# 3

# Inventory of Distributed Computing Concepts

Before examining SOA elements in detail in the following chapters, we will review existing concepts of distributed computing. This is important because we are not planning to develop an SOA from scratch. Instead, an SOA will have to incorporate existing middleware technologies and distributed computing concepts. This is particularly important because earlier attempts to replace existing middleware with a new, ubiquitous software bus (e.g., CORBA) have failed, and a successful SOA will have to embrace existing and upcoming technologies instead of replacing or precluding them. Many authors cover the intrinsic details of communication networks and middleware, such as Tanenbaum [Tan2002, Tan2003] and Coulouris [Cou2001]. Aiming our discussion at the application architecture level, we will provide only a brief overview of the most fundamental communication middleware concepts here (including Remote Procedure Calls, Distributed Objects, and Message-Oriented Middleware), followed by a more detailed discussion on the impact that different types of communication middleware have on the application level (including synchrony, invocation semantics, and application coupling).

## 3.1 Heterogeneity of Communication Mechanisms

Techniques for the distribution of enterprise software components are manifold. As will be seen in the remainder of this book, this heterogeneity is inevitable due to the various communication requirements of enterprises.

The situation is comparable to communication in real life—many forms of communication exist (verbal, non-verbal, written, etc.), and every form has its own purpose. It is not possible to replace one form with another without reducing expressiveness.

Figure 3-1 depicts three possible levels of heterogeneity of distribution techniques:

- Communication mode
- Products
- Additional runtime features

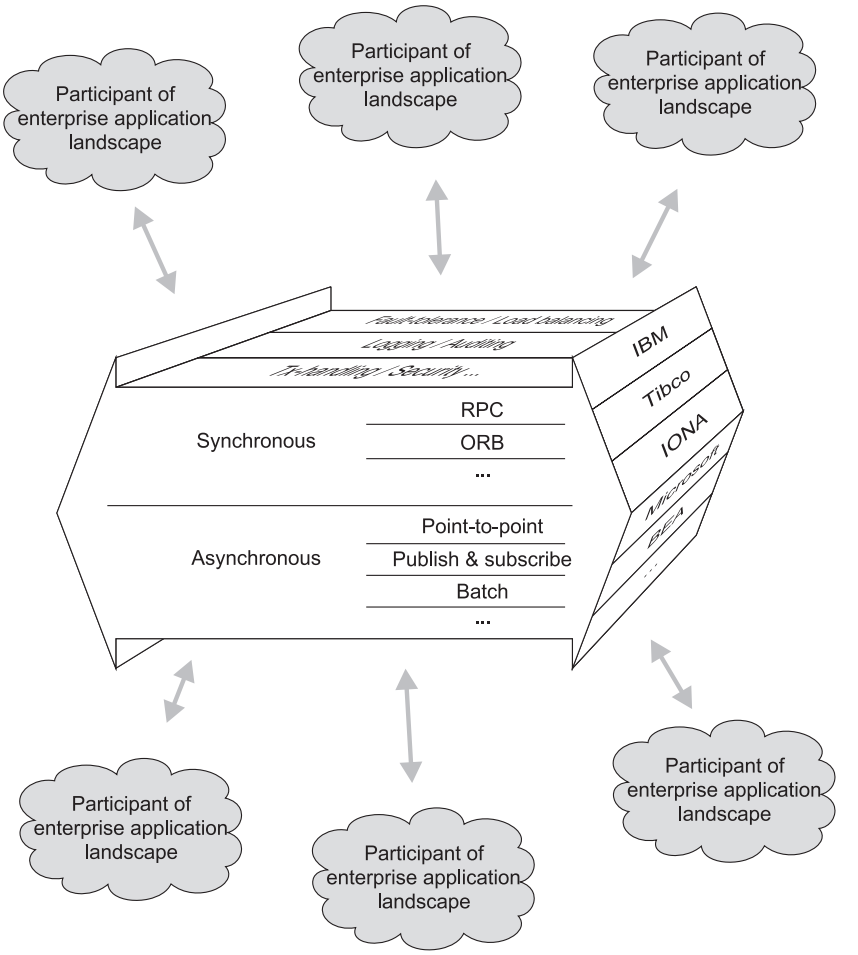



**FIGURE 3-1** Distribution techniques for enterprise applications are characterized by manifold requirements and consequently by various dimensions of heterogeneity.

*Communication modes* are basically distinguished between synchronous and asynchronous mechanisms. Evidently both are required in real-world projects. However, in practice, there are usually numerous variants of these basic modes of communication. Obviously, one can encounter numerous *products* that

provide distribution mechanisms. In addition, a concept that is supposed to cover all the distribution issues of an enterprise must also provide a set of *additional runtime features* such as security support, fault tolerance, load balancing, transaction handling, logging, usage metering, and auditing.

It should be noted that our classification scheme is arbitrary. It is possible to define other classifications or to find additional levels of heterogeneity. However, independent of the classification scheme, it is true that enterprise distribution techniques tend to create heterogeneity at different levels.

From a technical point of view, this scenario leads to three different layers, as shown in Figure 3-2. The first layer contains the core assets of the enterprise application landscape, including all business logic. The second layer provides technology-dependent adapters that connect the core assets to various software busses. Finally, the third layer represents the sum of the enterprise's communication facilities.

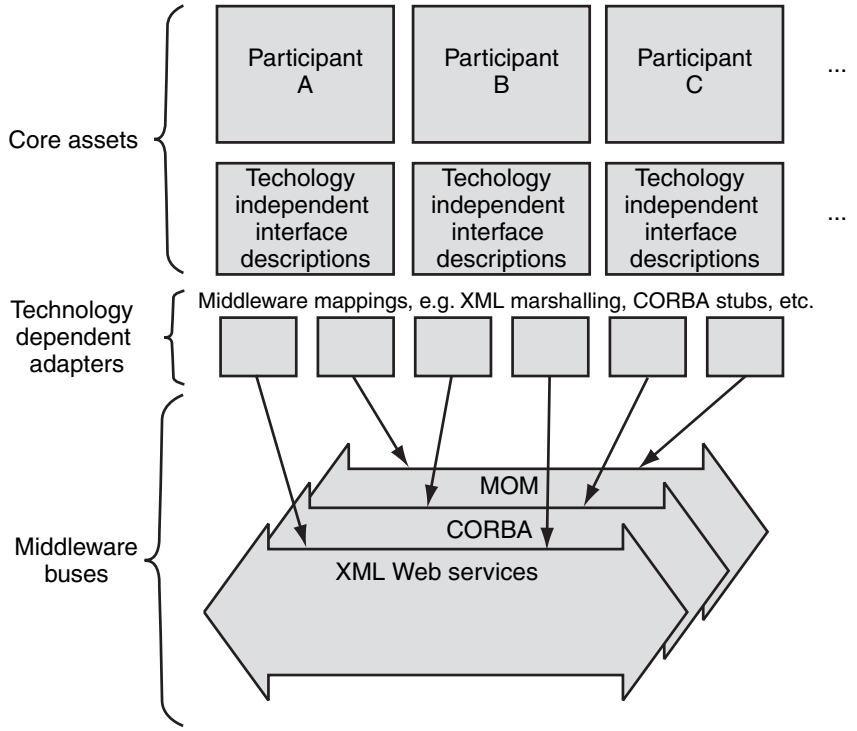



**FIGURE 3-2** Technology-dependent adapters connect participants of an enterprise application landscape with its communication infrastructure.

