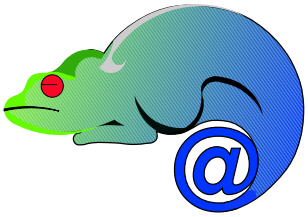
ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

CS Middleware Architecture



Deliverable Identifier: D9
Delivery Date: 21 Mar 2003
Classification: Public Circulation
Editors: Gustavo Alonso, Daniel Jönsson, Biörn Biörnstad, Simon Woodman, Stuart Wheater, Santosh Shrivastava
Document version: 1.0 (20 Mar 2003)

Contract Start Date: 1 September 2002
Duration: 36 months
Project coordinator: Universidad Politécnica de Madrid (Spain)
Partners: Universitá di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Universitá degli Studi di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd (UK)

Project funded by the European Commission under the Information Society Technology Programme of the 5th Framework (1998-2002)

CONTENTS

1. Introduction

The composition of Web Services is a non-trivial task using the basic frameworks available at present. ADAPT will build upon the current standards to simplify this process.

1.1. Overview of the State of the Art

As part of the initial efforts in ADAPT, we have invested considerable time in researching the area of Web Services. The aim has been to build a unified and coherent view on what Web Services are and how they can be used. We have also paid special attention to those specifications that may play an important role from the point of view of the activities to be pursued within ADAPT.

1.2. Service Based Architecture of Web Services

The typical Web Service architecture follows a proposal made by IBM. The architecture has three components: the service requester, the service provider, and the service registry, thereby closely following a client/server model with an explicit name and directory service (the service registry). Albeit simple, this architecture illustrates quite well the original purpose of UDDI, WSDL and SOAP. In all cases, the information managed by these specifications is in the form of XML documents.

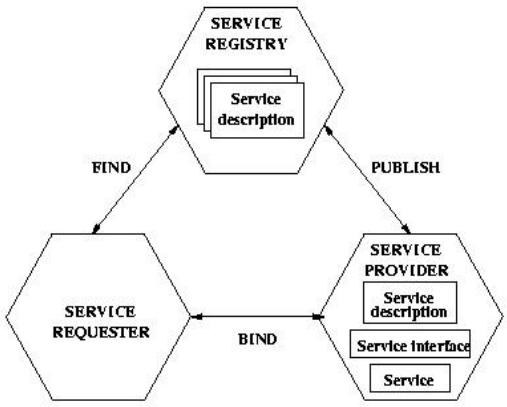


Figure 1.1: The IBM WS architecture

UDDI

The service registry is based on the UDDI specification (Universal Description, Discovery, and Integration). The specification defines how to interact with a registry and what the entries on that registry look like. Interactions are of two types: registration and lookup. Registration is the procedure whereby new service descriptions are added to the registry. Lookup corresponds to queries sent by service requesters in search of the right services. The entries contain three types of information: white, yellow and green pages. The white pages contain generic information about the service provider (e.g. address, contact person, etc.). The yellow pages include categorisation information that allows the registry to classify the service (e.g. flight reservation, search engine, or bookstore). The green pages contain information about the service's interface and pointers to the service provider (where the actual WSDL interface definition can be found).

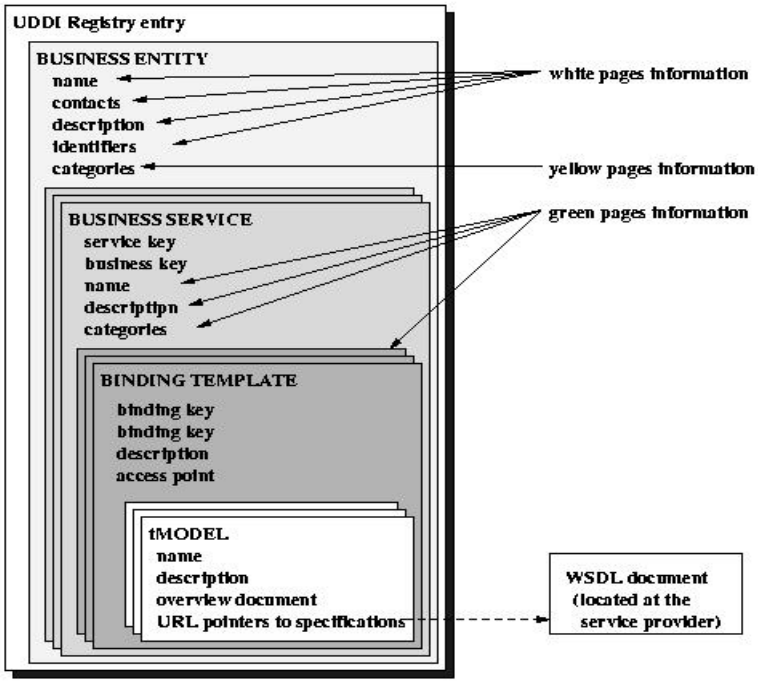


Figure 1.2: Overview of an UDDI Registry entry

There are already several UDDI registries maintained by software vendors. These public registries are meant as low level, generic systems supporting only the most basic of interactions. The underlying idea is that more sophisticated repositories (e.g. with advanced query capabilities) will be built on top of UDDI repositories. Such service databases are, however, not part of the specification. UDDI also describes how to interact with a repository using SOAP. Such support is intended not so much for dynamic binding to services (in the middleware sense) as for developers building advanced service databases and other applications on top of UDDI repositories. Finally, there are two types of UDDI registries: public and private. Public ones are accessible to everyone and play the role of open search engines for Web Services. Private ones are those that a company or a group of companies create for their own use. For obvious reasons, industrial strength Web Service implementations are likely to be based on private repositories rather than on public ones. It remains to be seen to what extent private repositories use UDDI, as much of its functionality is not needed for private use.

In terms of its use in ADAPT, UDDI will play a marginal role at this stage. The reason is that it is not a key component and it is not the most complex one (in ADAPT, the concern is about performance issues, etc., that occur after binding). Additionally, in practice it is also the component of the Web Service architecture that is attracting less attention from the industry as the interest shifts from the notion of universal repository to a more realistic private repository supporting Web Service documentation rather than dynamic binding or complex searches.

WSDL

The interface to a Web Service is defined using WSDL (Web Services Definition Language). By using WSDL, designers specify the type system used in the description, the messages necessary to invoke an operation of the service (and their format), the operation protocol (whether it returns a response, etc.), the port type or set of operations that conform an instance of a service, and the binding or actual protocol to be used to invoke the operations of an instance of a service (e.g. HTTP). Note that what is known as a "service" is a logical unit encompassing all port types mapped to the same logical service (e.g. flight reservations through RPC or through e-mail, each one of them being a port type of the flight reservation service).

In ADAPT we will follow the WSDL specification in all work packages for describing both Basic and Composite Services. We do not expect major changes in this specification except for improvements that will make our job easier (support for attachments, etc.).

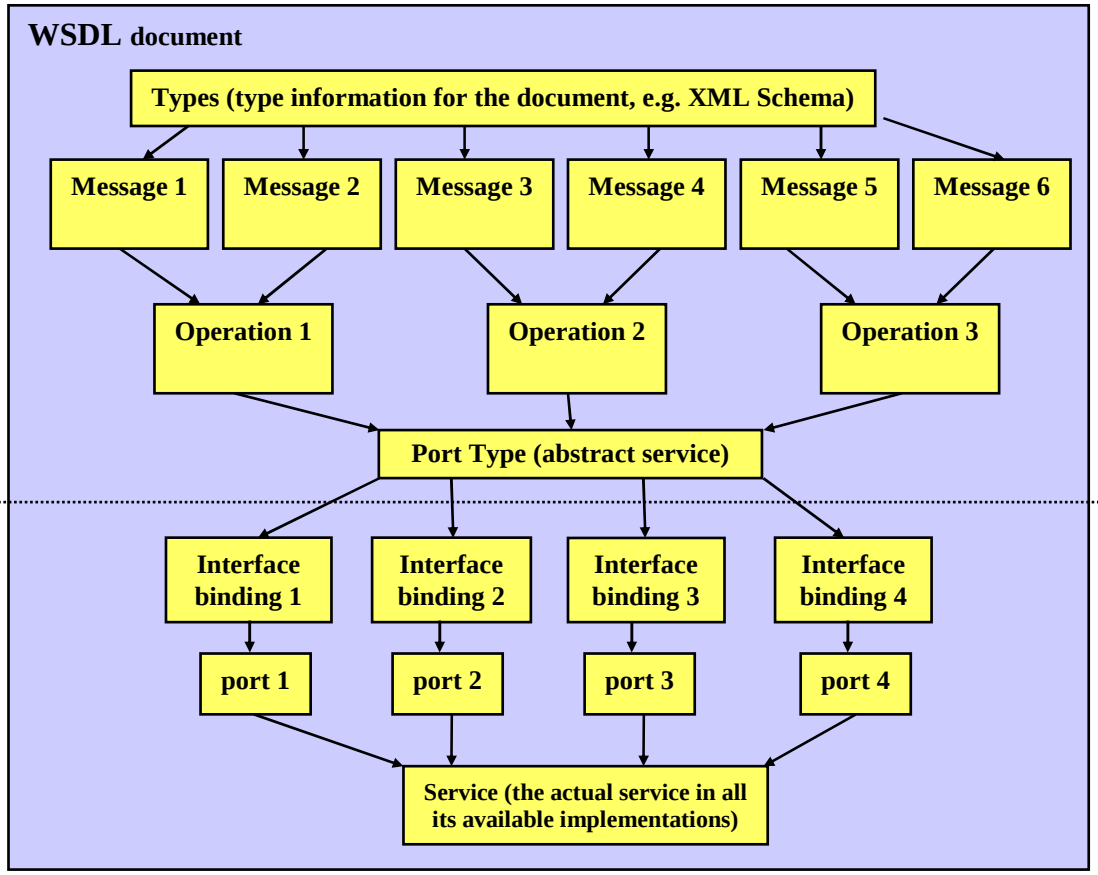


Figure 1.3: The structure of a WSDL document example

SOAP

Interaction between requester, provider and registry happens through SOAP (Simple Object Access Protocol). SOAP specifies messages as documents encoded in XML divided into two parts: header and body. Both the header and the body can be subdivided into blocks. Header blocks carry information related to the interaction: e.g. security, authentication, transactional context, etc. Body blocks store the data used in the interaction, e.g. which procedure is being called, each individual parameter, etc. SOAP also defines bindings to actual transport protocols. A binding specifies how a SOAP message is transmitted using, e.g. HTTP.

