC can exploit the semantics of the requests.

Figure 5.12 demonstrates the reduction of bandwidth consumption. Running the trace without cache results in the transfer of 298 MB across the network. The smallest semantic cache reduces the transfer volume by approximately 28%, the standard semantic cache by approximately 41%. The large semantic cache reduces the transfer volume even more, but the difference is not linear with the cache size increment due again to the reasons above. On average, the transfer volume of NSC is more than 12% larger than that of SSPLC.

Figure 5.13 shows results for varying time-to-live periods. Of course, the longer the TTL period is, the more effective the caches are. Depending on the TTL, SSPLC performs about 43% to 50% better than NSC.

[5]Please note that the sum of the rates of exact matches, containing matches, and other matches is not always exactly 100% due to rounding errors.

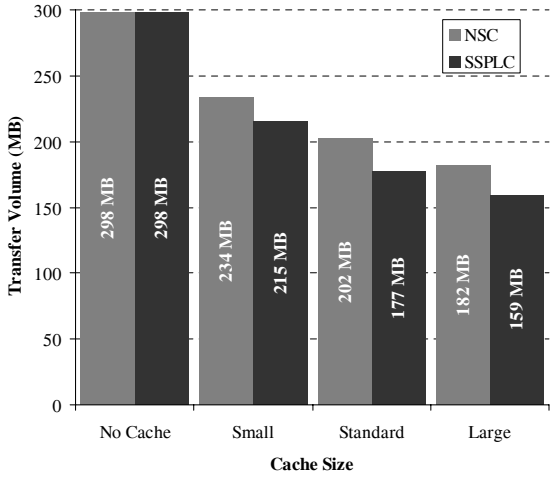


Figure 5.12: Transfer Volume Varying Cache Size

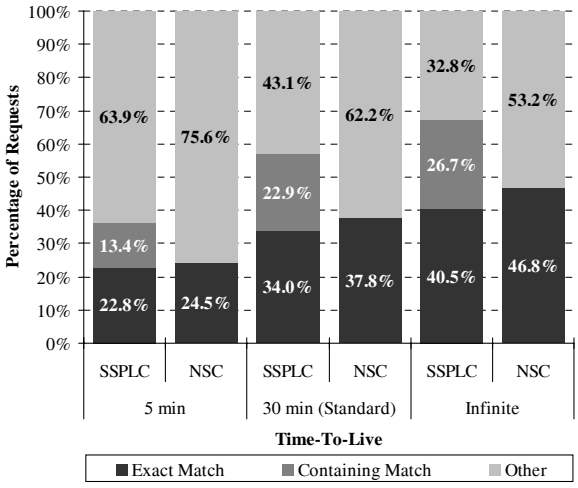


Figure 5.13: Match Distribution Varying TTL

Figure 5.14 shows the influence of the slot size parameter (see Section 5.3.1) of the replacement strategy on the hit rates of the caches. Smaller slot sizes lead to better hit rates because only minimal cache memory is wasted by partly used slots (clippings). If the slot size increases, the amount of wasted cache memory increases as well, leading to comparably slightly lower hit rates of both SSPLC and NSC. The disadvantages of smaller slot sizes are higher administrative costs. Thus, we decided to use a medium slot size for our performance experiments.

The results for varying maximum cached response size is shown in Figure 5.15. Of course, these results depend on the access trace of the cache. The average distribution of the requests of our traces (see Figure 5.10) shows that large responses are accessed very infrequently. Thus, the caching of large responses is not beneficial in the presented scenario. Both NSC and SSPLC suffer equally from larger maximum cached response sizes.

Figures 5.16 and 5.17 show the results for varying value of the theta (θ) parameter of the Zipf distribution for the SSPLC respectively for a non-semantic cache (NSC). The larger the value of theta is, the more distinct the hot spots of a trace are. Thus, if $\theta = 0.99999$ (this value is rounded to 1.0 in the figure), there are few distinct hot spot authors that are accessed very frequently. If $\theta = 0.0$, there are no real hot spot authors. The figures show that SSPLC is always superior to a non-semantic cache by far. Additionally, Figure 5.16 demonstrates that SSPLC works very well even for small values of θ in this scenario.

5.5.2 Benchmark Scenario 2 (TPC-W 2)

The Transaction Processing Performance Council quite recently published a first draft of TPC-W Version 2 (TPC-W 2) for public review. This new version of TPC-W is aiming at Web services. Thus, we decided to conduct some additional performance experiments based on TPC-W 2. Due to incomplete specifications and time constraints, we did not implement the full benchmark. Rather, we chose the "product detail Web service inter-

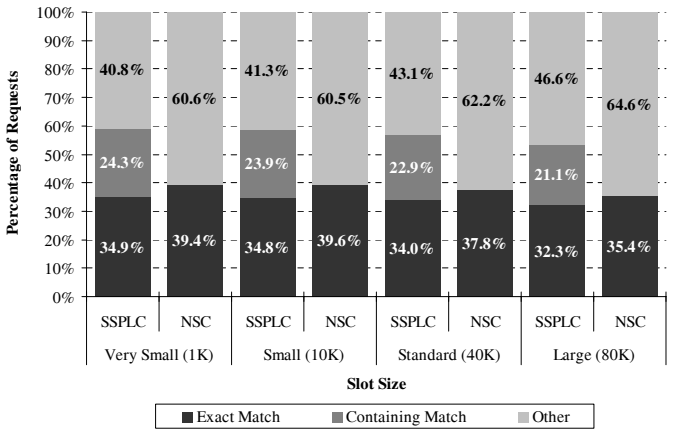


Figure 5.14: Match Distribution Varying Slot Size

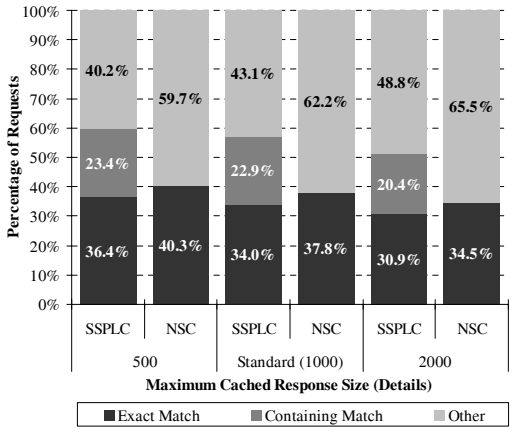


Figure 5.15: Match Distribution Varying Maximum Cached Response Size

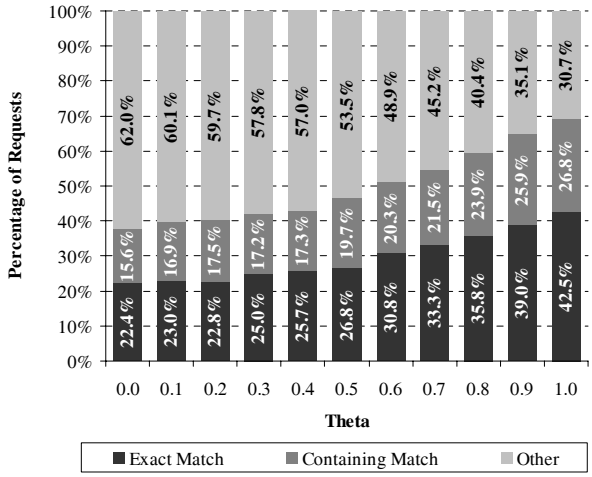


Figure 5.16: Match Distribution of SSPLC Varying Theta

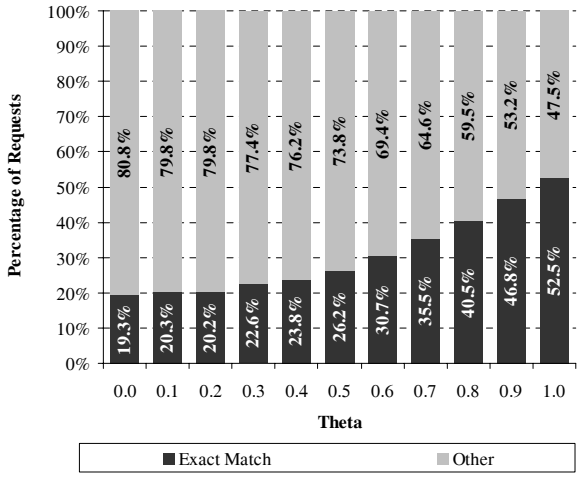


Figure 5.17: Match Distribution of NSC Varying Theta

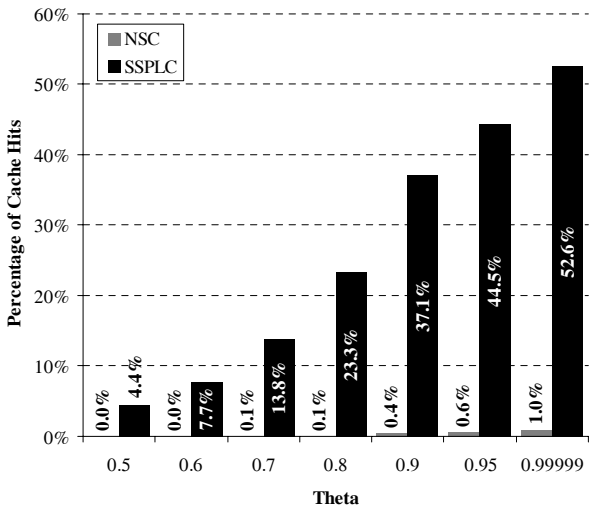


Figure 5.18: Cache Hits Varying Theta (TPC-W 2)

action" of TPC-W 2 to conduct our experiments. The data was generated conforming to the rules of TPC-W Version 2, i.e., 100000 books were generated and stored in the DBMS. We configured our remote business emulator (RBE) to run 8 emulated businesses (EB) concurrently. The TTL was set to 5 minutes[6] and a total of 3000 requests were sent to the SSPLC. The cache size was 4.6 MB, i.e., about 5% of the data volume available at the origin server could be cached. The slot size was set to the maximum size of one book, i.e., the cache was able to store about 2500 books. Every request asked for detailed information about a randomly chosen number (1 to 10) of books. According to the TPC-W 2 specifications, the books should be selected using a given non-uniform random distribution, but this distribution generates values which are distributed too uniformly for any cache. Again, we use a Zipf distribution to select the books.

If a client requests product details for, e.g., book 2 and book 8, SSPLC translates the request to the predicate "book = 2 $\vee$ book = 8". Thus, SSPLC splits up the request into two CPs, as described above, and generates a request for every single book if not available in the cache. For this reason, there are only exact matches and disjoint matches in this scenario. If not all books of a request are available in the cache, the SSPLC rates the request as exact match and disjoint match according to the ratio of books available in the cache to books not available in the cache. For example, if a client requests details about eight books and six books are available in the cache, the request is rated as 0.75 exact match and 0.25 disjoint match.

Figure 5.18 shows the exact matches for the benchmark varying theta of the Zipf distribution. A non-semantic cache (NSC) is virtually useless in this scenario because the cache hits are less than 1%, even if $\theta = 0.99999$. This is because NSC can only answer requests from the cache if two requests are exactly the same, i.e., the number of product details requested must be the same, the books must be the same, and the order of the books must be the same. SSPLC works very well for sufficient large θ, even though the cache size is small (about 5% of the data volume available at the origin server) and the TTL is short. For a realistic θ, i.e., greater or equal to 0.8, the SSPLC is able to answer more than 23% of the requests.

5.6 Related Work

Caching in the context of Web services has been addressed, e.g., by Akamai [Not01] and by the usage scenarios S032 and S037 of the World Wide Web Consortium [HHO04]. The proposed approaches are either described very abstractly, or are limited to a more or less straightforward store-and-resend of SOAP responses. Our approach differs in that it takes advantage of the fact that query-style requests can be cached more efficiently using semantic caching. Thus, this chapter proposes an alternative solution which is more flexible and powerful.

[6] Every benchmark run lasted for about 20 minutes.

A solution for a similar but simpler problem in the area of Web sources and respectively Web databases, was presented by [LC99, LC01]. They focus on wrapper[7] level caching. Therefore, they are able to take advantage of the semantics of the declarative query language SQL, i.e., they automatically deduce region predicates from SQL queries. In the area of Web services, no such standardized declarative language exists. Due to our declarative language for the annotation of WSDL documents with information about caching-relevant semantics, we are able to apply semantic caching to Web services in, e.g., B2B and B2C scenarios. Additionally, we investigate sorting and generalization issues. Thus, our solution is more comprehensive and more flexible. The basic techniques of both SSPLC and [LC99, LC01] are based on prior work on semantic caching, e.g., [DFJ+96].