The remainder of this chapter focuses on the second layer. Chapters 4 to 7 provide an in-depth discussion of the first layer, while Chapter 9 discusses the third layer.

## 3.2 Communication Middleware

A communication middleware framework provides an environment that enables two applications to set up a conversation and exchange data. Typically, this exchange of data will involve the triggering of one or more transactions along the way. Figure 3-3 shows how this middleware framework acts as an intermediary between the application and the network protocol.

In the very early days of distributed computing, the communication between two distributed programs was directly implemented based on the raw physical network protocol. Programmers were involved with acute details of the physical network. They had to create network packets, send and receive them, acknowledge transmissions, and handle errors. Therefore, a lot of effort was spent on these technical issues, and applications were dependent on a specific type of network. Higher-level protocols such as SNA, TCP/IP, and IPX provided APIs that helped reduce the implementation efforts and technology dependencies. They also provided abstraction and a more comfortable application development approach. These protocols enabled programmers to think less in terms of frames at OSI layer 2 or packets at layer 3 and more in terms of communication sessions or data streams. Although this was a significant simplification of the development of distributed applications, it was still a cumbersome and error-prone process. Programming at the protocol layer was still too low-level.

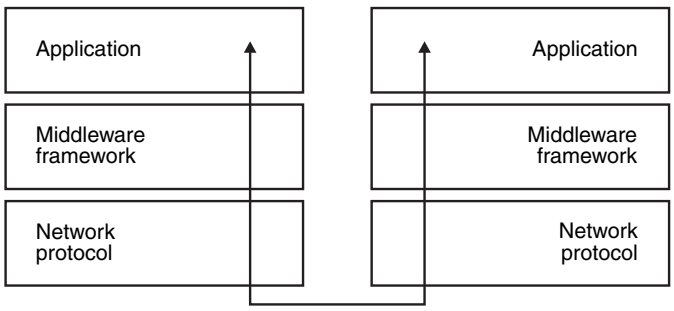



**FIGURE 3-3** A communication middleware framework isolates the application developers from the details of the network protocol.

As the next evolutionary step, communication infrastructures encapsulated the technical complexity of such low-level communication mechanisms by insulating the application developer from the details of the technical base of the communication. A communication middleware framework enables you to access a remote application without knowledge of technical details such as operating systems, lower-level information of the network protocol, and the physical network address. A good middleware framework increases the flexibility, interoperability, portability, and maintainability of distributed applications. However, it is the

experience of the recent two decades that the developer's awareness of the distribution is still crucial for the efficient implementation of a distributed software architecture. In the remainder of this chapter, we will briefly examine the most important communication middleware frameworks.

### 3.2.1 RPC

Remote Procedure Calls (RPCs) apply the concept of the local procedure call to distributed applications. A local function or procedure encapsulates a more or less complex piece of code and makes it reusable by enabling application developers to call it from other places in the code. Similarly, as shown in Figure 3-4, a remote procedure can be called like a normal procedure, with the exception that the call is routed through the network to another application, where it is executed, and the result is then returned to the caller. The syntax and semantics of a remote call remain the same whether or not the client and server are located on the same system. Most RPC implementations are based on a synchronous, request-reply protocol, which involves blocking the client until the server replies to a request.

The development of the RPC concept was driven by Sun Microsystems in the mid 1980s and is specified as RFC protocols 1050, 1057, and 1831. A communication infrastructure with these characteristics is called RPC-style, even if its implementation is not based on the appropriate RFCs.

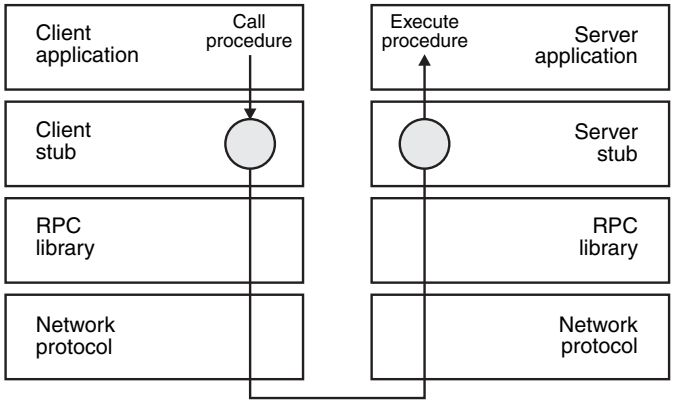



**FIGURE 3-4** RPC stubs and libraries enable location transparency, encapsulate the functional code for the RPC communication infrastructure, and provide a procedure call interface.

It is interesting to note that the need to provide platform-independent services was one of the main drivers in the development of RPC-style protocols. Particularly, the widely used SUN RPC protocol (RFC 1057) and its language bindings were developed to enable transparent access to remote file systems. NFS (RFC 1094) was implemented on top of SUN RPC and is one of the most popular ways to enable networked file system access in Unix-like environments.

At the end of the 1980s, DCE (Distributed Computing Environment) emerged as an initiative to standardize the various competing remote procedure call technologies. DCE also adds some higher-level services such as security and naming services. However, for reasons that were mainly political, DCE failed to win widespread industry support.

### 3.2.2 DISTRIBUTED OBJECTS

In the early 1990s, object-oriented programming emerged as a replacement for the traditional modular programming styles based on procedure or function calls. Consequently, the concept of Distributed Objects was invented to make this new programming paradigm available to developers of distributed applications.

Typically, Distributed Objects are supported by an Object Request Broker (ORB), which manages the communication and data exchange with (potentially) remote objects. ORBs are based on the concept of *Interoperable Object References*, which facilitate the remote creation, location, invocation, and deletion of objects (see Figure 3-5) often involving object factories and other helper objects. By doing so, ORB technology provides an object-oriented distribution platform that promotes object communication across machine, software, and vendor boundaries. ORBs provide location transparency and enable objects to hide their implementation details from clients.

The most common ORB implementations are CORBA, COM/DCOM, and RMI. While RMI is limited to Java and COM/DCOM is restricted to Microsoft platforms, CORBA spans multiple platforms and programming languages.