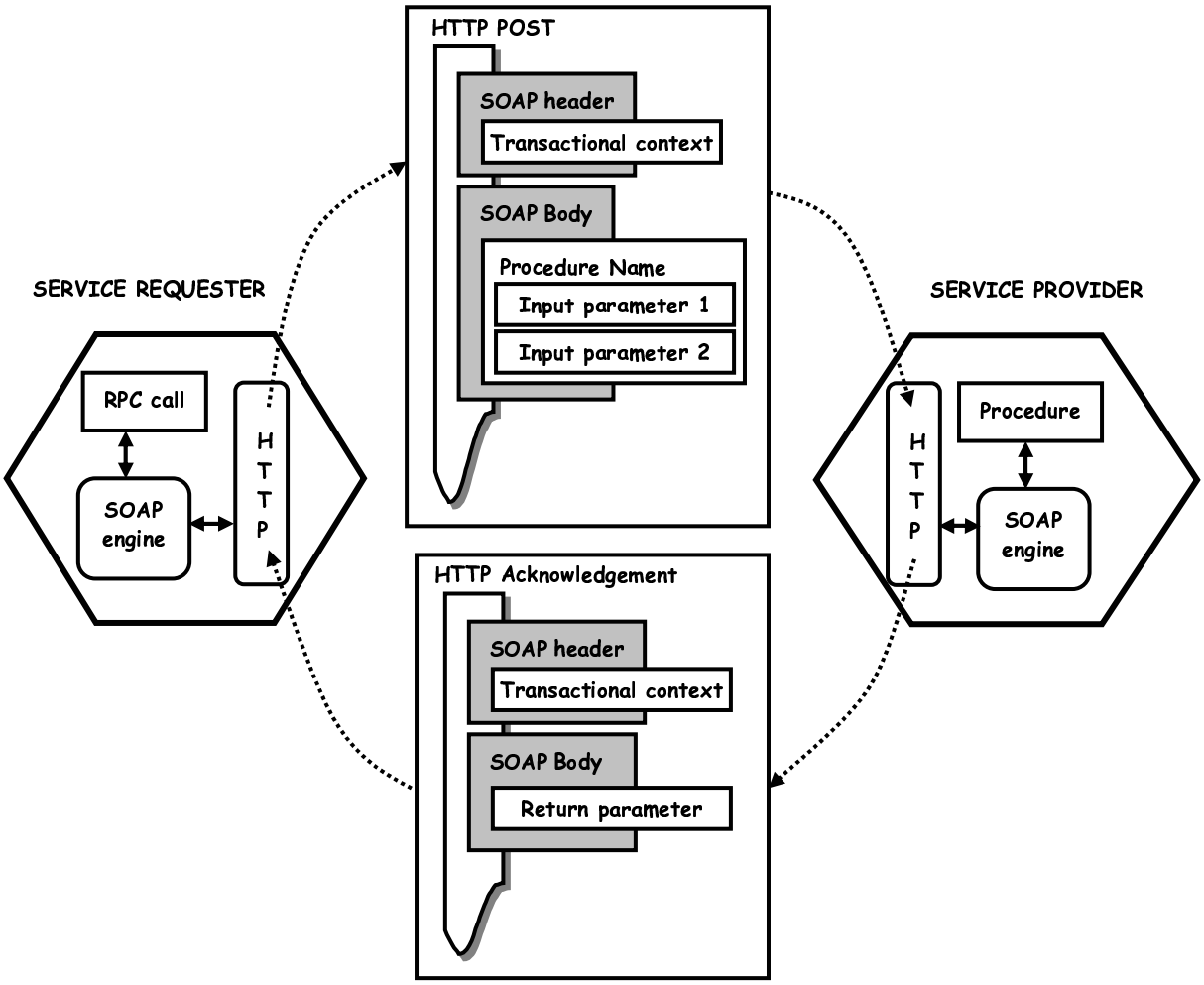


Figure 1.4: Overview of SOAP

SOAP can be best understood when it is considered as the specification of a protocol wrapper rather than a communication protocol itself. The main point of SOAP is to provide a standardised way to transform different protocols and interaction mechanisms into XML documents. As such, each concrete protocol needs a SOAP specification. An example is the specification of how to use RPC over HTTP. The specification describes how to encode an RPC invocation into an XML document and how to transmit the XML document via HTTP.

As with WSDL, in ADAPT we will use SOAP for exchanges of messages with and between Web Services at all levels and in all work packages. Important developments to consider here are the possible standardisation of additional bindings, the potential support for attachments (either based on MIME or DIME), and asynchronous messaging.

1.3. Document and Message Based Architecture of Web Services

While the service architecture discussed above is the one most widely used when discussing Web Services, it is important to keep in mind that there are alternative proposals. These proposals may end up playing a crucial role and we intend to keep close track of them as part of ADAPT. In particular, most of these alternative proposals are typically free of Intellectual Property Right (IPR) problems, something that isn't yet the case for UDDI, WSDL and associated specifications.

Among these alternative proposals, the one that appears to be the most interesting for the purposes of ADAPT is the architecture put forward by ebXML. In the appendix there are several examples and references to these architectures. Here we will focus solely on its potential impact on the work done in ADAPT.

The main difference between the ebXML architecture and the service based architecture is that interaction is always between partners rather than a requester and a provider. Moreover, the interaction is always based on the interleaving of business processes rather than single service calls. This interleaving or ordered exchange of messages between cooperating business processes is what is called a conversation or business protocol. Part of the goals of ebXML is to standardise business protocols rather than services, following up on the model already established by the Electronic Data Interchange (EDI) standard.

The dilemma that these two different approaches pose to ADAPT is whether to concentrate on simple Web Services of the request-response type or concentrate on the conversations between Web Services. From the point of view of basic services this may not make much of a difference. From the point of view of Composite Services it changes the picture radically. Currently we are keeping our options open since it is not clear which one of these two views on Web Services will dominate the electronic commerce arena. At this stage, however, we tend to support more the conversation based approach although this trend will be revised later on in the project in view of developments in these standards.

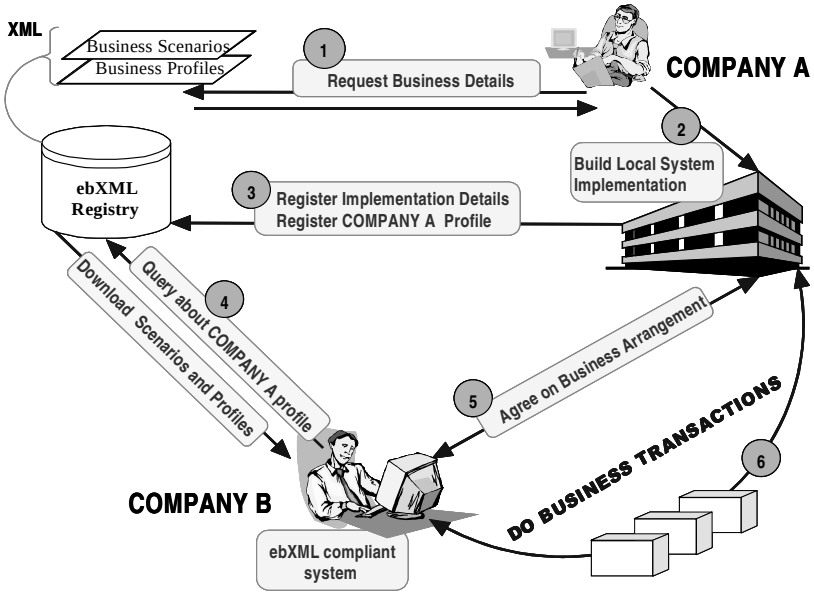


Figure 1.5: ebXML architecture

1.4. Keeping Up with Continuous Change

One of the main problems with Web Services is that they are still in a very early stage and specifications as well as perceptions from both industry and research are in a constant flux. This will be a challenge for ADAPT as we will need to continuously revisit our design decisions and to match any developments that may happen around Web Services. In fact, the task of keeping up to date regarding changes to specifications and new proposals will already be quite demanding.

This problem has been clear to the ADAPT consortium from the beginning and we have put in place a strategy to make sure all partners are timely informed of interesting developments. In combination with the ADAPT effort, and also with a view to establish a vehicle for dissemination of the results, ETHZ has started to give an industry course on Web Services. This is a course for people in the industry that will be given at least twice a year (the first course took place in February 2003) and where the latest developments on Web Services will be presented from a critical perspective. We plan to maintain the course for the duration of ADAPT and we will take advantage of it to present the results of ADAPT to the industry as they become available. Keeping the course running will also require a continuous update of its material. We will take advantage of the efforts around this course to keep everybody in the consortium up to date on the area of Web Services and to make sure there is a common understanding of the technology. Hence, as supporting material for this overview, Appendix [1] contains the basic material for the first industry course on Web Services. As new editions of the course are offered, the new materials for the course will be distributed to all partners using the facilities available for that purpose within ADAPT. The material for the first course was already distributed to partners during the ADAPT plenary meeting of February 2003 (Bologna, Italy). In that course, there was also a practical session where some of the initial work and the purposes of ADAPT were presented to the participants.

2. ADAPT Model

2.1. Virtual Business Processes, Virtual Enterprises, Trading Communities

As we aim to allow composition of Web Services/operations beyond corporate boundaries, the following definitions for Virtual Business Processes, Virtual Enterprises, and Trading Communities should come in handy.

