1

Message-Oriented Middleware

Edward Curry
National University of Ireland, Galway, Ireland

1.1 Introduction

As software systems continue to be distributed deployments over ever-increasing scales, transcending geographical, organizational, and traditional commercial boundaries, the demands placed upon their communication infrastructures will increase exponentially. Modern systems operate in complex environments with multiple programming languages, hardware platforms, operating systems and the requirement for dynamic flexible deployments with 24/7 reliability, high throughput performance and security while maintaining a high Quality-of-Service (QoS). In these environments, the traditional direct Remote Procedure Call (RPC) mechanisms quickly fail to meet the challenges present.

In order to cope with the demands of such systems, an alternative to the RPC distribution mechanism has emerged. This mechanism called Message-Oriented Middleware or MOM provides a clean method of communication between disparate software entities. MOM is one of the cornerstone foundations that distributed enterprise systems are built upon. MOM can be defined as any middleware infrastructure that provides messaging capabilities.

A client of a MOM system can send messages to, and receive messages from, other clients of the messaging system. Each client connects to one or more servers that act as an intermediary in the sending and receiving of messages. MOM uses a model with a peer-to-peer relationship between individual clients; in this model, each peer can send and receive messages to and from other client peers. MOM platforms allow flexible cohesive systems to be created; a cohesive system is one that allows changes in one part of a system to occur without the need for changes in other parts of the system.

1.1.1 Interaction Models

Two interaction models dominate distributed computing environments, synchronous and asynchronous communication. This section introduces both interaction models; a solid

Middleware for Communications. Edited by Qusay H. Mahmoud
© 2004 John Wiley & Sons, Ltd ISBN 0-470-86206-8

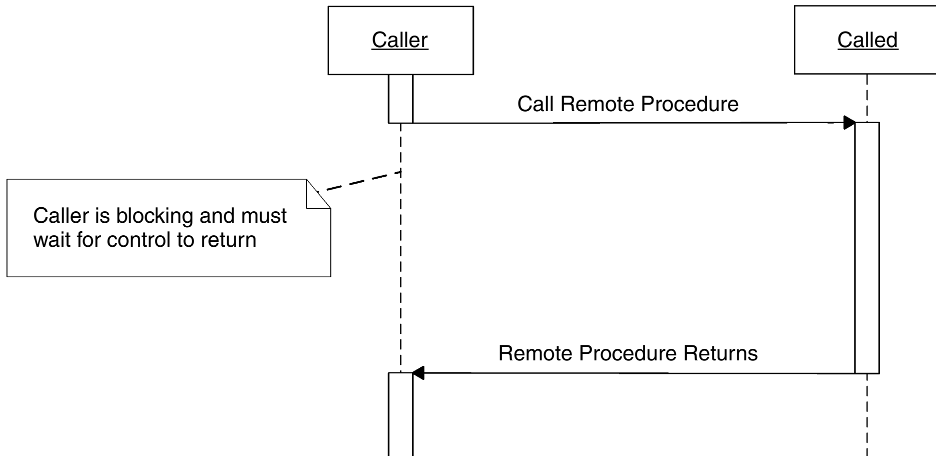

Figure 1.1 Synchronous interaction model

knowledge of these models and the differences between them is key to understanding the benefits and differences between MOM and the forms of distribution available.

1.1.2 Synchronous Communication

When a procedure/function/method is called using the synchronous interaction model, the caller code must block and wait (suspend processing) until the called code completes execution and returns control to it; the caller code can now continue processing. When using the synchronous interaction model, as illustrated in Figure 1.1, systems do not have processing control independence; they rely on the return of control from the called systems.

1.1.3 Asynchronous Communication

The asynchronous interaction model, illustrated in Figure 1.2, allows the caller to retain processing control. The caller code does not need to block and wait for the called code to return. This model allows the caller to continue processing regardless of the processing state of the called procedure/function/method. With asynchronous interaction, the called code may not execute straight away. This interaction model requires an intermediary to handle the exchange of requests; normally this intermediary is a message queue.

While more complex than the synchronous model, the asynchronous model allows all participants to retain processing independence. Participants can continue processing, regardless of the state of the other participants.

1.1.4 Introduction to the Remote Procedure Call (RPC)

The traditional RPC model is a fundamental concept of distributed computing. It is utilized in middleware platforms including CORBA, Java RMI, Microsoft DCOM, and XML-RPC. The objective of RPC is to allow two processes to interact. RPC creates the façade of making both processes believe they are in the same process space (i.e., are

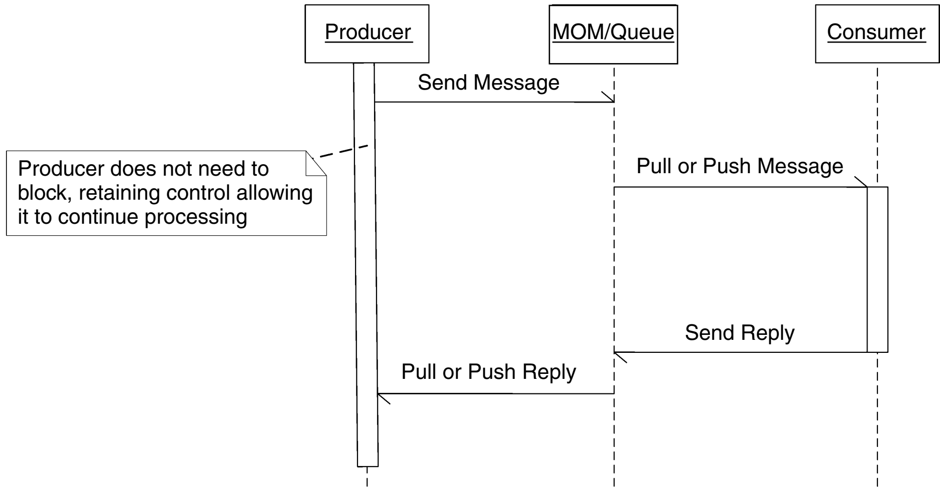

Figure 1.2 Asynchronous interaction model

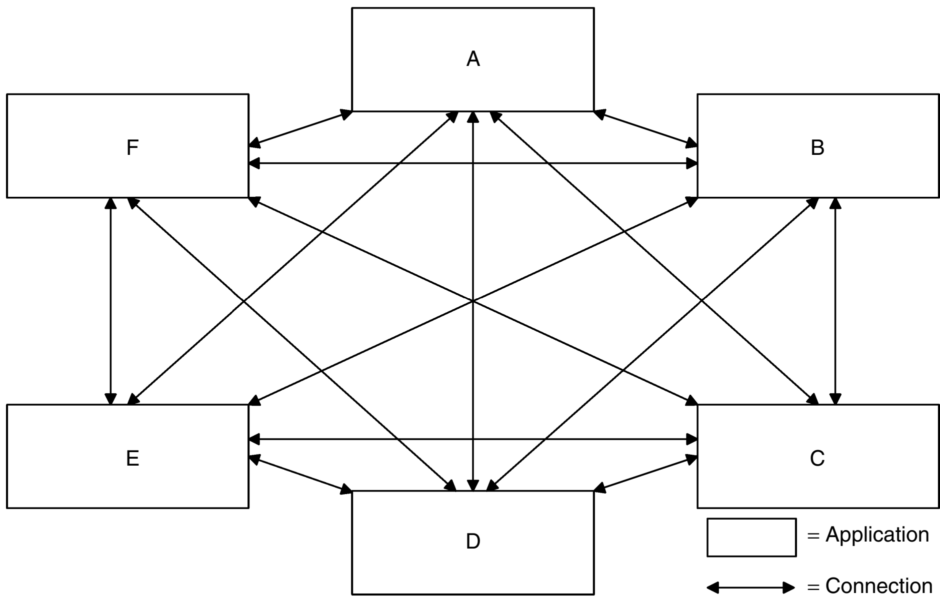

Figure 1.3 An example remote procedure call deployment

the one process). On the basis of the synchronous interaction model, RPC is similar to a local procedure call whereby control is passed to the called procedure in a sequential synchronous manner while the calling procedure blocks waiting for a reply to its call. RPC can be seen as a direct conversation between two parties (similar to a person-to-person telephone conversation). An example of an RPC-based distributed system deployment is detailed in Figure 1.3.

