Chapter 4
An Introduction to Middleware Architectures and Technologies

4.1 Introduction

I'm not really a great enthusiast for drawing strong analogies between the role of a software architect and that of a traditional building architect. There are similarities, but also lots of profound differences.[1] But let's ignore those differences for a second, in order to illustrate the role of middleware in software architecture.

When an architect designs a building, they create drawings, essentially a design that shows, from various angles, the structure and geometric properties of the building. This design is based on the building's requirements, such as the available space, function (office, church, shopping center, home), desired aesthetic and functional qualities and budget. These drawings are an abstract representation of the intended concrete (sic) artifact.

There's obviously an awful lot of design effort still required to turn the architectural drawings into something that people can actually start to build. There's detailed design of walls, floor layouts, staircases, electrical systems, water and piping to name just a few. And as each of these elements of a building is designed in detail, suitable materials and components for constructing each are selected.

These materials and components are the basic construction blocks for buildings. They've been created so that they can fulfill the same essential needs in many types of buildings, whether they are office towers, railway stations or humble family homes.

Although perhaps it's not the most glamorous analogy, I like to think of middleware as the equivalent of the plumbing or piping or wiring for software applications. The reasons are:

- Middleware provides proven ways to connect the various software components in an application so they can exchange information using relatively easy-to-use mechanisms. Middleware provides the pipes for shipping data between components, and can be used in a wide range of different application domains.

[1]The following paper discusses of issues: J. Baragry and K. Reed. *Why We Need a Different View of Software Architecture*. The Working IEEE/IFIP Conference on Software Architecture (WICSA), Amsterdam, The Netherlands, 2001.

I. Gorton, *Essential Software Architecture*,
DOI 10.1007/978-3-642-19176-3_4,

- Middleware can be used to wire together numerous components in useful, well-understood topologies. Connections can be one-to-one, one-to-many or many-to-many.
- From the application user's perspective, middleware is completely hidden. Users interact with the application, and don't care how information is exchanged internally. As long as it works, and works well, middleware is *invisible* infrastructure.
- The only time application users are ever aware of the role middleware plays is when it fails. This is of course very like real plumbing and wiring systems.

It's probably not wise to push the plumbing analogy any further. But hopefully it has served its purpose. Middleware provides ready-to-use infrastructure for connecting software components. It can be used in a whole variety of different application domains, as it has been designed to be general and configurable to meet the common needs of software applications.

4.2 Middleware Technology Classification

Middleware got its label because it was conceived as a layer of software "plumbing-like" infrastructure that sat between the application and the operating system, that is, the middle of application architectures. Of course in reality middleware is much more complex than plumbing or a simple layer insulating an application from the underlying operating system services.

Different application domains tend to regard different technologies as middleware. This book is about mainstream IT applications, and in that domain there's a fairly well-understood collection that is typically known as middleware. Figure 4.1 provides a classification of these technologies, and names some example products/technologies that represent each category. Brief explanations of the categories are below, and the remainder of this chapter and the next two go on to describe each in detail:

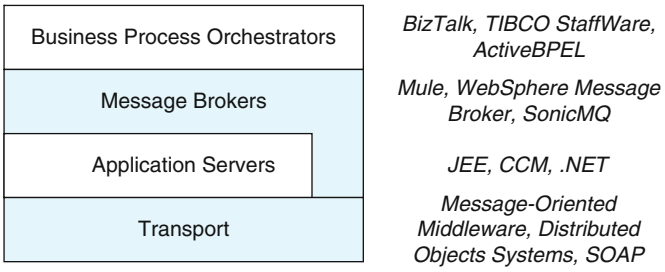

Fig. 4.1 Classifying middleware technologies

- The transport layer represents the basic pipes for sending requests and moving data between software components. These pipes provide simple facilities and mechanisms that make exchanging data straightforward in distributed application architectures.
- Application servers are typically built on top of the basic transport services. They provide additional capabilities such as transaction, security and directory services. They also support a programming model for building multithreaded server-based applications that exploit these additional services.
- Message brokers exploit either a basic transport service and/or application servers and add a specialized message processing engine. This engine provides features for fast message transformation and high-level programming features for defining how to exchange, manipulate and route messages between the various components of an application.
- Business process orchestrators (BPOs) augment message broker features to support workflow-style applications. In such applications, business processes may take many hours or days to complete due to the need for people to perform certain tasks. BPOs provide the tools to describe such business processes, execute them and manage the intermediate states while each step in the process is executed.