A different usage of caching for Web services is presented in [TR03]. They use caching techniques for reliable access to Web services from, e.g., PDAs or similar unreliably connected mobile devices. While connected to the Internet, the cache stores requests and associated results but does not answer them itself. In the case of a disconnection, the cache tries to answer the requests and additionally caches them. The cache plays the requests back to the origin server as soon as it is online again. The authors use one representative service to demonstrate the benefits of a Web service cache and expose a number of issues in caching Web services. They do not present a generic solution, but they do conclude that extensions to WSDL are needed to support cache managers. We think that the language presented in this chapter constitutes a good base for such extensions.

5.7 Status and Future Work

We presented the semantic cache SSPLC that is suitable for caching responses from Web services on the SOAP protocol level. We therefore introduced an XML-based declarative language to annotate WSDL documents with information about the semantics of service requests and responses. We demonstrated the validity of our proposed caching scheme by performing a set of experiments. The results of these experiments confirm the reduction of processing demands on the central servers and the diminishment of bandwidth consumption, as well as competitive average response time.

We currently conduct benchmarks to compare the performance of our semantic caching scheme without sorting or generalization against the performance with sorting and generalization enabled. We plan to investigate some ideas on how SSPLC can be further improved. First, the declarative language can be extended to integrate additional semantic knowledge like *fragment inclusion dependencies* [LC01] to be able to transform as many overlapping or contained matches as possible into exact or containing matches. Furthermore, we intend to improve our caching scheme by taking advantage of richer interfaces of services. Additionally, we plan to investigate techniques which enable Web services to load caches with relevant data [KFD00].

[7] Wrappers are used to extract data from Web sources.

Chapter 6

An Autonomic Computing Concept for Application Services

In this chapter, we present a novel autonomic computing concept which is hiding the ever increasing complexity of managing IT infrastructures. For this purpose, we virtualize, pool, and monitor hardware to provide a dynamic computing infrastructure. A fuzzy-logic-based controller module supervises all services running on this virtual platform. According to IBM's vision of autonomic computing, this infrastructure is a step towards a self-managing, self-optimizing, and self-healing virtual platform for services. Higher-level services such as business applications profit from running on this supervised virtual platform. For example, failed services are restarted automatically. A service overload is detected and remedied by either starting additional service instances or by moving the service to a more powerful server. The capabilities and constraints of the services and the hardware environment are specified in a declarative XML language. We used our prototype implementation of AutonomicGlobe for first tests managing a blade server configuration and for comprehensive simulation studies which demonstrate the effectiveness of our proposed autonomic computing concept.

This chapter is organized as follows: Section 6.1 motivates and introduces our autonomic computing concept. In Section 6.2 we present the architecture of AutonomicGlobe, which relies on our ServiceGlobe platform for location-independent execution of Web services. The basic concepts of fuzzy controllers are described in Section 6.3. A detailed description of the fuzzy controller integrated in AutonomicGlobe is presented in Section 6.4. Simulation study results follow in Section 6.5. After a discussion of related work in Section 6.6, the chapter is concluded in Section 6.7.

6.1 Motivation

Complexity and consequently administration costs of IT infrastructures are ever increasing. To overcome this, IBM coined the term *autonomic computing* [Hor01] for a concept that refers to some kind of self-management of hardware and software. Comprehensive

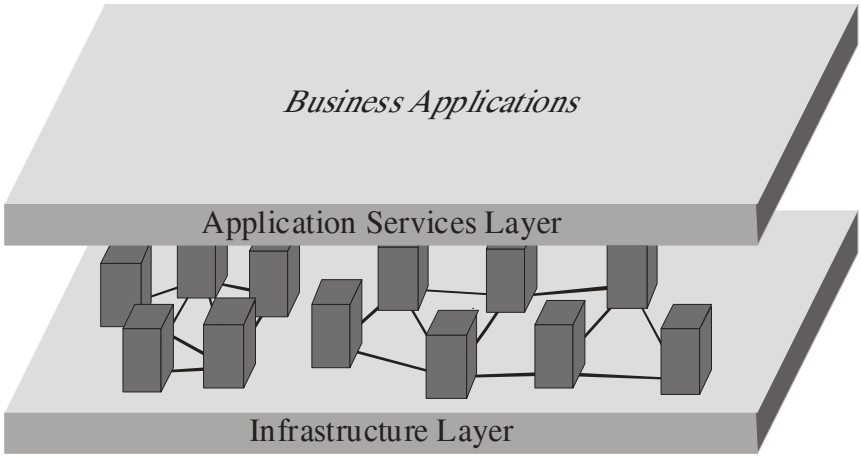


Figure 6.1: Architecture of the AutonomicGlobe Computing Concept

self-management capabilities for systems include self-configuration, self-optimization, self-healing, and self-protection. Several global players conduct research in this area and have already integrated some aspects of self-management into their hardware and software products.

We use a complex *enterprise resource planning* (ERP) environment as an example for our autonomic computing concept. Figure 6.1 shows the layered architecture of AutonomicGlobe. Application services such as business applications are conceptually running on the application services layer. The services are virtualized, i.e., not running on a fixed server. The self-management capabilities are provided by the infrastructure layer below. It virtualizes, pools, and monitors hardware to provide an adaptive and dynamic computing infrastructure which is supervised by a fuzzy-logic-based controller. This controller can, for example, automatically restart failed services. It detects overloaded service instances and remedies overload situations by either starting new service instances or by moving services to more powerful servers. Available resources are shared between all services as appropriate for a particular situation. Thus, by dynamically allocating the services, we improve the average utilization of the available hardware and minimize idle times. Thereby, total cost of ownership (TCO) is reduced either because more users can be handled using the existing hardware or because less hardware is required initially.

The capabilities and constraints of the application services and the hardware environment are described using a declarative XML language. Among other things, with this language the maximum and minimum number of instances of a service can be defined, the performance of hosts can be related to each other, and the rules for the fuzzy controller can be specified. As a decision finding component, a fuzzy controller is employed because of its robustness, adaptability, and intuitive specification of rules.

Although AutonomicGlobe is designed to run in arbitrary heterogeneous computing environments, we will use a blade server environment (see Section 6.2.5) for the presentation of our autonomic computing concept and for some performance experiments.

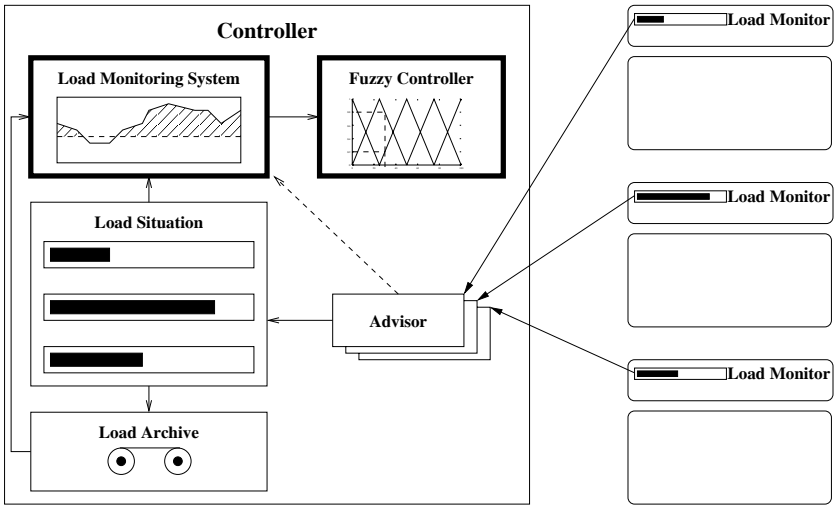


Figure 6.2: Architecture of the Controller Framework

6.2 Architecture of the Controller Framework

AutonomicGlobe is based on our distributed and open service platform ServiceGlobe (see Chapter 4). More precisely, it is based on the dispatcher service presented in Section 4.3. The ambition of the AutonomicGlobe project is to add an active control component for autonomic service and server management to ServiceGlobe. The architecture of our controller framework is shown in Figure 6.2. *Load monitors* run on every server and report their measurements to *advisors*. These measurements are used to maintain an up-to-date local view of the load situation of the system (see Section 4.3.2). Imminent overload situations[1] are reported to a *load monitoring system* which observes the load changes for a while and triggers a *fuzzy controller* in case of a real overload situation. This fuzzy controller initiates actions to prevent critical load situations. For example, if a CPU overload on a service host is detected, the controller can move services from this overloaded host to currently idle hosts. A *load archive* stores aggregated historic load data. These modules are described in more detail in the following sections.

The controller framework is a service itself, i.e., it runs in a ServiceGlobe environment. Thus, this flexible and extensible controller framework is seamlessly integrated into the ServiceGlobe platform.

6.2.1 Load Monitors and Advisor Modules

Every server and every service is monitored by a load monitor service.[2] Whenever a new service is started in the ServiceGlobe system, a new advisor is instantiated by the controller.

[1]The controller also reacts in failure and idle situations. Because the handling of these situations is quite analogous, we focus on overload situations in the remainder of this chapter.

[2]Figure 6.2 only shows the load monitors and advisors responsible for the servers. For simplicity of the illustration, services running on the servers and their load monitors and advisors are omitted.

During instantiation, the advisor remotely executes the corresponding load monitor on the service host running the new service. Load monitors are specialized services for resource monitoring of service hosts and for monitoring of resource usage of services. The advisors and corresponding monitors use the UDP protocol to send load messages from a monitor to an advisor. These messages are used to maintain an up-to-date local view on the load situation of the system. Service hosts report their current CPU and main memory usage, whereas services report one numerical value representing their load.

6.2.2 Load Monitoring System

In real systems, short load peaks are quite common. If these peaks are relatively short, the controller should not react because there may not actually be a real overload situation. If the system always reacted immediately, it would be very instable. Thus, if load values exceed a tunable threshold, the advisor passes the load data to the load monitoring system module for further observation. Then the load data is observed for a tunable time (*watchTime*). After this period, the load monitoring system calculates the arithmetic mean of the load during the watchTime. It determines a real overload situation if the average load is above the threshold. If such an overload situation is detected, the fuzzy controller module is triggered.

6.2.3 Fuzzy Controller

If an overload situation is detected by an advisor and verified by the load monitoring system, the controller identifies an appropriate action to remedy the overload situation. For this purpose, it initializes the fuzzy controller with information about the current load situation of the affected services and servers. After that, the fuzzy controller calculates the applicability of all actions. Our controller is able to handle the following actions: start, stop, move, scale-in, scale-out, scale-up, and scale-down. If required, as for a move action, the fuzzy controller then calculates the score of all suitable target service hosts. Finally, the action with the highest applicability is executed and the host with the highest score is selected as target host of the action. The fuzzy controller is described in more detail in Section 6.4.

6.2.4 Load Archive

The load archive stores a persistent aggregated view of historic load data. This data is used to calculate the average load of services during their specific watchTime. These values are used to initialize all resource variables of the fuzzy controller before execution.

6.2.5 Environment and Service Virtualization

We use a blade server environment for the presentation of our autonomic computing concept and for first performance experiments. This is because our research prototype is being field

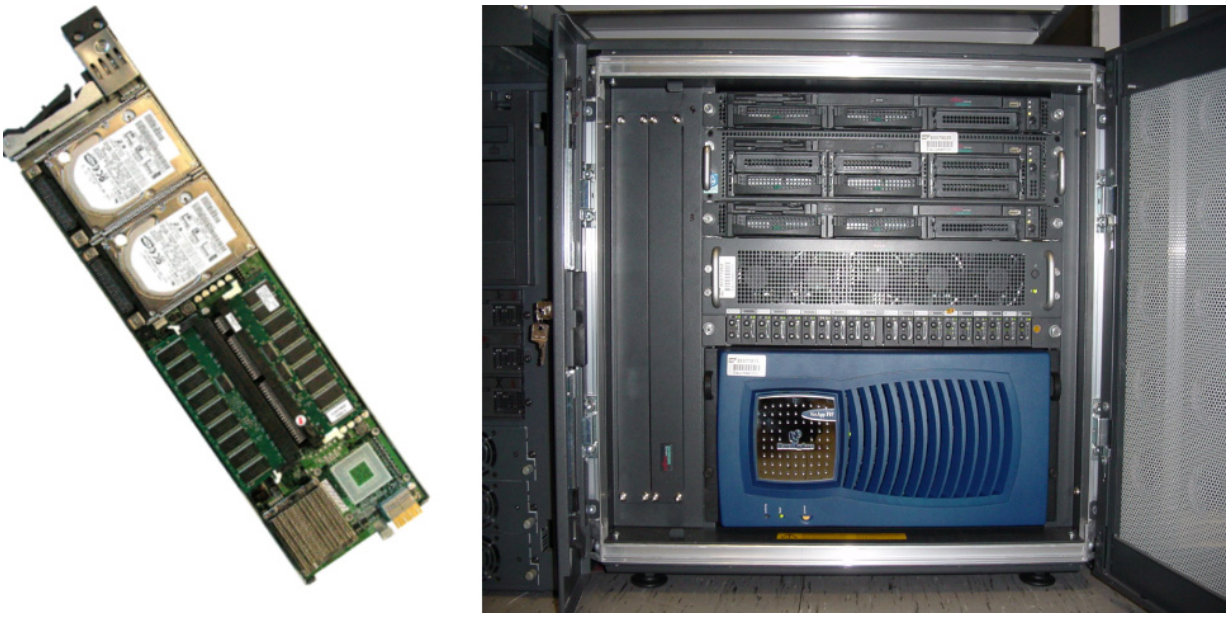


Figure 6.3: Blade Server and a Blade Server Rack

tested on a blade server environment. Blade servers are relatively cheap and the processing power can easily be scaled to the current processing demand by adding additional blades on the fly. Figure 6.3 shows a blade and a blade server rack containing infrastructure hardware like redundant power supplies and network switches.

