# Integration of heterogenous services in the Adaptive Services Grid

Harald Böhme and Alexander Saar

Hasso-Plattner-Institute at University of Potsdam,
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{harald.boehme|alexander.saar}@hpi.uni-potsdam.de

**Abstract.** In times of increasing grid oriented computing environments, the integration and orchestration of business services become more and more emergent. This challenge is taken in the context of the EU project *Adapative Services Grid*. As a base for the execution of workflow specifications a distributed execution environment for simple business services and service managment is needed.

This paper shows how such environments can be designed and implemented as a result of the *Adapative Services Grid* research.

## 1 Introduction

The availability of standalone business services is the state of the art in e-commerce solutions. But business consists normally of more complex processes. This leads to the need of integration and orchestration of such business services. In this context the EU project *Adaptive Services Grid* (ASG, http://asg-platform.org) is placed. The goal of ASG is the development of an open software platform for adaptive services discovery, creation, composition, and enactment.

The described goals lead to two major issues that have to be worked on. First the problem of orchestration and choreography of simple services to build bigger business solutions must be solved. For example the combination of a flight booking and a hotel booking service will provide an added value service in this area. The second problem is the execution and management of services during the orchestration.

The paper will focus on the second problem domain, where integration of services based on different communication frameworks, languages and platforms has do be addressed. To tackle these challenges a management and execution environment is defined by the ASG project. This paper will present a basic design approach for such an environment, because the project is in early stages.

The requirements for the management and execution environment are presented in section 2. Section 3 will explain the architectural concepts. Afterwards section 4 gives an implementation proposal. At the end of the paper a short conclusion will be given.

## 2   Requirements

In the e-commerce area the introduction of service oriented architectures (SOA) is one of the major enhancements in the last few years. The switch from plain web browser based service provision to real web services is the most visible effect. But today web services are very restricted in functionality. Thus building business cases will normally use a set of web services, e.g. booking a travel could consist of two services, one for booking the flight and one for booking the hotel. Therefore the orchestration of services is in the focus of many development and research activities.

The complex business service as a result of an orchestration is normally represented as workflow plan, which has to be processed. This task is usual performed by a workflow engine, which will invoke the services step by step. To prevent a vendor lock in the implementation of the workflow engine, the services should be independent from the underlying operating system.

Even if the web service technology is leading in the SOA area, many services were developed with other remote interface technologies. Some well known examples are the *Common Object Request Broker Architecture* (CORBA, [1]) and .NET [2]. Thus the integration of such kind of heterogenous services is a major issue for a business service execution platform.

Today business platforms in e-commerce have to run without major interruptions, because they have to be available over the whole globe and anytime. Therefore the integration of new services can't be done offline. The deployment of new services should be done dynamically, during runtime.

One well known example in e-commerce are shopping platforms like Amazon. In such shopping applications you will fill your shopping cart with products and finally pay. If this simple example should be implemented as a service, stateful behaviour in the service implementation must be possible, because a customer will search products and add them to the shopping cart over multiple service calls. The service has to remember all selected products in your shopping cart. Therefore a realistic business execution platform has to deal with stateful services.

Furthermore not only functional aspects have to be taken into account for the process of orchestration. If the service user is interested in high performance of the service, the orchestration component has to select the services according to the performance requirements of the user. Further on it is imaginable that a user wants to negotiate performance parameters like number of processors or main memory before he invokes the service. Therefore, in addition to the functional properties, the services should provide information about the quality properties they provide.

## 3   Architecture proposal

One aim of the ASG project is the design of a service composition environment based on out of the box software products. This means the execution environment for services should be one of the many already existing application servers.

With this in mind some of the requirements listed in section 2 are already fulfilled. At first with the capability of deployment in application servers, the issue of adding new services to the composition framework is solved at the execution layer.

Second the integration of various remote interface technologies can be simply done by a proxy concept. The proxy has to adapt the remote communication with the service. This allows the uniform handling of services regardless whether the service is running in the ASG execution environment or integrated via the proxies. The programming model for the proxies of external services is quite clear. It was first described by Erich Gamma as the "adapter" pattern [3].

The execution environment has to deal with two main topics:

1. Service invocation and runtime management
2. Service deployment and execution

This leads to a two layer architecture which is presented in detail in the following section.

### 3.1 Two layer architecture

For a heterogeneous design of the execution environment, it was decided to separate it into two layers where the upper layer is assigned with the management issues and the lower layer deals with execution of heterogeneous services. These two layers together build the *Services Grid Infrastructure* (SGI) of ASG.

