Service-Oriented Computing

edited by Dimitrios Georgakopoulos and Michael P. Papazoglou

The MIT Press
Cambridge, Massachusetts
London, England

For information about special quantity discounts, please e-mail special_sales@mitpress.mit.edu

This book was set in Times Roman by SNP Best-set Typesetter Ltd., Hong Kong.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Service-oriented computing / edited by Dimitrios Georgakopoulos and Michael P. Papazoglou.
p. cm.—(Cooperative information systems)
Includes bibliographical references and index.
ISBN 978-0-262-07296-0 (hardcover : alk. paper) 1. Web services. I. Georgakopoulos, Dimitrios. II. Papazoglou, M., 1953–
TK5105.88813.S45 2009
006.7′6—dc22

2007039722

10 9 8 7 6 5 4 3 2 1

1 Overview of Service-Oriented Computing

Dimitrios Georgakopoulos and Michael P. Papazoglou

1.1 Introduction

Service-Oriented Computing (SOC) is a computing paradigm that utilizes services as fundamental elements to support rapid, low-cost development of distributed applications in heterogeneous environments. The promise of Service-Oriented Computing is a world of cooperating services that are being loosely coupled to flexibly create dynamic business processes and agile applications that may span organizations and computing platforms, and can adapt quickly and autonomously to changing mission requirements.

Realizing the SOC promise involves developing Service-Oriented Architectures (SOAs) [13] [23] and corresponding middleware that enables the discovery, utilization, and combination of interoperable services to support virtually any business process in any organizational structure or user context. SOAs allow application developers to overcome many distributed enterprise computing challenges, including designing and modeling complex distributed services, performing enterprise application integration, managing (possibly cross-enterprise) business processes, ensuring transactional integrity and QoS, and complying with agreements, while leveraging various computing devices (e.g., PCs, PDAs, cell phones, etc.) and allowing reuse of legacy systems [4]. SOA strives to eliminate these barriers so that distributed applications are simpler/cheaper to develop and run seamlessly. In addition, SOA provides the flexibility and agility that business users require, allowing them to define coarse-grained services, which may be aggregated and reused to address current and future business needs.

The design principles of an SOA [13] [3] are independent of any specific technology, such as Web Services or J2EE Enterprise Java Beans. In particular, SOA prescribes that all functions of a SOA-based application are provided as services [2]. That is, SOA services include all business functions and related business processes that comprise the application, as well as any system-related function that is necessary to support the SOA-based application. In addition to providing for the functional decomposition of applications to services, SOA requires services to be:

- Self-contained
- Platform-independent
- Dynamically discoverable, invokable, and composable.

A service is *self-contained* when it maintains its own state independently of the application that utilizes it. Services are *platform-independent* if they can be invoked by a client using any network, hardware, and software platform (e.g., OS, programming language, etc.). Platform independence also implies that an SOA service has an interface that is distinct from, and abstracts the details of, the service implementation. The service interface defines the identity of a service and its invocation mechanism. The service implementation implements the SOA function that the service is designed to provide. Finally, SOA requires that services may be *dynamically discovered, invoked,* and *composed*. Dynamic service discovery assumes the availability of an SOA service that supports service discovery. This may include a service directory, taxonomy, or ontology that service clients query to determine which service(s) can provide the functions they need. To ensure invocability, SOA requires that service interfaces include mechanisms that allow clients to invoke services and/or be notified by services as needed. This implies that clients are unaware of the network protocol used to perform the service invocation and of the middleware platform components required to establish the connection. The combination of service invocability and platform independence permits clients to invoke any service from anywhere and at any time the service is needed. Finally, services are composable if they can be combined and used by business processes that may span multiple service providers and organizations.

Each service in an SOA-based application may implement a brand-new function, it may use parts of old applications that were adapted and wrapped by the service implementation, or it may combine new code and legacy parts. In any case, the developers of the service clients typically do not have direct access to the service implementation other than indirectly through its interface. For example, web services publish their service interfaces only without revealing their implementation or the inner workings of their provider. Therefore, SOA permits enterprises to create, deploy, and integrate multiple services and to choreograph new business functions and processes by combining new and legacy application assets encapsulated in services. Furthermore, due to its dynamic nature, SOA can potentially provide just-in-time integration of services that offer a new product or a client and/or time-dependent service that has never been provided to a client before. This is a key enabler for real-time enterprises.

Web Services have become the preferred implementation technology for realizing SOAs [34]. Their success is due to basing their development on existing, ubiquitous infrastructure such as HTTP, SOAP, and XML.

In this chapter, we survey the underpinnings of SOA and discuss technologies that can springboard enterprise integration projects. In addition, we review proposed enhancements of SOA, such EDA and xSOA. The Event-Driven Architecture (EDA) is an event-driven SOA that provides support for complex event processing and provides additional flexibility. The extended SOA (xSOA) provides SOA extensions for service composition and management. This, chapter

is unique in that it unifies the principles, concepts, and developments in enterprise application integration, middleware, SOAs and event-driven computing. It also explains how these contribute to an emerging distributed computing technology known as the Enterprise Service Bus. Moreover, this chapter introduces the remaining chapters in the book that discuss enhancements to the conventional SOA, EDA, and xSOA.

1.2 Service Roles in SOA

SOAs and Web Services solutions support two key roles: a service requester (client) and a service provider, which communicate via service requests. A role thus reflects a type of participant in an SOA [13] [33].

Service requests are messages formatted according to the Simple Object Access Protocol (SOAP) [11]. SOAP entails a light-weight protocol allowing RPC-like calls over the Internet using a variety of transport protocols including HTTP, HTTP/S, and SMTP. In principle, SOAP messages may be conveyed using any protocol as long as a binding is defined. The SOAP request is received by a runtime service (an SOAP "listener") that accepts the SOAP message, extracts the XML message body, transforms the XML message into a protocol that is native to the requested service, and delegates the request to the actual function or business process within an enterprise. After processing the request, the provider typically sends a response to the client in the form of an SOAP envelope carrying an XML message.

Requested operations of Web Services are implemented using one or more Web Service components [55]. Web Service components may be hosted within a Web Services container [21] serving as an interface between business services and low-level infrastructure services. In particular, Web Service containers are similar to J2EE containers [3], and provide facilities such as location, routing, service invocation, and management. A service container can host multiple services, even if they are not part of the same distributed process. Thread pooling allows multiple instances of a service to be attached to multiple listeners within a single container [17].

SOAP is by nature a platform-neutral and vendor-neutral standard. These characteristics allow for a loosely coupled relationship between requester and provider, which is especially important over the Internet, where two parties may reside in different organizations or enterprises. However, SOA does not require the usage of SOAP and other service transports have been used in the past, for example in [45].

The interactions between service requesters and service providers can be complex, since they involve discovering/publishing, negotiating, reserving, and utilizing services from potentially different service providers. An alternative approach for reducing such complexity is to combine the service provider and requester functionality into a new role, which we refer to as service aggregator [37]. The service aggregator thus performs a dual role. First, it acts as an application service provider, offering a complete "service" solution by creating composite, higher-level services. Service aggregators can accomplish this composition using specialized composition

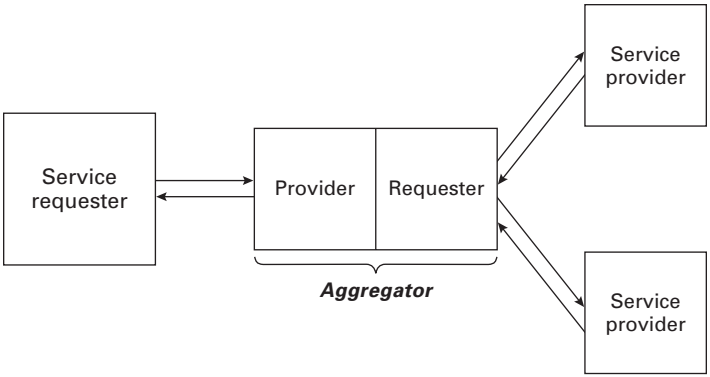

Figure 1.1
The role of the service aggregator.

languages such as BPEL [5] and BPML [6]. A service agreegator also acts as a service requester by requesting and reserving services from other service providers. This process is shown in figure 1.1.

Though service aggregation may offer these service composition benefits to the requester, it is also a form of service brokering that offers a convenience function that groups all the required services "under one roof." However, an important question that needs to be addressed is how a service requester selects a specific application service provider for its service offerings. The service requester can retain the right to select an application service provider based on those that can be discovered from a registry service, such as UDDI [1]. SOA technologies such as UDDI, and security and privacy standards such as SAML [40] and WS-Trust [80], introduce another role which aids service selection and it is called the service broker [19].