1.1.4.1 Coupling

RPC is designed to work on object or function interfaces, resulting in the model producing tightly coupled systems as any changes to the interfaces will need to be propagated thought the code base of both systems. This makes RPC a very invasive mechanism of distribution. As the number of changes to source or target systems increase, the cost will increase too. RPC provides an inflexible method of integrating multiple systems.

1.1.4.2 Reliability

Reliable communications can be the most important concern for distributed applications. Any failure outside of the application – code, network, hardware, service, other software or service outages of various kinds (network provider, power, etc) – can affect the reliable transport of data between systems. Most RPC implementations provide little or no guaranteed reliable communication capability; they are very vulnerable to service outages.

1.1.4.3 Scalability

In a distributed system constructed with RPC, the blocking nature of RPC can adversely affect performance in systems where the participating subsystems do not scale equally. This effectively slows the whole system down to the maximum speed of its slowest participant. In such conditions, synchronous-based communication techniques such as RPC may have trouble coping when elements of the system are subjected to a high-volume burst in traffic. Synchronous RPC interactions use more bandwidth because several calls must be made across the network in order to support a synchronous function call. The implication of this supports the use of the asynchronous model as a scalable method of interaction.

1.1.4.4 Availability

Systems built using the RPC model are interdependent, requiring the simultaneous availability of all subsystems; a failure in a subsystem could cause the entire system to fail. In an RPC deployment, the unavailability of a subsystem, even temporally, due to service outage or system upgrading can cause errors to ripple throughout the entire system.

1.1.5 Introduction to Message-Oriented Middleware (MOM)

MOM systems provide distributed communication on the basis of the asynchronous interaction model; this nonblocking model allows MOM to solve many of the limitations found in RPC. Participants in a MOM-based system are not required to block and wait on a message send, they are allowed to continue processing once a message has been sent. This allows the delivery of messages when the sender or receiver is not active or available to respond at the time of execution.

MOM supports message delivery for messages that may take minutes to deliver, as opposed to mechanisms such as RPC (RMI) that deliver in milliseconds or seconds. When using MOM, a sending application has no guarantee that its message will be read

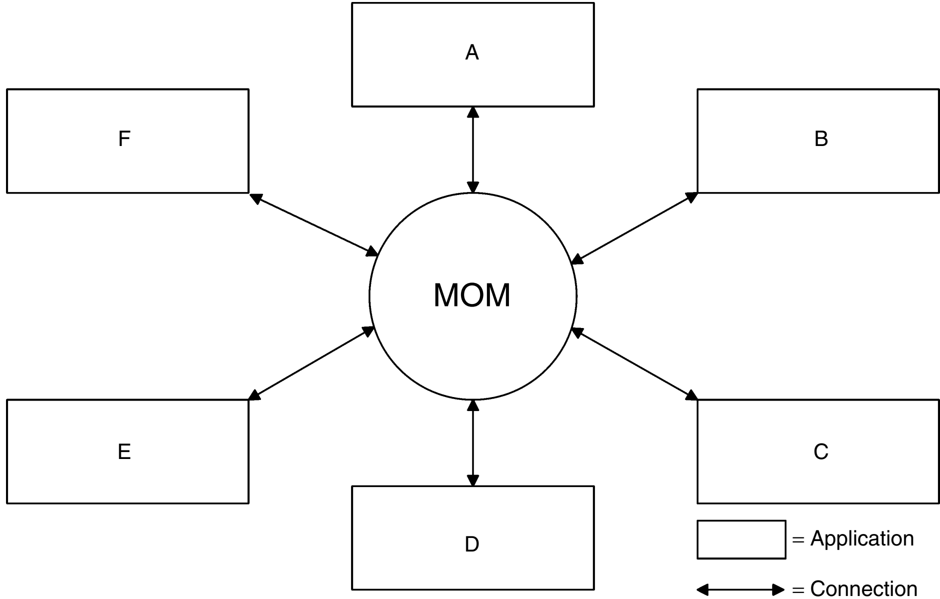

Figure 1.4 An example message-oriented middleware deployment

by another application nor is it given a guarantee about the time it will take the message to be delivered. These aspects are mainly determined by the receiving application.

MOM-based distributed system deployments, as shown in Figure 1.4, offer a service-based approach to interprocess communication. MOM messaging is similar to the postal service. Messages are delivered to the post office; the postal service then takes responsibility for safe delivery of the message [1].

1.1.5.1 Coupling

MOM injects a layer between senders and receivers, which allows message senders and receivers to use this independent layer as an intermediary to exchange messages, see Figure 2.2 for an illustration of this concept. A primary benefit of MOM is the loose coupling between participants in a system – the ability to link applications without having to adapt the source and target systems to each other, resulting in a highly cohesive, decoupled system deployment [2].

1.1.5.2 Reliability

With MOM, message loss through network or system failure is prevented by using a store and forward mechanism for message persistence. This capability of MOM introduces a high level of reliability into the distribution mechanism, store and forward prevents loss of messages when parts of the system are unavailable or busy. The specific level-of-reliability is typically configurable, but MOM messaging systems are able to guarantee

that a message will be delivered, and that it will be delivered to each intended recipient exactly once.

1.1.5.3 Scalability

In addition to decoupling the interaction of subsystems, MOM also decouples the performance characteristics of the subsystems from each other. Subsystems can be scaled independently, with little or no disruption to other subsystems. MOM also allows the system to cope with unpredictable spikes in activity in one subsystem without affecting other areas of the system. MOM messaging models contain a number of natural traits that allow for simple and effective load balancing, by allowing a subsystem to choose to accept a message when it is ready to do so rather than being forced to accept it. This load-balancing technique will be covered in more detail later in the chapter. State-of-the-art enterprise-level MOM platforms have been used as the backbone to create massively scalable systems with support for handling 16.2 million concurrent queries per hour and over 270,000 new order requests per hour [3].

1.1.5.4 Availability

MOM introduces high availability capabilities into systems allowing for continuous operation and smoother handling of system outages. MOM does not require simultaneous or "same-time" availability of all subsystems. Failure in one of the subsystems will not cause failures to ripple throughout the entire system. MOM can also improve the response time of the system because of the loose coupling between MOM participants. This can reduce the process completion time and improve overall system responsiveness and availability.

1.1.6 When to use MOM or RPC

Depending on the scenario they are deployed in, both MOM and RPC have their advantages and disadvantages. RPC provides a more straightforward approach to messaging using the familiar and straightforward synchronous interaction model. However, the RPC mechanism suffers from inflexibility and tight coupling (potential geometric growth of interfaces) between the communicating systems; it is also problematic to scale parts of the system and deal with service outages. RPC assumes that all parts of the system will be simultaneously available; if one part of the system was to fail or even become temporarily unavailable (network outage, system upgrade), the entire system could stall as a result.

There is a large overhead associated with an RPC interaction; RPC calls require more bandwidth than a similar MOM interaction. Bandwidth is an expensive performance overhead and is the main obstacle to scalability of the RPC mechanism [4]. The RPC model is designed on the notion of a single client talking to a single server; traditional RPC has no built-in support for one-to-many communications. The advantage of an RPC system is the simplicity of the mechanism and straightforward implementation. MOM simplifies the process of building dynamic high-flexible enterprise-class distributed systems.

An advantage that RPC has over MOM is the guarantee of sequential processing. With the synchronous RPC model, you can control the order in which processing occurs in the system. For example, in an RPC system you can be sure that at any one time all the new orders received by the system have been added to the database and that they have

been added in the order of which they were received. However, with an asynchronous MOM approach this cannot be guaranteed, as new orders could exist in queues waiting to be added to the database. This could result in a temporal inaccuracy of the data in the database. We are not concerned that these updates will not be applied to the database, but that a snapshot of the current database would not accurately reflect the actual state of orders placed. RPC is slow but consistent; work is always carried out in the correct order. These are important considerations for sections in a system that requires data to have 100% temporal integrity. If this type of integrity is more important than performance, you will need to use the RPC model or else design your system to check for these potential temporal inaccuracies.