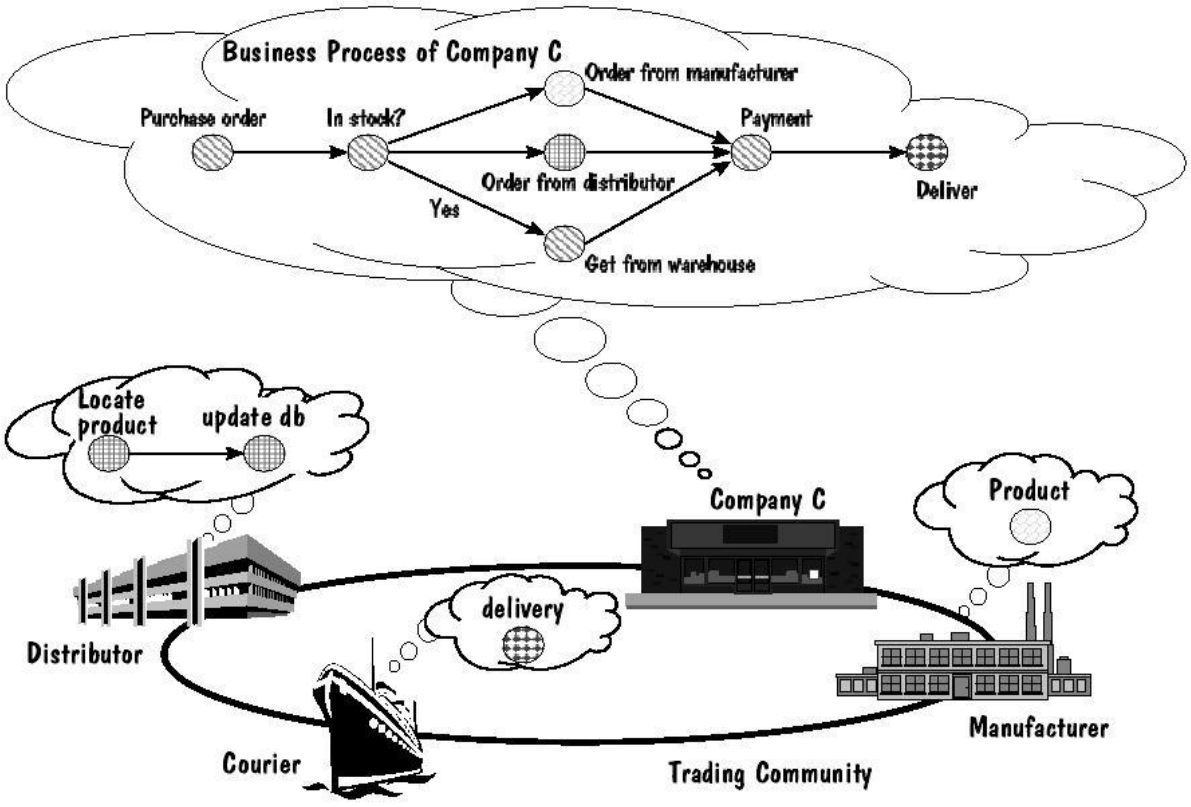


Figure 2.1: A company incorporating a virtual process as its own business processes

These notions can be briefly characterised as follows:

A Virtual Business Process is used to achieve concrete business goals and describe the corresponding activities. Unlike normal processes, in a Virtual Business Process the definition and enactment is not tied to a single organisational entity. Two examples of such processes are shown in Figures 2.1 and 2.2. In both cases, the Virtual Business Process appears as a normal process except for the fact that some steps within the process correspond to individual activities or entire sub-processes in different organisations. In a way, the Virtual Business Process can be seen as a meta-process: its building blocks are the sub-processes provided by the participating companies. For instance, in Figure 2.1 a company incorporates activities (as part of one of its own processes) that are carried out at other companies. In this case, the company acts as a dealer in merchandise that either it has in stock or obtains directly from other distributors or the manufacturer. It also uses a fourth company for the delivery of the merchandise. As the figure shows, the Virtual Business Process is the one at Company C since

it is the only one reaching across the participating companies. Note, however, that it is not the only process involved: the distributor, manufacturer and courier may implement their steps as business processes themselves. This illustrates one important aspect of our notion of Virtual Business Processes; in order to build such a process, we do not necessarily need to know the details of the component processes. Much like encapsulation and modular programming in modern programming languages, we only need to know the interface to the component process in order to incorporate it into the Virtual Business Process. Here's where the Web Services architecture shows its utility, by greatly simplifying the integration of these processes. Note that the level of nesting is not limited, i.e. the component process itself could be another Virtual Business Process.

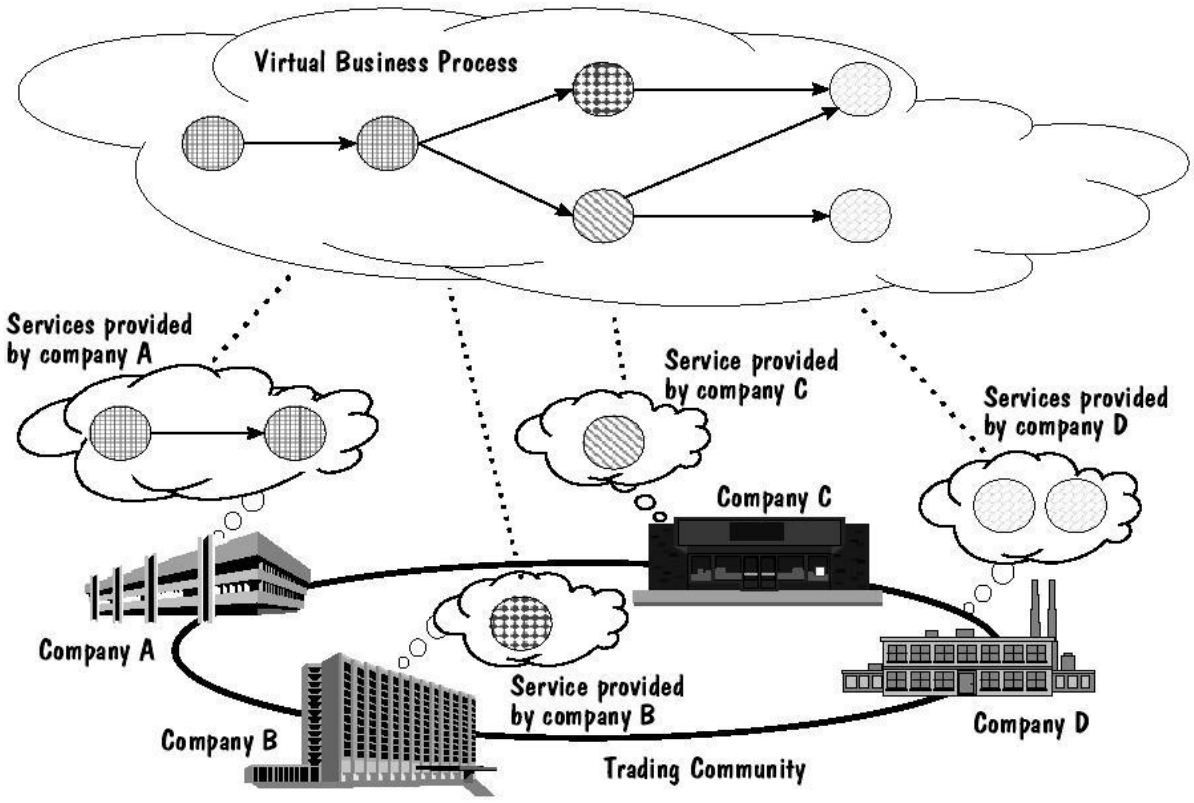


Figure 2.2: A virtual process combining the Web Services of different companies

Another important aspect to consider is that Virtual Business Processes are independent of the language used to represent either the Virtual Business Process or the component processes. In fact, since what is needed for integration and validation is just the interface, Virtual Business Processes and component processes could use entirely different representations. To make a process available for usage by the other involved participants, it's enough to publish the interface in a WSDL file. The internal details of the individual activities (often proprietary in nature) stay hidden. In general, defining the interface is not a significant problem since it is usually specified as part of the contract binding the companies. As a last point, there are many organisational and formal aspects of interest related to Virtual Business Processes, but not all those will be brought to light here.

A Virtual Business Process cannot be defined without a context, i.e. without a set of goals, rules, requirements, constraints, and resources. This context is what's termed "Virtual Enterprise". Alternatively, a Virtual Enterprise can be seen as an organisation based on Virtual Business Processes, independently of whether there is a real organisation behind the Virtual Enterprise or not. For instance, in Figure 2.1 the Virtual Enterprise is part of Company C, while in Figure 2.2 the Virtual Enterprise is indeed virtual, in the sense that there is not necessarily a (single) physical organisation behind it.

The concept of "Virtual Enterprise" is well motivated. In practice, the context of a Virtual Business Process is very important and the determining factor in terms of its feasibility. Everything that cannot be resolved at the level of the component processes must be resolved at the Virtual Enterprise level, that is, within the context of the virtual process. Naming this context explicitly allows us to have a much better perspective on the tools to develop and how they should interact with each other. For instance, it allows the specification of what to do in case of exceptions at the virtual process level.

To identify or define the Virtual Enterprise is in some cases straightforward – as in Figure 2.1 – while in other cases it can become a major endeavour from the organisational and legal point of view – as tends to happen in scenarios like the one depicted in Figure 2.2.

Typical issues which arise at this stage are:

- Who owns the information about the virtual process? (one or all of the participants)
- Who manages this information?
- Who has the right to sell this information as a value-added service?
- Where should the software platform be located? (fully decentralised, in one of the participants, in a neutral organisation, in an intermediary company offering the Virtual Enterprise as a service to the Trading Community?)

All these are organisational and legal issues beyond the scope of this overview, but they should be kept in mind since an adequate software platform will simplify them. However, as is the case with existing tools, a poor design will make the problem even more complex, greatly detracting from the potential of composite Web Services.

Once we have defined what to do (the Virtual Business Process) and the context in which it should be done (the Virtual Enterprise), we need to define the actors in the scenario. For this purpose, we use the notion of Trading Community, which can be best described as the set of companies participating in a Virtual Enterprise. Alternatively, a Trading Community could be defined as the set of companies which provide the building blocks of the Virtual Business Process. These two definitions are roughly equivalent: we consider a 1:1:n mapping between the Trading Community, the Virtual Enterprise and the Virtual Business Process. That is, each Virtual Enterprise has one Trading Community and can run a number of Virtual Business Processes. From a practical standpoint, defining the Trading Community is the first step towards defining access rights, responsibilities, authentication and encryption mechanisms, and the configuration of the underlying distributed system.

2.2. The model as a whole

How the model is used in practice can be best seen with an example. Consider, for instance, the scenario shown in Figure 2.3. In this scenario, the Trading Community consists of two different departments of an insurance company (policies department and claims department) and a loss adjuster company. Each member of the Trading Community provides services (Check Customer, Claim Classification, Damage Assessment) which are used as building blocks for the virtual process. Based on these services, the Virtual Enterprise can be created by defining a virtual process in which individual activities correspond to services provided by the participants. Note that there are several ways to interpret this virtual process. One is to see it as totally virtual, as shown in Figure 2.2, in the sense that the virtual process does not belong to one company within the Trading Community. Another possibility is that in which a company within the Trading Community incorporates the services of other companies as elements of its own business processes, as shown in Figure 2.1. In both cases, the concept is the same and poses the same challenges and difficulties.