The two layers allow independent decisions for design and implementation for both layers. This means the implementation of services running in the execution layer is not bound to any implementation language or application platform selected on the users side. All communication with the user is done by the upper layer and thus this communication is decoupled from the technologies of the execution layer. Figure 1 illustrates the whole SGI architecture and its separation into the two layers.

The execution layer is mainly responsible for the execution of the services. The coordination layer provides a well defined interface for the invocation and management of the services. Four sets of functionalities for working with services have been identified (see also figure 1).

**Instantiation** provides the functionality of creating new service instances. A service instance can also be seen as a stateful web service. In SGI a service factory is used to create service instances from a given service implementation.

**Invocation** provides the functionality for starting a service instance with a given set of input parameters. This is the base for the execution of workflows in other parts of ASG. The order of service invocations and the provision of input parameters is left to this parts.

**Deployment** provides the functionality of dynamically integrating and enabling new service and proxy implementations. Its implementation is spread over
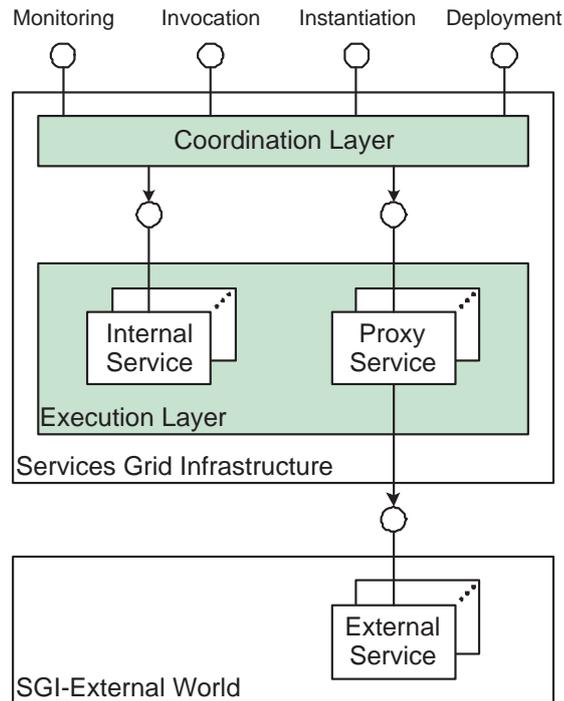
**Fig. 1.** Architectural overview

both layers. The coordination layer holds management information for all deployed services and the execution layer gets the appropriate code for the final execution.

**Monitoring** provides the functionality of observing values, which represent the state of service instances. Beyond the monitoring of functional properties like shopping cart items, this part is also useful for *Quality of Service* (QoS) management.

An important question is about the used interface technology between the two layers as well as the interface technology between SGI and the other parts of ASG. In both cases it was decided to use the "web service" [4] technology.

As leading technology for interworking with services, the web service standard has been chosen as interface technology for ASG in general and for the execution environment in special. The big problem is, that with ordinary web services handling of state is not possible, because they are based on the stateless *Simple Object Access Protocol* [5].

The requirements of functional and non-functional properties leads from the natural stateless web services to some more stateful services. Therefore the *Web Service Resource Framework* (WSRF, [6]) standard was developed in the area of grid computing. WSRF provides a set of useful web service standards that can

be used to provide stateful web services including monitoring, value change notification, lifecycle management and so on. The WSRF web service enhancements which are of interest for SGI are briefly described below.

**WS-Addressing** On the technical side the WS-Addressing (WSA, [7]) framework allows the introduction of implicit parameters for each web service invocation. This technology can be used for different purposes. One well known pattern is the transmission of an implicit "object identity". With this kind of functionality the realization of more stateful web services is quite easy. One only has to choose a fixed parameter name for the state selected and the stateful web service is finished. This means an unique service instance identifier, added to each service invocation message, is used to identify a specific service instance and its state respectively.

The introduction of stateful web services raises the question: When is the state of a stateful web service created and when to destroy it? In the design of the execution environment it was decided to use a factory for the creation of service instances. By calling the create operation on the factory the state of such a service instance comes into live.