MOM allows a system to evolve its operational environment without dramatic changes to the application assets. It provides an integration infrastructure that accommodates functionality changes over time without disruption or compromising performance and scalability. The decoupled approach of MOM allows for flexible integration of clients into a system and support for large numbers of consumers/clients and producer/consumer anonymity. Commercial MOM implementations provide high scalability with support for tens of thousands of clients, advanced filtering, easy integration into heterogeneous networks, and clustering reliability [3].

The RPC method is ideal if you want a strongly typed/Object-Oriented (OO) system with tight coupling, compile-time semantic checking and an overall more straightforward system implementation.

If the distributed systems will be geographically dispersed deployments with poor network connectivity and stringent demands in reliability, flexibility, and scalability, then MOM is the ideal solution.

1.2 Message Queues

The message queue is a fundamental concept within MOM. Queues provide the ability to store messages on a MOM platform. MOM clients are able to send and receive messages to and from a queue. Queues are central to the implementation of the asynchronous interaction model within MOM. A queue, as shown in Figure 1.5, is a destination where messages may be sent to and received from; usually the messages contained within a queue are sorted in a particular order. The standard queue found in a messaging system is the First-In First-Out (FIFO) queue; as the name suggests, the first message sent to the queue is the first message to be retrieved from the queue.

Many attributes of a queue may be configured. These include the queue's name, queue's size, the save threshold of the queue, message-sorting algorithm, and so on. Queuing is of particular benefit to mobile clients without constant network connectivity, for example, sales personnel on the road using mobile network (GSM, GRPS, etc) equipment to remotely send orders to head office or remote sites with poor communication infrastructures. These clients can use a queue as a makeshift inbox, periodically checking the queue for new messages. Potentially each application may have its own queue, or applications may share a queue, there is no restriction on the setup. Typically, MOM platforms support multiple queue types, each with a different purpose. Table 1.1 provides a brief description of the more common queues found in MOM implementations.

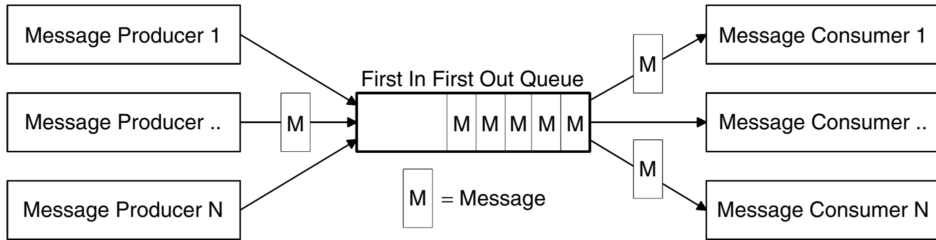

Figure 1.5 Message queue

Table 1.1 Queue formats

Queue type	Purpose
Public Queue	Public open access queue
Private Queue	Require clients to provide a valid username and password for authentication and authorization
Temporary Queue	Queue created for a finite period, this type of queue will last only for the duration of a particular condition or a set time period
Journal Queues	Designed to keep a record of messages or events. These queues maintain a copy of every message placed within them, effectively creating a journal of messages
Connector/Bridge Queue	Enables proprietary MOM implementation to interoperate by mimicking the role of a proxy to an external MOM provider. A bridge handles the translation of message formats between different MOM providers, allowing a client of one provider to access the queues/messages of another
Dead-Letter/Dead-Message Queue	Messages that have expired or are undeliverable (i.e., invalid queue name or undeliverable addresses) are placed in this queue

1.3 Messaging Models

A solid understanding of the available messaging models within MOM is key to appreciate the unique capabilities it provides. Two main message models are commonly available, the point-to-point and publish/subscribe models. Both of these models are based on the exchange of messages through a channel (queue). A typical system will utilize a mix of these models to achieve different messaging objectives.

1.3.1 Point-to-Point

The point-to-point messaging model provides a straightforward asynchronous exchange of messages between software entities. In this model, shown in Figure 1.6, messages from producing clients are routed to consuming clients via a queue. As discussed earlier, the most common queue used is a FIFO queue, in which messages are sorted in the order

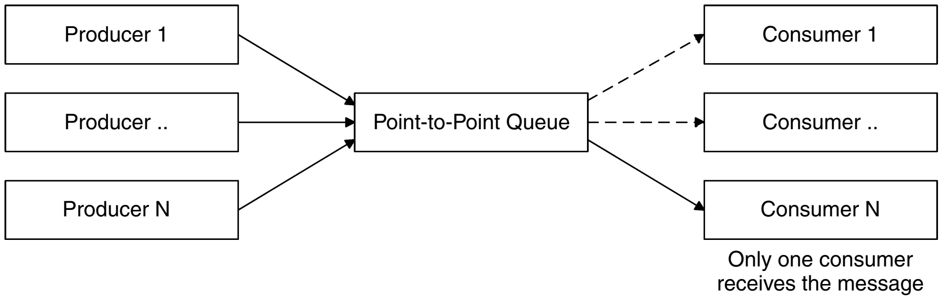

Figure 1.6 Point-to-point messaging model

in which they were received by the message system and as they are consumed, they are removed from the head of the queue.

While there is no restriction on the number of clients who can publish to a queue, there is usually only a single consuming client, although this is not a strict requirement. Each message is delivered only once to only one receiver. The model allows multiple receivers to connect to the queue, but only one of the receivers will consume the message. The techniques of using multiple consuming clients to read from a queue can be used to easily introduce smooth, efficient load balancing into a system. In the point-to-point model, messages are always delivered and will be stored in the queue until a consumer is ready to retrieve them.

Request-Reply Messaging Model

This model is designed around the concept of a request with a related response. This model is used for the World Wide Web (WWW), a client requests a page from a server, and the server replies with the requested web page. The model requires that any producer who sends a message must be ready to receive a reply from consumers at some stage in the future. The model is easily implemented with the use of the point-to-point and publish/subscribe model and may be used in tandem to complement.

1.3.2 Publish/Subscribe

The publish/subscribe messaging model, Figure 1.7, is a very powerful mechanism used to disseminate information between anonymous message consumers and producers. These one-to-many and many-to-many distribution mechanisms allow a single producer to send a message to one user or potentially hundreds of thousands of consumers.

In the publish/subscribe (pub/sub) model, the sending and receiving application is free from the need to understand anything about the target application. It only needs to send the information to a destination within the publish/subscribe engine. The engine will then send it to the consumer. Clients producing messages "publish" to a specific topic or channel, these channels are then "subscribed" to by clients wishing to consume messages. The service routes the messages to consumers on the basis of the topics to which they have subscribed as being interested in. Within the publish/subscribe model, there is no restriction on the role of a client; a client may be both a producer and consumer of a topic/channel.

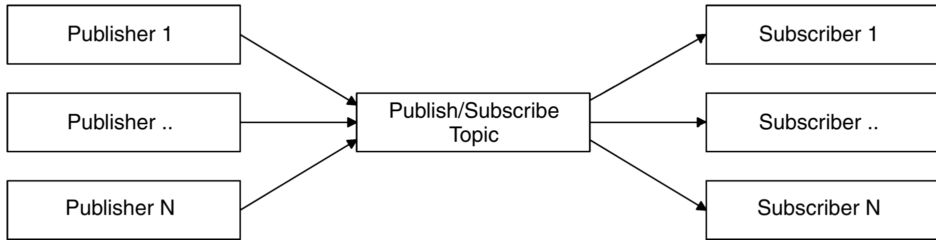

Figure 1.7 Publish/subscribe messaging model

A number of methods for publish/subscribe messaging have been developed, which support different features, techniques, and algorithms for message filtering [5], publication, subscription, and subscription management distribution [6].

PUSH and PULL

When using these messaging models, a consuming client has two methods of receiving messages from the MOM provider.

Pull

A consumer can poll the provider to check for any messages, effectively **pulling** them from the provider.

Push

Alternatively, a consumer can request the provider to send on relevant messages as soon as the provider receives them; they instruct the provider to **push** messages to them.

1.3.2.1 Hierarchical Channel Namespaces

Hierarchical channels or topics are a destination grouping mechanism in pub/sub messaging model. This type of structure allows channels to be defined in a hierarchical fashion, so that they may be nested under other channels. Each subchannel offers a more granular selection of the messages contained in its parent. Clients of hierarchical channels subscribe to the most appropriate level of channel in order to receive the most relevant messages. In large-scale systems, the grouping of messages into related types (i.e., into channels) helps to manage large volumes of different messages [7].