Service brokers are trusted parties that force service providers to adhere to information practices that comply with privacy laws and regulations or, in the absence of such laws, industry best practices. In this way broker-sanctioned service providers are guaranteed to offer services that are in compliance with local regulations and to create a more trusted relationship with customers and partners. A service broker maintains a registry of available services and providers, as well as value-added information about the provided services. This may include information about service reliability, trustworthiness, quality, and possible compensation, to name a few.

Figure 1.2 shows an SOA where a service broker acts as an intermediary between service requesters and service providers. A UDDI-based service registry is a specialized instance of a service broker. Under this configuration the UDDI registry serves as a broker where the service providers publish the definitions of the services they offer using WSDL, and the service requesters find information about the services available.

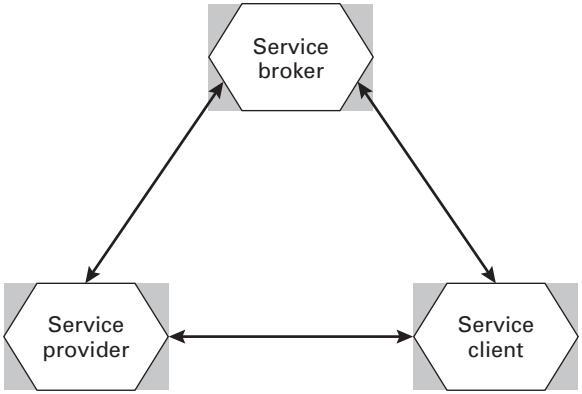

Figure 1.2
Service brokering.

1.3 Enterprise Service Bus

Though Web Services technologies are currently most commonly used in implementing SOAs, many other conventional programming languages and enterprise integration platforms may be used in an SOA as well [51]. In particular, any technology that complies with WSDL and communicates with XML messages can participate in an SOA. Such technologies include J2EE and message queues, such as IBM's WebSphere MQ.

Since clients and services may be developed by different providers using different technologies and conceptual designs, there may be technological mis-matches (e.g., they may use different communication protocols) and heterogeneities (e.g., message syntax and semantics) between them. Dealing with such technological mismatches and heterogeneity involves two basic approaches:

- Implement clients to conform exactly with the technology and conceptual model (e.g., semantics and syntax) of the services they may invoke.
- Insert an integration layer providing reusable communication and integration logic between the services and their clients.

The first approach requires the development of a service interface for each connection, resulting in point-to-point interconnections. Such point-to-point interconnection networks are hard to manage and maintain because they introduce a tighter coupling between clients and services. This coupling involves significant effort to harmonize transport protocols, document formats, interaction styles, etc. [35]. This approach results in hard-to-change clients and services, since any change to a service may impact all its clients. In addition, point-to-point integrations are complex and lack scalability. As the number of services and clients increases, they may quickly become unmanageable. To deal with this problem, existing Enterprise Application

Integration (EAI) middleware supports a variety of hub-and-spoke integration patterns [39]. This leaves the second approach as the more viable alternative.

The second approach introduces an integration layer that provides for interoperability between services and their clients. The Enterprise Service Bus (ESB) [48] [17] addresses the need to provide an integration infrastructure for Web Services and SOA. The ESB exhibits two prominent features [24]. First, it promotes loose coupling of the clients and services. Second, the ESB divides the integration logic into distinct, easily manageable pieces.

The ESB is an open, standards-based message bus designed to enable the implementation, deployment, and management of SOA-based solutions. To play this role the ESB provides the distributed processing, standards-based integration, and enterprise-class backbone required by the extended enterprise [24]. In particular, the ESB is designed to provide interoperability between large-grained applications and other components via standards-based adapters and interfaces. To accomplish this the ESB functions as both transport and transformation facilitator to allow distribution of these services over disparate systems and computing environments.

Conceptually, the ESB has evolved from the store-and-forward mechanism found in middleware products (e.g., message-oriented middleware), and combines conventional EAI technologies with Web services, XSLT [96], and orchestration and choreography technologies (e.g., BPEL, WS-CDL, and ebXML BPSS). Physically, an ESB provides an implementation backbone for an SOA. It establishes proper control of messaging and also supports the needs of security, policy, reliability, and accounting in an SOA. The ESB is responsible for controlling message flow and performing message translation between services. It facilitates pulling together applications and discrete integration components to create assemblies of services that form composite business processes, which in turn automate business functions in an enterprise.

Figure 1.3 depicts a simplified architecture of an ESB that integrates a J2EE application using JMS, a .NET application using a C# client, an MQ application that interfaces with legacy applications, and other external applications and data sources. An ESB, as portrayed in the upper and midde parts of figure 1.3, enables the more efficient value-added integration of a number of different application components by positioning them behind a service-oriented facade and by applying Web Services technology. In this figure, a distributed query engine, which is normally based on XQuery [10] or SQL, enables the creation of data services to abstract the complexity of underlying data sources. Portals in the upper part of figure 1.3 are user-facing ESB aggregation points of a variety resources represented as services.

Endpoints in the ESB, which are depicted as small rectangles in figure 1.3, provide abstraction of physical destination and connection information (such as TCP/IP hostnames and port numbers) transcending plumbing-level integration capabilities of traditional, tightly coupled, distributed software components. Endpoints allow services to communicate using logical connection names, which an ESB will map to actual physical network destinations at runtime. This destination independence offers the services that are connected to the ESB the ability to be upgraded, moved, or replaced without having to modify code and disrupt existing ESB applications. For instance, an existing invoicing service could easily be upgraded by a new service without disrupting the

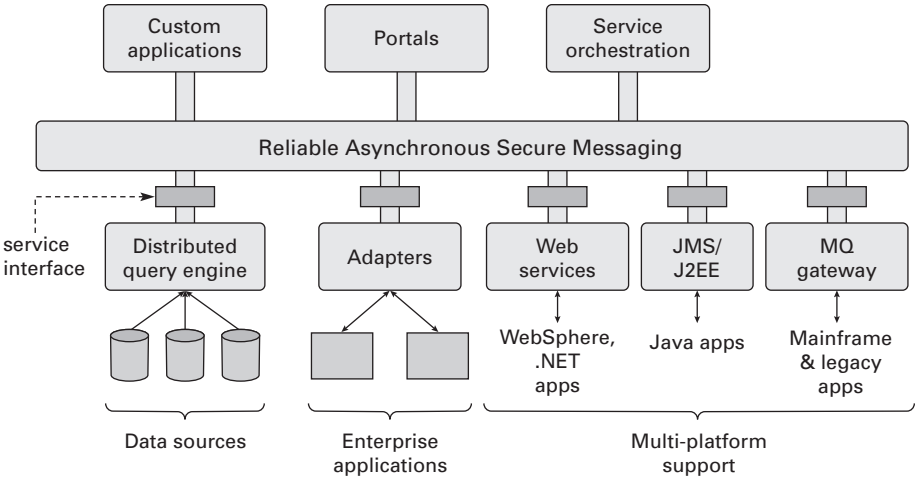

Figure 1.3
Enterprise Service Bus connecting diverse applications and technologies.

execution of other applications. Additionally, duplicate processes can be set up to handle fail-over if a service is not available. Endpoints rely on the asynchronous and highly reliable communication between service containers. They can be configured to use several levels of quality of service, which guarantees communication despite network failures and outages [17].

To successfully build and deploy a distributed Service-Oriented Architecture, the following four primary aspects need to be addressed:

1. *Service enablement* Each discrete application is exposed as a service.
2. *Service orchestration* Distributed services are configured and orchestrated in clearly specified processes.
3. *Deployment* As the SOA-based application is developed, completed services and processes must be transitioned from the testing to the production environment.
4. *Management* Services must be monitored, and their invocations and selection may need to be adjusted to better meet application-specific goals.

Services can be assembled using a variety of application development tools (e.g., Microsoft .NET, Borland JBuilder, or BEA WebLogic Workshop) which allow new or existing distributed applications to be exposed as Web services. Technologies such as J2EE Connector Architecture (JCA) may also be used to create services by integrating packaged applications (such as ERP systems), which would then be exposed as services.

To achieve its operational objectives, the ESB integration services such as connectivity and routing of messages based on business rules, data transformations, and application adapters [18].

These capabilities are themselves SOA-based in that they are spread out across the bus in a highly distributed fashion and are usually hosted in separately deployable service containers. This is a crucial difference from traditional integration brokers, which usually are highly centralized and monolithic [39]. The distributed nature of the ESB container model allows individual Web Services to plug into the ESB backbone as needed. This enables ESB containers to be highly decentralized and work together in a highly distributed fashion, though they are scaled independently from one another. This is illustrated in figure 1.3, where applications running on different platforms are decoupled from each other, and can be connected through the bus as logical endpoints that are exposed as Web services.

