3

Enterprise Application Integration

Middleware and enterprise application integration (EAI) are not completely orthogonal concepts. They are, however, distinct enough to warrant separate treatment. As we saw in Chapter 2, middleware constitutes the basic infrastructure behind any distributed information system. Initially, middleware was used to construct new systems and to link to mainframe-based systems (2-tier architectures). Later, it was used to distribute the application logic and to integrate the many servers created by 3-tier architectures.

When the systems involved were compatible and comparable in their functionality and did not involve many platforms, middleware could be used without further ado to integrate the servers. Unfortunately, for more ambitious projects, plain middleware was not enough. The main limitation was that any concrete middleware platform makes implicit assumptions about the nature of the underlying systems. When these systems are very different in nature and functionality, using conventional middleware to integrate them becomes rather cumbersome, and in some cases simply infeasible.

EAI can be seen as a step forward in the evolution of middleware, extending its capabilities to cope with application integration, as opposed to the development of new application logic. Such extensions involve some significant changes in the way the middleware is used, from the programming model to the marked shift toward asynchronous interaction. In this chapter we examine how these extensions came about and how they facilitate large scale integration. In the first part, we discuss in detail the problem of application integration (Section 3.1). Then we address the use of message brokers as the most versatile platform for integration (Section 3.2), and conclude with a review of workflow management systems as the tools used to make the integration logic explicit and more manageable (Section 3.3).

3.1 From Middleware to Application Integration

The difference between conventional middleware and EAI can sometimes be rather subtle. To understand the differences in terms of requirements and actual systems, it helps to differentiate between application development and application integration.

3.1.1 From a Mainframe to a Set of Servers

In Chapter 1, we discussed in detail how client/server systems came to be. As the available bandwidth increased and PCs and workstations became increasingly powerful, the client/server paradigm gained momentum. Functionality that was previously only available in a single location (the mainframe) began to be distributed across a few servers. At the same time, companies became more decentralized and more geographically dispersed and also started to increasingly rely upon computers. As a result, information servers established their presence everywhere within a company. Because of the limitations of client/server architectures, these servers were effectively information islands; clients could communicate with servers, but the information servers did not communicate with each other. At a certain point, this severely limited the ability to develop new services and introduced significant inefficiencies in the overall functioning of the enterprise.

When 3-tier architectures and middleware emerged, they addressed two issues. First, by separating the application logic layer from the resource management layer, the resulting architecture became more flexible. This approach gained even more relevance when systems began to be built on top of computer clusters instead of powerful servers, a trend that was markedly accentuated by the Web. Second, they served as a mechanism for integrating different servers. In this regard, middleware can be seen as the infrastructure supporting the middle tier in 3-tier systems. As such, it is the natural location for the integration logic that brings different servers together. For instance, the accepted way to integrate different databases was to use a TP monitor.

3.1.2 From a Set of Servers to a Multitude of Services

The use of middleware led to a further proliferation of services. In fact, 3-tier architectures facilitate the integration of different resource managers and, in general, the integration of services. The functionality resulting from this integration can be then exposed as yet another service, which can in turn be integrated to form higher-level services. This process can go on ad infinitum, leading to a proliferation of services. The big advantage is that each new layer of services provides a higher level of abstraction that can be used to hide complex application and integration logic. The disadvantage is that now integration is not only integration of resource managers or servers, but also the integration of services. Unfortunately, while for servers there has been a significant effort to standardize the interfaces of particular types of servers (e.g., databases), the same cannot be said of generic services. As long as the integration of services takes place within a single middleware platform, no significant problems should appear beyond the intrinsic complexity of the system being built. Once the problem became the integration of services provided by different middleware platforms, there was almost no infrastructure available that could help to reduce the heterogeneity and standardize the interfaces as well as the interactions between the systems.

Thus, while 3-tier architectures provided the means to bridge the islands of information created by the proliferation of client/server systems, there was no general way to bridge 3-tier architectures. Enterprise application integration appeared in response to this need. Middleware was originally intended as a way to integrate servers that reside in the resource management layer. EAI is a generalization of this idea that also includes as building blocks the application logic layers of different middleware systems.