Further, administration costs of blade servers are low compared to traditional mainframe hardware which was favored by large ERP installations in the past. Administration overhead can be further reduced by implementing, e.g., NetBoot based on the PXE Protocol [Sch03]. Using NetBoot, new blades added to the rack are booted instantly by loading a kernel and a software image over the network. After the boot process is finished, AutonomicGlobe is started automatically on this blade to make the blade available to the controller. Blade servers normally store their data using a storage area network (SAN) or a network attached storage (NAS). Thus, CPU power and storage capacity can be scaled independently and services can be executed on any blade because services can access their persistent data regardless of the blade on which they are running.

Services running on the blade servers are virtualized by usage of service IP addresses, i.e., every service has its own IP address assigned. This IP address is bound to the physical network interface card (NIC) of the host running the service. Thus, if a service is moved from one host to another, the virtual IP address is unbound from the NIC of the old host running the service and afterwards bound to the NIC of the target host. Thus, services are decoupled from servers. This service virtualization is a basic requirement for AutonomicGlobe.

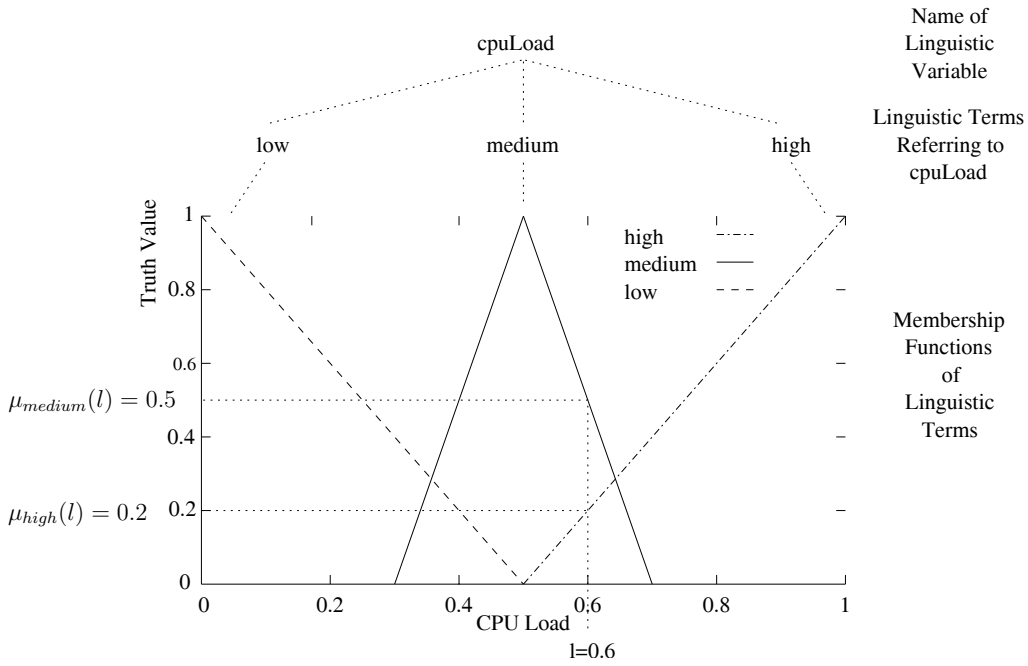


Figure 6.4: Linguistic Variable cpuLoad

6.3 Fuzzy Controller Basics

In general, fuzzy controllers are special expert systems based on *fuzzy logic* [KY94]. Fuzzy controllers are used in control problems for which it is difficult or even impossible to construct precise mathematical models. In the area of autonomic computing, these difficulties stem from inherent nonlinearities, the time-varying nature of the services to be controlled, and the complexity of the heterogeneous system. Contrary to classical controllers, fuzzy controllers are capable of utilizing the knowledge of an experienced human operator as an alternative to a precise model. This knowledge is expressed using intuitive linguistic descriptions of the manner of control.

Fuzzy logic is the theory of *fuzzy sets* devised by Lotfi Zadeh in 1965 [Zad65]. The membership grade of elements of fuzzy sets ranges from 0 to 1 and is defined by a membership function. Let X be an ordinary (i.e., crisp) set, then

$$A = \{(x, \mu_A(x)) \mid x \in X\} \quad \text{with} \quad \mu_A : X \rightarrow [0, 1]$$

is a fuzzy set in X. The membership function μ_A maps elements of X into real numbers in $[0, 1]$. Thereby, a larger value (truth value) denotes a higher membership grade.

Linguistic variables are variables whose states are fuzzy sets. These sets represent *linguistic terms*, such as low, medium, and high. A linguistic variable is characterized by its name, a set of linguistic terms, and a membership function for each linguistic term. An example for the linguistic variable *cpuLoad* is shown in Figure 6.4. The figure shows the three linguistic terms *low*, *medium*, and *high* with their assigned trapezoid membership

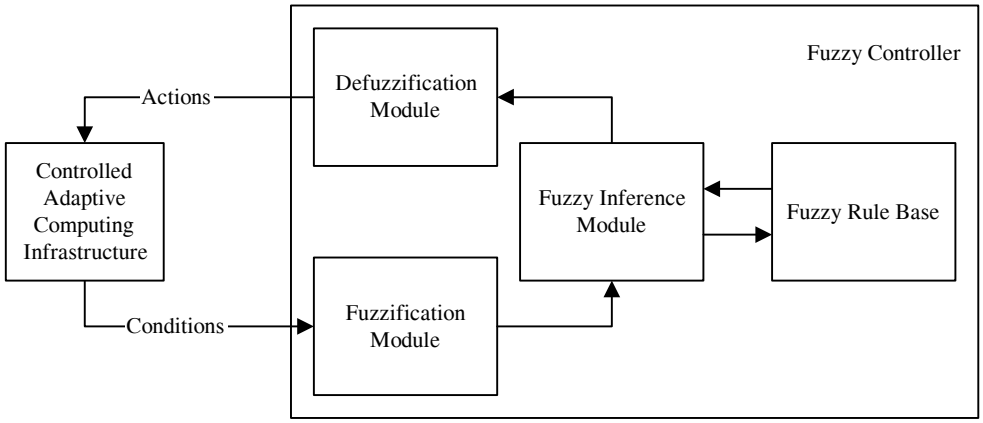


Figure 6.5: Architecture of a Fuzzy Controller

functions[3]. Other shapes of membership functions can be used as well, but since trapezoid functions were empirically proven to be effective, we decided to use them.

Figure 6.5 shows the general architecture of a fuzzy controller according to [KY94]. The controller works by repeating a cycle of three steps. First, measurements are taken of all variables representing relevant conditions of the controlled infrastructure. These measurements are converted into appropriate fuzzy sets (input variables) in the fuzzification step. After that, these fuzzified values are used by the inference engine to evaluate the fuzzy rule base. At last, the resulting fuzzy sets (output variables) are converted into a vector of crisp values during the defuzzification step. The defuzzified values represent the actions taken by the fuzzy controller to control the infrastructure. We will now explain the fuzzy controller mechanisms in more detail by way of an example from the area of autonomic computing.

During the fuzzification phase, the crisp values of the measurements (e.g., CPU load of a host) are mapped onto the corresponding linguistic input variables (e.g., cpuLoad) by calculating membership rates using the membership functions of the linguistic variables. For example, according to Figure 6.4, a host having a measured CPU load $l = 0.6$ (60%) has 0.5 medium and 0.2 high cpuLoad.

In the inference phase, the fuzzy rule base is evaluated using the fuzzified measurements. The form of the rules is exemplified by the two sample rules[4]

```
IF cpuLoad IS high AND
   (performanceIndex IS low OR performanceIndex IS medium)
THEN scaleUp IS applicable

IF cpuLoad IS high AND performanceIndex IS high
```

[3]In the figure, triangular functions are shown which are a special form of trapezoid functions. Actually, our controller can handle arbitrary trapezoid functions.

[4]These simple rules are only used to explain the inference phase. The rules used in our autonomic computing system are generally more complex.

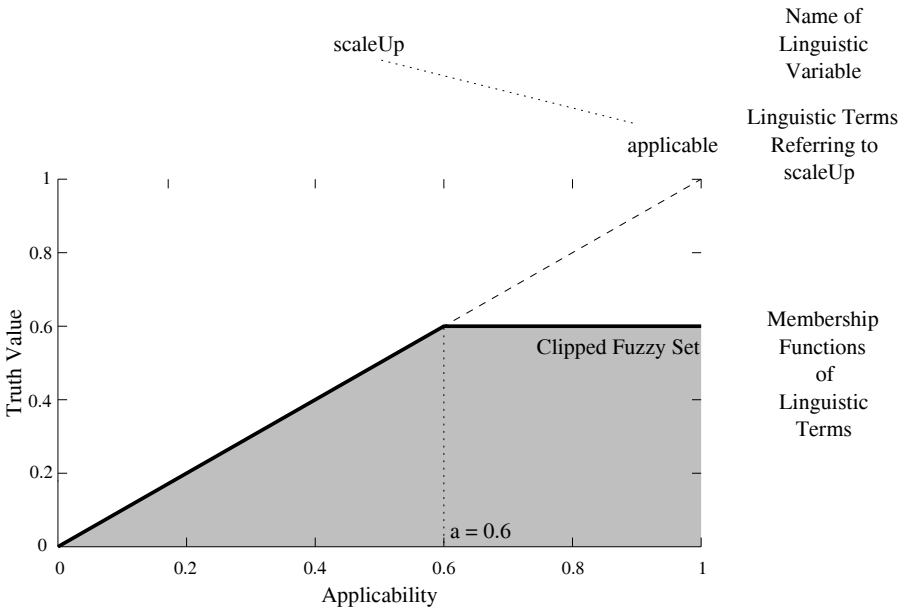


Figure 6.6: Max-Min Inference Result

```
THEN scaleOut IS applicable
```

where `cpuLoad` and `performanceIndex` (specifying the relative performance of a server) are the input variables and `scaleUp` and `scaleOut` are the output variables. Typical fuzzy controllers have dozens of rules. Actually, AutonomicGlobe's fuzzy controller currently comprises about 40 rules. The first sample rule states that it is reasonable to move a service to a more powerful host (scale-up) if the host running the service has a high load and a low or medium performance index (the higher the performance index of a host, the more powerful it is). The second rule states that it is reasonable to start an additional service instance (scale-out) if the host running the service is highly loaded despite being quite powerful.

Conjunctions of truth values in the antecedent of a rule are evaluated using the minimum function. Analogously, disjunctions are evaluated using the maximum function. Given a CPU load of $l = 0.9$, the membership grades for the linguistic variable cpuLoad are $\mu_{low}(l) = 0$, $\mu_{medium}(l) = 0$ and $\mu_{high}(l) = 0.8$. Given a performance index of $i = 5$, we assume for this example that the membership grades for the linguistic variable performanceIndex are $\mu_{low}(i) = 0$, $\mu_{medium}(i) = 0.6$ and $\mu_{high}(i) = 0.3$. Thus, the truth value of the antecedent of the first rule evaluates to $min(0.8, max(0, 0.6)) = 0.6$ and the truth value of the antecedent of the second rule evaluates to $min(0.8, 0.3) = 0.3$.