**WS-ResourceLifetime** To be symmetric an explicit termination of service instances, and thus deletion of its state, was also defined. With the set of WS-ResourceLifetime (WSRL, [8]) operations a web service user is able to manage the lifetime of resources bound to a web service. In our context the state of a service instance is such a resource. The basic principle in WSRL is a timer concept. After expiration of the timer the bound resources are not accessible anymore. Additionally WSRL allows you to explicitly terminate resources by defining the expiration time as "now".

**WS-ResourceProperties** The WS-ResourceProperties (WSRP, [9]) standard defines operations for querying and updating the properties of a stateful web service, or in our context of a service instance. In a service composition environment like ASG, this mechanism can be used for monitoring of functional and non-functional properties.

Furthermore the orchestration of multiple services to a composed service has to fulfill some QoS characteristics. This may start with a negotiation of some QoS parameters, but has to continue with the observation of the QoS parameters while executing the services. The operations from WSRP are used to monitor a given set of QoS parameters on running and also terminated service instances.

## 3.2 Performance issues

With the usage of the mentioned WSRF standards many of the requirements for a service composition environment can be addressed. The two layer design decouples the execution of services and service proxies from the coordination layer and enables some further functionalities.

For example the coordination layer is enabled to select between multiple instances of application servers to perform the invocations of a service. The service factory is responsible for making the service implementation available at the several used grid nodes during the creation of a service instance. This hides the actual structure of the grid and provides grid location transparency to the users of SGI. Furthermore some degree of load balancing is achieved with this and negotiation of execution platform properties can be provided.

In the next section we will have a closer look to some application platforms and technologies which seem to be appropriate candidates for the implementation of SGI. Moreover we will see how they fit to the requirements mentioned before.

## 4 Implementation out of the Box

This section first describes the target platform requirements for an implementation of SGI as well as the platforms and frameworks that were analysed as candidates for a fast, easy and standards based implementation.

Second the decision for a particular platform as well as the implementation issues of the SGI concepts described in section 3 are explained.

### 4.1 Target platform requirements

The requirements for SGI - and therefore also for the target platform - are derived from the previously described design and identified as follows.

**Dynamic deployment:** Dynamic deployment must be possible to realize the concepts of dynamic service creation and deployment respectively. Moreover it allows an easy realization of grid location transparency, load balancing and negotiated quality of service (QoS) parameters, because the factory can make deployment decisions at runtime to achieve the mentioned goals.

**Web service interface:** The platform for the execution environment should support web service interfaces. A fully WSRF compliant SOAP interface is not required, because not all WSRF standards are needed. Anyway it should be possible to extend the web service interface easily to integrate the needed standards.

**OS independence:** According to the grid paradigm of shared resources between multiple partners over open networks, the execution platform should be independent of the underlying operating system and hardware configuration.

**Stateful services:** In order to support functional and non-functional properties, the target platform must be able to handle service properties for negotiation and monitoring purposes.

### 4.2  Target platform evaluation

The considered execution platform candidates are the .NET platform, the OSGi framework, Apache Axis and Enterprise Java Beans (EJB). Below these are briefly introduced and analysed with regard to the identified requirements.

**.NET platform** The .NET platform [2] developed by Microsoft provides a virtual machine as execution platform for the standardized intermediate language (IL) as well as a set of common class libraries, e.g. for XML processing, networking, user interfaces, IO operations, etc. Thus it can be best compared with the *Java 2 Standard Edition* (J2SE, [10]).

The .NET framework supports hosting of web services via its *Active Server Pages* (ASP.NET) technology. Thereto an Internet Information Services (IIS) container is needed which already supports dynamic deployment. A full WSRF compliant interface can be provided through use of the WSRF.NET [11] grid service platform.

In spite of the possibility of full WSRF compliant interfaces including WSRP for stateful web services, the .NET framework and IIS are limited to the Windows operation system these days. There are some activities for providing web services hosted on Apache webservers and the alternative .NET implementation MONO, but this approach is not very sophisticated.

**Open Services Gateway Initiative** The *Open Services Gateway Initiative* (OSGi, [12]) Alliance standardizes an open service platform for the delivery and management of multiple applications and services for all types of networked devices in home, vehicle, mobile and other environments.

By using the OSGi Service Platform networked devices (embedded or servers) get the capability to manage the life cycle of software components in the device from anywhere in the network. The core part of the OSGi Service Platform is the OSGi framework that offers an execution environment layer (e.g. J2SE, J2ME), a modules layer that defines advanced class loading policies, and a service registry. Furthermore the OSGi Alliance has defined many standard services, e.g. for logging, networking, administration, etc.