1.3.1 Event-Driven Architecture

In the enterprise context, business events (e.g., a customer order, the arrival of a shipment at a loading dock, or the payment of a bill) may affect the normal course of a business process at any point in time [36]. This implies that business processes cannot be designed a priori, assuming that events follow predetermined patterns, but must be defined more dynamically to permit process flows to be driven by asynchronous events. To support such applications, SOA must be enhanced into an *event-driven extension of* SOA, which we refer to as *Event-Driven Architecture* (EDA) [55] [17] [57]. Therefore, EDA is a service architecture that permits enterprises to implement an SOA while respecting the highly volatile nature of business events. An ESB requires that applications and event-driven Web Services be tied together in the context of an SOA in a loosely coupled fashion. EDA allows applications and Web Services to operate independent of each other while collectively supporting a business processes and functions [18].

In an ESB-enabled EDA, applications and services are treated as abstract service endpoints which can readily respond to asynchronous events [18]. EDA provides a means of abstracting away from the details of underlying service connectivity and protocols.

Services in this SOA variant are not required to understand protocol implementations or have any knowledge on routing of messages to other services. An event producer typically sends messages through an ESB, and then the ESB publishes the messages to the services that have subscribed to the events. The event itself encapsulates an activity, constituting a complete description of a specific action. To achieve its functionality, the ESB supports the established Web Services technologies, including, SOAP, WSDL, and BPEL, as well as emerging standards such as WS-ReliableMessaging [43] and WS-Notification [52].

As was noted in the previous section, in a brokered SOA (depicted in figure 1.2) the only dependency between the provider and the client of a service is the service contract that is typically described in WSDL and is advertised by a service broker. The dependency between the service provider and the service client is a runtime dependency, not a compile-time dependency. The client obtains and uses all the information it needs about the service at runtime. The service interfaces are discovered dynamically, and messages are constructed dynamically. The service consumer does not know the format of the request message, or the location of the service, until it needs the service.

Service contracts and other associated metadata (e.g., about policies and agreements [20]), lay the groundwork for enterprise SOAs that involve many clients operating with a complex, heterogeneous application infrastructure. However, many of today's SOA implementations are not that elaborate. In many cases, when small or medium enterprises implement an SOA, neither service interfaces in WSDL nor UDDI lookups are provided. This is often due either to the fact that the SOA in place provides for limited functionality or to the fact that sufficient security arrangements are not yet in place. In these cases an EDA provides a more lightweight, straightforward set of technologies to build and maintain the service abstraction for client applications [9].

To achieve less coupling between services and their clients, EDA requires event producers and consumers to be fully decoupled [9]. That is, event producers need no specific knowledge of event consumers, and vice versa. Therefore, there is no need for a service contract (e.g., a WSDL specification) that explicates the behavior of a service to the client. The only relationship between event consumers and producers is through the ESB, to which services and clients subscribe as event publishers and/or subscribers. Despite the focus of EDA on decoupling event consumers and producers, the event consumers may require metadata about the events they may receive and process. To address this need, event producers often organize events on the basis of some application-specific event taxonomy that is made available to (and in some cases is mutually agreed upon with) event consumers. Such taxonomies typically specify event types and other event metadata that describe published events that consumers can subscribe to, including the format of the event-related attributes and corresponding messages that may be exchanged between event producer and consumer services.

1.3.2 An ESB-Based Application Example

As an example of an ESB-based application, consider the simplified distributed procurement process in figure 1.4 that has been implemented using an ESB. The process is initiated when an "Inventory service" publishes a replenishment event (in figure 1.4 this is indicated by the dashed arrow between "Inventory service" and "Replenishment" service). This event is received by the subscribing "Replenishment" service as prescribed by EDA. On receipt of such an event, the "Replenishment" service starts the procurement process that follows the traditional SOA (e.g.,

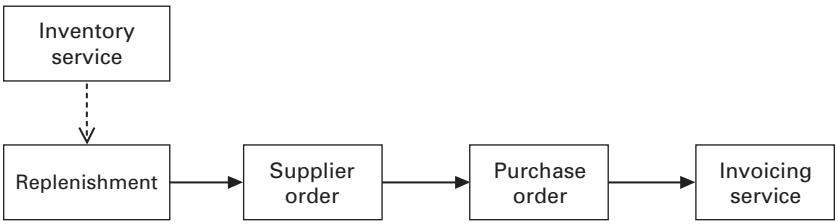

Figure 1.4
Simplified distributed procurement process.

service invocations are depicted as solid arrows). In particular, the "Replenishment" service first invokes the "Supplier order" service that chooses a supplier based on some criterion. Next, the purchase order is automatically generated by the "Purchase order" service (this service encapsulates an ERP purchasing module), and it is sent to the vendor of choice. Finally, this vendor uses an "Invoicing" service to bill the customer.

The services that are part of the procurement business process in figure 1.4 interact via an ESB that is depicted in figure 1.5. This ESB supports all aspects of SOA and EDA needed for implementing this service-based application. In particular, the ESB receives the published event and delivers it asynchronously to the subscribing "Replenishment" service (the event publisher is not depicted in figure 1.5). When the "Replenishment" service invokes the "Supplier order" service, the ESB transports the invocation message. Although this figure shows only a single "Supplier order" service as part of the inventory, a plethora of supplier services may exist. The

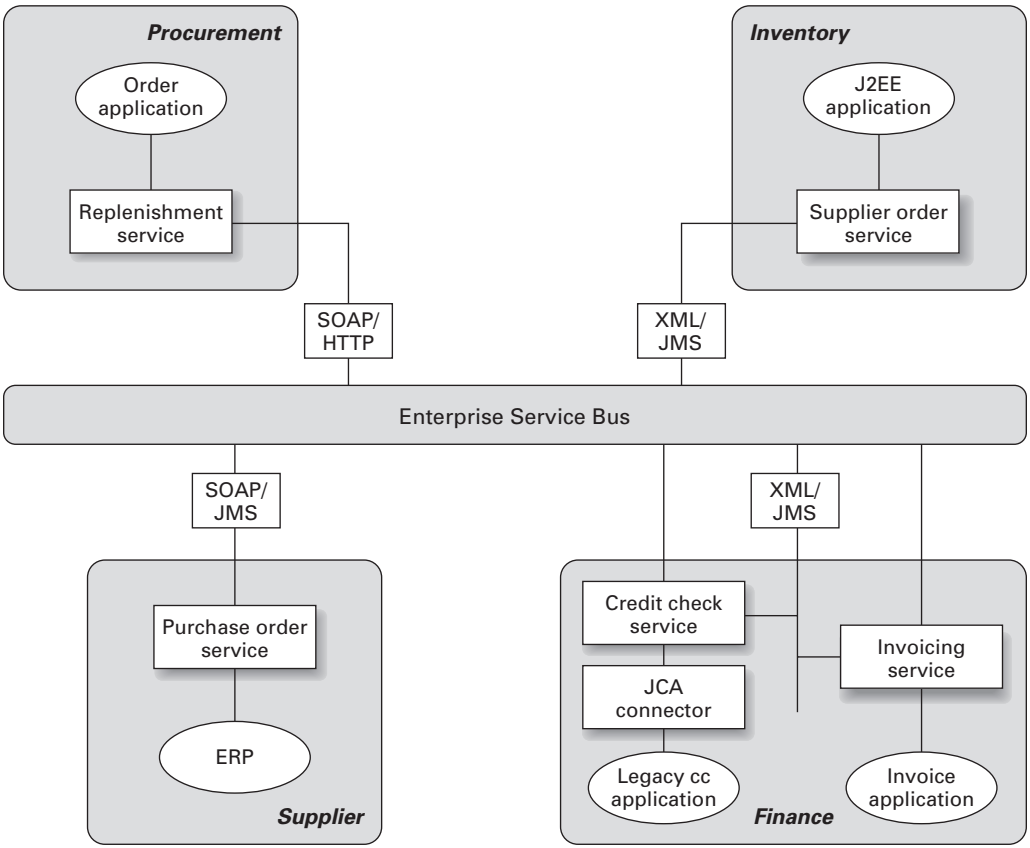

Figure 1.5
Enterprise Service Bus connecting remote services.

"Supplier order" service, which executes a remote Web Service at a chosen supplier to fulfill the order, generates its response in XML, but the message format is not understood by the "Purchase order" service. To deal with this and other heterogeneity problems, the message from the "Supplier order" service leverages the ESB's transformation service to convert the XML message that has been generated by the "Supplier order" service into a format that is accepted by the "Purchase order" service. Figure 1.5 also shows that legacy applications that are placed onto the ESB through JCA resource adapters employed by the "credit check" service.