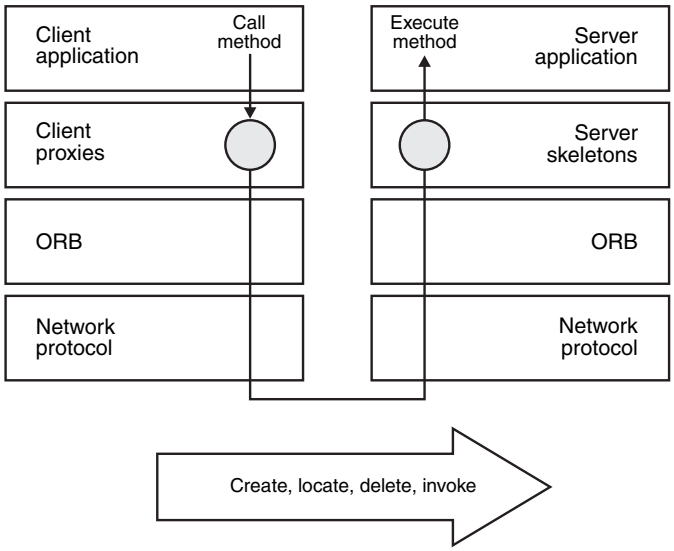



**FIGURE 3-5** ORBs enable client applications to remotely create, locate, and delete server objects (e.g., through factory objects) and communicate with them through remote method invocations.

Today, most enterprise architectures embody object-oriented components (such as for the implementation of graphical user interfaces), but there is rarely an enterprise architecture that is built purely on object technology. In most cases, legacy applications based on programming languages such as COBOL or C are critical parts of an enterprise application landscape (some people prefer the term *software assets* over *legacy software*). It is therefore vital that a component that should be reused on an enterprise-wide level provides an interface that is suitable both for object-oriented and traditional clients. This is particularly true for the service of an SOA.

In this context, it is important to understand that object-oriented applications typically come with a fine-grained interaction pattern. Consequently, applying the object-oriented approach to building distributed systems results in many remote calls with little payload and often very complex interaction patterns. As we will see later, service-oriented systems are more data-centric: They produce fewer remote calls with a heavier payload and more simple interaction patterns.

Nevertheless, it is entirely possible to use ORB technology to implement a data-oriented, coarse-grained SOA interface. This leads to a very restricted application of the ORB technology and typically to an RPC-style usage of the ORB (see Figure 3-6).

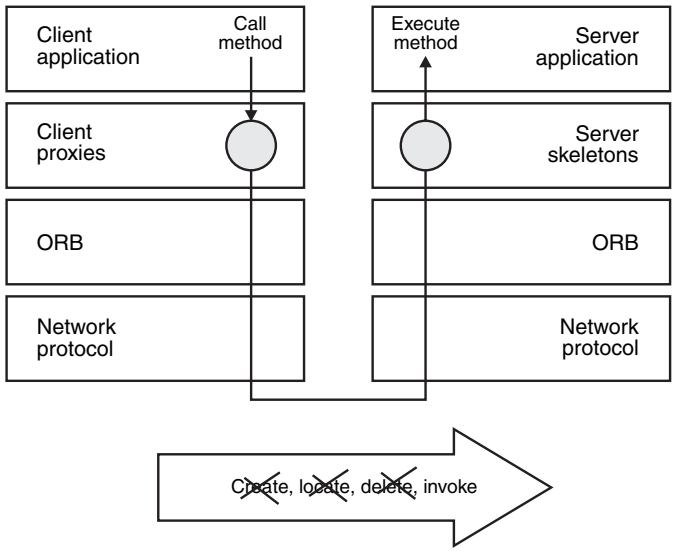



**FIGURE 3-6** An ORB can be used as a communication infrastructure for the implementation of an SOA. In this case, the advanced capabilities of the ORB to cope with multiple instances of remote objects are not used.

### 3.2.3 MOM

With the advent of IBM's MQSeries (now IBM WebSphere MQ) and Tibco Software's Rendezvous in the middle of the 1990s, Message-Oriented Middleware (MOM) technology became popular, and it has since become an integral part of the communication infrastructure landscape of large enterprises.

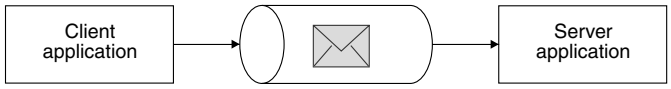
Although there are alternative implementation approaches (e.g., UDP multicast-based systems), the most common MOM implementations are based on the concept of message queuing. The two key components of a message queuing system are *message* and *queue*.

Typically, a message consists of a header and a payload. The structure of the header field is usually predefined by the system and contains network routing information. The payload is application-specific and contains business data, possibly in XML format. Messages typically relate to a specific transaction that should be executed upon receiving the message. Depending on the queuing system, the name of the transaction is either part of the header or the application payload.

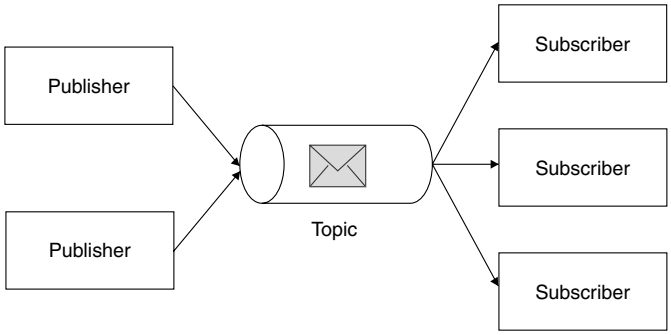
The queue is a container that can hold and distribute messages. Messages are kept by the queue until one or more recipients have collected them. The queue acts as a physical intermediary, which effectively decouples the message senders and receivers. Message queues help to ensure that messages are not lost, even if the receivers are momentarily unavailable (e.g., due to a network disconnection). Email is a good example of the application of messaging concepts. The email server decouples sender and receiver, creating durable storage of email messages until the receiver is able to collect them. Email messages contain a header with information that enables the email to be routed from the sender's email server to the receiver's email server. In addition, email messages can be sent from a single sender to a single receiver (or to multiple recipients, through mailing lists for example), and one can receive email from multiple senders.

Message queuing systems provide similar concepts of connecting senders and receivers in different ways (one-to-one, one-to-many, many-to-many, etc.) (see Figure 3-7). The underlying technical concepts are typically referred to as point-to-point and publish-subscribe models. Point-to-point represents the most basic messaging model: One sender is connected to one receiver through a single queue. The publish-subscribe model offers more complex interactions, such as one-to-many or many-to-many. Publish-subscribe introduces the concept of *topics* as an abstraction, to enable these different types of interactions. Similar to point-to-point, a sender can publish messages with a topic without knowing anything about who is on the receiving side. Contrary to point-to-point communications, in the publish-subscribe model, the message is distributed not to a single receiver, but to all receivers who have previously indicated an interest in the topic by registering as subscribers.

