very name implies what object monitors really are: an *object* broker and a TP-*monitor*. This marriage between object brokers and TP monitors was an obvious step to take at the time and, in many cases, the only feasible way to obtain commercially competitive products.

Part of the problem encountered by object brokers, particularly in the case of CORBA, is that the only real novelty they offered was object orientation as a way to standardize interfaces across different systems and programming languages. In fact, CORBA was meant to be implemented on top of conventional middleware platforms such as DCE, TP monitors, and even primitive forms of message-oriented middleware. Unfortunately, programming paradigms are a small part of the picture in a distributed information system. Many of the CORBA *services* were mere specifications that took quite a long time to be implemented in real products. The Object Transactional Service (OTS) of CORBA is probably the most illustrative case of the dilemma faced by object brokers. OTS essentially described what TP monitors had been doing very successfully for many years before CORBA appeared. Since the first commercial products available were typically systems implemented almost from scratch, when compared with already existing middleware platforms, object brokers were extremely inefficient and lacked key functionality such as transactions. This turned out to be a decisive factor limiting the adoption of object brokers.

The way out of this dilemma was to actually use TP monitors and other forms of middleware with an additional layer that would make them object-oriented (and, in some cases, CORBA compliant). When TP monitors were used, the result were object monitors. In terms of functionality, however, object monitors offered very little over what TP monitors already offered. This was the first step toward the assimilation of object broker ideas into other forms of middleware. Java, C#, and the middleware environments around them represent the culmination of this convergence process.

2.5 Message-Oriented Middleware

The previous chapters and sections have presented interoperability concepts and techniques that are mainly based on synchronous method invocation, where a client application invokes a method offered by a specific, although possibly dynamically selected, service provider. When the service provider has completed its job, it returns the reply to the client. In this section we explore abstractions supporting more dynamic and asynchronous forms of interaction, along with the corresponding middleware platforms.

2.5.1 Historical Background

Message-oriented middleware is often presented as a revolutionary technology that may change the way distributed information systems are built. The idea is, however, not new. Originally, asynchronous interaction was used to implement batch systems. RPC implementations already offered asynchronous versions of RPC, and many TP monitors had queuing systems used to implement message-based interaction. For instance, the first versions of the TP monitor Tuxedo were based on queues. Furthermore, the notion of a persistent queue was already well understood at the beginning of the 1990s [27].

Modern message-oriented middleware is, for the most part, a direct descendant of the queuing systems found in TP monitors. In TP monitors, queuing systems were used to implement batch processing systems. But as TP monitors were confronted with the task of integrating a wider range and larger numbers of systems, it quickly became obvious that asynchronous interaction was a more useful way to do this than RPC. As computer clusters started to gain more ground as platforms for distributed information systems, the queuing systems of TP monitors started to play a bigger role when designing large information systems. Eventually they became independent systems on their own. Today, most large integration efforts are done using message-oriented middleware. As discussed in Chapter 6, messages may also become the preferred way of implementing Web services.

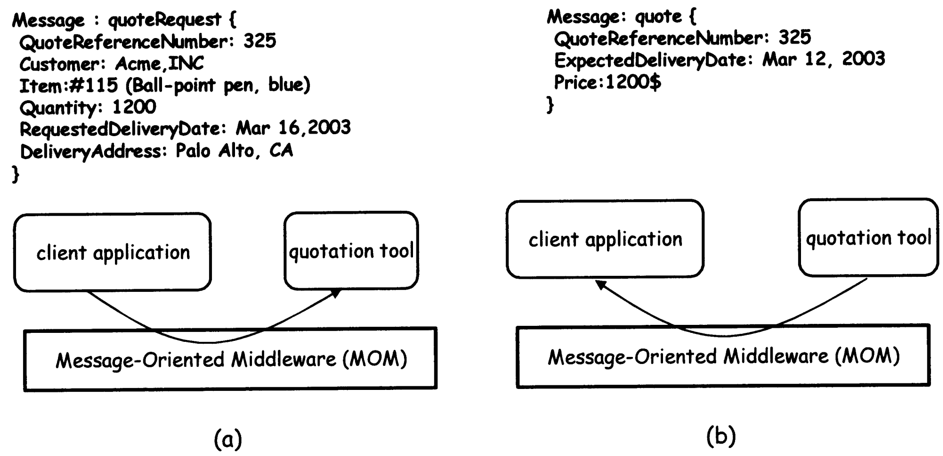
Some of the best-known MOM platforms include IBM WebSphere MQ (formerly known as MQ Series) [100], MSMQ by Microsoft [137], or WebMethods Enterprise by WebMethods [213]. CORBA also provides its own messaging service [155].