The capabilities of ESB are discussed in more detail next.

1.3.3 Enterprise Service Bus Capabilities

Developing or adapting an application for SOA involves the following steps:

1. Creating service interfaces to existing or new functions, either directly or through the use of adapters;
2. routing and delivering service requests to the appropriate service provider; and
3. providing for safe substitution of one service implementation for another, without any effect to the clients of that service.

The last step requires not only that the service interfaces be specified as prescribed by SOA, but also that the SOA infrastructure used to develop the application allow the clients to invoke services regardless of the service location and the communication protocol involved. Such service routing and substitution are among the key capabilities of the ESB.

Additional capabilities/functional requirements for an ESB are described in the following paragraphs. We consider these capabilities as being necessary to support the functions of an effective ESB. Some of the functional capabilities described below have been discussed in other publications (e.g., [46] [15] [32] [17]).

Service Communication Capabilities
A critical ability of the ESB is to route service interactions through a variety of communication protocols, and to transform service interactions from one protocol to another where necessary. Other important aspects of an ESB implementation are the capacity to support service messaging models consistent with the SOA interfaces and the ability to transmit the required interaction context, such as security, transaction, or message correlation information.

Dynamic Connectivity Capabilities
Dynamic connectivity pertains to the ability to connect to services dynamically, without using a separate static API or proxy for each service. Most enterprise applications today operate on a static connectivity mode, requiring some static piece of code for each service. Dynamic service connectivity is a key capability for a successful ESB implementation. The dynamic connectivity API is the same, regardless of the service implementation mechanism (Web Services, JMS, EJB/RMI, etc.).

Endpoint Discovery with Quality of Service Capabilities
The ESB should support the discovery, selection, and binding to services. Increasingly these capabilities will be based on Web Services standards such as WSDL, SOAP, UDDI, and WS-PolicyFramework. As many network endpoints can implement the same service contract, it may be desirable for the client to select the best endpoint at runtime, rather than hard-coding endpoints at build time. In addition the ESB should be capable of supporting various qualities of service. Clients can query a Web Service, such as an organizational UDDI service, to discover the best service instance to use based on its QoS properties. Ideally, these capabilities should be controlled by declarative policies associated with the services involved, using a policy standard such as WS-Policy [26].

Integration Capabilities
To support SOA in a heterogeneous environment, the ESB needs to integrate with a variety of systems that do not directly support service-style interactions. These may include legacy systems, packaged applications, and other COTS components. When assessing the integration requirements for ESB, several levels of integration must be considered, including application, process, and information integration. Each of these imposes specific technical requirements that need to be addressed by a service-oriented integration solution [32] [24].

Application integration is concerned with building and evolving an integration backbone capability that enables fast assembly and redeployment of business software components. Such integration is an integral part of the assembly process that facilitates strategies which combine legacy applications, acquired packages, external application subscriptions, and newly built components. The ESB should focus on a service-based application integration solutions that deliver (1) applications composed of interchangeable parts that are designed to be adaptable to accommodate business and technology changes; (2) evolutionary application portfolios that protect investment and can respond rapidly to new requirements and business processes; and (3) integration of various platform and component technologies.

Process integration is concerned with the development of processes that combine other business processes, and integrate applications into business processes. Process-level integration at the level of ESB generally involve Enterprise Application Integration (EA), i.e., the integration of business processes and applications within the enterprise. It may also involve the integration of processes, not simply individual services, from other organizations external such as organizations participating in a supply chain or financial services that span multiple institutions.

Information integration [42] is the process of providing consistent access to all the data in the enterprise, by all the applications that require it, in whatever form they need it, without being restricted by the format, source, or location of the data. This may require adapters and other data transformation facilities, aggregation of services to merge and reconcile disparate data (e.g., to merge two profiles for the same customer), and data validation to ensure data consistency (e.g., the minimum computed income should be greater that zero).

Portal-based integration is concerned with how to fabricate a standard portal framework that provides efficient, uniform, and consistent presentation of complex business functions and processes to Web users. It permits the ESB to provide one face to Web users, resulting in consistent user experience and unified information delivery. This allows the underlying services and applications to remain distributed. Two complementary industry standards, JSR 168 and WSRP, are emerging in the portal space and can help in such integration efforts [44]. JSR 168 defines a standard way to develop portlets. It allows portlets to be interoperable across portal vendors. For example, portlets developed for BEA's WebLogic Portal can be interoperable with IBM Portal. This allows organizations to have a lower dependency on the portal product vendor. WSRP (Web Service for Remote Portals) allows remote portlets to be developed and used in a standard manner and facilitates federated portals. It combines the power of Web services and portal technologies and is fast becoming the major enabling technology for distributed portals in an enterprise. JSR 168 complements WSRP by dealing with local rather than distributed portlets. A portal page may have certain local portlets which are JSR 168-compliant and some remote, distributed portlets that are executed in a remote container. With JSR 168 and WSRP maturing, the possibility of a true EJB federated portal can become a reality.

It is important to note that all these integration levels must be considered when embarking on an ESB implementation. To be effective, ESB-based integration must rely on a methodology that facilitates reuse, eliminates redundancy, and simplifies integration, testing, deployment, and maintenance.

Message Transformation Capabilities

Legacy and new components that are integrated as services into the ESB typically have different expectations of messaging models and data formats. A major source of value in an ESB is that it shields any individual component from any knowledge of the implementation details of any other component. The ESB transformation services make it possible to ensure that messages and data received by any component are in the format it expects, thereby removing the need to change the sender or the receiver. The ESB plays a major role in transforming heterogeneous data and messages, including converting legacy data formats (e.g., a COBOL/VSAM application running on an OS/390 host) to XML, transforming XML to WSDL messages, and transtating input XML to a different XML format.

Reliable Messaging Capabilities

Reliable messaging refers to the ability to queue service request messages and ensure guaranteed delivery of these messages to their destinations. It also includes the ability to provide message delivery notification to the message senter/service requestor. Reliable messaging supports asynchronous store-and-forward delivery as well as guaranteed delivery capabilities. Primarily used for handling events, this capability is crucial for responding to clients in an asynchronous manner.

Topic/Content-Based Routing Capabilities
The ESB should be equipped with routing mechanisms to facilitate not only topic-based routing but also a more sophisticated content-based routing. *Topic-based* routing assumes that messages can be grouped into fixed, topical classes, so that subscribers can state interest in a topic and, as a consequence, receive messages associated with that topic [28]. *Content-based* routing permits the content of the message to determine its routing to different endpoints in the ESB. This requires subscriptions to be based on constraints involving properties or attributes of asynchonous messages and events. Therefore, content-based routing is particularly important for EDA.

Content-based routing is often implemented by XML messages and JMS or other message-oriented middleware or is based on emerging standards, such as WS-Notification.

WS-Notification defines a general, topic-based Web Service system for publish/subsribe interactions, which relies on the WS-Resource framework [30]. WS-Notification [52] is a family of related specifications that define a standard Web Services approach to notification using a topic-based publish/subscribe pattern. Its specification defines standard message exchanges to be implemented by service providers that wish to participate in notifications and standard message exchanges, thus allowing publication of messages from entities that are not themselves service providers. WS-Notification also allows expression of operational requirements for service providers and requesters that participate in notifications. It permits notification messages to be attached to WSDL PortTypes. The current WS-Notification specification provides support for both brokered and peer-to-peer publish/subscribe.

Security Capabilities
Enforcing security is a key success factor for ESB implementations. The ESB needs to provide security to service consumers and to integrate with the (potentially different) security models of the service providers. Both point-to-point (e.g., SSL encryption) and end-to-end security capabilities are required. The latter include federated authentication, which intercepts service requests and adds the appropriate username and credentials; validation of each service request and authorization to make sure that the sender has the appropriate privilege to access the service; and encryption/decryption of XML message content. To address these intricate security requirements, trust models, WS-Security [8], and other security-related standards have been developed.

Long-Running Process and Transaction Capabilities
If the ESB support, long-running business processes and transactions, such as those encountered in an online reservation system that interacts with the users as well as various service providers (airline ticketing, car reservation, hotel reservation, online payment such as paypal, etc) it is of vital importance that the ESB provide necessary transactional correctness and reliability guarantees. More specifically, the ESB must provide mechanisms that isolate the side effects of concurrent transactions from each other and must support recovery from technical and process failures. The challenge at hand is to ensure that complex transactions are handled in a highly

reliable manner, and ESB-supported transactions can roll back their processing to the original/prerequest state in the event of a failure.