In classical logic, the consequent of an implication is true if the antecedent evaluates to true. For fuzzy inference, there are several different inference functions proposed in the literature. We use the popular max-min inference function. Using this function, the fuzzy set specified in the consequent of a rule (e.g., applicable) is clipped off at a height corresponding to the rule's antecedent degree of truth. After rule evaluation, all fuzzy sets referring to the same output variable are combined using the standard fuzzy union

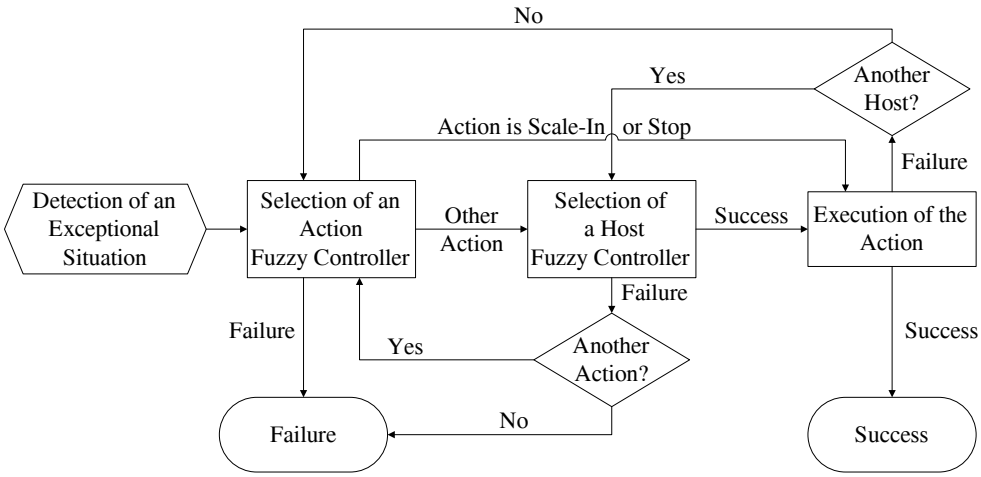


Figure 6.7: Interaction Flow Chart of the Fuzzy Controllers

operation:

$$\mu_{A \cup B}(x) = \max\left(\mu_A(x), \mu_B(x)\right) \quad \text{for all } x \in X$$

The resulting combined fuzzy set is the result of the inference step. Figure 6.6 shows the result of the inference for the linguistic output variable scaleUp.

During the defuzzification phase, a sharp output value is calculated from the fuzzy set that results from the inference phase. There are several defuzzification methods described in the literature. We use a *maximum* method such that the result is determined as the leftmost of all values at which the maximum truth value occurs. Regarding our example shown in Figure 6.6, the crisp value for the action scale-up is $a = 0.6$, i.e., the action is applicable to a degree of 0.6. The linguistic variable scaleOut is defined analogously. Thus, the action scale-out is applicable to a degree of 0.3. Therefore, the controller will favor the scale-up action for execution.

6.4 Fuzzy Controller for Load Balancing

The fuzzy controller module in AutonomicGlobe consists of two separate fuzzy controllers. The first one reacts on exceptional situations and determines an appropriate action. If the selected action requires a target host, e.g., scale-out, a second fuzzy controller is triggered to determine a suitable service host. Figure 6.7 shows the interaction of the two fuzzy controllers *selection of an action* and *selection of a host*. After a rearrangement has taken place, the involved services and servers are *protected* for a certain time, i.e., they are excluded from further actions. This protection mode prevents the system from oscillation, i.e., the moving of services back and forth.

Variable	Description
cpuLoad	CPU load of the server (average load of all CPUs)
memLoad	main memory load of the server
performanceIndex	performance index of the server
instanceLoad	load of the service instance
serviceLoad	average load of all instances of the service
instancesOnServer	number of services running on the server
instancesOfService	number of instances of the service

Table 6.1: Input Variables for the Action-Selection

6.4.1 Action-Selection Process

In the first phase, the input variables of the fuzzy controller are initialized. Table 6.1 shows the input variables of our controller. All variables of the fuzzy controller regarding CPU or memory load are set to the arithmetic means of the load values during the service specific watchTime. The other variables are initialized using the current measurements or using available metadata, e.g., for the performanceIndex.

The fuzzy controller distinguishes between exceptional situations induced by a service, and exceptional situations induced by a server (see Figure 6.8). If a service has triggered the controller, it decides on the basis of information about the considered service, the service instance, and the server on which it is executed. Other services running on the considered host are not taken into account. If a server triggered the fuzzy controller, the gathered information of all services running on the considered host must be taken into account.

Since the action-selection process depends on the specific situation, our controller is able to handle dedicated rule bases for different exceptional situations (triggers). We distinguish between four different triggers: *serviceOverloaded*, *serviceIdle*, *serverOverloaded*, and *serverIdle*. Further, our controller facilitates dynamic adaptations. For example, an administrator can add service-specific rule bases for mission critical services to favor powerful servers for these services.

A rule base comprises dozens of rules, each consisting of an antecedent and a consequent. Figure 6.9 shows an example rule base for the serviceOverloaded trigger. The rules presented are the same as in Section 6.3, given in XML format.

The fuzzy controller evaluates the appropriate rule base and calculates crisp values for the output variables. Table 6.2 shows the output variables. These output variables represent the actions executed by the controller to control the infrastructure.

The fuzzy controller only considers actions that do not violate any given constraint, e.g., a database service normally does not support a scale-out. Thus, the action scale-out is not possible for such a service. These constraints are defined using a declarative XML language. The result of the fuzzy controller is a list of actions along with their ratings between 0% and 100%. These ratings determine how applicable the actions are in the current situation. In case a server triggered the controller, we execute the fuzzy controller for each service running on the server and subsequently collect the possible actions of all services.

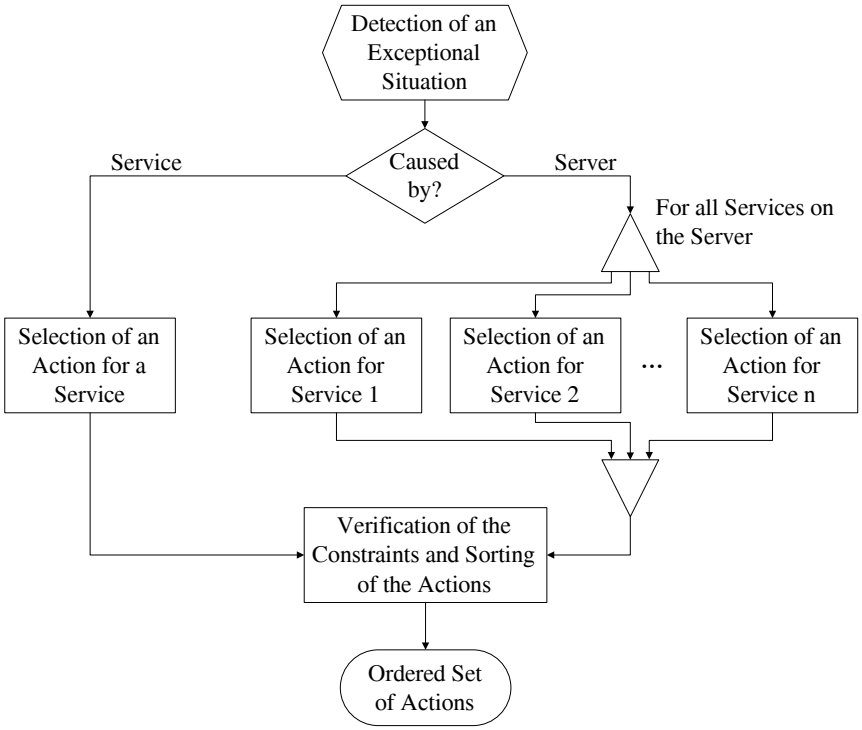


Figure 6.8: Flow Chart of the Action-Selection Process

Afterwards, the actions are sorted by their applicability in descending order. Actions whose applicability value is lower than an administrator-controlled minimum threshold are discarded. The first action of the list is selected and verified once more. This is necessary because the fuzzy controller is able to handle several exceptional situations concurrently. Thus, if as an example the maximum number of instances of a service is currently running, no additional instance can be started. Consequently, a scale-out cannot be performed.

Variable	**Description**
start	starting of a service
stop	stopping of a service
scaleIn	stopping of a service instance
scaleOut	starting of a service instance
scaleUp	movement of a service instance to a more powerful host
scaleDown	movement of a service instance to a less powerful host
move	movement of a service instance to an equivalently powerful host

Table 6.2: Output Variables for the Action-Selection

```
<ruleBase name="serviceOverloaded">
  <rule>
    <condition>cpuLoad is high and
        (performanceIndex is low or performanceIndex is medium)
    </condition>
    <action>scaleUp is applicable</action>
  </rule>
  <rule>
    <condition>cpuLoad is high and performanceIndex is high</condition>
    <action>scaleOut is applicable</action>
  </rule>
  ...
</ruleBase>
```

Figure 6.9: Rule Base for the serviceOverloaded Trigger

Variable	Description
cpuLoad	CPU load of the server (average load of all CPUs)
memLoad	main memory load of the server
instancesOnServer	number of instances on the server
performanceIndex	performance index of the server
numberOfCpus	number of CPUs of the server
cpuClock	clock speed of the CPUs of the server
cpuCache	cache size of the CPUs of the server
memory	main memory size of the server
swapSpace	size of the available swap space
tempSpace	size of the available temporary disk space

Table 6.3: Input Variables for the Selection of a Server

6.4.2 Server-Selection Process

In the case of a scale-out, scale-up, scale-down, move, or start, an appropriate target server must be chosen to specify where the action should take place. The selection of a server proceeds analogously to the selection of an action. First, a list of all possible servers is compiled. Initially, these are all servers on which an instance of the service can be started and that are not in protection mode. For each server the fuzzy controller is executed with the input variables initialized to the current values. Table 6.3 shows the input variables for the server-selection.

Since the server-selection process depends on the specific action, our controller is able to handle different rule bases for different actions. With these rules we determine the applicability of a server for handling the given situation. In the defuzzification phase, the controller calculates a crisp value for every possible host and selects the most applicable server.

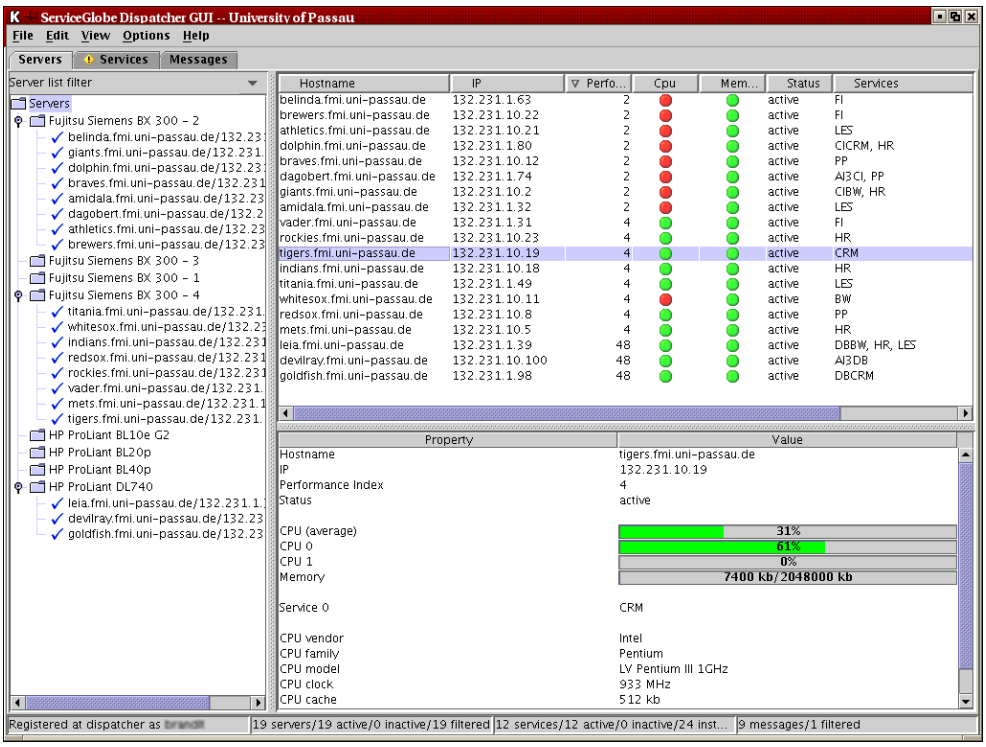


Figure 6.10: Administrator Controller GUI

6.4.3 Execution of the Controller's Decision

The controller can operate in two different modes: in the *automatic mode*, the actions are logged and then executed. In *semi-automatic mode*, the human administrator is contacted to confirm the action prior to execution. Before an action is actually executed, the controller checks once more whether or not the concerned resources are in protection mode. This is necessary because a concurrently running selection process might have resulted in a conflicting action in the meantime. If all preconditions hold, the controller carries out the action.