3.1.3 An Example of Application Integration

Behind any system integration effort there is a need to automate and streamline procedures. During the late 1980s and the 1990s, enterprises increasingly relied on software applications to support many business functions, ranging from "simple" database applications to sophisticated call center management software or customer relationship management (CRM) applications. The deployment of information systems allowed the automation of the different steps of standard business procedures. The problem of EAI appears when all these different steps are to be combined into a coherent and seamless process.

To understand what this entails, consider the problem of automating a *supply chain*. A supply chain is the set of operations carried out to fulfill a customer's request for products and services. Simplifying and abstracting the procedure, a basic supply chain comprises the following steps: *quotation*, *order processing*, and *order fulfillment*. Quotation involves processing a *request for quotes* (RFQ) from a customer. In an RFQ, the customer queries the company about the price, availability, and expected delivery dates of particular goods. Based on this information, the customer may eventually place a purchase order with the company. Order processing involves the analysis of the purchase order placed by the customer. This includes verifying that the purchase order corresponds to a previously given quote and that it can be fulfilled under the conditions requested by the customer. It also involves placing the order internally to schedule the manufacturing of the goods, purchase the necessary components, and so on. Order fulfillment includes several steps: *procurement*, *shipment*, and *financial aspects*. Procurement is the actual acquisition of components and manufacturing of the requested product. Shipment is the delivery of the product to the customer. The financial aspects include invoicing the customer, paying suppliers, and so forth.

An actual supply chain is much more complex and involves many more steps. We can nevertheless already understand the difficulties of EAI using this basic example. Each one of the steps mentioned is likely to be implemented and supported using a different information system. For instance, companies maintain extensive customer, product, and supplier databases. Responding to an RFQ may involve checking the availability of the product, their production schedule, and even checking with suppliers for delivery dates and prices for the required components. In many cases, each of these databases is a separate system that may even reside in a different geographic location. Processing the purchase order may involve interacting with a warehouse control system that indicates the current stock levels of the requested product and where it can be obtained. As part of the order fulfillment step, the purchase order may be forwarded to a manufacturing system. In this case there might be many additional steps to purchase components from suppliers, arrange for delivery dates, schedule the production and testing, and so on, all of them involving more interactions implemented using different information systems. Finally, shipment and billing also require more or less complex interactions with invoice databases, purchase order archives, and the like. Of the systems involved, some are home-grown, others are based on off-the-shelf packages, and yet others are the result of previous integration efforts. Moreover, each has different characteristics:

- Each system may run on a different operating system (e.g., Windows, Linux, Solaris, HP-UX, or AIX).
- Each system may support different interfaces and functionality. For example, some will be transactional, while others will not understand transactions; some will use a standard IDL to publish interfaces, others would use a proprietary syntax, etc.
- Each system may use a different data format and produce information that cannot be easily cast into parameters of a procedure call (e.g., a complex multimedia document).
- Each system may have different security requirements (for example, some systems may require authentication based on X.509 certificates, while others may need a simple username/password authentication).
- Each system may use a different infrastructure as well as different interaction models and protocols (e.g., a DCE installation, a TP monitor, a CORBA-based system, etc.).

Automating the supply chain implies bringing all these disparate systems together. To make matters worse, EAI is also complicated by non technical challenges: the systems to be integrated are typically owned and operated by different departments within a company. Each department is autonomously managed, and uses its systems to perform a variety of department-specific functions whose needs and goals are not necessarily aligned with those of the integrating application.

In spite of all these difficulties, it is often critical to automate the supply chain. When it is not automated, all the operations that involve going from one step to the next in the chain are carried out manually (Figure 3.1). When that is the case, the whole process involves a lot of repetitive human labor, exchange of paper documents, inefficiencies, and errors. Orders are difficult to monitor and track. It is difficult, if not impossible, to have an overall view of operations and to give information about the status of an order. Any tracking or monitoring can only be done through the cumbersome procedure of following the paper trail left behind by the process. Such inefficiencies strongly affect the quality of the supply chain and severely harm the ability to sustain growth. In the following we describe the middleware technologies that facilitate the integration of such coarse-grained, heterogeneous components.

