Web Services:
Distributed Applications without Limits
- An Outline -

Frank Leymann

IBM Software Group
Schoenaicherstr. 220
71032 Böblingen
Germany
LEY1@de.ibm.com

Abstract: Web services technology is all about distributed computing. There is no fundamentally new basic concept behind this and related technologies. What is really new is the reach of Web services and its ubiquitous support by literally all major vendors. Most likely, heterogeneity will at the end no longer be an obstruction for distributed applications. This will have impact on application architectures, middleware, as well as the way in which people will think about computing and businesses use computing resources. We sketch these impacts as well as some exemplary research work to be done to actually build the outline environment.

1 Introduction

Since the advent of Web services about two years ago most software vendors have embraced this technology and support it in their products. The spectrum of products supporting Web services reach from database systems over application servers over standard applications to office suites; corresponding support in tools is already available too. And Web services became an integral aspect of modern system architectures (e.g. [13]).

In a nutshell, a Web service is a virtual component that can be accessed via multiple formats and protocols. Such a component can be located anywhere in the network, e.g. on a machine on a different continent or within a thread in the same operating system process. Consequently, the environment for Web services is heterogeneous and distributed from the outset. Furthermore, Web services support a service-oriented architecture in which requestors can discover Web services and dynamically bind to them. But the primary focus of Web service technology is communication between Web services themselves, i.e. requestors are again Web services. Thus, to make the corresponding heterogeneous, distributed, and dynamic discovery-based environment work in practice, interoperability is key and standards are a must. A whole stack of standards has already been proposed (e.g. WSDL [11], SOAP [5], UDDI [3], and WS-Security [23]) and others will follow (see for example the roadmaps in [10] and [21]). Based on these standards a set of interoperability profiles will be published that describe artefacts from collections

Invited Talk & Joint Opening Speech at BTW2003 and KiVS2003

In: Proceedings Database Systems For Business, Technology and Web BTW 2003 (Leipzig, Germany, February 26 – 28, 2003), Springer, 2003.

of Web services standards and its recommended collective usage to ensure interoperability across platforms and languages (e.g. [1]). We describe the overall Web service environment and underlying basic concepts in section 2.

Grid technology [15] is about to evolve towards a "virtualisation layer" for hosting Web services ([16], [42]). Corresponding environments are under implementation, for example for Java [39]. This will enable what has been called recently "utility computing" or "on demand computing" [20]. Section 3 sketches this development.

Applications in this environment will consist of two parts, namely collections of individual and autonomic Web services (i.e. components) and aggregation specifications defined as business processes [12]. This will make the two-level programming model (e.g. [44], [29]) pervasive and will even allow involving human beings in applications. The corresponding application structure is outlined in section 4.

Finally, Web services also need to be aggregated in a less structured manner: Corresponding aggregation models for Web services appear (e.g. [8], [31], [41]) that allow building unstructured collections of Web services. Section 5 sketches the basics.

We conclude in chapter 6 and present the draft of a high-level middleware stack that supports the execution of this kind of applications.

2 Virtual Components

Web service technology makes functions available independent of many aspects of the proper implementation of the Web Service: A requestor has no need to know the programming model chosen to implement a Web service, i.e. whether the Web service is implemented in procedural or object oriented manner, for example. The programming language used to implement a Web service is completely irrelevant for a requestor. It doesn't matter whether the Web service is based on functions of a monolithic application system or whether it is build as a component, and if it is a component what the underlying component model is (e.g. J2EE, .NET). Any specific formats and protocols assumed by the Web service for direct communication is irrelevant for a requestor, i.e. it is hidden whether the implementation of the Web service expects ASCII files or Java objects, or whether it is invoked via a local call, an RPC or via a message queue, for example.

The concept of a WSDL *port type* is used to define what functions a Web service provides, i.e. a port type specifies the interface of a Web service. Different WSDL *bindings* can be used to specify how these functions can be accessed via different formats and protocols, e.g. via SOAP over JMS, or via Java objects via method call. And a WSDL *port* defines an actual endpoint where these functions can be accessed according to a certain format and protocol, e.g. a queue name, or a class name and JNDI name. In this sense, a Web service is a *virtual component* that can be implemented in many different ways, e.g. by real components or by any other piece of executable code (see Figure 1). Especially, a Web service is not at all coupled with any kind of Web technology; because of this we will often simply use the term *service* instead of the Web service and we will use both terms interchangeably.

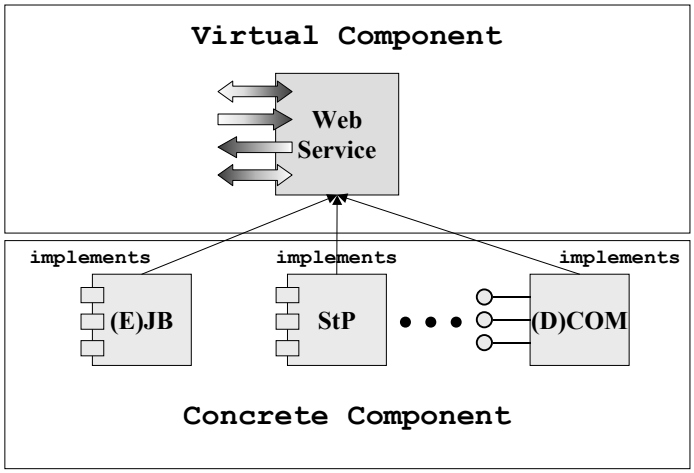

Figure 1 - Web Service as Virtual Component

2.1 Invocation

A user of a service should not be aware of the concrete implementation model chosen to realize the service: Whether the service is implemented as an EJB or a stored procedure or something else should be hidden as far as possible from the user. Thus, the user should be given a consistent “programming model” when dealing with services of different kinds. For this purpose, the environment of the user should provide features to deal with services of any kind in a manner specific to the environment and that appears seamless to the user.

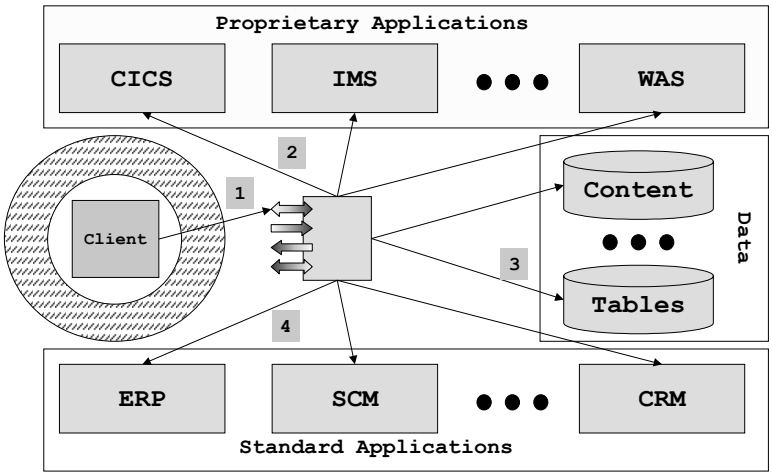

Figure 2 - Accessing Web Services

For example, a J2EE programmer should deal with Web services in a J2EE "style". In [26] we present a J2EE building block called WSIF (Web Service Invocation Framework – represented by the annular area in Figure 2) that exactly facilitates the latter; other environments may provide similar building blocks. In Figure 2, the client accesses a Web service (❶) in the programming model of its hosting environment (e.g. in Java based on WSIF); it doesn't even know that such different executables like a program in a TP monitor (❷), a table via an SQL statement (❸), or an ERP system (❹) may actually implement the Web service.

