UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

A SURVEY OF WEB SERVICE TECHNOLOGIES

Michael P. Papazoglou and Jean-jacques Dubray

June 2004

Technical Report # DIT-04-058

A Survey of Web Service Technologies

Michael P. Papazoglou[1], Jean-jacques Dubray[2]
[1]INFOLAB, Tilburg University, PO Box 90153,
Tilburg 5000 LE, The Netherlands
mikep@uvt.nl

[2]Attachmate, 3617 131st Ave NE
Bellevue WA 98006, USA
jeanjadu@attachmate.com

Abstract

The Web has become the means for organizations to deliver goods and services and for customers to search and retrieve services that match their needs. Web services are self-contained, Internet-enabled applications capable not only of performing business activities on their own, but also possessing the ability to engage other web services in order to complete higher-order business transactions. Simple web services may provide simple functions such as credit checking and authorization, inventory status checking, or weather reporting, while complex services may appropriately unify disparate business functionality to provide a whole range of automated processes such as insurance brokering, travel planning, insurance liability services or package tracking. The act of building applications and processes as sets of interoperating services is enabled by means of unified service-oriented architecture (SOA). SOA introduces a new philosophy for building distributed applications where elementary services can be published, discovered and bound together to create more complex valued-added services. This article aims at providing a comprehensive survey of web service technologies, examining it usage, its relation with other technologies, the newest developments in the field, architectural models and standards. The article presents an extended architecture on the basis of whose functional layers we taxonomize research activities.

Categories and subject descriptors: H [Information Systems]: distributed information systems, H1 [Models and Principles]: Application modeling and integration
General terms: Design, Languages, Standards, Management.
Additional keywords and phrases: Service Oriented Computing, web services, process modeling and management, workflow systems, coordination and collaboration.

1 Introduction

Service-Oriented Computing (SOC) utilizes services as the constructs to support the development of rapid, low-cost and easy composition of distributed applications. Services are self-contained processes - deployed over standard middleware platforms, e.g., J2EE - that can be described, published, located, and invoked over a network. Any piece of code and any application component deployed on a system can be transformed into a network-available service. Services reflect a new "service-oriented" approach to programming, based on the idea of composing applications by discovering and invoking network-available services rather than building new applications or by invoking available applications to accomplish some task [Papa03a]. Services perform functions that can range from answering simple requests to executing sophisticated business processes requiring peer-to-peer relationships between service consumers and providers. However, services are most often built in a way that is independent of the context in which they are used, i.e. service provider and consumers are loosely coupled. At the middleware level, loose coupling requires that the "service-oriented" approach be independent of specific technologies or operating systems. In particular, services and service composition does not rely on existing programming languages. It allows systems and organisations alike to expose their core competencies declaratively over the Internet or a variety of networks, e.g., cable, UMTS, XDSL, Bluetooth, etc., using standard (XML-based) languages and protocols, *via* self-describing interfaces based on open standards. By building upon such standards, developers are given the opportunity to access systems and applications deployed over the network based on what they do, rather than on how they do it, or how they have been implemented. The visionary

promise of SOC is a world of cooperating services where applications are assembled with little effort as a network of loosely coupled services. These agile applications can support dynamic business processes that span organisations and computing platforms.

SOC is expected to have an impact on all aspect of software construction as wide as that of object-oriented programming. The premise of its foundation is that an application can no longer be thought of as a single process running within a single organization. The value of an application is actually no longer measured by its functionality but by its ability to integrate with its surrounding environment. For instance, services can help integrate applications that were not written with the intent to be easily integrated with other applications and define architectures and techniques to build new functionality leveraging existing application functionality. A new type of applications can be based solely on sets of interacting services offering well-defined interfaces to their potential users. These applications are often referred as: *composite applications*. In the business-to-business (e-business) world, service orientation enables loosely coupled relationships between applications of transacting partners, the model does not even mandate any kind of pre-determined agreements before the use of an offered service is allowed. The service model allows for a clear distinction to be made between *service providers* (organizations that provide the service implementations, supply their service descriptions, and provide related technical and business support); *service clients* (end-user organizations that use some service); and *service aggregators* (organizations that consolidate multiple services into a new, single service offering).

Services are offered by service providers: organizations that procure the service implementations, supply their service descriptions, and provide related technical and business support. Since services may be offered by different enterprises and communicate over the Internet, they provide a distributed computing infrastructure for both intra and cross-enterprise application integration and collaboration. Clients of services can be other solutions or applications within an enterprise or clients outside the enterprise, whether these are external applications, processes or customers/users. This distinction between service providers and consumers is independent of the relationship between consumer and provider which can be either client / server or peer to peer. For the service oriented paradigm to exist, we must find ways for the services to be:

- Technology neutral: they must be invoked through standardized lowest common denominator technologies that are available to almost all IT environments. This implies that the invocation mechanisms (protocols, descriptions and discovery mechanisms) should comply with widely accepted standards.
- Loosely coupled: they must not require knowledge or any internal structures or conventions (context) at the client or service side.
- Support location transparency: services should have their definitions and location information stored in a repository such as UDDI (see section-8.3) and be accessible by a variety of clients that can locate and invoke the services irrespective of their location.

Services may be implemented on a single machine or on a large number and variety of devices, and be distributed on a local area network or more widely across several wide area networks (including mobile and ad hoc networks). A particularly interesting case is when the services use the Internet (as the communication medium) and open Internet-based standards. The resulting *web services* share the characteristics of more general services, but they require special consideration as a result of using a public, insecure, low-fidelity mechanism for inter-service interactions.

Web services constitute a distributed computer infrastructure made up of many different modules trying to communicate over the network to virtually form a single logical system. Web services are modular, self-describing, self-contained applications that are accessible over the Internet. They are the answer to the problems of rigid implementations of predefined relationships and isolated services scattered across the Internet. A web service is a service available via a network such as the Internet that completes tasks, solves problems or conducts transactions.

Web services can vary in function from simple requests (for example, currency conversion, credit checking and authorization, inventory status checking, or a weather report) to complete business applications that access and combine information from multiple sources, such as an insurance brokering system, a travel planner, an insurance liability computation or a package tracking system. Enterprises can use a single web

service to accomplish a specific business task, such as billing or inventory control or they may compose several web services together to create a distributed e-business application such as customised ordering, customer support, procurement, and logistical support.

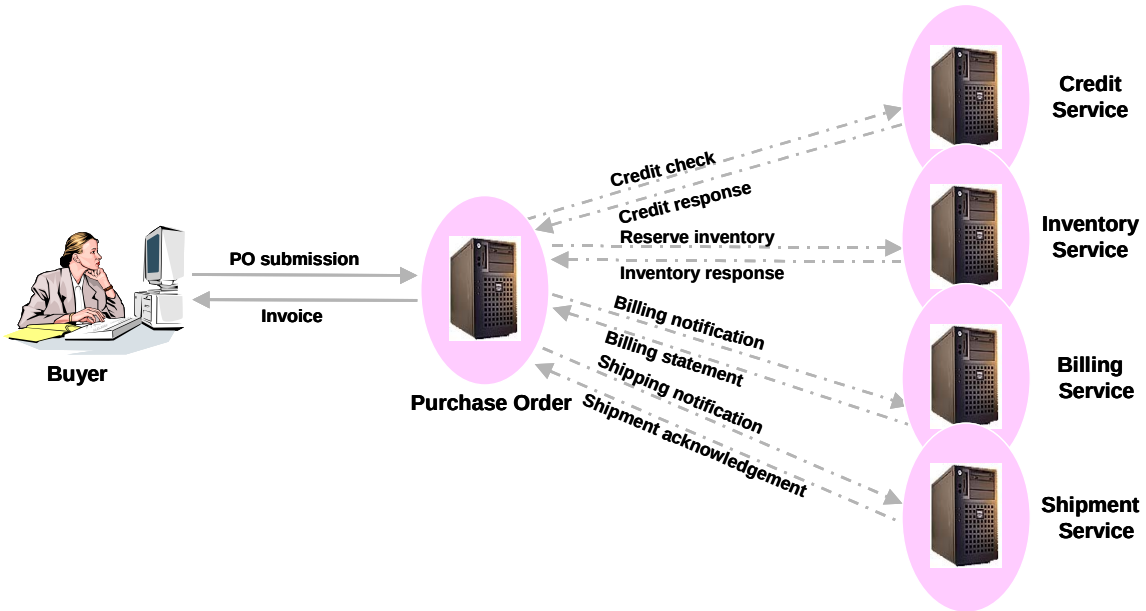


Figure 1 A purchase order application involving interacting web services.

Consider the example of a purchasing protocol. A buyer or buying organization initially creates a purchase order and sends the request to fulfil the order to a seller. The seller has a service that receives a purchase order and responds with either acceptance or rejection based on a number of criteria, including availability of the goods and the credit of the buyer. Figure 1 shows how such a purchase order process can be developed in terms of interacting web services involving purchase orders, credit checks, automated billing, stock updates and shipping originating from various service providers who can gradually package their offerings to create turnkey products. Submitting a purchase order using the Web can be represented as a complex set of interacting web services. The product order example herein has been adapted from a similar example used for orchestrating web services [Andrews03]. On receiving the purchase order from a buyer, the purchase order process initiates five tasks concurrently: checking the credit worthiness of the user, determining whether or not an ordered part is available in the product inventory, calculating the final price for the order and billing the customer, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between these tasks. For instance, the customer's creditworthiness must be ascertained before accepting the order, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfilment schedule. When these tasks are completed successfully, invoice processing can proceed and the invoice is sent to the customer.

Tracking and adjusting purchase orders due to unexpected events such as the buyer initiating a purchase order change or cancellation involves a lot of coordination work. This calls for the use of reactive services. For instance, if a single event in the purchase order needs to change or is cancelled, the entire process can unravel instantly. Employing a collection of web services that work together to adjust purchase orders for such situations creates an automated solution to this problem. In the case of a purchase order cancellation the purchase service can automatically reserve a suitable replacement product and notify the billing and inventory services of the changes. When all of these web service interactions have been completed and the new adjusted schedule is available, the purchase order web service notifies the customer sending her an updated invoice.

In another example, an insurance company may decide to offer an on-line quoting web service to its customers. Rather than developing the entire application from scratch, this enterprise looks to supplement its home grown applications with modules that perform industry standard functions. Therefore, it may seamlessly link up with the web service of another enterprise that specialises in insurance liability computations. The insurance liability web service may present a quote form to the customer to collect

customer information based on the type of the desired insurance. Subsequently, the web service would present the customer with a quote including a premium estimate. If the customer selected to buy that particular insurance policy, the system will take the customer's payment information and run it through a payment system offered by yet another company (service provider) web service. This payment web service will ultimately return billing information to the customer and to the originating company.

In summary, web services enable developers to construct distributed applications using Internet standards and any platform and programming language that is required. Once a web service is deployed, other applications and web services can discover and invoke that service. The eventual goal of web services technology is to enable distributed applications that can be dynamically assembled according to changing business needs, and customised based on device and user access while enabling wide utilization of any given piece of business logic wherever it is needed.

The aim of this paper is fourfold. First is to introduce the concept of software as a service and describe the broad characteristics and types of web services. Second is to describe the notion of the service-oriented architecture and explain how web services standards help develop distributed applications under this architectural scheme. Third is to introduce more advanced web service features such as coordination and orchestration principles, transactions and security, quality of service issues and web service interoperability problems. Finally is to introduce an extended service-oriented architecture which stratifies services by means of their functional characteristics in three broad layers: service description and basic operations, service composition, and service management. Web service research activities are also classified and discussed according to the layers of this extended architecture.

2 The Concept of software as a service

Web services are very different from web pages that also provide access to applications across the Internet and across organisational boundaries. Web pages are targeted at human users, whereas web services are developed for access by other applications. Web services are about machine-to-machine communication, whereas web pages are about human to machine communication. As terminology is often used very loosely it is easy to confuse someone by describing a 'service' as a web service when it is in fact not. Consequently, it is useful to examine first the concept of software as-a-service on which web services technology builds upon and then compare web services with web pages and web server functionality.

The concept of software-as-a-service is revolutionary and appeared first with the ASP (Applications Service Provider) software model. Application Service Providers (ASP) are companies that package software and infrastructure elements together with business and professional services to create a complete solution that they present to the end customer as a service on a subscription basis. An ASP is a third party entity that deploys, hosts and manages access to a packaged application and delivers software-based services and solutions across a network to multiple customers across a wide area network from a central data center. Applications are delivered over networks on a subscription or rental basis. In essence, ASPs were a way for companies to outsource some or even all aspects of their information technology needs. The ASP industry Consortium [ASP00] defined that application service providers are service organizations that deploy, host, manage, and enhance software applications for customers at a centrally managed facility, offering application availability, performance and security. End-uses access these applications remotely using Internet or leased lines.

The basic idea behind an ASP is to "rent" applications to subscribers. The whole application is developed in terms of the user interface, workflow, business and data components that are all bound together to provide a working solution. An ASP hosts the entire application and the customer has little opportunity to customize it beyond set up tables, or perhaps the final appearance of the user interface (such as, for example, adding company logos). Access to the application for the customer is provided simply via browsing and manually initiated purchases and transactions occur by downloading reports. This activity can take place by means of a browser. This is not a very flexible solution – but offers considerable benefits in terms of deployment providing the customer is willing to accept it 'as is'.

By providing a centrally hosted Intent application, the ASP takes primary responsibility for managing the software application on its infrastructure, using the Internet as the conduit between each customer and the

primary software application. What this means for an enterprise is that the ASP maintains the application, the associated infrastructure, and the customer's data and ensures that the systems and data are available whenever needed.

An alternative of this is where the ASP is providing a software module that is downloaded to the customer's site on demand – this is for situations where the software does not work in a client/server fashion, or can be operated remotely via a browser. This software module might be deleted at the end of the session, or may remain on the customer's machine until replaced by a new version, or the contract for using it expires. In many respects this is no different to 'traditional' methods of installing a piece of software from say a CD-ROM. However, this form of deployment from the ASP can be automated, reducing software deployment costs, though not saving the customer any hardware costs.

Although the ASP model introduced the concept of software-as-a-service first, it suffered from several inherent limitations such as the inability to develop highly interactive applications, inability to provide complete customisable applications and inability to integrate applications [Goepfert02]. This resulted in monolithic architectures, highly fragile, customer-specific, non-reusable integration of applications based on tight coupling principles.

Today we are in the midst of another significant development in the evolution of software-as-a-service. The new architecture allows for loosely-coupled asynchronous interactions on the basis of XML standards with the intention of making access to, and communications between, applications over the Internet easier. The SOC paradigm allows the software-as-a-service concept to expand to include the delivery of complex business processes and transactions as a service, while permitting applications to be constructed on the fly and services to be reused everywhere and by anybody. Perceiving the relative benefits of service-oriented technology many ASPs are modifying their technical infrastructures and business models to be more akin to those of web service providers.

The web services paradigm allows the software-as-a-service concept to expand to include the delivery of complex business processes and transactions as a service, while permitting that applications are constructed on the fly and services to be reused everywhere and by anybody. Perceiving the relative benefits of web service technology many ASPs are modifying their technical infrastructures and business models to be more akin to those of web service providers.

The use of web services provides a more flexible solution. The core of the application – the business and data components remain on the ASP's machines, but are now accessed programmatically via web service interfaces. The customers can now build their own custom business processes and user interfaces, and are also free to select from a variety of web services that are available over the network and satisfy their needs.

When comparing web services with web-based applications we may distinguish four key differences [Aldrich02]:

- Web services act as resources to other applications that can request and initiate those web services, with or without human intervention. This means that web services can call on other web services to outsource parts of a complex transaction to those other web services. This provides a high degree of flexibility and adaptability not available in today's Web-based applications.
- Web services are modular, self-aware and self-describing applications; a web service knows what functions it can perform and what inputs it requires to produce its outputs and can describe this to potential users and to other web services. A web service can also describe its non-functional properties: for instance the cost of invoking the service, the geographical areas the web service covers, security measures involved in using the web service, contact information and more.
- Web services are more visible and manageable than Web-based applications; the state of a web service can be monitored and managed at any time by using external application management and workflow systems. Despite the fact that a web service may not run on an in-house (local) system or may be written in an unfamiliar programming language it still can be used by local applications, which may detect its state (active or available) and manage the status of its outcome.

- Web services may be brokered or auctioned. If several web services perform the same task, then several applications may place bids for the opportunity to use the requested service. A broker can base its choice on the attributes of the "competing" web services (cost, speed, degree of security).

3 What are web services?

Web services that can be published to and accessed over the Internet and corporate intranets form the building blocks for creating distributed applications. They rely on a set of open Internet standards that allow developers to implement distributed applications – using different tools provided by many different vendors – to create corporate applications that join together software modules from systems in diverse organisational departments or from different enterprises. For example, an application that tracks the inventory level of parts within an enterprise can provide a useful service that answers queries about the inventory level. But more importantly, web services can also be combined and/or configured by these distributed applications, behind the scenes and even on the fly, to perform virtually any kind of (business-related) task or transaction. These applications usually already exist within an enterprise or may be developed from scratch using a Web services toolkit.

Web services can discover and communicate with other web services and trigger them to fulfil or outsource part of a higher-level transaction by using a common vocabulary (business terminology) and a published directory of their capabilities according to a reference architecture called the Service Oriented Architecture (see section-5). A web service can be a specific service, such as an online car rental service; a business process, such as the automated purchasing of office supplies; an application, such as a language translator program; or an IT resource, such as access to a particular database.

A web services toolkit exposes the useful business service in an Internet- accessible format. For instance, an IBM Web services development environment or Microsoft Visual Studio .NET toolkit may be used to expose the inventory level query application (being originally coded in say, C or Visual Basic) as a web service that can be accessed over the Internet by any other module as part of a distributed application. Consequently, the modularity and flexibility of web services make them ideal for e-business application integration. For example, an inventory web service referenced above can be accessed together with other related web services by a business partner's warehouse management application or can be part of a new distributed application that is developed from scratch and implements an extended value chain supply planning solution.

At this stage a more complete definition of a web service can be given. A web service is a platform-independent, loosely coupled, self-contained programmable web-enabled application that can be described, published, discovered, coordinated and configured using XML artefacts for the purpose of developing distributed interoperable applications. Web services possess the ability to engage other services in a common computation in order to:

- complete a task,
- conduct a business transaction, or
- solve a complex problem, and
- expose their features programmatically over the Internet (or intra-net) using standard Internet languages and protocols like XML, and
- can be implemented via a self-describing interface based on open Internet standards.

In the following we shall examine this definition more closely and deconstruct its meaning.

- *Web services are loosely coupled software modules.* Web services interact with one another dynamically and use Internet standard technologies, making it possible to build bridges between systems that otherwise would require extensive development efforts. Traditional application design depends upon a tight interconnection of all subsidiary elements, often running in the same process. The complexity of these connections requires that developers thoroughly understand and have control over both ends of the connection; moreover, once established, it is exceedingly difficult to extract one element and replace it with another. Loosely coupled systems, on the other hand, require a much simpler level of coordination and allow for more flexible reconfiguration. As

opposed to tight coupling principles that require agreement and shared context between communicating systems as well as sensitivity to change, loose coupling allows systems to connect and interact more freely (possibly across the Internet). Loose coupling also implies that a change in the implementation of the web service functionality does not require a subsequent change in the client program that invokes it, the conditions and cost of using the service and so on.

- *Web services semantically encapsulate discrete functionality*. A web service is a self-contained software module that performs a single task. The module describes its own interface characteristics, i.e., the operations available, the parameters, data-typing and the access protocols, in a way that other software modules can determine what it does, how to invoke its functionality, and what result to expect in return. In this regard, web services are contracted software modules as they provide publicly available descriptions of the interface characteristics used to access the service so that potential clients can bind to it. The service client uses a web service's interface description to bind to the service provider and invoke its services.
- *Web services can be accessed programmatically.* A web service provides programmable access – this allows to embed web services into remotely located applications. This enables information to be queried and updated in real-time, thus, improving efficiency, responsiveness and accuracy – ultimately leading to provide high added value to the web service clients. Unlike web sites, web services are not targeted at human users, and they do not have a graphical user interface. Rather, web services operate at the code level; they are called by and exchange data with other software modules and applications. However, web services can certainly be incorporated into software applications designed for human interaction.
- *Web service can be dynamically found and included in applications*. Unlike existing interface mechanisms. Web services can be assembled, even on the fly, to serve a particular function, solve a specific problem, or deliver a particular solution to a customer.
- *Web services are distributed over the Internet*. Web services make use of existing, ubiquitous transport Internet protocols like HTTP. By relying on the same, well-understood transport mechanism as Web content, web services leverage existing infrastructure and can comply with current corporate firewall policies.
- Web services are described in terms of a description language that provides functional as well as non-functional characteristics. Functional characteristics include operational characteristics that define the overall behaviour of the service while non-functional characteristics include service availability, reliability, security, authorisation, authentication, performance characteristics, e.g., speed or accuracy, timeliness information as well as payment schemes on a "Finance it", "Lease it", "Pay for it" or "Pay per use" basis.

By employing a web service architecture, distributed applications which are dynamic and loosely coupled can be created based on different web services accessible for different purposes from any kind of internet ready devices such as personal computers, workstations, laptops, WAP-enabled cellular phones, personal digital assistants (PDAs), and with WI-FI or UMTS capabilities even household appliances fitted with computer chips.

The web services technologies and architecture also allow for a new kind of business logic to emerge: "global business logic", i.e. software components that can be used by thousands of consumers. If we take the example of "sales tax calculation", this operation is used typically on quotes, orders and invoices. One could imagine in the US alone, with 50 states, with all the possible combinations of geography, goods and ways of doing business (internet, mail order, retail stores, …), this component quickly becomes almost impossible to manage. If we now expand this problem to the world one can start asking why should every company in the world "own" a component capable of calculating sales tax. With the current bandwidth and computing power, regardless of the volumes of orders, it would be very easy to outfit existing applications with using a web services hosted and maintained by specialized companies that track all the possible changes in this type of calculation.

The same technologies can address one of the major challenges in e-business namely providing seamless connectivity between business processes and applications external to an enterprise and the enterprise's back office applications, such as billing, order processing, accounting, inventory, receivables, and services focused on total supply chain management and partnership including product development, fulfilment, and distribution. In addition, web services technologies address security concerns.

For security reasons the firewall, which is essential for the survival of a business site, prevents access to back office systems in order to maintain the integrity of business data stored in business databases and guarantee privacy. The approach with the firewall is that it disallows any kind of binary method invocation, except on predesignated guarded, i.e., secure, ports. As a result even if all the Web sites come equipped with the same component technology such as, for instance, CORBA, firewalls prevent calls from going through. Web services address the problems of firewall blocking and singular solutions simply. The only thing that is common to business Web sites and is firewall "friendly" is HTTP. Web services use the Simple Object Access Protocol (SOAP) as a transport protocol to transport the call from one Web site to the next, see section-8.1. SOAP combines the proven Web technology of the HTTP with the flexibility and extensibility of XML. It facilitates interoperability among a wide range of programs and platforms, making existing applications accessible to a broader range of users. Web services exploit this technology by allowing service providers to provide their clients with secure, personalised access to their back-office information and processes. This allows the service provider can monitor Web access using its Web server, control database permissions, monitor the database log files and retain control of its corporate business data and processes.

4 Web services: types and characteristics

Web services can be classified in accordance with three models on basis of the business functionality they provide and exhibit several important characteristics all of which are examined in this section.

4.1 Types of web services

Topologically, web services can come in two flavours. Informational, or type I, web services support only inbound operations. As such they always wait for a request, process it and respond. This type is very common and is generally stateless. Complex, or type II web services implement some form of coordination between inbound and outbound operations and are almost always statefull, see Figure 2.

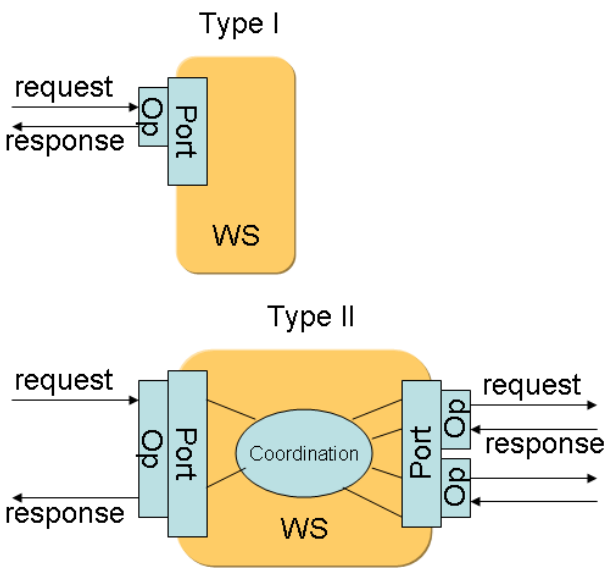


Figure 2 High-level view of informational and complex services.

1. Informational services

Informational services are services of relatively simple nature, they either involve simple request/response sequences between interacting services thus providing access to content (content services) or they expose back-end business applications to other applications located outside the firewall (business process services). Simple services are programmatic services as they encapsulate a programmatic process and expose the logic of the applications and components that underlie them, e.g., currency conversion. The

exposed programmatic services perform a request-response type of business task and return a concrete result, in this sense they can be viewed as "atomic" (or singular) operations. The clients of these services can assemble them to build new applications. Informational services can be further subdivided into:

- Pure *content services* give programmatic access to content such as weather report information, simple financial information, stock quote information, design information, news items and so on.
- More complicated forms of information services that can also provide a seamless aggregation of information across disparate systems and information sources, including back-end systems, giving programmatic access to a business service so that the requester can make the best decisions. Typical examples of such *simple trading services* include services such as reserving a rental car or submitting a purchase order.
- *Information syndication* services. Information syndication services are value-added information web services that purport to "plug into" commerce sites of various types, such as e-Marketplaces, or sell-sites. Generally, these services are offered by a third-party and run the whole range from commerce-enabling services, such as logistics, payment, fulfilment, and tracking services, to other value-added commerce services, such as rating services. Typical examples of syndicated services might include reservation services on a travel site or rate quote services on an insurance site.

Informational services (including information syndication services) are singular in that they perform a complete unit of work that leaves its underlying data stores in a consistent state. However, they are not transactional in nature (although their back-end realizations may be). Informational and simple trading services require support by the three evolving standards: (i) Service description (WSDL), (ii) Service Publication and Discovery (UDDI) and (iii) Communication Protocol (SOAP), all described in section-8. The key limitations of informational services (including simple trading services) are that they do not define any standards for business collaboration, process definition or security over the Web. Today, most of the software vendors who support web services provide either information syndication or simple trading functionality.

2. Complex services

For enterprises to obtain the full benefit of web services, transactional-like web service functionality is required. True business-to-business collaboration requires functionality that is well beyond that found in informational web services and involves choreographies of service invocations between businesses to complete a multi-step business interaction. Business-to-business collaboration relies on numerous document exchanges, multi-party, long running transactions (or "business conversations") that involve sophisticated security techniques, such as non-repudiation and digital signatures, as well as business process management. Business-to-business collaboration usually involves business agreement descriptions, which define roles such as buyer and seller and a purchasing protocol between them. The agreement definition outlines the requirements that each role must fulfil. For example, the seller must have web services that receive request for quote (RFQ) messages, purchase order (PO) messages and payment messages. The buyer role must have web services that receive (RFQ response messages), invoice messages and account summary messages. This choreography of web services into business roles is critical for establishing multi-step, service-oriented interactions between business partners and modelling business agreements.

By explicitly modelling business agreements and each business participant's ability to play roles in the business agreement, the requester can choose which business agreement to embark on with potential service providers. Consider for instance a secure supply-chain marketplace application where buyers and suppliers collaborate and compete for orders and the fulfilment of those orders. Numerous document exchanges will occur in this process including requests for quotes, returned quotes, purchase order requests, purchase order confirmations, delivery information and so on. Long running transactions and asynchronous messaging will occur, and business "conversation" and even negotiations may occur before the final agreements are reached. This type of functionality is exhibited by complex web services. Complex web services, just like informational services, require the support of standards such as SOAP, WSDL and UDDI, however, they also require emergent standards for the following:

- Business processes and associated XML messages and content;
- A registry for publishing and discovering business processes and collaboration protocol profiles;
- Collaboration partner agreements;

- Standard business terminology;
- A uniform message transport infrastructure.

Two key emerging business protocols that are widely accepted by industry on which complex web services can rely are ebXML (electronic-business XML) [Chappel01], [Grangard01] and RosettaNet [Masud03]. The complex web services standards are still evolving and are converging on SOAP, WSDL, UDDI and the web services Business Process Execution Language (BPEL) currently under standardisation at OASIS [Andrews03].

Web services can also be categorised according to the way they are programmed in applications. Some web services exhibit programmatic behaviour whereas others exhibit mainly interactive behaviour where input has to be supplied by the user. This makes it natural to distinguish between the following two types of web services:

1. *Programmatic web services:* Programmatic web services encapsulate a programmatic business processes and expose the business logic functionality of the applications and components that underlie them. Programmatic business services expose function calls, typically written in programming languages such as Java/EJB, Visual Basic or C++. Applications access these function calls by executing a web service through standard WSDL programmatic interface. The exposed programmatic services perform a request-response type of business task and return a concrete result, in this sense they can be viewed as "atomic" operations. The clients of these web services can assemble them to build new applications. An example typical of programmatic behaviour could be an inventory checking function of an inventory management system, which is exposed as a web service accessible to applications via the Internet. The `inventory checking` web service can then be invoked by a `create order` service that also uses a `create purchase order` web service from an order entry system to create orders, if the inventory is available.