Management and Monitoring Capabilities

Managing applications in a SOA environment is a serious challenge [7]. Examples of issues that need to be addressed include dynamic load balancing, fail-over when primary systems go down, and achieving topological or geographical affinity between the client and the service instance. Effective application management in an ESB requires a management framework that is consistent across an heterogeneous set of participating component systems and supports Service Level Agreements (SLAs). Enforcing SLAs requires the ability to select service providers dynamically, based on the quality of service they offer and the business value of individual transactions.

An additional requirement for a successful ESB implementation is the ability to monitor the health, capacity, and performance of services. Monitoring is the ability to track service activities that take place via the ESB and provide various metrics and statistics. Of particular significance is the ability to be able to spot problems and exceptions in the business processes and to move toward resolving them as soon as they occur. Process monitoring capabilities are currently provided by tool sets in platforms for developing, deploying, and managing service applications, such as the WebLogic Workshop.

Scalability Capabilities

With a widely distributed SOA, there is need to scale some of the services or the entire infrastructure to meet integration demands. For example, transformation services are typically very resource-intensive and may require multiple instances across two or more computing nodes. The loosely coupled nature of an SOA requires that the ESB use a decentralized model to provide a cost-effective solution that promotes flexibility in scaling any aspect of the integration network. A decentralized SOA enables independent scalability of individual services as well as of the communications infrastructure itself.

1.4 xSOA

A basic SOA (i.e., the architecture depicted in figure 1.2) implements concepts such as service registration, discovery, and service invocation. ESB requirements, however, suggest that the basic SOA needs to be extended to support capabilities such as service orchestration, monitoring, and management. These are addressed by the *extended SOA* (xSOA) [37] [41]. The xSOA is an attempt to streamline, group together, and logically structure the functional requirements of complex applications that make use of the service-oriented computing paradigm. The xSOA is a stratified service-based architecture. Its architectural layers, which are depicted in figure 1.6, embrace a multidimensional separation of concerns [40] in such a way that each layer defines a set of constructs, roles, and responsibilities, and relies on constructs of a lower layer to accomplish its mission. The logical separation of functionality is based on the need to separate basic

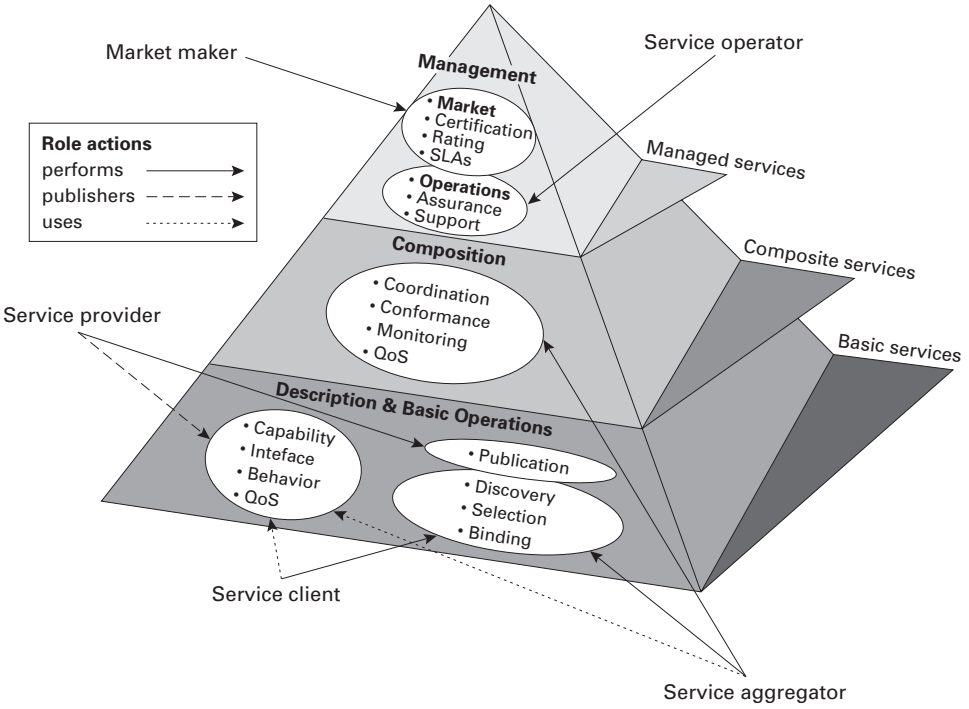

Figure 1.6
xSOA, an extended SOA.

service capabilities provided by the basic SOA (for example, building simple service-based applications) from more advanced service functionality needed for composing monitoring, and managing services. As shown in figure 1.6, the xSOA utilizes the basic SOA constructs as its foundational layer, and adds layers of service composition and management on top of it. The ESB middleware capabilities (communication, message routing and translation, service discovery, etc.) fall within the xSOA foundation layer. ESB capabilities that deal with service composition and management are in the composition and management layers of the xSOA. However, these layers include more advanced functionality than in an ESB.

In a typical service-based application employing the bottom layer of xSOA, a service provider hosts a network-accessible software module. The service provider defines a service description, and publishes it to a client or a registry through which a service description is advertised. The service client (requester) discovers a service (endpoint) and retrieves the service description directly from the service or from a registry (e.g. a UDDI repository). The client uses the service description to bind with the service provider and invoke the service. The service provider and service client roles are logical constructs, and a service may exhibit characteristics of both. For reasons of conceptual simplicity, in figure 1.6 we assume that service providers and aggregators

can act as service brokers and advertise the services they provide. The role actions in this figure also indicate that a service aggregator entails a special type of provider.

The service composition layer in the xSOA encompasses a more advanced role of service agreegator and corresponding functionality for the aggregation of multiple services into a single composite service. Resulting composite services may be used by (1) service aggregators as basic services in further service compositions, or (2) clients as applications/solutions. Service aggregators thus become service providers in the bottom layer of xSOA by publishing the service descriptions of the composite service they create.

The development of composite services at the composition layer of xSOA involves managing the following:

• *Metadata, standard terminology, and reference models* Web Services need to use metadata to describe what other endpoints need to know in order to interact with them. Metadata describing a service typically contain descriptions of the interfaces of a service, including vendor identifier, narrative description of the service, Internet address for messages, and format of request and response messages, as well as choreographic descriptions of the order of interactions. Service descriptions may range from simple service identifiers implying a mutually understood protocol to a complete description of the service interaction. Metadata describing services provide high-level semantic details regarding the structure and contents of the messages received and sent by Web Services, message operations, concrete network protocols, and endpoint addresses used by Web Services. They also describe abstractly the capabilities, requirements, and general characteristics of Web Services and how they can interoperate with other services. Web Service metadata need to be accompanied with standard terminology to address business terminology fluctuations and reference models, such as RosettaNet PIPS [56], to allow applications to define data and processes that are meaningful not only to their own businesses but also to their business partners, while maintaining interoperability (at the semantic level) with other business applications. The purpose of combining metadata, standard terminology, and reference models is to enable business processes to capture and convey their intended meaning and exchange requirements, identifying (among other things) the meaning, timing, sequence, and purpose of each business collaboration and associated information exchange.

• *Conformance* This involves managing service, behavior, and semantics conformance. Service conformance achieves the following: (1) ensures the integrity of a composite service by matching its operations and parameter types with those of its constituent component services, (2) imposes constraints on the component services (e.g., to enforce business rules), and (3) performs data fusion activities. Service conformance has three parts: typing, behavioral, and semantic conformance. Typing (syntactic) conformance is performed at the data-typing level by using principles such as type safeness, covariance, and contravariance for signature matching [53]. Behavioral conformance ensures the correctness of logical relationships between component operations that need to be blended into higher-level operations. Behavioral conformance guarantees that composite operations do not lead to spurious results and that the overall process

behaves in a correct and unambiguous manner. Finally, semantic conformance ensures that services and operations are annotated with domain-specific semantic properties (descriptions) so that they preserve their meaning when they are composed and can be formally validated. Service conformance is a research topic [53]. Concrete solutions exist only for typing conformance [38] [16], as they are based on conformance techniques for programming languages such as Eiffel or Java.

• *Coordination* This involves managing the execution order and dataflow between the component service (e.g., by specifying workflow processes and using a workflow engine for runtime control of service execution).

• *Monitoring* This allows monitoring events or information produced by the component services, monitoring instances of business processes, viewing process instance statistics, and suspending and resuming or terminating selected process instances. Of particular significance is the ability to be able to spot problems and exceptions in the business processes and be able to resolve them as soon as they occur. Process monitoring capabilities are currently provided by tools in platforms for developing, deploying, and managing service applications (e.g., BEA's WebLogic and Vitria's BusinessWare).