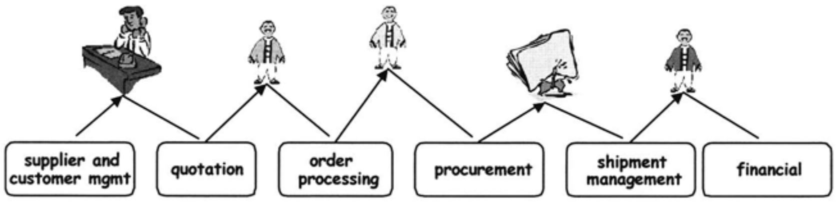

Fig. 3.1. Manual implementation of a supply chain where human users act as relays between the different steps by extracting data from one system, reformatting it, and feeding it into the next

3.2 EAI Middleware: Message Brokers

Traditional RPC-based and MOM systems create point-to-point links between applications, and are thus rather static and inflexible with regard to the selection of the queues to which messages are delivered. Message brokers address this limitation by acting as a broker among system entities, thereby creating a (logical or physical) "hub and spoke" communication infrastructure for integrating applications. Message brokers provide flexibility in routing, as well as other features that support the integration of enterprise applications. This functionality, together with asynchronous messaging, is exactly what is needed in generic EAI settings, and message brokers are thus emerging as the dominant EAI tool used today.

3.2.1 Historical Background

Message brokers are direct descendants of the platforms for message oriented middleware discussed in Chapter 2. They are derived from the new requirements posed by EAI, in terms of supporting the integration of heterogeneous, coarse-grained enterprise applications such as enterprise resource planning (ERP) and CRM systems. Indeed, as soon as the problems behind EAI (exemplified in the previous section) were recognized, the limitations of using MOM systems to support EAI become manifest. Specifically, MOM did not provide support for defining sophisticated logic for routing messages across different systems and did not help developers to cope with the heterogeneity.

In response to these needs, message brokers extend MOM with the capability of attaching logic to the messages and of processing messages directly at the middleware level. In message-oriented middleware the task of the middleware is to move messages from one point to another with certain guarantees. A message broker not only transports the messages, but is also in charge of routing, filtering, and even processing the messages as they move across the system. In addition, most message brokers provide *adapters* that mask heterogeneity and make it possible to access all systems with the same programming model and data exchange format. The combination of these two factors was seen as key to supporting EAI.

The first examples of modern message brokers were developed in the early and mid-1990s, as soon as the need for EAI was recognized. In the beginning, the area was dominated by startup companies such as ActiveSoftware, founded in 1995 and later acquired by WebMethods. With time, software heavyweights such as IBM entered this profitable area, typically by enhancing existing MOM infrastructure. The final "blessing" came from the Java Message Service, a Java API that provides a standard way to interact with a message broker, as least for the basic message broker functionality [194]. Examples of leading commercial implementations of EAI platforms today are Tibco ActiveEnterprise [199], BEA WebLogic Integration [18], WebMethods Enterprise [213], and WebSphere MQ [100].

3.2.2 The Need for Message Brokers

To better understand the limitations of RPC-based and basic MOM systems, let us again consider the execution of a supply chain operation where a company receives a *purchase order* (PO) from a customer and needs to fulfill it. Many different systems will need to process the PO, including, for instance, inventory management applications (to check availability), ERP systems to manage payments, and the shipping application that interfaces with the shipping department to arrange for delivery of the goods. With RPC and message queuing systems, the application that receives and dispatches the PO needs to get a reference (either statically or dynamically) to these three PO processing applications and either invoke one of their methods or send them a message (Figure 3.2). Now, assume that because of a change in the business or IT environment, company policies require additional applications to be notified of the PO. An example of such an application could be a *month-end closing* system that performs book-keeping operations and provides monthly summaries of revenues and profits. In this case, the dispatching application needs to be modified to cope with the change and include the code for notifying this additional system.