The relationship between a channel and subchannels allows for super-type subscriptions, where subscriptions that operate on a parent channel/type will also match all subscriptions of descendant channels/types. A channel hierarchy for an automotive trading service may be structured by categorizing messaging into buys or sells, then further subcategorization breaking down for commercial and private vehicle types. An example hierarchy illustrating this categorizing structure is presented in Figure 1.8. A subscription to the "`Sell.Private_Vehicles`" channel would receive all messages classified as a private vehicle sale, whereas subscribing to "`Sell.Private_Vehicles.Cars`" would result in only receiving messages classified as a car sale.

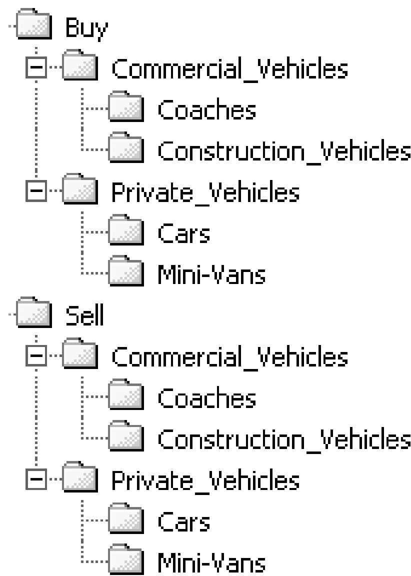

Figure 1.8 An automotive hierarchical channel structure

Hierarchical channels require the channel namespace schema be both well defined and universally understood by the participating parties. Responsibility for choosing a channel in which to publish messages is left to the publishing client. Hierarchical channels are used in routing situations that are more or less static; however, research is underway on defining reflective hierarchies, with adaptive capabilities, for dynamic environments [8].

Consumers of the hierarchy are able to browse the hierarchy and subscribe to channels. Frequently used in conjunction with the publish/subscribe messaging model, hierarchical channels allow for the dissemination of information to a large number of unknown consumers. Hierarchical channels can compliment filtering as a mechanism of routing relevant messages to consumers; they provide a more granular approach to consumer subscription that reduces the number of filters needed to exclude unwanted messages, while supporting highly flexible easy access subject-based routing.

1.3.3 Comparison of Messaging Models

The two models have very similar capabilities and most messaging objectives can be achieved using either model or a combination of both. The fundamental difference between the models boils down to the fact that within the publish/subscribe model every consumer to a topic/channel will receive a message published to it, whereas in point-to-point model only one consumer will receive it. Publish/subscribe is normally used in a broadcast scenario where a publisher wishes to send a message to 1-N clients. The publisher has no real control over the number of clients who receive the message, nor have they a guarantee any will receive it. Even in a one-to-one messaging scenario, topics can be useful to categorize different types of messages. The publish/subscribe model is the more powerful messaging model for flexibility; the disadvantage is its complexity.

In the point-to-point model, multiple consumers may listen to a queue; however, only one consumer will receive each message. However, point-to-point will guarantee that a consumer will receive the message, storing the messages in a queue until a consumer is ready to receive the message; this is known as '*Once and only once messaging*'. While the point-to-point model may not be as flexible as the publish/subscribe model, its power is in its simplicity.

A common application of the point-to-point model is for load balancing. With multiple consumers receiving from a queue, the workload for processing the messages is distributed between the queue consumers. The exact order of how the messages are assigned to consumers is specific to the MOM implementation, but if you utilize a pull model, a consumer will receive a message only when they are ready to process it.

1.4 Common MOM Services

When constructing large-scale systems, it is vital to utilize a state-of-the-art enterprise-level MOM implementation. Enterprise-level messaging platforms will usually come with a number of built-in services for transactional messaging, reliable message delivery, load balancing, and clustering; this section will now give an overview of these services.

1.4.1 Message Filtering

Message filtering allows a message consumer/receiver to be selective about the messages it receives from a channel. Filtering can operate on a number of different levels. Filters use Boolean logic expressions to declare messages of interest to the client, the exact format of the expression depends on the implementation but the WHERE clauses of SQL-92 (or a subset of) is commonly used as the syntax. Filtering models commonly operate on the properties (name/value pairs) of a message; however, a number of projects have extended filtering to message payloads [9]. Message filtering is covered further in the Java Message Service section of this chapter.

Since •there are a number of filter capabilities found in messaging systems, it is useful to note that as the filtering techniques get more advanced, they are able replicate the techniques that proceed them. For example, subject-based filtering is able to replicate channel-based filtering, just like content-based filtering is able to replicate both subject and channel-based filtering.

1.4.2 Transactions

Transactions provide the ability to group tasks together into a single unit of work. The most basic straightforward definition of a transaction is as follows:

All tasks must be completed or all will fail together

In order for transactions to be effective, they must conform to the following properties in Table 1.3, commonly referred to as the **ACID** transaction properties.

In the context of transactions, any asset that will be updated by a task within the transaction is referred to as a resource. A resource is a persistent store of data that is

Table 1.2 Message filters

Filter type	Description
Channel-based	Channel-based systems categorize events into predefined groups. Consumers subscribe to the groups of interest and receive all messages sent to the groups
Subject-based	Messages are enhanced with a tag describing their subject. Subscribers can declare their interests in these subjects flexibly by using a string pattern match on the subject, for example, all messages with a subject starting of "Car for Sale"
Content-based	As an attempt to overcome the limitations on subscription declarations, content-based filtering allows subscribers to use flexible querying languages in order to declare their interests with respect to the contents of the messages. For example, such a query could be giving the price of stock 'SUN' when the volume is over 10,000. Such a query in SQL-92 would be "stock_symbol = 'SUN' AND stock_volume >10,000"
Content-based with Patterns (Composite Events)	Content-based filtering with patterns, also known as *composite events* [10], enhances content-based filtering with additional functionality for expressing user interests across multiple messages. Such a query could be giving the price of stock 'SUN' when the price of stock 'Microsoft' is less than $50

Table 1.3 The properties of a transaction

Atomic	All tasks must complete, or no tasks must complete
Consistent	Given an initial consistent state, a final consistent state will be reached regardless of the result of the transaction (success/fail)
Isolated	Transactions must be executed in isolation and cannot interfere with other concurrent transactions
Durable	The effect of a committed transaction will not be lost subsequent to a provider/broker failure

participating in a transaction that will be updated; a message broker's persistent message store is a resource. A resource manager controls the resource(s); they are responsible for managing the resource state.

MOM has the ability to include a message being sent or received within a transaction. This section examines the main types of transaction commonly found in MOM. When examining the transactional aspects of messaging systems, it is important to remember that MOM messages are autonomous self-contained entities. Within the messaging domain, there are two common types of transactions, Local Transactions and Global Transactions. Local transactions take place within a single resource manager such as a single messaging broker. Global transactions involve multiple, potentially distributed heterogeneous resource managers with an external transaction manager coordinating the transaction.

1.4.2.1 Transactional Messaging

When a client wants to send or retrieve messages within a transaction, this is referred to as *Transactional Messaging*. Transactional messaging is used when you want to perform several messaging tasks (send 1-N messages) in a way that all tasks will succeed or all will fail. When transactional messaging is used, the sending or receiving application has the opportunity to commit the transaction (all the operations have succeeded), or to abort the transaction (one of the operations failed) so all changes are rolled back. If a transaction is aborted, all operations are rolled back to the state when the transaction was invoked.

Messages delivered to the server in a transaction are not forwarded on to the receiving client until the sending client commits the transaction. Transactions may contain multiple messages. Message transactions may also take place in transactional queues. This type of queue is created for the specific purpose of receiving and processing messages that are sent as part of a transaction. Nontransactional queues are unable to process messages that have been included in a transaction.

1.4.2.2 Transaction Roles

The roles played by the message producer, message consumer, and message broker are illustrated in Figure 1.9 and Figure 1.10.

Producer
The producer sends a message or set of messages to the broker. On Commit, the broker stores then sends the message(s) On Rollback, the broker disposes of the message(s).