2.5.2 Message-Based Interoperability

The term *message-based interoperability* refers to an interaction paradigm where clients and service providers communicate by exchanging *messages.* A message is a structured data set, typically characterized by a *type* and a set of <name,value> pairs that constitute the message *parameters.* The type used to be system dependent; nowadays, most products use XML types. As an example, consider a message that requests a quotation from a vendor about the price of a set of products. The message parameters include the name of the requesting company, the item for which a quote is being requested, the quantity needed, and the date on which the items should be delivered at a specified address. In general, the language for defining message types varies with the messaging platform adopted.

```
Message quoteRequest {
 QuoteReferenceNumber: Integer
 Customer: String
 Item: String
 Quantity: Integer
 RequestedDeliveryDate: Timestamp
 DeliveryAddress: String
}
```

To show how message-based interoperability works, we assume that an application receives the customers' requests for a quote and has to transfer them to a quotation system, to retrieve the quote, and notify the requesting customer. This interaction is depicted in Figure 2.11. We refer to the quotation system as being the service provider, while the application receiving the customer's request is the client in this case.



Message : quoteRequest {
QuoteReferenceNumber: 325
Customer: Acme,INC
Item:#115 (Ball-point pen, blue)
Quantity: 1200
RequestedDeliveryDate: Mar 16,2003
DeliveryAddress: Palo Alto, CA
}
client application
quotation tool
Message-Oriented Middleware (MOM)
(a)
Message: quote {
QuoteReferenceNumber: 325
ExpectedDeliveryDate: Mar 12, 2003
Price:1200$
}
client application
quotation tool
Message-Oriented Middleware (MOM)
(b)

Fig. 2.11. Example of message-based interoperability: an application sends a message to a quotation tool (**a**). The tool serves the request by sending another message (**b**)

With message-based interoperability, once clients and service providers agree on a set of message types, they can communicate by exchanging messages. To request a service, the client application sends a message (for example, a *quoteRequest* message) to the desired provider. The service provider will receive the message, perform appropriate actions depending on the message content (for example, determine a quote), and send another message with the required information back to the client.

The class of middleware applications that support message-based interoperability is called *message-oriented middleware* (MOM). Note that although we use the terms client and service provider, this distinction is blurred in pure message-oriented interactions, at least from the perspective of the middleware. Indeed, to the MOM, all objects look alike; i.e., they send and receive messages. The difference between "clients" and "service providers" is purely conceptual and can only be determined by humans who are aware of the semantics of the messages and of the message exchange. This is different with respect to the other forms of interaction discussed earlier, where objects acting as clients invoke methods provided by other objects, acting as servers.

2.5.3 Message Queues

MOM, per se, does not provide particular benefits with respect to other forms of interactions presented earlier in the book. However, it forms the basis on which many useful concepts and features can be developed, considerably simplifying the development of interoperable applications and providing support for managing errors or system failures. Among these, one of the most important abstractions is that of *message queuing.*

In a message queuing model, messages sent by MOM clients are placed into a queue, typically identified by a name, and possibly bound to a specific intended recipient. Whenever the recipient is ready to process a new message, it invokes the suitable MOM function to retrieve the first message in the queue (Figure 2.12).

