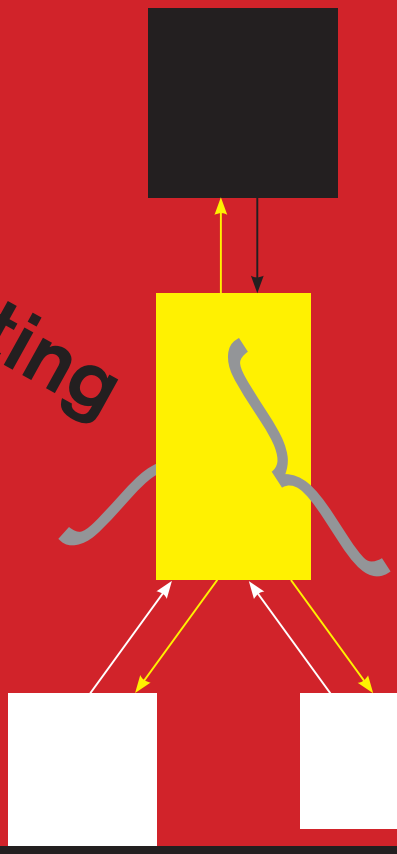
Service-Oriented Computing



edited by
Dimitrios Georgakopoulos
and Michael P. Papazoglou

Service-Oriented Computing

Cooperative Information Systems
Michael P. Papazoglou, Joachim W. Schmidt, and John Mylopoulos, editors

Advances in Object-Oriented Data Modeling
Michael P. Papazoglou, Stefano Spaccapietra, and Zahir Tari, editors

Workflow Management: Models, Methods, and Systems
Wil van der Aalst and Kees Max van Hee

A Semantic Web Primer
Grigoris Antoniou and Frank van Harmelen

Meta-Modeling for Method Engineering
Manfred Jeusfeld, Matthias Jarke, and John Mylopoulos, editors

Aligning Modern Business Processes and Legacy Systems: A Component-Based Perspective Willem-Jan van den Heuvel

A Semantic Web Primer, *Second Edition*
Grigoris Antoniou and Frank van Harmelen

Service-Oriented Computing
Dimitrios Georgakopoulos and Michael P. Papazoglou, editors

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

Contents

Service-Oriented Computing

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

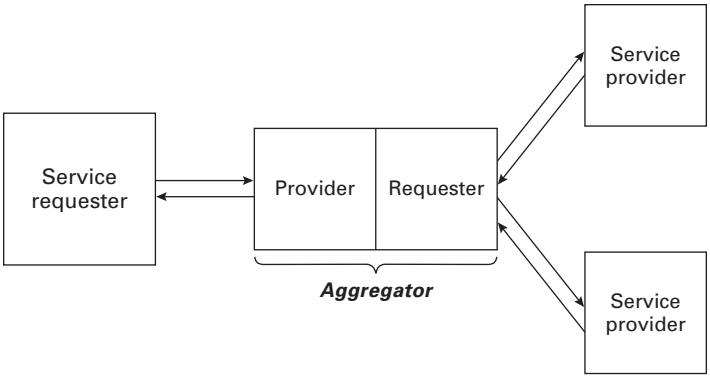


Figure 1.1
The role of the service aggregator.

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.

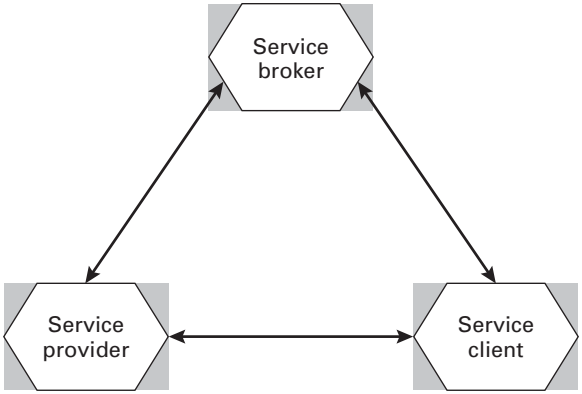


Figure 1.2
Service brokering.