2. *Interactive web services:* these expose the functionality of a Web application's presentation (browser) layer. They expose a multi-step web application behaviour that combines a web server, an application server and underlying database systems and typically deliver the application directly to a browser. Clients of these web services can incorporate interactive business processes into their web applications, presenting integrated (aggregated) applications from external service providers. As interactive web services may involve multiple web pages interacting with an end user, in many cases both the service provider and the service aggregator are unable to take full advantage of the potential synergies that result from combining their applications and integrating their business models. The problem being that the aggregator's application is often unaware of the page content that is delivered to end user, and is typically unable to alter this content or its behaviour.

Obviously the two types of web services, namely programmatic and interactive, can be combined delivering, thus, business processes that combine typical business logic functionality with web browser interactivity.

4.2 Service characteristics

Services exhibit the following characteristics, which we will describe in the following.

- Functional and non-functional properties
- State properties
- Granularity
- Complexity
- Synchronicity
- Extensibility

4.2.1 Functional and non-functional properties

Services are described in terms of a description language that provides functional as well as non-functional characteristics. Functional characteristics include operational characteristics that define the overall behaviour of the service while non-functional characteristics include non-functional service quality attributes, such as service metering and cost, performance metrics, e.g., response time or accuracy,

security attributes, authorisation, authentication, (transactional) integrity, reliability, scalability, and availability.

Functional properties of services are examined in sections 6.2 and 6.3 while non-functional are examined in section-12.

4.2.2 State properties

Services could be stateless or stateful. If services can be invoked repeatedly without having to maintain context or state they are called stateless, while services that may require their context to be preserved from one invocation to the next are called stateful. The services access protocol is always connectionless.

- *Stateless Web Service:* A web service in its simplest form, e.g., an informational weather report service, does not keep any memory of what happens to it between requests. Here, stateless means that each time a consumer interacts with a web service, an action is performed. After the results of the service invocation have been returned, the action is finished. There is no assumption that subsequent invocations are associated with prior ones. Consequently, all the information required to perform the service is either passed with the request message or can be retrieved from a data repository based on some information provided with the request.
- *Stateful Web Service:* In contrast to a stateless web service, a stateful web service maintains some state between different operation invocations issued by the same or different web service clients. If a particular "session" or "conversation" involves web services then transient information between operation invocations is stateful. A message sent to a web service stateful instance, would be interpreted in relation to that instance specific state, while the state of the instance would be the context for interpreting the message. Typically, business processes specify stateful interactions involving the exchange of messages between partners, where the state of a business process includes the messages that are exchanged as well as intermediate data used in business logic and in composing messages sent to partners. Consider for instance, a supply chain, where a seller's business process might offer a service that begins an interaction by accepting a purchase order through an input message, and then returns an acknowledgement to the buyer if the order can be fulfilled. It might later send further messages to the buyer, such as shipping notices and invoices. The seller's business process must "remember" the state of each such purchase order interaction separately from other similar interactions. This is necessary when a buyer has many purchase processes with the same seller that are executed simultaneously.

4.2.3 Complexity and granularity

Services can vary in function from simple requests (for example, currency conversion, credit checking and authorization, inventory status checking, or a weather report) to complex systems that access and combine information from multiple sources, such as an insurance brokering system, a travel planner, an insurance liability computation, or a package tracking system. Simple service requests may have complicated realizations. Consider for example, "pure" business services, such as logistic services, where automated services are the actual front-ends to fairly complex physical organisational business processes. Informational services are discrete in nature, exhibit normally a request/reply mode of operation and are of fine granularity, viz. atomic in nature.

Complex services are coarse-grained and involve interactions with other services and possibly end-users in a single or multiple sessions. Enterprises can use a single (discrete) service to accomplish a specific business task, such as billing or inventory control or they may compose several services together to create a distributed e-business application such as customised ordering, customer support, procurement, and logistical support. These services are collaborative in nature and some of them may require transactional functionality.

4.2.4 Synchronicity

We may distinguish between two programming styles for services: synchronous or remote procedure call (RPC)-style versus asynchronous or message (document)-style.

Synchronous services: Clients of synchronous services express their request as a method call with a set of arguments, which returns a response containing a return value. This implies that when a client sends a request message, it expects a response message before continuing with its computation. Because of this type of bilateral communication between the client and service, RPC-style services require a tightly coupled model of communication between the client and service provider. RPC-style web services are normally used when an application exhibits the following characteristics:

- The client invoking the service requires an immediate response.
- The client and service work in a back-and-forth conversational way.
- The service is process-oriented (part of a process to be precise) rather than data-oriented.
- Examples of typical simple information services with an RPC-style include returning the current price for a given stock; providing the current weather conditions in a particular location; or checking the credit rating of a potential trading partner prior to the completion of a business transaction.

Asynchronous services: Asynchronous services are document-style or message driven services. When a client invokes a message-style service, the client typically sends it an entire document, such as a purchase order, rather than a discrete set of parameters. The service accepts the entire document it processes it and may or may not return a result message. A client that invokes an asynchronous service does not need to wait for a response before it continues with the remainder of its application. The response from the service, if any, can appear hours or even days later. Asynchronous services promote a looser coupling between the client and service provider, as there is no requirement for a tightly coupled request-response model between the client and the web service. Document-style web services are normally used when an application exhibits the following characteristics:

- The client does not require (or expect) an immediate response.
- The service is process-oriented.

Examples of document-style web services include processing a purchase order; responding to a request for quote order from a customer; or responding to an order placement by a particular customer. In all these cases, the client sends an entire document, such as a purchase order, to the web service and assumes that the web service is processing it in some way, but the client does not require an immediate answer.

4.2.5 Service usage context

In addition to the types and characteristics of web services mentioned above it also useful to divide information services into different categories based on the web service requester's perspective. We base the following classification and discussion on a revision of the findings reported in [Pér00]. Here we may distinguish between the following service categories:

- *Commodity service:* this kind of service is provided by a large number of different providers. Replacing one service provider by another will not compromise system functionality and unavailability of the web service does not affect productivity. Each access to a service is preceded by discovery phase until an appropriate provider (in terms of price and service quality) is located. Using such a service does not require an data or process integration. An example of this kind of web service is a weather report as part of a travel application.

- *Replaceable service:* a replaceable web service is a service provided by several providers and replacing one provider with another does not affect application functionality. The productivity is not reduced severely if the service is unavailable for a short period of time. A discrete (enumerated) discovery process involving several alternative possibilities may be pursued here. An example of this kind of service is a car rental service. Here we may pursue different car rental agencies, e.g., Avis, Hertz, Budget, and choose the first service response that arrives and satisfies our needs. This type of service is usually well integrated with the consumer processes (e.g. rent-a-car activity) but not tightly integrated at the data level.

- *Mission-critical service:* a specific provider always provides the service, replacing this provider severely compromises the functionality of the application. If the service is unavailable for a period

of time it would drastically reduce the productivity of the application. This type of service would typically hold some critical enterprise data and be integrated at the process level.

5 Services, interfaces and components

One important aspect of services is that they distinguish between an interface and implementation part. The interface part defines the functionality visible to the external world and the means to access this functionality. The service describes its own interface characteristics, i.e., the operations available, the parameters, data-typing and the access protocols, in a way that other software modules can determine what it does, how to invoke its functionality, and what result to expect in return. In this regard, services are contractible software modules as they provide publicly available descriptions of the interface characteristics used to access the service so that potential clients can bind to it. The service client uses the service's interface description to bind to the service provider and invoke its functionality.

The implementation realizes the interface and the implementation details are hidden from the users of the service. Different service providers using any programming language of their choice may implement the same interface. One service implementation might provide the functionality itself directly, while another service implementation might use a combination of other services to provide the same functionality.

The eventual goal of service-oriented computing is to enable distributed applications that can be dynamically assembled according to changing business needs, and customised based on device and user access.

To better understand how to design and develop services, it is important to understand the relationship between services, interfaces and components. When designing an application developers develop a logical model of what an enterprise does in terms of business objects (such as product, customer, order, bill, etc) and the services the business requires from these business objects (what is the stock level, what is the delivery schedule and so on). The developer may implement these concepts as a blend of interface specifications in terms of services and component implementations (the business objects). Components are normally used to implement (realize) the service functionality. The services need to be designed and implemented in ways that make them re-useable in various contexts as defined by service consumer but often unknown to the service designer. This is very similar to "human services" where, for instance, we do not pay a different price for posting an envelope depending whether it contains a letter, an invoice, an order, a patent, a check, etc, while at the same time the post office is unaware of the content of the envelope.

From an enterprise point of view it is much more desirable to deal with service interfaces than with component implementations. Frequently, the interfaces that the components realize are too low level and not representative of, or even relevant to, the actual business services provided. This implies that we are dealing with two largely complementary elements: the *service interface* and its corresponding implementation component (*service realization*). It is important to distinguish between these two elements because in many cases the organisations that provide service interfaces are not the same as the organisations that implement the services. A service is a business concept that should be specified with an application or the user of the service in mind, while the service realization may be provided by a software package, e.g., an ERP package, a special purpose built component, a commercial off the shelf application (COTS), or a legacy application.

A service is usually a business function implemented in software, wrapped with a formal documented interface that is well known and known where to be found not only by agents who designed the service but also by agents who do not know about how the service has been designed and yet want to access and use it. Black box encapsulation inherits this feature from the principles of modularity in software engineering, e.g., modules, objects and components. Services are different from all of these forms of modularity in that they represent complete business functions, they are intend to be reused and combined in new transactions not at the level of an individual program or even application but at the level of the enterprise or even across enterprises. They are intended to represent meaningful business functionality that can be assembled into a larger and new configurations depending on the need of particular kinds of users particular client channels.

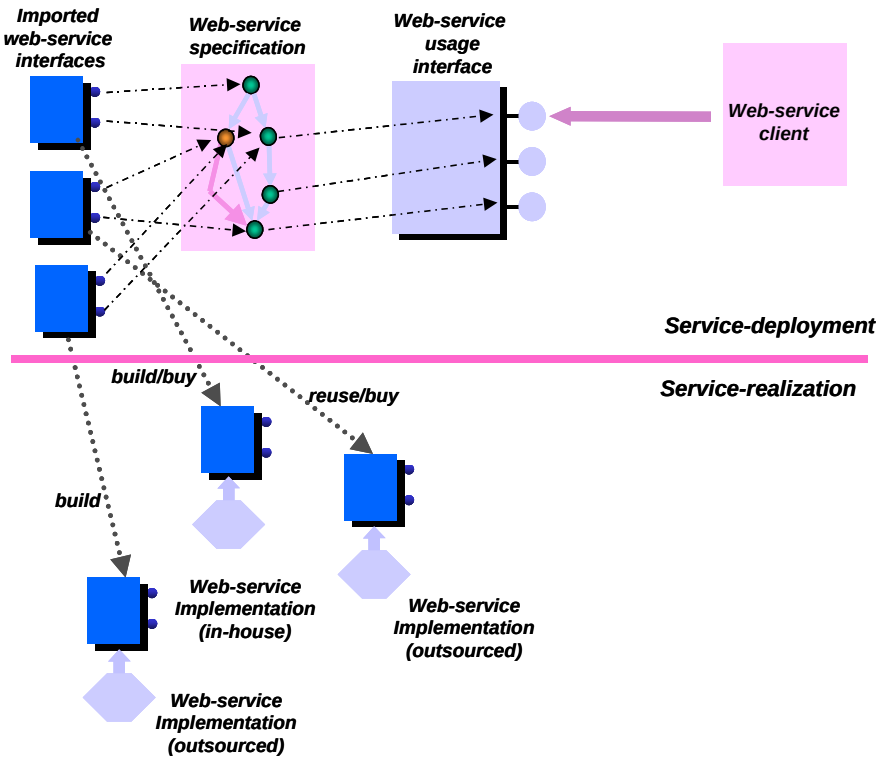


Figure 3 Services, interfaces and service realizations.

To a service client is irrelevant whether the services are provided by a fine-grained suite of components, or a single monolithic ERP. However, it is important that the developer who implements the service still thinks about granularity so they can change parts of the implementation with the minimum of disruption to other components, applications and services. The granularity of components should be the prime concern of the developer responsible for providing component implementations for services, whereas service designers should be more interested in the process operations and assembly potential of the provided services.

The only way one service can interact with another is via its interface. To cater for this requirement for service-based development we need to introduce the concept of *service specification* in addition to the concept of an interface. Recall that the purpose of the service interface is to define which interfaces the service offers to the outside world. The interface simply provides the mechanism by which services communicate with applications and other services. Technically, the service interface is the description of the signatures of a set of operations that are available to the service client for invocation. The service specification must explicitly describe all the interfaces that a client of this service expects as well as the service interfaces that must be provided by the environment into which the service is assembled/composed. As service interfaces of composed services are provided by other (possibly singular) services, the service specification serves as a means to define how a composite service interface can be related to the interfaces of the imported services and how it can be implemented out of imported service interfaces. This is shown in Figure 3. In this sense the service specification has a mission identical to a composition meta-model that provides a description of how the web service interfaces interact with each other and how to define a new web service interface (<PortType>, see section-8.2) as a collection (assembly) of existing ones (imported <PortType>s), see Figure 3. A service specification, thus, defines the encapsulation boundary of a service, and consequently determines the granularity of replaceability of web service interface compositions. This is the only way to design services reliably using imported services without knowledge of their implementations. As service development requires that we deal with multiple imported service interfaces it is useful to introduce this stage the concept of *service usage interface*. A service usage interface is simply the interface that the service exposes to its clients. This means that the service usage interface is not different from the imported service interfaces in Figure 3, it is, however, the only interface viewed by a client application.

Figure 3 distinguishes between two broad aspects of services: service deployment, which we examined already, versus service realization. The service realization strategy involves choosing from an increasing diversity of different options for services, which may be mixed in various combinations including [Papa02]:

- In house service design and implementation. Once a service is specified, the design of its interfaces or sets of interfaces and the coding of its actual implementation happens in-house.
- Purchasing/leasing/paying for services. Complex web services that are used to develop trading applications are commercialisable software commodities that may be acquired from a service provider, rather than implemented internally. These types of services are very different from the selling of shrink-wrapped software components, in that payment should be on an execution basis for the delivery of the service, rather than on a one-off payment for an implementation of the software. For complex trading web services, the service provider may have different charging policies such as payment per usage, payment on a subscription basis, lifetime services and so on.
- Outsourcing service design and implementation. Once a service is specified, the design of its interfaces or sets of interfaces and the coding of its actual implementation may be outsourced. Software outsourcings are advantageous in the case of organisations that have become frustrated with the shortcomings of their internal IT departments.
- Using wrappers and/or adapters. Non-component implementations for services may include database functionality or legacy software accessed by means of *adapters* or *wrappers*. Wrappers reuse legacy code by converting the legacy functionality and encapsulating it inside components. Adapters use legacy code in combination with newly developed component code. This newly developed that may contain new business logic and rules that supplement the converted legacy functionality.

6 The service-oriented architecture

Web services hold the promise of moving beyond the simple exchange of information – the dominating mechanism for application integration today – to the concept of accessing, programming and integrating application services that are encapsulated within old and new applications. This would mean organisations will be able not only to move information from application to application, but also to create complex customisable composite applications, leveraging any number of back-end and older technology systems found in local or remote applications.

Key to this concept is the service-oriented architecture (SOA). SOA is a logical way of designing a software system to provide services to either end-user applications or to other services distributed in a network, via published and discoverable interfaces. To achieve this SOA reorganises a portfolio of previously siloed software applications and support infrastructure into an interconnected set of services, each accessible through standard interfaces and messaging protocols. Once all the elements of enterprise architecture are in place, existing and future applications can access these services as necessary without the need of convoluted point-to-point solutions based on inscrutable proprietary protocols.

The term service-oriented architecture signifies the way web services are described and organised so that dynamic, automated discovery and use of network-available services can take place. This architectural approach is particularly applicable when multiple applications running on varied technologies and platforms need to communicate with each other. In this way, enterprises can mix and match services to perform business transactions with minimal programming effort.

SOA is a logical way of designing a software system to provide services to either end-user applications or other services distributed in a network through published and discoverable interfaces. The basic SOA defines an interaction between software agents as an exchange of messages between service requesters (clients) and service providers. Clients are software agents that request the execution of a service. Providers are software agents that provide the service. Agents can be simultaneously both service clients and providers. Providers are responsible for publishing a description of the service(s) they provide. Clients must able to find the description(s) of the services they require and must be able to bind to them.

The service-oriented architecture builds on today's web services baseline specifications of SOAP, WSDL, and UDDI that are going to be examined in section-8. The main building blocks of the web services architecture

are three-fold and they are determined on the basis of three primary roles that can be undertaken by these architectural modules. These are the service provider, the service registry and the service requester.

6.1 Roles of interaction in the service-oriented architecture

Web service provider

The first important role that can be discerned in the web service architecture is that of the web service provider. The web service provider is the organization that owns the web service and implements the business logic that underlies the service. From an architectural perspective this is the platform that hosts and controls access to the service.

The web service provider is responsible for publishing the web services it provides in a service registry hosted by a service broker. This involves describing the business, service and technical information of the web service and registering that information with the web service registry in the format prescribed by the discovery agency.

Web service requester (client)

The next major role in the web service architecture is that of the web service requester (or client). From a business perspective this is the enterprise that requires certain functions to be satisfied. From an architectural perspective, this is the application that is looking for, and subsequently invoking, the service. The web service requester searches the service registry for the desired web services. This effectively means discovering the web service description in a registry provided by a discovery agency and using the information in the description to bind to the service. Two different kinds of web service requesters exist. The requester role can be played either by a browser driven by an end user or by another web service as part of an application without a user interface.

Service Registry

The last important role that can be distinguished in the web services architecture is that of the web-registry which is a searchable directory where service descriptions can be published and searched. Service requestors find service descriptions in the registry and obtain binding information for services. This information is sufficient for the service requester to contact, or bind to, the service provider and thus make use of the services it provides.

It is unreasonable to assume that there would be a single global registry containing all of the information required to find and interact with businesses throughout the world. What we will see are local communities of service providers and requesters organised in vertical markets and gathering around portals. These marketplaces will consist of UDDI registries containing business data for that specific vertical market. This gives raise to the idea of a web service discovery agency that is the organisation (acting as a third trusted party) whose primary activities focus on hosting the registry, publishing and promoting web services. The service discovery agency can further improve the searching functionality for web service requesters by adding advertising capabilities to this infrastructure and by supporting facilities for web service matchmaking between providers and requesters.

The web service discovery agency is responsible for providing the infrastructure required to enable the three operations in the web service architecture as described in the previous section: publishing the web services by the web service provider, searching for web services by web service requesters and invoking the web services.

6.2 Operations in the service-oriented architecture

For an application to take advantage of the web service interactions between the three roles in the SOA three primary operations must take place. These are publication of the service descriptions, finding the service descriptions and binding or invocation of services based on their service description. These three basic operations can occur singly or iteratively.

6.2.1 Publish

Publishing a web service so that other users can find it actually consists of two equally important operations. The first operation is describing the web service itself; the other is the actual registration of the web service.

If a service provider wishes to publish its web services with the registry then the first requirement is to properly describe these web services in WSDL. To achieve this goal it is necessary to decide what information is relevant, not only from the viewpoint of the web service provider but also from the viewpoint of the web discovery agency and the web service requester. We may distinguish between three basic categories of information necessary for proper description:

- business information; information on the web service provider, the owner of the web service;
- service information; information about the nature of the web service; and
- technical information; information about the invocation methods of the web service.

The next step in publishing a web service is registration. Registration deals with storing the web service descriptions in the web services registry provided by the discovery agency. For web service requesters to be able to find a web service this needs to be published with at least one discovery agency.

6.2.2 Find

In a similar fashion to publishing, finding web services is also a twofold operation. Finding the desired web services consists of first discovering the services in the registry of the discovery agency and then selecting the desired web service(s) from the search results.

Discovering web services involves querying the registry of the discovery agency for web services matching the needs of a web service requester. A query consists of search criteria such as type of service, preferred price range, what products are associated with this service, with which categories in company and product taxonomies is this web service associated as well as other technical characteristics (see section-8.3.1) and is executed against the web service information in the registry entered by the web service provider. The find operation can be involved in two different instances by the requester. Statically at design time to retrieve a service's interface description for program development. Dynamically (at run-time) to retrieve a service's binding and location description for invocation.

Selection deals with deciding about which web service to invoke from the set of web services the discovery process returned. Two possible methods of selection exist: *manual* and *automatic* selection. Manual selection implies that the web service requester selects the desired web service directly from the returned set of web services after manual inspection. The other possibility is automatic selection of the best candidate between potentially matching web services. A special client application program provided by the web service registry (broker) can achieve this. In this case the web service requester has to specify preferences to enable the application to infer which web service the web service requester is most likely to wish to invoke.

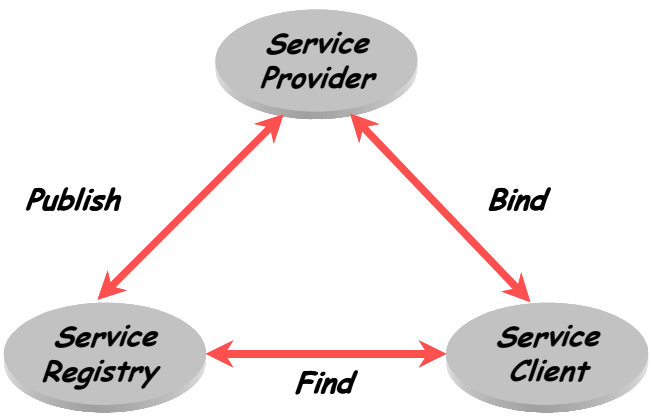


Figure 4 Web service roles and operations.

6.2.3 Bind

The final operation in the web service architecture and perhaps the most important one is the actual invocation of the web service. During the binding operation the service requester invokes or initiates an interaction at run-time using the binding details in the service description to locate and contract to the service. The technical information entered in the registry by the web service provider is used here. Two different possibilities exist for this invocation. The first possibility is direct invocation of the web service by the web service requester using the technical information included in the description of the service. The second possibility is mediation by the discovery agency when invoking the web service. In this case all communication between the web service requester and the web service provider goes through the web service registry of the broker.

A logical view of the service-oriented architecture is given in Figure 4. This figure illustrates the relationship between the three roles and the three operations mentioned in the previous. First, the web service provider publishes its web service(s) with the discovery agency. Next, the web service requester searches for desired web services using the registry of the discovery agency. Finally, the web service requester, using the information obtained from the discovery agency, invokes (binds to) the web service provided by the web service provider [Boubez00].

The three operations discussed above are central to realizing an effective service oriented architecture. They will be revisited and briefly discussed in the next sections where we shall examine the different standards used in conjunction with web services.

6.3 Aggregated services

Moving up in the complexity scale, let us consider the example of a travel planning service. This application employs an aggregate service that makes use of other services. The SOA representation of this type of aggregate service is illustrated in Figure 5. This figure illustrates that the requester may be a corporate portal site that supports business-travel reservations for the employees of a particular company. This figure involves a hierarchical service provision scheme whereby a requester (client) sends a request to the aggregator, a system that offers a web service for complex travel reservations (step-1). The aggregator who is just another service provider receives the initial request and decomposes it into two parts one involving an airline reservation service for the air-travel portion of the trip and a hotel booking service request. The portal subsequently acts as a web service requester and forwards these two requests to airline (step-2) and hotel (step-4) service providers. These check the availability of airline seats and hotel rooms and reply to the aggregator (steps 3 and 5). Finally, reverts to its role as service provider and relays the ultimate response to the initial client (portal).

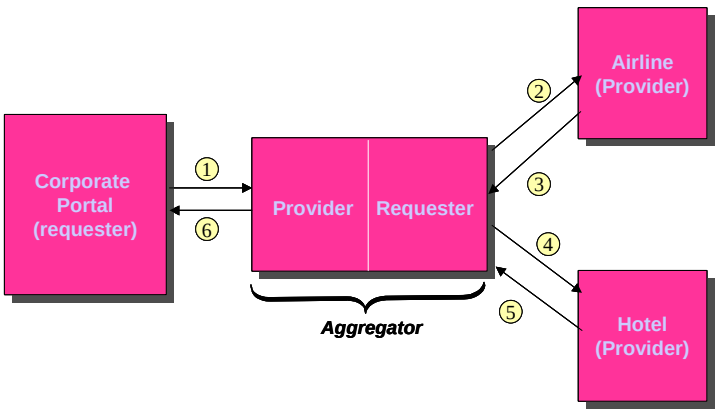


Figure 5 Aggregated SOA example.

7 The web services technology stack

The goal of the web service technology is to allow applications to work together over standard Internet protocols, without direct human intervention. By doing so, we can automate many business operations,

creating new functional efficiencies and new more effective ways of doing business. The minimum infrastructure required by the web services paradigm is purposefully low to help ensure that web services can be implemented on and accessed from any platform using any technology and programming language. By intent, web services are not implemented in a monolithic manner, but rather represent a collection of several related technologies. The more generally accepted definition for web services leans on a stack of specific, complementary standards, see Figure 6.

The core layers that define basic web services communication have been widely accepted and are implemented quite uniformly. Higher-level layers that define strategic aspects of business processes still remain an open problem and it is quite likely that different vendors will propose divergent approaches. The development of open and accepted standards is a key strength of the coalitions that have been building web services infrastructure. At the same time, these efforts have resulted in the proliferation of a dizzying number of emerging standards and acronyms. We provide high-level descriptions of the most important ones below.

Core layers

Although not specifically tied to any specific transport protocol, web services build on ubiquitous Internet connectivity and infrastructure to ensure nearly universal reach and support. In particular, web services will take advantage of HTTP, the same connection protocol used by Web servers and browsers.

- *Extensible Markup Language (XML):* XML is a widely accepted format for exchanging data and its corresponding semantics. It is a fundamental building block for nearly every other layer in the web services stack.
- *Simple Object Access Protocol (SOAP):* SOAP is a simple XML-based messaging protocol on which web services rely to exchange information among themselves. It is based on XML and uses common Internet transport protocols like HTTP to carry its data. SOAP implements a request-response model for communication between interacting web services. It consists of three parts: an envelope (which describes what is in the message and how to process it); a set of coding rules, and a convention for representing RPCs and responses. SOAP uses HTTP to penetrate firewalls, which are usually configured to accept HTTP and FTP service requests. It relies on XML to define the format of the information and then adds the necessary HTTP headers to send it.

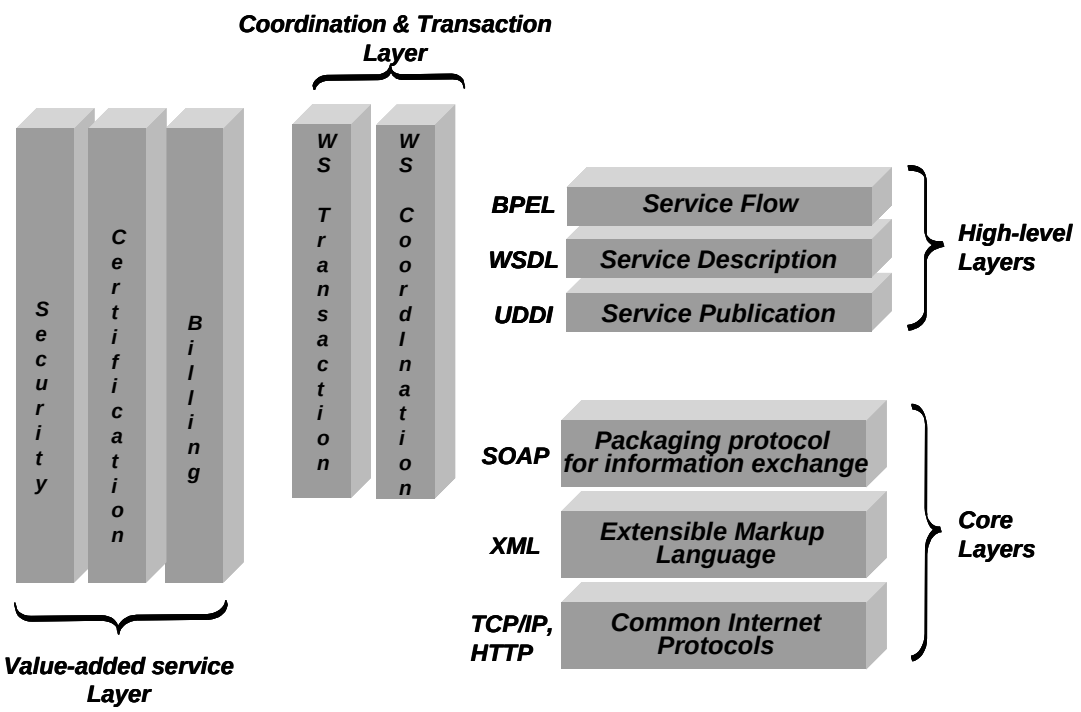


Figure 6 The web services technology stack.

Higher-Level layers

The key to web service interoperability is reliance solely on the following standards that are found in the higher levels of the web services technology stack.