2.2 Lifecycle

A service can be statefull or stateless. For our discussion it is not important whether state is introduced via persistent instances or via session-like interactions. It is more important for our discussion whether or not the fact that a service is statefull or not is hidden from or visible to its clients: This has impact on the client programming model, i.e. whether a client has to explicitly manage the lifecycle of a service or not. When services are dynamically discovered, having to distinguish between statefull and stateless services causes complexity. Today, as a matter of fact, different application areas follow one approach or the other: In an OGSA Grid environment [42] statefull services are explicitly dealt with, while a BPEL business process environment [12] implicitly manages the statefullness of a service on behalf of a client.

At the level of details sufficient for us, OGSA uses an explicit factory-based approach to deal with the lifecycle of a Web service: A client uses a factory to create "an instance" of a particular kind of service. The client can then explicitly manage the destruction of such an instance, or it can be left to the Grid environment. In the latter case, a client registers its interest in the instance for a particular period of time (which can be extended). When no client is any longer interested in a given instance it can be destructed.

BPEL facilitates the implicit management of the lifecycle of an instance of a service via correlation identifiers embedded in messages: Application data exchanged with a service is assumed to carry enough information to identify a particular instance of a service. The state of a service is described via a process specification in BPEL. Depending on the actual state a service is in an incoming message results either in the automatic creation of an instance of a service, or the message is automatically routed to the appropriate existing instance. Finally, instances are automatically destructed when they reach their "final state".

2.3 Policies

Services need to describe their capabilities and requirements to their environment and potential users. A collection of capabilities and requirements is referred to as a *policy* [24]. A policy may express such diverse characteristics as transactionality, security, response time, pricing, etc. For example, a policy of a service may specify that all interactions must be invoked under transaction protection, that incoming messages have to be encrypted, that outgoing messages will be signed, that responses may only be accepted

within 5 seconds, and that certain operations are subject to a fee to be paid via credit card by the invoker.

Since policies might get quite complex they should be reusable. For this purpose, a policy can be specified as a separate document. Such a document can be associated with (constituents of) a Web service via an *attachment* [25]. Basically, an attachment consists of both, a policy and a subject the policy applies to ("resource"). Such subjects include port types, operations, messages, and also endpoints, i.e. individual ports or Web services, respectively. Attachments can be specified as follows (see Figure 3):

- Policies can be referenced out of the WSDL definitions of subjects. This method is suited to attach policies at the time when Web service resources are specified.
- Web services resources that are already deployed can be associated with policies by simply pointing to these resources and to the policies to be applied. Pointing to resources can be done based on *domain expressions* that describe the subjects and that have to be resolved in order to find the resources characterized by the policies. This method is especially suited to attach policies to existing resources.
- Finally, a policy can be registered itself in UDDI (as tModels). It can be associated with a UDDI business service (as key in a category bag).

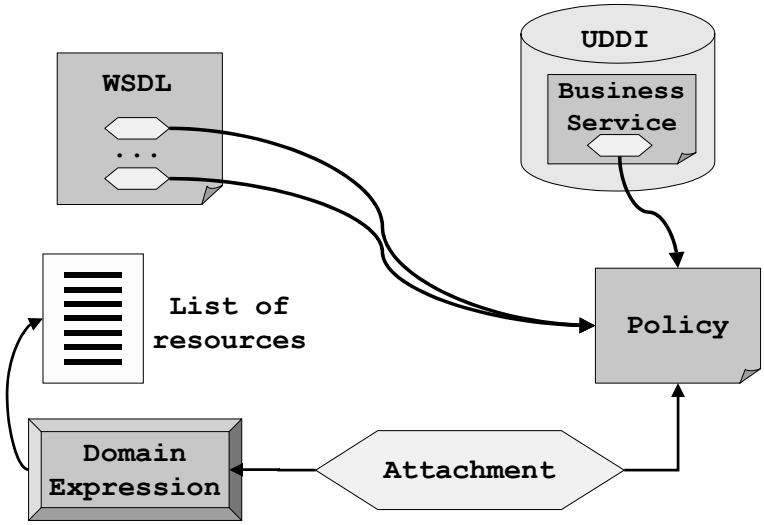

Figure 3 - Attaching Policies to Services

Service level agreements [34] can make use of policies and policy attachments. They do specify characteristics bilaterally agreed to in advance between the provider of a Web service and particular users. A service level agreement specifies aspects like committed

quality of services (like availability and average response time), payment methods for calling on a Web service, fees to be paid when service levels are not met etc.

2.4 Service Bus

Web service technology enables a new kind of architecture for composing applications referred to as *service oriented architecture* (SOA – see [7]). In SOA, services are registered in a service directory (e.g. in UDDI). Requestors find services they are interested in by enquiring service directories. The information they retrieve from a directory suffices to bind to a service and use it (see Figure 4).

When a service provider publishes a service in a service directory he specifies technical information about the service as well as business relevant information. Technical information about a service includes its interfaces, supported bindings, and endpoint information (e.g. the corresponding WSDL definitions). Business relevant information about a service falls into two categories: One category contains information about the suitability of a service from a functional perspective; the other category contains information about the suitability of a service from an operational perspective. The first category helps to understand whether a service is instrumental in achieving a business goal, e.g. buying a certain kind of sheet metal that is available within a certain period of time at a given price. Information provided are semantic descriptions about the kind of service facilitated by each of its interfaces, information about the service provider itself etc. The second category helps to understand whether a service satisfies the business policies of the requestor, e.g. all data are exchanged in an encrypted manner and are deleted once the trade is settled, messages are exchanged via reliable protocols, and payment is can be done once a month collectively for all orders. Information provided in this category includes payment methods, charging models, quality of services supported. The policy mechanism is expected to be used to describe this kind of information. Finally, all this information should be understandable by large communities, both, people as well as programs; it is expected that ontologies will play a major role in this area [14].