Although the basic concepts of message queuing systems (message, queue, and topic) are relatively simple, a great deal of complexity lies in the many different ways that such a system can be configured. For example, most message queu-




a) A message queue acts as an intermediary between a client application and a server application.




b) Publish-subscribe enables messages to be sent to multiple subscribers.

**FIGURE 3-7** MOM decouples the creators and consumers of messages providing concepts such as point-to-point messaging and publish-subscribe.

ing systems enable the interconnection of multiple physical queues into one logical queue, with one queue manager on the sender side and another on the receiver side, providing better decoupling between sender and receiver. This is similar to email, where senders transmit email to their own mail server, which in turn routes the mail to the receiver's mail server, a process that is transparent to the users of email. Furthermore, queues can be interconnected to form networks of queues, sometimes with intelligent routing engines sitting between the different queues, creating event-driven applications that employ logic similar to that of Petri nets [Rei1992].

Message queuing systems typically also provide a number of different service levels (QoS—quality of service), either associated with specific messages or specific queues. These service levels determine, for example, the transactional capability of the message, the send/receive acknowledge modes, the number of allowable recipients, the length of time a message is valid, the time at which the message was sent/received, the number of times to attempt redelivery, and the priority of the message, relative to other messages.

Generally, MOM encourages loose coupling between message consumers and message producers, enabling dynamic, reliable, flexible, high-performance systems to be built. However, one should not underestimate the underlying complexity of ensuring that MOM-based systems work efficiently, a feature that is not often visible at the outset.

### 3.2.4 TRANSACTION MONITORS

With the rising demand for user-friendly online applications in the 1980s, transaction monitors[1] became popular. They provide facilities to run applications that service thousands of users. It is the responsibility of a transaction monitor to efficiently multiplex the requirements for computing resources of many concurrent clients to resource pools. Most importantly, they manage CPU bandwidth, database transactions, sessions, files, and storage. Today, transaction monitors also provide the capability to efficiently and reliably run distributed applications. Clients are typically bound, serviced, and released using stateless servers that minimize overhead by employing a non-conversational communication model. Furthermore, up-to-date transaction monitors include services for data management, network access, authorization, and security.

Popular examples of transaction monitors are CICS (Customer Information Control System), IMS (Information Management System), Encina, or Tuxedo, which all provide facilities for remote access. A variety of different distribution concepts can be found to support the particular strengths of respective transaction monitors.

Although it is not the intention of this book to discuss current technology and products in detail, a short glance at IBM's CICS and IMS can provide useful insights. CICS is a time-sharing system. Although more than 20 years old, it is still a key element of IBM's enterprise product strategy. It is probably today's most important runtime environment for mission-critical enterprise applications. Even today, there are new applications developed for CICS. Native protocols such as SNA or TCP/IP and various communication infrastructures such as object brokers and messaging middleware can be used to integrate CICS applications with non-CICS applications [Bras2002]. One of the most common ways to connect to a CICS application is CICS's External Call Interface (ECI). The ECI basically provides an RPC-like library that enables remote applications to invoke CICS transaction programs. Based on the ECI, the CICS Transaction Gateway (CTG) provides an object-oriented interface for Java. Contrary to the CICS time-sharing concept, its predecessor IMS was based on processing queues. Although IMS appears to be archaic, it is still important in practice due to its base of installed transaction programs. There are also many different ways to remotely invoke an IMS transaction program [Bras2002]. The most popular ways are IMS Connect and MQSeries OTMA-Bridge [Lon1999]. While IMS Connect imposes an RPC-like access to IMS transaction programs, the MQSeries OTMA-Bridge is based on the MOM concept.

[1] Often referred to as TP monitors, TPMs, or TX monitors.

### 3.2.5 APPLICATION SERVERS

With the booming demand for Web applications in the dot-com era of the late 1990s, the application server became extremely popular. An application server mediates between a Web server and backend systems, such as databases or existing applications. Requests from a client's Web browser are passed from the Web server to the application server. The application server executes code that gathers the necessary information from other systems and composes an HTML reply, which is returned to the client's browser.

An application server can be simple, such as Microsoft ASP (Active Server Pages), which comes with IIS (Internet Information Server), or they can be complex and expensive systems that implement load balancing, data management, caching, transaction management, and security.

The basic functions of an application server can be described as hosting components, managing connectivity to data sources, and supporting different types of user interfaces, such as *thin* Web interfaces or *fat* client applications. Taking a closer look at the basic mechanisms of an application server, one can sometimes get the impression that not much has changed from the days of IBM CICS and VT3270 terminals, only that these days, the user interfaces are more colorful.

When looking at the high end of application servers, notably Microsoft .NET Server, BEA WebLogic, and IBM WebSphere, it can sometimes be difficult to find out exactly what is still part of the core application server functionality because these companies have started to use their respective brands to describe a very broad array of products. For example, J2EE, the application server framework from the non-Microsoft camp, started with core application server functionality including JSP (Java Server Pages) for the dynamic generation of HTML and EJB (Enterprise Java Beans) for managing and hosting more complex software components. Within a couple of years, J2EE has become a complex and sophisticated framework (see Figure 3-8).

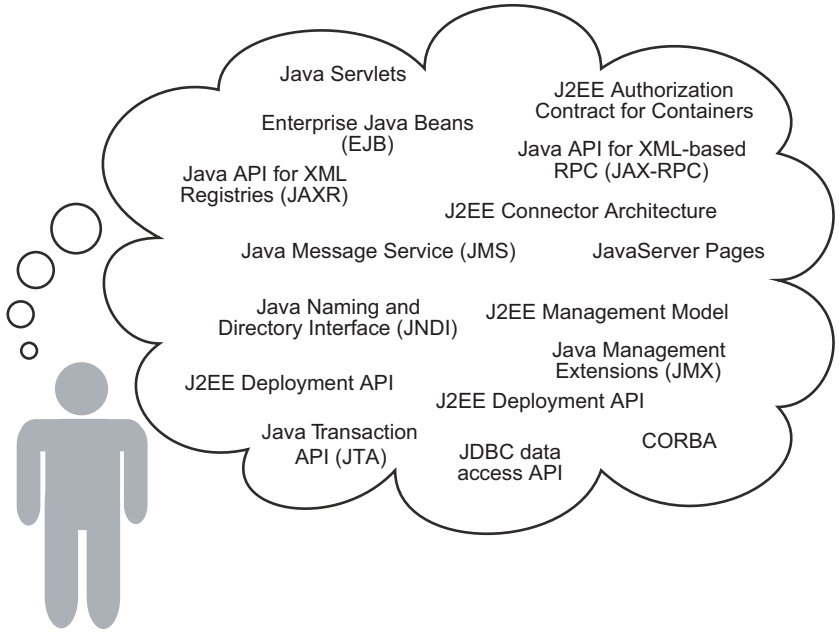



**FIGURE 3-8** J2EE comprises a variety of standards.

## 3.3 Synchrony

Synchronous and asynchronous communications are two different forms of interaction that both require the support of a generic technology for distributed systems.