As mentioned before service life cycle management in OSGi can be done from anywhere in the network including installation, updating and so on. Therefore dynamic deployment should be easy implementable. The possibility of J2SE as execution layer enables OS independence and thus an universal use in heterogenous grid environments. But even if some OSGi implementations support web service based interfaces, this is not yet part of the OSGi standard.

**Apache Axis** The Axis project [13] from the *Apache Software Foundation* (ASF) is essentially an open-source Java implementation of a SOAP engine. Axis includes a simple stand-alone server, a server which plugs into servlet engines such as Tomcat, extensive support for the *Web Service Description Language*

(WSDL, [14]) and much more auxiliary functionality like stub generation and monitoring.

A further Axis based project of the ASF is the incubator project Muse [1]. Muse provides an implementation of the *Management using Web Services* (MuWS, [15]) specification which is based on a subset of the WSRF standards family as well as some further web service standards.

Because Axis is Java-based OS independence is guaranteed. Stateful web services are also provided through WSRP, but dynamic deployment is not supported.

**Enterprise Java Beans** The actual release 2.1 of the *Enterprise Java Beans* (EJB, [16]) standard is part of Sun's *Java 2 Enterprise Edition* (J2EE, [17]), which is currently available in version 1.4. It provides a server-side component architecture and enables development of distributed, transactional, secure and portable applications based on Java.

Since version 2.1 the EJB standard defines standardized web services interfaces (JSR109, [18]) based on the *Java API for XML-based Remote Procedure Call* (JAX-RPC, [19]) specification[2]. Advanced processing of SOAP extensions like WSA can be implemented through JAX-RPC handlers. Those can be added as interceptors between the EJB container and the invoked EJB.

Most J2EE application servers that implement the EJB standard support dynamic deployment very well and stateful services can be achieved with stateful session beans. But today stateful session beans can only be accessed through *Remote Method Invocation* (RMI, [20]). The standard defines web service interfaces only for stateless session beans, according to the stateless character of SOAP. Along with this WSRF support is also missing.

### 4.3 Implementation proposal

After analyzing various platforms as possible candidates for the implementation of SGI, it seems that there is no ideal candidate that supports all necessary features. At last the decision was made for a combination of Axis and EJB, because EJB standardizes web service interfaces and Axis provides advanced message processing features. Furthermore most J2EE application servers are platform independent and support dynamic deployment.

The following sections will explain in detail how the missing requirements can be implemented through this combination and how the SGI architecture can be implemented with EJB.

**Coordination layer** Most of the infrastructure services of the SGI coordination layer can be implemented as stateless session beans or POJO based servlets

---

[1] The Apache Muse project can be found at: http://ws.apache.org.

[2] Note that in J2EE 1.4 it's also possible to register plain Java objects (POJO) as servlet with a web service interface.

providing web service interfaces. In detail the factory service can be implemented as session bean while the deployment service should better be implemented as servlet, because it has to extract and process deployment units like JAR or WAR files and thus it needs access to the hard disk[3].

Beyond it is proposed to implement the invocation service Axis based, because Axis offers easy and full access to the messages. As described in section 4.2 such an implementation can be plugged into servlet engines provided by most J2EE application servers.

As outlined in section 3 the invocation service is intended to forward service calls to dedicated grid nodes. The message processing functionality of Axis allows an easy realization of the necessary WSRF standards (e.g. WSRP) as well as the concepts of grid location transparency, load balancing and error handling. That means the invocation service can map WSRF standards based messages (for example WSRP requests) to appropriate interfaces of the underlying execution platforms. Users only invoke web service interfaces without any knowledge about the characteristics of those platforms.

Specific service instances are identified through the unique service instance identifier which is added as a WSA rference property to the SOAP header. Handling different types of service instances (e.g. RMI servers or web services) is part of the invocation service responsibilities. For example for WSRF based platforms the service instance identifier is forwarded as WSA reference property to the execution layer while in RMI exactly one stateful instance of the server can be used. This approach makes integration of further technologies like RMI or .NET based platforms simpler and prevents expensive modification of them.

Any information about grid nodes, service implementations or service instances must be shared between the infrastructure services, for example after choosing a target grid node for execution, the service factory has to get a service implementation and deploy it on the selected node. It is obvious to store those information in a central database, called SGI storage. For increasing the scalability of SGI, this database can be seen as a logical central storage realized on top of a distributed database system.