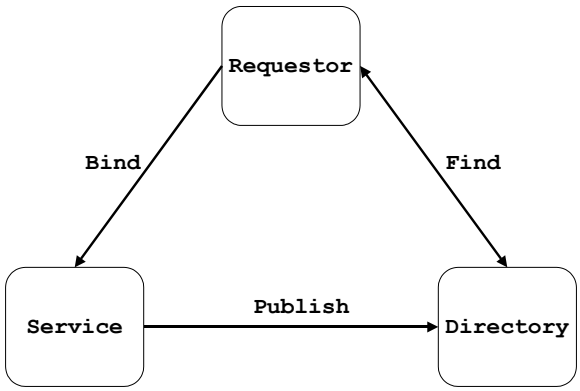

Figure 4- Service Oriented Architecture

Underlying SOA, there are really two distinguishing features: First, a requestor finds suitable services mainly based on queries in business terms (in contrast to technical terms). Second, the infrastructure hides as many technicalities as possible from a requestor. For example, a requestor specifies that he wants to analyze a gene based on a particular algorithm, and that he is wants to exchange all of the corresponding data encrypted. The infrastructure should find a service provider that matches the requestor's criteria and handle the corresponding request automatically on behalf of the requestor.

In Figure 5, this infrastructure is called *service bus*. The service bus receives the request and peals off the declarative description of the service required (❶). The description contains both, the business goals as well as the business policies of the requestor, and this description is used to derive the set of matching services offered by various service providers SP^j (❷). From a requestor's perspective, all qualified services are equivalent; i.e. the set of qualified services represent the *virtual service* (❸) described by the requestor by his request. If more than one service has qualified the service bus will decide on one of them (❹); this decision will be based on overall environmental properties like actual workload at the service provider side, average response time etc (e.g. measured or based on service level agreements with the service providers). Finally (❺), the service bus will bind to the service selected, pass the request message proper to it, and deliver the response to the requestor. Note that during step ❺ the invocation component sketched in section 2.1 is involved.

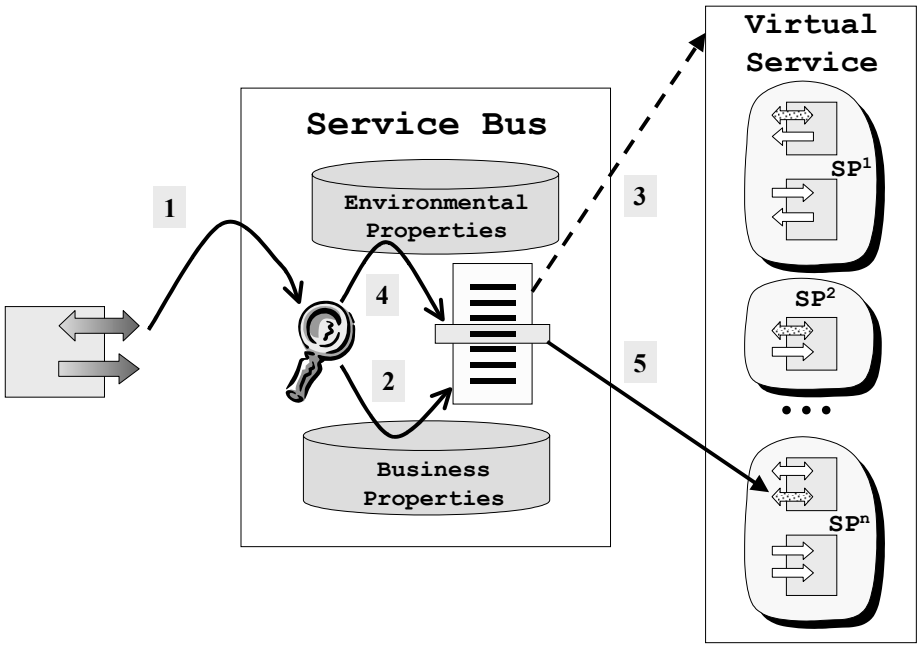

Figure 5 - Service Bus for Virtualizing Services

2.5 A Clarification

It should be clear until now that the sometimes-heard belief, Web service technology is all about SOAP, is erroneous. As shown above, Web Service technology is about SOA, a certain architectural style, which is far more than just SOAP: SOAP is primarily one particular wire-format used to exchange data as well as a set of conventions about how to appropriately process SOAP messages. The acronyms are close, but the goals are at different scale.

2.6 Sample Work to Be Done

Today, the component model underlying Web services is relatively simple. To allow more complex usage patterns ([19], [40]) work must be undertaken to define a more complex component model for Web services. For example, what mechanisms for component aggregation are required or desirable (see also section 5)? What are their advantages or disadvantages in a Web service world?

According to SOA Web services can be dynamically discovered and used. Today, the difference between statefull and stateless Web services is visible. Can and should the environment hide the difference and allow for a single client programming model? What is the impact of this programming model on the service bus?

Policies play a key role in the discovery of Web services. Often, policies are added to a Web service in an incremental manner. What are efficient algorithms to combine multiple policies into a single policy that describes a service or a request? A service as well as a request is decorated by a policy; how is matchmaking of policies done efficiently?

3 Virtual Operational Environments

The service bus introduced above virtualizes services: As long as a service qualifies under a request the service bus has the liberty to target the request to it. In doing so, the service bus can optimise the execution of a single request having the optimal exploitation of the overall environment in mind. It will use algorithms and mechanisms from scheduling, workload management etc that apply to the heterogeneous and distributed environment of Web services.

3.1 Grid Services

Middleware for scientific computing with similar goals has already been developed in the Grid computing area [15]. It thus seems only natural to bring the area of Grid computing and Web services together: [16] outlines an architecture for such a combined environment called Open Grid Services Architecture (OGSA). The most fundamental aspects of the special kind of Web services, called *Grid Services* that are hosted in such a combined environment are under specification (see [42]).

In order to become a Grid services, a Web service has to support a set of pre-defined interfaces and has to comply with some conventions. The interfaces to be supported facilitate the discovery, creation, and lifetime management of services; they further facilitate a notification mechanism to especially enable the manageability of services. The conventions deal primarily with naming services. Based on these interfaces and conventions a standard semantics for interacting with a Grid service is defined: How services are created, how their lifetime is determined, how to invoke functions of a service etc.

It is expected that many different environments, especially application server environments like J2EE [22] or .NET [2] will evolve to support Grid services. This would mean that the application server might provide a special container hosting these services or that existing containers are modified to support the semantics of these services (*Grid service container*). [39] describes the design of such a container based on both, native Java as well as on J2EE.

Such a (new or modified) container specifies the interface defining the interactions between the container and an implementation of a Grid service such that the implemented service appears to a requestor as a Grid service. As of today (year end 2002), this interface is not standardized; a corresponding standard would allow creating Grid services that are portable at least between homogeneous environments (e.g. J2EE compliant application servers). Nevertheless, requestors that use a Grid service based on the OGSA standard specified in [42] would be independent of the actual environment that hosts the Grid service used.