1. *Service Description layer:* Web services become easy to use when a web service and its client rely on standard ways to specify data and operations, to represent web service contracts, and to understand the capabilities that a web service provides. To achieve this, the functionality of web services is first described by means of a Web services Description Language (WSDL) and subsequently is published in a Universal Description and Discovery and Integration (UDDI) service repository for discovery. WSDL defines the XML grammar for describing services as collections of communicating end-points capable of exchanging messages. Companies can publish WSDL specifications for services they provide and other enterprises can access those services using the description in WSDL. In this way, independent applications can advertise the presence of business processes or tasks that can be utilised by other remote applications and systems. Links to WSDL specifications are usually offered in an enterprise's profile in the UDDI registry.
2. *Service Publication layer:* Web service publication is achieved by UDDI, which is a public directory that provides publication of on-line services and facilitates eventual discovery of web services. It stores and publishes the WSDL specifications of available web services. Searches can be performed on the basis of company name, specific service, or types of service. This allows enterprises providing or needing web services to discover each other, define how they interact over the Internet, and share such information in a truly global and standardized manner in order to create value added applications.
3. *Service Flow layer:* describes the execution logic of web services based applications by defining their control flows (such as conditional, sequential, parallel and exceptional execution) and prescribing the rules for consistently managing their unobservable business data. In this way enterprises can describe complex processes that include multiple organisations— such as order processing, lead management, and claims handling—and execute the same business processes in systems from other vendors. This layer is representative of a family of XML-based process definition languages intended for expressing abstract and executable processes that address all aspects of enterprise business processes, including in particular those areas important for web-based services. Such languages include the Business Process Modelling Language (BPML) [Arkin01], the XML Process Definition Language [Wfmc02], and the Business Process Execution Language for Web Services (BPEL4WS) [Andrews03]. BPML is a block-structured programming language, which provides an abstract model and XML syntax for expressing business processes and supporting entities. Flow control (routing) is handled entirely by block structure concepts, e.g. execute all the activities in the block sequentially). BPML itself does not define any application semantics such as particular processes or application of processes in a specific domain; rather it defines an abstract model and grammar for expressing generic processes. XPDL is conceived of as a graph-structured language to handle blocks where process definitions cannot be nested. The activities in a process can be thought of as the nodes of a directed graph, with the transitions being the edges. Conditions associated with the transitions determine at execution time which activity or activities should be executed next. BPEL is a block-structured workflow-like language that describes business processes that can orchestrate web services. BPEL allows recursive blocks but restricts their definitions and declarations to the top level. This XML-based flow language defines how business processes interact. BPEL combines the former IBM WSFL and Microsoft XLANG efforts.
4. *Service Collaboration layer:* describes cross-enterprise collaborations of web service participants by defining their common observable behaviour, where synchronized information exchanges occur through their shared contact-points, when commonly defined ordering rules are satisfied. This layer is materialized by Web Services Choreography Description Language (WS-CDL) [Kavantzas04], which specifies the common observable behaviour of all participants engaged in business collaboration. Each participant could be implemented by completely different languages such as web services applications, whose implementation is based on executable business process languages like BPEL, XPDL and BPML. As WS-CDL does not depend on a specific business process execution language it can be used to specify truly interoperable collaborations between any type of web service participant regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Coordination/Transaction layer

Solving the problems associated with service discovery and service description retrieval is the key to success of web services. Currently there are attempts underway towards defining transactional interaction among web services. The WS-Coordination and WS-Transaction initiatives complement BPEL4WS to provide mechanisms for defining specific standard protocols for use by transaction processing systems, workflow systems, or other applications that wish to coordinate multiple web services. These three specifications work in tandem to address the business workflow issues implicated in connecting and executing a number of web services that may run on disparate platforms across organisations involved in e-business scenarios.

Value-added services layer

Additional elements that support complex business interactions must still be implemented before web services can automate truly critical business processes. These are defined in the value-added services layer, see Figure 6. Mechanisms for security and authentication, contract management, quality of service, and more will soon follow, some as standards, others as value-added solutions from independent software vendors.

In the following we concentrate mainly on describing the core and higher-level layers of the web services stack.

8 Web service standards

For there to be wide spread acceptance of web services there needs to be a set of clear, widely adopted standards Fortunately, there is wide agreement across the industry on a set of standards to support web services. Four sets of services have emerged as the basis for standards in web services.

8.1 SOAP: Simple Object Access Protocol

In the emerging world of web services, it will be possible for enterprises to leverage mainstream application development tools and Internet application servers to bring about inter-applications communication, which has been historically associated with EDI technology. This will enable enterprises to conduct business electronically by making a broader range of services available faster and cheaper. For this paradigm to become a reality we need to overcome proprietary systems running on heterogeneous infrastructures. However, the tools and common conventions required to interconnect these systems were lacking until recently.

The conventional approach is to use distributed communication technologies such as CORBA/IIOP, DCOM, Java/RMI, or any other application-to-application communication protocols for server-to-server communications. However, both DCOM and CORBA/IIOP have severe weaknesses for client-to-server communications, especially when the client machines are scattered across the Internet. These distributed communications protocols have a *symmetrical* requirement. Both ends of the communication link would need to be implemented under the same distributed object model and would require the deployment of libraries developed in common. Using Java/RMI also requires that the server application to be written in Java and, for all practical purposes, the client application as well. Moreover, both DCOM and CORBA/IIOP rely on single-vendor solutions to use the protocol to maximum advantage. Though both protocols have been implemented on a variety of platforms and products, the reality is that a given deployment needs to use a *single-vendor's implementation*. In the case of DCOM, this means every machine runs Windows NT. In the case of CORBA, this means that every machine runs the same ORB product. It is possible to get two CORBA products to call one another using IIOP. However, many of the higher-level services (such as security and transactions) are not generally interoperable at this time. Additionally, any vendor-specific optimisations for same-machine communications are very unlikely to work unless all applications are built against the same ORB product (symmetricity). An even more limiting issue is the *difficulty of getting these protocols to work over firewalls or proxy servers.*

To address the problem of overcoming proprietary systems running on heterogeneous infrastructures, web services rely on SOAP, an XML-based communication protocol for exchanging information between computers regardless of their operating systems, programming environment, or object model framework. SOAP is defined as lightweight protocol for exchange of structured and typed information between computers and systems in decentralised and distributed environment such as the Internet or even a LAN (Local Area Network) [Cauldwell01].

The goal of SOAP is to diffuse the barriers of heterogeneity that separate distributed computing platforms. SOAP achieves this by following the same recipe as other successful Web protocols: simplicity, flexibility, firewall friendliness, platform neutrality and XML messaging-based (text-based). SOAP is simply an attempt to codify the usage of existing Internet technologies to standardise distributed communications over the Web, rather than being a new technological advancement. SOAP provides a *wire protocol* that specifies how service-related messages are structured when exchanged across the Internet. SOAP is a lightweight protocol that allows applications to pass messages and data back and forth between disparate systems in a distributed environment enabling remote method invocation. By lightweight we mean that the SOAP protocol possesses only two fundamental properties:

1. sending and receiving HTTP (or other) transport protocol packets, and
2. processing XML messages.

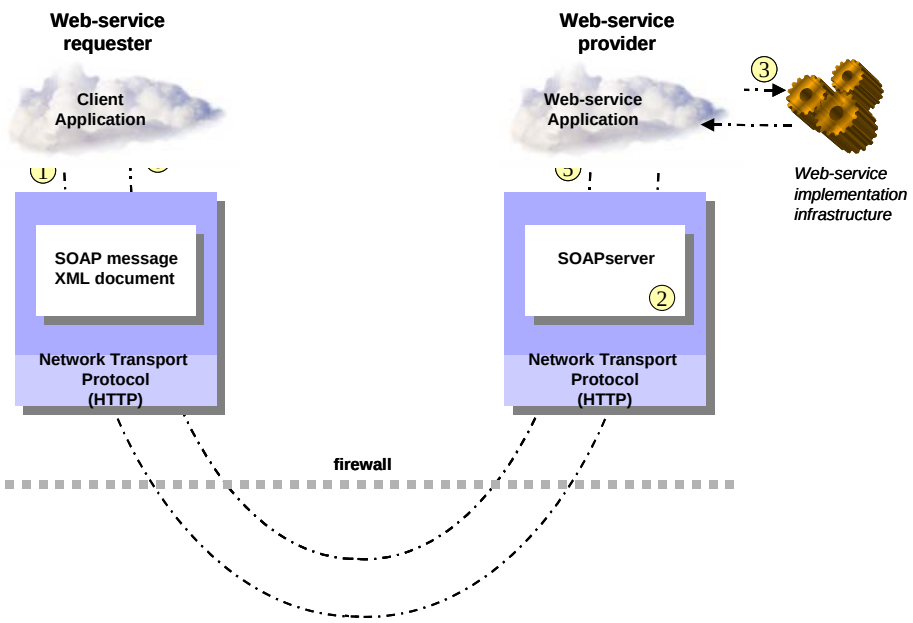


Figure 7 Distributed messaging using SOAP.

Although SOAP may use different protocols such as HTTP, FTP or RMI to transport messages and locate the remote system and initiate communications, its natural transport protocol is HTTP. Layering SOAP over HTTP means that a SOAP message is sent as part of an HTTP request or response, which makes it easy to communicate over any network that permits HTTP traffic. SOAP uses the HTTP protocol to *transport* XML-encoded serialised method argument data from system to system. *Serialisation* is the process of converting application data to XML. XML is then a serialised representation of the application data. The process of generating application data from XML is called *deserialisation*. SOAP's serialisation mechanism converts method calls to a form suitable for transportation over the network, using special XML tags and semantics. The serialised argument data is used on the remote end to execute the client's method call on that system's, rather than on the client's local system. Because SOAP can reside on HTTP, its request/response method operates in a very similar way to HTTP. When a client makes an HTTP request, the server attempts to service the request and can respond in one of two ways. It can either respond that communication was successful by returning the requested information. Alternatively, it can respond with a fault message notifying the client of the particular reason why the request could not be serviced.

Even though SOAP is an "object" access protocol, it does not mandate any object-orientated approach like CORBA does. SOAP rather defines a model for using simple request and response messages written in XML as the basic protocol for electronic communication. SOAP plays the role of a binding mechanism between two conversing endpoints. A SOAP endpoint is simply an HTTP-based URL that identifies a target for a method invocation.

The basic requirement for an Internet node to play the role of requester or provider in XML massaging-based distributed computing is the ability to construct and parse a SOAP message and the ability to communicate over the network by sending and receiving messages [Kreger01]. Any SOAP runtime system executing in a Web application server performs these functions. Distributed application processing with SOAP can be achieved in terms of the basic steps illustrated in Figure 7 and outlined in the following.

A service requester's application creates a SOAP message as a result of a request to invoke a desired web service operation hosted by a remote service provider (1). The request is formed by the SOAP client, which is a program that creates an XML document containing the information needed to invoke remotely a method in a distributed system. The XML code in body of the SOAP request is the place where the method request and its arguments are placed. The service requester forwards the SOAP message together with the provider's URI (typically over HTTP) to the network infrastructure.

The network infrastructure delivers the message to the message provider's SOAP runtime system (for example a SOAP server) (2). The SOAP server is simply special code that listens for SOAP messages and acts as a distributor and interpreter of SOAP documents. The SOAP server routes the request message to the service provider's web service implementation code (3). The SOAP server ensures that documents received over an HTTP SOAP connection are converted from XML to programming language-specific objects required by the application implementing the web services at the provider's site. This conversion is governed by the encoding scheme found within the SOAP message envelope. In doing so the SOAP server also ensures that the parameters included in the SOAP document are passed to the appropriate methods in the web service implementation infrastructure.

The web service is responsible for processing the request and formulating a response as a SOAP message. The response SOAP message is presented to the SOAP runtime system at the provider's site with the service requester's URI as its destination (4). The SOAP server forwards the SOAP message response to the service requester over the network (5).

The response message is received by the network infrastructure on the service requester's node. The message is routed through the SOAP infrastructure, potentially converting the XML response into objects understood by the source (service requester's) application (6).

8.1.1 Structure of a SOAP message

The current SOAP specification 1.1 describes how the data types defined in associated XML schemas are serialised over HTTP or other transport protocols. Both the provider and requester of SOAP messages must have access to the same XML schemas in order to exchange information correctly. The schemas are normally posted on the Internet, and may be downloaded by any party in an exchange of messages. A SOAP message contains a payload, the application specific information it delivers. Every SOAP message is essentially an XML document. SOAP messages can be broken down to three basic parts:

SOAP envelope: The purpose of SOAP is to provide a uniform container for XML messages. Prior to SOAP and its predecessor XML-RPC, the quality of XML being a universal document interchange standard were somewhat undermined by the fact that there was no uniform way to transport messages between two endpoints. The SOAP envelope serves to wrap any XML document interchange and provide a mechanism to augment the payload with additional information that is required to route it to its ultimate destination. The SOAP envelope is the single root of every SOAP message and must present for the message to be SOAP compliant. The <envelope> defines a framework for describing what is in a message and how to process it. All elements of the SOAP envelope are defined by a W3C XML Schema (XSD). The URI where this schema is located is also the identifier for the SOAP envelope namespace: "http://schema.xmlsoap.org/soap/envelope".
SOAP header: The header contains processing or control information, such as for example, information about where the document shall be sent, where it originated and may even carry digital signatures. This

type of information must be separated from the SOAP body. XML documents are typically application-specific and at the same time transport agnostic. This means that an XML document can express meaningful information about a service, such as authentication or transaction control-related information, as well as quality of service, billing or accounting information regarding an application and at the same time does not care about how it arrived at that particular application in the first place or how it can move from one application to the other.
SOAP body: The SOAP body is the area of the SOAP message where the application specific XML data (payload) being exchanged in the message is placed. The <body> element must be present and must be an immediate child of the envelope. It may contain an arbitrary number of child elements, called body entries, but it may also be empty. All body entries that are immediate children of the <body> element must be namespace qualified. By default, the body content may be an arbitrary XML and is not subject to any special encoding rules. The body must be contained within the envelope, and must follow any headers that might be defined for the message. The SOAP <body> is where the method call in formation and its related arguments are encoded. It is where the response to a method call is placed, and where error information can be stored.

8.1.2 The SOAP communication model

The web services communication model describes how to invoke web services and relies on SOAP. SOAP supports two possible communication styles: *remote procedure call (RPC)* and *document* (or *message*).

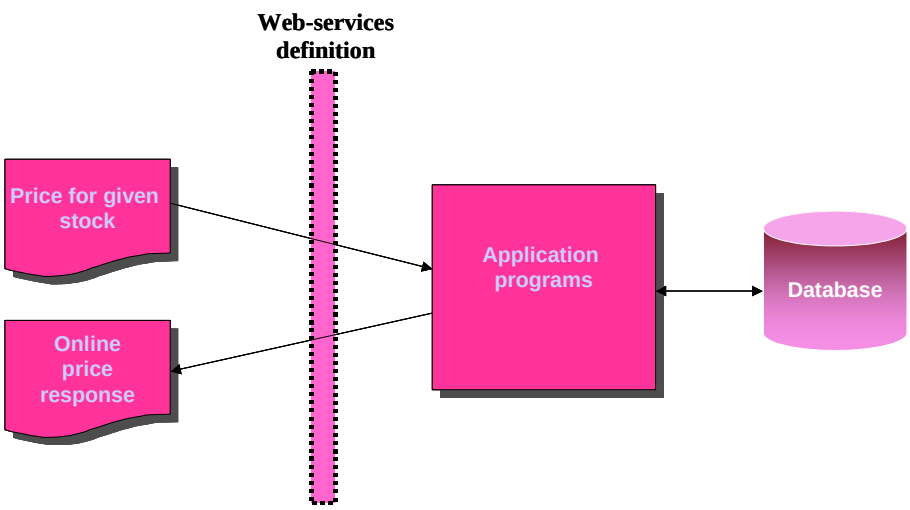


Figure 8 RPC-style web service for calculating the price of a given stock.

PRC-style web services

A remote procedure call (RPC)-style web service appears as a remote object to the client application. The interaction between a client and an RPC-style web service centres around a service-specific interface. Clients express their request as a method call with a set of arguments, which returns a response containing a return value. RPC style supports automatic serialisation/deserialisation of messages, permitting developers to express a request as a method call with a set of parameters, which returns a response containing a return value. Because of this type of bilateral communication between the client and web service, RPC-style web services require a tightly coupled model of communication between the client and service provider. Moreover, this communication is synchronous, meaning that when a client sends a request message, it expects a response message before continuing with the remainder of its application. This style resembles traditional distributed object paradigms, such as RMI, CORBA or DCOM.

RPC-style web services are normally used when an application exhibits the following characteristics [BEA01]:

1. The client invoking the web service needs an immediate response.
2. The client and web service work in a back-and-forth conversational way.
3. The web service is process-oriented rather than data-oriented.

Examples of typical simple information services with an RPC-style include returning the current price for a given stock; providing the current weather conditions in a particular location; or checking the credit rating of a potential trading partner prior to the completion of a business transaction.

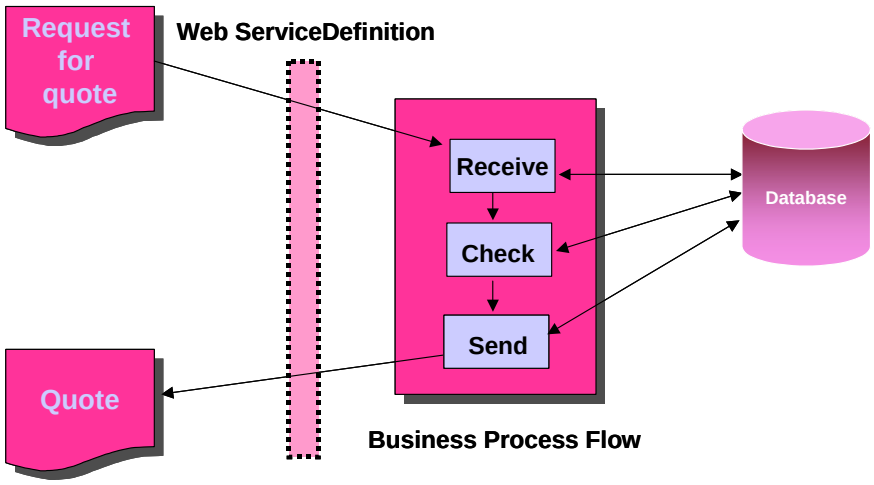


Figure 9 Processing a request for a quote.

Document (Message)-style web services

Document-style web services are message driven. When a client invokes a message-style web service, the client typically sends it an entire document, such as a purchase order, rather than a discrete set of parameters. The web service is sent an entire document, which it processes, however, it may or may not return a response message. This style is thus *asynchronous* in that the client invoking the web service can continue with its computation without waiting for a response. A client that invokes a web service does not need to wait for a response before it continues with the remainder of its application. The response from the web service, if any, can appear hours or even days later. The document style does not support automatic serialisation/deserialisation of messages. Rather it assumes that the contents of the SOAP message are well-formed XML documents, e.g., a purchase order.

Document-style web services promote a looser coupling between the client and service provider, as there is no requirement for a tightly coupled request-response model between the client and the web service. Document-style web services are normally used when an application exhibits the following characteristics [BEA01]:

1. The client does not require (or expect) an immediate response.
2. The web service is data-oriented rather than process-oriented.

Examples of document-style web services include processing a purchase order; responding to a request for quote order from a customer; or responding to an order placement by a particular customer. In all cases, the client sends an entire document, such as a purchase order, to the web service and assumes that the web service is processing it in some way, but the client does not require an immediate answer.

While it is important to understand the SOAP foundation for services, most web service developers will not have to deal with this infrastructure directly. Most web services use optimised SOAP bindings generated from WSDL. Thanks to WSDL, SOAP implementations can self-configure exchanges between web services while masking most of the technical details.

8.2 WSDL: Web Services Description Language

A SOAP service would require some documentation explaining the operations exposed along with their parameters in a machine-understandable standard format. In the Web services context, the analogous file

is a Web Services Description Language (WSDL) document. WSDL is the service representation language used to describe the details of the complete interfaces exposed by web services and thus is the means to accessing a web service. It is through this service description that the service provider can communicate all the specifications for invoking a particular web service to the service requester. For instance, neither the service requester nor the provider should be aware of each other's technical infrastructure, programming language or distributed object framework (if any).

The service description is a key to making the service oriented architecture loosely coupled and reducing the amount of required common understanding and custom programming and integration between the service provider and the service requester's applications. It is the header file for a web service. It is a machine understandable standard describing the operations of a web service. It also specifies the wire format and transport protocol that the web service uses to expose this functionality. It can also describe the payload data using a type system. The service description combined with the underlying SOAP infrastructure sufficiently isolates all technical details, e.g., machine- and implementation language-specific elements, away from the service requester's application and the service provider's web service. In particular, it does not mandate any specific implementation on the service requester side provided that the contract specified in a WSDL definition is abided by.

The Web services Description Language (WSDL) provides a mechanism by which service providers can describe the basic format of web requests over different protocols (e.g., SOAP) or encoding (e.g., Multipurpose Internet Messaging Extensions or MIME). WSDL is an XML based specification schema for describing the public interface of a web service. This public interface can include operational information relating to a web service such as all publicly available operations, the XML message protocols supported by the web service, data type information for messages, binding information about the specific transport protocol to be used, and address information for locating the web service. WSDL allows the specification of services in terms of XML documents for transmission under SOAP. We can think of web services as software modules that are accessed via SOAP.

A WSDL document describes how to invoke a service and provides information on the data being exchanged, the sequence of messages for an operation, protocol bindings, and the location of the service. WSDL represents a contract between the service requester and the service provider, in much the same way that an interface in an object-oriented programming language, e.g., Java, represents a contract between client code and the actual object itself. The prime difference is that WSDL is platform and language-independent and is used primarily (but not exclusively) to describe SOAP-enabled services. Essentially, WSDL is used to describe precisely ***what*** a service does, i.e., the operations the service provides, ***where*** it resides, i.e., details of the protocol-specific address, e.g., a URL, and ***how*** to invoke it, i.e., details of the data formats and protocols necessary to access the service's operations.

The WSDL specification can be conveniently divided into two parts: the service interface definition (abstract interface) and the service implementation (concrete endpoint) [Kreger01]. This enables each part to be defined separately and independently, and reused by other parts.

- The *service-interface definition* describes the general web service interface structure. This contains all the operations supported by the service, the operation parameters and abstract data types.
- The *service implementation part* binds the abstract interface to a concrete network address, to a specific protocol and to concrete data structures. A web service client may bind to such an implementation and invoke the service in question.

The service interface definition together with the service implementation definition makes up a complete WSDL specification of the service. The combination of these two parts contains sufficient information to describe to the service requester how to invoke and interact with the web service at a provider's site. Using WSDL, a requester can locate a web service and invoke any of the publicly available operations. With WSDL-aware tools, e.g., IBM's Web services Invocation Framework (WSIF), this process can be entirely automated, enabling applications to easily integrate new services with little or no manual coding. If a service requester's environment supports automated discovery of web services, e.g., uses Visual Studio .NET, the service requester's application can then point to the service provider's WSDL definitions and generate proxies for the discovered web service definitions automatically. This simplifies the invocation of

web services by the service requester's applications as it eliminates the need for constructing complex calls and thus saves many hours of coding.

8.2.1 Web service interface definition

The web service interface definition describes messages and operations in a platform and language independent manner. It describes exactly what types of messages need to be sent and how the various Internet standard messaging protocols and encoding schemes can be employed in order to format the message in a manner compatible with the service provider's specifications. A service interface definition is an abstract service description that can be instantiated and referenced by multiple concrete service implementations. This allows common industry-standard service types to be defined and implemented by multiple service implementers. The service interface contains the WSDL elements that comprise the reusable portion of the service description, these include: the <portType>, <operation>, <message> <part> and <types> elements. These are briefly summarised in the following.

A type attribute in WSDL is comparable to a data type in Java or C++. The WSDL <types> element is used to contain XML schemas or external references to XML schemas that describe the data type definitions used within the WSDL document. WSDL uses a few primitive data types that XML Schema Definition (XSD) defines, such as int, float, long, short, string, boolean and so on, and allows developers to either use them directly or build complex data types based on those primitive ones before using them in messages. This is why developers need to define their own namespace when referring to complex data types. Any complex data type that the service uses must be defined in an optional <types> section immediately before the <message> section. Messages are abstract collections of typed information cast upon one or more logical units, used to communicate information between systems. A <message> element corresponds to a single piece of information moving between the invoker and the service. A regular round trip method call is modelled as two messages, one for the request and one for the response.

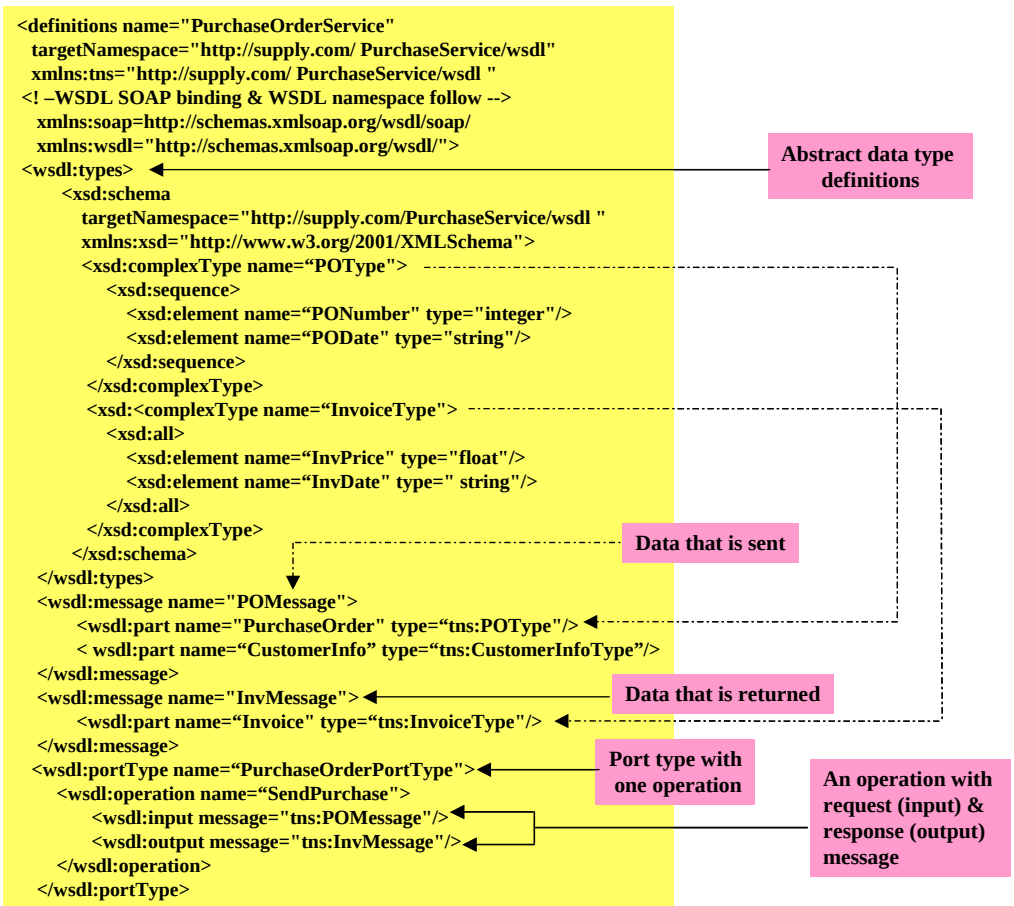


Figure 10 Simple WSDL interface definition.

A message can consist of one or more <part> elements with each part representing an instance of a particular type (typed parameter). When WSDL describes a software module, each part maps to an argument of a method call. If the method returns void, the response is an empty message.

The WSDL <portType> element describes the interface of a web service. It is simply a logical grouping of operations. The <portType> element is central to a WSDL description; the rest of the elements in the definition are essentially details that the <portType> element depends upon. The <portType> is used to bind the collection of logical operations to an actual transport protocol such as SOAP, providing thus the linkage between the abstract and concrete portions of a WSDL document. A WSDL definition can contain zero or more <portType> definitions. Typically, most WSDL documents contain a single <portType>. This convention separates out different web service interface definitions into different documents. This granularity allows each business process to have separate binding definitions, providing for reuse, significant implementation flexibility for different security, reliability, transport mechanism and so on [Cauldwell01].

Figure 10 shows an excerpt of a WSDL interface definition describing a purchase order service. This service takes a purchase order number, a date and customer details as input and returns an associated invoice. The root element in Figure 10 (and every WSDL specification) is the `<definitions>` element, in which a complete description of the web services is provided. The `<definitions>` element consists of attributes, which define the name of the service, the target namespace of the service, and other standard namespace definitions (such as SOAP) used in the service specification. When creating a WSDL document, we need to specify an XML namespace to which the service and related types will belong. In Figure 10 the `<definitions>` element contains an attribute called `targetNamespace`, which defines the logical namespace for information about the service, and is usually chosen to be unique to the individual service (a URL set to the name of the original WSDL file). This helps clients differentiate between web services and prevents name clashes when importing other WSDL files. These namespaces are simply unique strings – they usually do not point to a page on the Web. The `xmlns:tns` (sometimes referred to as *this namespace*) attribute is set to the value of `targetNamespace` and is used to qualify (scope) properties of this service defintion. The namespace definitions `xmlns:soap` and `xmlns:xsdl` are used for specifying SOAP-specific information as well as data types, respectively. The final statement defines `xmlns:` as the default namespace for all WSDL elements defined in a WSDL specification such as messages, operations, and portTypes. The `wsdl:types` definition encapsulates schema definitions of all types using xsd.