**Execution layer** The service implementation approach for the execution layer is based on EJB and an assigned JAX-RPC handler. The restriction to stateless session beans demands an explicit handling of internal state information in the bean implementation. The execution layer therefore introduces the concept of a logical instance. A factory operation on the coordination layer leads to the creation of a logical instance, which is identified with one service instance identifier.

It may happen that multiple bean instances of the implementation run at the same time, for example when a monitoring request is performed while a synchronous operation is running (see figure 2). All these bean instances, which are triggered through the same service endpoint, form a common logical instance.

---

[3] In contrast to servlets file I/O operations are forbidden in EJBs.
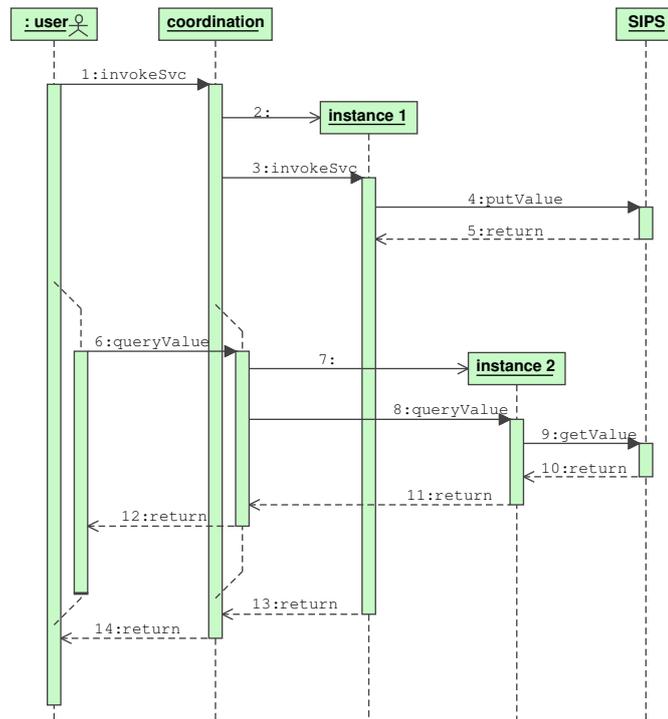
**Fig. 2.** Service invocation procedure

The JAX-RPC handler is transparently plugged between the EJB container and the invoked bean that implements the service. It is responsible for extracting WSA information (e.g. service instance identifier) and passing it to the bean by using the thread local storage concepts of Java[4].

The service implementation bean can use the service instance indentifier together with some property identifiers to store a set of properties in a database, called *Service Instance Property Storage* (SIPS). Like shown in figure 2 those properties are queried over special interfaces, if a WRSP request was received at the coordination layer.

If a service bean acts as a proxy for an external service (e.g. a RMI service), it should provide a check method for testing the availability of the external service. However not all external services provide such a functionality.

**Open issues** In the previous sections it has been shown how many of the concepts from section 3 can be implemented with a combination of EJB and Axis.

---

[4] Notice that this limits application of this concepts to environments where a new thread is created for each service call.

But still some issues are open today. For example intermediate result handling has no assigned standard.

It is also unclear what should happen if a service instance becomes invalid while a service call is running. Maybe this can lead to a call of a cancel method and a deletion of state information in SIPS. In case that the cancel call on the bean is not successful or not available, the coordination layer may start an escalation procedure for the removal of the instance. The availability of such mechanisms in the EJB container or the surrounding operating system is currently under research. Anyway it must be ensured that the specified invalidation behaviour at the coordination layer remains stable.

## 5 Related work

Even though the integration of heterogeneous platforms is a well known topic in the area of computational research, the concepts of dynamic in SOA environments adaptation and interaction are a relatively new research field. Thus there are not much solutions for the challenges described in this paper.

One solution to deal with this kind of challenges is the usage of component platforms, which can handle the integration of implementation in different implementation languages. The Object Management Group (OMG) has extend their Object model to the CORBA Component Model (CCM, [21]). This includes the possibility of different implementation for software components. With the CCM technology, integration of functionality in different implementation languages is possible, but only for component-oriented applications. The term service has here only the meaning of object services from CORBA. Hence real service integration, like proposed by SOA, is not supported. CCM may be used as technological base for the execution layer in a service integration environment (see section 4), but the new services have to be defined in terms of CCM then. So CCM for it's own is not able to meet all requirements.