3.2 Grid Services Environment Stack

Based on [38], Figure 6 depicts the stack building the overall environment for applications of Grid services. At the bottom, it shows a Grid service container based on an environment like an application server; the container provides the functions discussed before. But the overall environment might consist of many different Grid service containers that are hosted on different autonomous and heterogeneous application servers. Thus, clustering capabilities are needed to “federate” the different Grid service containers resulting in a virtual environment for scalability and resource sharing. Also, such a virtual environment has to support distributed and heterogeneous problem determination and logging, the association of policies with Grid services as a base for request scheduling etc. The corresponding functions are referred to a meta-operating system services.

Often, collections of Grid services are needed to perform more complex functions that are not offered by individual services (see also section 5). Capabilities for managing such collections of services as well as making them jointly accessible are shown as a separate building block referred to as domain services. For example, domain services allow that individual instances of a particular Grid service type may join or leave a collection. Domain services also include functions for provisioning such collections to individual requestors.

At the top layer functions are shown that represent various autonomic services of the Grid: For example, Grid-wide workload management that enable a broad range of

mechanisms for scheduling requests in the Grid reaching from simple round-robin schedulers to policy-based meta-schedulers in hierarchical Grid topologies (see [4], [37]) enhancing overall availability and scalability within the Grid. Also, functions enabling utility computing (see next section) are at this layer.

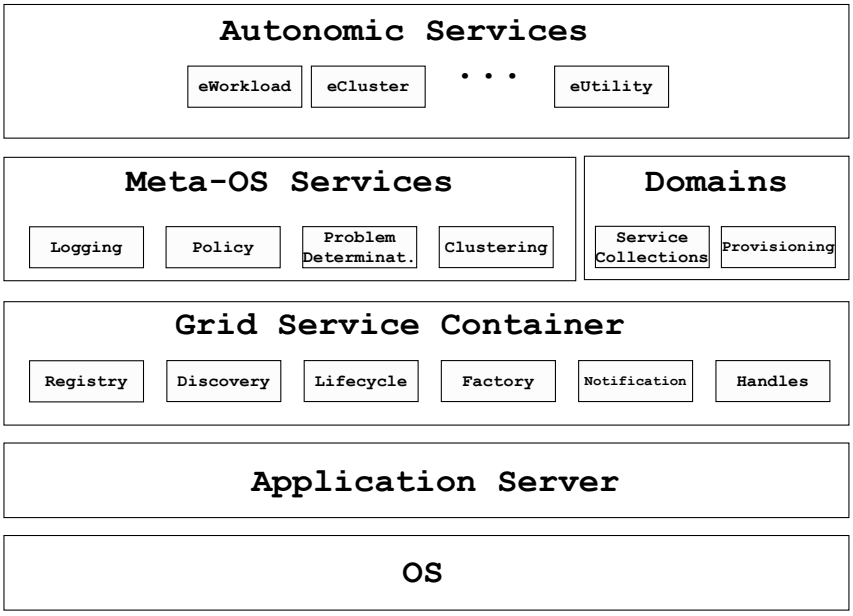

Figure 6 - The Grid Services Stack

3.3. On Demand Computing

Finally, such an environment will enable a new computing model called *on-demand computing* [20]. In a nutshell, this term refers to the ubiquitous availability of compute resources whenever needed and wherever needed. This bares the potential to turn computing into a public utility like water, power, gas, and telephone connections – which is why this model is also referred to as *utility computing*.

An important step on this path is represented by the concept of a hosted e-utility. A hosted *e-utility* is a collection of application-related services (both, hardware as well as all required software) that is made available by a service provider to a requestor on demand based on particular service level agreements for a certain fee. For example, a requestor wants to analyze new genomic data and needs for this purpose a set of certain algorithms, large amount of temporary storage, a set of servers to provide the corresponding compute power, as well a high-bandwidth connections to the Internet for access to public genomic data. A service provider can provide all of this as a collection of Grid services.

For this purpose, the service provider will make use of the Grid services stack sketched in the section before. For example, the required collection of services will be managed by the collection services. The collection will be assembled based on business rules and business processes depending on the ordered quality of services and negotiated service level agreements; for this purpose, the eUtility service of the autonomic layer can exploit workflow technology. Provisioning services are used to reserve the necessary resources to meet the service level agreement for the time period ordered. Note the relation between hosted e-utilities and application service providers (ASPs).

3.4 Sample Work to Be Done

It is obvious that there is a lot of work to be done to establish Grid computing and further on-demand computing as a broadly accepted model in practice. The spectrum of work reaches from low level technical work like specifying the agreed upon interfaces between the Grid service container and Grid service implementations such that these implementations become portable, over theoretical work on meta-schedulers, to business-related work on payment models, for example.

Finally, Web services and Grid services will further have to converge: It has to be investigated which properties currently specified as characteristics of Grid services do make sense in the more broader context of Web services, and which properties do only make sense in the more specific context of on-demand computing – if there are any such properties at all.

Note: We do not distinguish between Grid services and Web services in what follows and will often simply talk about services.

4 Application Structure

Services are either fine grained or coarse grained. From a requestor’s perspective, a fine grained service achieves a business goal based on a single interaction, while a coarse grained service typically requires multiple interactions to achieve a business goal. Because a single interaction with a fine grained service suffices, a fine grained service typically does not reveal any of its inner structure, i.e. it is opaque hiding its implementation details. In contrast to this, a coarse grained service does reveal implementation details, especially the set of interactions required as well as their order, i.e. it is transparent making some of its inner structure visible to a requestor. The implementation details revealed by a coarse grained service describe its potential message exchange with the outside world, i.e. business rules that specify in which order and under which conditions which messages are sent to or expected from the requestor and perhaps other third party Web services. These details are important because it allows a requestor to determine whether he can interact with a particular service at all, for example.

4.1 Two-Level Programming Paradigm

In a Web services world actual messages are sent to ports via their corresponding operations. Thus, at the type level a potential message exchange can be specified by defining

the potential order in which operations of port types are used and under which conditions. As depicted in Figure 7 this is the same as specifying a business process or a workflow, respectively, the activities of which are realized by operations of port types (see [30]). Especially, a coarse grained service appears to be composed of the corresponding services, and consequently coarse grained services are also referred to as *composite services*. Vice versa, fine grained services are also referred to as *elemental services*.

In [12], a language called Business Process Execution Language for Web Services (BPEL for short) has been defined to specify how to compose a service from other services based on business process models (see [33] for a quick overview on BPEL). First, BPEL requires the specification of all of the port types a composite service offers to the outside world and in turn all port types from the outside world, which it expects to use. Second, it requires specifying the potential ordering in which operations of these port types may be used or have to be used, respectively, and this ordering can be specified dependent on business rules. I.e. a composite service is specified by sets of port types and a business process model exploiting operation of these port types.

This introduces the paradigm of two-level programming [44] to Web services: Programming in the small for implementing the elemental services used by a composite service, and programming in the large for specifying the composite service itself. Programming in the small, i.e. the implementation of elemental services, is done based on usual programming languages (e.g. Java, C#), and based on known component technologies and application server environments (e.g. J2EE, .NET). The corresponding components are hosted and rendered by the environment as Web services, i.e. the elemental services. Programming in the large is done based on a business process language (e.g. BPEL) hosted and run by a workflow system (see [29]). The corresponding business process is rendered again as a Web service resulting in a composite service.