Synchronous communication is characterized by the immediate responses of the communication partners. The communication follows a request/reply pattern that enables the free flow of conversation, often based on the use of busy waits. Applications with user interaction specifically require this conversational mode of interaction. Synchronous communication requires that the client and server are always available and functioning.

Asynchronous communication is less stringent. Both communication partners are largely decoupled from each other, with no strict request/reply pattern. Typically, one party creates a message that is delivered to the recipient by some mediator, and no immediate response is needed. The sender can store context information and retrieve it when the recipient returns the call, but there is not necessarily a response. In contrast to a synchronous request-reply mechanism, asynchronous communication does not require the server to be always available, so this type can be used to facilitate high-performance message-based systems.

Typically, synchronous communication is implemented by RPC-style communication infrastructures, while asynchronous mechanisms are implemented by MOM. However, it is entirely possible to implement synchronous communication based on MOM, and it is also possible to build MOM-style interaction over RPC. Nevertheless, RPC is more suitable if immediate responses are required, and MOM is the technology of choice for decoupled, asynchronous communication.

Due to the manifold requirements of most real-world scenarios, typical enterprise systems embody both synchronous and asynchronous communication. For this purpose, a variety of different communication infrastructures is used, ranging from simple FTP (File Transfer Protocol) to more advanced middleware platforms, such as RPC and MOM. In addition, there are also communication infrastructures that support both communication modes—for example, pipelining RPC, which supports asynchronous communication in addition to the standard synchronous RPC communication.

To conclude this discussion on synchrony, we will provide an overview of the most common ways of implementing synchronous and asynchronous communication with both RPC/ORB and MOM technology. We will look at the following examples:

- Simulated synchronous services with queues
- Asynchronous one-way: fire-and-forget RPC
- Callbacks and polling services

The first example, *simulated synchronous communication*, can often be found in mainframe environments, where a message queuing system has been chosen as the standard means for remote interaction with the host, such as OS/390 with MQSeries access. This is a common scenario in many large enterprises. Often, these companies have gone one step further, developing frameworks on top of this combination of OS/390 and MQSeries that enable service-like interfaces to the most widely used transactions on the mainframe. This is done by implementing client-service wrappers that shield the underlying MQ infrastructure from the client developer. These service wrappers often simulate synchronous interactions with the mainframe by combining two underlying queues, one with request semantics and the other with reply semantics, using correlation IDs to pair messages into request/reply tuples. Effectively, this relegates the message queuing system to playing a low-level transport function only, not generally leveraging any of the advanced features of the messaging system. Figure 3-9 provides an overview of this approach.

The second example, *fire-and-forget RPC*, assumes an RPC or ORB implementation with asynchronous one-way semantics: The client fires off a request to the server without expecting an answer. This can be achieved either by defining an operation signature that does not include any return values or by using specific features of the middleware, such as a CORBA IDL operation using the keyword `oneway`. Figure 3-10 provides an overview of this approach.

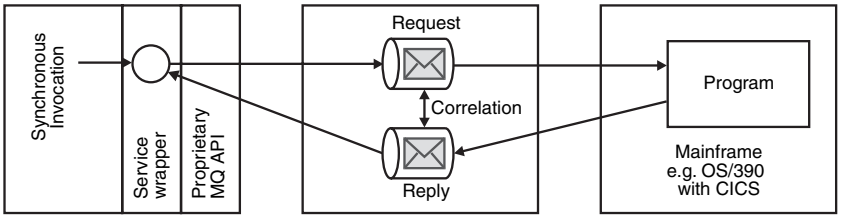



**FIGURE 3-9** Simulated synchronous services with queues. A correlation ID maps a reply message to the corresponding request. On the client side, this is hidden by a service wrapper, which gives the caller the impression of synchrony.

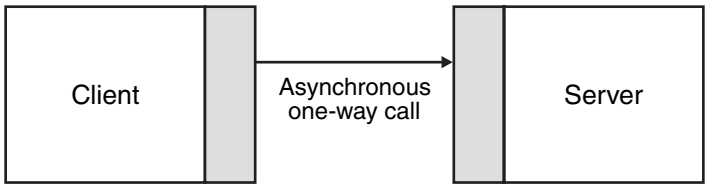



**FIGURE 3-10** A synchronous one-way call implies fire-and-forget semantics. The request is fired off by the client without a reply from the server.

There are two key issues with this approach: The first is that the client has no guarantee that the server will receive and process the request appropriately. This problem reduces the applicability of this method significantly. The second problem is that most RPCs/ORBs typically use a reliable communication protocol such as TCP. Sending a one-way request through TCP generally means that the client is blocked until delivery to the server on the TCP level has been completed. If a server is getting swamped with requests, it might become unable to process all incoming one-way requests on the TCP layer. Effectively, this means that the client is blocked until the server is at least able to read the request from the network. Therefore, it is not advisable to use this approach to implement large-scale event notification. Instead, an appropriate MOM should be chosen.

The third example, *callbacks and polling services*, is the most common way of decoupling clients and server in RPC-style applications, without having to move to a fully fledged MOM solution. The basic idea is similar to the conventional callback, as it is realized in functional programming languages with function pointers, or in OO languages using object references: A client sends a request to the server, and the server stores the request and returns control back to the client (possibly sending an acknowledgment that it received the request). After having processed the request, the server (now acting as a client) sends the result back to the client (now acting as a server), using the standard RPC/ORB invocation mechanism (see Figure 3-11). Sometimes, it is not possible for the cli-

ent to act as a server (e.g., due to firewall restrictions). In these cases, the client can periodically poll the server for the availability of the result.

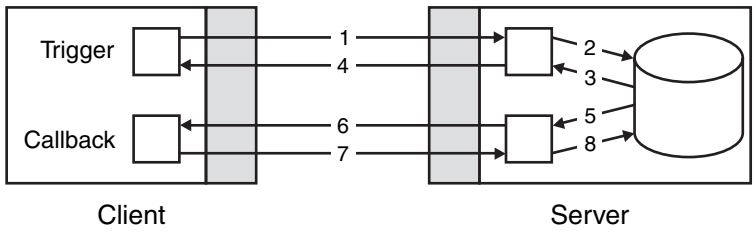



**FIGURE 3-11** Callbacks and polling services: A client sends a request to a server ("trigger"). The server stores the requested activity in a database before replying with an acknowledgment to the client. The server has a thread that takes pending requests from the database, processes them, and sends back the result to the originating client using callback.