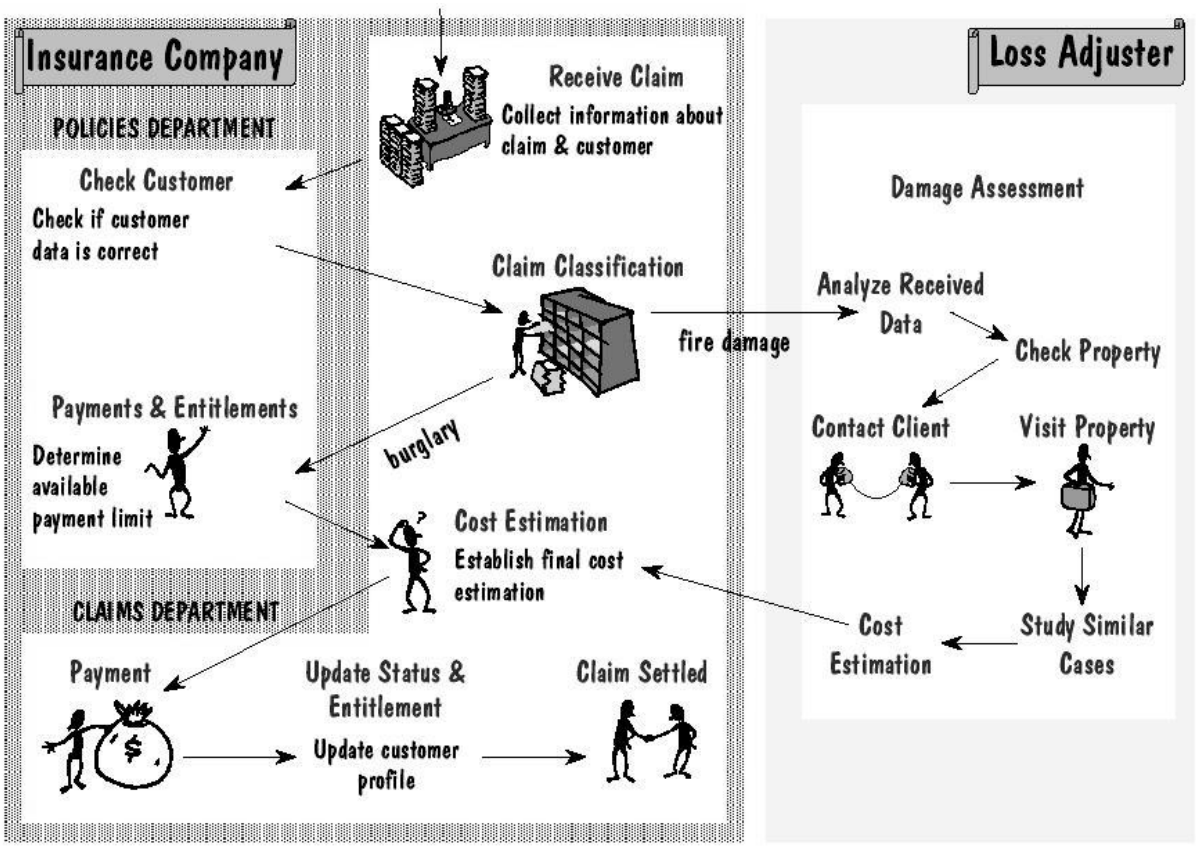


Figure 2.3: Example of a Virtual Business Process

The flow in this process is to be interpreted as follows. The insurance company defines a Virtual Business Process to handle insurance claims. In the first step of the process, a clerk in the claims department receives the claim and collects all the necessary information about the claim itself, the customer, the involved parties, etc. This information is processed in the policies department. In this department, the data provided is correlated with the information available in the database; whether the customer is up to date on payments, whether it is a case covered by the existing policies, and so forth. Once this step is completed, the information is returned to the claims department where the claim is classified, i.e. the specific type of claim (burglary, flood, fire, car accident, damages by third parties, etc.) is determined.

For the purposes of this example, we will consider only two types of claims: burglary and fire. In case of burglary, the claim is again sent to the policies department where, based on the police report, the total value of the stolen objects is calculated, the payment limit is established and an estimate is made of how much the insurance company should pay. In case of fire damage, the process is more complicated. To deal with such cases, the insurance company resorts to a loss adjuster company which will be the one responsible for making an estimate of what needs to be paid. In the example, the loss adjuster, which uses a workflow engine to drive its business processes, provides an entry point (API) which the insurance company can invoke. Through this interface, the loss adjuster receives the necessary data and triggers its own business process.

This process consists of checking the property (i.e. who is the legal owner of a building), arranging a meeting with the client, visiting the damaged property, comparing with similar cases or, in case of major disasters like floods or earthquakes, determining what other sources of payment may need to be considered. From that, a cost estimate is made, which then is forwarded to the insurance company. They, using a similar mechanism, can incorporate this step into their own business process. After the estimation is completed, the payment is made, the corresponding records updated (so that a customer isn't paid several times for the same claim), and the claim settled.

This example shows how to introduce the loss adjuster process as an element of the overall claim processing procedure, even if the loss adjuster is an entirely different company.

The practical questions which arise when implementing such a virtual process can be best answered by following the proposed model. Thus, the overall goals for the process are part of the Virtual Enterprise. For instance, if the goal is a reduction of the claim processing time, this can only be expressed in relation to the Virtual Enterprise. The monitoring mechanisms cannot work if limited to one participant, therefore they should be part of the global agreement between the participating companies. All these agreements and the way information is distributed and accessed by the partners form the Virtual Enterprise. Similarly, when concrete queries arise, the system needs to have some sort of user identifier so that the information is given to authorised parties. Who the authorised parties are is part of the description of the Trading Community. The same can be said of the physical distribution; each element of the process is specified (owning partner/APIs/URLs/etc.) and listed in the Trading Community.

2.3. Applicability of the model

We believe Trading Communities, Virtual Enterprises and Virtual Business Processes are a very powerful approach to interpret and identify the needs of a wide range of electronic commerce practices. For instance, in the case of retailing, a company can provide a much more sophisticated product by outsourcing aspects of the operation which are not central to its activities. A common example is companies offering products (books, CD, flowers) without actually handling (producing, storing or delivering) the products themselves. Most of the handling is left to companies providing specialised services, which allows significant reduction of the operational costs. The Virtual Enterprise model naturally captures such scenarios by simply having the handling services incorporated as activities within the business processes of the company selling the products, as shown in Figure 2.1.

3. ADAPT Architecture

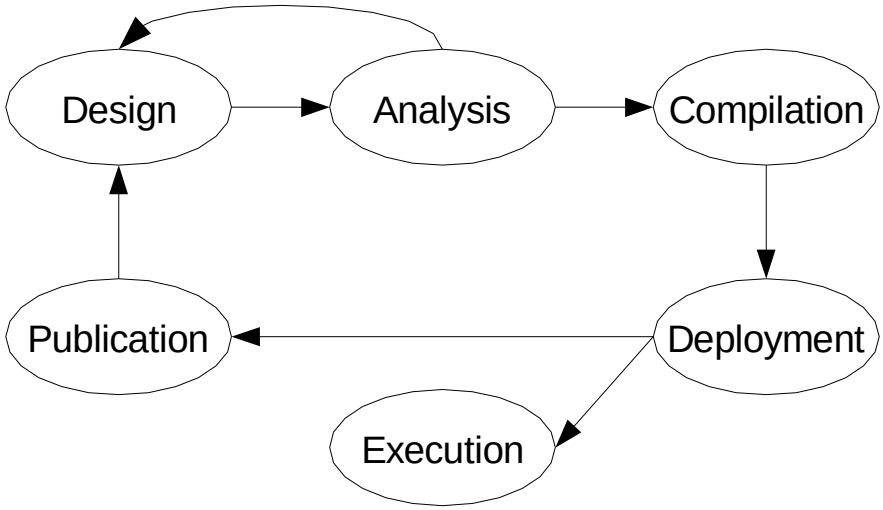


Figure 3.1: The life cycle of a Composite Service

3.1. Life cycle of a Composite Service

The life cycle can be divided into six steps, as shown in the above picture. Here are short descriptions of these steps:

Publication
Each participant in a Trading Community (TC) publishes the services it wants to make available to the community. These are both Basic (BS) and Composite Services (CS), and the details about them are put in a catalogue. One example of a catalogue standard is the UDDI registry, but we don't intend to limit ourselves to a specific technology or implementation at this point. The participants can browse the catalogue for service descriptions, including how to interact with them, etc. The description could be a WSDL file, but possibly augmented with further information. It is too early to say exactly what other information this will contain, but we expect this to become apparent as the BS and CS specifications evolve [WP1 & WP2].

Design
Based on the services published by the Trading Community, Composite Services can be built. This is done using a visual composition tool [Deliverable D14]. The tool reads the catalogue containing the available services, and makes them available to the service designer. Some possible standards the composition tool may use are BPEL4WS, WSCL or XLANG. This doesn't necessarily mean one of these standards will be used for the tool's internal representation, but that the tool should be capable of importing definitions conforming to one of these standards. These questions will be addressed as part of WP2.

Analysis
When creating a Composite Service, it must be verified to ensure that it's well formed. The design tool will be complemented with an analysis tool, which will allow different properties to be checked on the composite level [Deliverable D8]. A number of these properties are mentioned in the technical Annex.

The service can then be revised based on the results of the analysis. This ensures that the CS will conform to any constraints. The two steps Design and Analysis form the main development cycle of Composite Services.

Compilation
The compiler transforms the service description into an executable format. Syntactical checking is made, like type-matching, etc. [Deliverable D14].

Deployment
Since it has to be possible for the CS to be executed entirely distributed, a deployment step needs to split the CS into different parts, which are to be executed at different sites. These parts are assigned to the appropriate site, and configured to make sure the control flow between them is correct. See Chapter 4 for more detailed information [Deliverable D14].

Execution
At last the CS is executed by the (distributed) execution engine. Note that the flexibility of the model allows all the execution steps to be run on the same server, if all the integrated BSs/CSs indeed belong to a single TC participant. The execution engine is a part of the “CS Platform” container, and each of these controls the transition from its predecessor, the steps to be executed within its own environment and the transition to its successor [Deliverable D14].

During execution, many forms of adaptability can be applied. For instance, a task could migrate to a more suitable server if the configuration changes at run-time, look for alternative execution paths or parallelise parts of its operations to balance load.

Closing the life cycle
The newly developed CS can of course, in turn, be made available to the Trading Community through publication in the catalogue. This closes the life cycle of a Composite Service.