In a nutshell, the set of port types offered by a composite service to the outside world represents the interface of this service. This notion of a service as an aggregate goes beyond WSDL, but offering just a single port type corresponds to the known notion of a service in today’s WSDL. In section 5 we discuss other aggregation models for Web service.

BPEL can also be seen as a language for implementing a service based on other services. In this case, the Web service to be implemented is a composite service that offers a single port type to the outside world. If the services used to implement the composite service are publicly available the composite service is even portable to other environments that support BPEL, i.e. it will be able to be executed without any further actions; otherwise, the services used must be made available via appropriate deployment (see next section).

4.2 Reuse

The two-level programming paradigm introduces reuse at both levels: At the component level, i.e. elemental service level, and at the business process model level, i.e. composite service level. In practice, a vast number of isolated component functionalities does al-

ready exist in an enterprise, e.g. in form of purchased standard applications or home grown special applications. Typically, it is the knowledge of how to integrate these component functionalities into a business process that solves a (new) business problem. As a consequence, to become an artefact of reusability a business process model has to have the ability to be easily linked to the component functionalities available at an individual enterprise; a business process model with this property is sometimes called a *solution template* – or solution for short [32].

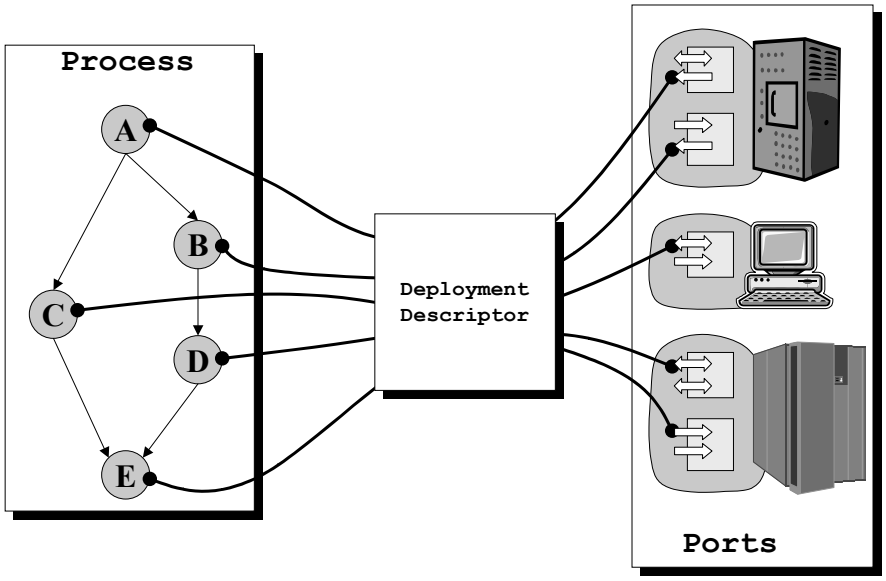

Figure 7 – Two-Level Application Structure

Linking a business process model to components is done in a step called *deployment* (see [22] for the original concept in J2EE, or [31] for a specialized concept called "locators"). During deployment for each port type referred to within the business process model it must be specified how it is bound to a corresponding port when an instance of the business process model is executed and makes use of an operation of a certain port type. *Binding* a port type to a port can be static or dynamic. *Static* binding assigns a fixed port to a port type. *Dynamic* binding assigns a mechanism to a port type that defines how a corresponding port is derived when needed at runtime. For example, one mechanism can be to assign a UDDI query to a port type that is to be evaluated at run time to determine a matching port. Another mechanism can be to expect a reference to the actual port to be used as a field in an incoming message (e.g. via service references and partner assignments in BPEL). The collection of deployment specifications associated with a business process model is called its *deployment descriptor*. Thus, a deployment descriptor links a solution template (i.e. a business process model) to existing ports, i.e. its turns a solution template into an executable solution (or application, respectively – see Figure 7). Those

ports might be both, elemental as well other composite services; the latter shows that the resulting programming model is recursive.

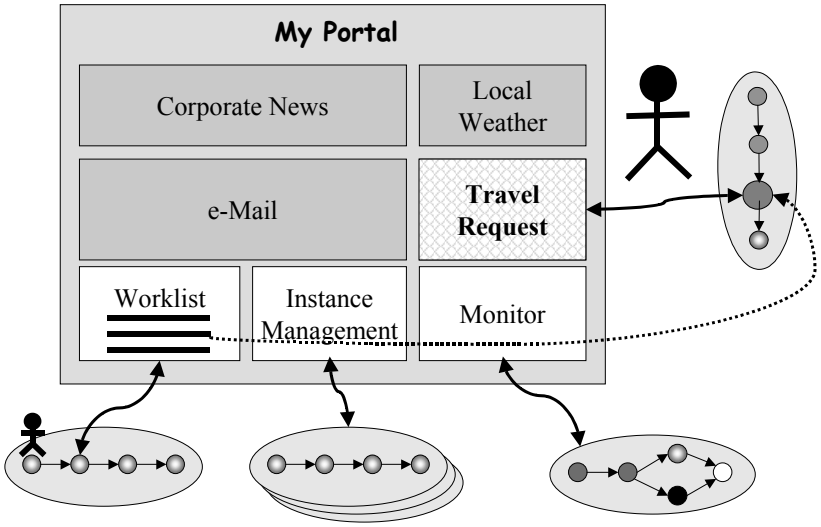

Figure 8 - Involving People in Business Processes

4.3 Involving People

Business processes may involve human beings [30]: A long running business process may be monitored by human beings interested in the actual state of the business process. Instances of business processes may be managed by human beings, e.g. an instance might be suspended and resumed later on. And a business process may involve people directly by creating work requests for certain people; these work requests are bundled into worklists for each person involved. A worklist may be perceived as a launch pad for tools supporting people in performing work requests: By selecting an item from a worklist the user initiates that the environment invokes the associated tool.

Typically, such an involvement of human beings in business processes is done today via portal technology. As shown in Figure 8, a portal may contain portlets that show a worklist of the person logged-on to the portal, and functions for the management and monitoring of business process instances. The tool to be launched when the person selects a work request from his worklist can be a Web service interacting with the person and that produces a user interface rendered within the portal (see [27] for the specification about how Web services can interact with portals).

For example, a person starts a business process for arranging a business trip based on instance management functionality made available in his portal. The corresponding business process first creates a work request for providing input about the trip to be arranged. This work request appears on the worklist of the person. When selecting the work request from the worklist the user interface appears that allows to key in the re-

quired data. Once this is done, the person can monitor the progress of his travel request via the monitoring functionality made available in the portal.