The central element externalising a service interface description is the `<portType>` element. This element contains all named operations supported by a service. The WSDL example in Figure 10 defines a web service that contains a `<portType>` named `PurchaseOrderPortType` that supports a single `<operation>`, which is called `SendPurchase`. If there are multiple `<portType>` elements in a WSDL document then each `<portType>` element must have a different name. The example assumes that the service is deployed using the SOAP 1.1 protocol as its encoding style, and is bound to HTTP.

Operations in WSDL are the equivalent of method signatures in programming languages. Operations in WSDL represent the various methods being exposed by the service. An operation defines a method on a web service, including the name of the method and the input and output parameters. A typical operation defines the input and output parameters or exceptions (faults) of an operation.

The `<operation>` element `SendPurchase` in the listing above will be called using the message `POMessage` and will return its results using the message `Inv(oice)Message`. Operations can be used in a web service in four fundamentals patterns: request/response, solicit/response, one-way and notification. The operation `SendPurchase` is a typical example of a request/response style of operation as it contains an input and an output message. The operation patterns are described in section- 8.2.3.

An operation potentially holds all messages potentially exchanged between a web service consumer and a web service provider. If fault messages had been defined, these would also be part of the `<operation>` element. As shown in Figure 10 message `POMessage` is linked by name to the input message element of the `SendPurchase` operation. This message represents the data that is sent from a service requester to a service provider. Similarly, the message `InvMessage` is linked by name to the output message element of the `SendPurchase` operation. This message encapsulates the data of the return value. The input and

output message elements of an operation link the services method, SendPurchase in the case of Figure 10, to SOAP messages that will provide the transport for input parameters and output results.

A message consists of <part> elements, which are linked to <types> elements. While a message represents the overall data type formation of an operation, parts may further structure this formation. In Figure 10 the input message called POMessage contains two <part> elements that refer to the complex types POType and CustomerType, respectively. POType consists of PONumber and PODate elements. The dotted arrow at the right hand side of the WSL definition in Figure 10 links the POType definition to the input message. The same applies to the output message InvMessage and the InvoiceType definition.

The <types> element is a container that contains all abstract data types that define a web service interface. Part elements may select individual type definitions contained in a <types> element. The chosen type definition is directly attached to <part> element. Figure 10 illustrates two complex types that have been defined in its <types> section: POType and InvoiceType. The elements <sequence> and <all> is a standard XSD element. The construct <sequence> requires that the content model follows the element sequence defined, while the construct <all> denotes that all the elements that are declared in the <complexType> statement must appear in an instance document. XSD is also used to create the type namespace and the alias xsd1 is used to reference these two complex types in order to define messages.

8.2.2 WSDL implementation

In the previous WSDL operations and messages have been defined in an abstract manner without worrying about the details of implementation. In fact, the purpose of WSDL is to specify a web service abstractly and then to define how the WSDL developer will reach the implementation of these services. The service implementation part of WSDL contains the elements <binding> (although sometimes this element is considered as part of the service definition) <port> and <service> and describes how a particular service interface is implemented by a given service provider. The service implementation describes where the service is located, or more precisely, to which network address the message must be sent in order to invoke the web service. A web service is modelled as a WSDL service element. The web service implementation elements are summarised below.

1. In WSDL a <binding> element contains information of how the elements in an abstract service interface (<portType> element) are converted into concrete representation in a particular combination of data formats and concrete protocols. The WSDL <binding> element defines how a given operation is formatted and bound to a specific protocol.
2. A <port> defines the location of a service and we can think of it as the URL where the service can be found.
3. A <service> element contains a collection (usually one) of WSDL <port> elements. A <port> associates an endpoint, for instance, a network address location or URL, with a WSDL binding element from a service definition. Each <service> element is named, and each name must be unique among all services in a WSDL document.

The WSDL example in Figure 11 is an implementation description for the abstract service interface listed in Figure 10. The central element of the implementation description is the <binding> element. The <binding> element specifies how the client and web service should exchange messages. The client uses this information to access the web service. This element binds the port type, i.e., the service interface description, to an existing service implementation. It provides information about the protocol and the concrete data formats expected by a service offered from a distinct network address [Zimmermann03]. The binding name must be unique among all the <binding> elements in a WSDL document.

The structure of the <binding> element resembles that of the <portType> element. This is no concidence as the binding must map an abstract port type description to a concrete implementation. The <type> attribute identifies which <portType> element this binding describes. As illustrated in Figure 11 the <binding> element POMessageSOAPBinding links the <portType> element named PurchaseOrderPortType (refer to Figure 10) to the <port> element named POMessagePort.

This is affected through the binding `name POMessageSOAPBinding` as can be seen from the dotted arrow in Figure 11. Several bindings may represent various implementations of the same <portType> element. If a service supports more than one protocol, then the WSDL <portType> element should include a <binding> for each protocol it supports. For a given <portType> element, a <binding> element can describe how to invoke operations using a single messaging/transport protocol, e.g., e.g., SOAP over HTTP, SOAP over SMTP or a simple HTTP POST operation, or any other valid combination of networking and messaging protocol standards. Currently, the most popular binding technique is to use SOAP over HTTP.

It must be noted that a binding does not contain any programming language or service implementation-specific details. How a service is implemented is an issue completely external to WSDL.

In Figure 11 the <binding> element is shown to contain a second <binding> element (in this case <soap:binding>) that specifies the protocol by which clients access the web service. More specifically, the purpose of the SOAP binding element <soap:binding> is to signify that the SOAP protocol format is going to be used as a binding and transport service. This declaration applies to the entire binding. It signifies that all operations of the PurchaseOrderPortType are defined in this binding as SOAP messages. It then becomes the responsibility of SOAP to take the client from the abstract WSDL specification to its implementation. Since SOAP is used for this purpose, SOAP's namespace must also be used. A WSDL implementation allows the use of other protocols, such as HTTP without using SOAP and MIME. If either of these protocols needs to used, the second <binding> element must be declared using the namespace prefixes associated with it, i.e., HTTP or MIME.

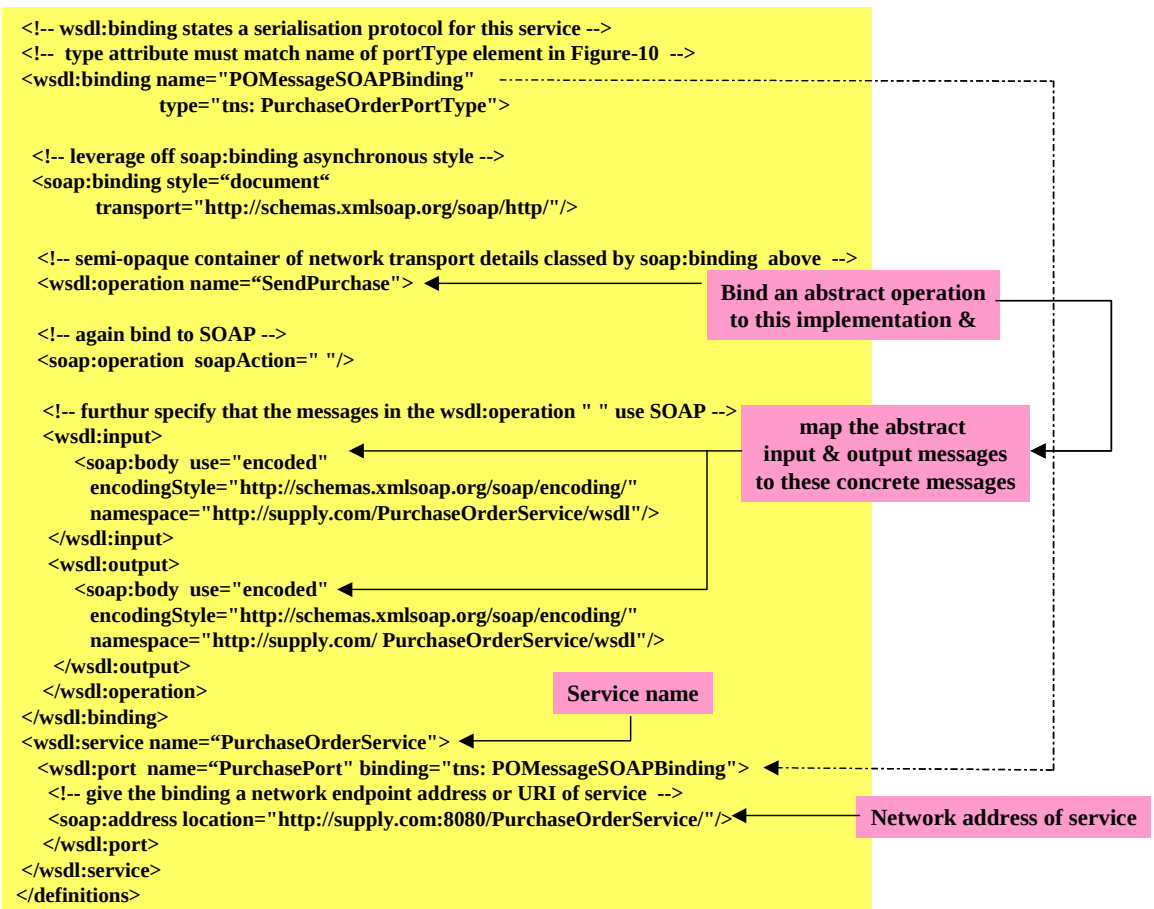


Figure 11 WSDL implementation description.

Figure 11 indicates that the `transport` attribute specifies HTTP as the lower-level transport service that this binding will use. The `style` attribute defines the type of default operations within this binding, which

is “document”; with the other type being “rpc” (see section-8.1.2). The `transport` and `style` attributes are part of the SOAP binding element <soap:binding> (not to be confused with the WSL <binding> element). The abstract operation `SendPurchase` together with its input and output messages from the abstract service interface description (see Figure 10) is mapped to SOAP messages. The data types of these messages that are abstractly described by means of XSD in the service interface description should be SOAP encoded for the transfer [Zimmermann03].

The <operation> element contains instructions on how to access the `PurchaseOrderService`. The <operation> element provides the actual details for accessing the web service. Here, the <soap:operation> element is used to indicate the binding of a specific operation, e.g., `SendPurchase`, to a specific SOAP implementation. The `SOAPAction` attribute in the <soap:operation> element is an attribute that a SOAP client will use to make a SOAP request The `SOAPAction` attribute is a server specific URI used to indicate the intent of request. It can contain a message routing parameter or value that helps the SOAP runtime system dispatch the message to the appropriate service. The value specified in this attribute must also be specified in the `SOAPAction` attribute in the HTTP header of the SOAP request. The purpose of this is to achieve interoperability between client and service provider applications. The SOAP client will read the SOAP structure from the WSDL file and coordinate with a SOAP server on the other end.

The <soap:body> element enables applications to specify the details of the input and output messages and enable the mapping from the abstract WSDL description to the concrete protocol description. In the case of `PurchaseOrderService`, the <soap:body> element specifies the SOAP encoding style and the namespace URN associated with the specific service. The <input> and <output> elements for the <operation> `SendPurchase` specify exactly how the input and output messages of this operation should appear in the SOAP message. Both input and output contain a <soap:body> element with the value of its namespace corresponding to the name of the service that is deployed on the SOAP server. Consider for example, the <input> elements for the `SendPurchase` operation. The entire `POMessage` message from the <portType> declaration for the `SendPurchase` operation is declared to be abstract. This is indicated by the `use=”encoded”` attribute. This means that the XML defining the input message and its parts are in fact abstract, and the real, concrete representation of the data is to be derived by applying the encoding scheme indicated in the `encodingStyle` attribute [Graham02]. This implies that the message should appear as part of the <soap:body> element and that the SOAP runtime system on the service provider’s network should deserialise the data from XML to another format, e.g., Java data types, according the encoding rules defined in the SOAP specification.

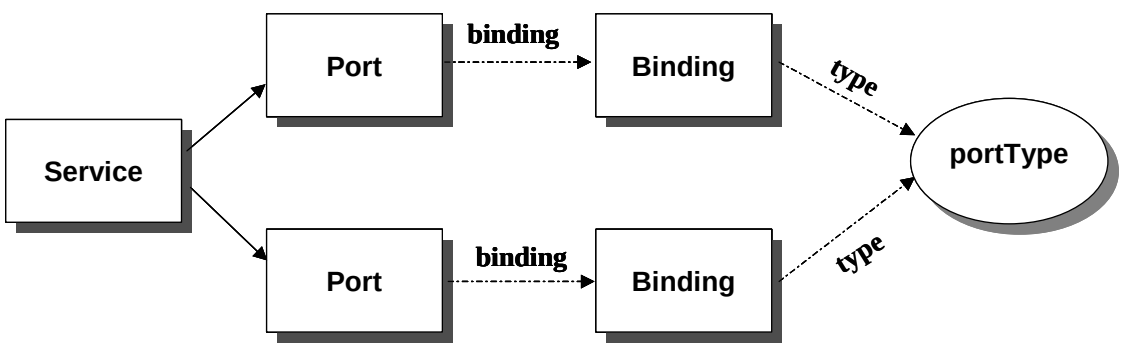


Figure 12 Connecting the service interface withy the service implementation.

The <port> element represents the actual network endpoint(s) on which the service communicates. A web service exchanges messages in a defined format through a <port> element. More precisely, the <port> element a single protocol-specific address to an individual binding element. Here, a mandatory location URI must be provided to denote the physical endpoint that requesters must use to connect to the service. The <soap:binding> attribute is a SOAP extension to WSDL used to connect the port (URI) with the protocol in the <binding> element. The <soap:address> attribute is another SOAP extension to WSDL

and is used to signify the URI of the service or the network endpoint. A web service client is expected to bind to a port and interact with the service, provided that it understands and respects the concrete message expected by the service. The same service may be offered with various data formats over multiple ports. A client that wishes to interact with the service can then choose one of these ports.

A service may be exposed via multiple ports each with a different binding, e.g., one for SOAP and one for HTTP GET. WSDL is extensible to allow description of endpoints and their messages, regardless of what message formats or network protocols are used to communicate. The currently described bindings are for SOAP 1.1, HTTP POST and MIME.

A <service> is modelled as a collection of related ports - where a <port> element is a single end point defined as a combination of binding and a network address - at which a service is made available. The <service> element may be the starting point for a client exploring a service description. A single service can contain multiple ports that all use the same <portType>, i.e., there could be multiple service implementations for the same service interface provided by different service providers, see Figure 12. This figure shows that a service may contain more than one ports, which are bound to binding elements, and that binding elements are associated with a <portType > by means of a type relationship. Service providers all have different bindings and/or addresses. Port addresses are specified by the <saop:address> element of <port>, as already explained. In the case of multiple ports, these ports are alternatives, so that the client of the service can choose which protocol they want to use to communicate with the service (possibly programmatically by using the namespaces in the bindings), or which address is closest [Cauldwell01].

The previous example contains only one web service, viz. the PurchaseOrderService, thus only the <port> element named PurchasePort (refer to Figure 11) is used to reveal the service location. However, the service PurchaseOrderService could, for instance, contain three ports all of which use the PurchaseOrderPortType but are bound to SOAP over HTTP, SOAP over SMTP, and HTTP GET/POST, respectively. This would give the client of the POMessage service a choice of protocols over which to use the service. For instance, a PC-desktop application may use SOAP over HTTP, while a WAP application designed to run on a cellular phone may use HTTP GET/POST, since an XML parser is typically not available in a WAP application. All three services are semantically equivalent in that they all take a purchase order number, a date and customer details as input and return an associated invoice. By employing different bindings, the service is more readily accessible on a wider range of platforms [Cauldwell01].

A <service> is modelled as a collection of related ports - where a <port> element is a single end point defined as a combination of binding and a network address - at which a service is made available. The <service> element may be the starting point for a client exploring a service description. A single service can contain multiple ports that all use the same <portType>, i.e., there could be multiple service implementations for the same service interface provided by different service providers. These all have different bindings and/or addresses. Port addresses are specified by the <saop:address> element of <port>, as already explained. In the case of multiple ports, these ports are alternatives, so that the client of the service can choose which protocol they want to use to communicate with the service (possibly programmatically by using the namespaces in the bindings), or which address is closest [Cauldwell01].

Figure 13 summarises several of the constructs explained in the previous by illustrating the various WSDL elements involved in a client-service interaction. This figure shows one client invoking a web service by means of SOAP over HTTP and another client invoking the same service by means of HTTP.

8.2.3 WSDL interaction patterns

WSDL interfaces (port types in the WSDL terminology) support four types of operations. These operations represent the most common interaction patterns for web services. Since each operation defined by WSDL can have an input and/or an output, the four WSDL interaction patterns represent possible combinations of input and output messages [Cauldwell01]. Thus the WSDL operations correspond to the incoming and outgoing versions of two basic operation types: an incoming single message passing operation and its outgoing counterpart (“one-way” and “notification” operations), and the incoming and outgoing versions of a synchronous two-way message exchange (“request-response” and “solicit response”).

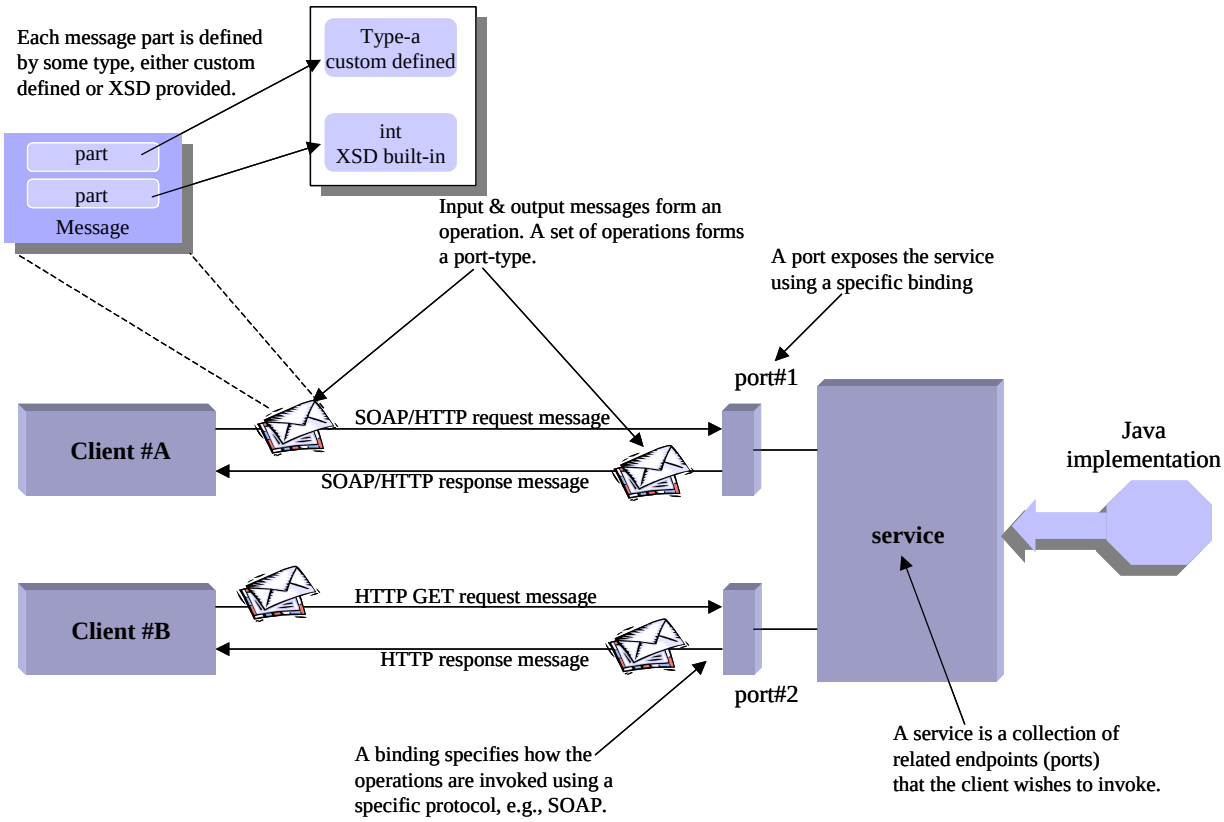


Figure 13 Elements of the WSDL as part of requester-service interaction.

1. One way operation: A one-way operation is an operation in which the service end point receives a message, but does not send a response. An example of a one-way operation might be an operation representing the submission of an order to a purchasing system. Once the order is sent, no immediate response is expected. In an RPC environment, a one-way operation represents a procedure call to which no return value is assigned. A one-way message defines only an input message. It requires no output message and no fault.
2. Request/response operation: A request/response operation is an operation in which the service end point receives a message and returns a message in response. In the request/response message pattern a client requests that some action is taken from the service provider. An example of this is the `SendPurchase` operation, which receives as input message a containing a flight number and date and responds with a message containing its current price. In an RPC environment this is equivalent to a procedure call, which takes a list of input arguments and returns a value.
3. Solicit/response operation: A solicit/response operation is an operation in which the service end point sends a message and expects to receive an answering message in response. This is the opposite of the request/response operation since the service endpoint is initiating the operation (soliciting the client), rather than responding to a request. An example of this operation might be a service that sends out order status to a client and receives back a receipt.
4. Notification operation: A notification operation is an operation in which the service end point sends a message and receives no response. This type of messaging is used by services that need to notify clients of events. An example of this could be a service model in which events are reported to the service and where the endpoint periodically reports its status. No response is required in this case, as most likely the status data is assembled and logged and not acted upon immediately.

Any combination of incoming and outgoing operations can be included in a single WSDL interface. As a result, the four types of operations presented above provide support for both push and pull interaction models at the interface level. The inclusion of outgoing operations in WSDL is motivated by the need to support loosely coupled peer-to-peer interactions between services.

8.3 UDDI: Universal Description, Discovery, and Integration

One of the main reasons for enterprises engaging in electronic business is to open new markets and find new sources of supply more easily than with conventional means. To achieve this desired state, however, enterprises need a common way of identifying potential trading partners and cataloguing their business functions and characteristics. The solution is the creation of a service registry architecture that presents a standard way for enterprises to build a registry to describe and identify e-business services, query other service providers and enterprises, and enable those registered enterprises to share business and technical information globally in a distributed manner. To address this challenge, the Universal Description, Discovery and Integration (UDDI) specification was created. UDDI is a cross industry initiative to create a registry standard for web service description and discovery together with a registry facility that facilitates the publishing and discovery processes. UDDI provides a global, platform-independent, open framework to enable service clients to:

- discover information about enterprises offering web services,
- find descriptions of the web services these enterprises provide,
- find technical information about web service interfaces and definitions how the enterprises may interact over the Internet.

A business may set up multiple UDDI registries in-house to support intranet and e-business operations, and a business may use UDDI registries set up by its customers and business partners. The UDDI Business Registry (UBR) is a free, public registry operated by IBM, Microsoft, SAP, and NTT. The UBR provides a global directory about accessing publicly available web services. UBR has a role analogous to the role that DNS (Domain Name Service) has in the Internet infrastructure. It enables users to locate businesses, services and service specifications.

UDDI is designed for use by developer tools and applications that use web services standards such as SOAP/XML and WSDL. UDDI is a group of web-based registries designed to house information about businesses and web services they provide in a structured way. One key difference between a UDDI registry and other registries and directories is that UDDI provides a mechanism to categorize businesses and services using taxonomies. For example, service providers can use a taxonomy to indicate that a service implements a specific domain standard, or that it provides services to a specific geographic area [Manes03]. Such taxonomies make it easier for consumers to find services that match their specific requirements. Once a web service has been developed and deployed it is important that it is published in a registry, such as UDDI, so that potential clients and service developers can discover it, see Figure 4. Web service discovery is the process of locating and interrogating web service definitions, which is a preliminary step for accessing a web service. It is through this discovery process that web service clients learn that the web service exists, what its capabilities are, and how to properly interact with it.

The core concept of the UDDI initiative is the UDDI business registration, an XML document used to describe a business entity and its web services. Conceptually, the information provided in a UDDI business registration consists of three inter-related components: “white pages” including address, contact, and other key points of contact; “yellow pages” classification information according to industrial classifications based on standard industry taxonomies; and “green pages”, the technical capabilities and information about services that are exposed by the business including references to specifications for web services and pointers to various file and URL based discovery mechanisms. Using a UDDI registry, enterprises can discover the existence of potential trading partners and basic information about them (through white pages), find companies in specific industry classifications (through yellow pages), and uncover the kind of web services offered to interact with the enterprises (through green pages).

UDDI has been designed in a highly normalised fashion, not bound to any technology. In other words, an entry in the UDDI registry can contain any type of resource, independently of whether the resource is XML-based or not. For instance, the UDDI registry could contain information about an enterprise’s EDI system, DCOM or CORBA interface, or even a service that uses the fax machine as its primary communication channel. The point is that while UDDI itself uses XML to represent the data it stores, it allows for other kinds of technology to be registered.

8.3.1 UDDI data structures

Although UDDI is often thought simply as a directory mechanism, it also defines a data structure standard for representing company and service description information. Through UDDI, enterprises can publish and discover information about other businesses and the services they provide. This information can be classified using standard taxonomies so that information can be discovered on the basis of categorisation. UDDI contains also information about the technical interfaces of an enterprise's services.

The data model used by the UDDI registries is defined in an XML schema. XML was chosen because it offers a platform-neutral view of data and because it allows hierarchical relationships to be described in a natural way. The XML schema standard was chosen because of its support for rich data types as well as its ability to easily describe and validate information based on information models represented in schemas. The data UDDI contains is relatively lightweight; as a registry its prime purpose is to provide network addresses to the resources it describes, e.g., schemas, interface definitions, or endpoints, in locations across the network. The UDDI specification provides a platform-independent way of describing services, discovering businesses, and integrating business services using the Internet. The UDDI data structures provide a framework for the description of basic business and service information, and architects an extensible mechanism to provide detailed service access information using any standard service description mechanism.

The UDDI XML schema defines four core types of information that provide the white/yellow/green page functions. These are: business entities; business services, binding templates; and information about specifications for services (technical or tModels). A service implementation registration represents a service offered by a specific service provider. The UDDI XML schema specifies information about the business entity e.g., a company, that offers the service (<businessEntity>), describes the services exposed by the business (<businessService>), and captures the binding information (<bindingTemplate>) required to use the service. The <bindingTemplate> captures the service endpoint address, and associates the service with the <tModel>s that represent its technical specifications. Each business service can be accessed in one or more ways. For example, a retailer might expose an order entry service accessible as a SOAP-based web service, a regular web form or even a fax number. To convey all the ways a service is exposed each service is bound to one or more <tModels> via a binding template.

The data model hierarchy and the key XML element names that are used to describe and discover information about Web services are shown in Figure 14. These are outlined briefly in the following.

Business information: Partners and potential clients of an enterprise's services that need to be able to locate information about the services provided would normally have as starting point a small set of facts about this service provider. They will know, for example, either its business name or perhaps some key identifiers, as well as optional categorisation and contact information (white pages). The core XML elements for supporting publishing and discovering information about a business – the UDDI Business Registration -- are contained in an element named <businessEntity>. This element serves as the top-level structure and contains information about a particular business unit itself. The XML element <businessEntity> contains information such as the company name and contacts (white page listing). The <businessEntity> construct is a top-level structure that corresponds to "white page".

Business service information: The <businessService> structures represent logical service classification about a family of web services offered by the company. The top-level entity <businessEntity> described above can contain one or more <businessService> elements for each of these service families. The <businessService> structure is a descriptive container that is used to group a series of related web services related to either a business process or category of services. It is used to reveal service related information such as the name of a web service aggregate, a description of the web service or categorisation details. Examples of business processes that would include related web service information include purchasing services, shipping services, and other high-level business processes. <businessService> information sets, such as these, can each be further categorised – allowing web service descriptions to be segmented along combinations of industry, product and service or geographic category boundaries. The kind of information contained in a <businessService> element maps to the "yellow pages" information about a company.

Binding information: The access information required to actually invoke a service is described in the information element named <bindingTemplate>. The <binding Template> (the structure that models

binding information) data structure exposes a service endpoint address required for accessing a service from a technical point of view. Technical descriptions of web services – the "green pages" data -- reside within sub-structures of the <businessService> element. Within each <businessService> reside one or more technical web- service descriptions. These structures provide support for determining a technical end point or optionally support remotely hosted services, as well as a lightweight facility for describing unique characteristics of a given implementation. Support for technology and application specific parameters and settings are also supported. This information is relevant for application programs and clients that need to connect to and then communicate and invoke a remote web service. Each <businessService> may potentially contain multiple <binding Template> structures, each of which describes a web service (see Figure 14).