3.2. Global View

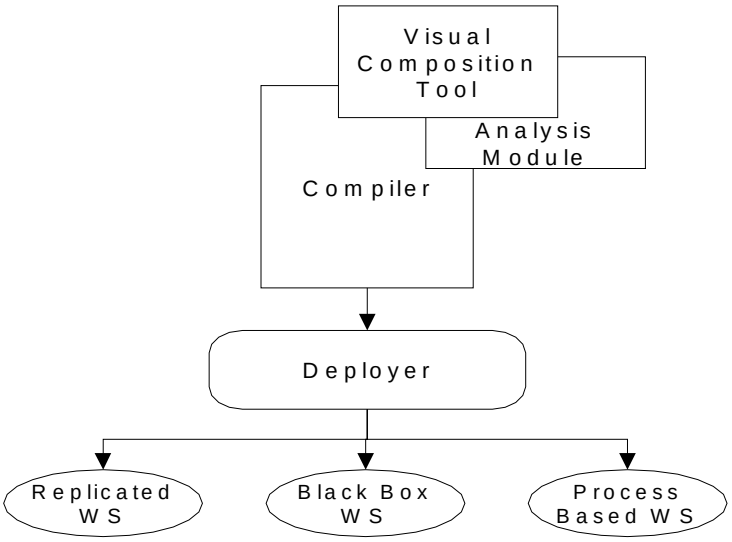


Figure 3.2: Global view of CS development and enactment

The figure above shows how a CS is developed, deployed and executed. The visual composition tool will be an integrated development environment for CSs. It will include the analysis module and the compiler. That means the tool will produce service descriptions in an executable format.

Once a CS is available in an executable format the Deployer analyses it and splits it into modules that will be executed at different sites according to where the constituent WSs are located. These sites will enact the CS in a completely distributed manner.

There are three types of WSs that can be part of a CS:

- ADAPT replicated WS
- Black Box WS
- Process-Based WS

In the case of an ADAPT replicated Web Service (developed in WP1) there is a “CS Platform” available at the corresponding site. This platform is responsible for calling the local Web Services and activating the next site in the control flow. The module is therefore configured to ensure the proper control flow. Then the module can be installed directly on the CS platform at the appropriate site.

Black Box Web Services (e.g. Amazon, Google) don't have an associated CS platform. Therefore the call to these Web Services has to be incorporated into a module residing at another site. This is not a problem since the Web Services called by a CS platform need not be local.

A Process-Based WS may have the same capabilities as the ADAPT CS platform. This way, an existing workflow might be incorporated into a Composite Service.

4. Distributed Enactment of a Composite Service

This section is intended to describe the high level architecture of a Service Enactment Coordinator (SEC) for Web Services. The function is to coordinate the execution of Composite Services (CSs) where multiple Basic Services (BSs) and CSs are combined to form a business process. It will also provide methods for run-time reconfiguration of these services to address fault tolerance and adaptability issues. The SEC will be a completely decentralised system acting on a peer-to-peer basis to control the execution of the composite service.

4.1. System Architecture

The system is composed of a number of SEC nodes, coordinating the execution of CSs by invoking BSs, Web Services and other CSs. There are administration nodes present to perform actions such as CS deployment and clients which are able to invoke the CSs as Web Services. The SECs act in a peer-to-peer based manner to control the execution of the CS by informing the other nodes when certain events occur.

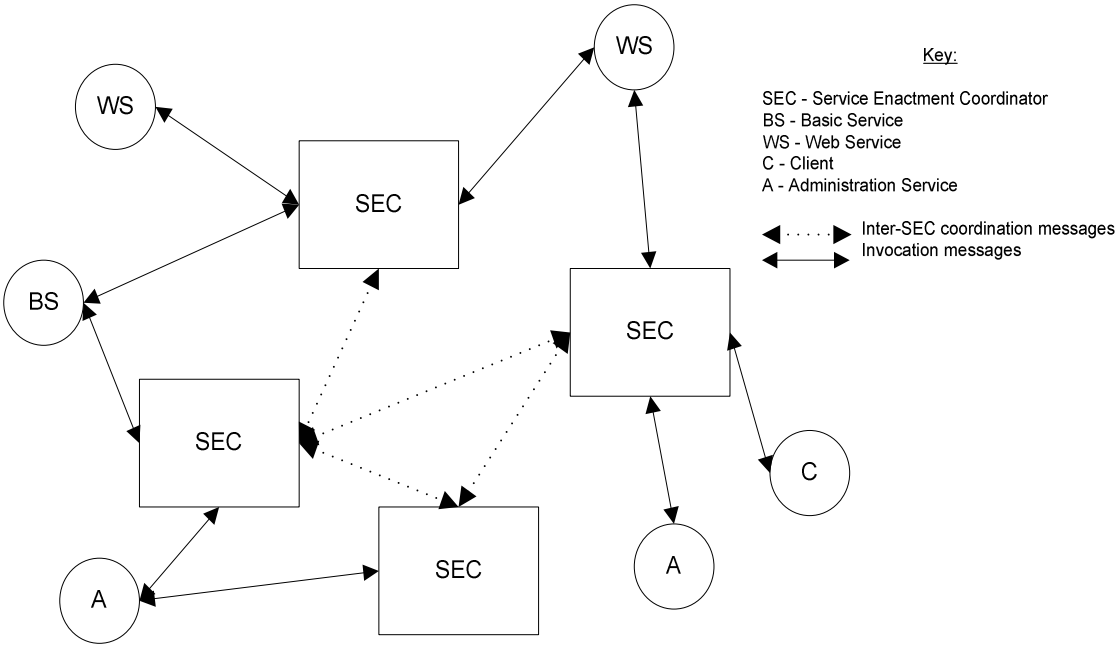


Figure 4.1: System Overview

The SEC provides four interfaces through which interactions can be performed:

- *Coordination Interface*: this interface is used for inter-SEC communication to coordinate the execution of CSs, when the responsibility of coordinating the business process is split over multiple SECs. Through this interface the SEC can request, send and receive information about events in other SEC nodes, within different parts of the process. For example, the starting of a process or the completion of a task with an exception.
- *Invocation Interface*: through this interface the SEC initiates Web Service invocations. These invocations could be to ADAPT BSs, CSs or existing Web Services.
- *Initiation Interface*: Through this interface a client can initiate a process by making a Web Service invocation.
- *Administration Interface*: This interface is used for deploying process definitions, monitoring process instances and adapting process instances.

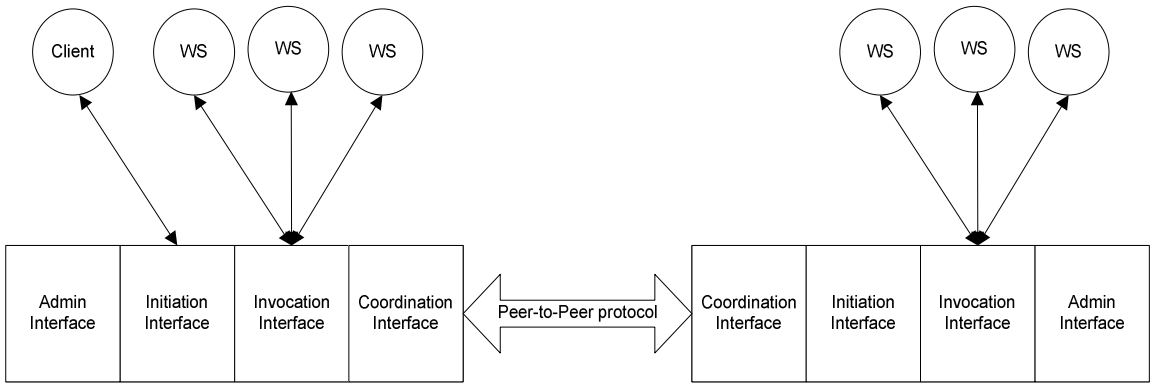


Figure 4.2: SEC Interfaces

4.2. Task Model

A CS in execution can be thought of as a process consisting of one or more tasks. Each task corresponds to one Web Service invocation which could be a BS, an existing Web Service or another CS. We have developed a task model to capture the properties of a Web Service and describe interactions between services from the aspect of CSs. The task model is based on the OPENflow task model [1] but modified to suit Web Services.

The designer of a CS is restricted to specifying process definitions which adhere to the task model. At deployment time the process definition is divided between multiple SEC nodes who coordinate the execution through the coordination interface (Discussed further in Section 4.5).

Some of the salient points of the task model are described below:

- *Single input set*: a task can be started with one message containing multiple parts.
- *Alternative input sources*: A task can acquire a given input from more than one source. This is the principle way of introducing redundant data sources for a task and for a task to control input selection. Inputs from different sources can have priorities attached to them allowing for default values to be overridden if other data becomes available in time. When the complete input set is available the highest priority parts that are available will be used to invoke the task.

- *Alternative outputs*: A task can terminate in one of several states producing distinct outcomes. One of these outcomes will be considered normal, corresponding to an output message type in WSDL and the others will be considered abnormal, corresponding to the fault message type in WSDL.
- *Compound tasks*: A task can be composed from other tasks. This is the principle way of composing a CS out of other CSs and BSs. This also allows abstraction at the design phase.
- *Genesis tasks*: A genesis task is a placeholder for a task structure and is used to allow run-time instantiation of tasks. This allows the execution of repetitive and recursive tasks as well as enabling the system to only instantiate those parts of a large process which are strictly necessary.

The tasks described in the model can have restrictions placed on their execution order in terms of dependencies on other tasks. The types of these dependencies are described below:

- *Data Dependencies*: A task can be dependant on receiving parts of its input data from other tasks. This indicates that it cannot be started until this data becomes available.
- *Temporal Dependencies*: A task cannot be executed until another task is in a particular state. This could be that a task has terminated in a particular state or has started execution.

Figure 4.3 shows an example of a compound task. For task t_2, I_1 is the input set and O_1 & O_2 are the output sets. Input set I_1 is composed of two parts, i_1 and i_2. Once both of these parts are available, the task will be started. On completion of the task either O_1 or O_2 will be produced, each containing one part, o_1 and o_2 respectively. The solid lines represent data dependencies, for example the input of task t_3 is dependant on the output set O_1, containing o_1, being produced by t_2. The dotted line between O_2 and I_2 represents a temporal dependency meaning that task t_5 is not dependant on data from task t_2 but it will only be executed if t_2 produces output set O_2. Task t_1 is an example of a compound task as it is composed of other tasks $t_2 \ldots t_5$.