Otherwise, the execution of the action fails and the controller tries the next available host respectively action. If there are no more possible hosts and actions with a sufficient applicability, the controller requests human interaction by alerting the system administrator. For this purpose, our controller is coupled with a graphical control console which displays the monitored state of the system. Using this console, the administrator can manually execute actions normally triggered by the fuzzy controller. Figure 6.10 shows the GUI of the controller console. There are three different views: the server view displays information about the controlled servers, the service view is analogously displaying information about the controlled services, and the message view lists administrative messages and notifications. The presented screenshot shows the server view. The panel on the left-hand side shows a list of all controlled servers grouped by category. The upper right-hand panel displays overview information about all servers belonging to the selected category. Finally,

the lower right-hand panel displays detailed information about the selected server.

6.5 Simulation Studies

We performed comprehensive simulation studies using our prototype implementation of AutonomicGlobe to assess the effectiveness of our autonomic computing concept. They have been conducted using a simulation environment that models a realistic ERP installation.

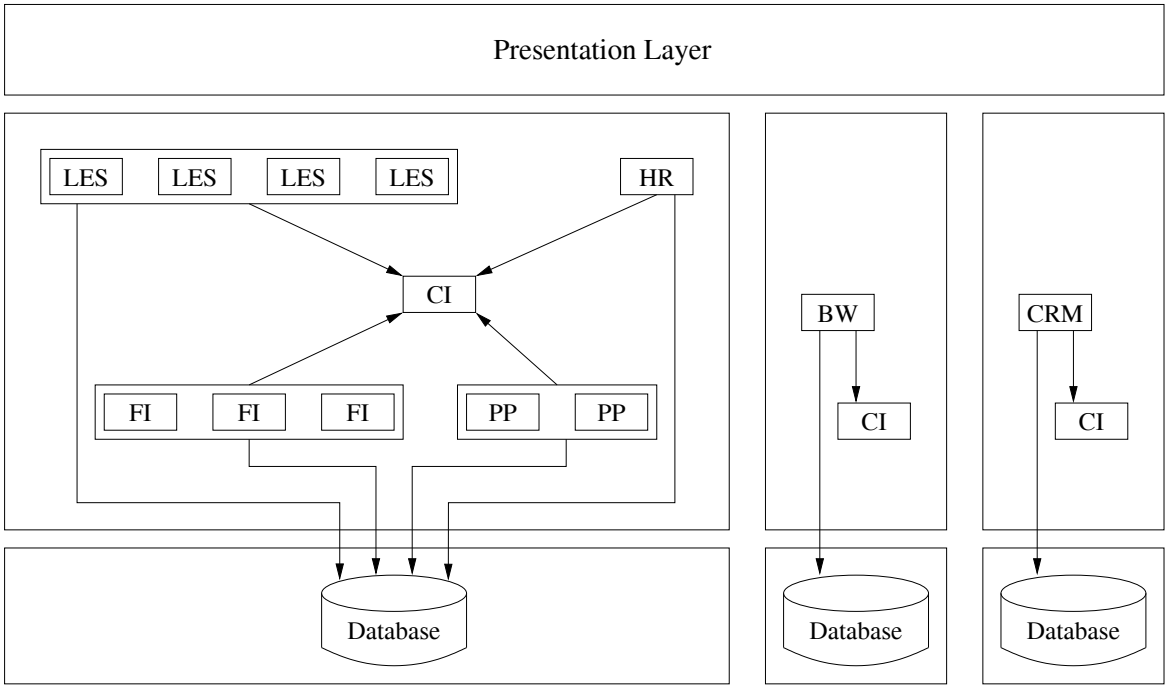
6.5.1 Description of the Simulation Environment

The simulation environment models a realistic ERP system with the corresponding hardware. Our simulated ERP system is based on an SAP installation. The simulated services and servers are described using our declarative XML language, just like real existing services and servers. Figure 6.11 shows the architecture of our simulated ERP installation, which is—like, e.g., SAP ERP systems—divided into three layers: the database layer, the application server layer, and the presentation layer. End-users communicate with the ERP installation using clients in the presentation layer. The end-users' clients themselves do not affect the system, thus we only simulate the number of users connected to services of our simulated ERP installation. Our installation comprises three subsystems in the application and database layer: classical *Enterprise Resource Planning (ERP)*, *Business Warehouse (BW)*, and *Customer Relationship Management (CRM)*, each running its own dedicated database and central instance. The central instance application servers (CIs) are responsible for the global lock management of their particular subsystem. The other application servers (BW, CRM, FI, HR, LES, PP) execute the application logic, i.e., process user requests. Our controller supervises these application servers, databases, and central instances.

In a real system, there is a great deal of communication between the individual services. In our simulation environment, we neglect communication costs because we assume a local high-bandwidth network connection. This is realistic in blade server environments which are normally equipped with Gigabit Ethernet or Infiniband.

Our system simulates a varying number of users generating requests. As observed in running SAP installations, the course of a request is simulated as follows. First, a request increases the load of the affected service host for a short period. Before handling the request in the database, the lock management of the central instance (CI) is requested. Therefore, the load drops on the application server and increases on the central instance. In case of a positive check the request is passed to the database. Thus, load drops on the central instance and increases on the database for the processing time. Finally, the database sends the answer back to the application server. Thus, for a short period, the load drops on the database and rises on the application server. Since the load caused by a single request depends on the specific service, e.g., an FI request produces lower load than a BW request, our simulation system uses service-specific parameters to simulate the impact of requests.

In addition to the load produced by user requests, every application server itself induces



BW: Business Warehouse
CI: Central Instance
CRM: Customer Relationship Management
FI: Financial Accouting
HR: Human Resources
LES: Logistics Execution System
PP: Production and Planing System

Figure 6.11: Example of an ERP Environment

a basic load. The load curves generated by simulated services follow predetermined patterns that can be observed in many companies running SAP software. Figure 6.12 shows example load curves for an LES and for a BW application server over one day. At eight o'clock, when the employees begin work, the number of requests sent to the LES application servers increases. There are three peaks—one in the morning, one before midday, and one before the employees leave. The load curve of a BW application server is different. During the night, several heavy-load batch jobs are processed. During the day, only few user requests have to be processed based on the aggregated data.

We assume a hardware environment that is scaled for peak load as that is quite common in today's computing centers. A standard single processor blade in our simulation (performance index = 1) is dimensioned to handle at most 150 users of one service. The CPU load of the blades is between 60% and 80% during main activity in order to retain reserves for unpredictable load bursts. Figure 6.13 shows the simulated hardware and the initial deployment of the services. The simulated servers are[5]:

[5]The performance index values stated are based on estimations and do not necessarily reflect the true

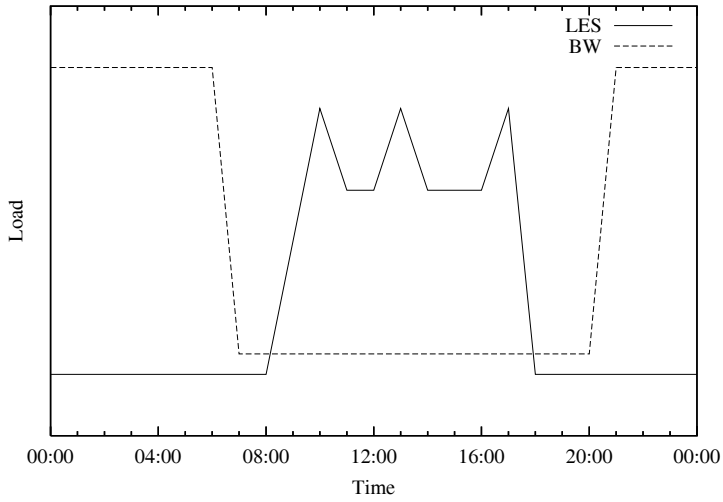


Figure 6.12: Qualitative Load Curve of LES and BW

Service	Number of Users	Number of Instances
BW	60	2
CRM	300	1
FI	600	3
HR	300	1
LES	900	4
PP	450	2

Table 6.4: Initial Number of Users

- 8 FSC-BX300 blades with one Intel Pentium III 933 MHz processor and 2 GB main memory each (performance index = 1).
- 8 FSC-BX600 blades with two Intel Pentium III 933 MHz processors and 4 GB main memory each (performance index = 2).
- 3 HP-Proliant BL40p servers with four Intel Xeon MP 2.8 GHz processors and 12 GB main memory each (performance index = 9).

Table 6.4 shows the number of users per service and the number of instances that are started initially. These numbers are reasonable for a medium-sized company running an SAP system, e.g., most departments use the LES application servers while only the staff department uses the HR application servers.

Every simulation starts with the same reasonable initial deployment of the services shown in Figure 6.13. We run different simulation series and continually increase the number of users by 5% until the system becomes overloaded. The BW is an exception because it processes batch jobs instead of interactive requests. Thus, we increase the load per batch job by 5% and leave the number of jobs constant.

performance of the servers.

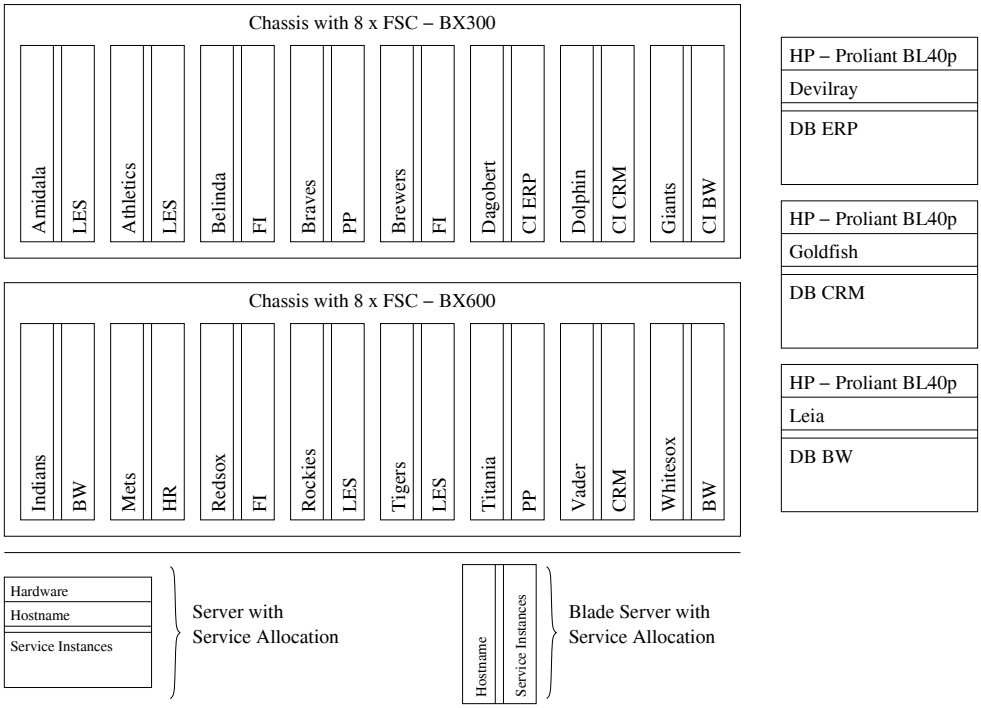


Figure 6.13: Simulated Hardware and Initial Deployment

Service	**Conditions**	**Possible Actions**
database ERP	exclusive	–
database BW, CRM	–	–
central instances	–	–
application server	minimum 2 FI instances minimum 2 LES instances	scale-in, scale-Out

Table 6.5: Services in the Constrained Mobility Scenario

We perform simulation series using three different scenarios. In the *static scenario*, a computing environment with all services being static is simulated. This is the standard environment used in most computing centers today. In the *constrained mobility scenario*, we simulate an ERP environment supervised by our controller, where some but not all services support the scale-in and scale-out action (Table 6.5 gives an overview). The *exclusive* condition states that no other service may be executed on the host. The *minimum instances* condition defines the minimum number of instances of a service allowed. In this scenario, all databases and central instances are static. Users are distributed to the instances of a service according to the performance indexes of the servers running the instances. Application servers support scale-in and scale-out. After an scale-out, the system does not dynamically redistribute the users, i.e., users are logged in at one service instance during their complete session. We simulate a fluctuation of the users, i.e., users infrequently log themselves off of the application server they are connected to and reconnect to the currently least-

Service	Conditions	Possible Actions
database ERP	exclusive	–
database CRM	–	–
database BW	minimum performance index 5	scale-in, scale-out
central instances	–	scale-up, scale-down, move
application server	minimum 2 FI instances minimum 2 LES instances	scale-in, scale-out, scale-up, scale-down, move

Table 6.6: Services in the Full Mobility Scenario

loaded server. This behavior can be observed in real systems, too. In the *full mobility scenario*, we simulate a system where the BW database can be distributed across several servers. The central instances and the other application servers can be moved from one host to another (see Table 6.6 for details). The *minimum performance index* constraint defines the minimum performance index requirements of a service. Furthermore, users are dynamically redistributed across all instances of a service if any instance gets overloaded.