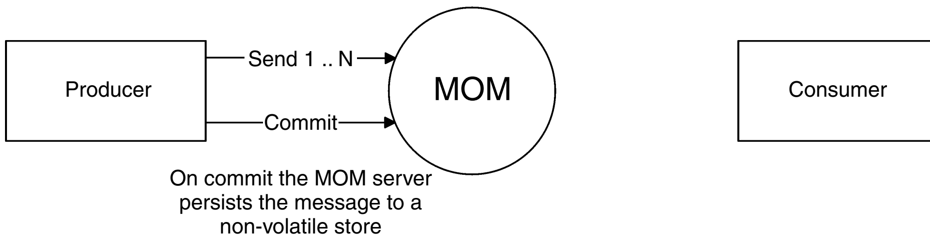

Figure 1.9 Role of a producer in a transaction

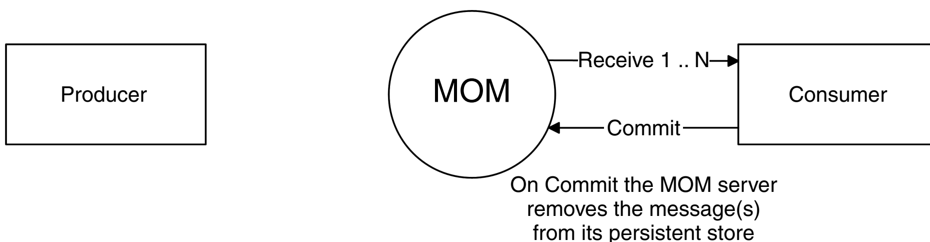

Figure 1.10 Role of a consumer in a transaction

Consumer
The consumer's wish is to receive a message/set of messages from the broker. On Commit, the broker disposes of the set of messages On Rollback, the broker resends the set of messages.

To summarize the roles of each party in a message transaction, the message producer has a contract with the message server; the message server has a contract with the message consumer. Further information on integrating messaging with transactions is available in [11, 12] and Chapter **??** (Transaction Middleware).

1.4.2.3 Reliable Message Delivery

A MOM service will typically allow the configuration of the Quality-of-Service (QoS) delivery semantics for a message. Typically, it is possible to configure a message delivery to be of *at-most once*, *at-least-once*, or *once-and-once-only*. Message acknowledgment can be configured, in addition to the number of retry attempted on a delivery failure. With persistent asynchronous communication, the message is sent to the messaging service that stores it for as long as it takes to deliver the message, unless the *Time-to-Live* (TTL) of the message expires.

1.4.3 Guaranteed Message Delivery

In order for MOM platforms to guarantee message delivery, the platform must save all messages in a nonvolatile store such as a hard disk. The platform then sends the message to the consumer and waits for the consumer to confirm the delivery. If the consumer does not acknowledge the message within a reasonable amount of time, the server will resend the message. This allows for the message sender to "fire and forget" messages, trusting the MOM to handle the delivery. Certified message delivery is an extension of the guaranteed message delivery method. Once a consumer has received the message, a consumption report (receipt) is generated and sent to the message sender to confirm the consumption of the message.

1.4.4 Message Formats

Depending on the MOM implementation, a number of message formats may be available to the user. Some of the more common message types include Text (including XML), Object, Stream, HashMaps, Streaming Multimedia [13], and so on. MOM providers can provide mechanisms for transforming one message format into another and for transforming/altering the format of the message payload; some MOM implementation allows XSL• transformation to be carried out by an XML message payload. Such MOM providers are often referred to as Message brokers and are used to "broker" the difference between diverse systems [14].

1.4.5 Load Balancing

Load balancing is the process of spreading the workload of the system over a number of servers (in this scenario, a server can be defined as a physical hardware machine or

software server instance or both). A correctly load balanced system should distribute work between servers, dynamically allocating work to the server with the lightest load.

Two main approaches of load balancing exist, "push" and "pull". In the push model, an algorithm is used to balance the load over multiple servers. Numerous algorithms exist, which attempt to guess the least-burdened and push the request to that server. The algorithm, in conjunction with load forecasting, may base its decision on the performance record of each of the participating servers or may guesstimate the least-burdened server. The push approach is an imperfect, but acceptable, solution to load balancing a system.

In the pull model, the load is balanced by placing incoming messages into a point-to-point queue, and the consuming servers can then pull messages from this queue at their own pace. This allows for true load balancing, as a server will only pull a message from the queue once they are capable of processing it. This provides the ideal mechanism as it more smoothly distributes the loads over the systems.

1.4.6 Clustering

In order to recover from a runtime server failure, the server's state needs to be replicated across multiple servers. This allows a client to be transparently migrated to an alternative server, if the server it is interacting with fails. Clustering is the distribution of an application over multiple servers to scale beyond the limits, both performance and reliability, of a single server. When the limits of the server software or the physical limits of the hardware have been reached, the load must be spread over multiple servers or machines to scale the system further. Clustering allows us to seamlessly distribute over multiple servers/machines and still maintain a single logical entity and a single virtual interface to respond to the client requests. The grouping of clusters creates highly scalable and reliable deployments while minimizing the number of servers needed to cope with large workloads.

1.5 Java Message Service

A large number of MOM implementations exist, including WebSphere MQ (formerly MQSeries) [15], TIBCO [16] SonicMQ [17], Herald [18], Hermes [19], SIENA [20], Gryphon [21], JEDI [22], REBECCA [23] and OpenJMS [24]. In order to simplify the development of systems utilizing MOMs, a standard was needed to provide a universal interface to MOM interactions. To date, a number of MOM standardization have emerged such as the CORBA Event Service [25], CORBA Notification Service [26] and most notably the Java Message Service (JMS).

The Java Message Service (JMS) provides a common way for Java programs to create, send, receive, and read an enterprise messaging system's messages [27]. The JMS provides a solid foundation for the construction of a messaging infrastructure that can be applied to a wide range of applications. The JMS specification defines a general purpose Application Programming Interface (API) to an enterprise messaging service and a set of semantics that describe the interface and general behavior of a messaging service. The goal of the JMS specification is to provide a universal way to interact with multiple heterogeneous messaging systems in a consistent manner. The learning curve associated

with many proprietary-messaging systems can be steep, thus the powerful yet simple API defined in the JMS specification can save a substantial amount of time for developers in a pure Java environment.

This API is designed to allow application programmers to write code to interact with a MOM. The specification also defines a Service Provider Interface (SPI). The role of the SPI is to allow MOM developers to hook up their proprietary MOM server to the API. This allows you to write code once using the API and plug-in the desired MOM provider, making client-messaging code portable between MOM providers that implement the JMS specification, reducing vendor lock-in and offering you a choice. It should be noted that JMS is an API specification and does not define the implementation of a messaging service. The semantics of message reliability, performance, scalability, and so on, are not fully defined. JMS does not define an "on the wire" format for messages. Effectively, two JMS compatible MOM implementations cannot talk to each other directly and will need to use a tool such as a connector/bridge queue to enable interoperability.

1.5.1 Programming using the JMS API

The format of a general message interface to a MOM would need to have the following minimum functionality detailed in Table 1.4. The JMS API provides this basic functionality through its programming model, illustrated in Figure 1.11, allowing it to be compatible with most MOM implementations. This section gives a brief overview of the programming model and presents an example of its usage. The code presented in the section is pseudocode to illustrate the main points of the JMS API, in order to conserve space; error/exception handling code has been omitted.

1.5.1.1 Connections and Sessions

When a client wants to interact with a JMS-compatible MOM platform, it must first make a connection to the message broker. Using this connection, the client may create one or more sessions; a JMS session is a single-threaded context used to send and receive messages to and from queues and topics. Each session can be configured with individual transactional and acknowledgment modes.