• *Policy compliance* This involves the management of the compliance of service compositions to policies (e.g., for security, information assurance, and QoS). In particular, policies [49] may be used to manage a system or organize the interaction between Web services [4]. 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 what kind of security tokens the service is capable of processing, and which one it expects. 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. Finally, the client must determine when to encrypt the messages, which algorithm to use, and how to exchange a shared key with the service. Composing services without understanding these details and complying to the appropriate policies may lead to erroneous results.

Standards such BPEL and WS-Choreography [14] that operate at the service composition layer in xSOA enable the creation of large service-based applications that allow two companies to conduct business in an automated fashion. We expect to see much larger service collaborations spanning entire industry groups and other complex business relationships. These developments necessitate the use of tools and utilities that provide insights into the health of systems that implement Web Services and into the status and behavior patterns of loosely coupled applications. A related methodology is essential for leveraging a management framework for a production-quality, service-based infrastructure and applications. The rationale is very similar to the situation in traditional distributed computing environments, where systems administrators rely on programs/tools/utilities to make certain that a distributed computing environment operates reliably and efficiently.

Managing loosely coupled applications in an SOA inherently entails even more challenging requirements. Failure or change of a single application component can bring down numerous interdependent enterprise applications. The addition of new applications or components can overload existing components, causing unexpected degradation or failure of seemingly unrelated systems. Application performance depends on the combined performance of cooperating components and their interactions. To counter such situations, enterprises need to constantly monitor the health of their applications. The performance should be in tune at all times and under all load conditions.

Managing critical Web service-based applications requires sophisticated administration and management capabilities that are supported across an increasingly heterogeneous set of participating distributed component systems and provide complex aggregate (cross-component) service-level agreement enforcement and dynamic resource provisioning. Such capabilities are provided by the xSOA service management layer.

We could define Services management as the functionality required for discovering the existence, availability, performance, health, usage, control, configuration, life cycle support, and maintenance, of Web Services or business processes in the context of a SOA. Service management encompasses the control and monitoring of SOA-based applications throughout their life cycle [22]. It spans a range of activities from installation and configuration to collecting metrics and tuning to ensure responsive service execution. The management layer in xSOA requires that services need to be managed. In fact, the very same well-defined structures and standards that form the basis for Web Services also provide the foundation for managing and monitoring communications between services and their underlying resources, across numerous vendors, platforms, technologies, and topologies.

Service management includes many interrelated functions [25]. The most prominent functions of service management are summarized in the following categoriers:

1. *Service-Level Agreement (SLA) management* This includes QoS (e.g., sustainable network bandwidth with priority messaging service) [31]; service reporting (e.g., acceptable system response time); and service metering.

2. *Auditing, monitoring, and troubleshooting* This includes providing service performance and utilization statistics, measurement of transaction arrival rates and response times, measurement of transaction loads (number of bytes per incoming and outgoing transaction), load balancing across servers, measuring the health of services, and troubleshooting.

3. *Dynamic service provisioning* This includes provisioning resources, dynamic allocation/deallocation of hardware, installation/deinstallation of software on demand-based changing workloads, ensuring SLAs, and reliable SOAP messaging delivery.

4. *Service life cycle/state management* This includes exposing the current state of a service and permitting life cycle management, including the ability to start and stop a service, make specific configuration changes to a deployed Web Service, support different versions of Web

services, and notify the clients of a service about a change or impending change to the service interface.

5. *Scalability/extensibility* The Web Services support environment should be extensible and must permit discovery of supported management functionality in a given instantiation.

To manage critical applications, cross-organizational collaborations, and other complex business relationships the xSOA service management layer is divided into two complementary categories [37] [41]:

1. Service operations management. This manages the service platform and the deployment of services, and, more importantly, monitors the correctness and overall functionality of aggregated/orchestrated services.
2. Service market management. This typically supports integrated supply chain functions and provides a comprehensive range of services supporting an industry/trade, including providing services for business transaction negotiation, financial settlement, service certification and quality assurance, service rating, etc.

The xSOA'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. xSOA's service operations management typically gathers information about the managed service platform, Web Services and business processes, and managed resource status and performance. It also supports specific service 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 help assess 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 a decision condition is reached. xSOA refers to the role responsible for performing such operation management functions as the service operator.

It is increasingly important for service operators to define and support active capabilities versus traditional passive capabilities [29]. 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 platform should be able to take corrective action. This action could take the form of rerouting requests to a backup service that is less loaded, or provisioning a new application server with an instance of the software providing the service if no backup is currently running and available.

Finally, service operations management should provide tools for the management of running processes that are comparable with those provided by BPM tools. Management in this care taker the form of real-time and historical reports, and the triggering of actions. For example, deviations from key performance indicator 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.

Another aim of xSOA's service management layer is to provide support for specific markets by permitting the development of open service marketplaces. Currently, there exist several vertical industry marketplaces, such as those for the semiconductor, automotive, travel, and financial services industries. Their purpose is to create opportunities for buyers and sellers to meet and conduct business electronically, or to aggregate service supply/demand by offering added-value services and group buying power (just like a cooperative). The scope of such a service marketplace is 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. 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 markets must offer a comprehensive range of services supporting industry/trade, including services that provide business transaction negotiation and facilitation [12], financial settlement, service certification and quality assurance, rating services, service metrics (such as number of current service requesters, average turnaround time), and manage the negotiation and enforcement of SLAs. The xSOA market management functionality as illustrated in figure 1.6 is aimed to support these open service market functions.

xSOA Service markets introduce the role of a market maker. A market maker is a trusted third party or consortium of organizations that brings the suppliers and vendors together. Essentially, a service market maker is a special kind of service aggregator that has added responsibilities, such as issuing certificates, maintaining industry standard information, introducing new standards, and endorsing service providers/aggregators. The market maker assumes the responsibility for service market administration and performs maintenance tasks to ensure the administration is open and fair for business and, in general, provides facilities for the design and delivery of integrated service offerings that meet specific business needs and conform to industry standards.

1.5 Chapter Summary and Remaining Chapters in this Book

Modern enterprises need to streamline both internal and cross-enterprise business processes by integrating new, legacy, and home-grown applications. This requires an agile approach that allows enterprise business services (those offered to customers and partners) to be easily assembled from a collection of smaller, more fundamental business services. This challenge in automated business integration has driven major advances in technology within the integration software space. As a result the SOA has emerged to address the roles of service requesters, service providers, and service brokers, and to foster loosely coupled, standards-based, and protocol-independent distributed computing that offers solutions for achieving the desired business integration and mapping IT implementations more closely to the overall business process flow.

Combining the SOA paradigm with event-driven processing lays the foundation for EDA. Both SOA and EDA can be effectively supported by an ESB that is capable of combining various

conventional distributed computing, middleware, BPM, and EAI technologies. Therefore ESB offers a unified backbone on which enterprise services can be advertised, composed, planned, executed, and monitored.

To capture essential ESB functions that include capabilities such as service orchestration, "intelligent" routing, provisioning, and integrity and security of messages, as well as service management, the conventional SOA was extended into the extended SOA (xSOA).

In writing this chapter we have intentionally exposed the complexity of SOA-based application development in environments where SOA-based applications are not developed entirely from new functionality. Though this sacrifices some of the clarity in SOA, it faithfully represents the issues in the development of many real SOA-based applications.

The remaining chapters in this book cover topics related to SOA, EDA, and xSOA, as well as engineering aspects of SOA-based applications. In particular, the book includes chapters on modeling of SOA-based applications, SOA architecture design, business process management, transactions, QoS and service agreements, service requirements engineering, reuse, and adaptation. The remaining chapters are outlined below.

Chapter 2: Conceptual Modeling of Service-Driven Applications

In this chapter, Boualem Benatallah, Fabio Casati, Woralak Kongdenfha, Halvard Skogsrud, and Farouk Toumani present a model-driven methodology for Web Services development and management. In particular, the proposed methodology is based on (1) UML and high-level notations to model key service abstractions, (2) operators and algebras for comparing and transforming such models, and (3) a CASE tool, partially implemented, that manages the entire service development life cycle. The proposed methodology focuses on the aspects of Web Services mentioned above, and on the relationship among them, and between them and the Web Service implementation.

The motivation behind this effort comes from the observation that there has been little concern so far regarding the potential complexity of service-based applications development, despite initial results in supporting interoperability in terms of a concerted standardization effort. There is no framework for helping Web Services beginners, or even developers, wander through the maze of available specifications and their usefulness for developing service artifacts and abstractions (e.g., interfaces, business protocols, composition models, and policies). In addition, there has been little support for model-driven development of Web Services, which is very useful to bridge the gap between requirements and the subsequent phases of the service artifacts development process.