This approach can sometimes provide a good compromise for satisfying the need to decouple clients and servers without having to add technology such as a MOM. However, often the implementation of such a remote callback can be more difficult than it originally appears. This is especially true if the application requires a high degree of reliability. In these cases, it is necessary to introduce some kind of mechanism for ensuring reliability, such as through combining a server-side database with some kind of custom-built acknowledgment protocol. Also, the server-side logic can become quite complex: To ensure that all requests are eventually processed, the database must be constantly polled for pending requests, potentially adding a huge burden on database performance. For this reason, one should carefully weigh the use of database triggers. Here, it is important to ensure that the execution of the trigger is not part of the same transaction that puts the new request in the database. In this case, you could encounter a situation where the client is blocked because it has to wait not only until the server has stored the request in the database before returning an acknowledgment to the client, but also until the database trigger has been executed. This will effectively eliminate the decoupling effect of the callback implementation.

As shown in Figure 3-12, the server-side implementation can alternatively use internal message queues to ensure an efficient means of storing incoming requests in a reliable and efficient manner, thus avoiding many of the issues with the pure-database approach described previously.

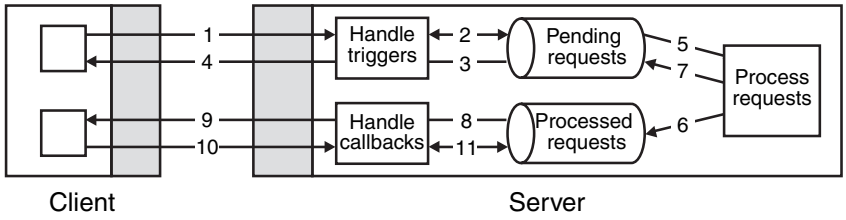



**FIGURE 3-12** Callbacks and queues. Similar to the previous example, except that queues are introduced on the server side to ensure better decoupling on the server side.

## 3.4 Interface Versus Payload Semantics

Typically, an interaction between a client and a server (or a sender and a receiver) results in the execution of a transaction (or some other activity) on the receiving end. In order to determine the type of transaction or activity that was requested by the caller (or sender), it is necessary to specify the operation. This is normally performed in one of two ways: The requested transaction/activity can be encoded in the operation signature of the server component's interface, or it can be embedded in the message itself.

In the first case, the requested transaction (or other kind of activity) is defined by using self-descriptive function names such as `saveCustomer()`, `retrieveCustomer()`, or `transferMoney()`. RPC-style interfaces provide this type of semantically rich interface, which we refer to as *interface semantics* (see Figure 3-13).

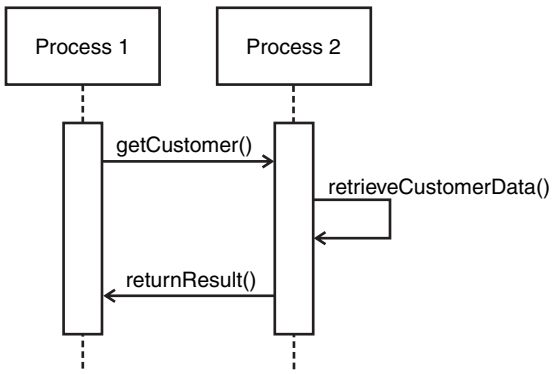



**FIGURE 3-13** RPC-style interaction is typically based on interface semantics. Every procedure call has a meaningful name that indicates its purpose.

In the second case, the requested transaction is embedded directly into the message (see Figure 3-14). This can be done as part of the message header (if the MOM provides such a field as part of the message header data structure), or as part of the application specific payload. We refer to this as *payload semantics*.

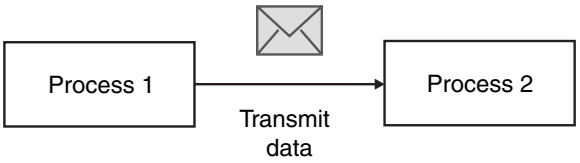



**FIGURE 3-14** The name of a remote call with payload semantics has no functional meaning in its own right. The remote functionality required is encoded in the message that is sent. The receiver typically has to determine the function name and the dispatch message to the associated business function.

Payload semantics is widely used in the context of MOMs that provide APIs with functions such as `MQGET()/MQPUT()` or `sendMessage()/onMessage()/receiveMessage()` for the clients and servers to communicate with each other. The semantics of these functions is purely technical (see Figure 3-15).

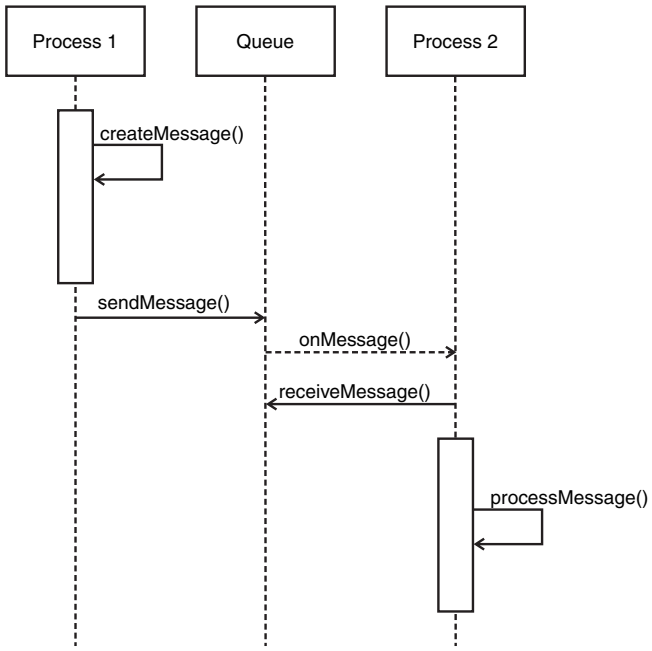



**FIGURE 3-15** MOM is generally based on payload semantics. Functions such as `sendMessage()` and `processMessage()` are purely technical, without any business semantics.

Interface semantics provide users with well-defined interfaces that are intuitive and easy to understand. Changes to these interfaces require modifications to all applications that depend on the particular interface, even if they do not depend on the operation or argument that was added or changed. Payload semantics, on the other hand, result in systems where changes to message formats can have a potentially lesser impact on the different components of the system. New functionality can easily be added to a system by creating new messages types. Consumers that are not dependent on the new messages types remain unaltered. Thus, payload semantics results in a weaker coupling at the type level.