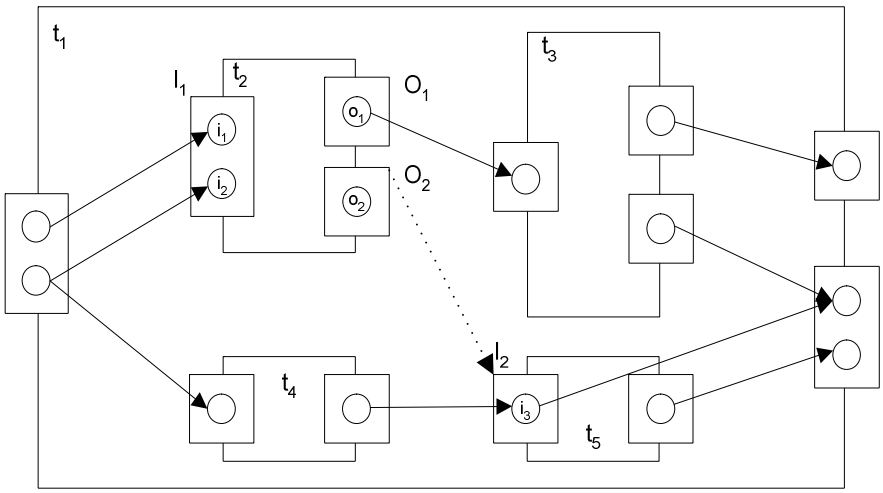


Figure 4.3: A compound task

4.3. Implementation Approach

The system will be built using J2EE technologies wherever possible, coupled together with stand-alone Java applications where explicit thread control is necessary. One approach to implementation is described below and shown in Figure 4.4.

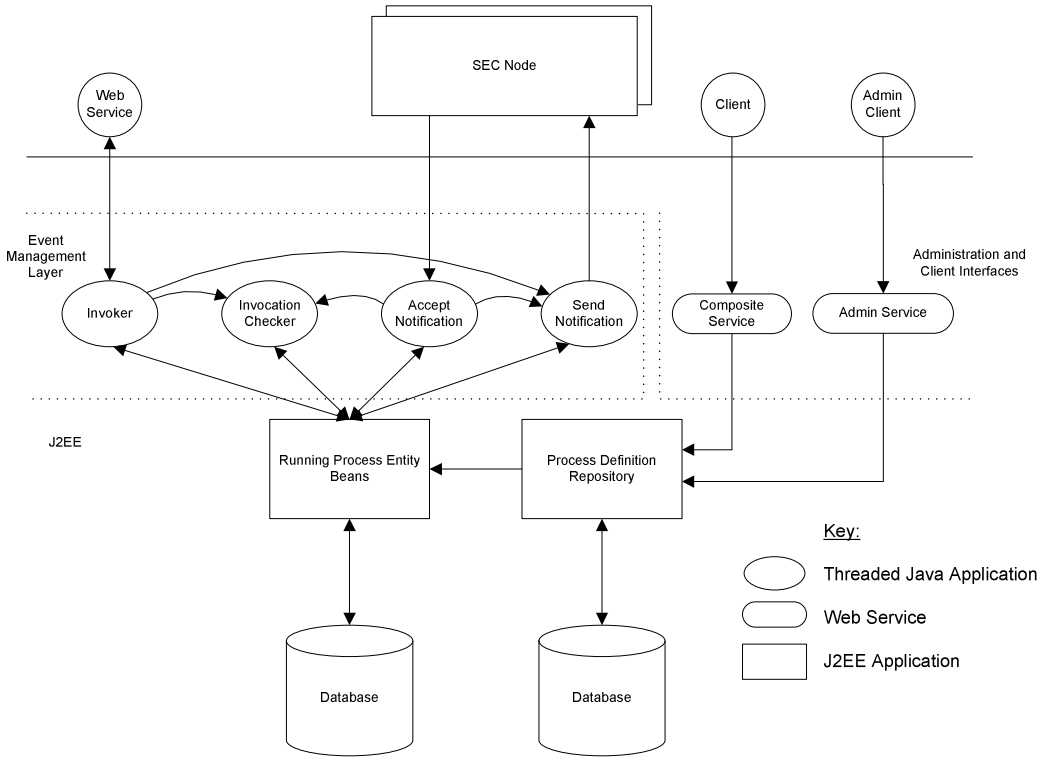


Figure 4.4: Implementation Approach

- *The J2EE layer*: The J2EE layer maintains the information about the running processes in the form of entity beans with relationships between them. These entity beans are created from the data stored in the process definition repository at invocation time. The data stored in the repository relates to a process definition rather than an instance of that process. Both the running process data and the process definition repository will be mapped onto a database in traditional J2EE architecture.
- *The Administration and Client interfaces*: These are available through two Web Services, one for administrative purposes and one for client invocation. The purpose of the Administration service is to provide an interface for deploying and un-deploying process definitions, monitoring the execution of a process instance and adapting process instances. The service stores the process definitions in the process definition repository and may also create a Web Service as an endpoint for the definition which can be invoked by a client. When a client makes an invocation of a CS, entity beans describing the process definition – in terms of methods to be executed and the dependencies between those methods – are created. The Event Management Layer uses the entity beans to control the execution of the CS.

- *The Event Management Layer*: The Event Management Layer is responsible for controlling the execution of a process instance once a client has made the initial request. There are a number of parts to this layer, all of which are multi-threaded to prevent blocking:
    - Invoker: The invoker makes the calls to external services (CSs, BSs or existing Web Services) which correspond to tasks in the process. It also collects the responses and populates the data structures in the J2EE layer with the corresponding data.
    - Invocation Checker: This component is responsible for checking the dependencies of a task, which describe when the task can be executed. When all the dependencies of a task are fulfilled the Invocation Checker informs the Invoker which executes the task.
    - Send and Accept Notification: These components are used for communicating between different instances of the SEC. The accepting application receives data relating to events which have occurred at different nodes and requests for data relating to events on that node. The sending application is responsible for sending the data to the other nodes who have registered interest in such events.

  To avoid unnecessary database scanning by the Invocation Checker and the Send Notification applications, an event queue is used in these applications. When an event occurs, such as a response is received by the Invoker, the details are put on the event queue of the Checker and the Notifier. This increases efficiency as the only dependencies that the Checker needs to check are those that have just been updated. The event queue is discussed further in Section 4.10.

4.4. Lifecycle of a service

This section is intended to describe the lifecycle of a service according to the steps detailed in Section 3.1.

4.4.1. Design

The author of a service will design it using a graphical composition tool residing on a client machine. The tool will allow the browsing of services published in the catalogue, and composition based on combining these services. The graphical language will be expressive to allow the user to describe a process in terms of tasks (services to be invoked) and the dependencies between them.

4.4.2. Analysis

The graphical tool will provide functionality for verifying the correctness of a process definition. One form of analysis which will be performed will be checking that the tasks are invoked in a legal order as specified in a “conversation language” exposed by the service.

4.4.3. Compilation

The graphical tool will output a description of the process which can be deployed in the SEC using the administrator interface.

4.4.4. Deployment

The act of deploying a CS will have the effect of storing a process definition in the repository. Deployment will also cause a Web Service to be dynamically created and exposed, allowing clients to invoke the process. This issue is discussed further in the Section 4.5.

4.4.5. Execution

A SOAP endpoint will be provided for clients to invoke the service, and a WSDL description of the interface will be made available. When a client invokes the service, a concrete representation of the process will be created on a per instance basis. This will take the form of entity beans representing the tasks (Web Services to be invoked) and the dependencies between them. These beans will store the data necessary for the Event Management Layer to coordinate the process execution adhering to the dependencies specified by the designer. On completion of the process the results is returned to the client and the beans representing the invocation is removed.

4.4.6. Removal

It will be possible to un-deploy the CS from the administration interface. This will involve removing the process definition from the repository.

4.5. Deployment

Deployment of a service is achieved using the administration Web Service. This provides functionality for taking the output of the graphical design tool and deploying a composite service. Responsibility for coordination of a CS can be split over multiple nodes. In such cases, the process definition will be split up at deployment time and different parts deployed on different nodes. Only one Web Service will be exposed for clients to invoke. The Web Service will be exposed by the SEC designated as “primary” for that CS. In most cases, this will be the SEC which controls the first task in the process definition. The act of deployment is described further below:

1. Update the Process Definition Repository: The details of the process in terms of tasks and dependencies are stored in the process definition repository. Only one copy of a process definition can be stored per composite service and an exception will be thrown if duplicates are added. The process definition holds all the details about the process which are necessary to invoke it, except the input parameters. For example, it will hold the endpoints of the Web Services that will be invoked along with the dependencies placing restrictions on when that Web Service can be invoked.

2. Create a Web Service as an endpoint for the composite service: Deploying a composite service also must make the service available to clients. This will be done by providing a Web Service which “fires” the process in the SEC. There are two options available for the creation of the Web Service which acts as an endpoint for the CS:

    - Dynamic Creation of Web Service: Apache Axis provides functionality for dynamically creating and deploying new Web Services at run-time. This

feature will be investigated with the intention of being able to create a specific Web Service for each CS which is deployed. This method is similar to stub generation in other distributed applications.

- Generic Web Service: If the features provided by Apache Axis are not deemed suitable, a generic Web Service will be written to allow clients to invoke a CS. This will involve implementing a listener which receives SOAP requests and based on the request instantiates the correct process definition from the repository.

4.6. Scalability

The proposed system is a decentralised invocation coordinator which is believed to be scalable. The designer of a service is able to specify how many nodes they wish the process to be divided between. For a simple process there could be one single node coordinating the execution. For complex processes the designer could choose to run parts of the process on different machines, with one possible separation being along organisational boundaries.

The system acts in a peer-to-peer based fashion informing the other nodes when certain events occur. Each distinct node is then responsible for assessing the impact of these events. It will be a run-time option whether to send single notifications about events or to wait for multiple notifications to need sending. This is likely to be an application specific decision.

More instances of the invocation coordinator can be started on different nodes allowing future deployment of CSs on these. It will also be possible to move CSs (that are not currently executing) from one node to another. When coordination of a CS is moved, it will be necessary to inform the other nodes which are coordinating other parts of the execution so that the notifications reach the correct nodes. The movement of running services is discussed in Section 4.7.

As well as being able to move entire process definitions from one node to another it will also be possible to move coordination of individual tasks to another node. The administration interface will provide a mechanism to achieve this which will involve removing the dependencies on these tasks and requesting notifications from the target node.