Oracle has introduced solutions for heterogeneous services for enabling its next generation of open gateway products. The "heterogeneous services" platform from Oracle provide a common architecture and administration mechanisms for heterogeneous access facilities to non-Oracle systems.

Oracle's heterogeneous services consist of two service types, SQL services and procedural services. Both service types allow the integration of non-Oracle systems into Oracle transactions and sessions. The SQL services are used to transform Oracle SQL statements into other SQL dialects and to map non-Oracle datatypes onto Oracle datatypes. By using the procedural services, non-Oracle messaging or queuing systems can be accessed from Oracle software. For example, Oracle systems can start authenticated sessions at and coordinate distributed transaction with non-Oracle systems.

The idea of Oracle's heterogeneous services does not really cover the requirements for dynamic service-oriented adaptation and interaction, but they have one thing in common with the approach presented in this paper. Like the coordination layer described in section 3, Oracle provide additional software layers,

called agents, which are responsible for the management of interaction with the non-Oracle systems. In contrast to the mentioned coordination layer, which provides centralized access to various heterogeneous platforms through standardized web service interfaces, the Oracle approach requires an agent for each non-Oracle system that should be integrated.

## 6 Conclusion

The authors have shown that the integration and composition of heterogenous services become more and more important today. It has shown how an execution environment for those purposes can be designed and implemented.

The layered architecture ensures the decoupling of service invocation and managment on the one hand, and service execution on the other hand. Moreover it allows the simple integration of additional heterogenous platforms.

With the full featured and mature EJB 2.1 standard and its several implementations we found an appropriate application platform which already provides many of the required features. Furthermore we demonstrated that the missing functionality can be implemented in a standard conform manner. The implementation does not consume much effort and is portable over most EJB conformant products. However the further development will not be focused on EJB as application platform only.

Finally it is safe to say that at the current stage of the ASG (Adaptive Service Grid) project not all features of SGI (Service Grid Infrastructure) are completely designed and implemented, but the proposed approach seems to be promising.

## 7 Acknowledgments

## References

1. Inc., O.M.G.: The Common Object Request Broker : Architecture and Specification. Revision 2.3 (1999)
2. Thuan Thai, Hoang Q. Lam: .NET Framework Essentials 2nd Edition. O Reilly (2002)
3. Erich Gamma, Richard Helm, R.J., Vlissides, J.: Entwurfsmuster. Addison-Wesley (1995)
4. W3C: Web services activity. http://www.w3.org/2002/ws/ (2004)
5. W3C — World Wide Web Consortium: Simple Object Access Protocol. (2003)
6. Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Sedukhin, I., Snelling, D., Storey, T., Vambenepe, W., Weerawarana, S.: Modeling stateful resources with web services. IBM DeveloperWorks Whitepaper (2004)
7. W3C — World Wide Web Consortium: Web Services Addressing. (2004)

8. OASIS Open: Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). (2004)
9. OASIS Open: Web Services Resource Properties 1.2 (WS-ResourceProperties). (2004)
10. Zukowski, J.: Mastering Java 2, J2SE 1.4. Sybex Inc (2002)
11. Wasson, G., Morgan, M., Humphrey, M., Beekwilder, N.: An early evaluation of WSRF and WS-notification via WSRF.NET (2004)
12. Osgi: OSGi Service Platform (Release 2). IOS Press (2002)
13. Wang, D., Bayer, T., Frotscher, T., Teufel, M.: Java Web Services mit Apache Axis. Software & Support Verlag (2004)
14. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR (2005)
15. OASIS: Management using web services (2005)
16. Microsystems, S.: Enterprise Java Beans (TM) Specification Version 2.1 (2002)
17. Haugland, S., Cade, M., Orapallo, A.: J2EE 1.4 : The Big Picture. Prentice Hall PTR (2004)
18. Knutson, J., Kreger, H.: Web Services for J2EE, Version 1.0 (2002)
19. Singh, I., Brydon, S., Murray, G., Ramachandran, V., Violleau, T., Stearns, B.: Designing Web Services with the J2EE(TM) 1.4 Platform : JAX-RPC, SOAP, and XML Technologies. Addison-Wesley (2004)
20. Microsystems, S.: Java Remote Method Invocation Specification, Revision 1.5, JDK 1.2 edition (1998)
21. Neubauer, B., Ritter, T., Stoinski, F.: CORBA Komponenten : Effektives Software-Design und Programmierung. Springer (2004)