4.4 Sample Work to Be Done

Considering user-facing actions in the business process based two-level programming paradigm as well as the corresponding middleware aspects is something that has to be done in more detail. For example, how is a series of interactions with one and the same end user (i.e. a “dialog”) reflected best in a business process? How is this related to the model-view-controller paradigm (e.g. [36]) that is typically used in environments that are not workflow-based? What is the relation between workflow-based implementations of dialogs and other implementation techniques for end user interactions (e.g. Struts or Java Server Faces [35])?

One aspect of BPEL is to put constraints on the possible usage of operations of (collections of) port types; this specifies a certain kind of semantics for the corresponding port types. How does this contribute to shape “the semantic Web”? Another aspect of BPEL is that of an executable language: How can BPEL support Grid applications, i.e. what modifications or extensions of BPEL are needed? For example, how can a Grid scheduler exploit workflow functionality, especially based on BPEL?

The concept of a solution template is worth to be considered further: Not only complete business process models are “templates” for application functionality but also “appropriate” fragments of a business process model. What properties characterize reusable fragments? How can fragments be expanded to become complete solutions? How is the semantics of a fragment changed when it is expanded?

5 Aggregation

The model of building a composite service as introduced in section 4.1 is one example of an aggregation model for Web services. In this model aggregation is done at the port type level by specifying both, the port types offered as well as required by the aggregate. Furthermore, the aggregation is very much structured and constrained in its behaviour by the associated business process model, i.e. it is “choreography”-centric: It prescribes the potential order in which the operations of the aggregated port types are to be used. And it is “pro-active” by defining an execution model that actually drives the usage of the aggregated port types. On the other hand, it is non-recursive in the sense that defining new port types based on its aggregated port types is not its focus.

Other aggregation models for Web services are possible:

- Aggregation models at the port type level focussed on the recursive definition of new port types (section 5.1).
- Aggregation models at the instance level (i.e. port level or service level, respectively) focused on (statically or dynamically) collecting services of certain port types without any assumption about structural relations between the services (section 5.2).

- Aggregation models at the instance level focussed on reaching outcome agreement between services that cooperate in a not explicitly prescribed manner (section 5.3).

5.1 Global Models

The definition of a recursive aggregation model (called *global model*) for specifying collections of new port types is included in [31] (see Figure 9). This model defines the notion of a *service provider type* as a set port types (e.g. SP^a). The only structural relation between service provider types is that they make use of each other's services. The relation between service provider types and the aggregate itself is that the aggregate's interface is built from the service provider types' interfaces.

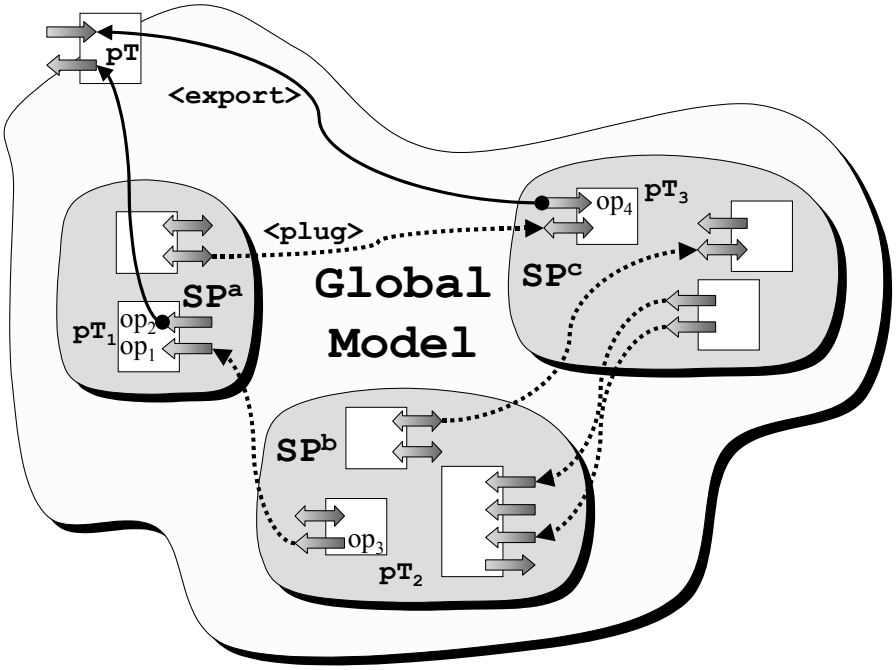

Figure 9 - Aggregation via Global Models

Operations of port types of different service provider types can be connected via a directed *plug link*. A plug link defines a client-server relationship between operations specifying who the initiator is and who the follower within an interaction is. For example, the out-operation op_3 of port type pT_2 of service provider SP^b is the source of a message send to the in-operation op_1 of port type pT_1 of service provider SP^a that consumes this message. It is not required that all operations are source or target of a plug link, i.e. a service provider might offer operations that are not used by other service providers of the aggregate. Furthermore, a plug link allows defining message transformations to handle

cases where the signatures of the linked operations do not match; for example, such a situation appears quite often in EAI environments.

The (new) interface of the aggregate is defined by *exporting* operations of constituent port types that are not used within a plug link. The semantics of exporting an operation is that the implementation of an operation from the interface of the aggregate is delegated to the operation of a port type of a service provider. For example, the in-operation of port type pT in Figure 9 is in fact the exported operation op_4 of port type pT_3 of service provider SP^c, i.e. if a requestor uses the in-operation of the aggregate the environment hosting the aggregate will forward the incoming message to op_4 of pT_3. The collection of service provider types, plug links and exports needed to define new port types make up a global model.

5.2 Service Domains

In some application scenarios, a requestor needs a collection of related services that he will use in a non-predefined manner. Properties beyond the signature level of a concrete service are irrelevant to a requestor, i.e. individual ports providing the same service are indistinguishable from a requestor's point of view. [41] specifies a complete environment for such aggregations; the corresponding aggregation model is referred to as *service domain*. For conciseness reasons, we will take the liberty here to use the same name but describe a variant of this aggregation model.

Basically, a service domain is a set of ports implementing a predefined set of port types. In general, for each particular port type associated with a service domain there is more than one port implementing this port type. A service domain aggregates these ports by providing for each of its port types a port that functions as a proxy for the set of ports implementing the same port type. When a requestor sends a message to this proxy the environment will select one implementing port and dispatch the message to it.

An extension of this base model introduces more dynamics: Providers can register and unregister ports with a service domain. Registration includes specification of the service levels (e.g. throughput, average response time) for the offered operations. Requestors are using services of a service domain based on formerly established service level agreements. Consequently, the environment will select implementing ports based on matching service levels and optimizing the utilization of the overall environment.

5.3 Coordination

Often, the final outcome of the usage of some services is dependent on the final outcome of the usage of some other services. As a result, an aggregation model is needed that allows dynamically creating temporary collections of services the joint outcome of their usage is determined once the period of usage of the services within the collection is over. The determination and dissemination of the joint outcome is based on a collection-specific set of protocols supported by the participating services, i.e. member of the collection.