4.7. Fault Tolerance

To achieve fault tolerance, it is possible to replicate the system as a whole or in parts. System wide fault tolerance could be achieved using passive replication. Due to the potential non-determinism of the services being invoked, an active replication strategy is not appropriate. For example, a ticket booking service should only be invoked once per client. However, a passive replication scheme with updates being sent to the backup nodes would be possible. The notifications would be sent using the notifier interface, but sent to a replica instead of another active SEC node.

Instead of replicating the system as a whole, it may be desirable to replicate parts of the system individually giving a finer grained control over fault tolerance. This could be achieved using the techniques described below:

- J2EE Layer: The J2EE layer will be built on top of the Basic Service Platform (BSP) being developed in ADAPT and will be classed as a Basic Service Platform Service (BSPS). This will allow use of the facilities provided in the BSP for replication of entity beans to provide fault tolerance at this level.

- The Event Management Layer: This layer holds no state and therefore does not need to be replicated. However, thought needs to be given to the impact of each application in this layer failing, or the node failing as a whole. At present we are only considering fault tolerance with respect to crash failures. Should the invoker fail, any tasks whose state is running (service has been invoked but not returned) may not be able to return the results of the invocation (assumes request-response model). If these tasks are the invocation of Basic Services, it will be possible to invoke the service again when the invoker comes back up as BSs provide exactly-once semantics. The situation where the services being invoked are not BSs needs investigating further. If the Invocation checker failed, recovery is trivial. When the component is started again it must check all the dependencies to see which tasks can now be executed. This is likely to be an expensive database operation so normal operation will be suspended until it has completed. The Send Notification and Accept Notification applications are responsible for inter-SEC communications. These communications may be either synchronous or asynchronous, which will be decided at a later stage. Should the communications be synchronous, the acknowledgements should be "end-to-end" where the receiver only acknowledges once the notifications have been persistently stored. Sequence numbers will be used to detect duplicated messages. If the communications are asynchronous, JMS could be used to store them persistently and deliver them when the receiver comes back online.

- Database: As the SEC will be built on top of the BSP the DB will be replicated as described in deliverables D1 and D5.

Another fault tolerant aspect of the system could make use of the location transparency of J2EE. It would be possible to run the Event Management, J2EE and database layers on distinct machines. Should the Event Management Layer fail, another node could be started and perform the job of the failed node. This would require a complete scan of the database to determine which tasks are able to be invoked, and issues exist about the exactly-once semantics provided by BSs in this situation.

4.8. Reconfiguration

The SEC is designed to be reconfigurable to allow load balancing, changing the definition of a process at runtime and to respond to failures. In the task model described earlier, implicitly "upstream" tasks do not know about "downstream" tasks. It describes a process in terms of downstream tasks being dependant on certain aspects of upstream tasks, but as soon as an upstream task has completed it is unaware of what is happening to its output. This structure allows for simpler reconfiguration, as it is possible to add and remove tasks at run-time. As long as a task has not yet started, it is possible to remove it. This also involves removing all downstream dependencies relating to the process. A task can be added at any time during the running of a process. The dependencies of tasks can only be updated if they have not yet started [2].

It is possible to move responsibility for coordinating tasks from one node to another, to achieve load balancing or adaptability (discussed below). To achieve this, a notification will be sent to the target node asking it to add a task with certain dependencies. At the same time, the source node will request a notification with the results of the tasks execution and the target node will request notifications for the tasks dependencies. It will be possible to move responsibility for single tasks, whole processes or subsets of processes. If a subset is moved, notifications will be requested for all actions outside the subset to allow the successful coordination of the process.

4.9. Transactional Support

Internally the SEC will make use of the JTA to ensure that updates of tasks and dependencies are carried out in a transactional manner. As mentioned earlier, this will also apply to the notification applications with respect to end-to-end acknowledgements.

A CS can be transactional, although initially, application level transactions will need to be designed and developed explicitly by the service designer. At a later date, advanced transaction models will be investigated and possibly integrated into the SEC. These are described in deliverable D5.

When implementations of WS-C and WS-T become available, it will be possible to invoke services in a transactional manner. These will be integrated when available.

4.10. Adaptability

There will be a number of adaptability considerations when developing the SEC. These are described below:

1. Equivalent Services: The notion of services which are semantically equivalent but have different interfaces is an example of application level adaptability. An example of this is a flight booking service for British Airways and Air France. A holiday booking service should be able to use either of the services with minimal development effort. Work done in the SELF-SERV project showed that this was possible by providing a mapping between the physical interface to a service and the interface to a container holding multiple equivalent services. As related work to the SEC containers will be developed as BSPSs to hold equivalent services for use in composite services or on their own.
2. Priority based Event Queue: It may be possible to assign a priority to events related to high priority processes. This would result in these events being dealt with first and other events being dealt with eventually. Thus, high priority tasks would receive the best possible service at the expense of lower priority tasks.
3. Load Monitoring: The SEC may have a load limit specified on a per node basis. If this limit is exceeded the notification applications will be used to move running processes to other nodes. The selection criteria for moving a process is unspecified at this time. This functionality could provide methods for reserving capacity on certain nodes for higher priority customers.
4. High priority jobs: This is similar to 3, but the movement of processes would occur at a different time. Should the SEC receive a request from a high priority customer, some

processes could be moved to other nodes to allow the high priority job to access more resources. Another option available could be to suspend some processes instead of moving them. This would involve waiting for the current tasks to finish execution and not starting any new tasks for this process.

4.11. Summary

This section has described the design and one possible implementation of a distributed Service Enactment Coordinator (SEC). The purpose of the SEC is to orchestrate the execution of multiple services into a Composite Service. Coordination of Composite Services can be distributed across multiple SEC nodes which aids scalability and fault tolerance. The SEC is designed to allow reconfiguration of Composite Services at run-time giving flexibility and adaptability. Other adaptability issues are being addressed by attempting to utilise semantically equivalent services and give priority for certain clients.

5. Centralised Composition and Enactment (in BioOPERA)

BioOPERA is a workflow engine, specialised for cluster computing. The technical details of how it works internally are out of scope here, and only the conceptual aspects will be mentioned.

OPERA (Open Process Engine for Reliable Activities) doesn't have version numbers. The present version is mainly being tested with bio-chemical applications (like comparing DNA sequences), hence the name. But this doesn't mean the system is limited to applications of a specific kind.

A very useful feature of the system is that the processes designed in the graphical tool can be made available as Web Services. This is one way of making “Process-Based WSs” available as composable components in ADAPT.

The following sections explain how the steps in Section 3.1 are handled in BioOPERA.

5.1. Design

The higher-level process design can be used to combine tasks, and order their execution to depend either on control flow or data flow (which implies control flow). The tasks are associated with underlying programs, with the novelty that also SOAP calls to (external) Web Services now can be integrated this way. These can be tested individually before being integrated in the higher-level process.

The main advantage of the system is that both the process design (the template) and the execution statistics (the instances) are in persistent storage. This is very useful for measuring availability and to enable load prediction/balancing. With the integration of a more advanced scheduling mechanism, the status of a selected set of WSs could be periodically polled and the statistics stored for future use. In the decentralised architecture, this kind of performance tracking becomes very complex (where should the DB be kept?).

5.2. Analysis

This step does not apply in BioOPERA.

5.3. Compilation

BioOPERA uses an internal representation called OCR (OPERA Canonical Representation).

5.4. Deployment

BioOPERA is using an integrated Servlet Engine (Tomcat) to expose its processes and administrative functionality as Web Services. So any sub-process, as well as the composite services, can be deployed and made available to the outside.

5.5. Execution

Once a process has been deployed, it can be called like any other Web Service. The workflow engine executes the tasks (and their underlying WS operation calls) and keeps track of the generated statistics. A nice feature is the separate administrative Web Service interface that allows for progress tracking, which is very useful when the process involves many steps and is expected to be long-running. Details like status, input/output parameters, execution statistics, etc. can also be queried.

The system allows for smart recovery. If a server crashes, the finished tasks don't need to be repeated when the server is functional again. Instead the execution can be resumed by re-starting the tasks that were aborted at the time of the crash.

5.6. Closing the life cycle

The composite process, now having been exposed to the world, can then have its existence made known to a wider audience, by having its API details, etc. put in a registry.

6. Challenges in the Web Services field

6.1. Alternative standards

The hype around UDDI, WSDL and SOAP has eclipsed many parallel (and previous) efforts along the same direction. As a result, there is an obvious trend towards systems that are UDDI, WSDL, and SOAP specific. Such trend thrives on the myth that Web Services are an accepted and dominant standard. However, it is by no means clear that Web Services will displace existing technologies. It has been mentioned in the literature that up to a few hundred competing B2B standards may coexist. Examples of such established standards that will not simply go away are the Electronic Data Interchange (EDI), used in manufacturing, and SWIFT, used in the financial world. Such standards, which represent successful efforts in the area of e-commerce, fit very well with the document based approach discussed earlier (see Section 1.4). It is not clear to what extent the service approach is useful for such standards except at the lowest levels of the software hierarchy for B2B exchanges.

The impact of these alternative standards on ADAPT is that, to remain relevant, the overall architecture will have to be agnostic towards B2B standards. This implies a generic and open architecture that can be extended and used in different ways depending on whether the emphasis is placed on conventional services or on document/conversation-based exchanges.

There are, however, dangers associated with this design decision. Generality is certainly a solution to the lack of standardisation. If no standard dominates, a generic architecture can be easily adapted to whatever specification comes along. Unfortunately, generality comes at a price and undermines the standardisation efforts. The reason is that, in practice, Web Services are not being built from scratch. They are being built on top of existing multi-tier systems, systems that are all but general. Hence, many Web Services are biased from the start towards specific protocols, representations, and standards, i.e. those already supported by the underlying middleware. The necessary generality will only be achieved, if at all, by additional software layers. Indeed, Web Services add even more layers to the already overly complex multi-tier architecture typical of B2B interactions. Aiming for generic systems will make matters even worse. Translation to and from XML, tunnelling of RPC through SOAP, clients embedded in Web servers, alternative port types, and many of the technologies typical of Web Services do not come for free. They add significant performance overheads and increase the already extreme complexity of developing, tuning, maintaining and evolving multi-tier systems. An important part of our efforts in ADAPT will be to understand these overheads and how to reduce them as much as possible.