Today, the movement of services as well as the dynamic redistribution of users are only supported by few services because services must explicitly assist the movement or redistribution. Movement requires that the service be able to store its internal state before it is stopped, and that the newly started instance can restore the old state. Furthermore, it must be guaranteed that the users can be reconnected automatically to the newly instantiated service instance. Dynamic redistribution requires that the service be able to move parts of its state to another instance. In the future, we expect that more services will support dynamic relocation and redistribution and thus consider them in the full mobility scenario.

To prevent the system from reacting too late, we set the controller's threshold value for a CPU overload to 70%, i.e., if a server has more than 70% CPU load it is considered overloaded. In this case, the controller monitors the load values of the service for 10 minutes (watchTime) in order to prevent the system from over-reacting on short load bursts. After execution of an action, the affected services are protected for 30 minutes and affected servers are protected for 60 minutes. The threshold value for an idle situation depends on the performance index of the server and is 12.5%/performance_index. An idle situation is recognized after a watchTime of 20 minutes.

All simulation runs are carried out in 40-fold acceleration and are simulating a system for 80 hours. The shown time intervals correspond to simulated time.

6.5.2 Results of the Simulation Studies

Figures 6.14, 6.15, and 6.16 show simulation results with the number of users increased by 15% compared to the user numbers shown in Table 6.4. This demonstrates how the ERP installation handles an increasing number of users. The figures show the load curves of all servers and the average load of the whole system, indicated by a thick line.

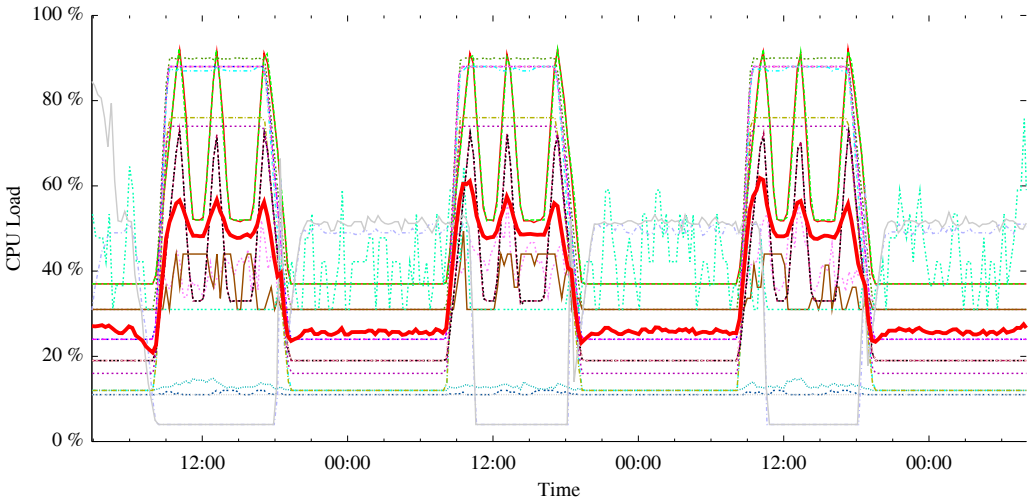


Figure 6.14: CPU Load of all Servers (Static Scenario)

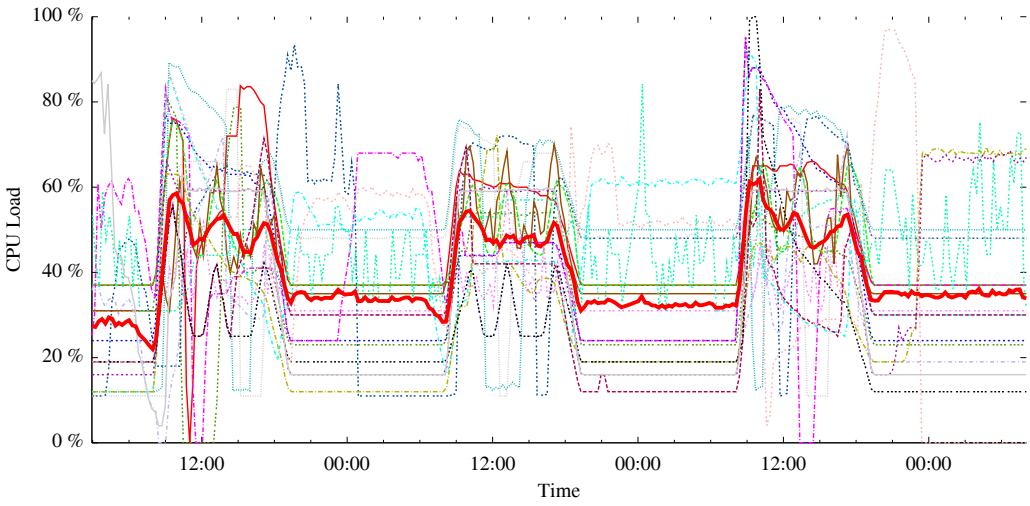


Figure 6.15: CPU Load of all Servers (Constrained Mobility Scenario)

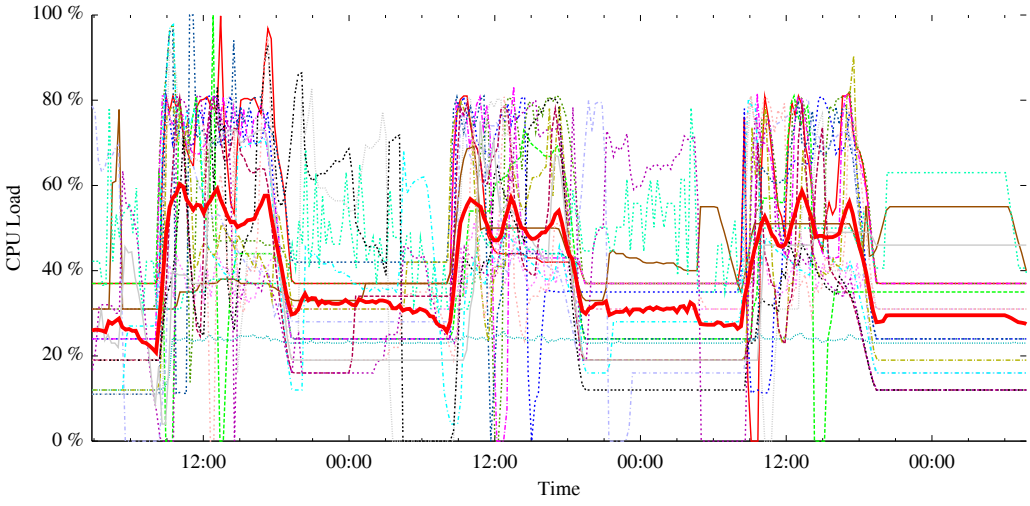
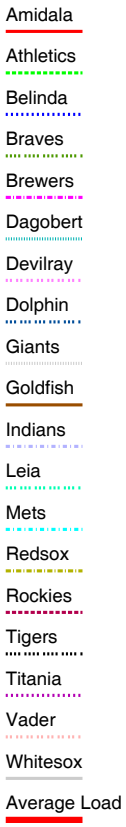


Figure 6.16: CPU Load of all Servers (Full Mobility Scenario)



In the static scenario, several servers become overloaded, i.e., have a CPU load of more than 80% for a long time[6], at regular intervals, thus a non-adaptive computing environment cannot handle this situation satisfactorily. If a host running an interactive service is overloaded, the service requires more time to process the requests and, therefore, delays new requests. Thus, users cannot perform all their requests in a given period, e.g., a working day, and requests will be delayed until the next day. If a BW application server is overloaded, the batch jobs require more time. Thus, they may become conflicted with other services and compete against them for resources.

The situation already improves in the constrained mobility scenario. The controller reacts on arising overload situations by automatically starting additional instances of services. Because the users are not dynamically redistributed after a scale-out has taken place, the original servers remain quite loaded for a while. Due to user fluctuations, the load of the initially overloaded services slowly decreases. Altogether, the overload situations are on average shorter than in the static scenario, but due to the restrictions of the static user distribution, the overload situations cannot be prevented completely.

In the full mobility scenario, the results are much better than in the constrained mobility scenario. Idle resources are efficiently used to relieve the load on heavily used resources. Thus, the utilization of the hardware is well-balanced. Due to the dynamic redistribution of users across all service instances, the effects of controller actions are observable almost instantly. Another advantage of the full mobility scenario is that the controller can react more flexibly on overload situations. The remaining short overload peaks at the beginning stem from the watchTime. If the instances of a service become overloaded, the controller monitors the instances for 10 minutes before starting a new instance. Therefore, for a short time, the existing instances stay overloaded. After the first day, there are normally more instances of every application server running than in the beginning. Thus, if the controller does not stop too many instances, the load can be distributed across a sufficient number of instances, and overload situations can be avoided.

In order to demonstrate the behavior of our controller in more detail, we present the FI application servers' load curves of the above described simulations.

Figure 6.17 shows the load curve of the FI application servers in the static scenario. There are three instances running on Belinda, Brewers, and Redsox. As services are static, the controller cannot remedy the overload situations. Thus, the service instances running on the less powerful blades become overloaded periodically. If a service or a server is overloaded, it can no longer be used in a reasonable way because the processing of mission critical OLTP-requests is slowed down.

Figure 6.18 shows load curves in the constrained mobility scenario. When the employees begin to work, the instances on Belinda and Brewers become overloaded. The controller's reaction is to start an additional instance on Dagobert ("Out Dagobert"). Since users are not redistributed dynamically, the load of Belinda and Brewers only decreases slowly.

[6]The controller considers a server already overloaded if it has more than 70% CPU load to prevent the system from reacting too late. Actually, we consider a server overloaded if it has more than 80% CPU load for a long time.

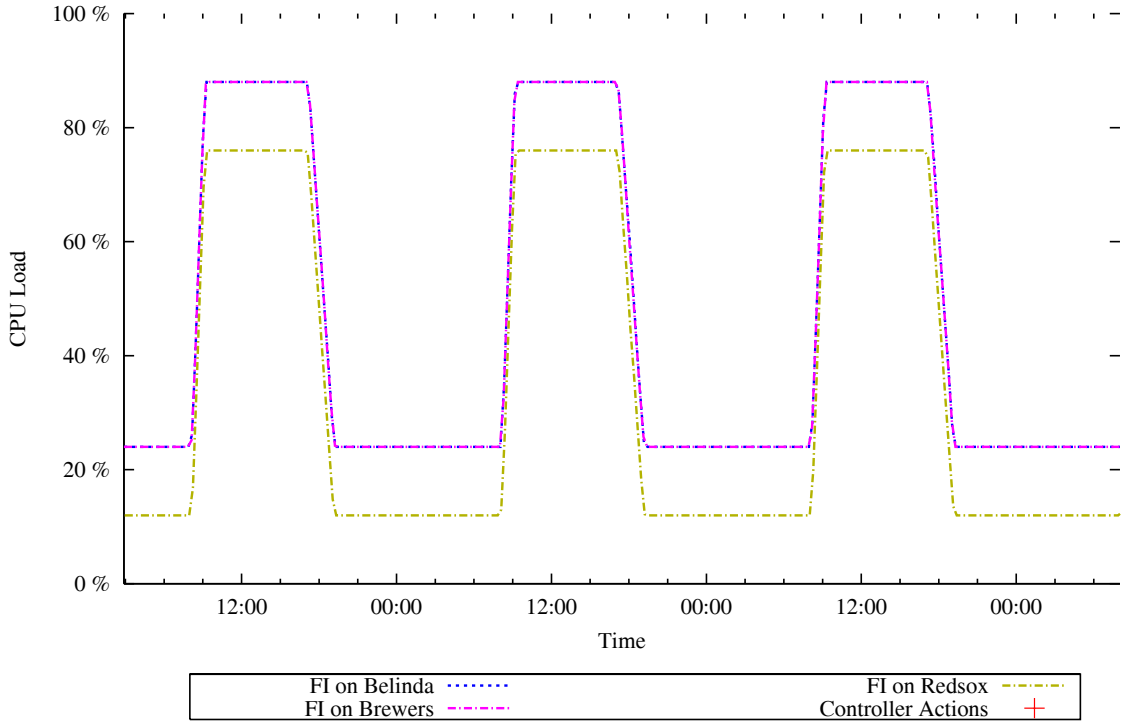


Figure 6.17: CPU Load of the FI Instances (Static Scenario)