4.3 Distributed Objects

Distributed object technology is a venerable member of the middleware family. Best characterized by CORBA,[2] distributed object-based middleware has been in use since the earlier 1990s. As many readers will be familiar with CORBA and the like, only the basics are briefly covered in this section for completeness.

A simple scenario of a client sending a request to a server across an object request broker (ORB) is shown in Fig. 4.2. In CORBA, servant objects support interfaces that are specified using CORBA's IDL (interface description language). IDL interfaces define the methods that a server object supports, along with the parameter and return types. A trivial IDL example is:

```
module ServerExample {
    interface MyObject
    {
        string isAlive();
    };
};
```

This IDL interface defines a CORBA object that supports a single method, `isAlive`, which returns a string and takes no parameters. An IDL compiler is used to process interface definitions. The compiler generates an object skeleton in

[2]Common Object Request Broker Architecture.

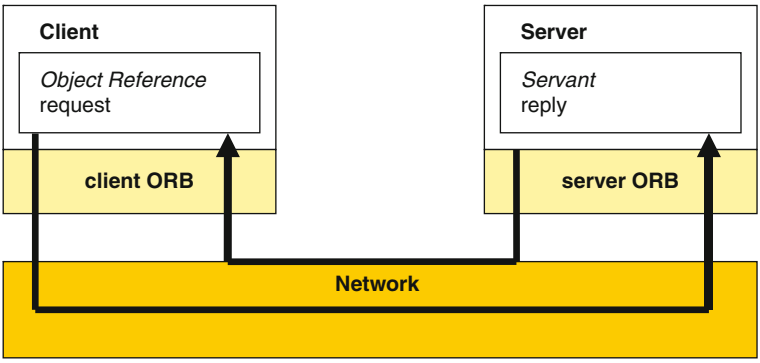

Fig. 4.2 Distributed objects using CORBA

a target programming languages (typically, but not necessarily, C++ or Java). The object skeleton provides the mechanisms to call the servant implementation's methods. The programmer must then write the code to implement each servant method in a native programming language:

```
class MyServant extends _MyObjectImplBase {
  public String isAlive()    {
    return "\nLooks like it…\n";
    }
}
```

The server process must create an instance of the servant and make it callable through the ORB:

```
ORB orb = ORB.init(args, null);
MyServant objRef = new MyServant();
orb.connect(objRef);
```

A client process can now initialize a client ORB and get a reference to the servant that resides within the server process. Servants typically store a reference to themselves in a directory. Clients query the directory using a simple logical name, and it returns a reference to a servant that includes its network location and process identity.

```
ORB orb = ORB.init(args, null);
// Lookup is a wrapper that actually access the CORBA Naming //
Service directory - details omitted for simplicity
MyServant servantRef = lookup("Myservant")
String reply = servantRef.isAlive();
```

The servant call looks like a synchronous call to a local object. However, the ORB mechanisms transmit, or marshal, the request and associated parameters across the network to the servant. The method code executes, and the result is marshaled back to the waiting client.

This is a very simplistic description of distributed object technology. There's much more detail that must be addressed to build real systems, issues like exceptions, locating servants and multithreading to name just a few. From an architect's perspective though, the following are some essential design concerns that must be addressed in applications:

- Requests to servants are remote calls, and hence relatively expensive (slow) as they traverse the ORB and network. This has a performance impact. It's always wise to design interfaces so that remote calls can be minimized, and performance is enhanced.
- Like any distributed application, servers may intermittently or permanently be unavailable due to network or process or machine failure. Applications need strategies to cope with failure and mechanisms to restart failed servers.
- If a servant holds state concerning an interaction with a client (e.g., a customer object stores the name/address), and the servant fails, the state is lost. Mechanisms for state recovery must consequently be designed.

4.4 Message-Oriented Middleware

Message-oriented middleware (MOM) is one of the key technologies for building large-scale enterprise systems. It is the glue that binds together otherwise independent and autonomous applications and turns them into a single, integrated system. These applications can be built using diverse technologies and run on different platforms. Users are not required to rewrite their existing applications or make substantial (and risky) changes just to have them play a part in an enterprise-wide application. This is achieved by placing a queue between senders and receivers, providing a level of indirection during communications.