Table 1.4 General MOM API interface

Action	Description
SEND	Send a message to a specific queue
RECEIVE (BLOCKING)	Read a message from a queue. If the queue is empty, the call will block until it is nonempty
RECEIVE (NONBLOCKING POLL)	Read a message from the queue. If the queue is empty, do not block
LISTENER (NOTIFY)	Allows the message service to inform the client of the arrival of a message using a callback function on the client. The callback function is executed when a new message arrives in the queue

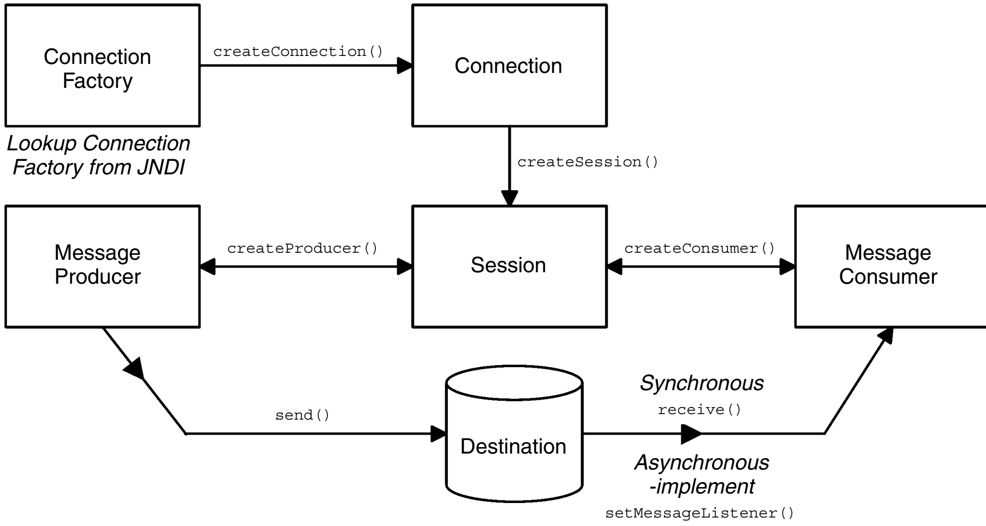

Figure 1.11 The JMS API programming model

```
try {
  // Create a connection
  javax.jms.QueueConnectionFactory queueConnectionFactory
            = (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");
  QueueConnection queueConnection
                           = queueConnectionFactory.createQueueConnection();

  // Create a Session
  javax.jms.QueueSession queueSession
                              = queueConnection.createQueueSession(false,
                              Session.AUTO_ACKNOWLEDGE);
  // Create Queue Sender and Receiver
  javax.jms.Queue myQueue = queueSession.createQueue("MyQueue");
  javax.jms.QueueSender queueSender = queueSession.createSender(myQueue);
  javax.jms.QueueReceiver queueReceiver
                                  = queueSession.createReceiver(myQueue);

  // Start the Connection
  queueConnection.start();

  // Send Message
  javax.jms.TextMessage message = queueSession.createTextMessage();
  message.setText(" Hello World ! ");
  queueSender.send(message);

  // Synchronous Receive
  javax.jms.Message msg = queueReceiver.receive();
  if (msg instanceof TextMessage) {
    javax.jms.TextMessage txtMsg = (TextMessage) msg;
    System.out.println("Reading message: " + txtMsg.getText());
```

```
} else {
  // Handle other message formats
}

// Asynchronous Receive
MessageListener msgListener = new MessageListener() {
 public void onMessage(javax.jms.Message msg) {
  if (msg instanceof TextMessage) {  // Only supports text messages
   javax.jms.TextMessage txtMsg = (TextMessage) msg;
   System.out.println("Reading message: " + txtMsg.getText());
  }
 }
};
queueReceiver.setMessageListener(msgListener);

} catch (javax.jms.JMSException jmse) {
  // Handle error
}
```

1.5.1.2 Message Producers and Consumers

In order for a client to send a message to or receive a message from a JMS provider, it must first create a message producer or message consumer from the JMS session. For the publish/subscribe model, a `javax.jms.TopicPublisher` is needed to send messages to a topic and a `javax.jms.TopicSubscriber` to receive. The above pseudocode is an example of the process of connecting to a JMS provider, establishing a session to a queue (point-to-point model) and using a `javax.jms.QueueSender` and `javax.jms.QueueReceiver` to send and receive messages. The steps involved in connecting to a topic are similar.

Receive Synchronously and Asynchronously

The JMS API supports both synchronous and asynchronous message delivery. To synchronously receive a message, the `receive()` method of the message consumer is used. The default behavior of this method is to block until a message has been received; however, this method may be passed a *time-out* value to limit the blocking period. To receive a message asynchronously, an application must register a `Message Listener` with the message consumer. Message listeners are registered with a message consumer object by using the `setMessageListener(javax.jms.MessageListener msgL)` method. A message listener must implement the `javax.jms.MessageListener` interface. Further detailed discussion and explanations of the JMS API are available in [29] and [27].

1.5.1.3 Setting Message Properties

Message properties are optional fields contained in a message. These user-defined fields can be used to contain information relevant to the application or to identity messages. Message properties are commonly used as the data filtered by consuming clients using message selectors.

1.5.1.4 Message Selectors

Message selectors are used to filter the messages received by a message consumer, and they assign the task of filtering messages to the JMS provider rather than to the application. The message consumer will only receive messages whose headers and properties match the selector. A message selector cannot select messages on the basis of the content of the message body. Message selectors consist of a string expression based on a subset of the SQL-92 conditional expression syntax.

"Property_Vehicle_Type = 'SUV' and Property_Mileage =< 60000"

1.5.1.5 Acknowledgments Modes

JMS supports the acknowledgment of the receipt of a message. Acknowledgment modes are controlled at the sessions level with the modes in Table 1.5 supported.

1.5.1.6 Delivery Modes

The JMS API supports two delivery modes for message. The default PERSISTENT delivery mode instructs the service to ensure that a message is not lost because of system failure. A message sent with this delivery mode is placed in a nonvolatile memory store. The second option available is the NON_PERSISTENT delivery mode; this mode does not require the service to store the message or guarantee that it will not be lost because of system failure. This is a more efficient delivery mode because it does not require the message to be saved to nonvolatile storage.

1.5.1.7 Priority

The priority setting of a message can be adjusted to indicate to the message service urgent messages that should be delivered first. There are ten levels of priority ranging from 0 (lowest priority) to 9 (highest priority).

Table 1.5 JMS acknowledgement modes

Acknowledgment modes	Purpose
AUTO_ACKNOWLEDGE	Automatically acknowledges receipt of a message. In asynchronous mode, the handler acknowledges a successful return. In synchronous mode, the client has successfully returned from a call to receive()
CLIENT_ACKNOWLEDGE	Allow a client to acknowledge the successful delivery of a message by calling its acknowledge() method
DUPS_OK_ACKNOWLEDGE	A lazy acknowledgment mechanism that is likely to result in the delivery of message duplicates. Only consumers that can tolerate duplicate messages should use this mode. This option can reduce overhead by minimizing the work to prevent duplicates

1.5.1.8 Time-to-Live

JMS messages contain a use-by or expiry time known as the Time-to-Live (TTL). By default, a message never expires; however, you may want to set an expiration time. When the message is published, the specified TTL is added to the current time to give the expiration time. Any messages not delivered before the specified expiration times are destroyed.

1.5.1.9 Message Types

•

The JMS API defines five message types, listed in Table 1.6•, that allow you to send and receive data in multiple formats. The JMS API provides methods for creating messages of each type and for filling in their contents.

1.5.1.10 Transactional Messaging

JMS clients can include message operations (sending and receiving) in a transaction. The JMS API session object provides commit and rollback methods that are used to control the transaction from a JMS client. A detailed discussion on transactions and transactional messaging is available in Chapter **??**.

1.5.1.11 Message Driven Enterprise Java Beans

The J2EE includes a Message Driven Bean (MDB) as a component that consumes messages from a JMS topic or queue, introduced in the Enterprise Java Beans (EJB) 2.0 specification they are designed to address the integration of JMS with EJBs. MDB is a stateless, server-side, transaction-aware component that allows J2EE applications to process JMS and other message such as HTTP, ebXML, SMTP, and so on, asynchronously. Traditionally a proxy was needed to allow EJBs to process an asynchronous method invocation. This approach used an external Java program that acted as the listener, and on receiving a message, invoked a session bean or entity bean method synchronously using RMI/JRMP or RMI/IIOP. With this approach, the message was received outside the application. MDB solves this problem by allowing the message-processing code access to the infrastructure services available from an EJB container such as transactions, fault-tolerance, security, instances pooling, and so on. The EJB 2.0 specification also provides concurrent processing for MDBs with pooling of bean instances. This allows for the simultaneous processing

Table 1.6 JMS message types

Message type	Message contains
`javax.jms.TextMessage`	A java.lang.String object
`javax.jms.MapMessage`	A set of name/value pairs, with names as strings and values as java primitive types. The entries can be accessed by name
`javax.jms.BytesMessage`	A stream of uninterrupted bytes
`javax.jms.StreamMessage`	A stream of Java primitive values, filled and read sequentially
`javax.jms.ObjectMessage`	A Serializable Java object

of messages received, allowing MDBs a much higher throughput with superior scalability than traditional JMS clients.