This is where we face a difficult dilemma. The proliferation of competing standards, whether based on the same syntax (XML) or not, will require additional software layers to address interoperability problems. Even in those cases where a single set of standards can be used, Web Services are being almost universally built as additional tiers over existing middleware platforms. Unfortunately, multi-tier architectures are already too complex and cumbersome. Adding more layers will not make them any better and the sheer complexity and cost of such systems may prevent the widespread use of the technology. Without widespread use, standards will fragment even further, thereby making it almost impossible to produce sufficiently generic platforms which, in turn, increase the development and maintenance costs. At this stage in the project, we have tried to be as generic as possible.

6.2. Web Services in conventional applications

One of the drawbacks of Web technology is that it is still too tightly related to humans and browsers. Web Services have computers as their main users and are not based on browsers at all. Nevertheless, many of us still think about Web Services in the same terms we think about a Web browser: our first image of a Web Service is that of an interactive one. Maybe with the execution driven by a computer instead of a human, but interactive nonetheless. The examples available in the literature, and not only in the research literature, corroborate this bias. We have all seen many different variations of the travel planner service, which has been misused so often that it should become a standard of its own. Flight reservation, car rental and hotel booking, or buying a travel guide, are all examples of interactive services. Moreover, all these services are typical Business to Consumer (B2C) interactions, rather than B2B exchanges. This is an interesting development since Web Services are being pursued mainly because of their potential impact on B2B not on B2C.

There are of course practical advantages in using Web Services interactively and on-line. One example often mentioned are applications that embed a search engine by using Web Services. Other examples are applications or operating systems that send periodic bug reports to the software vendor using a Web Service, applications that automatically download and install patches, or systems that use a remote service to provide functionality that cannot be provided locally (e.g. access to a very large database that is not locally available). These are all very appealing scenarios, but it is not immediately obvious that Web Services are the best way to implement them. In some cases (e.g. information flow from the application to a server), this functionality is already being provided without Web Services and it is not clear that switching to Web Services will bring any significant advantages. In other cases, it does not seem reasonable to bloat the application with the whole machinery of Web Services to implement just a fancy feature. If the operating system eventually provides support for accessing Web Services to all applications, then this may make sense but we are quite far from that stage. Perhaps an even more decisive factor is that many of the features of Web Services are irrelevant in these settings. For instance, application specific information does not need to be sent as an XML document. Likewise, interfaces used internally by a software vendor do not need to be described using WSDL (and certainly do not need advertising through an UDDI).

From a practical perspective, it is also not clear how to build applications that rely on Web Services for part of their functionality. It has been pointed out that Web Services are still plumbing for the exchange of XML documents using SOAP. For interactive and on-line use within applications, several crucial issues remain unsolved. One of them is trust: how far can the application trust and rely on external Web Services which it does not control? Another one is the fact that we do not yet understand the impact of Web Services on software design as many of the techniques for component based software development do not work with Web Services. Answers to these questions are needed before Web Services are widely used as extensions to conventional applications. To answer some of these questions, we are intensifying our contacts to projects where such issues are being or have been addressed (e.g. TAPAS, CROSSFLOW).

6.3. Synchronous vs. Asynchronous exchanges

A misleading interpretation of Web Services leads to the assumption that Web Services provide a direct link between middleware platforms of different corporations. Most conventional middleware platforms are implemented on top of RPC: TP-Monitors, Object Monitors, CORBA implementations, and even message-oriented middleware. Because of its pervasiveness, RPC over HTTP was one of the first interaction mechanisms specified using SOAP. By doing so, a Web Service becomes an extension of existing multi-tier architectures but with the client residing now at the other side of the firewall and behind a Web server. Since B2B services are implemented using multi-tier systems, being able to use RPC through SOAP is seen by many as a gateway to directly interconnect the IT infrastructure of different companies.

There are several problems with such an interpretation. One of them is that RPC results in a tight integration that makes components dependent on each other. This is unacceptable in any industrial strength setting, especially if the components belong to different companies. Not only would the complexity of the resulting system increase exponentially, the mere act of maintaining the system would become a coordination nightmare with tremendous costs. This is why the vast majority of B2B interactions happen asynchronously and in batch mode, not interactively. Rather than direct invocations, requests are batched and routed through queues. Responses are treated in the same way. The actual elements of the interaction (the client and the server, to simplify things) are kept as decoupled as possible so that they can be designed, maintained, and evolved independently of each other. Systems based on EDI and SWIFT are, again, good examples of the typical loosely coupled architectures of B2B systems.

Proof of this is the strong trend towards asynchronous SOAP. The fact that the most widespread use of SOAP is to tunnel RPC does not contradict this statement. Many queuing systems are implemented on top of RPC. A message is placed on a queue and a daemon makes an RPC call to another remote daemon that takes the message and places it on the receiving queue. Technically this is not only possible, it is also a reasonable way of implementing B2B interactions. From the point of view of Web Services, however, it means that the Web Service description will be far more complex than an RPC invocation encoded as an XML message. The description may have more to do with the interaction mechanism (the queues) than with the service interface itself. In fact, in many cases, the actual service interface will not necessarily be made explicit. For instance, a service may simply indicate that it is a queue that accepts EDIFACT purchase order messages without describing such messages (since their format is already known to those using them). We are already busying ourselves with this issue in ADAPT, particularly looking at it from the point of view of QoS and adaptation.

6.4. UDDI and dynamic binding

An UDDI registry is conceptually similar to a name and directory server. There are, however, significant practical differences between the two, differences that tend to be ignored and lead to the assumption that an UDDI registry has the same purpose as a name and directory server. The result is the widespread misconception that dynamic binding will be a common way of working with Web Services. This is far from being the case and there are two very strong arguments against this assumption.

From the point of view of functionality, UDDI registries have been created as standardised catalogues of Web Services. The information they contain is intended for humans, not for computers. First of all, there is the problem of the semantic interpretation of the parameters and operations defined by the interface. These parameters indicate the expected type but not what the parameter actually means (e.g. a price is given as an integer but there might not be any indication of the currency used). There is also the issue of how to deal with exceptions and how to link them to the internal business processes. The service might also provide different ways to proceed depending on the outcome of intermediate operations. Only a person can make sense of this information while using it will require careful analysis and a significant design effort. Second, interactions between different companies are regulated by contracts and business agreements. Without a proper contract, not many companies will interact with each other. To think that companies will (or can) invoke the first Web Service they find on the network is unrealistic. Web Service based B2B systems will be developed by specialists who locate the necessary services, identify the interfaces, draw up the necessary business agreement, and then design and build the actual application with the Web Service either hardwired into the application code or defined as a deployment parameter.

From the software engineering point of view, dynamic binding is a double edge sword. If dynamic binding is used simply to determine the location of a well defined service, it is indeed a useful feature. Any other form of dynamic binding makes it almost impossible to develop real applications. CORBA already provided designers with very fancy dynamic binding capabilities. An object could actually query for a service it had never heard of and build a call to that service on the fly. Such a level of dynamism makes sense only (if at all) in very concrete, low level scenarios that appear almost exclusively when writing system software. Application designers have no use for such dynamic binding capabilities. How can one write a solid application without knowing what components will be called? It is nearly impossible to write sensible, reliable application logic without knowing what exceptions might be raised, what components will be used, what parameters these components take, etc. In its full generality, dynamic binding does not make sense at the application level and this also holds for Web Services. In regard to dynamic binding as a fault tolerance and load balancing mechanism, in the context of Web Services, the UDDI registry is simply the wrong place for it. UDDI has been designed neither with the response time capabilities, nor the facilities necessary to support such dynamic binding. Moreover, the UDDI registry cannot do any load balancing or automatic fail-over to a different URI in case of failures. It is simply not designed to do that. Such problems must be solved at the level of individual Web Service provider, using known techniques like replication, server clustering, and hot-backup techniques.

Thus, UDDI registries will be used by programs only to the extent that service publishing will be automatic in many systems and search over an UDDI registry will happen through specialised added-value tools built on top of the UDDI registries.

6.5. Data in XML

XML is a blessing as a syntax standard. It allows the construction of generic parsers that can be used in a multitude of applications, thereby ensuring robustness and low cost for the technology. Unfortunately, this significant advantage does not compensate for the fact that XML is a performance nightmare. There are also many data types that do not get along well with XML, e.g. anything that is binary or nested XML documents. In many cases, even if it is possible, there is no point in formatting the application data as an XML document. We have already mentioned an example: a Web Service implemented as a queue expecting EDIFACT e-mail messages does not gain much by having the message encoded in XML. In fact, it only loses performance and introduces unnecessary software layers.

XML-encoding makes sense when linking completely heterogeneous systems or passing data around that cannot be immediately interpreted. It also makes sense when there is no other syntax standard and designers must choose one. When Web Services are built based on already agreed upon data formats, then the role of XML is reduced to be the syntax of the SOAP messages involved. This is why there is such a strong demand for SOAP to support a binary or blob type. There are several ways of doing this: using URLs as pointers, as an attachment or with the recently proposed Direct Internet Message Encapsulation (DIME) protocol. Whatever mechanism becomes the norm, expect an increasing amount of Web Service traffic to contain binary rather than XML data.

The use of binary rather than XML for formatting application data has a wide range of implications for Web Services. First, it will provide a vehicle for vertical B2B standards to survive even if Web Service related specifications become dominant. In practice, Web Services become just a mechanism to tunnel interactions through the Internet, their original intended goal. The actual interaction semantics will be supported by other standards, those used to encode the data in binary format (e.g. once more, EDI or SWIFT). The question will then be whether Web Services provide enough added value to justify the overhead. Second, Web Services implemented over binary data will describe only the interaction. They cannot specify the actual programmatic interface of the service as this is hidden in the binary document and, therefore, cannot be controlled by the Web Services infrastructure. This will reduce even further the chances of having tightly coupled architectures built around Web Services. Finally, Web Services based on binary formats will increase the dependency on humans for binding to services, as much of the information needed to bind to a service might be external to the Web Service specification. Although the issue is in principle orthogonal to the architecture being pursued in ADAPT, it has clear practical implications that we are still evaluating.

References

[1] F. Ranno, S.K. Shrivastava, and S.M. Wheater, "A Language for Specifying the Composition of Reliable Distributed Applications", in *Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS-98). 1998*, Amsterdam, The Netherlands.

[2] J. J. Halliday, S. K. Shrivastava and S. M. Wheater, "Flexible Workflow Management in the OPENflow system", in *Proc. of 5th IEEE/ OMG International Enterprise Distributed Object Computing Conference (EDOC 2000)*, September 2001, Seattle, pp.82-92.