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

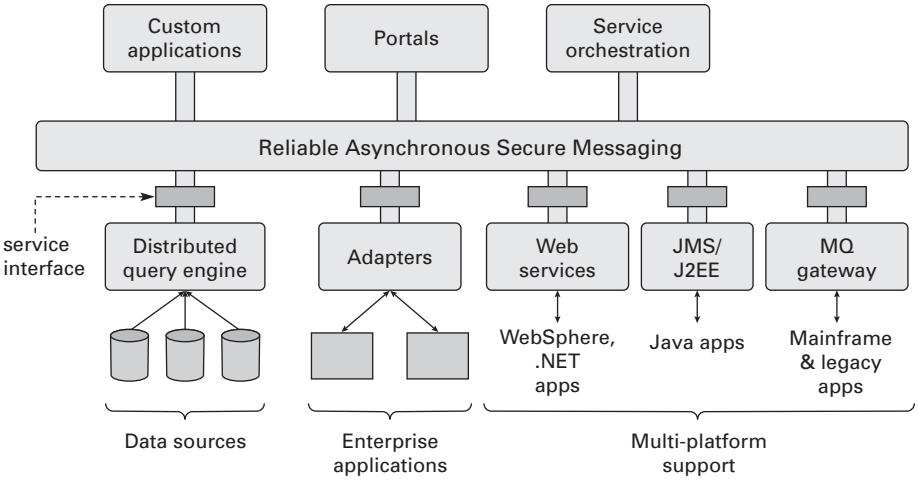


Figure 1.3
Enterprise Service Bus connecting diverse applications and technologies.

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.,

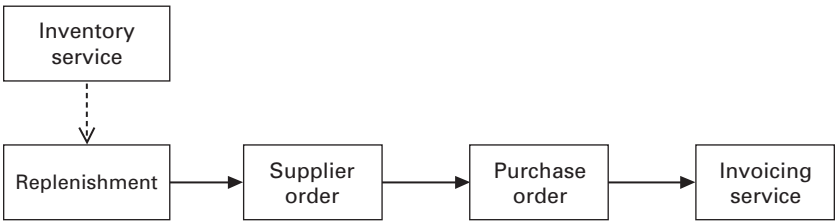


Figure 1.4
Simplified distributed procurement process.

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

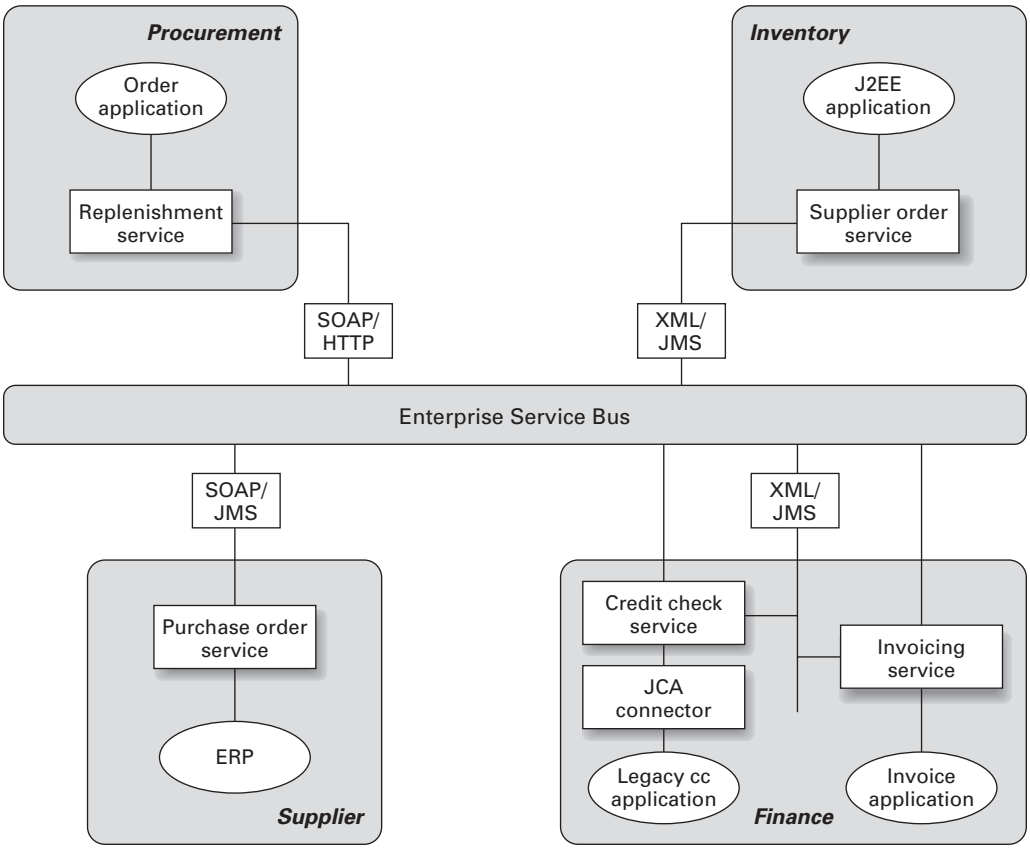


Figure 1.5
Enterprise Service Bus connecting remote services.

"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.).