Specification pointers and technical fingerprints: It is not always enough to simply know where to contact a particular web service through its endpoint address revealed by a <bindingTemplate> data type. For instance, if we know that a business partner provides a web service that accepts purchase orders, knowing the URL for that service is not very useful unless technical details such as what format the purchase order should be sent in, what protocols are appropriate, what security is required, and what form of a response will result after sending the purchase order, are also provided. Integrating all parts of two systems that interact via web services can become quite complex and thus requires information about compatibility of service specifications to make sure that the right web services are invoked. For this reason, each <bindingTemplate> data type contains a special <tModel> data structure (short for "Technology Model") that provides information about a web service interface specification. This data structure forms a *technical fingerprint* that represents technical fingerprints, abstract types of metadata and interfaces, which can be used to recognize a web service that implements a particular behaviour or programming interface. For instance, in the case of a purchase order, the web service that accepts the purchase order exhibits a set of well-defined behaviours if the proper document format is sent to the proper address in the right way. A UDDI registration for this service would consist of an entry for the business partner <businessEntity>, a logical service entry that describes the purchasing service <businessService>, and a <bindingTemplate> entry that describes the purchase order service by listing its URL and a reference to a <tModel> that is used to provide information about the service's interface and its technical specification. The <tModel> contains metadata about a service specification, including its name, publishing organization, and URL pointers to the actual specifications themselves.

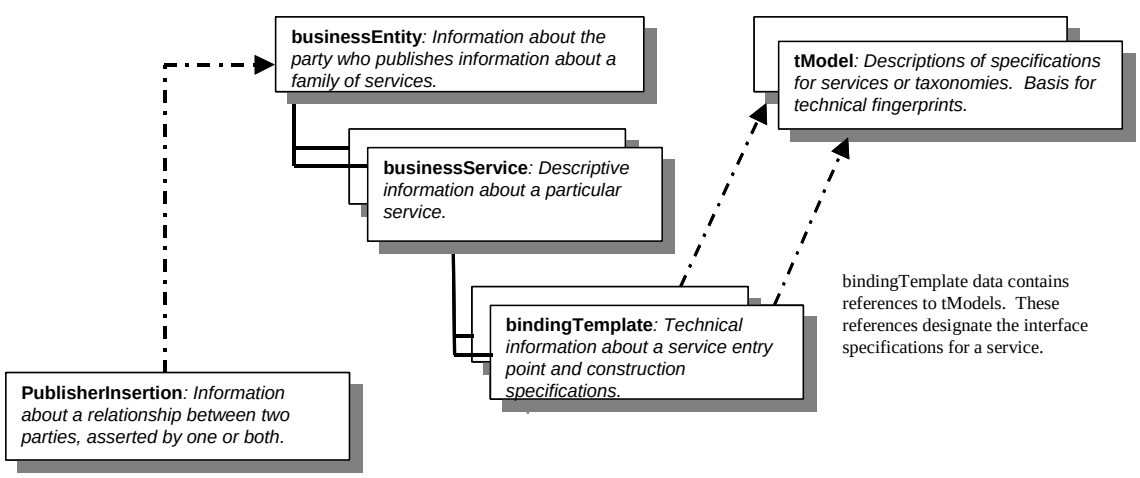


Figure 14 Overview of UDDI data structures.

The UDDI, just like WSDL, draws a sharp distinction between abstraction and implementation. In fact, the primary role that a <tModel> plays is to represent technical information about an abstract interface specification. An example might be a specification that outlines wire protocols and interchange formats [Ehnebuske 01]. These can, for instance be found, in the RossettaNet Partner Interface Processes, the Open Applications Group Integration Specification and various Electronic Document Interchange (EDI) efforts and so on. A corollary of the <tModel> structure is the <bindingTemplate>, which is the concrete implementation of one or more <tModel>s. Inside a binding template, businesses can register the access point for a particular implementation of a <tModel>. <tModel>s can be published separately from <bindingTemplate>s that reference them. For instance, a standard's body or industry group might publish

the canonical interface for a particular use case or vertical industry sector, and then multiple enterprises could code implementations to this interface. Accordingly, each of those business's implementations would refer to the same <tModel>. A set of canonical <tModel>s has been defined that standardize commonly used classification mechanisms. The UDDI operator sites have registered a number of canonical <tModels> for NAICS (an industry code taxonomy), UNSPC (a product and service code taxonomy) and ISO 3166 (a geographical region code taxonomy), identification taxonomies like Dun and Bradstreet's D-U-N-S and Thomas Register supplier identification codes.

Due to the fact that both the UDDI and WSDL schema have been architected to delineate clearly between interface and implementation, these two constructs are quite complementary work together naturally. The WSDL-to-UDDI *mapping model* is designed to help users find services that implement standard definitions. The WSDL-to-UDDI mapping model describes how WSDL <portType> and <binding> element specifications can become <tModel>s; how the <port>s of WSDL become UDDI <bindingTemplate>s; and how each WSDL service is registered as a <businessService> [Manes03].

By decoupling a WSDL specification and registering it in UDDI, we can populate UDDI with standard interfaces that have multiple implementations, providing a landscape of business applications that share interfaces.

8.3.2 UDDI usage model and deployment variants

Figure 15 shows the basic UDDI usage model. The UDDI usage model involves standard bodies and industry consortia publishing the descriptions of available services. Subsequently, the service providers implement and deploy web services conforming to these type definitions. Prospective clients can then query the UDDI registry based on various criteria such as the name of the business, product classification categories, or even services that implement a given service type definition. These clients can then get the details of the service type definition from the location specified. Finally, the clients can invoke the web service as they have the service end point, and also the details on how to exchange messages with it. The UDDI usage model envisages different business information provider roles such as:

- Registry operators: these refer to the organisations (referred to as operator nodes) that host and operate the UDDI Business Registry. The operator nodes manage and maintain the directory information, cater for replication of business information and other directory related functions. These operators provide a Web interface to the UDDI registry for browsing, publishing and un-publishing business information. The UDDI operator nodes allow businesses to publish their information and services they offer and they follow a well-defined replication scheme. An enterprise does not need to register with each of these operators separately it can register at any one of the operator companies. The registry works on a "register once, published everywhere" principle.
- Standard bodies and industry consortia: these publish descriptions in the form of service type definitions (<tModel>s). These <tModel>s do not contain the actual service definitions, instead they have a URL that points to the location where the service descriptions are stored (definitions can be in any form, however UDDI, recommends using WSDL).
- Service providers: commonly implement web services conforming to service type definitions supported by UDDI. They publish information about their business and services in the UDDI. The published data also contains the end point of the web services offered by these enterprises.

The term UDDI in this paper is often used to mean both the protocol with its data structures and Application Program Interface (API), that is publish and find operations, as well as the global UDDI Business Registry described in the previous. This is not, however, the only way that UDDI registry can be deployed, there are other deployment possibilities, which we will briefly cover in what follows.

The structure of the UDDI allows the possibility of private UDDI nodes. A private (non-operator) UDDI node can implement all the UDDI functionality, but it does not participate in the replication scheme defined by the UDDI operator's agreement. Because of the data volumes, the breadth of industry, geography, products covered, and the requirement of adherance to the operator agreement, operator nodes are limited to the degree of additional or variant behaviour they can provide. Private nodes are not under the same restrictions. Currently, we can discern between the following UDDI deployment possibilities [Graham 01]:

- The e-marketplace UDDI: an e-marketplace, a standards body, or a consortium of organisations that participate and compete in the industry can host this private UDDI node. The e-marketplace could run a local version of a UDDI registry with its data shielded from the global UDDI registry. The entries in this private UDDI relate to a particular industry or narrow range of related industries. The e-marketplace node can then provide value added services such as quality of service monitoring, validation of content published by companies, ensure that participants in the UDDI registry have been vetted by a rigorous selection procedure, and also ensure that all entries pertain to the market segment of interest. In such an environment publish and find operations provided by an API could be restricted to the legitimate businesses registered with the marketplace. Such a deployment might be not free of charge like the global registry and may charge a fee, either from the service providers or from the clients for providing such value added services [Wahli04].

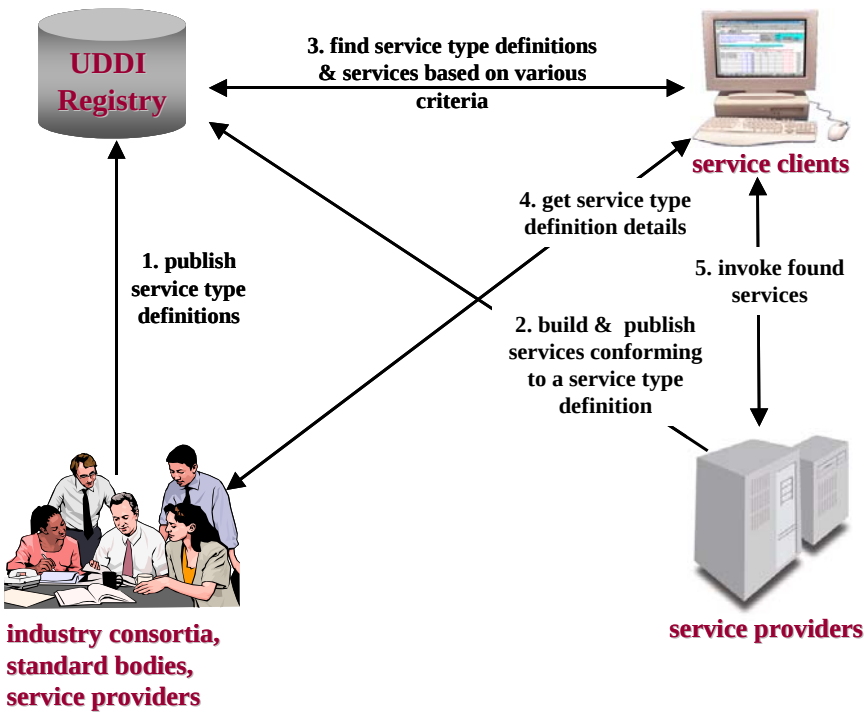


Figure 15 The UDDI usage model.

- The business partner UDDI registry: this variant of the above scheme is a private UDDI node hosted behind one of the business partner's firewall and only trusted or vetted partners can access the registry. It also contains web service description meta-data published by trusted business parties (that is, those organisations with which the hosting organisation has formal agreements/relationships).
- The portal UDDI: this type of deployment is on an enterprise's firewall and is a private UDDI node that contains only meta-data related to the enterprise's web services. External users of the portal would be allowed to invoke find operations on the registry, however, a publish operation would be restricted to services internal to the portal. The portal UDDI gives a company ultimate control over how the meta-data describing its web services is used. For example, an enterprise can restrict access. It can also monitor and manage the number of lookups being made against its data and potentially get information about who the interested parties are.
- The internal UDDI: this allows applications in different departments of the organisation to publish and find services, and is useful for large organisations. The major distinction of this UDDI variant is the potential for a common administrative domain that can dictate standards (for example a fixed set of tModels can be used). This allows the UDDI node to operate with different publish restrictions than those suggested for the business partner UDDI. For example, the node could restrict the publication of new tModels and thereby restrict publishing of <businessService>s and <bindingTemplate> to accept only entries associated with a fixed set of tModels [Graham 01].

These kinds of UDDI deployments are called EAI UDDI, as they allow corporations to deploy and advertise Intranet web services.

The closed registries covered above offer some advantages over the global UDDI Business Registry. The global Business registry does not restrict how the service is described; hence an enterprise could describe its services by a variety of means. It could use a URL pointing to a text description of the service, a description in WSDL, or whatever means the company chooses to use. While this allows for flexibility, it severely restricts the ability of applications to interoperate as an application can really do anything meaningful with the results of a find operation. Instead, if the description (meta-data) were modelled using WSDL (this is recommended as best practice), an application could use dynamic find and bind operations on the service [Wahli04].

8.3.3 UDDI application programming interface

UDDI uses SOAP as its transport layer, thus enterprises can interact with UDDI registries through SOAP-based XML API calls in order to discover technical data about an enterprise's services. In this way enterprises can link up with service providers and invoke and use their services.

The UDDI API is an interface that accepts XML messages wrapped in SOAP envelopes [McKee01]. All UDDI interactions use a request-response model, in which each message requesting service from the site generates some kind of response. A developer would not be creating messages in this format – there are toolkits available that shield developers from the task of manipulating XML documents. These toolkits (or the APIs they provide) internally "talk" to the UDDI registry using the XML message formats defined by the UDDI protocol. Developers can build applications in Java, VisualBasic or any other language of their choice to access UDDI registries and either publish their services, find services provided by other companies or unpublish their service listings. Currently, UDDI client side libraries/toolkits exist for Java, VisualBasic and Perl [Cauldwell01].

The UDDI specifications allow the following types of exchanges with UDDI registered sites: *enquiries* and *publishing*.

- *Enquiries* enable parties to find business businesses, services, or bindings (technical characteristics) meeting certain criteria. The party can then get the corresponding <businessEntity>, <businessService>, or <bindingTemplate> information matching the search criteria. The UDDI enquiry API has two usage patterns: *browse* and *drill down*. A developer would, for instance, use a browse pattern (**find( )** API calls) to get a list of all entries satisfying broad criteria to find the entries, services or technical characteristics and then use the drill down pattern (**get( )** API calls) to get the more specific features. For example, a **find_business( )** call could be first issued to locate all businesses in a specific category area, and then a **get_BusinessDetail( )** call could be used to get additional information about a specific business.

    1. The browse pattern uses the following five methods: **find_binding( )**, **find_business( ), find_relatedBusinesses( ), find_service ( ),** and **find_tModel( ).** The **find_binding( )** method is used to locate specific bindings within a registered business service and returns the binding template(s) that match the search criteria. The binding templates have information on invoking services. The method **find_business( )** helps to locate one or more business entries that match the search criteria. The search can be performed on the partial name of the business, the business identifiers, the category/classification identifiers or the technical fingerprints of the services. The method **find_relatedBusinesses( )** is used to locate information about business entity registrations that are related to the business entity. The **find_service( )** method returns a list of business services that match the search criteria. The **find_tModel( )** method returns a list of <tModels>.

    2. The drill down pattern uses the following five methods: **get_bindingDetail( )**, **get_BusinessDetail( )**, **get_BusinessDetailExt( )**, **get_serviceDetail( )**, and **get_tModelDetail( )**. The method **get_bindingDetail( )** returns the run-time binding information (<bindingTemplate> structure) used for invoking methods against a business service. The method **get_BusinessDetail( )** returns the complete <businessEntity> object for one or more business entities. The method **get_BusinessDetailExt( )** is identical to the method

get_BusinessDetail( ), but returns extra attributes in case the source registry is not an operator node. The method **get_serviceDetail( )** returns the complete <businessService> object for a given business service. The method **get_tModelDetail( )** returns <tModel> details.

- UDDI sites use *publishing* functions to manage the information provided to requestors. The publishing API essentially allows applications to save and delete the five data structures supported by UDDI and described earlier in section-8.3.1 and Figure 14. These calls are used by service providers and enterprises to publish and un-publish information about themselves in the UDDI registry. These API calls require authenticated access to the registry, unlike the enquiry API [Cauldwell01]. UDDI does not specify authentication procedures, but leaves them up to the operator site.

9 Web services coordination, orchestration and choreography

Web service technologies provide the foundation for peer systems to perform units of work cooperatively over complex interactions materialized by long running message interchanges. However, the interaction model that is directly supported by WSDL is essentially a stateless model of uncorrelated synchronous or asynchronous interactions. For instance, models for e-business interactions typically require specifying sequences of peer-to-peer message exchanges between a collection of web services, both synchronous and asynchronous, within stateful, long-running interactions involving two or more parties. Such interactions require that business processes are described in terms of a business protocol (or abstract business model) that precisely specifies the mutually visible message exchange behaviour of each of the parties involved in the protocol, without revealing their internal implementation. It also requires modelling the actual behaviour of participants involved in a business interaction. To define such business processes and protocols, a formal description of the message exchange protocols used by business processes in their interactions is needed [Bloch03].

The concept of service *coordination* has been introduced to support the execution of complex units of work involving multiple services. There are two competing specifications that address this concept: WS-Coordination and WS-CAF (Web Service Composite Application Framework), both are still at the draft stage [Little03c]. These specifications suggest that at a minimum peer message interchanges require a context to be managed and be available during the interchange. This context can be passed by value or by reference. Most often, the boundaries of the units of work that constitutes the interchange will be made explicit. An activity lifecycle service may be used to identify and manage the corresponding activity instances.

WS-CAF presents a well abstracted view of the architecture needed to support long running interactions between web services. This architecture can be used as a foundation to address all aspects of complex web service interactions: B2B collaborations, business processes, orchestration, composition, choreography and of course transactions which can all share the architecture proposed by WS-CAF.

9.1 Orchestration versus choreography

When the message interchange occurs within the context of an e-business solution, it enables the automation of cross-enterprise business processes. However, additional semantics are required to enable the description and handling of the collaboration aspects of the business processes, e.g., commitments and exchange of economic resources, in a standard form that can be consumed by tools for process implementation and monitoring. Enterprise workflow systems today support the definition, execution and monitoring of long-running processes that coordinate the activities of multiple business applications. But because these systems are activity oriented and not communication (message) oriented, they do not separate internal implementation from external protocol description [Leymann00]. When processes span business boundaries, loose coupling based on precise external protocols is required because the parties involved do not share application and workflow implementation technologies, and will not allow external control over the use of their backend applications. Such business interaction protocols are by necessity message-centric; they specify the flow of messages representing business actions among trading partners, without requiring any specific implementation mechanism.

The full potential of web services as a means of developing e-business solutions will only be achieved when applications and business processes are able to integrate their complex interactions. Web services technologies offer a viable solution to this problem since they support coordination and offer an asynchronous and message oriented way to communicate and interact with application logic. However, when looking at web services, it is important to differentiate between baseline specifications of SOAP, UDDI and WSDL that provide the infrastructure that supports publishing, finding and binding operations in the service-oriented architecture (see Figure 4) and higher-level specifications required for e-business integration. These higher-level specifications provide functionality that supports and leverages web services and enables specifications for integrating automated business processes.

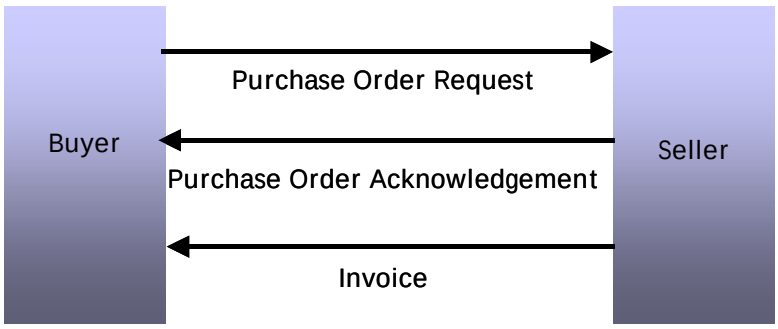


Figure 16 A typical business process.

Currently, there are competing initiatives for developing business process definition specifications, which aim to define and manage business process activities and business interaction protocols comprising collaborating web services. The terms "orchestration" and "choreography" have been widely used to describe business interaction protocols comprising collaborating web services. There is an important distinction between web services orchestration and choreography [Pelz03a]:

- *Orchestration* describes how web services can interact with each other at the message level, including the business logic and execution order of the interactions from the perspective and under control of a single endpoint. Orchestration refers to an executable business process that may result in a long-lived, transactional, multi-step process model. With orchestration, the business process interactions are always controlled from the (private) perspective of one of the business parties involved in the process. In this context, web service *composition* represents a limited application of orchestration.
- *Choreography* is typically associated with the public (globally visible) message exchanges, rules of interaction and agreements that occur between multiple business process endpoints, rather than a specific business process that is executed by a single party. Choreography is more collaborative in nature than orchestration. It is described from the perspectives of all parties (common view), and defines the complementary observable behaviour between participants in business process collaboration. A common view, in essence, defines the shared state of the interactions between business entities. This common view can be used to determine specific deployment implementations for each individual entity. Choreography offers a means by which the rules of participation for collaboration can be clearly defined and agreed to, jointly. Each entity may then implement its portion of the choreography as determined by their common view. Choreography tracks the sequence of messages that may involve multiple parties and multiple sources, including customers, suppliers, and partners, where each party involved in the process describes the part they play in the interaction and no party "owns" the conversation.

We use Figure 16 to exemplify the concept of orchestration and choreography. This figure shows a typical business process comprising a purchase order. A buyer may start a correlated message exchange with a seller by sending a purchase order (PO) and using a PO number in the purchase order document. The seller can use this PO number in the PO acknowledgement. The buyer may later send an invoice containing an invoice document that carries both the PO number, to correlate it with the original purchase order, and also an invoice number.

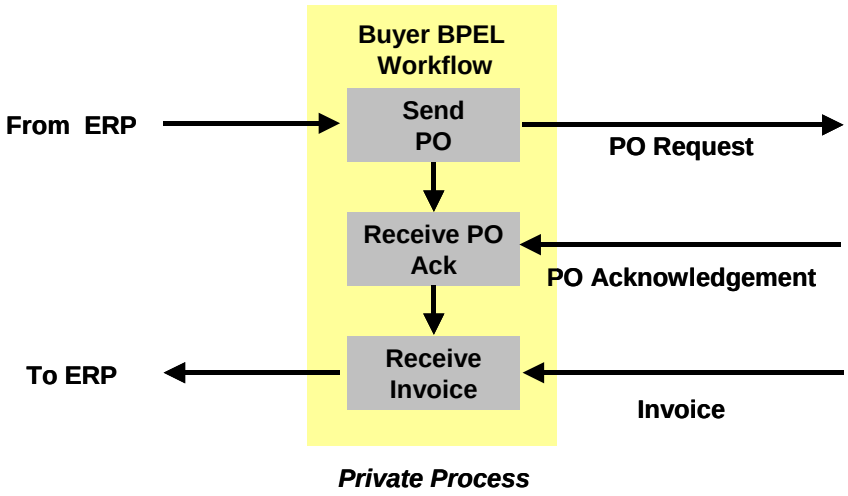


Figure 17 Purchase order from an orchestration perspective.

Figure 17 shows the purchase order process from an orchestration perspective. This process is shown from the perspective of an Enterprise Resource Planning (ERP) application, which can be employed to check inventory, place orders and organise buyer resources. The private buyer process can be specified using a typical business process execution language such as BPEL.

Figure 18 shows the purchase order from a choreography perspective involving an observable public exchange of messages. This choreography can be specified using a service choreography specification language such as WS-CDL.

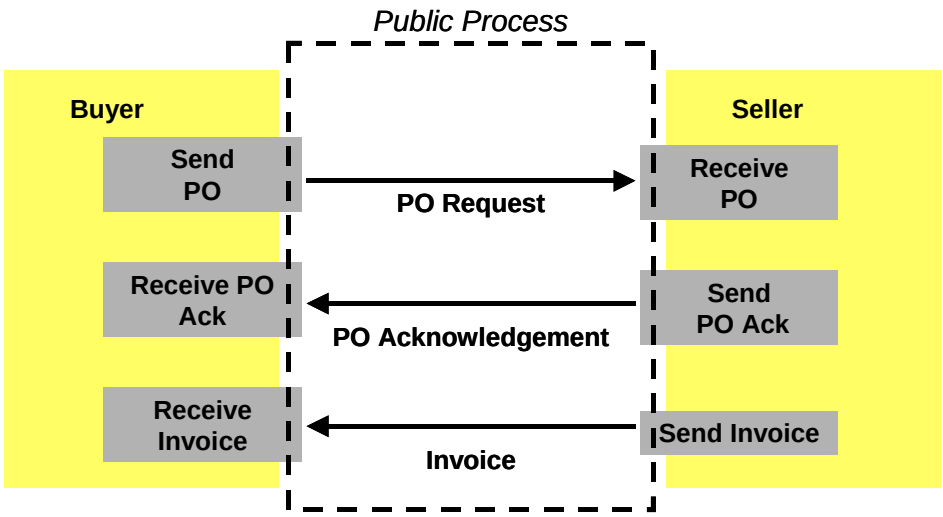


Figure 18 Purchase order from a choreography perspective.

Finally in Figure 19, the buyer and seller are shown to integrate their business processes. The respective business analysts at both companies agree upon the rules and processes involved for the process collaboration. Using a GUI and a tool that can serve as a basis for the collaboration, the buyer and seller agree upon their interactions and generate a WS-CDL representation. The WS-CDL representation can then be used to generate a BPEL workflow template for both the buyer and seller. The two BPEL workflow templates reflect the business agreement.

In the following section we shall first concentrate on the Business Process Execution Language for Web Services (BPEL for short), which is the standard industry specification that is designed specifically for web services based orchestration, and then briefly summarise WS-CDL.

9.2 The orchestration business process execution language

BPEL has recently emerged as the standard to define and manage business process activities and business interaction protocols comprising collaborating web services. This is an XML-based flow language for the formal specification of business processes and business interaction protocols. By doing so, it extends the web services interaction model and enables it to support business transactions. Enterprises can describe complex processes that include multiple organisations— such as order processing, lead management, and claims handling—and execute the same business processes in systems from other vendors.

BPEL provides support for both executable and abstract business processes. An *executable process* models the actual behaviour of participants in the overall business process, essentially modelling a private workflow. This can be perceived as a control "meta-process" that is laid on top of web services controlling their invocations and abstracting their behaviour into a shape that looks very much like a composition of services. The logic and state of the process determine the nature and sequence of the web service interactions conducted at each business partner, and thus the interaction protocols. No attempts are made to separate externally visible (public) aspects of the business process from its internal behaviour. *Abstract processes* are modelled as business protocols in BPEL. Their purpose is to specify the public message exchanges of each of the parties leveraging the protocol. Unlike executable business processes, business protocols are not executable and do not reveal the internal details of a process flow. Abstract business processes link web service interface definitions with behavioural specifications that can be used to control the business roles and define the behaviour that each party is expected to perform within the overall business process. For example, in a supply-chain business protocol, the buyer and the seller are two distinct roles, each with its own abstract process. Their relationship is typically modelled as a <partner-link>. Essentially, executable processes provide the orchestration support described earlier while the business protocols concentrate more on the choreography aspects of the services.

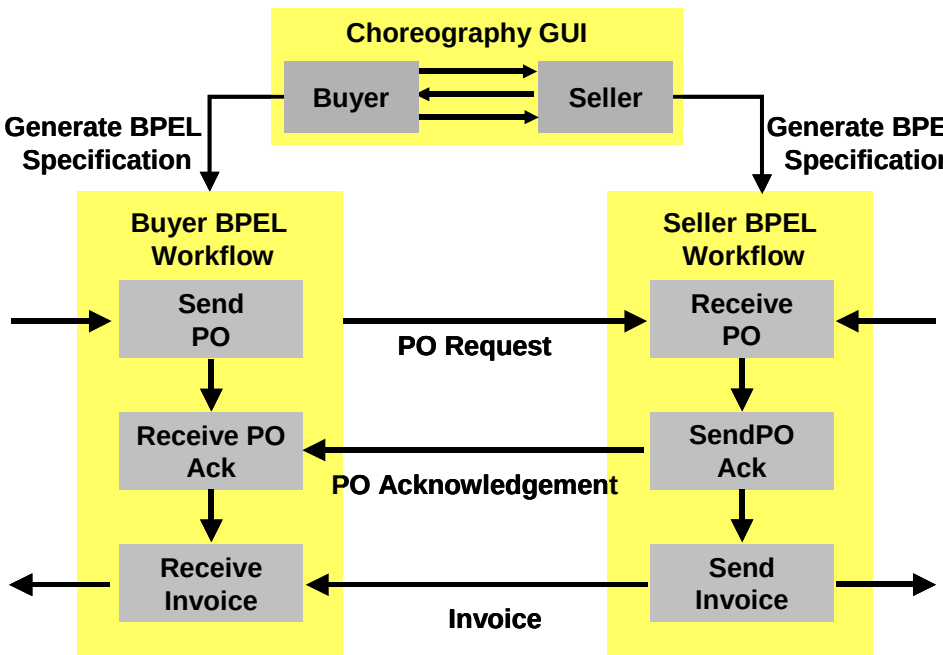


Figure 19 Combining choreography and orchestration.

A BPEL process is a flow-chart-like expression specifying process steps and entry-points into the process that is layered on top of WSDL, with WSDL defining the specific operations allowed and BPEL defining how the operations can be sequenced [Curbera03]. At the core of the BPEL process model lays the notion of peer-to-peer interaction between services described in WSDL. Both the process and its partners are modelled as WSDL services. BPEL uses WSDL to specify activities that should take place in a business process and describes the web services provided by the business process. A BPEL document leverages WSDL in the following three ways [Pelz03b].

Every BPEL process is exposed as a web service using WSDL. WSDL describes the public entry and exit points for the process.

1. WSDL data types are used within a BPEL process to describe the information that passes between requests.
2. WSDL might be used to reference external services required by a business process.

The role of BPEL is to define a new web service by composing a set of existing services through a process-integration type mechanism with control language constructs. The entry-points correspond to external clients invoking either input-only (request) or input-output (request-response) operations on the interface of the composite service. BPEL provides a mechanism for creating implementation and platform independent compositions of services weaved strictly from the abstract interfaces provided in the WSDL definitions. The definition of a BPEL business process also follows the WSDL convention of strict separation between the abstract service interface and service implementation. In particular, a BPEL process represents parties and interactions between these parties in terms of abstract WSDL interfaces (<portTypes> and <operation>s), while no references are made to the actual services (binding and address information) used by a process instance. Both the interacting process as well as its counterparts are modelled in the form of WSDL services. Actual implementations of the services themselves may be dynamically bound to the partners of a BPEL composition, without affecting the composition's definition. Business processes specified in BPEL are fully executable portable scripts that can be interpreted by business process engines in BPEL-conformant environments.

We distinguish four main sections in BPEL: the *message flow*, the *control flow*, the *data flow,* and the *process orchestration* sections.