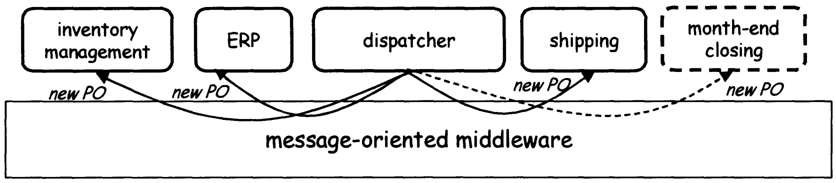

Fig. 3.2. With RPC or message-based interoperability, applications need to be changed if they need to interoperate with a new system (*dashed*)

As another even more "dynamic" example, consider an application that monitors the stock market and interoperates with other applications interested in stock price changes, such as systems that manage investments for stock brokers. In such a scenario, the number of applications interested in stock price modifications changes continuously, as a result of changes in the number of brokers or in the brokers' investing strategies. Therefore, using a basic MOM system to achieve interoperation between the stock monitor and the brokers' applications would be extremely complex, involving the generation of many messages whose number and destination queues are a priori unknown and continuously changing. The problem becomes even more complex if not one but multiple applications can generate the *new PO* or *stock price change* messages, as all these applications then need to be changed.

3.2.3 Extending Basic MOM

The cause of the problem in the above examples is that, in a basic MOM system, the responsibility for defining the receiver of a message lies with the sender. As we have seen, this sort of point-to-point addressing scheme becomes increasingly complex to manage as the number of senders and recipients grows and as the environment becomes more dynamic.

Message brokers are enhanced MOM systems that attempt to overcome this limitation by factoring the message routing logic out of the senders and placing it into the middleware (Figure 3.3). In fact, with a message broker, users can define application logic that identifies, for each message, the queues to which it should be delivered. In this way, senders are not required to specify the intended recipients of a message. Instead, it is up to the message broker to identify the recipients by executing user-defined rules.

The advantage of this approach is that regardless of how many applications can dispatch *new PO* or *stock price change* messages, there is now a single place where we need to make changes when the routing logic for these messages needs to be modified. Later in this section we will see that, using message brokers, there are ways to avoid even this maintenance effort.

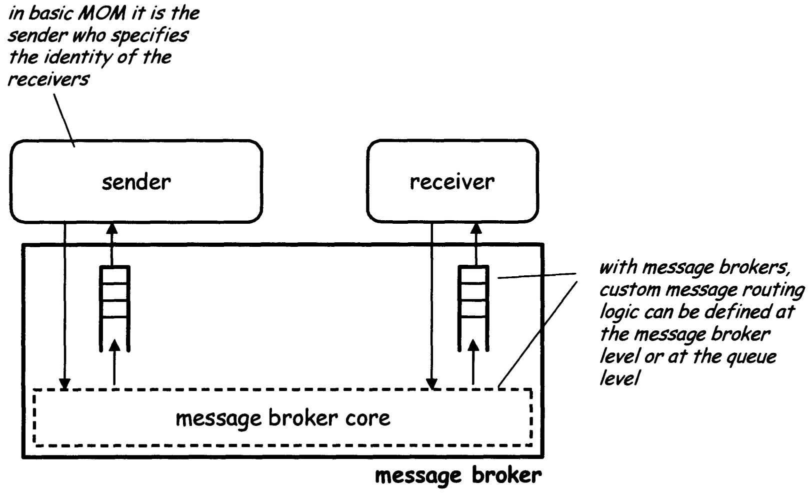
in basic MOM it is the sender who specifies the identity of the receivers
sender
receiver
with message brokers, custom message routing logic can be defined at the message broker level or at the queue level
message broker core
message broker

Fig. 3.3. Message brokers enable users to define custom message routing logic

Routing logic can be based on the sender's identity, on the message type, or on the message content. It is typically defined in a rule-based language, where each rule includes a Boolean condition to be evaluated over the message data and an action defining the queues to which messages satisfying the condition should be delivered. This logic can be defined at the message broker level or at the queue level. If defined at the message broker level, it applies to all messages that are then routed accordingly. If it is associated with a specific queue, it defines the kind of messages that the queue is interested in receiving.