Example 1: Consider a sealed-bid auction for divisible goods. Sellers inform the auctioneer about their goods to sell and buyers submit to the auctioneer the maximum price they are willingly to pay for a certain quantity of the good. Once the bidding period is over the auctioneer uses a clearing algorithm to determine the winners as well as the actual price each individual winner has to pay for his quantity. Finally, the auctioneer informs the seller about the winners as well as the corresponding prices and quantities, and he notifies winners and losers accordingly. Technically, the auctioneer, the seller, and the buyers are represented by appropriate Web services. When the seller offers his good he opens up a temporary collection of Web services that incrementally consists of his own service, the auctioneer's service, as well as the services of all bidders. The auctioneer, the bidders and the seller follow a certain protocol: First, the seller begins the trade by informing the auctioneer about the good to sell, then, the auctioneer is sending out request for bids, next, the bidders submit their bids to the auctioneer, and finally the seller notifies the seller, the winners, and the losers. After that, the temporary collection of services ceases to exist.

Example 2: Consider a distributed transaction that is managed via a two-phase-commit protocol ([18], [43]). An application begins a transaction with a transaction manager and issues manipulation requests to various resource managers. When the application has finished all of its manipulations it will closes the transaction by issuing a commit request to the transaction manager. The well-known two-phase commit protocol is then run amongst the players on the scene to determine the joint outcome of the transaction. Again, the application, the transaction manager, as well as the resources can be Web services. In the course of the transaction these services are dynamically aggregated into a temporary collection of services managed by a certain protocol set in terms of outcome agreement.

The corresponding abstraction at the Web service level – called Web Services Coordination (or WS-C for short) – has been specified in [8] (see [17] for a quick overview on WS-C): A *distributed activity* is a unit of computation that consists of a set of different services, and that requires to jointly agree on the outcome of the unit of computation between the constituting services. Agreement is reached base on *coordination protocols*. A coordination protocol is a collection of messages together with a prescription about how these messages are to be exchanged in order to reach agreement (e.g. the protocol between a bidder and an auctioneer, or a transaction manager and a resource manager). A coordination protocol is represented by a collection of port types, where each port type is the result of a logical grouping of messages appropriate for a class of participants in the protocol (e.g. the port types representing a bidder or an auctioneer, respectively). Coordination protocols are grouped into *coordination types*. A coordination type is a set of coordination protocols needed to reach agreement between the different kinds of services of a certain type of distributed activity (e.g. the protocols between seller and auctioneer, and bidder and auctioneer need to determine the outcome of a sealed-bid auction).

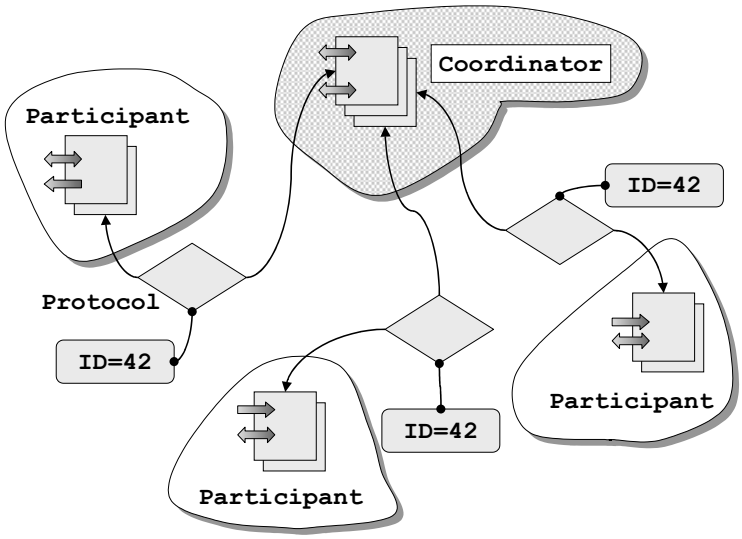

Figure 10 - Aggregation via Coordination

A *coordinator* provides services to create a distributed activity and to register for participation in a distributed activity (see Figure 10). An activity is created by specifying the coordination type used to agree on the outcome between the various participants. After creation, each activity has a unique *activity identity*. A service registers with an activity as *participant* by specifying the coordination protocol(s) it will honour. During registration, the participant and the coordinator exchange references to the ports that provide the operations allowing to mutually receiving the messages of the protocol(s). At the end of the activity the coordinator will communicate with each participant according to its registered protocol(s) to determine the agreed outcome of the activity.

WS-C specifies the framework sketched above but does not specify any coordination type. An accompanying specification called Web Services Transaction [9] (or WS-Tx for short) specifies coordination types for two major application areas: For “traditional” short-running distributed transactions, and for the extended transaction model for business processes specified in BPEL that allows to manage long-running units of work based on compensation actions (an extension of the model introduced in [28]).

5.4 Sample Work to Be Done

The aggregation models described in this section are not meant to be exhaustive. What other aggregation models are possible, and what are their application areas? How can the presented aggregation models be combined to become applicable in new areas (e.g. distributed activities across service domains)? Are there some sorts of “basic” aggregation models that can be used to describe other aggregation models?

6 Summary

In this paper we have demonstrated that Web services are the base for a new era of distributed computing. Web services are virtual components hiding from their users idiosyncrasies of the concrete (application server) technology chosen to implement the Web service. Especially, users can easily mix and match functions from heterogeneous environments into a single application if those functions are rendered as Web services. Based on a service-oriented architecture a user does not even have to care about a particular Web service he is communicating with because the underlying infrastructure, i.e. the service bus, will make an appropriate choice on behalf of the user. This choice is based on policies of both, the user and the Web services qualifying under the user's functional request, and the choice is also influenced by service level agreements and demand for an optimal utilization of the overall environment. We have shown that Grid computing technology and Web service technology are about to converge to provide these features and more, enabling utility computing and on-demand computing. Aggregations of Web services support a broad spectrum of requirements reaching from recursive component construction over advanced provisioning of groups of services to transaction management. Application construction and execution will be based on business process technology composing Web services into higher-level business functionality based on a two-level programming paradigm. Involvement of human beings in these applications is achieved by appropriate exploitation of portal technology. Figure 11 summarizes the corresponding middleware stack graphically.

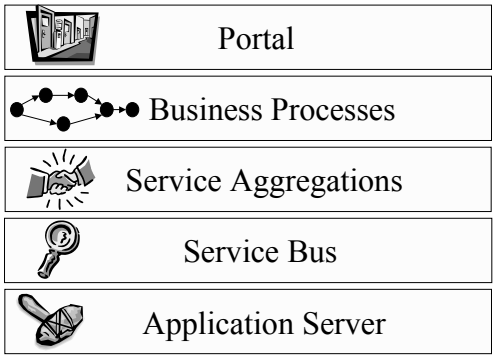

Figure 11 - Middleware Stack For Services-Oriented Applications

For the subject areas discussed in this paper we listed some selective research items. Many aspects that are of high importance for this kind of an environment like systems management, tooling for application construction and monitoring etc have not even been touched in this paper. Security aspects are utmost importance from a business perspective [6], but have been left out too. Payment methods, contracting etc appropriate in a Web service and on-demand environment are to be investigated, especially in situations in which services are recursively aggregated.