An MDB is a message listener that can reliably consume messages from a queue or a durable subscription associated with a single JMS destination (queue or topic). Similar to a message listener in a standalone JMS client, an MDB contains an `onMessage(javax.jms.Message msg)`. The EJB container invokes this method when it intercepts an incoming JMS message, allowing the bean to process the message. A detailed discussion on implementing MDBs is presented in [29].

1.6 Service-Oriented Architectures

The problems and obstacles encountered during system integration pose major challenges for an organizations IT department:

> "70% of the average IT department budget is devoted to data integration projects"–IDC
>
> "PowerPoint engineers make integration look easy with lovely cones and colorful boxes"–Sean McGrath, CTO, Propylon
>
> "A typical enterprise will devote 35%–40% of its programming budget to programs whose purpose is solely to transfer information between different databases and legacy systems."–Gartner Group

Increasing pressure to cut the cost of software development is driving the emergence of open nonproprietary architectures to utilize the benefits of reusable software components. MOM has been used to create highly open and flexible systems that allow the seamless integration of subsystems. MOM solves many of the transport issues with integration. However, major problems still exist with the representation of data, its format, and structure. To develop a truly open system, MOM requires the assistance of additional technologies such as XML and Web Services. Both of these technologies provide a vital component in building an open cohesive system. Each of these technologies will be examined to highlight the capabilities they provide in the construction of open system architectures.

1.6.1 XML

The eXtensible Mark-up Language (XML) provides a programming language and platform-independent format for representing data. When used to express the payload of a message, this format eliminates any networking, operating system, or platform binding that a binary proprietary protocol would use. XML provides a natural independent way of representing data. Once data is expressed in XML, it is trivial to change the format of the XML using techniques such as the eXtensible Stylesheet Language: Transformations

(XSLT). In order for XML to be used as a message exchange medium, standard formats need to be defined to structure the XML messages. There are a number of bodies working on creating these standards such as ebXML and the OASIS Universal Business Language (UBL). These standards define a document layout format to provide a standard communicative medium for applications. With UBL, you convert your internal message formats to the standard UBL format and export to the external environment. To import messages, you mirror this process. An extensive examination of this process and relevant standards is outside the scope of this chapter, for further information see [30, 31].

1.6.2 Web Services

Web Services are platform- and language-independent standards defining protocols for heterogeneous system integration. In their most basic format, web services are interfaces that allow programs to run over public or private networks using standard protocols such as Simple Object Access Protocol (SOAP). They allow links to be created between systems without the need for massive reengineering. They can interact with and/or invoke one another, fulfilling tasks and requests that in turn carry out specific parts of complex transactions or workflows. Web services can be seen in a number of ways, such as a business-to-business/enterprise application integration tool or as a natural evolution of basic RPC mechanism. The key benefit of a web services deployment is that they act as a façade to the underlying language or platform, a web service written in C and running on Microsoft's Internet Information Server can access a web service written in Java running on BEA's Weblogic server.

1.6.2.1 SOAP

The Simple Object Access Protocol (SOAP) provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML [32]. SOAP messages contain an envelope, message headers, and a message body. SOAP allows you to bind it to a transport mechanism such as SMTP, HTTP, and JMS. SOAP can be used to implement both the synchronous and asynchronous messaging models. SOAP has a number of uses; it can be used as a document exchange protocol, a heterogeneous interoperability standard, a wire protocol standard (something not defined in JMS), and an RPC mechanism. A detailed discussion on SOAP is available in [32]. For the purposes of this section, SOAP is seen as a document-exchange protocol between heterogeneous systems.

1.6.3 MOM

Message-Oriented Middleware provides an asynchronous, loosely coupled, flexible communication backbone. The benefits of utilizing a MOM in distributed systems have been examined in this chapter. When the benefits of a neutral, independent message format and the ease of web service integration are combined with MOM, highly flexible open systems may be constructed. Such an approach is more likely to be robust with respect to change over a systems lifecycle.

1.6.4 Developing Service-Oriented Architectures

A service is a set of input messages sent to a single object or a composition of objects, with the return of causally related output messages

Through the combination of these technologies, we are able to create Service-Oriented Architectures (SOA). The fundamental design concept behind these architectures is to reduce application processing to logic black boxes. Interaction with these black boxes is achieved with the use of a standard XML message format; by defining the required XML input and output formats, we are able to create black box services. The service can now be accessed by transmitting the message over an asynchronous messaging channel.

With traditional API-based integration, the need to create adaptors and data format converters for each proprietary API is not a scalable solution. As the number of APIs increases, the adaptors and data converters required will scale geometrically. The important aspect of SOAs is their message-centric structure. Where message formats differ, XML-based integration can convert the message to and from the format required using an XML transformation pipeline, as is shown in Figure 1.12.

In this approach, data transformation can be seen as just another assembly line problem, allowing services to be true black box components with the use of an XML-in and XML-out contract. Users of the services simply need to transform their data to the services contract XML format. Integration via SOA is significantly cheaper than integration via APIs; with transformations taking place outside of the applications, it is a very noninvasive method of integration. Service-Oriented Architecture with transforming XML is a new and fresh way of looking at Enterprise Application Integration (EAI) and distributed systems and a new way of looking at web services that are often touted as a replacement for traditional RPC. Viewed in this light, they are an evolution of the RPC mechanism but still suffer from many of its shortcomings.

With Service-Oriented Architectures, creating connections to trading partners and legacy systems should be as easy as connecting to an interdepartmental system. Once the initial infrastructure has been created for the architecture, the amount of effort to connect to further systems is minimal. This allows systems created with this framework to be highly dynamic, allowing new participants to easily join and leave the system.

Figure 1.13 illustrates an example SOA using the techniques advocated. In this deployment, the system has been created by interconnecting six subsystems and integrating

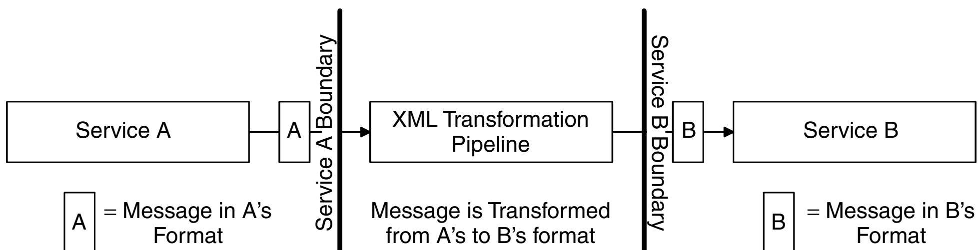

Figure 1.12 XML transformation pipeline

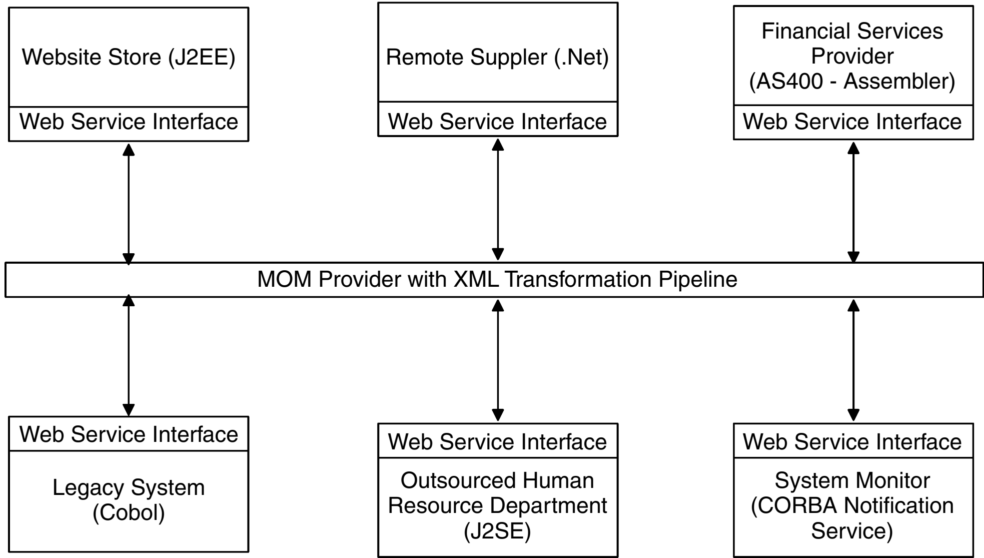

Figure 1.13 System deployed using web service, XML messages and MOM to create a SOA

them, each of the subsystems is built using a different technology for their primary implementation.