1. The message flow section of BPEL is handled by primitive activities that include invoking an operation on some web service (<invoke>), waiting for a process operation to be invoked by some external client (<receive>), and generating the response of an input-output operation (<reply>). As a language for composing web services, BPEL processes interact by making invocations to other services and receiving invocations from clients. The prior is done using the <invoke> activity, and the latter using the <receive> and <reply> activities. BPEL calls these other services partners. A partner is a web service that the process invokes and/or any client that invokes the process. The Invoke element only support synchronous operations in the current version of BPEL.

2. The control flow section of BPEL is a hybrid model principally based on block structured and with the ability to define selective state transition control flow definitions for synchronization purposes. The control flow part of BPEL includes the ability to define an ordered sequence of activities (<sequence>), to have branching (<switch>), to define a loop (<while>), and to execute one of several alternative paths (<pick>). It main structured activity is the <flow> statement that allows defining sets of activities (including other flow activities) that are connected via <links>. Thus, a flow activity in BPEL may create a set of concurrent activities directly nested within it. It also enables expressing synchronisation dependencies between activities that are nested directly or indirectly within it. Within activities executing in parallel, execution order constraints can be specified. All structured activities can be recursively combined. The <link> construct is used to express these synchronisation dependencies providing among other things the potential for parallel execution of parts of the flow. The links are defined inside the flow and are used to connect exactly one source activity to exactly one target activity. A <link> may be associated with a transition condition, which is a Boolean expression using values in the different data variables in a process.

3. Business processes specify stateful interactions involving the exchange of messages between partners. The state of a business process includes the content of the messages that are exchanged as well as intermediate data used in business logic and in composing messages sent to partners. The data flow section of BPEL requires that information is passed between the different activities in an implicit way through the sharing of globally visible data <variables>. Data <variable>s specify the business context of a particular process. These are collections of WSDL messages, which represent data that is important for the correct execution of the business process, e.g., for routing decisions to be made or for constructing messages that need to be sent to partners. <Variables> provide the

means for holding message content that constitute altogether the state of a business process. The messages held are often those that have been received from partners or are to be sent to partners. <Variables> can also hold data that are needed for holding state related to the process and never exchanged with partners [Bloch03]. Variables may exchange specific data elements via the use of <assign> statements. The <assign> statement allows not only data manipulation, but also dynamic binding to different service implementations. Currently, there is no capability provided by BPEL to specify and execute transformations on the data.

4. The process orchestration section of BPEL uses service links to establish peer-to-peer partner relationships. A <partner> could be any service that the process invokes or any service that invokes the process. Each <partner> is mapped to a specific role that it fills within a process. A specific partner may play one role in one business process but completely different role in another process. Data <variables> are then used to manage the persistence of data across web service requests. Partners are connected to a process in a bilateral manner using a <partner-link type>. Partner-links define the shape of a relationship with a partner by specifying the WSDL <message> and <port-types> constructs used in the interactions in both directions. A <partner-linkType> is a third party declaration of a relationship between two or more services, comprising a set of roles, where each role indicates a list of <portType>s. A BPEL partner is then defined to play a role from a given <partner-link type>. Clients of the process are treated as partners, because a process may need to invoke operations on clients (for example, in asynchronous interactions), and because the service offered by the process may be used (wholly or in parts) by more than one client [Curbera03]. For example, a process representing a loan servicing system may offer a single web service, but only parts of it are accessible to the customer applying for the loan, while other parts are accessible for the customer service representative, and the entire service is accessible to the loan underwriters. The approach of using partners to model clients allows the process to indicate that certain clients may only invoke certain operations.

```
<process ...>
<!-- Web services the process interacts with -->
<partnerLinks> ... </partnerLinks>

<!– Data used by the process -->
<variables> ... </ variables >

<!– Supports asynchronous interactions -->
<correlationSets> ... </correlationSets>

<!– Activities that the process performs -->
   (activities)*

 <!–Exception handling: Alternate execution path to deal with faulty situations -->
<faultHandlers> ... </faultHandlers>

 <!–Code  that is executed when an action is “undone” -->
<compensationHandlers> ... </compensationHandlers>

<!–Handling of concurrent events -->
<eventHandlers> ... </eventHandlers>
</process>
```

Figure 20 Structure of BPEL process.

BPEL comprises basic and structured activities. Each activity is implemented as an interaction with a web service provided either by a particular provider or by one of its business partners. *Basic activities* are the simplest form of interaction with a service. They are not sequenced and comprise individual steps to interact with a service, manipulate the exchange data, or handle exceptions encountered during execution. One can think of a basic activity as a module that interacts with messages external to the process itself. For example, basic activities would handle receiving or replying to message requests as well as invoking

external services. The typical scenario is that there is a message received into the BPEL process. The process may then invoke a series of external services to gather additional data, and then respond to the requestor in some fashion. BPEL messages such as <receive>, <reply>, and <invoke> all represent basic activities for connecting services together. In contrast, *structured activities* (BPEL messages including <sequence>, <switch>, <while>, <pick>, and <flow>) manage the overall process flow, specifying the order in which activities execute. They might also specify that certain activities should run sequentially or in parallel. One may think of structured activities as the underlying programming logic for BPEL as they describe how a business process can be created by composing the basic activities it performs into structures. These structures involve the control patterns, data flow, fault handling, external event handling and coordination of message exchanges between process instances.

Abstract processes use all the concepts of BPEL but approach data handling in a way that reflects the level of abstraction required to identify protocol-relevant data embedded in messages [Bloch03]. Using the concept of message properties, properties can be viewed as *transparent data* relevant to public aspects as opposed to *opaque data* that internal/private functions use [Bloch03]. Transparent data affects the public business protocol in a direct way. Abstract processes handle only protocol-relevant data and use non-deterministic data values (opaque data) to hide private aspects of behaviour. Opaque data is related to back-end systems and affects the business protocol only by creating non-determinism because the manner it affects decisions is opaque. For instance, in the case of a purchase business protocol the seller may provide a service that receives a purchase order and responds with either acceptance or rejection of the transaction based on a number of criteria, such as availability of the goods, the creditworthiness of the buyer, etc. The decision processes is opaque, but the fact of the decision is reflected as behaviour alternatives exhibited by the external business protocol. In other words, the protocol acts as a "switch" within the behaviour of the seller's service, although the selection of the decision branch taken in the process flow is non-deterministic.

9.2.1 A simple example in BPEL

Figure 20 shows the structure of a typical BPEL process. We shall explain the constructs in this structure by means of a simplified version of the purchase order application shown in Figure 21. In this example we show a buyer placing a purchase order. In the process the buyer works through the seller to fulfil his/her request. The seller then communicates with credit service, billing service and inventory service providers to fulfil the client's wishes. Once an invoice is generated it is then sent back to the client. Several BPEL details are skipped in the interest of brevity.

The first part in a BPEL document is the definition of the process itself ("PurchaseOrder"). This is accomplished by means of the <process> element at the process root level (see Figure 20). The <process> element provides a name for the process and supplies references to the XML namespaces used. In this way the process places WSDL specific references in a BPEL process definition.

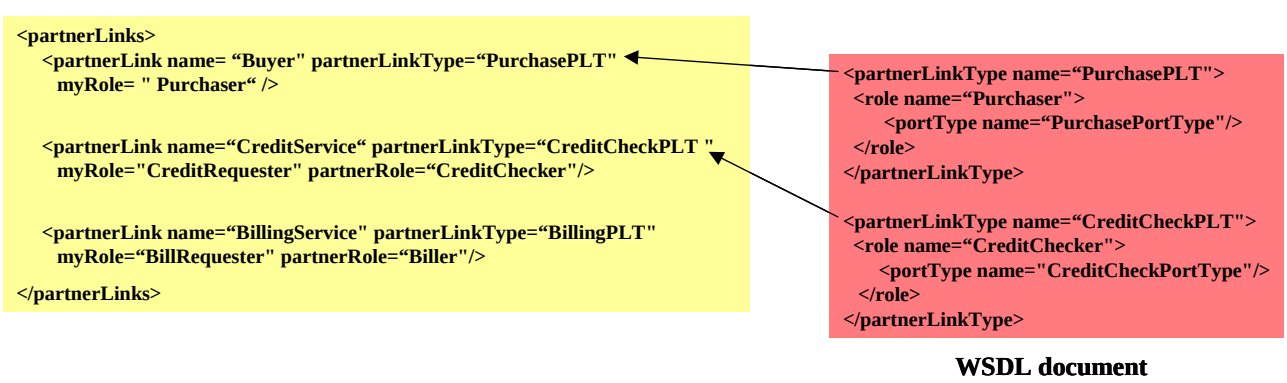


Figure 21 Definition of roles in a BPL document.

The <partnerLinks> section in Figure 21 defines the different parties that interact with the business process in the course of processing a buyer's order. In the purchase order process there are four interacting roles: that of the buyer, that of the credit service, the billing service and the inventory service (not shown in this figure). Each <partnerLinks> definition is characterised by a <partnerLinkType>. Further, it specifies which role of the underlying <partnerLinkType> the process itself accepts (myRole) and which role has to be

accepted by the partner (partnerRole). The "PurchaseOrder" process is defined from the perspective of the seller. When the buyer interacts with the seller, the buyer is the requester while the seller is the credit requester and bill requester (on behalf of the buyer). Conversely, the roles are reversed when the seller interacts with the "CreditService" or "BillingService" providers. Figure 21 shows this situation as well as the WSDL <portTypes> that are associated with each role via the <partnerLinkType> element. The <partnerLinkType> element defines the dependencies between the services and the WSDL <portType>s used. For instance, it is shown that the WSDL <portType> "Purchase" is associated with a request initiated with the buyer. The seller will also have a reference to the "CreditService" provider for requesting a credit service for the client. The port types themselves are defined elsewhere.

BPEL processes manage flow of data between partners represented by their service interfaces. The <variables> section of BPEL defines the data variables used by the process, providing their definitions in terms of WSDL message types and XML Schema elements. Variables allow processes to maintain state data and process history based on messages exchanged. For example, the business process stores a POMessage in a PO variable. Using <assign> and <copy>, data can be copied and manipulated between variables. In particular, <copy> supports XPath queries to sub-select data expressions. The PurchaseOrder part of POMessage is assigned to the PurchaseOrder part of a creditRequester. This message is stored in the creditRequest variable once received. The process (purchase order) passes on the credit service (inventory service) part of the message to the credit service (inventory service) provider. This message uses the part-purchase-order PO variable. The credit service (inventory service) providers then process the requests. Each variable shown in Figure 22 is followed by a reference to a specific WSDL message type.

```
<variables>
    <variable name="PO" messageType="POMessage"/>
    <variable name="Inv" messageType="InvMessage"/>
    <variable name="OrderAcceptance"  messageType="OrderAcceptMessage"/>
 </variables>

<assign>
 <copy>
  <from variable="PO" part="PurchaseOrder"/>
  <to variable="creditRequest" part="PurchaseOrder"/>
 </copy>
</assign>
```

Figure 22 BPEL data variables for the PurchaseOrder process.

A fundamental part of a BPEL specification is the definition of the activities and the sequence of steps required to make up a given process. This is where basic and structured activities are employed. A business process provides services to its partners through receive activities and corresponding reply activities. The <receive> construct allows the business process to do a blocking wait for a matching message to arrive. A receive activity specifies the partner link it expects to receive from, and the port type and operation that it expects the partner to invoke. In addition, it may specify a variable that is to be used to receive the message data received. The only way to instantiate a business process in BPEL is to annotate a receive activity with the createInstance attribute set to "yes". The <reply> construct allows the business process to send a message in reply to a message that was received through a <receive> activity. The combination of a <receive> and a <reply> forms a request-response operation on the WSDL <portType> for the process. Such responses are only meaningful for synchronous interactions. A <reply> activity may specify a variable that contains the message data to be sent in reply. To invoke a web service a process uses the <invoke> construct. This construct allows the business process to invoke a one-way or request/response operation on a <portType> offered by a partner. When invoking partners one needs to determine the partner to invoke as well as which operation to involve.

A key part of the BPEL4WS document is the definition of the basic sequence of steps required to handle the request. This is where basic and structured activities come into play. The process flow in Figure 23 is shown to comprise of an initial request from a customer, followed by an invocation of credit service and billing service providers in parallel, and ultimately a response to the client from the seller sending a concrete purchase order.

To model this, the <sequence> tag is used for running components sequentially, the <flow> tag is used for parallel execution, and the <receive>, <reply>, and <invoke> tags handle the basic activities required to interact with the services.

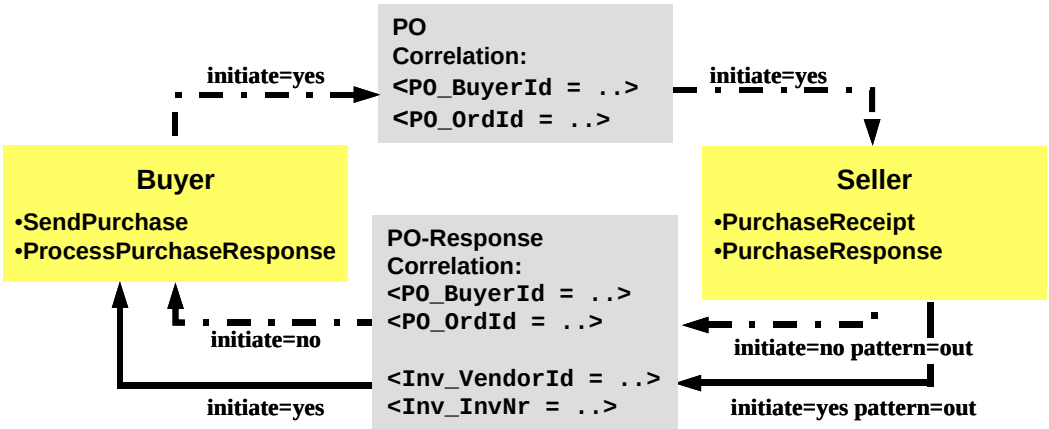
```
<sequence>
 <receive name="receive_PurchaseOrder"
         partnerLink="Buyer" portType="PurchaseOrderPortType"
         operation="SendPurchaseOrder" variable="PO"
         createInstance="yes" />
 <flow>
  <invoke partnerLink="CreditService" portType="CreditCheckPortType"
        operation="CheckCredit" inputVariable="CreditRequest"
        outputVariable="CreditResponse" >
  <invoke/>

  <invoke partnerLink="BillingService" portType="BillingPortType"
        operation="BillClient" inputVariable="BillRequest"
        outputVariable="BillResponse" >
  <invoke/>
 </flow>
 ...
 <reply name="respond_PurchaseOrder"
      partnerLink="Buyer" portType="PurchaseOrderPortType"
      operation="SendPurchaseOrder" variable="Inv"/>
</sequence>
```

Figure 23 BPEL process flow for PurchaseOrder process.

The first step in the process flow is the initial buyer request. The "createInstance" flag is used to identify the start of a new process instance. Once this request is received, there is a parallel set of activities that are executed using the <flow> tag. Here, a credit service is contacted in order to receive a credit check for the buyer, while a billing service is contacted to bill the buyer. Each references a specific WSDL operation (e.g., CheckCredit), and uses the available containers for input and output. Upon receiving the responses back from these suppliers, the seller would construct a message back to the buyer. This could involve use of the XPath language to take the various containers received from the service providers and building a final proposal to the buyer.

A correlation mechanism exists with BPEL processes to connect service instances with associated messages. For example, a buyer identification number might be used to identify an individual buyer in a long-running multiparty business process relating to a purchase order. In correlation, the property name, e.g., buyer-id, purchase-order-id, invoice-number, vendor-id, etc, must have global significance to be of any use. Figure 24 shows a graphical representation as well as a BPEL specification of a correlation. In particular, this figure shows that there exists a unique purchase order identifier for each purchase order forwarded by a buyer and received by a seller and a unique number for each corresponding invoice created by the seller. A buyer may start a correlated exchange with a seller by sending a purchase order (PO) and using a PO id in the purchase order document as the correlation token. This PO id is used in the PO acknowledgement by the seller. The seller may later send an invoice document that carries both the PO id, to correlate it with the original purchase order, and also an invoice number. In this way future payment related messages may carry only the invoice number as the correlation token. The invoice message thus carries two separate correlation tokens and participates in two overlapping correlated exchanges. The scope of correlation is not, in general, the entire interaction specified by a service, but may span a part of the service behaviour.

The BPEL <faultHandlers> section contains structures defining the activities that must be performed in response to faults resulting from the invocation of services. In BPEL, all faults, whether internal or resulting from a service invocation, are identified by a qualified name. In particular, each WSDL fault is identified in BPEL by a qualified name formed by the target namespace of the WSDL document in which the relevant <portType> and fault are defined, and the name of the fault [Bloch03]. Certain operations can return faults, as defined in their WSDL definitions. For example, Figure 25 illustrates the case where there is an error when a buyer submits a purchase order. In this situation the seller may use a fault handler ("OrderNotComplete") employing a <reply> element to return a fault to the buyer.




```
<correlationSets>
   <correlationSet name="POCorr" properties="cor:buyerId cor:ordId"/>
   <correlationSet name="InvoiceCorr" properties="cor:vendorId cor:invNr"/>
</correlationSets> ...

<receive partnerLink="Buyer" portType="PurchaseOrderPortType"
     operation="PurchaseReceipt" variable="PO">
  <correlations>
     <correlation set="POCorr" initiate="yes">
  </correlations>
</receive> ...

<invoke partnerLink="Buyer" portType="BuyerPortType"
    operation="ProcessPurchaseResponse" inputVariable="POResponse">
 <correlations>
    <correlation set="POCorr" initiate="no" pattern="out">
    <correlation set="InvoiceCorr" initiate="yes" pattern="out">
 </correlations>
</invoke> ...
```

Figure 24 Correlation properties and sets for the PurchaseOrder process.

In many cases error handling in business processes relies heavily on the well-known concept of *compensation*, that is, application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned. BPEL provides a compensation protocol that has the ability to define fault handling and compensation in an application-specific manner, resulting in long-running (business) transactions [Bloch03]. To set up a transactional context in BPEL, the <scope> element is used. This element groups related activities together. To exemplify the concept of compensation conisder the case where a purchase is made and this purchase needs to be cancelled. In this case a <compensationHandler> can be issued to invoke a cancellation operation at the same port of the same WSDL <partnerLink>, using the response to the purchase order as its input.

```
<faultHandlers>
 <catch faultName="OrderNotComplete"
      faultVariable="POFault">
  <reply partnerLink="Buyer"
       portType="PurchaseOrderPT"
       operation="SendPurchaseOrder"
       variable="POFault"
       faultName="OrderNotComplete "/>
 </catch>
</faultHandlers>
```

Figure 25 A simple fault handler for the PurchaseOrder process.

9.3 The choreography definition language

A large class of e-Business activities and applications requires the ability to perform long-lived, peer-to-peer message interchanges between the participating services, i.e. choreographies, within or across the trusted domains of an organization. The primary goals of a choreography definition are:

- to verify at run time that all message interchanges are proceeding according to plan, and
- to guaranty that changes in the implementations of services are still in compliance with the message interchange definition.

As such a *choreography* is not executed but rather monitored and validated (or invalidated).

The first draft of the WS-CDL has been released in April 2004 [Kavantzas04]. In the specification, the message interchanges take place in a jointly agreed set of ordering and constraint rules. Choreography definitions can involve two (binary) or more (multiparty) participants. WS-CDL, describes a global view of the message interchange without taking any participants point of view, unlike the BPEL global model which takes the point of view of one participant. This approach is a lot more scalable when the number of participants increases. However, and like BPEL, WS-CDL is an infrastructure specification which does not contain any business semantics (e.g. resources, commitments, agreements …).

A choreography definition is always defined abstractly between *roles* which are later bound to *participants*. Roles are related to each other via *relationships.* A relationship is always between exactly two roles. A participant may implement any number of non-opposite roles in the choreography. A distributor may implement the buyer-to-manufacturer and seller-to-customer roles, which is yet different from the seller-to-distributor role.

Choreographies are composed of activities. The main activity is called an *interaction* which results in exchange of messages between participants and possible synchronization of their states and the actual values of the exchanged information. Other activities are used to create meaningful combinations of interactions. For instance ordering activities (*sequence, parallel, choice*) or a *perform* activity which composes another choreography in the parent choreography. Choreography neutral activities are also defined: *assign,* which associates a given value from a variable to another one, a *no-action* activity and a *workunit* activity. Interactions may also be organized in *work units* which may be guarded by a series of pre-conditions, and generate a series of post-conditions as they complete.

Choreography definitions may be data driven, i.e. the data contained in the messages impacts the ordering of interactions. Data is modelled as *variables* which may either be associated to message content, a *channel* or the *state* of roles involved in the choreography. *Tokens* are aliases which may represent parts of a variable. Both token and variables have types that specify their data structure. The type is either an XML schema or the element definition of an XML schema. State variables are bound to roles, and describe any relevant state of a given role. State variables may have the same name at different roles (e.g. OrderState), but each role may have different values (e.g. OrderSentState at the buyer, and OrderReceivedState at the seller). It would be preferable though to align state definitions, because semantically, Buyer:OrderSent is equivalent to Seller:OrderReceived, assuming that the state "OrderSent" is really reached when the message has reliably been delivered to the Seller.

The notion of role state is relatively new in the web service architecture and adds a new layer of semantics. The web services architecture was based on typed messages as defined in WSDL and used for instance in BPEL or WS-CDL. It was introduced to be able to achieve "state alignment" which is a major issue in web services interactions. The question has often been neglected reducing it to a reliable messaging problem. Reliable messaging is actually just one layer in the path to achieve state alignment: this is not because an application receives a message that it agrees that its content is valid or that the application is able to process this message. If a message is received and this message cannot be understood or processed (it is encrypted), a state misalignment may occur, especially if there is no protocol in place to ensure that the receiver of the message can signal the sender that a error occurred. Without such protocol in place, the sender may consider that since the message was received, the receiver must be in the same state. WS-CDL offers an *isAligned()* function which should be computed and return true when the state of two roles is aligned. Unfortunately, at the current stage, the WS-CDL does not mandate a state alignment protocol like ebXML BPSS offers. It also groups state alignment guards at the work unit level and not at the

operation level. While, it is very useful to specify series of interactions performing a given activity, it may also be as important to guaranty state alignment within the work unit itself, otherwise it could force users of WS-CDL to define work units as big an interaction and defeat the purpose of a very useful construct.

Interactions specify the unit of message exchange between roles. An interaction corresponds to the invocation of a web service operation on a role. Consequently, an interaction is defined as a request with zero or more responses. An interaction definition is bound to an operation definition (which requires at least an abstract WSDL definition) and a channel variable. The channel variable will be initialized a run-time, most likely at the start of the choreography, though, like pi-calculus –its theoretical foundation- WS-CDL allows for dynamic passing of channel values.

WS-CDL represents an important new layer of the web services stack. As it is early in the W3C process, it is difficult to picture exactly what the final recommendation will be and whether or not the OASIS BEPL and W3C WS-CDL working group will come to an agreement on how the two standards would work together.

9.4 Web service transactions

One key requirement in making cross-enterprise business process automation happen is the ability to describe the collaboration aspects of the business processes, such as commitments and exchange of monetary resources, in a standard form that can be consumed by tools for business process implementation and monitoring. Enterprise workflow and business process management systems today support the definition, execution and monitoring of long running processes that coordinate the activities of multiple business applications. However, the loosely coupled, distributed nature of the Web prevents a central workflow authority (or a centralized implementation of middleware technology) from exhaustively and fully coordinating and monitoring the activities of the enterprise applications that expose the web services participating in message exchanges [Papazoglou03b].

Business collaboration requires transactional support in order to guarantee consistent and reliable execution. Database transactions are a well-known technique for guaranteeing consistency in the presence of failures. A classical transaction is a unit of work that either completely succeeds, or fails with all partially completed work being undone. Such classical transactions have ACID properties:

- Atomicity: executes completely or not at all.
- Consistency: preserves the internal consistency of an underlying data structure.
- Isolation: runs as if it were running alone with no other transactions running.
- Durability: the transaction's results will not be lost in the event of a failure.

The ACID properties of atomic transactions ensure that even in complex business applications consistency of state is preserved, despite concurrent accesses and failures. This is an extremely useful fault-tolerance technique, especially when multiple, possibly remote, resources are involved. However, traditional transactions depend upon tightly coupled protocols, and thus are often not well suited to more loosely-coupled web services based applications, although they are likely to be used in some of the constituent technologies. Strict ACIDity and isolation, in particular, is not appropriate to a loosely coupled world of autonomous trading partners, where security and inventory control issues prevent hard locking of local resources that is impractical in the business world.

A web service environment requires the same coordination (see also section 9) behaviour provided by a traditional transaction mechanism to control the operations and outcome of an application. However, it also requires the capability to handle the coordination of processing outcomes or results from multiple services, in a more flexible manner. This requires more relaxed forms of transactions -- those that do not strictly have to abide to the ACID properties --such as collaborations, workflow, real-time processing, etc. Additionally, there is a need to group Web services into applications that require some form of correlation, but do not necessarily require transactional behaviour. In the loosely coupled environment represented by web services, long running applications will require support for coordination, recovery and compensation, because machines may fail, processes may be cancelled, or services may be moved or withdrawn. Web service transactions also must span multiple transaction models and protocols native to the underlying infrastructure onto which the web services are mapped.

The concept of a *business transaction* is central to web service applications as it defines a shared view of messages exchanged between web services from multiple organisations for the purpose of completing a business process [Papazoglou03b]. A business transaction is a consistent change in the state of the business that is driven by a well-defined business function. At the end of a business transaction the state of both parties must be aligned, i.e. they must have the same understanding of the outcome of the message interchange during the business transaction. Usually, a business process is composed of several business transactions. In a web service environment business transactions essentially signify transactional web service interactions between organisations in order to accomplish some well-defined shared business objective. Business transactions are long-running activities that can take minutes, hours, or even more to complete. Business transactions, just like database transactions, either execute to completion (succeed) or fail as a unit. A business transaction in its simplest form could represent an order of some goods from some company. The completion of an order results in a consistent change in the state of the affected business: the back-end order database is updated and a document copy of the purchase order is filed. More complex business transactions may involve activities such as payment processing, shipping and tracking, coordinating and managing marketing strategies, determining new product offerings, granting/extending credit, managing market risk, product engineering and so on. Such complex business transactions are usually driven by interdependent workflows, which must interlock at points to achieve a mutually desired outcome. This synchronization is one part of a wider business coordination protocol that defines the public, agreed interactions between interacting business parties.

The problem of coordinating web services is tackled by a trio of standards that have been recently proposed to handle this next step in the evolution of Web services technology. The standards that support business process orchestration while providing web service transactional functionality are: Business Process Execution Language for Web Services (see section-9.2), WS-Coordination [Cabrera02a] and WS-Transaction [Cabrera02a]. As already explained, BPEL is a workflow-like definition language that describes sophisticated business processes that can orchestrate web services. WS-Coordination and WS-Transaction complement BPEL4WS to provide mechanisms for defining specific standard protocols for use by transaction processing systems, workflow systems, or other applications that wish to coordinate multiple web services. WS-Coordination provides a framework for coordinating the actions of distributed applications via context sharing. WS-Transaction provides standards for atomic transactions as well as long-running transactions. These three specifications work in tandem to address the business workflow issues implicated in connecting and executing a number of web services that may run on disparate platforms across organisations involved in e-business scenarios.

The WS-Coordination specification describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that need to reach consistent agreement on the outcome of distributed transactions. Typically, coordination is the act of one entity (known as the coordinator) disseminating information to a number of participants for some domain-specific reason, e.g., reaching consensus on a decision like a distributed transaction protocol, or simply to guarantee that all participants obtain a specific message, as occurs in a reliable multicast environment [Webber03a]. When parties are being coordinated, information known as the coordination context is propagated to tie together operations that are logically part of the same activity. The WS-CAF specification supports self-coordinated message interchanges that do not require necessarily a third party coordinator. However, a transaction always requires a coordinator.

WS-Coordination provides developers with a way to manage the operations related to a business activity. A business process may involve a number of web services working together to provide a common solution. Each service needs to be able to coordinate its activities with those of the other services for the process to succeed. WS-Coordination sequences operations in a process that spans interoperable web services to reach an agreement on the overall outcome of the business process. The specification supplies standard mechanisms to create an activity via an activation service and register via a registration service with transaction protocols that coordinate the execution of distributed operations in a web services environment. WS-Coordination provides a definition of the structure of coordination context and the requirements for propagating this context between cooperating services via its coordination context.

WS-Transaction provides transactional coordination mechanisms for Web services. An important aspect of WS-Transaction that differentiates it from traditional transaction protocols is that it does not assume a

synchronous request/response model. This derives from the fact that WS-Transaction is layered upon the WS-Coordination protocol whose own communication patterns are asynchronous. WS-Coordination provides a generic framework for specific coordination protocols, like WS-Transaction, to be plugged in. The WS-T specification leverages WS-Coordination by extending it to define specific protocols for transaction processing. WS-Coordination provides only context management – it allows contexts to be created and activities to be registered with those contexts. WS-Transaction leverages the context management framework provided by WS-Coordination in two ways [Little03a]. Firstly, it extends the WS-Coordination context to create a transaction context. Second, it augments the activation and registration services with a number of additional services, e.g., completion, completion withAck, two phase-commit, outcome notification, etc) and two protocol message sets (one for each of the transaction types supported in WS-Transaction) to build a full-fledged transaction coordinator on top the WS-Coordination protocol infrastructure.