The choice of *interface semantics* versus *payload semantics* is not an obvious one, as each approach has its pros and cons. Strongly typed languages, such as Java, limit the flexibility of the programmer by applying strict type checking at compile time. Almost all dependencies caused by the change of a type in the system can be discovered at compile time, thus significantly reducing the number of runtime errors. Weakly typed languages, such as TCL, offer much more flexible data manipulation, often based on string manipulation. These types of languages are generally used for scripting, especially in Web environments, where fast results are required.

However, the application of interface semantics and payload semantics cannot be viewed in black-and-white terms. While RPCs are not restricted to pure functional call semantics, neither are MOMs limited to payload semantics. This can be illustrated with one insightful example. Consider an RPC call such as `transferMoney()`. The transmitted data can significantly contribute to the determination of the code that is executed:

```
String transferMoney (amount: decimal; cur, accFrom, accTo: String);
{
   switch (cur)
   case 'EUR':
      handleEurTransfer (amount, accFrom, accTo);
   case 'GBP':
      handleGbpTransfer (amount, accFrom, accTo);
   case 'USD':
      handleUsdTransfer (amount, accFrom, accTo);

   . . .
}
```

Going one step further, it is possible to remove all interface semantics from an RPC-style interface. In the following example, a service supports exactly one function, `executeService()`. This function has only one parameter, a plain string. This string encodes all functional parameters and the requested functionality:

```
String executeService (message: String);
{
   int i = determineFunctionNumber (message);
```

```
switch (i)
   case 1:
      handleCase1 (message);
   case 2:
      handleCase2 (message);
   case 3:
      handleCase3 (message);
   . . .
}
```

### 3.4.1 DOCUMENT-CENTRIC MESSAGES

With the emergence of self-descriptive data structures such as XML, an approach to handling message types referred to as *document-centric* has become popular. Document-centric messages are semantically rich messages where the operation name, its parameters, and the return type are self-descriptive. However, the passed parameters and returned object can be extremely flexible and can include any number of optional parameters. SOAP (Simple Object Access Protocol) is a technology that is particularly suitable for this type of solution. As long as the underlying XML Schemas impose only loose constraints on the document structure, the parameters can be extended in any manner required without breaking compatibility with previous software versions. Consider the following example that describes the booking of a flight. It is straightforward to enhance the protocol with additional parameters, such as the time of day of the flight, the date and time of arrival, or the verbose names of the airports. As long as these fields are not required, the previous version of the protocol, in addition to all software that relies on that version, remains totally valid. Because this type of communication includes a structured document in both reply and response, it is called document-driven communication:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/
soap/encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Body>
      <ns1:bookFlight xmlns:ns1="http://www.openuri.org/">
         <ns1:inbound>
            <ns1:flightNumber>LH400</ns1:flightNumber>
            <ns1:flightDate>2003-11-08</ns1:flightDate>
            <ns1:isConfirmed>false</ns1:isConfirmed>
         </ns1:inbound>
         <ns1:outbound>
            <ns1:flightNumber>LH401</ns1:flightNumber>
            <ns1:flightDate>2003-11-17</ns1:flightDate>
            <ns1:isConfirmed>false</ns1:isConfirmed>
         </ns1:outbound>
         <ns1:passenger>
```

```
            <ns1:Passenger>
               <ns1:firstName>Karl</ns1:firstName>
               <ns1:lastName>Banke</ns1:lastName>
               <ns1:birthday>1970-08-05</ns1:birthday>
            </ns1:Passenger>
         </ns1:passenger>
      </ns1:bookFlight>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

## 3.5 Tight Versus Loose Coupling

Recently, a lot of attention has focused on comparisons between *loose coupling* and *tight coupling* approaches to application interactions. On the technology side, this has mainly been driven by the potential of Web services to dynamically discover and bind to other services, such as through UDDI (Universal Description, Discovery and Integration). On the business side, this has been driven by the growing need of enterprises to increase flexibility with respect to changes in their own business processes and the ways in which they interact with partner companies.

Traditionally, business processes have been designed within the boundaries of an enterprise, or even within the different business units of the enterprise. These activities were managed with the help of detailed, real-time information. Processes that span multiple business units or enterprises typically have to deal with a very different set of requirements, needing a higher degree of flexibility. In these kinds of scenarios, one sees a much higher degree of uncertainty, a much more frequent change in terms of participants and their roles, and a constant evolution of the types of interactions required.

There appears to be a consensus that for these types of "in-flux" situations to operate, a loosely coupled architecture is required because loose coupling is seen as helping to reduce the overall complexity and dependencies. Using loose coupling makes the application landscape more agile, enables quicker change, and reduces risk. In addition, system maintenance becomes much easier. Loose coupling becomes particularly important in the B2B world, where business entities must be able to interact independently. The relationships between business partners often change rapidly—alliances are settled and cancelled, and business processes between trading partners are adopted to new market requirements. Two companies that are partners in one market might be competitors in another market. Therefore, it is essential that the underlying IT infrastructure reflect this need for flexibility and independence. Ideally, no business relationship should impact another—new business relationships should be able to be established without any effect on existing ones. Functionality that is offered to one business partner might not necessarily be available to others. A change that is relevant for one business

partner should have no impact on other partners. One trading partner may not cause another to block while waiting for a synchronous response, nor may one IT system depend on the technical availability of the IT system of a business partner.

The term *coupling* refers to the act of joining things together, such as the links of a chain. In the software world, *coupling* typically refers to the degree to which software components depend upon each other. However, the remaining question is: "*What are these dependencies, and to what degree can one apply the properties of* tight *and* loose*?*" Software coupling can happen on many different levels. One of the first issues is to differentiate between build time (compile time) dependencies and runtime dependencies. However, this is typically only sufficient when looking at monolithic applications. In a distributed environment, we believe that in order to determine the degree of coupling in a system, one needs to look at different levels. Table 3-1 provides an overview of these levels and shows how they relate to the tight versus loose coupling debate.

**Table 3-1** Tight Versus Loose Coupling

| Level | Tight Coupling | Loose Coupling |
|---|---|---|
| Physical coupling | Direct physical link required | Physical intermediary |
| Communication style | Synchronous | Asynchronous |
| Type system | Strong type system (e.g., interface semantics) | Weak type system (e.g., payload semantics) |
| Interaction pattern | OO-style navigation of complex object trees | Data-centric, self-contained messages |
| Control of process logic | Central control of process logic | Distributed logic components |
| Service discovery and binding | Statically bound services | Dynamically bound services |
| Platform dependencies | Strong OS and programming language dependencies | OS- and programming language independent |

In the following, we will examine the items in Table 3-1 in detail.

For distributed systems, the way that remote components are connected is possibly the most obvious technical factor when looking at the problem of "coupling." A physical intermediary enables loose coupling on this level. Therefore, MOM systems are loosely coupled on the physical level, with message queues acting as an intermediary, decoupling senders and receivers of messages. RPC-style applications are tightly coupled on this level because clients and servers interact directly with each other—clients require servers to be alive and accessible in order to interact with them.