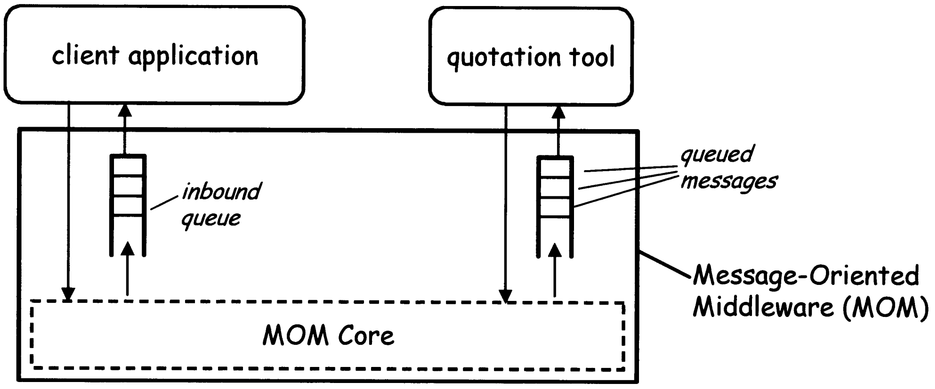


Fig. 2.12. Message queuing model

Queuing messages provide many benefits. In particular, it gives recipients control of when to process messages. Recipients do not have to be continuously listening for messages and process them right away, but can instead retrieve a new message only when they can or need to process it. An important consequence is that queuing is more robust to failures with respect to RPC or object brokers, as recipients do not need to be up and running when the message is sent. If an application is down or unable to receive messages, these will be stored in the application's queue (maintained by the MOM), and they will be delivered once the application is back online and pulls them from the queue. Of course, this also means that the messaging infrastructures must themselves be designed to be very reliable and robust with regard to failures. Queued messages may have an associated expiration date or interval. If the message is not retrieved before the specified date (or before the interval has elapsed), it is discarded.

Queues can be shared among multiple applications, as depicted in Figure 2.13. This approach is typically used when it is necessary to have multiple applications provide the same service, so as to distribute the load among them and improve performance. The MOM system controls access to the queue, ensuring that a message is delivered to only one application. The queuing abstraction also enables many other features. For example, senders can assign *priorities* to messages, so that once the recipient is ready to process the next message, messages with higher priorities are delivered first.

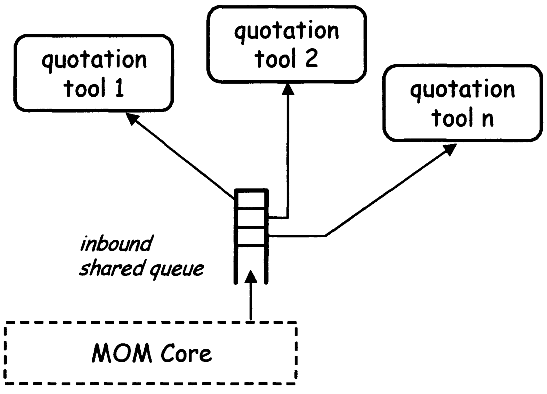


Fig. 2.13. Message queuing model with shared queues

2.5.4 Interacting with a Message Queuing System

Queuing systems provide an API that can be invoked to send messages or to wait for and receive messages. Sending a message is typically a non blocking operation, and therefore once an object has sent a message to another object, it can continue processing. Receiving a message is instead often a blocking operation, where the receiving object "listens" for messages and processes them as they arrive, typically (but not necessarily) by activating a new dedicated thread, while the "main" thread goes back to listen for the next message. Recipients can also retrieve messages in a non blocking fashion by providing a callback function that is invoked by the MOM each time a message arrives. Note therefore that this approach, unlike basic RPC, is naturally asynchronous.

Java programmers can use an industry-standard API for interacting with MOM systems: the Java Message Service (JMS) [194]. In JMS, a message is characterized by a *header*, which includes metadata such as the message type, expiration date, and priority; by an optional set of *properties* that extend the header metadata attributes, for example, to support compatibility with a specific JMS implementation; and a *body*, which includes the actual application-specific information that needs to be exchanged. In JMS, as in most MOM systems, addressing is performed through queues: senders (receivers) first *bind* to a queue, i.e., identify the queue to which they want to send messages (receive messages from), based on the queue name. Then, they can start sending (retrieving) messages to (from) the queue.

As already indicated, JMS is simply an API and not a platform. In fact, several, but not all, MOM products are JMS compliant. JMS can be implemented as a stand-alone system or as a module within an application server (Chapter 4). For instance, the Java Open Reliable Asynchronous Messaging (JORAM, an open source implementation) [109], is an example of a system that can be used as both stand-alone or as part of an application server. Another open source implementation is *JBossMQ* [108]. There are also several commercial implementations such as FioranoMQ [71].

2.5.5 Transactional Queues

Another very important MOM feature, aimed at providing robustness in the face of errors and failures, is *transactional queuing* (sometimes called reliable messaging). Under the transactional queuing abstraction, the MOM ensures that once a message has been sent, it will be eventually delivered once and only once to the recipient application, even if the MOM system itself goes down between the time the message is notified and the time it is delivered. Messages are saved in a persistent storage and are therefore made available once the MOM system is restarted.