WS-transaction defines two transaction types: *atomic transaction* and *business activity* while WS-CAF provides three types of transaction within an extensible framework: atomic, activity and business process. Atomic transactions are suggested for transactions that are short-lived atomic units of work within a trust domain, while business activities are suggested for transactions that are long-lived units of work comprising activities of potentially different trust domains. Atomic transactions compare to the traditional distributed database transaction model (short-lived atomic transactions). The coordination type correspondingly comprises protocols common to atomic transactions, where resource participants register for the two-phase commit protocol. The business activity coordination type supports transactional coordination of potentially long-lived activities. These differ from atomic transactions in that they take much longer time to complete and do not require resources to be held. To minimise the latency of access by other potential users of the resources used by an activity, the results of interim operations need to be realised prior to completing the overall activity. They also require that business logic be applied to handle exceptions. Participants are viewed as business tasks (scopes) that are children of the business activity for which they register. Participants may decide to leave a business activity (for example, to delegate processing to other services), or, a participant may declare its outcome before being solicited to do so. Developers can use either or both of these coordination types when building applications that require consistent agreement on the outcome of distributed activities. The WS-Transaction specification monitors the success of specific, coordinated activities in a business process. Just like in WS-CAF, WS-Transaction uses the structure that WS-Coordination provides to make sure the participating web services end the business process with a shared understanding of its outcome. For example, a purchase order process contains various activities that have to complete successfully but might run simultaneously (at least to some extend), such as credit check, inventory control, billing and shipment. The combination of WS-Transaction and WS-Coordination makes sure that these tasks succeed or fail as a unit.

Finally, the BPEL specification suggests WS-Transaction as the protocol of choice for coordinating distributed transactions across workflow instances. Thus, when a scope containing invocations on a partner's Web services is compensated, the underlying BPEL engine should ensure that the appropriate WS-Transaction messages are sent to the transaction coordinator so that any partner's systems can be informed of the need to compensate the invoked activities.

In a related activity, the Committee Specification of the Organization for the Advancement of Structured Information Standards (OASIS) Business Transaction Protocol (BTP) Working Group is an industry initiative that is working on creating a vendor neutral standard for business transaction interoperability [Little03b]. BTP is designed to support transactional coordination of participants of services offered by multiple autonomous organisations as well as within a single organization. The BTP specification defines communications protocol bindings that target the web services arena, while preserving the capacity to carry BTP messages over other communication protocols. Protocol message structure and content constraints are schematised in XML, and message content is encoded in XML instances. As of April 2004, the two OASIS technical committees, WS-CAF and BTP, are seeking alignment and integration.

10 Web services security

As web services using the insecure Internet for mission-critical transactions with the possibility of dynamic, short-term relationships, security is a major concern. An additional concern is that web service applications expose their business processes and internal workflows. This calls for securing them against a wide range of

attacks, both internal and external. With Web services, more of the application internals are exposed to the outside world.

Web services can be accessed by sending SOAP messages to service endpoints identified by URIs, requesting specific actions, and receiving SOAP message. Within this context, the broad objective of securing Web services breaks into providing facilities for securing the integrity and confidentiality of the messages and for ensuring that the service acts only on requests in messages that express the claims required by policies.

Traditionally, the Secure Socket Layer (SSL) along with the de facto Transport Layer Security (TLS) and the Internet Protocol Security (IPSec) are some of the common ways of securing content. SSL/TLS offers several security features including authentication, data integrity and data confidentiality. SSL/TLS enables point-to-point (server-to-server) secure sessions. IPSec is another network layer standard for transport security that may become important for Web services. Like SSL/TLS, IPSec also provides secure sessions with host authentication, data integrity and data confidentiality. However, these are point-to-point technologies. They create a secure tunnel through which data can pass [Mysore03]. For instance, SSL is a good solution for server-to-server security but it cannot adequately address the scenario where a SOAP request is routed via more than one server. In this case the recipient has to request credentials of the sender and the scalability of the system is compromised. The session-based authentication mechanisms used by SSL have no standard way to transfer credentials to service providers via SOAP messages. Web services require much more granularity. They need to maintain secure context and control it according to their security policies.

To address the security issues facing organizations as they adopt web services technology, it is essential to apply the principles of application security. Application security contains five basic requirements, expressed in terms of the messages exchanged between parties [Pilz03]. Such messages include any kind of communication between the sender (party who wishes to access an application) and the recipient (the application itself). The six requirements for application level security can be summarised as follows:

1. Authentication: Verifies that the identity of entities is provided by the use of public key certificates and digital signature envelopes. Authentication in the Web services environment is performed very well by public key cryptographic systems incorporated into Public Key Infrastructure (PKI). The primary goal of authentication in a PKI is to support the remote and unambiguous authentication between entities unknown to each other, using public key certificates and trust hierarchies.

2. Authorisation: Verifies that the identity has the necessary permissions to obtain the requested resource or act on something before providing access to it. Normally, authorization is preceded by authentication.

3. Non-Repudiation: Constantly monitors and record service requests such that audits may be performed at a later time to positively identify the party which performed any given request.

4. Message integrity and confidentiality: Integrity ensures that data cannot be corrupted or modified, and transactions cannot be altered. Message (data) integrity comprises two requirements: first, the data received must be the same as the data sent. In other words, data integrity systems must be able to guarantee that a message did not change in transit, either by mistake or on purpose. The second requirement for message integrity is that at any time in the future, it is possible to prove whether different copies of the same document are in fact identical. Confidentiality means that an unauthorized person cannot view or interfere with a communication between two parties.

5. Operational defence. The system must be able to detect and guard against attacks by illegitimate messages, including XML Denial of Service (XDoS) and XML viruses, and must be operationally scalable with existing personnel and infrastructure.

6. Standards for securing web services are heavily Public Key Infrastructure (PKI) oriented. PKI relies upon public key cryptography and uses a secret private key that is kept from unauthorized users and a public key that is handed to trusted partners. A PKI is a foundation upon which other applications and network security components are built. The specific security functions for which a PKI can provide a foundation are confidentiality, integrity, non-repudiation, and authentication. Public key infrastructure (PKI) plays an essential role in web services security, enabling end users

and web services alike to establish trusted digital identities which, in turn, facilitate trusted communications and transactions.

When securing web services, enterprises should focus on authentication, authorization, non-repudiation, confidentiality, and data integrity, as well as threat detection and defence. The requirement for securing web services concentrates on mutually authenticating all trading partners and communicating infrastructure, i.e., users and servers. Authorization is particularly important because of the need for tiered security administration in service-oriented environments. This situation is even more complex when multiple, heterogeneous systems are involved, either within an enterprise or across two or more companies. Every company will likely have its own security policies, in addition to its own authorization technology. Therefore, the ability to provide and administer authorization across multiple systems is an important problem that a web services specification known as WS-Security is intended to address [Atkinson02].

WS-Security specifies an abstraction layer on top of any company's particular application security technology (PKI, Kerberos, etc.) that allows such dissimilar infrastructures to participate in a common trust relationship. WS-Security provides a set of SOAP extensions that can be used to implement message integrity, confidentiality and authentication. It is designed to provide support for multiple security tokens, trust domains, signature formats and encryption technologies. No specific type of security token is required by WS-Security. It is designed to be extensible (e.g. support multiple security token formats). For example, a requester might provide proof of identity and proof that they have a particular business certification. Message integrity is provided by leveraging XML Signature in conjunction with security tokens (which may contain or imply key data) to ensure that messages are transmitted without modifications. XML Signature defines rules for digitally signing XML documents and processing the signatures. The integrity mechanisms are designed to support multiple signatures, potentially by multiple actors, and to be extensible to support additional signature formats. The signatures may reference (i.e. point to) a security token. Similarly, message confidentiality is provided by leveraging XML Encryption in conjunction with security tokens to keep portions of SOAP messages confidential. XML Encryption specifies the syntax for encrypting XML documents so that only authorized Web services can access the contents. The encryption mechanisms are designed to support additional encryption technologies, processes, and operations by multiple actors. The encryption may also reference a security token.

While current technologies enable an e-Business applicatin to authenticate users and manage user access privileges, it takes considerable effort and cost to extend these capabilities across an enterprise or share them among trading partners. Security Assertions Markup Language (SAML) [SAML03] addresses this challenge. SAML is an XML-based framework that enables web services to readily exchange information relating to authentication and authorisation. SAML defines a protocol by which clients can request assertions from SAML authorities and receive responses from them (exchange of security information). This protocol, consisting of XML-based request/response message formats, can be bound to many different underlying communications and transport protocols. The security information is expressed in the form of assertions about subjects. A subject is an entity that has an identity in some security domain. Assertions can convey information about authentication acts performed by subjects, attributes of subjects, and authorization decisions about whether subjects are allowed to access certain resources.

Another important requirement for securing web services is providing end-to-end message integrity and confidentiality. Public key certificates and digital signature envelopes are good examples of information that must have an assurance of message integrity. Integrity can be provided by the use of either public (asymmetric), or secret (symmetric) cryptography. PKI can use encryption to protect the confidentiality of data both in transit and in storage. Virtual Private Networks (VPNs) and Secure Sockets Layer (SSL) can protect the confidentiality of messages between two endpoints, but neither secures the data in storage after it has been received, or across intermediaries, because both SSL and VPNs are point-to-point techniques. Therefore, an SSL-encrypted message, for example, would have to be unencrypted at an intermediary, which opens a security hole.

Preserving loose coupling while ensuring confidentiality data integrity, in a service-oriented environment is particularly challenging, because of the constraints such requirements put on both the web service providers and requesters. Specifications like WS-Security and SAML are intended to address the issue of

preserving loose coupling by providing standard ways for both ends of a secure web services message to participate in the various forms of application security.

The two specifications WS-Security and SAML, which are closely related, ensure the integrity and confidentiality of XML documents. They differ from existing capabilities in that they provide mechanisms for handling whole or partial documents, making it possible to address varying requirements for access authority, confidentiality and data integrity within one document.

11 Quality of Service requirements

The Service Level Agreement (SLA) is an important and widely used instrument in the maintenance of service provision relationships. Where contracts are clearly defined and closely monitored in order to guarantee adherence of all involved parties to the terms agreed upon, then participants are protected by the SLA. Both service providers and clients alike need to utilise SLAs in order to work well together.

An SLA is a contract between a service provider and a client that specifies, usually in measurable terms, what services the service provider will furnish. Understanding business requirements, expected usage patterns and system capabilities can go a long way toward ensuring successful deployments. To better understand requirements entering into a web services SLA, one needs to address several important concerns. These include: the levels of availability that are needed for a web service; whether the business can tolerate web services downtime and how much; whether there is adequate redundancy built in so that services can be offered in the event of a system or network failure; the transaction volumes expected of web services; whether underlying systems have been designed and tested to meet these peak load requirements and, finally, how important are request/response times.

One important function that an SLA should address is the Quality of Service (QoS) at the source. This refers to the level of service that a particular service provides [Mani02]. QoS is defined by important functional and non-functional service quality attributes, such as service metering and cost, performance metrics (e.g., response time), security attributes, (transactional) integrity, reliability, scalability, and availability. Service clients (end user organisations that use some service) and service aggregators (organisations that consolidate multiple services into a new, single service offering) utilize service descriptions to achieve their objectives.

The major requirements for supporting QoS in Web services are summarised in what follows and are partly based on [Mani02]:

1. Availability: Availability is the absence of service downtimes. Availability represents the probability that a service is available. Larger values represent that the service is always ready to use while smaller values indicate unpredictability of whether the service will be available at a particular time. Also associated with availability is time-to-repair (TTR). TTR represents the time it takes to repair a service that has failed. Ideally smaller values of TTR are desirable.

2. Accessibility: Accessibility represents the degree with which a web service request is served. It may be expressed as a probability measure denoting the success rate or chance of a successful service instantiation at a point in time. A high degree of accessibility means that a service is available for a large number of clients and that clients can use the service relatively easily.

3. Conformance to standards: describes the compliance of a web service with standards. Strict adherence to correct versions of standards (for example, WSDL version 2.0) by service providers is necessary for proper invocation of Web services by service requesters. In addition, service providers must stick to the standards outlined in the SLAs.

4. Integrity: describes the degree with which a web service performs its tasks according to its WSDL description as well as conformance with service-level agreement (SLA). A higher degree of integrity means that the functionality of a service is closer to its WSDL description or SLA.

5. Performance: Performance is measured in terms of two factors: throughput and latency. Throughput represents the number of web service requests served at a given time period. Latency represents the length of time between sending a request and receiving the response. Higher throughput and lower latency values represent good performance of a web service. When measuring the transaction/request volumes handled by a web service it is important to consider whether these come in a steady flow or burst around particular events like the open or close of the business day or seasonal rushes

6. Reliability: Reliability represents the ability of a service to function correctly and consistently and provide the same service quality despite system or network failures. The reliability of a web service is usually expressed in terms of number of transactional failures per month or year.

7. Scalability: Scalability refers to the ability to consistently serve the requests despite variations in the volume of requests. High accessibility of web services can be achieved by building highly scalable systems.

8. Security: Security involves aspects such as authentication, authorisation, message integrity and confidentiality (see section-10). Security has added importance because web service invocation occurs over the Internet. The amount of security that a particular web service requires is described in its accompanying SLA, and service providers must maintain this level of security.

9. Transactionality: There are several cases where web services require transactional behaviour and context propagation. The fact that a particular web service requires transactional behaviour is described in its accompanying SLA, and service providers must maintain this property.

A standard policy framework makes it possible for developers to express the policies of services in a machine-readable way and for web services to understand policies and enforce them at runtime. For example, a developer could write a policy stating that a given service requires digital signatures and encryption. Service clients could use the policy information to ascertain whether they can use the service.

The Web Services Framework (WS-Policy) [WS-Policy03] fills this gap by providing building blocks that may be used in conjunction with other web service and application-specific protocols to accommodate a wide variety of policy exchange models. The Web Services Policy Framework provides a general purpose model and corresponding syntax to describe and communicate the policies of a web service. WS-Policy defines a base set of constructs that can be used and extended by other web services specifications to describe a broad range of service requirements, preferences, and capabilities. WS-Policy defines a policy to be a collection of one or more policy assertions. A policy assertion represents an individual preference, requirement, capability, or other general characteristic.

WS-Policy provides a flexible and extensible grammar for expressing policies in an XML format referred to as a *policy expression*. A policy expression is bound to a *policy subject*, which is the resource it describes, e.g., a web service endpoint. The mechanism for associating a policy expression with one or more policy subjects is referred to as a *policy attachment*. The WS-Policy specification defines the general model and syntax for policy expressions and policy assertions but does not specify how policies are located or attached to a web service. To address this issue a Web Services Policy Attachment (WS-PolicyAttachment) specification is used to define how to attach policy expressions to XML elements, WSDL definitions, and UDDI entries.

12 Web services interoperability

Web service implementations details of specifications and best practises are becoming slowly established. Given the potential to have many necessary interrelated specifications at various versions and schedules of development, it becomes a very difficult task to determine which products support which levels of the specifications. Consequently, there are versions of products that implement the specifications in ways that they are different enough to prevent their implementations from being fully interoperable. Individual enterprises are forced to provide individual interpretations of how their specifications are to be used. Thus, implementing standards alone cannot ensure interoperability. The fundamental goal of interoperability in

web services is to blur the lines between the various development environments used to implement services. Web services interoperability addresses the problem of ambiguity among the interpretation of standards that have been agreed upon; differences among specifications that have yet to gain widespread adoption and insufficient understanding of the interaction among the various specifications [WSI03].

Web service interoperability concerns are addressed by the Web Services interoperability Organization (WS-I) an industry consortium focused on promoting web service interoperability across platforms, operating systems, and programming languages. WSI was formed specifically for the creation, promotion, or support of Generic Protocols for Interoperable exchange of messages between services. Generic Protocols are protocols that are independent of any specific action indicated by the message beyond actions necessary for the secure, reliable, or efficient delivery of messages.

Profiles make it easier to discuss web services interoperability at a level of granularity that makes sense for developers, users, and executives making investment decisions about web services and web services products. WS-I focuses on compatibility at both the individual specification and at the Profile level. Profiles contain a list of named and versioned web services specifications together with a set of implementation and interoperability guidelines recommending how the specifications should be used to develop interoperable Web services. WS-I will develop a core collection of profiles that support interoperability for general purpose web services functionality.

Basic Profile 1.0 includes implementation guidelines on using core Web services specifications together to develop interoperable web services. Those specifications include SOAP 1.1, WSDL 1.1, UDDI 2.0, XML 1.0, and XML Schema. The availability of Basic Profile 1.0 sets the stage for unified web services support in technologies such as the next major version of the enterprise Java specification, J2EE 1.4, and the upcoming upgrade of the IBM WebSphere Studio development environment. Version 1.0 of the profile is intended to provide a common framework for implementing interoperable solutions while giving buyers a common reference point for purchasing decisions.

Among the key deliverables of WS-I are testing tools, which developers can use to test conformance of their Web services with the test assertions that represent the interoperability guidelines of established WS-I Profiles. The process used to develop these Profiles, interoperability guidelines, test assertions, and testing tools generates other related resources useful to developers. The tools developed monitor the interactions with a web service, record those interactions, and analyse them to detect implementation errors.

13 Grid services

Grid computing, is much more than just the application of massive numbers of MIPS to effect a computing solution; it also will provide a framework whereby massive numbers of services can be dynamically located, relocated, balanced, and managed so that needed applications are always guaranteed to be securely available, regardless of the load placed on the system. One of the aims of grid computing is the ability to manage ever-growing and ever more complex networks without overheads. Grid technology is about to evolve towards a “virtualisation layer” for hosting web services [Tuecke02]. This will enable what has been called recently “utility computing” or “on demand computing” [Leymann03]. The grid service domain architecture is a high-level abstraction model that describes the common behaviours, attributes, and operations and interfaces to allow a collection of services to function as an integral unit and collaborate with others in a fully distributed, heterogeneous, grid-enabled environment.

The Open Grid Services Infrastructure (OGSI) builds on both grid and web services technologies and defines mechanisms for creating, managing, and exchanging information among entities called *grid services*. A grid service is a web service that conforms to a set of conventions (interfaces and behaviours) that define how a client interacts with a grid service. These conventions, and other OGSI mechanisms associated with grid service creation and discovery, provide for the controlled, fault resilient, and secure management of the distributed and often long-lived state that is commonly required in advanced distributed applications.

Grid services are *stateful* services that provide a set of well-defined interfaces and follow specific conventions to facilitate coordinating and managing collections of web service providers/aggregators. The grid service indicates how a client can interact with it and is defined in WSDL. The state of the service is

exposed to its clients as a standard interface that addresses web service filtering, discovery, routing, aggregation, selection, data and context sharing, notification and life-time management.

The principal strengths of web and grid services are complementary with web services focusing on platform-neutral description, discovery and invocation, and grid services focusing on the dynamic discovery and efficient use of distributed computational resources. This complementarity of web and grid services has given rise to the proposed Open Grid Services Architecture (OGSA) [Foster02], [Tuecke02], which makes the functionality of grid services available through web service interfaces.

14 The extended service oriented architecture

Service-oriented architectures provide major advantages by presenting the interfaces that loosely coupled connections require, albeit being in the first stages of emergence. Service oriented computing is based on the premise that logic, be it business, computational, data access, etc, can be organized in ways that make it independent of the context in which it is being used. Just like the post office ignores why a particular letter or parcel is being shipped, services in SOA are designed to be combined in ways that are unknown at design or implementation time. Just like object-orientation promotes concepts such as interface extensions and inheritance, services (which cannot be extended or inherited from) are re-used via the notions of coordination (of which the most common form is composition) to perform a given unit of work. Web service technologies offer building blocks which are highly decoupled from each other. The current deployment of web services technology is promising early initiative in this direction. However, before this becomes true other standards and protocols for service-orientated architectures, particularly standards relating to security, coordination and the management of business processes, will have to become more robust before they can fully support mission-critical, long-lived web service transactions. Currently, the basic SOA does not address overarching concerns such as management, service choreography and orchestration, service transaction management and coordination, security, and other concerns that apply to all components in a services-based architecture. Such concerns are addressed by the extended SOA (ESOA) [Papa03] that is depicted in Figure 26. The architectural layers in the ESOA describe a logical separation of functionality in such a way that each layer definers a set of roles and responsibilities and leans on constructs of its predecessor layer to accomplish its mission. The logical separation of functionality is based on the need to separate basic service capabilities provided by the conventional SOA (for example building simple applications) from more advanced service functionality needed for composing services and the need to distinguish between the functionality for composing services from the management of services. As shown in Figure 26, the ESOA utilizes the basic SOA constructs as its bottom layer and layers service composition and management on top of it.

The basic services layer in the ESOA (see section-6) defines an interaction between software agents as an exchange of messages between service requesters (clients) and service providers. Providers are responsible for publishing a description of the service(s) they provide. Clients must able to find the description(s) of the services they require and must be able to bind to them. The interactions involve the publishing, finding and binding of operations. For reasons of conceptual simplicity in Figure 26 we assume that service clients, providers and aggregators can act as service brokers or service discovery agency (see section-6) and publish the services they deploy.

In a typical service-based scenario employing the basic services layer in the ESOA, a service provider hosts a network accessible software module (an implementation of a given service). The service provider defines a service description of the service and publishes it to a client (or service discovery agency) through which a service description is published and made discoverable. The service client (requestor) uses a find operation to retrieve the service description typically from a registry or repository like UDDI and uses the service description to bind with the service provider and invoke the service or interact with service implementation. Service provider and service client roles are logical constructs and a service may exhibit characteristics of both.

The *service composition layer* in the ESOA encompasses necessary roles and functionality for the consolidation of multiple services into a single composite service. Resulting composite services may be used by service aggregators as components (i.e., basic services) in further service compositions or may be utilized as applications/solutions by service clients. Service aggregators thus become service providers by publishing the service descriptions of the composite service they create. A service aggregator is a service

provider that consolidates services that are provided by other service providers into a distinct value added service. Service aggregators develop specifications and/or code that permit the composite service to perform functions that include the following (in addition to web services interoperation support, see section-12):

- Coordination: controls the execution of the component services, and manages dataflow among them and to the output of the component service (e.g., by specifying workflow processes and using a workflow engine for run-time control of service execution).
- Monitoring: allows subscribing to events or information produced by the component services, and publish higher-level composite events (e.g., by filtering, summarizing, and correlating component events).
- Conformance: ensures the integrity of the composite service by matching its parameter types with those of its components, imposes constraints on the component services (e.g., to ensure enforcement of business rules), and performs data fusion activities.
- QoS composition: leverages, aggregates, and bundles the component's QoS to derive the composite QoS, including the composite service's overall cost, performance, security, authentication, privacy, (transactional) integrity, reliability, scalability, and availability.
- Policy enforcement: web service capabilities and requirements can be expressed in terms of policies. For example, knowing that a service supports a web services security standard such as WS-Security is not enough information to enable successful composition. The client needs to know if the service actually requires WS-Security, what kind of security tokens it is capable of processing, and which one it prefers. The client must also determine if the service requires signed messages. And if so, it must determine what token type must be used for the digital signatures. And finally, the client must determine when to encrypt the messages, which algorithm to use, and how to exchange a shared key with the service. Trying to orchestrate with a service without understanding these details will lead to erroneous results.

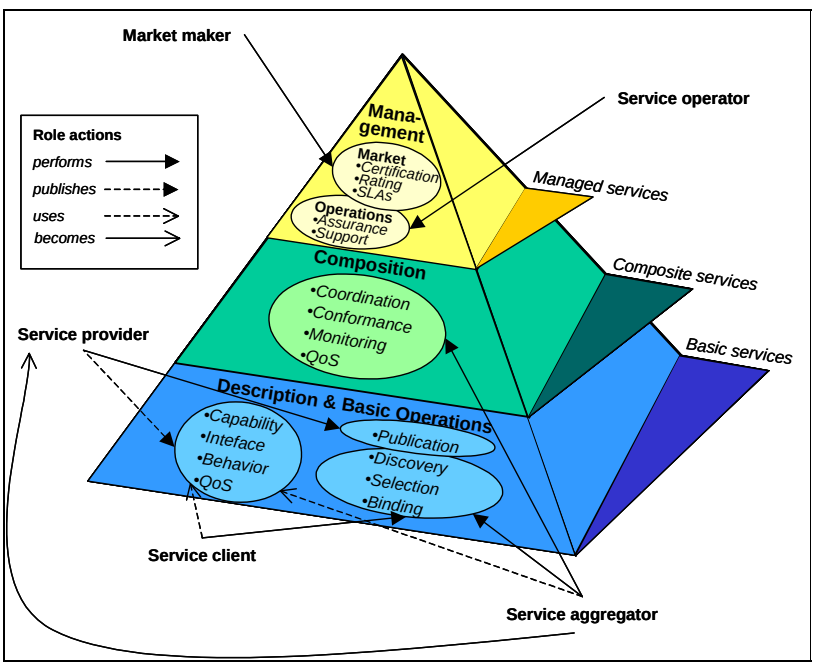


Figure 26 The Extended Service Oriented Architecture.

Managing critical web service based applications requires in-depth administration capabilities and integration across a diverse, distributed environment. For instance, any downtime of key e-business systems has a negative impact on businesses and cannot be tolerated. To counter such a situation, enterprises need to constantly monitor the health of their applications. The performance should be in tune, at all times and under all load conditions. Web service based application management is an indispensable element of the ESOA that includes performance management and business/application specific

management. This requires that a critical characteristic be realized: that services be managed. Service management includes many interrelated functions. The most typical functions include:

1. Deployment: The web services support environment should allow the service to be redeployed (moved) around the network for performance, redundancy for availability, or other reasons.
2. Metrics: The web services support environment should expose key operational metrics of a web service, at the operation level, including such metrics as response time and throughput. In addition it should allow web services to be audited.
3. Dynamic rerouting: The web services support environment should support dynamic rerouting for fail over or load balancing.
4. Life cycle/State management: The web services support environment should expose the current state of a service and permit lifecycle management including the ability to start and stop a service.
5. Configuration: The web services support environment should support the ability to make specific configuration changes to a deployed web service.
6. Change management and notification: The web services support environment should support the description of versions of web services and notification of a change or impending change to the service interface or implementation.
7. Extensibility: The web services support environment should be extensible and must permit discovery of supported management functionality in a given instantiation.
8. Maintenance: The web services support environment should allow for the management and correlation of new versions of the service.

Web services manageability could be defined as the functionality required for discovering the existence, availability, performance, health, patterns of usage, extensibility, as well as the control and configuration, life-cycle support and maintenance of a web service or business process within the context of the extended services architecture. This definition implies that web services can be managed using web services technologies. In particular, it suggests a manageability model that applies to both web services and business processes in terms of manageability topics, (identification, configuration, state, metrics, and relationships) and the aspects (properties, operations and events) used to define them [Potts03]. In fact, these abstract concepts apply to understanding and describing the manageability information and behaviour of any resource, including business processes and web services.

To manage critical applications/solutions and specific markets, ESOA provides managed services in the *service management layer* depicted at the top of the ESOA pyramid. The ESOA managed services are divided in two complementary categories:

- *Service operations management* that can be used to manage the service platform, the deployment of services and the applications and, in particular, monitor the correctness and overall functionality of aggregated/orchestrated services.
- *Open service marketplace management* that supports typical supply chain functions and by providing a comprehensive range of services supporting industry-trade, including services that provide business transaction negotiation and facilitation, financial settlement, service certification and quality assurance, rating services, service metrics, and so on.

The ESOA's *service operations management* functionality is aimed at supporting critical applications that require enterprises to manage the service platform, the deployment of services and the applications. ESOA's service operations management typically gathers information about the managed service platform, web services and business processes and managed resource status and performance, and supporting specific management tasks (e.g., root cause failure analysis, SLA monitoring and reporting, service deployment, and life cycle management and capacity planning). Operations management functionality may provide detailed application performance statistics that support assessment of the application effectiveness, permit complete visibility into individual business processes and transactions, guarantee consistency of service compositions, and deliver application status notifications when a particular activity is completed or when a decision condition is reached. We refer to the organization responsible for performing such operation management functions as the *service operator*. Depending on the application requirements a service operator may be a service client or aggregator.

In the context of service operations management it is increasingly important for management to define and support active capabilities versus traditional passive capabilities. For example, rather than merely raising an alert when a given web service is unable to meet the performance requirements of a given service-level agreement, the management framework should be able to take corrective action. This action could take the form of rerouting requests to a backup service that is less heavily loaded, or provisioning a new application server with an instance of the software providing the service if no backup is currently running and available.

Service operations management should also provide global visibility of running processes, comparable to that provided by Business Process Management (BPM). BPM promises the ability to monitor both the state of any single process instance and all instances in the aggregate, using present real-time metrics that translate actual process activity into key performance indicators (KPIs). Management visibility is expressed in the form of real-time and historical reports, and in triggered actions. For example, deviations from KPI target values, such as the percent of requests fulfilled within the limits specified by a service level agreement, might trigger an alert and an escalation procedure.