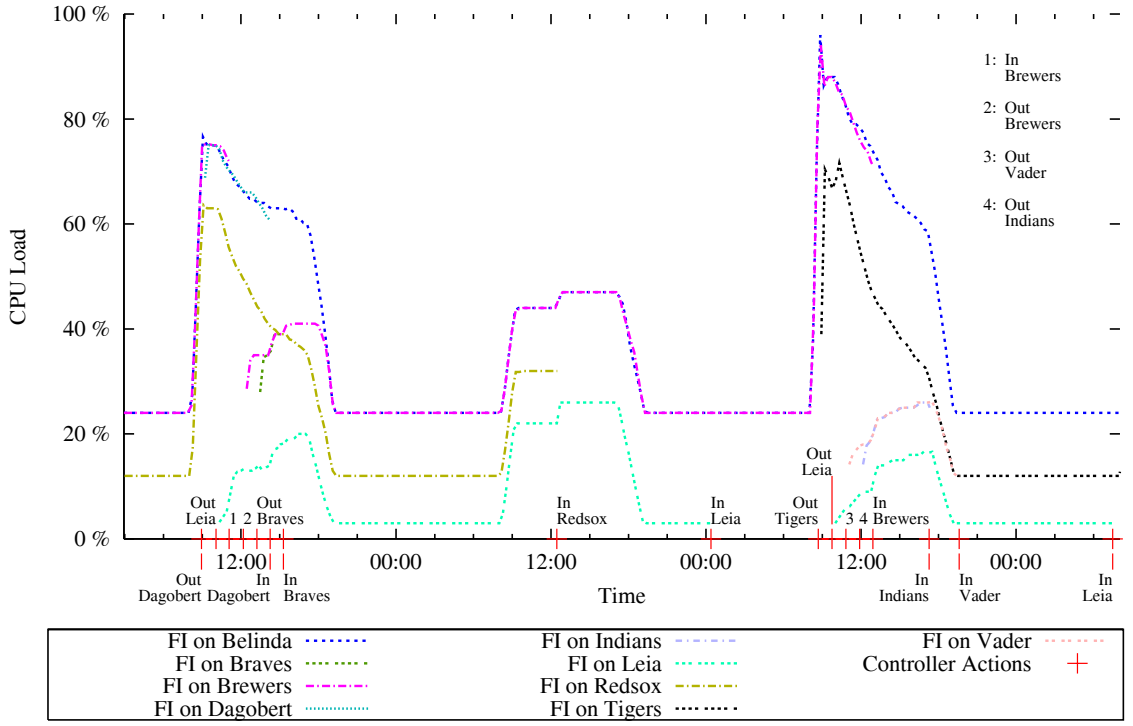


Figure 6.18: CPU Load of the FI Instances (Constrained Mobility Scenario)

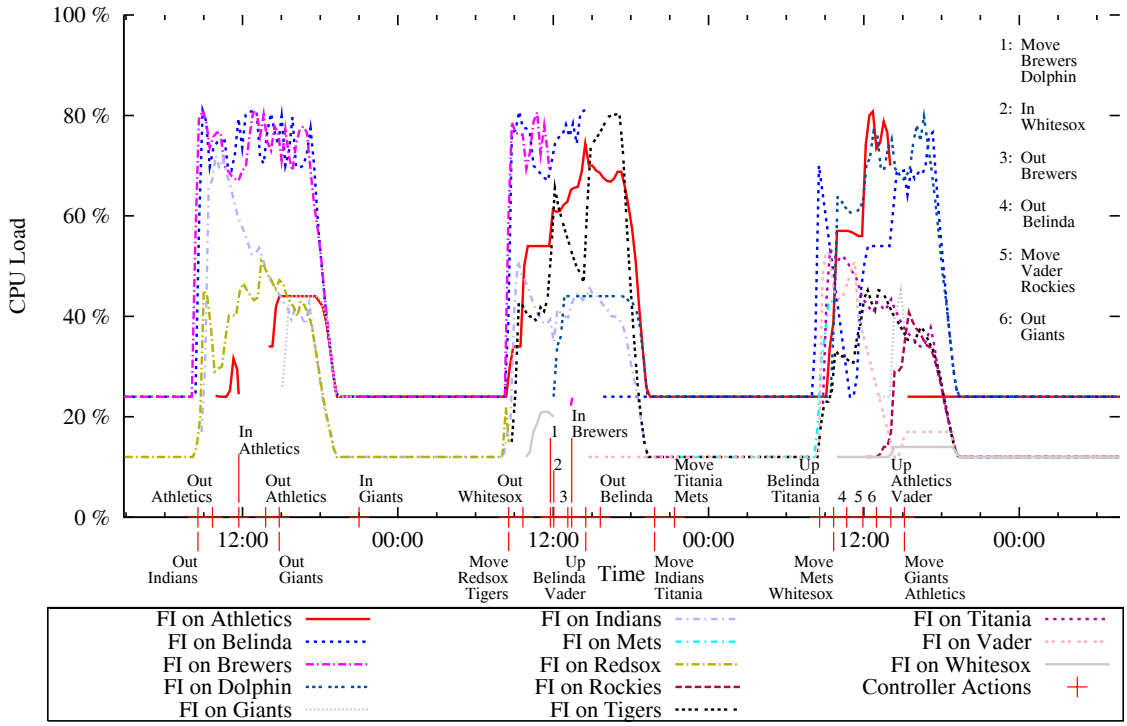


Figure 6.19: CPU Load of the FI Instances (Full Mobility Scenario)

These two hosts are still overloaded after the protection time, thus the controller starts another instance on Leia (“Out Leia”). Because these actions do not remedy the overload on Brewers fast enough, the controller decides to stop the instance running on Brewers (“In Brewers”) to protect the host from a continuous overload situation. This FI instance is started again (“Out Brewers”) after a short period of time due to an overload situation on Dagobert. Another FI instance is started on Braves (“Out Braves”). Further on, the controller starts new FI instances as required and stops instances running on overloaded blades and idle instances. During the second day, the controller needs only to execute one scale-in action because the FI instances running on Belinda, Brewers, and Leia can handle the load. The FI instance on Redsox is stopped (“In Redsox”) because Redsox is additionally running a CRM instance and, thus, is overloaded. The FI instance running on Leia is stopped (“In Leia”) in the night because the database of the BW subsystem uses the resources of Leia heavily. Thus, at the beginning of the third day, the remaining FI instances become overloaded. To remedy this overload situation, the controller starts new FI instances as required. In summary, the controller can avert most imminent overload situations from the FI. The remaining overload situation periods are short.

Figure 6.19 shows load curves in the full mobility scenario. Again, the controller adds and stops instances as required. Additionally, service instances are moved from heavy loaded servers to other servers. In this scenario, users are dynamically redistributed, thus the effects of controller actions are observable instantly and overload situation can be

Scenario	Maximum Number of Users
static	100%
constrained mobility	115%
full mobility	135%

Table 6.7: Maximum Possible, Relative Number of Users

averted completely.

6.5.3 Summary of Simulation Assessment

We ran simulation series for the three scenarios and each time increased the number of users by 5% until the system became overloaded, i.e., one or more servers had a CPU load of more than 80% for a long time. Table 6.7 shows the maximum numbers of users that can be handled by the existing hardware in the different scenarios. The values are relative to the number of users stated in Table 6.4.

In the static scenario, the hardware is sized for the initial number of users. Thus, if we increase the number of users by 5%, some servers immediately become overloaded. Using our controller in the constrained mobility scenario, the ERP installation can handle 15% more users because otherwise idle resources are used to remedy overload situations. Due to the restrictions of the static user distribution and of the available actions, idle resources cannot be used as efficiently as in the full mobility scenario. Nevertheless, our controller already works quite well for the constrained mobility scenario. In the full mobility scenario, our controller can push the number of users that can be handled by the ERP installation to 135% compared to the static scenario. The number of users is higher than in the constrained mobility scenario because idle resources can be used more efficiently.

The conclusion of our studies is that our controller can improve the capability of current IT infrastructures if static services like databases and central instances are deployed well. Additional degrees of freedom and dynamic user redistribution result in much more effective controller actions and, thus, a higher number of users that can be handled by the available hardware.

6.6 Related Work

Weikum [WMHZ02] motivates automatic tuning in the database area and concludes that it should be based on the paradigm of a feedback control loop which consists of three phases: observation, prediction, and reaction. [MLR03] presents IBM's autonomic query optimizer—based on a feedback control loop—that automatically self-validates its model without requiring any user interaction to repair incorrect statistics or cardinality estimates. [KMP93] developed a similar self-optimizing query optimizer that is based on a blackboard architecture known from the area of artificial intelligence. [LL02] presents several self-healing and self-optimizing features of IBM DB2. These features optimize, e.g., indexes

and performance parameters. Rather than concentrating on a single software system, we focus on the optimization of a complex adaptive computing infrastructure. This is necessary as the complexity of current computer infrastructures is evermore increasing.

Since IBM coined the term autonomic computing [Hor01] in October 2001, several global players initiated research projects in this area. An autonomic computing system provides self-managing capabilities, i.e., it handles self-configuration, self-healing, self-optimization, and self-protection. First results of this research area are already integrated in the IBM Director 4.1 [IBMa]. Using this tool, administrators can view and track the hardware configuration of remote systems and monitor the usage and performance of critical components. Further, it contains tightly integrated tools, e.g., for monitoring the availability of hardware and software and distributing system resources according to administrator-defined policy entitlements, to optimize performance and maximize availability. Sun N1 Grid [Suna] is Sun's vision, architecture, and product for optimizing network computing. It virtualizes the data center and monitors the computing infrastructure. The HP Utility Data Center [HP] is designed for on demand computing systems where processing needs are constantly changing. It virtualizes, consolidates, and standardizes the hardware. Thus, the administrators can dynamically allocate and reallocate resources via a Web-based interface. While the commercial products depend on vendor-specific hardware-features, our solution is independent of the underlying hardware. Further, our autonomic controller supervises and controls a complex computing infrastructure without human interaction and/or supports administrators by giving recommendations.

For the description of the servers and services we use a proprietary XML language that is based on preliminary versions of an XML language for the description of servers and services from the "Scheduling and Resource Management" project group of the Global Grid Forum [GGF]. If this XML language becomes standard, we will adopt it.

The author of [Bou01] pragmatically explains the concepts and terminology of load balancing. This book shows the complexity of load balancing in computing infrastructures.

Our work is also related to the academic projects Autonomia [DHX+03] and AutoMate [ABL+03]. The Autonomia environment provides a core autonomic middleware service to maintain autonomic requirements. Its methodology to achieve self-control and management is based on three procedures: monitoring, analysis and verification, and adaptation. Until now, they have implemented a proof-of-concept prototype that handles several aspects of self-healing. Currently, they are working on integrating self-optimizing features. AutoMate is a framework for enabling autonomic grid applications. They use a decentralized deductive engine, that provides the core capabilities for supporting autonomic compositions, adaptations, and optimizations. In their system, the monitored services themselves communicate informations about their behavior, resource requirements, performance, and adaptability to the AutoMate system. In contrast to Autonomia and AutoMate, our autonomic controller uses an adaptive fuzzy controller. Thus, the load balancing and reaction on exceptional situations is easy to configure and to administrate. Further, the monitored services need not to be modified.

6.7 Status and Future Work

We presented our novel autonomic computing concept which is hiding the ever increasing complexity of managing IT infrastructures. AutonomicGlobe is based on our distributed and open service platform ServiceGlobe. We described the architecture of AutonomicGlobe and its controller framework, which enhances ServiceGlobe with an active control component for autonomic service and server management. We presented a fuzzy controller which generates actions to remedy imminent overload, failure, and idle situations. We demonstrated the effectiveness of our proposed solution by performing a set of comprehensive simulation studies using our prototype. The results of these studies confirm the applicability of a fuzzy controller for the supervision of an adaptive computing infrastructure, and the benefits of such an infrastructure even for already existing complex environments.

We implemented a research prototype of AutonomicGlobe and currently field test it on blade server environments using a rule base comprising about 40 rules. Up to now, the largest environment used for testing was a blade server system with 160 processors overall (with 2 and 4 processors per blade, respectively).

We are currently working on real-world experiments and on further simulation studies which are additionally simulating random one-time events and work loads which do not recur on a daily basis (e.g., payroll accountings running at the last business day of every month) to check the behavior of our controller. Additionally, we are investigating some ideas on how the controller can exercise more comprehensive control. First, we will enhance the controller in such a way that it can manage explicit reservations, i.e., that an administrator can register critical tasks along with their resource requirements. Second, we will work on predicting the future load of services based on historic data stored in the load archive using pattern matching and data mining techniques. Based on explicit reservations for mission critical services and on the predicted load situation of services, we plan to develop a landscape designer tool. This tool calculates a statically optimized pre-assignment of all services to improve the dynamic optimization potential of the fuzzy controller. Additionally, this information can be used to support (and improve) both the action-selection and server-selection process of the controller. Thus, for example, the controller could avoid starting a new service instance on a host which will most likely soon be heavily loaded by another service.