References

[1] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham and P. Yendluri, Basic Profile Version 1.0, http://www.ws-i.org/Profiles/Basic/2002-10/BasicProfile-1.0-WGD.htm

[2] W. Beer, D. Birngruber, H. Mössböck and A. Wöß, Die .Net Technologie, dpunkt Verlag, 2003.

[3] T. Belwood et al, UDDI Version 3.0, http://uddi.org/pubs/uddi-v3.00-published-20020719.htm

[4] V. Berstis, Fundamentals of Grid computing, IBM Corporation (2002), http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf

[5] D. Box et al, SOAP 1.1, http://www.w3.org/TR/SOAP

[6] C. Boyens and O. Guenther, Trust is not enough: privacy and security in ASP and Web services environments, Proc. ADBIS 2002 - 6th East-European Conference on Advances in Databases and Information Systems (September 8-11, 2002, Bratislava, Slovakia).

[7] S. Burbeck, The Tao of e-business services, IBM Corporation, 2000, http://www-4.ibm.com/software/developer/library/ws-tao/index.html

[8] F. Cabrera, G. Copeland, T. Freund, J. Klein, D. Langworthy, D. Orchard and J. Shewchuk, Web Services Coordination, BEA Systems & IBM Coporation & Microsoft Corporation, 2002, http://www-106.ibm.com/developerworks/library/ws-coor/

[9] F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey and S. Thatte, Web Services Transactions, BEA Systems & IBM Coporation & Microsoft Corporation, 2002, http://www-106.ibm.com/developerworks/library/ws-transpec

[10] M. Champion, Ch. Ferris, E. Newcomer and D. Orchard, Web Services Architecture, http://www.w3.org/TR/ws-arch/

[11] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, WSDL 1.1, http://www.w3.org/TR/WSDL

[12] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte and S. Weerawarana, Business Process Execution Language For Web Services, BEA Systems & IBM Coporation & Microsoft Corporation, 2002, http://www-106.ibm.com/developerworks/library/ws-bpelwp

[13] B. Daum and U. Merten, System architecture with XML, Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[14] D. Fensel, Ontologies: A silver bullet for knowledge management and electronic commerce, Springer, 2001.

[15] I. Foster and C. Kesselman, The Grid: Blueprint for a new computing infrastructure, Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[16] I. Foster, C. Kessleman, J.M. Nick and S. Tuecke, The physiology of the Grid – An open Grid services architecture for distributed systems integration, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002, http://www.globus.org/research/papers/ogsa.pdf

[17] T. Freund and T. Storey, Transactions in the world of Web services, http://www-106.ibm.com/developerworks/webservices/library/ws-wstx1

[18] J. Gray and A. Reuter, Transaction processing, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[19] V. Gruhn and A. Thiel, Komponentenmodelle, Addison-Wesley, 2000.

[20] IBM, Living in an on demand world, IBM Corporation (2002), http://www-3.ibm.com/e-business/doc/content/feature/offers/whitepaper.pdf

[21] IBM and Microsoft, Security in a Web Services World: A Proposed Architecture and Roadmap, IBM Corporation and Microsoft Corporation (2002), msdn.microsoft.com/ws-security

[22] Java™ 2 Enterprise Edition, Java™ Platform Enterprise Edition Specification, Version 1.4, Sun Microsystems 2002.

[23] Ch. Kaler (ed.), Web Services Security, http:// www-106.ibm.com/developerworks/ web-services/library/ws-secure

[24] Ch. Kaler (ed.), Web Services Policy Framework, http://www-106.ibm.com/developerworks/library/ws-polfram

[25] Ch. Kaler (ed.), Web Services Policy Attachment, http://www-106.ibm.com/developerworks/library/ws-polatt

[26] D. König, M. Kloppmann, F. Leymann, G. Pfau and D. Roller, Web service invocation framework: A step towards virtualizing components, Proc. XMIDX'2003 (Berlin, Germany, February 17 – 18, 2003).

[27] A. Kropp, Ch. Leue and R. Thompson (editors), Web Services for Remote Portlets, Working draft 0.85 (OASIS, 26 November 2002), http://www.oasis-open.org/committees/wsrp

[28] F. Leymann, Supporting business transactions via partial backward recovery in workflow management systems, Proc. BTW'95 (Dresden, Germany, March 22-24, 1995), Springer, 1995.

[29] F. Leymann and D. Roller, Workflow based applications, IBM Systems Journal 36(1) (1997).

[30] F. Leymann and D. Roller, Production Workflow, Prentice Hall Inc., Upper Saddle River, New Jersey, 2000.

[31] F. Leymann, Web Services Flow Language, IBM Corporation (2001), http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[32] F. Leymann, D. Roller and M.-T. Schmidt, Web services and business process management, IBM Systems Journal 41(2) (2002).

[33] F. Leymann and D. Roller, Business processes in a Web services world, IBM Corporation 2002, http://www-106.ibm.com/developerworks/library/ws-bpelwp/

[34] H. Ludwig, A. Keller, A. Dan, and R. King: A Service Level Agreement Language for Dynamic Electronic Services. Proceedings of WECWIS 2002, Newport Beach, CA, pp. 25 - 32, IEEE Computer Society, Los Alamitos, 2002.

[35] C.R. McClanahan (ed.), Java Server Faces, Sun Microsystems, Inc., 2002 - http://java.sun.com/j2ee/javaserverfaces/.

[36] S. Middendorf, R. Singer and J. Heid, Java, dpunkt Verlag, 2003.

[37] S. Mullender, Distributed systems, ACM Press, 1993.

[38] J. Nick, OGSA – Framework for Grid Service Evolution, Presentation at OGSA Early Adopters Workshop, Argonne National Lab, CA (May 29 – 31, 2002), http://www.globus.org/ogsa/events/JeffNickOGSAFramework.pdf

[39] T. Sandholm et al, Java OGSI Hosting Environment Design: A portable Grid service container framework, http://www.globus.org/ogsa/java/OGSIJavaContainer_2002-07-19.pdf

[40] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, Pattern-orientierte Software Architektur, dpunkt Verlag, 2002.

[41] Y.-S. Tan, B. Topol, V. Vellanki and J. Xing, Implementing service Grids with the service domain toolkit, IBM Corporation, 2002

[42] S. Tuecke et al, Grid service specification, http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-04_2002-10-04.pdf

[43] G. Weikum and G. Vossen, Transactional information systems, Academic Press, San Diego, CA, 2002.

[44] G. Wiederhold, P. Wegner, S. Ceri, Towards Megaprogramming: A paradigm for component-based programming, Comm. ACM 35(22) 1992, 89 – 99.