Chapter 3: Realizing Service-Oriented Architectures with Web Services

In this chapter Jim Webber and Savas Parastatidis present a message-oriented architectural paradigm for Web Services, consisting of three distinct views that decouple the architecture layer (service-oriented) from the protocol layer (message-oriented) and the implementation layer (event-driven). The importance of XML messages and message exchange patterns are empha-

sized across all three views. This approach is illustrated by desribing the construction of a simple instant messaging application which highlights protocol design and implementation issues. In addition to setting out the architectural layers of message-oriented Web services, a set of architectural and implementation guidelines is presented. These guidelines show how to avoid common software pitfalls by adhering to a number of deliberately simple design principles which encompass architecture, protocol, and implementation.

Chapter 4: Service-Oriented Support for Dynamic Interorganizational Business Process Management

Paul Grefen's chapter analyzes requirements for the support of business processes and puts these in the context of the existing SOC technology. The chapter describes an application of SOC technology providing dedicated support for dynamic business process management across the boundaries of organizations. The combination of SOC technology and workflow management (WFM) technology provides the basis for full-fledged dynamic interorganizational business process management. Grefen concludes that the current state of the art does not yet provide an integrated solution, but that many capabilities are available or under development.

Chapter 5: Data and Process Mediation in Semantic Web Services

This chapter by Adrian Mocan, Emilia Cimpian, and Christoph Bussler views the Web as a highly distributed and heterogeneous source of data and information. In this environment, Web Services extend the heterogeneity problems from the data level to the behavior level of business logic, message exchange protocol, and Web Service invocation. The chapter first indentifies the need for mediator systems that are able to cope with these heterogeneity problems and offer the means for reconciliation, integration, and interoperability. Next, this chapter presents an overview of mediator systems, analyzes existing and future trends in this area, and describes the mediation architecture of the Web Service Execution Environment (WSMX).

The chapter addresses both data and process mediation. It provides an insight into the first topic together with a survey of the multitude of existing approaches in data mediation. The chapter also explores and characterizes the largely unexplored topic of process mediation that must be part of the mediation solution for the Semantic Web and Semantic Web Services.

Chapter 6: Toward Configurable QoS-Aware Web Services Infrastructure

L. Bahler, F. Caruso, C. Chung, B. Falchuk, and J. Micallef have a long experience in the development of mission-critical enterprise systems, such as telecommunications network management systems. Such systems have stringent nonfunctional requirements for reliability, performance, availability, security, and scalability. Web Services technologies that address these essential quality of service (QoS) aspects are still in their infancy, with several emerging and often overlapping specifications that address only a fraction of QoS issues. This chapter addresses the challenges of designing Web Services with QoS requirements for mission-critical operations. In particular, this chapter provides an analysis and design methodology for Web Services that

can be transparently deployed on different transports. It focuses on the QoS requirements for the message exchange to accomplish a business service—and decribes an adaptive Web Services gateway that can be configured to provide security (and other capabilities). The chapter also advocates the use of Semantic Web technologies to support and automate deployment configuration that satisfies the solution QoS requirements for a specific technology environment.

Chapter 7: Configurable QoS Computation and Policing in Dynamic Web Service Selection

In this chapter, Anne HH Ngu, Yutu Liu, Liangzhao Zeng, and Quan Z. Sheng cover QoS problems and solutions for supporting rapid and dynamic composition of Web Services. In a dynamic Web Service composition paradigm, Web Services that meet requesters' functional requirements must be located and bounded dynamically from a large and constantly changing number of service providers. To enable quality-driven Web service selection, an open, fair, dynamic, and secure framework is needed to evaluate the QoS of a vast number of Web services. The computational fairness and enforcement of the QoS of component Web Services should have minimal overhead, yet be able to achieve sufficient trust by both service requesters and providers. This chapter presents an open, fair, and dynamic QoS computation model for Web Services selection through implementation of and experimentation with a QoS registry in a hypothetical phone service provisioning marketplace application.

Chapter 8: WS-Agreement Concepts and Use: Agreement-Based, Service-Oriented Architectures

This chapter by Heiko Ludwig outlines the concept of agreement-driven SOA, explains the elements of the WS-Agreement specification, and discusses conceptual and pragmatic issues of implementing an agreement-driven SOA based on WS-Agreement. Agreements, such as Service Level Agreements (SLAs), are typically used to define the specifics of a service delivered by a provider to a particular customer. They include the service provider's obligations in terms of which services at which quality, the modalities of service delivery, and the quantity (i.e., the capacity) of the service to be delivered. Agreements also define what is expected of the service customer, typically the financial compensation and the terms of use. The chapter describes the specifications of WS-Agreement that are defined by the Grid Resource Allocation Agreement Protocol (GRAAP) Working Group of the Global Grid Forum (GGF) Such agreement specifications enable an organization to dynamically establish an SLA in a formal, machine-interpretable representation as part of an SOA. The chapter discusses an XML-based syntax for agreements and agreement templates, a simple agreement creation protocol, and an interface to monitor the state of an agreement.

Chapter 9: Transaction Support for Web Services

This chapter by Mark Little provides an overview of various transaction models and specifications that have been proposed as standards for transactional composition of Web services. In

particular, the chapter provides a tutorial of ACID transactions and the Business Transactions Protocol (BTP) proposed by the Organization for Advancement of Structured Information Systems (OASIS). It explains the Web Services Coordination (WS-C) protocol with its Web Services Atomic Transaction (WS-AT) and the Web Services Business Activity (WS-BA) specifications, and provides an overview of the the Composite Application Framework (WS-CAF) specification. Finally, the chapter compares these proposed standards by characterizing and comparing the transactional guarantees they provide.

Chapter 10: Transactional Web Services

This chapter by Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, Jonas Grundler, and Olaf Zimmermann addresses the problem of transactional coordination in Service-Oriented Computing (SOC). The chapter advocates the use of declarative policy assertions to advertise and match support for different transaction models, and to define transactional semantics of Web Services compositions. It presents concrete, protocol-specific policies that apply to relevant Web Services specifications. In particular, the chapter focuses on the Web Services Coordination (WS-Coordination) specification that defines an extensible framework that can be used to implement different coordination models for Web Services. These include traditional atomic transactions and long-running business transactions specified using the Web Services Atomic Transaction (WS-AT) and the Web Services Business Activity (WS-BA) specifications. The chapter presents a policy-based approach that extends BPEL with coordination semantics and uses policies to drive and configure corresponding middleware systems to support transactional service compositions in an SOC environment.

Chapter 11: Service Componentization: Toward Service Reuse and Specialization

Bart Orriens and Jian Yang introduce the concept of service component to facilitate the idea of Web Service component reuse, specialization, and extension, and discuss why the inheritance concepts developed by object-oriented programming language research cannot be applied directly to service component inheritance, but must be modified. The chapter introduces service components as a packaging mechanism for developing Web-based distributed applications in terms of combining existing (published) Web services. Generally speaking, a service component can range from a small fragment of a business process to an entire complex business process. A component's interface specification will be used when applications and other service compositions are built upon it. The chapter illustrates that service components have a recursive nature, in that they can be composed of published Web Services while in turn they themselves are also considered to be Web Services (albeit complex in nature). Finally, The chapter describes how once a service component is defined it can be reused, specialized, and extended.

Chapter 12: Requirements Engineering Techniques for Web Services

In this chapter, Jaap Gordijn, Pascal van Eck, and Roel Wieringa focus on creating a shared understanding of Web Services, and on analyzing whether the each Web Service is commercially

viable and technically feasible. Achieving these is complicated when many different stakeholders representing different enterprises and different interests are involved. The chapter presents an approach, based on requirements engineering techniques: (1) to understand and analyze the Web Service, and (2) to develop a blueprint for a Web Service-based implementation. The chapter takes a multiple perspective approach that includes a commercial value perspective, a process perspective, and an information systems perspective.

Chapter 13: Web Service Adaptation

In the last chapter of this book, Barbara Pernici and Pierluigi Plebani discuss Web Service adaptation. The solutions presented in the chapter are based on the results of the MAIS (Multichannel Adaptive Information System) project, in which the Web Service paradigm has been exploited in an adaptive way, studying (1) a set of models and methodologies which allow service provisioning through different channels, and (2) techniques to provide flexible services which are aware of the provisioning context. For this goal, the chapter proposes a Web Services description that takes into account the service provisioning channel as an orthogonal dimension which is independent from the user and the provider. The chapter considers Web Services to be both stand-alone and running inside a process. To offer a way to evaluate when and in what way the adaptation can take place, the chapter describes a quality model of Web Services. In particular, this quality model allows the client to know how the quality varies not only between different Web Services but also between the channels through which the same Web service can be invoked. In this way, the client has information with which to identify the best Web Service and the best channel to use.