The impact of synchronous versus asynchronous communication on the level of coupling is often closely related to the physical linking of the distributed components, as described previously. Asynchronous communication is generally

associated with loose coupling. However, this assumes that the underlying middleware is capable of supporting the asynchronous communication in a loosely coupled manner. Assume a one-way RPC call: There is still a notion of tight coupling here, even if the client does not wait for the reply of the server—the client will only be able to send the one-way request to the server if it is directly connected and if the server is up and running. This is a good example for the varying degrees of "coupledness"—asynchronous communication through a proper MOM is more loosely coupled than asynchronous one-way RPC calls.

Looking at the type system of a distributed application as the next level of "coupledness," we find that the stronger the type system, the stronger the dependencies between the different components of the system. This is true not only during the application development phase, but also (and perhaps more importantly) when changing or reconfiguring the running system. Earlier, we differentiated between *interface semantics* and *payload semantics*. Interface semantics provide an explicit interface and operation names and also strongly typed arguments. Effectively, this means components are tightly coupled together on this level because every change of an interface ripples through the entire application, as far as dependent components are concerned. The benefit is that we discover the affected parts of the application that need to be adapted to reflect these changes at compile time, thus avoiding runtime exceptions due to incompatible message formats. *Payload semantics,* on the other hand, enable a looser coupling of components because message formats are generally more flexible. In some cases, message format validation might be applied, such as through XML Schema validation. However, this requires efficient management of the up-to-date schema definitions between participants. Notice that the problems with changes to message formats is not eliminated by employing payload semantics: One must still know those parts of the system that are affected by changes in order to ensure that they can act appropriately on the new format. In many cases, this means that the problem has simply moved from build time to runtime.

Another important factor to examine is the interaction patterns of the distributed components. For example, an ORB-based system will typically impose an OO-style navigation of complex object trees. The client has to understand not only the logic of each individual object, but also the way to navigate across objects, again resulting in a fairly tight coupling. Given that RPC-style interfaces do not enable such complex navigation, the degree of coupling is lower when compared to a distributed object system. MOM-based systems typically impose a much simpler interaction model, where often a single queue is sufficient as an entry point for clients, and all input for server-side transactions is provided in a single message.

Related to this discussion is the question of whether we generally assume that the system is structured around RPC-style services or around queues and topics. Generally, topics and queues provide more flexibility for changing the system at runtime by rearranging the configuration of queues and how they are related to each

other. The powerful configuration management of most MOM systems greatly increase the "looseness" of the coupling between system components.

Another important factor is the ownership or control of process logic. If processes are managed centrally, this results in tight coupling between the different sub-processes and transactions. For example, database mechanisms might be used for ensuring referential integrity and general consistency of the data owned by the different sub-processes. This is often the case, for example, with large, monolithic ERP (Enterprise Resource Planning) systems. If business processes are highly distributed, as in a B2B environment, the different sub-processes and transactions are generally more independent of each other, or more loosely coupled, in the context of our current discussion. Often, this means that one must accept the fact that there is no globally defined consistent process state. Similarly, the data owned by the different participants might not always be consistent—one system might have already cancelled an order for which another system still owns an invoice.

Finally, the way in which participants in the system locate each other has a great impact on the level of coupling in the system. Statically bound services yield very tight coupling, whereas dynamically bound services yield loose coupling. Looking up services in a naming or directory server reduces the tightness with which components are tied together, although it still requires the client to know the exact name of the service to which it wants to bind. Services such as UDDI enable a more flexible location of services, using constraints such as "*Find me the next printer on the second floor.*" Notice that dynamic service discovery as provided by UDDI for Web Services is not a new concept; it has previously been provided by other standards such as the CORBA Naming Service. Notice also that past experience has shown that the number of applications requiring completely dynamic service discovery has been fairly limited.

When making architectural decisions, one must carefully analyze the advantages and disadvantages of the level of coupling. Generally speaking, OLTP-style (online transaction processing) applications, as they are found throughout large enterprises, do not normally require a high degree of loose coupling—these applications are tightly coupled by their nature. When leaving the scope of a single enterprise or single business unit, especially in B2B environments, loose coupling is often the only solution. However, in most cases, the increased flexibility achieved through loose coupling comes at a price, due to the increased complexity of the system. Additional efforts for development and higher skills are required to apply the more sophisticated concepts of loosely coupled systems. Furthermore, costly products such as queuing systems are required. However, loose coupling will pay off in the long term if the coupled systems must be rearranged quite frequently.

## 3.6 Conclusion

Today's enterprise application landscapes are characterized by a variety of different technologies and concepts for distribution. On one hand, this variety arises within the enterprise organization itself for historical reasons, personal preferences of different people, and the dynamics of acquisitions and mergers. As a matter of fact, many redundant concepts exist within the same organizational unit. On the other hand, complementary concepts and technologies also exist. Due to the requirements of different types of distribution problems that coexist in one corporation, different solutions arise as well.

A modern architecture must be able to embrace all these technologies and concepts. Heterogeneity—including heterogeneity of middleware—must be understood as a fundamental fact that cannot be fought but instead must be managed. Furthermore, an architecture must accommodate frequent changes of the underlying distribution infrastructure. As a matter of fact, the lifecycles of today's infrastructure products are largely incompatible with the lifecycles of enterprise applications. Thus, you must protect the assets of an existing application landscape and simultaneously take advantage of the latest infrastructure products.

In this chapter, we have discussed the necessity of carefully choosing the right approach to integrating two distributed software components. Among other issues, you must decide on the appropriate communication infrastructure, synchrony, call semantics, usage of an intermediary, and object-oriented versus data-centric interfaces. All these decisions impact the coupling of the two systems.

### References

Braswell, Byron, George Forshay, and Juan Manuel Martinez. *IBM Web-to-Host Integration Solutions*, 4th ed. IBM Redbook SG24-5237-03, 2002. [Bras2002]

Long, Rick, Jouko Jäntti, Robert Hain, Niel Kenyon, Martin Owens, and André Schoeman. *IMS e-business Connect Using the IMS Connectors*. IBM Redbook SG24-5427-00, 1999. [Lon1999]

Tanenbaum, Andrew S. *Computer Networks*, 4th ed. Prentice-Hall, 2003. [Tan2003]

Tanenbaum, Andrew S. and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2002. [Tan2002]

Coulouris, George, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design,* 3rd ed. Addison-Wesley, 2001. [Cou2001]

Reisig, Wolfgang. *A Primer in Petri Net Design.* New York: Springer Compass International, 1992. [Rei1992]

## URLs

http://www.rfc-editor.org

http://www.omg.org

http://www.microsoft.com/com

http://java.sun.com/j2ee

http://www.bea.com