The above discussion shows that message brokers can *decouple* senders and receivers. Senders do not specify and are not aware of which applications will receive the messages they send, and, conversely, receivers may or may not be aware of which applications are capable of sending messages to them.

Note that MOM systems that support shared queues also provide a limited form of decoupling, in that applications send messages to queues rather than to specific recipients. However, with shared queues, each message is delivered to at most one application. Applications taking messages from a shared queue are typically of the same type (or different threads of the same process), and shared queues have the purpose of load balancing. Indeed, shared queues may be combined with message brokering to provide both decoupling and load balancing: messages can be delivered to several queues, depending on the routing logic (decoupling). Multiple threads or applications can then share the load of retrieving and processing messages from a queue (load balancing).

Since in MOM systems (and in message brokers) communication between applications goes through a middle layer, it is possible to implement even more application-specific functionality in this layer, going beyond routing rules. For example, another reason for associating logic with queues is to enable the definition of *content transformation* rules. Refer again to the PO processing example: in that case, routing is only part of the problem. Another issue to be handled is that different applications support different data formats. For example, an application may assume that the weight of goods to be shipped is expressed in pounds, while another may require the weight to be expressed in kilograms. By defining content transformation rules and by associating them with the queue, it is possible to factorize these mappings in the message broker, as opposed to having each application perform them. Every application pulling messages from the queue will now receive the weight expressed in kilograms, as desired. By assigning different transformation rules to each queue, we can accommodate the needs of different applications without modifying these applications.

In general, there is no limit to the amount of application logic that can be "embedded" into the broker or into the queue. However, placing application logic into the broker is not always a good idea. In fact, although on the one hand we make applications more generic and robust to changes, we embed the integration logic into the message brokers' queues. Distributing such logic into the queues makes it difficult to debug and maintain, as no tools are provided to support this effort. Another problem is that of performance: although message brokers tend to be quite efficient, if they have to execute many application-specific rules each time a message is delivered, the overall latency and throughput is degraded. Finally, another limitation of message brokers is their inability to handle large messages. This is perhaps due to the fact that they have been designed to support OLTP-like interactions, which are short-lived and have a light payload. Whenever they are used to route large messages, their performance is greatly affected.

3.2.4 The Publish/Subscribe Interaction Model

Thanks to the possibility of defining application-specific routing logic, message brokers can support a variety of different message-based interaction models. Among those, perhaps the most well-known and widely adopted one is the *publish/subscribe* paradigm. In this paradigm, as in message-based interaction, applications communicate by exchanging messages, again characterized by a type and a set of parameters. However, applications that send messages do not specify the recipients of the message. Instead, they simply *publish* the message to the middleware system that handles the interaction. For this reason, applications that send messages are called *publishers*. If an application is interested in receiving messages of a given type, then it must *subscribe* with the publish/subscribe middleware, thus registering its interest. Whenever a publisher sends a message of a given type, the middleware retrieves the list of all applications that subscribed to messages of that type, and delivers a copy of the message to each of them.

Figure 3.4 exemplifies publish/subscribe interaction in the PO processing example described above. As the figure shows, the PO processing application simply needs to send a message to the message broker, publishing a notification that a new PO has been received. All applications interested in new PO notifications will then receive the message, provided that they subscribed to the *new PO* message type prior to the time the message was sent.