References

[1] *Universal Description, Discovery, and Integration (UDDI)*. Technical report. September 2000. http://www.uddi.org.

[2] Ali Arsanjani. Introduction to the special issue on developing and integrating enterprise components and services. *Communications of the ACM*, 45(10):30–34 (October 2002).

[3] Anjali Anagol-Subbarao. *J2EE Web Services on BEA WebLogic 1/e*. Prentice Hall PTR, Upper Saddle River, N.J., 2004.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, New York, 2004.

[5] T. Andrews et al. *Business Process Execution Language (BPEL), Version 1.1*. Technical report. BEA Systems, IBM, Microsoft, SAP, and Siebel Systems, May 2003.

[6] A. Arkin. *Business Process Modeling Language (BPML)*. Last call draft report. BPMI.org, November 2002.

[7] A. Arora et al. *Web Services for Management (WS-Management)*. Technical report. Advanced Micro Devices, Dell, Intel, Microsoft, and Sun Microsystems, October 2004.

[8] B. Atkinson et al. *Web Services Security (WS-Security)*. Technical report. Microsoft, IBM, and VeriSign, April 2002.

[9] J. Bloomberg. *Events vs. services: The real story*. ZapThink white paper, October 2004. Available at www.zapthink.com.

[10] Scott Boag et al. *XQuery 1.0: An XML Query Language*. Working draft, technical report. W3C, April 2005.

[11] D. Box et al. *Simple Object Access Protocol (SOAP) 1.1.* W3C note. May 2000. http://www.w3.org/TR/SOAP.

[12] Tung Bui and Alexandre Gachet. Web services for negotiation and bargaining in electronic markets: Design requirements and implementation framework. In *Proceedings of the 38th Hawaii International Conference on System Sciences.* IEEE, Waikoloa, Hawiaii 2005.

[13] S. Burbeck. *The tao of e-business services: The evolution of Web applications into service-oriented components with Web services*. IBM DeveloperWorks, October 1, 2000. http://www-106.ibm.com/developerworks/Webservices/library/ws-tao.

[14] David Burdett and Nickolas Kavantzas. *WS-Choreography Model Overview*. Working draft. W3C, March 2004.

[15] A. Candadai. A dynamic implementation framework for SOA-based applications. *Web Logic Developers Journal*, pp. 6–8 (September/October 2004).

[16] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522 (1985).

[17] D. Chappell. *Enterprise Service Bus*. O'Reilly, Sebastopol, Calif., 2004.

[18] D. Chappell. ESB myth busters: 10 Enterprise Service Bus myths debunked. Clarity of definition for a growing phenomenon. *Web Services Journal*, pp. 22–26 (February 2005).

[19] M. Colan. *Service-Oriented Architecture expands the vision of Web services, Part 2*. IBM DeveloperWorks. June 2004.

[20] A. Dan et al. Web services on demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1): 136–158 (March 2004).

[21] Arulazi Dhesiaseelan and Venkatavaradan Ragunathan. Web Services Container Reference Architecture (WSCRA). In *Proceedings of the International Conference on Web Services*, pp. 805–806. IEEE, 2004.

[22] A. Lazovik et al. Associating assertions with business processes and monitoring their execution. In *Proceedings of the Second International Conference on Service Oriented Computing*. ACM Press, New York, 2004.

[23] D. Booth et al. *Web Services Architecture*. 3WC Working Group note, 2003/2004. http://www.w3.org/TR/2004/NOTE-arch-20040211.

[24] M. Keen et al. Patterns: Implementing an SOA using an Enterprise Service Bus. *IBM Redbooks*, July 25, 2004.

[25] Nicolas Catania et al. *Web Services Management Framework—Overview, Version 2.0.* Technical report. HP, July 2003.

[26] Siddharth Bajaj et al. *Web Services Policy framework (WS-Policy)*. Technical report. BEA Systems, IBM, Microsoft, SAP, Sonic Software, and VeriSign, September 2004.

[27] Stephen Farrell et al. *Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML), V1.1*. Committee specification. OASIS, July 2003.

[28] B. Mukherjee et al. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 262–272. IEEE Computer Society, Washington, D.C., 1999.

[29] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computer era. *IBM Systems Journal*, 42(1):5–18 (2003).

[30] Steve Graham et al., eds. *Web Services Resource (WS-Resource)*, *Version 1.2.* Working draft 03, technical report. OASIS, March 2005.

[31] R. Hauck and H. Reiser. Monitoring quality of service across organizational boundaries. In *Trends in Distributed Systems: Towards a Universal Service Market. Proceedings of the Third International IFIP/GI Working Conference*. LNCS 1890. Springer, New York, 2000.

[32] K. Channabasavaiah, K. Holley, and E. M. Tuggle, Jr. *Migrating to a Service-Oriented Architecture, Part 1.* IBM DeveloperWorks, December 2003.

[33] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall Professional Technical References, Indianapolis, Ind., 2005.

[34] Heather Kreger. Fulfilling the Web services promise. *Communications of the ACM*, 46(6):29ff. (2003).

[35] D. Linthicum. *Next Generation Application Integration: From Simple Information to Web Services*. Addison-Wesley, Boston, 2003.

[36] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston, 2002.

[37] M. P. Papazoglou and D. Georgakopoulos. Introduction to a special issue on Service-Oriented Computing. *Communications of the ACM*, 46(10): 24–28 (October 2003).

[38] B. Meyer. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall Professional Technical Reference, Upper Saddle River, N.J., 1997.

[39] M. P. Papazoglou and P. M. A. Ribbers. *e-Business: Organizational and Technical Foundations*. Wiley, Hoboken, N.J., 2006.

[40] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.* Kluwer 2000.

[41] M. P. Papazoglou. Extending the Service Oriented Architecture. *Business Integration Journal*, February 2005.

[42] Christine Parent and Stefano Spaccapietra. Issues and approaches of database integration. *Communications of the ACM*, 41(5):166–178 (1998).

[43] Kazunori Iwasa, ed. WS-reliability, 1.1. Committee draft 1.086. OASIS, Web Services Reliable Messaging TC, August 2004. http://www.oasis-open.org/committees/wsrm/documents/specs/(tbd).

[44] R. Rana and S. Kumar. Service on demand portals: A primer on federated portals. *Web Logic Developers Journal*, pp. 22–24 (September/October 2004).

[45] Ueli Wahli et al. *Websphere Version 5.1 Application Developer 5.1.1 Web Services Handbook*. IBM Redbook, 2004.

[46] R. Robinson. *Understand Enterprise Service Bus Scenarios and Solutions in Service-Oriented Architecture*. IBM DeveloperWorks, 2004.

[47] S. Anderson et al. *Web Services Trust language (WS-Trust)*. Public draft release. Actional, BEA Systems, Computer Associates, IBM, Layer 7 Technologies, Microsoft, Oblix, OpenNetwork Technologies, Ping Identity, Reactivity, RSA Security, and VeriSign, February 2005.

[48] R. Schulte. *Predicts 2003: Enterprise Service Buses Emerge*. Report. Gartner, December 2002.

[49] M. Sloman. Policy driven management of distributed systems. *Journal of Network and Systems Management*, 2:333–360 (1994).

[50] D. Smith. Web services enable Service Oriented and Event-Driven Architectures. *Business Integration Journal*, pp. 12–13 (May 2004).

[51] Michael Stal. Web Services: Beyond component-based computing. *Communications of the ACM*, 45(10):71–76 (October 2002).

[52] Steve Graham, Peter Niblett, et al. *Web Services Base Notification*, *Version 1.0.* Akamai Technologies, Computer Associates, Fujitsu, Hewlett-Packard Development Company, IBM, SAP, Sonic Software, University of Chicago, and Tibco Software. Inc., 2004.

[53] W. J. van den Heuvel. *Aligning Modern Business Processes and Legacy Systems: A Component-Bases Perspective*. MIT Press, Cambridge, Mass., 2007.

[54] W3C. *XSL Transformations (XSLT)*, *version 2.0.* W3C working draft, technical report. April 2005.

[55] Jian Yang. Web Service componentization. *Communications of the ACM*, 46(10):35–40 (October 2003).

[56] P. Yendluri. RosettaNet Implementation Framework (RNIF), *version 2.0.* Technical report. RosettaNet, 2000.

[57] D. Baker, D. Georgakopoulos, M. Nodine, and A. Cichocki. From events to awareness. In *Proceedings of the First International Workshop on Event-driven Architecture, Processing, and Systems* (EDA-PS 2006); 2006 IEEE Services Computing Workshops (SCW 2006), IEEE, Chicago, September, 2006.