How MOM can be used within an organization is illustrated in Fig. 4.3. The MOM creates a *software bus* for integrating home grown applications with legacy

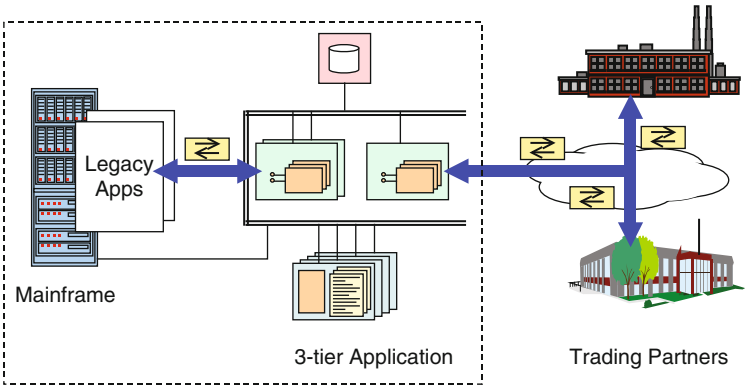

Fig. 4.3 Integration through messaging

applications, and connecting local applications with the business systems provided by business partners.

4.4.1 MOM Basics

MOM is an inherently loosely coupled, asynchronous technology. This means the sender and receiver of a message are not tightly coupled, unlike synchronous middleware technologies such as CORBA. Synchronous middleware technologies have many strengths, but can lead to fragile designs if all of the components and network links always have to be working at the same time for the whole system to successfully operate.

A messaging infrastructure decouples senders and receivers using an intermediate message queue. The sender can send a message to a receiver and know that it will be eventually delivered, even if the network link is down or the receiver is not available. The sender just tells the MOM technology to deliver the message and then continues on with its work. Senders are unaware of which application or process eventually processes the request. Figure 4.4 depicts this basic send–receive mechanism.

MOM is often implemented as a server that can handle messages from multiple concurrent clients.[3] In order to decouple senders and receivers, the MOM provides message queues that senders place messages on and receivers remove messages from. A MOM server can create and manage multiple messages queues, and can handle multiple messages being sent from queues simultaneously using threads organized in a thread pool. One or more processes can send messages to a message queue, and each queue can have one or many receivers. Each queue has a name which senders and receivers specify when they perform send and receive operations. This architecture is illustrated in Fig. 4.5.

A MOM server has a number of basic responsibilities. First, it must accept a message from the sending application, and send an acknowledgement that the message has been received. Next, it must place the message at the end of the queue that was specified by the sender. A sender may send many messages to a queue

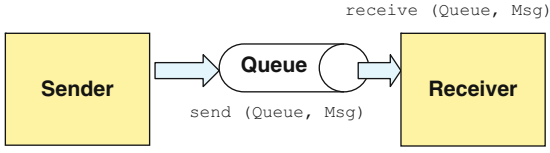

Fig. 4.4 MOM basics

[3]MOM can also be simply implemented in a point-to-point fashion without a centralized message queue server. In this style of implementation, 'send' and 'receive' queues are maintained on the communicating systems themselves.

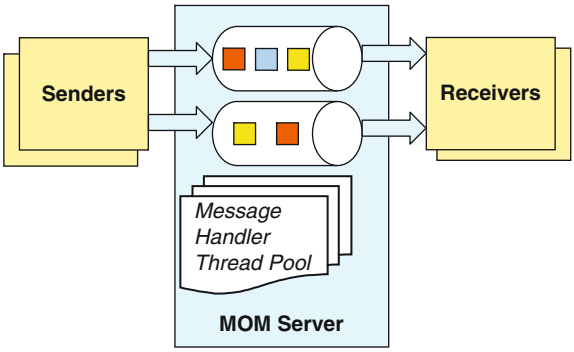

Fig. 4.5 Anatomy of a MOM server

before any receivers remove them. Hence the MOM must be prepared to hold messages in a queue for an extended period of time.

Messages are delivered to receivers in a First-In-First-Out (FIFO) manner, namely the order they arrive at the queue. When a receiver requests a message, the message at the head of the queue is delivered to the receiver, and upon successful receipt, the message is deleted from the queue.

The asynchronous, decoupled nature of messaging technology makes it an extremely useful tool for solving many common application design problems. These include scenarios in which:

- The sender doesn't need a reply to a message. It just wants to send the message to another application and continue on with its own work. This is known as *send-and-forget* messaging.
- The sender doesn't need an immediate reply to a request message. The receiver may take perhaps several minutes to process a request and the sender can be doing useful work in the meantime rather than just waiting.
- The receiver, or the network connection between the sender and receiver, may not operate continuously. The sender relies on the MOM to deliver the message when a connection is next established. The MOM layer must be capable of storing messages for later delivery, and possibly recovering unsent messages after system failures.