Chapter 7

Conclusions

In this thesis, several aspects of distributed information systems have been investigated: security, caching, and self-management.

The substantial growth of the Internet led to the growing desire to be able to handle the flood of information available online. Thus, more and more complex data integration systems have been developed. Modern systems like ObjectGlobe are extensible by mobile user-defined code. Of course, such code introduces specific security concerns. We presented an effective security framework for distributed and open systems and used ObjectGlobe as an example. The security requirements of users are satisfied by the OperatorCheck server, which is used to rate the quality of external operators and test their semantics. Privacy of data is guaranteed by isolating external operators and by usage of secure communication channels. Cycle providers are protected using a monitoring component which tracks the resource usage of external operators to prevent them from monopolizing resources, and an admission control system to guard providers against overload situations. A security manager and class loaders are used to protect cycle providers from unauthorized resource accesses and to shield the ObjectGlobe system from external operators. Additionally, we presented the authentication framework of ObjectGlobe which can be used by cycle providers to determine the identity of users in a reliable way.

The second wave of integration currently rolling through the Internet makes service-oriented architectures based on Web services broadly available. Although Web services offer solutions to plenty of hard integration problems, there still remain several others to be solved. One of these problems is the performance and scalability of globally accessible Web services. We presented the semantic cache SSPLC suitable for caching responses from Web services on the SOAP protocol level. We introduced an XML-based declarative language to annotate WSDL documents with information about the semantics of service requests and responses. We demonstrated the validity of our proposed caching scheme by performing a set of experiments. The results of these experiments confirm the reduction of the processing demands on the central servers and the diminishment of bandwidth consumption, as well as competitive average response times.

Another problem which is currently not addressed by existing Web service standards is the increasing complexity of managing IT infrastructures. We presented a novel auto-

nomic computing concept which is hiding this complexity. We described the architecture of AutonomicGlobe and its controller framework, which enhances ServiceGlobe with functionality to generate an up-to-date view of the load situation of the system. This view is used by a fuzzy controller to generate actions to remedy imminent overload, failure, and idle situations. We demonstrated the effectiveness of our proposed solution by performing a set of comprehensive simulation studies using our prototype. The results of these studies confirm the applicability of a fuzzy controller for the supervision of an adaptive computing infrastructure and the benefits of such an infrastructure.

Bibliography

[ABL+03] M. Agarwal, V. Bhat, H. Liu, V. Matossan, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. AutoMate: Enabling Autonomic Applications on the Grid. In *Proceedings of the International Workshop on Active Middleware Services (AMS)*, pages 48–59, Seattle, WA, USA, June 2003.

[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.

[AH02] L. A. Adamic and B. A. Huberman. Zipf's Law and the Internet. *Glottometrics*, 3:143–150, 2002.

[Aka] Akamai Technologies. Akamai: The Business Internet. `http://www.akamai.com/`.

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

[Axi] Apache Web Services Project: Axis. `http://ws.apache.org/axis/`.

[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.

[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.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP11`, May 2000. World Wide Web Consortium (W3C), W3C Note.

[Ber02] H. Berghel. Responsibe Web Caching. *Communications of the ACM (CACM)*, 45(9):15–20, 2002.

[Bir98] R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, London, United Kingdom, second edition, 1998.

[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.

[BKS01] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 421–430, Heidelberg, Germany, April 2001.

[Bou01] T. Bourke. *Server Load Balancing*. O'Reilly & Associates, Sebastopol, CA, USA, 2001.

[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.

[Bra01] R. Braumandl. *Quality of Service and Query Processing in an Information Economy*. PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, Passau, Germany, 2001.

[Bre98] R. Breton. Replication Strategies for High Availability and Disaster Recovery. *IEEE Data Engineering Bulletin*, 21(4):38–43, 1998.

[CAL+02] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung. View Invalidation for Dynamic Content Caching in Multitiered Architectures. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 562–573, Hong Kong, China, August 2002.

[CB00] B. Chidlovskii and U. M. Borghoff. Semantic Caching of Web Queries. *The VLDB Journal*, 9(1):2–17, 2000.

[CCCY02] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The State of the Art in Locally Distributed Web-Server Systems. *ACM Computing Surveys*, 34(2):263–311, 2002.

[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.

[CD99] 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.

[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.

[CIMP03] D. Carlisle, P. Ion, R. Miner, and N. Poppelier. Mathematical Markup Language (MathML) Version 2.0 (Second Edition). `http://www.w3.org/TR/2003/REC-MathML2-20031021`, October 2003. World Wide Web Consortium (W3C), W3C Recommendation.

[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.

[CMSvE98] G. Czajkowski, T. Mayr, P. Seshadri, and T. v. Eicken. Resource Control for Database Extensions. Technical Report TR98-1718, Cornell University, Computer Science Department, Ithaca, NY, USA, 1998.

[Cz02] L. Y. Cao and M. T. Özsu. Evaluation of Strong Consistency Web Caching Techniques. *World Wide Web*, 5(2):95–124, 2002.

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

[Dat] DataSynapse Inc. Homepage. `http://www.datasynapse.com/`.

[DC01] C. Dalton and T. H. Choo. An Operating System Approach to Securing E-Services. *Communications of the ACM (CACM)*, 44(2):58–64, 2001.

[DDH72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, New York, NY, USA, 1972.

[DeW93] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann Publishers, San Mateo, CA, USA, second edition, 1993.

[DFJ+96] S. Dar, M. J. Franklin, B. T. 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, Mumbai (Bombay), India, September 1996.

[DHX+03] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: An Autonomic Computing Environment. In *Proceedings of the International Performance Computing and Communications Conference (IPCCC)*, pages 61–68, Phoenix, AZ, USA, April 2003.

[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.

[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.

[Fra] X. Franc. Qizx/open. `http://www.xfra.net/qizxopen/`.

[GDN+03] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application Specific Data Replication for Edge Services. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 449–460, Budapest, Hungary, May 2003.

[GGF] Global Grid Forum (GGF) Project: Scheduling and Resource Management (SRM). `https://forge.gridforum.org/projects/srm/`.

[Glo] Globus Project Homepage. `http://www.globus.org/`.

[GMSvE98] M. Godfrey, T. Mayr, P. Seshadri, and T. v. Eicken. Secure and Portable Database Extensibility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 390–401, Seattle, WA, USA, June 1998.

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

[GS01] A. K. Ghosh and T. M. Swaminatha. Software Security and Privacy Risks in Mobile E-Commerce. *Communications of the ACM (CACM)*, 44(2):51–57, February 2001.

[GSW96] S. Guo, W. Sun, and M. A. Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM Transactions on Database Systems (TODS)*, 21(2):270–293, 1996.

[HCL+90] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(1):143–160, 1990.

[HD91] H. I. Hsiao and D. J. DeWitt. A Performance Study of Three High Availability Data Replication Strategies. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 18–28, Miami Beach, FL, USA, December 1991.

[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.

[HH76] J. Hartmanis and J. E. Hopcroft. Independence Results in Computer Science. *SIGACT News*, 8(4):13–24, 1976.

[HHO04] H. He, H. Haas, and D. Orchard. Web Services Architecture Usage Scenarios. `http://www.w3.org/TR/ws-arch-scenarios`, February 2004. World Wide Web Consortium (W3C), W3C Note.

[Hor01] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`, 2001.

[HP] HP Utility Data Center. `http://www.hp.com/go/hpudc`.

[IBMa] IBM Director 4.1. `http://www.ibm.com/servers/eserver/xseries/systems_management/director_4.html`.

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

[INST02] A. Iyengar, E. M. Nahum, A. Shaikh, and R. Tewari. Enhancing Web Performance. In *Communication Systems: The State of the Art (IFIP World Computer Congress)*, volume 220 of *IFIP Conference Proceedings*, pages 95–126, Montréal, Québec, Canada, August 2002.

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

[JS94] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 439–450, Santiago de Chile, Chile, September 1994.

[KFD00] D. Kossmann, M. J. Franklin, and G. Drasch. Cache Investment: Integrating Query Optimization and Distributed Data Placement. *ACM Transactions on Database Systems (TODS)*, 25(4):517–558, 2000.

[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.

[KKKK02] 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.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[KMP93] A. Kemper, G. Moerkotte, and K. Peithner. A Blackboard Architecture for Query Optimization in Object Bases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 543–554, Dublin, Ireland, August 1993.

[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.

[KSKK99] M. Keidl, S. Seltzsam, A. Kemper, and N. Krivokapić. Sicherheit in einem Java-basierten verteilten System autonomer Objekte. In *Proceedings of the GI Conference on Database Systems for Business, Technology, and Web (BTW)*, Informatik Aktuell, pages 38–58, Freiburg, Germany, March 1999.

[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.

[KY94] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, Upper Saddle River, NJ, USA, 1994.

[LC99] D. Lee and W. W. Chu. Semantic Caching via Query Matching for Web Sources. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 77–85, Kansas City, MO, USA, November 1999.

[LC01] D. Lee and W. W. Chu. Towards Intelligent Semantic Caching for Web Sources. *Journal of Intelligent Information Systems (JIIS)*, 17(1):23–45, 2001.

[LKK+97] P. C. Lockemann, U. Kölsch, A. Koschel, R. Kramer, R. Nikolai, M. Wallrath, and H.-D. Walter. The Network as a Global Database: Challenges of Interoperability, Proactivity, Interactiveness, Legacy. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 567–574, Athens, Greece, August 1997.

[LKRPM01] L. Li, B. König-Ries, N. Pissinou, and K. Makki. Strategies for Semantic Caching. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, volume 2113 of *Lecture Notes in Computer Science (LNCS)*, pages 284–298, Munich, Germany, September 2001.

[LL02] G. M. Lohman and S. Lightstone. SMART: Making DB2 (More) Autonomic. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 877–879, Hong Kong, China, August 2002.

[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.

[MLR03] V. Markl, G. M. Lohman, and V. Raman. LEO: An Autonomic Query Optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[Mye79] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, NY, USA, 1979.

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

[NKS+02] A. G. Ninan, P. Kulkarni, P. J. Shenoy, K. Ramamritham, and R. Tewari. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 1–12, Honolulu, HI, USA, May 2002.

[Not01] M. Nottingham. SOAP Optimisation Modules: Response Caching. `http://lists.w3.org/Archives/Public/www-ws/2001Aug/att-0000/01-ResponseCache.html`, August 2001.

[Oak98] S. Oaks. *Java Security.* O'Reilly & Associates, Sebastopol, CA, USA, 1998.

[OOW93] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 297–306, Washington, DC, USA, May 1993.

[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).

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, USA, 1985.

[RS01] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley, Reading, MA, USA, 2001.

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

[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.

[Sch03] D. Scheibli. Modular Computing for E-Business Solutions. In *Intel Developer Forum (IDF) Conference Presentations*, February 2003.

[SQL99] Database Language SQL. International Organization for Standardization Document ISO/IEC 9075:1999, 1999.

[SR86] M. Stonebraker and L. A. Rowe. The Design of Postgres. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 340–355, Washington, DC, USA, May 1986.

[Suna] Sun N1. `http://www.sun.com/software/solutions/n1/`.

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

[TPPC02] Transaction Processing Performance Council. TPC Benchmark W Version 1.8, 2002. `http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf`.

[TPPC03] Transaction Processing Performance Council. TPC Benchmark W Version 2.0r, 2003. `http://www.tpc.org/tpcw/spec/TPCWV2.pdf`.

[TR03] D. B. Terry and V. Ramasubramanian. Caching XML Web Services for Mobility. *ACM Queue*, 1(1):70–78, 2003.

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

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II: The New Technologies. Computer Science Press, Rockville, MD, USA, 1989.

[Wei01] G. Weikum. The Web in 2010: Challenges and Opportunities for Database Research. In *Informatics - 10 Years Back. 10 Years Ahead.*, volume 2000 of *Lecture Notes in Computer Science (LNCS)*, pages 1–23, Saarbrücken, Germany, August 2001.

[Wie93] G. Wiederhold. Intelligent Integration of Information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 434–437, Washington, DC, USA, May 1993.

[WLH90] W. K. Wilkinson, P. Lyngbæk, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(1):63–75, 1990.

[WMHZ02] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 20–31, Hong Kong, China, August 2002.

[Xal] Apache XML Project: Xalan-Java. `http://xml.apache.org/xalan-j/`.

[Xer] Apache XML Project: Xerces2 Java Parser. `http://xml.apache.org/xerces2-j/`.

[XMe] XMethods. `http://www.xmethods.org/`.

[YFIV00] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 188–199, Cairo, Egypt, September 2000.

[Zad65] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965.