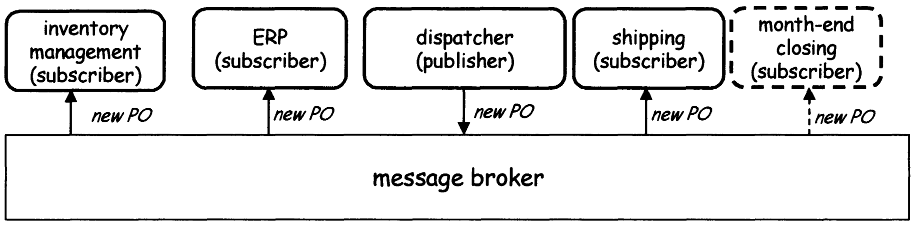

Fig. 3.4. Publish/subscribe models make interoperability more flexible and robust to changes

In a publish/subscribe model, subscribers have two main ways to define the messages they are interested in receiving. The first is to specify a message *type*, such as *new PO*. In simple cases, the type namespace is flat and is defined by a string. More sophisticated systems may allow the types' namespace to be structured into a type/subtype hierarchy of arbitrary depth. For example, if we assume that the type hierarchy is encoded by means of dot-separated strings of the form *type.subtype.subsubtype...*, then a legal message type name could be *Supply Chain.new PO*. With structured types, subscribers not only can register their interest in messages having a specific type and subtype, but can also subscribe to messages whose type T has another type A as ancestor in the type hierarchy. For example, a subscription to *Supply Chain.new PO* specifies interest in receiving all messages related to new purchase orders, while *Supply Chain.** is a more general subscription that declares interest in all supply chain-related messages, including but not limited to new purchase orders.

The second form of subscription is parameter-based: subscribers specify the messages they want to receive by means of a boolean condition on the message parameters. For example, the condition *type="new PO" AND customer="ACME Co." AND quantity>1200* specifies a subscription to all *new PO* messages related to orders by *ACME Co.* whose volume is above 1200 units.

Virtually every message broker today supports the publish/subscribe interaction paradigm. There have even been attempts to standardize the programming abstractions and the interfaces (APIs) between applications and publish/subscribe middleware infrastructure. JMS [194] is one such effort that is part of Java-related standards from Sun. The JMS specifications include a publish/subscribe API in addition to a point-to-point one. In JMS, publications and subscriptions are based on the notion of *topic*. A topic is analogous to a message broker's queue: it is identified by a string (such as "new PO"), and it is the entity to which clients bind to send and receive messages. The difference between topics and queues is that multiple recipients can subscribe to the same topic and receive the same message.

3.2.5 Distributed Administration of a Message Broker

Message broker systems include support for an *administrator*, a distinguished user that has the authority to define (1) the types of messages that can be sent and received, and (2) which users are authorized to send and/or receive a message and to customize routing logic. Administrators are also present in MOMs, but they are more relevant in message brokers due to the decoupling between senders and receivers that, in general, causes the senders to be unaware of which applications will receive the message. Publish/subscribe systems may, however, allow publishers to define limitations on the set of users that can receive a certain message.

Message broker architectures are naturally extensible to meet the needs of communication-intensive applications that span different administrative domains (possibly corresponding to different departments or companies). In fact, it is possible to compose several message brokers, as shown in Figure 3.5. In this architecture, a message broker (say, message broker MB-A) can be a client of another message broker MB-B, and vice versa. If a client of MB-B wants to receive messages sent by clients of MB-A, then it subscribes with MB-B. In turn, MB-B subscribes for the same message type with MB-A. When one of MB-A clients publishes the message of interest, then MB-A will deliver it to MB-B, in accordance with the subscription. As soon as MB-B receives the message, it will deliver it to all of its clients that subscribed for such a message. Note that from the perspective of MB-B, MB-A is conceptually just like any other subscriber. The only difference is that it belongs to a different administrative domain, and therefore the administrator of broker MB-A will set up subscription and publication permissions for this client accordingly. System administration and security are kept simple and modular with this approach, since each message broker administrator only needs to determine which messages can be sent to or received from another domain.

3.2.6 EAI with a Message Broker

Figure 3.6 shows the basic principles on which most EAI platforms are based. There are two fundamental components:

- **Adapters.** Adapters map heterogeneous data formats, interfaces, and protocols into a common model and format. The purpose of adapters is to hide heterogeneity and present a uniform view of the underlying heterogeneous world. A different adapter is needed for each type of application that needs to be integrated