The challenges faced in this deployment are common challenges faced daily by system developers. When developing a new system it is rare for a development team not to have some form of legacy system to interact with. Legacy systems may contain irreplaceable business records, and losing this data could be catastrophic for an organization. It is also very common for legacy systems to contain invaluable business logic that is vital for an organization's day-to-day operation. It would be preferable to reuse this production code, potentially millions of lines, in our new system. Transforming a legacy system into an XML service provides an easy and flexible solution for legacy interaction. The same principle applies to all the other subsystems in the deployment, from the newly created J2EE Web Store, or the *Financial Services* provider running on an AS400 to the *Remote Suppliers* running the latest Microsoft .NET systems. Each of these proprietary solutions can be transformed into a service and join the SOA. Once a subsystem has been changed into a service, it can easily be added and removed from the architecture. SOAs facilitate the construction of highly dynamic systems, allowing functionality (services) such as payroll, accounting, sales, system monitors, and so on, to be easily added and removed at run time without interruptions to the overall system. The key to developing a first-rate SOA is to interconnect services with a MOM-based communication; MOM utilization will promote loose coupling, flexibility, reliability, scalability, and high-performance characteristics in the overall system architecture.

$$\frac{\text{XML} + \text{Web Services} + \text{MOM}}{\text{Service Oriented Architecture}} = \text{Open Systems}$$

1.7 Summary

Distribution middleware characterizes a high-level remote-programming mechanism designed to automate and extend the native operating system's network programming facilities. This form of middleware streamlines the development of distributed systems by simplifying the mechanics of the distribution process.

Traditionally, one of the predominate forms of distribution mechanisms used is Remote Procedure Calls (RPC). This mechanism, while powerful in small- to medium-scale systems, has a number of shortcomings when used in large-scale multiparticipant systems. An alternative mechanism to RPC has emerged to meet the challenges presented in the mass distribution of large-scale enterprise-level systems.

Message-Oriented Middleware or MOM is a revolutionary concept in distribution allowing for communications between disparate software entities to be encapsulated into messages. MOM solves a number of the inadequacies inherent in the RPC mechanism. MOM can simplify the process of building dynamic, highly flexible enterprise-class distributed systems.

MOM can be defined as any middleware infrastructure that provides messaging capabilities. They provide the backbone infrastructure to create cohesive distributed applications. MOM platforms are one of the cornerstone foundations that distributed enterprise systems are built upon. The process of building dynamic, highly flexible enterprise-class distributed systems can be simplified by utilizing a state-of-the-art enterprise-level MOM as the communications backbone.

The main benefits of MOM come from the asynchronous interaction model and the use of message queues. These queues allow each participating system to proceed at its own pace without interruption. MOM introduces transaction capability and a high Quality of Service (QoS). It also provides a number of communication messaging models to solve a variety of different messaging challenges.

MOM-based systems are proficient in coping with traffic bursts while offering a flexible and robust solution for disperse deployments. Remote systems do not need to be available for the calling program to send a message. Loose coupling exists between the consumers and producers, allowing flexible systems to grow and change on demand. MOM also provides an abstract interface for communications. When MOM is used in conjunction with XML messages and web services, we are able to create highly flexible service-oriented architectures. This form of architecture allows for the flexible integration of multiple systems.

Bibliography

[1] Tanenbaum, A. S. and Steen, M. V. (2002) *Distributed Systems: Principles and Paradigms*, 1st ed., Prentice Hall●.

[2] Banavar, G., Chandra, T., Strom, R. E., et al.● (1999) A Case for Message Oriented Middleware. *Proceedings of the 13th International Symposium on Distributed Computing*, Bratislava, Slovak Republic.

[3] Massive Scalability Focus Group (2003) *Deployment Strategies Focusing on Massive Scalability*●.

[4] Curry, E., Chambers, D., and Lyons, G. (2003) A JMS Message Transport Protocol for the JADE Platform. *Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, Halifax, Canada; IEEE Press.

[5] Hinze, A. and Bittner, S. (2002) Efficient Distribution-Based Event Filtering. *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria; IEEE Press.

[6] Carzaniga, A., Rosenblum, D.S., and Wolf, A. L. (2001) Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, **19**(3), 332–383.

[7] Pietzuch, P. R. and Bacon, J. M. (2002) *Hermes: A Distributed Event-Based Middleware Architecture.*

[8] Curry, E., Chambers, D., and Lyons, G. (2003) Reflective Channel Hierarchies. *Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany.

[9] Mühl, G. and Fiege, L. (2001) Supporting Covering and Merging in Content-Based Publish/Subscribe Systems: Beyond Name/Value Pairs. *IEEE Distributed Systems Online*, **2**(7●).

[10] Pietzuch, P. R., Shand, B., and Bacon, J. (2003) A Framework for Event Composition in Distributed Systems. *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany.

[11] Tai, S. and Rouvellou, I. (2000) Strategies for Integrating Messaging and Distributed Object Transactions. *Proceedings of the Middleware 2000*, New York, USA; Springer-Verlag.

[12] Tai, S., Totok, A., Mikalsen, T., et al. (2003) Message Queuing Patterns for Middleware-Mediated Transactions. *Proceedings of the SEM 2002*, Orlando, FL; Springer-Verlag.

[13] Chambers, D., Lyons, G., and Duggan, J. (2002) A Multimedia Enhanced Distributed Object Event Service. *IEEE Multimedia*, **9**(3), 56–71.

[14] Linthicum, D. (1999) *Enterprise Application Integration*, Addison-Wesley.

[15] Gilman, L. and Schreiber, R. (1996) *Distributed Computing with IBM MQSeries*, John Wiley, New York.

[16] Skeen, D. (1992) An Information Bus Architecture for Large-Scale, Decision-Support Environments. *Proceedings of the USENIX Winter Conference*●.

[17] Sonic Software. *Sonic MQ*, http://www.sonicmq.com●.

[18] Cabrera, L. F., Jones, M. B., and Theimer, M. (2001) Herald: Achieving a Global Event Notification Service. *Proceedings of the 8th Workshop on Hot Topics in OS*.

[19] Pietzuch, P. R. (2002) *Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems?*

[20] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2000) Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000)*, Portland, OR.

[21] Strom, R., Banavar, G., Chandra, T., et al. (1998) Gryphon: An Information Flow Based Approach to Message Brokering. *Proceedings of the International Symposium on Software Reliability Engineering*, Paderborn, Germany.

[22] Cugola, G., Nitto, E. D., and Fuggetta, A. (2001) The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, **27**(9), 827–850.

[23] Fiege, L. and Mühl, G. *Rebeca*, http://gkpc14.rbg.informatik.tu-darmstadt.de/rebeca/.

[24] ExoLab Group. *OpenJMS*, http://openjms.sourceforge.net/

[25] Object Management Group (2001) *Event Service Specification.*

[26] Object Management Group (2000) *Notification Service Specification.*

[27] Sun Microsystems (2001) *Java Message Service: Specification.*

[28] •Haase, K. and Sun Microsystems. *The Java Message Service (JMS) Tutorial*, http://java.sun.com/products/jms/tutorial/.

[29] Monson-Haefel, R. and Chappell, D. A. (2001) *Java Message Service*, O'Reilly & Associates.

[30] Bosak, J. and Crawford, M. (in press•). *Universal Business Language (UBL) Specification*.

[31] Lyons, T. and Molloy, O. (2003) Development of an e-Business Skillset Enhancement Tool (eSET) for B2B Integration Scenarios. *Proceedings of the IEEE Conference on Industrial Informatics (INDIN 2003)*, Banff, Canada.

[32] W3C (2001) *SOAP Version 1.2.*