In addition to providing guaranteed delivery, transactional queuing provides support for coping with failures. In fact, recipients can bundle a set of message retrievals and notifications within an *atomic* unit of execution. As described earlier, an atomic unit identifies a set of operations that have the all-or-nothing property: either all of them are successfully executed, or none are. In case a failure occurs before all operations in the atomic unit have been executed, all completed operations are rolled back (undone). In message queuing, rolling back a message retrieval operation corresponds to placing the message back in the queue, so that it can be then consumed again by the same application or, in case of shared queues, by other applications. From the perspective of the sender, transactional queues mean that messages sent within a sender's atomic unit are maintained in a persistent storage by the MOM and are made visible for delivery only when the execution of the atomic unit is completed. Therefore, rolling back message notifications simply involves deleting the messages from the persistent storage.

For example, assume that a set of quotation applications $A = \{a_1, a_2, ..., a_n\}$ retrieves *quoteRequest* messages from a queue, and that the message retrieval operation is within an atomic unit that also involves getting pricing and availability information from the product suppliers, preparing a *quote* message, and sending the message back to the requester. If an application a_j retrieves message *quoteRequest*$_k$ and is unable to obtain the quote, either because the application a_j itself fails, or because for any reason it is unable to obtain pricing and availability information from suppliers, then message *quoteRequest*$_k$

is put back into the queue, so that other applications in set A can later retrieve message $quoteRequest_k$ again in an attempt to fulfill the request.

We observe here that although transactional queuing is a very useful mechanism to overcome short-lived failures, additional higher-level abstractions are in general needed to manage exceptional situations: indeed, putting a message back in the queue does guarantee that the message is not discarded and that it will be eventually processed again, enabling service providers to overcome problems that have a short time span. However, if the inability to process the message correctly is due to a cause other than a temporary failure, placing the message back in the queue neither addresses the problem nor provides feedback to the requester about the inability to complete the service. For example, if application a_j is unable to determine a quote because no supplier can deliver the requested goods by the specified delivery date, then rolling back the operations and putting the message back in the queue is not going to help in any way.

2.6 Summary

There are two important aspects to middleware that are sometimes blurred, but that should be differentiated. On the one hand, middleware provides programming abstractions for designing distributed applications. These abstractions can range from more sophisticated communication models (e.g., remote procedure calls instead of sockets), to transactions, queues for asynchronous interactions, or automatic properties conferred to the code (e.g., security or persistence). On the other hand, middleware implements the functionality provided by the programming abstractions. This might include transactional support, specialized communication primitives, name and directory services, persistence, and so on. Here we are no longer discussing the programming abstractions, but rather the mechanisms necessary to implement such programming abstractions. The type of middleware, its capabilities, and how it is used depend very much upon these two aspects.

In practice, however, middleware platforms differ much more in terms of the programming abstractions they provide than in their underlying infrastructure. In this chapter we took an evolutionary perspective when discussing the different forms of middleware available today. We started with RPC and progressed to TP monitors, object brokers, and message-oriented middleware. In this progression it is important to keep in mind the motivation behind each step. RPC provides a very basic form of distribution that initially was devised to avoid having to deal with the low-level communication interfaces available in UNIX (sockets). Soon, the designers of RPC realized that additional infrastructure was necessary to support a higher level of programming abstraction. TP monitors took advantage of RPC to extend to distributed environments the functionality they offered in large computers. By doing so, they became the predominant form of middleware, a role that they still enjoy today. Changes in programming paradigms and the increasing opportunities for distribution triggered the emergence of object brokers. At the same time, object brokers became the second attempt to standardize middleware platforms (the first was DCE, which tried to standardize the implementation for RPC). For practical reasons, few object brokers made it on their own. Their ideas and specifications were eventually taken over by TP monitors and have been by now almost completely superseded by the .NET and J2EE initiatives. Finally, to cope with the increasing use of cluster-based architectures and the greater demand for enterprise application integration solutions, the queuing systems found in TP monitors became stand-alone systems that soon evolved toward MOM systems and later into what we know today as message brokers. As we will see in Chapter 3, message brokers play a crucial role among the platforms used for enterprise application integration.