Considerations need also be made for modelling the scope in which a given service is being leveraged individual, composite, part of a long-running business process, and so on. Thus, in addition to the above concerns, which relate to individual business processes or services, in order to successfully compose web services processes), one must fully understand the service's WSDL contract along with any additional requirements, capabilities, and preferences (also referred to as policies). For example, knowing that a service supports a web services security standard such as WS-Security is not enough information to enable successful composition. The client needs to know if the service actually requires WS-Security, what kind of security tokens it is capable of processing, and which one it prefers. The client must also determine if the service requires signed messages. And if so, it must determine what token type must be used for the digital signatures. And finally, the client must determine when to encrypt the messages, which algorithm to use, and how to exchange a shared key with the service. Trying to orchestrate with a service without understanding these details will lead to erroneous results. Such concerns are addressed by the service operations management. Service operations management is a critical function that can be used to monitor the correctness and overall functionality of aggregated/orchestrated services thus avoiding a severe risk of service errors. In this way one can avoid typical errors that may occur when individual service-level agreements (SLAs) are not properly matched. This fact was illustrated by the failure of the rail network operator in the UK a few years ago, apparently triggered in part by a complete mismatch between the SLAs imposed on the track repair subcontractors and the SLAs and legitimate safety expectations of the train companies. Proper management and monitoring provides a strong mitigation of this type of risk, since the operations management level allows business managers to check the correctness, consistency and adequacy of the mappings between the input and output service operations and aggregate services in a service composition.

Another aim of ESOA's service management layer is to provide support for open service marketplaces. Currently, there exist several vertical industry marketplaces, such as those for the semiconductor, automotive, travel, and financial services industries. Open service marketplaces operate much in the same way like vertical marketplaces, however, they are open. Their purpose is to create opportunities for buyers and sellers to meet and conduct business electronically, or aggregate service supply/demand by offering added value services and grouping buying power (just like a co-op). The scope of such a service marketplace would be limited only by the ability of enterprises to make their offerings visible to other enterprises and establish industry specific protocols by which to conduct business. Open service marketplaces typically support supply chain management by providing to their members a unified view of products and services, standard business terminology, and detailed business process descriptions. In addition, service marketplaces must offer a comprehensive range of services supporting industry-trade, including services that provide business transaction negotiation and facilitation, financial settlement, service certification and quality assurance, rating services, service metrics such as number of current service requesters, average turn around time, and manage the negotiation and enforcement of SLAs. ESOA's service management layer includes market management functionality (as illustrated in Figure 26 that is aimed to support these marketplace functions. The marketplace is created and maintained by a *market maker* (a consortium of organizations) that brings the suppliers and vendors together. The market maker assumes the responsibility of marketplace administration and performs maintenance tasks to ensure the

administration is open for business and, in general, provides facilities for the design and delivery of an integrated service that meets specific business needs and conforms to industry standards.

The ESOA service management functions can benefit from grid computing as it targets manageability, see section-13. Service grids constitute a key component of the distributed services management as the scope of services expands beyond the boundaries of a single enterprise to encompass a broad range of business partners, as is the case in open marketplaces. For this purpose grid services can be used to provide the functionality of the ESOA's service management layer [Foster02], [Tuecke02]. Grid services used in the ESOA's service management layer to provide an enabling infrastructure for systems and applications that require the integration and management of services with the context of dynamic virtual marketplaces. Grid services provide the possibility to achieve end-to-end qualities of service and address critical application and system management concerns.

15 Composite Applications

Composite applications are built on the premise that application boundaries are becoming harder to define. In a not so distant past, applications were installed on a single server within a company firewall. Over time, people integrated these applications together bridging islands of functionality, delivering new value to old systems. Later, web technologies opened up new possibilities by providing a common front end while seamlessly interacting with multiple servers, sometimes, way beyond company boundaries. Moreover, web applications allowed new categories of users to securely access application functionality that would have normally been accessed by highly trained workers through complex clients. For instance many companies have built separate Order Entry and Inventory Management applications. When orders were coming through a paper trail (fax, mail, …) it was probably sufficient for a clerk to use both applications at the same time and key in the same kind of information on both systems. The web really changed the expectations of the customers in that kind of scenario and required a totally integrated experience hiding the intricacies of these systems to the self-served user. The web technologies themselves have transformed application integration, from being a back-end process only, often batch oriented, into a “front-end” driven integration whereby user activities on a browser could directly interact with multiple systems alleviating both the need to log into different systems and interact with different kinds of clients or the need to integrate these applications in the back-end when one front end could be used to perform the activity. The focus of composite applications is to provide a unified and seamless user experience regardless of the number and geographic location of systems that are involved in a given user activity.

The Service Oriented Architecture and web services are key enablers in the construction of composite applications. They enable an application model where some or most of the business logic is executed within services outside the composite application domain. The composite application itself is mostly responsible for managing user activities and the coordination of services. Some of these services represent the “systems of record”. The traditional Model-View-Controller pattern shifts to a Service-View-Coordination pattern. There are multiple forms of coordination as we have seen in section 9. In this new application model, the View and Coordination layers are loosely coupled: a given view might even interact with different coordination providers.

The traditional realm of application of Enterprise Application Integration is at the “Model” level (Figure 27). Because of this new application model, composite applications do not require as much “back-end” integration. The interactions with multiple systems can be handled directly by the coordination layer.

Composite applications differ quite extensively from “Portals”. In a portal, a user is very conscious of the different sources of information that they access. Portals are also often mostly read only. In a composite application, a user is performing enterprise activities like entering an order or creating a quote but these activities can now reach far beyond the boundaries of a single enterprise application.

The concept of composite applications will open the way for a new kind of business logic that we can label “Global Business Logic”. This kind of business logic lives outside the enterprise boundaries for various reasons: the amount of information on which it is based may be too large to host in every company, the information maybe a real-time feed, or the logic itself maybe fairly complex and changing often enough for justifying hosting it as services that can then be accessed by the consumer application. For instance, the problem of calculating sales taxes is usually handed by an enterprise’s ERP systems. However, it is easy to

imagine that an ERP system may never be able to give an enterprise an accurate calculation if it is selling worldwide because of the intricacies of scenarios and regulation, not to mention the impact of ever changing legislation. In a composite application scenario it would be relatively easy to invoke a service provided by SalesTax.com (a fictitious company) each time a user needs to prepare a quote, an order or an invoice. If a company sells worldwide and there are no services available that can give it a world wide response, it may have to create its own "service composition" to cover all possible geographical cases. However, it is likely that over time the regional "SalesTax.com" will federate and offer a composite service.

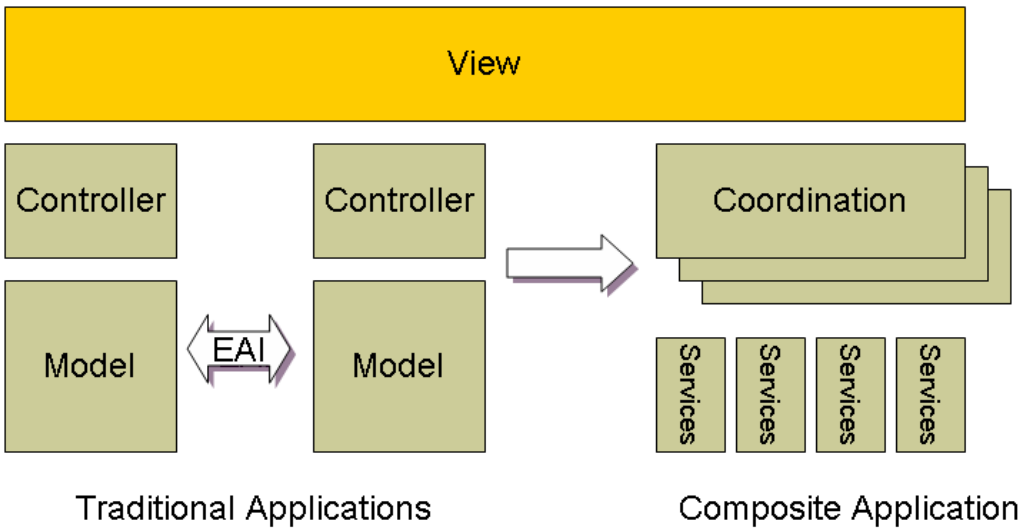


Figure 27 Evolution of the application model

Composite applications represent the natural evolution of existing enterprise and legacy systems. They represent yet another contribution to the return on investment to evolving information technology assets towards the Service Oriented Architecture.

16 Research directions

This section focuses on research activities conducted on services. We classify these research activities on the basis of the functional layers of ESOA and summarize several representative research initiatives under each functional layer.

16.1 ESOA basic services layer: research activities

Research activities in the basics services layer to date target formal service description language(s) for holistic service definitions addressing, besides functional aspects, also behavioural as well as non-functional aspects associated with services. They also concentrate on open, modular, extensible framework for service discovery, publication and notification mechanisms across distributed, heterogeneous, dynamic (virtual) organisations as well as unified discovery interfaces and query languages for multiple pathways. In the following we summarise several research activities contribute to these and related problems.

In addition to the application-specific functions that services provide, services may also support (different) sets of protocols and formats addressing extra-functional concerns such as transaction processing and reliable messaging. This raises the need for services to complement their functional service descriptions with descriptions of extra functional capabilities, requirements, and/or preferences, which must be matched and enforced for service interactions. Tai et. al [Tai04] address the problem of transactional coordination in service-oriented computing. The authors of this publication argue for the use of declarative policy assertions to advertise and match support for different transaction styles (direct transaction processing, queued transaction processing, and compensation-based transaction processing) and introduce the concept of and system support for transaction coupling modes as the policy-based contracts guiding transactional business process execution.

The web services approach is requires that developers discover (at development time) service descriptions on UDDI and, by reading these descriptions they are able to code client applications that can (at run time) bind to and interact with services of a specific type (i.e., compliant to a certain interface and protocol). Understanding the execution semantics is a rather cumbersome task. Thus, richer service descriptions and richer description models are needed for this purpose. Benatallah et. al. propose an extension of SOA basic services layer for defining extended service models to enable the definition and description of richer execution abstractions [Benatallah03]. This framework enables the definition of service properties in a way that can support: (i) humans in understanding the service execution properties, (ii) clients in searching services based on these properties, and (iii) applications in automating the enforcement of these properties, much like transactional middleware supports transactional abstractions.

Leymann proposes extended middleware facilities based on business process technology to enable the composition web services into higher-level business functionality based on a two-level programming paradigm [Leymann03]. The high-level middleware facilities are based on the concept of a service that channels service requests to service providers. Choices of prospective services are made on the basis of QoS 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). The service bus also considers combining of multiple policies into a single policy that describes a service or a request.

The AI and semantic web community has concentrated their efforts in giving richer semantic descriptions of Web services that describe the properties and capabilities of web services in an computer-intepretable form. For this purpose DAML-S language has been proposed to facilitate the automation of Web service tasks including better means of web service discovery, execution, "automatic" composition, verification and execution monitoring. The following two publications are representative publications from this community that propose semantic extensions to the basic SOA functionality.

In the basic SOA UDDI provides a simple browsing-by-business-category mechanism for developers to review and select published services. Stroulia and Wang [stroulia03] developed methods that utilize both the semantics of WSDL descriptions and the structure of their operations, messages and types to assess the similarity of any two WSDL services. Given only a textual description of the desired service, a semantic information-retrieval method can be used to identify and order the most similar service-descriptions. If a (potentially partial) specification of the desired service behaviour is also available, this set of likely candidates can be further refined by a signature-matching step assessing the structure similarity of the desired vs. the retrieved services.

In order for service-oriented architectures to become successful, powerful mechanisms are needed that allow service requestors to find service providers that are able to provide the services they need. Typically, this service trading needs to be executed in several stages as the offer descriptions are not completely specified in most cases and different parameters have to be supplemented by the service requestor and provider alternately. Unfortunately, existing service description languages (like DAML-S) treat service discovery as a one shot activity rather than as a process and accordingly do not support this stepwise refinement. Klein et. al introduce the concept of partially instantiated service descriptions containing different types of variables which are instantiated successively, thereby mirroring the progress in a trading process [Klein03].

16.2 ESOA composition layer: research activities

Service composition is today largely a static affair. All service interactions are anticipated in advance and there is a perfect match between output and input signatures and functionality. More ad hoc and dynamic service compositions are required very much in the spirit of lightweight and adaptive workflow methodologies. These methodologies will include advanced forms of co-ordination, less structured process models, and automated planning techniques as part of the integration/composition process. On the transactional front, although standards like WS-Transaction, WS-Coordination and BTP are a step in the right direction, they fall short of describing different types of atomicity needs for e-business and e-government applications. These do not distinguish between transaction phases and conversational sequences, e.g., negotiation. Another area that is lacking research results is advanced methodologies in support for the service composition lifecycle. Several research activities contribute to these and related problems.

Yang and Papazoglou present an integrated framework and prototype system that manage the entire life-cycle of service components ranging from abstract service component definition, scheduling, and construction to execution [Yang04]. Service compositions are divided in three categories: fixed, semi-fixed and explorative compositions. Fixed service compositions require that their constituent services be synthesized in a fixed (pre-specified) manner. Semi-fixed compositions require that the entire service composition s specified statically but the actual service bindings are decided at run time. Finally, explorative compositions are generated on the fly on the basis of a request expressed by a client (application developer). A companion article introduces the concept of a service component that raises the level of abstraction in web service compositions [Yang03]. Service components represent modularised service-based applications that associate service interfaces with business logic into a single cohesive conceptual module. Service components can be extended, specialized, and generally inherited, to facilitate the creation of applications.

In [Orriens03] the authors discuss how business processes can be built dynamically by composing web services in a model driven fashion where the design process is controlled and governed by a series of business rules. This publication examines the functional requirements of service composition and introduce a phased approach to the development of service compositions and analyses the information requirements for developing service compositions by identifying the basic elements in a web service composition and the business rules that are used to govern the development of service compositions.

The following two publications concentrate on the development of theoretical approaches for service composition and verification thereof.

Berardi et. al develop a theoretical framework in which the exported behaviour of a web service is described in terms of its possible execution sequences (execution trees) which are represented by finite state machines [Berardi03]. Subsequently, the complexity of synthesizing a composition is analysed and algorithms to check that valid compositions are proposed.

Meredith and Borg examine the complexity problem of distributed heterogeneous applications (assuming that service connectivity has been addressed) [Meredith03]. They propose a formal approach based on the development of type systems for the specification and automatic verification of crucial properties of service behaviour.

The AI community has been concerned with designing semantic web standards for adding semantic mark-up to web service descriptions, and has proposed semantic type matching algorithms, interleaved search mechanisms and execution algorithms that together allow for basic automated service composition. There has been some work on the instantiation (based on user preferences and service availability) of precompiled plans in [McIlraith02] as well as on extending the planning domain description language PDDL to handle information producing actions [McDermott02]. Other activities assume full knowledge of the semantics of operations [Aiello02], the authors use a non-deterministic planning language with extended-goals and constraint satisfaction to model the web services planning problem. A different approach was taken by the authors of [Thakkar02] in which automated service composition is achieved by modelling services as web information sources (exposed by automated web-site wrapping software) for which a common data model was already known. A common data model means that database query planning and transformation techniques can be used for plan synthesis and optimisation.

In all of these works the authors assume to be interacting with services that are described in a standard and possibly formal manner, i.e. all services which provide the same functionality are called in the same way, require the same inputs and produce the same outputs.

16.3 ESOA management layer: research activities

Service management constitutes the foundation of the upper layer of the extended SOA. Traditional management applications fail to meet enterprise requirements in a service-centric world. Conventional systems management approaches and products view the world in a very coarse (mostly applications oriented) manner. The most recent wave of management product categories does not have the business-awareness that services management will require. The finer grained nature of services (as opposed to

applications) requires evaluating processes and transactions at a more magnified rate and in a more contextually aware manner.

Casati et al. shift attention to the management layer of the SOA and more specifically to operations management [Casati03]. The proposed business oriented management of web services is an attempt to assess the impact of service execution from a business perspective and, conversely, to adjust and optimize service executions based on stated business objectives. This is a crucial issue as corporations strive to align service functionality with business goals.

The ability to gauge the quality of a service is critical if we are to achieve the service oriented computing paradigm. Many techniques have been proposed and most of them attempt to calculate the quality of a service by collecting quality ratings from the users of the service, and then combining them in one way or another. Collecting quality ratings alone from the users is not sufficient for deriving a reliable or accurate quality measure for a service. To address this problem Deora et. al. [Deora03] propose a quality of service management framework based on user expectations. This framework collects expectations as well as ratings from the users of a service and then the quality of the service is calculated only at the time a request for the service is made and only by using the ratings that have similar expectations.

17 Concluding remarks

Web services are lightweight constructs that enable the development of rapid, low-cost and easy composition of distributed applications. Web services are self-describing, platform-agnostic computational modules that support rapid, low-cost and easy composition of loosely coupled distributed applications. The promise of web service technology is a world of cooperating services where application components are assembled with little effort into a network of loosely coupled services to create dynamic business processes and agile applications that span organisations and computing platforms.

Key to developing web service-based applications is the service-oriented architecture (SOA). SOA is a logical way of designing software solutions to provide services to either end-user applications or to other services distributed in a network, via published and discoverable interfaces. *Service descriptions* are used to advertise the service capabilities, interface, behaviour, and quality. Publication of such information describing available services (in a service registry) provides the necessary means for the discovery, selection, binding, and composition of services. In particular, the *service interface description* publishes the service signature while the *service capability description* states the conceptual purpose and expected results of the service. The (expected) behaviour of a service during its execution is described by its *service behaviour description* (e.g., as a workflow process). Finally, the *Quality of Service* (*QoS) description* publishes important functional and non-functional service quality attributes, such as service metering and cost, performance metrics (response time, for instance), security attributes, (transactional) integrity, reliability, scalability, and availability.

Currently, the SOA provides the basic operations necessary to describe, publish, find and invoke services. However, those basic operations—while they help services to be ubiquitous and universal—are not a complete solution. For services to be used widely, there is additional functionality that must be considered for service composability — including specifications regarding the dynamic composition of services, transactional context and coordination, adaptability to varying circumstances, security and so on — as well as for service management. In addition, the SOA is accelerating and exacerbating a systems management challenge that has been growing in urgency in parallel with the development of enterprise-scale distributed computing. Such concerns are addressed by the extended service-oriented architecture (ESOA).

The ESOA extends the basic service description/publication/discovery functions of the conventional service-oriented architecture by providing a service composition tier to offer necessary roles and functionality for the consolidation of multiple services into a single composite service. It also provides a tier for service management that can be used to monitor the correctness and overall functionality of aggregated/orchestrated services, supporting complex aggregate (cross-component) management use cases, such as service-level agreement enforcement and dynamic resource provisioning. The layers of the

ESOA provide a natural conceptual framework for grouping and discussing current research activities in the field of web services.

18 References

[Aiello02] M. Aiello, et. al. "A request language for web services based on planning and constraint satisfaction.", Workshop on Technologies for E-Services (TES), Hong-Kong, Springer Verlag, September 2002.

[Aldrich02] S.E. Aldrich "Anatomy of Web Services", 2002 Patricia Seybold Group, Inc, www.psgroup.com

[Andrews03] T. Andrews et. al. "Business Process Execution Language for Web Services", Version 1.1, March 2003.

[Ankolenkar 01] A. Ankolenkar, at. al. "DAML-S: A Semantic Markup Language for Web Services. In Proceedings of SWWS'01, Stanford, USA, August 2001, also available at xml.coverpages.org/ISWC2002-DAMLS.pdf

[Arkin01] A. Arkin "Business Process Modelling Language" March 2001, bpmi.org.

[ASP00] ASP Industry Consortium, Internet Survey http://www.aspindustry.org/surveyresults.cfm, 2000.

[Atkinson02] B. Atkinson et. al. "Web Services Security (WS-Security)", http://www.ibm.com/developerworks/library/ws-secure/, April 2002.

[BEA01] BEA Systems Inc., "BEA WebLogic Server: Programming WebLogic Web Services", December 2001, available at www.bea.com.

[Benatallah03] B. Benatallah, F. Casati, F. Toumani, R. Hamadi "Conceptual Modeling of Web Service Conversations", Proc. of 15th International Conference on Advanced Information Systems Engineering (CAiSE'03), Springer Verlag, Velden, Austria, June 2003.

[Berardi03] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella "Composing eServices that export their behavior ", Proc. Of 1st International Conference on Service Oriented Computing, Springer Verlag, Trento, Italy, Dec. 2003.

[Beugnard99] Beugnard, A.; Jezequel, J.-M.; Plouzeau, N.; & Watkins, D. "Making Components Contract Aware." Computer 32, 7 (July 1999): 38-45.

[Bloch03] B. Bloch et al. (eds), "Web Services Business Process Execution Language", OASIS Open Inc. Working Draft 01, October 2003, http://www.oasis-open.org/apps/org/workgroup/wsbpel/

[Boubez00] T.Boubez, Web Services architecture overview: The next stage of evolution for e-business, IBM DeveloperWorks Web Architecture Library, October 20th 2000.

[Cabrera02a] F. Cabrera et al., "Web Services Coordination (WS-Coordination)," August 2002, http://www.ibm.com/developerworks/library/ws-coor/

[Cabrera02b] F. Cabrera et al., "Web Services Transaction (WS-Transaction)," August 2002, http://www.ibm.com/developerworks/library/ws-transpec/

[Casati03] F. Casati, E. Shan, U. Dayal, M. Shan "Business Oriented Management of Web Services", Communications of ACM, Special Section on Service-Oriented Computing, M.P. Papazoglou and D. Georgakopoulos (eds), October 2003.

[Cauldwell01] P. Cauldwell, et. al., "XML Web Services", Wrox Press Ltd., 2001.

[Chappell01] D.A. Chappell et. al. "ebXML Foundations", Wrox Press Ltd., 2001.

[Curbera03] P. Curbera, R. Khalaf, N. Mukhi, S. Tai, S. Weerawarana, "Web services, the next step: A framework for robust service composition," Communications of ACM, Special Section on Service-Oriented Computing, M.P. Papazoglou and D. Georgakopoulos (eds), October 2003.

[Deora03] V. Deora, J. Shao, W. A. Gray ,N.J Fiddian, "Rating Based Quality of Service Management on Expectations", Proc. Of 1st International Conference on Service Oriented Computing, Springer Verlag, Trento, Italy, Dec. 2003.

[Ehnebuske 01] D. Ehnebuske, D. Rogers and C. von Riegen (2001), UDDI Version 2.0 – Data Structure Specification, UDDI.org, June 2001.

[Eisenberg03] R. Eisenberg "Business Process Management: The next generation of software", EAI Journal, June 2003, pp. 28-35.

[Erlikh02] L. Erlikh "Integrating Legacy Systems Using Web services", EAI Journal, August 2002, pp. 12-17.

[Foster02] I. Foster, C. Kesselman, J. M. Nick, S. Tuecke, "Grid Services for Distributed System Integration", IEEE Computer, 35(6), 2002.

[Goepfert02] J. Goepfert, M. Whalen An Evolutionary View of Software as a Service, IDC White paper, www.idc.com, 2002.

[Graham02] S. Graham et. al. (2002), "Building Web Services with Java", SAMS Publishing, 2002.

[Grangard01] A. Grangard et al. (2001), ebXML Technical Architecture Specification v1.0.4, ebXML Technical Architecture Project Team, http://www.ebxml.org, February 16th 2001.

[Hoque00] F. Hoque "e-Enterprise", Cambridge Univ. Press, 2000.

[Holland02] P. Holland "Building Web Services from Existing Applications", EAI Journal, September 2002, pp. 45-47.

[Kavantzas04] Web Services Choreography Description Language 1.0, Editor's Draft, April 3 2004, http://lists.w3.org/Archives/Public/www-archive/2004Apr/att-0004/cdl_v1-editors-apr03-2004-pdf.pdf.

[Klein03] M. Klein, B. König-Ries, P. Obreiter "Stepwise Refinable Service Descriptions: Adapting DAML-S to Staged Service Trading", Proc. Of 1st International Conference on Service Oriented Computing, Springer Verlag, Trento, Italy, Dec. 2003.

[Leymann00] F. Leymann, D. Roller, "Production Workflow", Prentice Hall Inc., New Jersey, 2000.

[Leymann03] F. Leymann "Web Services: Distributed Applications without Limits", Database Systems For Business, Technology and Web BTW 2003, Leipzig, Germany, February, 2003.

[Little03a] M. Little, J. Webber "Introducing WS-Transaction: The basis of the WS-Transaction protocol", Web Services Journal, vol. 5, issue 6, June 2003, pp. 28-33.

[Little03b] M. Little "Transactions and Web Services", Communications of the ACM, vol. 46, no. 10, October 2003, pp. 49-54.

[Little03c] M. Little, J. Webber "Introducing WS-CAF - More than just transactions", Web Services Journal, vol. 3, issue 12, December 2003

[Manes 03] A.T. Manes, "Registering a Web service in UDDI", Web Services Journal, vol. 3, issue 10, October 2003, pp. 6-10.

[Mani02] A. Mani, A. Nagarajian "Web services : Understanding quality of service for Web services" IBM Developer Works January 2002, http://www-106.ibm.com/developerworks/library/ws-quality.html

[Masud03] S. Masud "RosettaNet Based Web Services", IBM Developer Works, July 2003.

[McDermott02] D. McDermott "Estimated-regression planning for interactions with web services", AI Planning Systems Conference, 2002.

[McIlraith 02] S. McIlraith, T. Son. Adapting golog for composition of semantic web services. In Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002). Morgan Kaufmann, 2002.

[McKee01] B. McKee, D. Ehnebuske and D. Rogers (2001), UDDI Version 2.0 – API Specification, UDDI.org, June 2001.

[Meredith03] G. Meredith, J. Borg "Contracts and Types", Communications of ACM, Special Section on Service-Oriented Computing, M.P. Papazoglou and D. Georgakopoulos (eds), October 2003.

[Mysore03] S. Mysore "Securing Web Services — Concepts, Standards, and Requirements", Sun Microsytems, October 2003, sun.com/software.

[Orriens03] B. Orriens, J. Yang, M. P. Papazoglou "Model Driven Service Composition" Proc. Of 1st International Conference on Service Oriented Computing, Springer Verlag, Trento, Italy, Dec. 2003.

[Potts03] M. Potts et. al "Web Service Manageability – Specification 1 (WS-Manageability)", OASIS, September 2003.

[Papazoglou02] M. Papazoglou, J. Yang "Design Methodology for Web Services and Business Processes", Workshop Technologies for Electronic Services, Sept. 2002, Hong-Kong.

[Papazoglou03a] M. Papazoglou, D. Georgakopoulos, "Service Oriented Computing", Communications of the ACM, vol. 46, no. 10, October 2003, pp. 25-28.

[Papazoglou03b] M. Papazoglou, "Web Services and Business Transactions", World Wide Web Journal, vol. 6, no.1, March 2003, pp. 49-91.

[Pilz03] G. Pliz "ANew World of Web Services Security", Web Services Journal, March 2003.

[SAML03] Security Assertions Markup Language SAML: Overview xml.coverpages.org/saml.html, December 2003.

[Samtani01] G. Samtani "EAI and Web Services: Easier Enterprise Application Integration?" Web Services Architect, October, 2001.

[Samtani02] G. Samtani, D. Sadhwani "Enterprise Application Integration and Web services", in Web Services Business Strategies and Architectures P. Fletcher, M. Waterhouse (eds), Expert Press, Birmingham, UK, 2002.

[Stroulia03] E. Stroulia, Y. Wang "WSDL Semantic Signature Matching", Proc. Of 1st International Conference on Service Oriented Computing, Springer Verlag, Trento, Italy, Dec. 2003.

[Tai04] S. Tai, T. Mikalsen, E. Wohlstadter, N. Desai, I. Rouvellou "Transaction Policies for Service-Oriented Computing", to appear in Knowledge and Data Engineering, 2004.

[Thakkar 02] S. Thakkar, C. A. Knoblock, J. L. Ambite, C. Shahabi "Dynamically composing web services from

on-line sources" Workshop on Intelligent Service Integration, 18th National Conference on Artificial Intelligence (AAAI), 2002.

[Tuecke 02] S. Tuecke, et. al. "Grid Service Specification", Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 2002. Draft 5, November 5, 2002.

[Vinugopal01] K.E. Vinugopal, J.G. Kupper, P.J. Murray (2001) EAI and Web Services – A Simple Guide, Webservices.org, December, 2001.

[Webber03] J. Webber, M. Little "Introducing WS-Coordination", Web Services Journal, vol. 5, issue 5, May 2003, pp. 12-16.

[WSI03] Web Services Interoperability Organization "WS-I Overview", January 2003, available at http://www.ws-i.org/Documents.aspx.

[WS-Policy03] OASIS Cover Pages "Updated Versions of Web Services Policy (WS-Policy) Specifications", June 2003, http://xml.coverpages.org/ni2003-06-04-a.html.

[Wahli04] U. Wahli, G.G. Ochoa, S. Cocasse, M. Muetschard "WebSphere Version 5.1: Web Services Handbook", Februry 2004, IBM Redbooks, available at ibm.com/redbooks/.

[Wfmc02] Workflow Management Coalition "Workflow Process Definition Interface -- XML Process Definition Language" document no. WFMC-TC-1025, October 25, 2002.

[Yang03] J. Yang "Web Service Componentization", Communications of ACM, Special Section on Service-Oriented Computing, M.P. Papazoglou and D. Georgakopoulos (eds), October 2003.

[Yang04] J. Yang, M.P.Papazoglou "Service Components for Managing the Life-Cycle of Service Compositions", Information Systems, vol. 28, no. 1, 2004.