4.4.2 Exploiting MOM Advanced Features

The basic features of MOM technology are rarely sufficient in enterprise applications. Mission critical systems need much stronger guarantees of message delivery and performance than can be provided by a basic MOM server. Commercial-off-the-shelf (COTS) MOM products therefore supply additional advanced features to increase the reliability, usability and scalability of MOM servers. These features are explained in the following sections.

4.4.2.1 Message Delivery

MOM technologies are about delivering messages between applications. In many enterprise applications, this delivery must be done reliably, giving the sender guarantees that the message will eventually be processed. For example, an application processing a credit card transaction may place the transaction details on a queue for later processing, to add the transaction total to the customer's account. If this message is lost due the MOM server crashing – such things do happen – then the customer may be happy, but the store where the purchase was made and the credit card company will lose money. Such scenarios obviously cannot tolerate message loss, and must ensure reliable delivery of messages.

Reliable message delivery however comes at the expense of performance. MOM servers normally offer a range of quality of service (QoS) options that let an architect balance performance against the possibility of losing messages. Three levels of delivery guarantee (or QoS) are typically available, with higher reliability levels always coming at the cost of reduced performance. These QoS options are:

- *Best effort*: The MOM server will do its best to deliver the message. Undelivered messages are only kept in memory on the server and can be lost if a system fails before a message is delivered. Network outages or unavailable receiving applications may also cause messages to time out and be discarded.
- *Persistent*: The MOM layer guarantees to deliver messages despite system and network failures. Undelivered messages are logged to disk as well as being kept in memory and so can be recovered and subsequently delivered after a system failure. This is depicted in Fig. 4.6. Messages are kept in a disk log for the queue until they have been delivered to a receiver.
- *Transactional*: Messages can be bunched into "all or nothing" units for delivery. Also, message delivery can be coordinated with an external resource manager such as a database. More on transactional delivery is explained in the following sections.

Various studies have been undertaken to explore the performance differences between these three QoS levels. All of these by their very nature are specific to a particular benchmark application, test environment and MOM product. Drawing some very general conclusions, you can expect to see between 30 and 80%

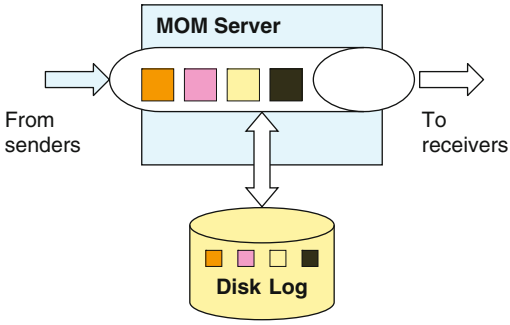

Fig. 4.6 Guaranteed message delivery in message oriented middleware

performance reduction when moving from best-effort to persistent messaging, depending on message size and disk speed. Transactional will be slower than persistent, but often not by a great deal, as this depends mostly on how many transaction participants are involved. See the further reading section at the end of this chapter for some pointers to these studies.

4.4.2.2 Transactions

Transactional messaging typically builds upon persistent messages. It tightly integrates messaging operations with application code, not allowing transactional messages to be sent until the sending application commits their enclosing transaction. Basic MOM transactional functionality allows applications to construct batches of messages that are sent as a single atomic unit when the application commits.

Receivers must also create a transaction scope and ask to receive complete batches of messages. If the transaction is committed by the receivers, these transactional messages will be received together in the order they were sent, and then removed from the queue. If the receiver aborts the transaction, any messages already read will be put back on the queue, ready for the next attempt to handle the same transaction. In addition, consecutive transactions sent from the same system to the same queue will arrive in the order they were committed, and each message will be delivered to the application exactly once for each committed transaction.

Transactional messaging also allows message sends and receives to be coordinated with other transactional operations, such as database updates. For example, an application can start a transaction, send a message, update a database and then commit the transaction. The MOM layer will not make the message available on the queue until the transaction commits, ensuring either that the message is sent and the database is updated, or that both operations are rolled back and appear never to have happened.

A pseudocode example of integrating messaging and database updates is shown in Fig. 4.7. The sender application code uses transaction demarcation statements (the exact form varies between MOM systems) to specify the scope of the transaction. All statements between the *begin* and *commit* transaction statements are considered to be part of the transaction. Note we have two, independent transactions occurring in this example. The sender and receiver transactions are separate and commit (or abort) individually.

4.4.2.3 Clustering

MOM servers are the primary message exchange mechanism in many enterprise applications. If a MOM server becomes unavailable due to server or machine failure, then applications can't communicate. Not surprisingly then, industrial strength MOM technologies make it possible to cluster MOM servers, running instances of the server on multiple machines (see Fig. 4.8).

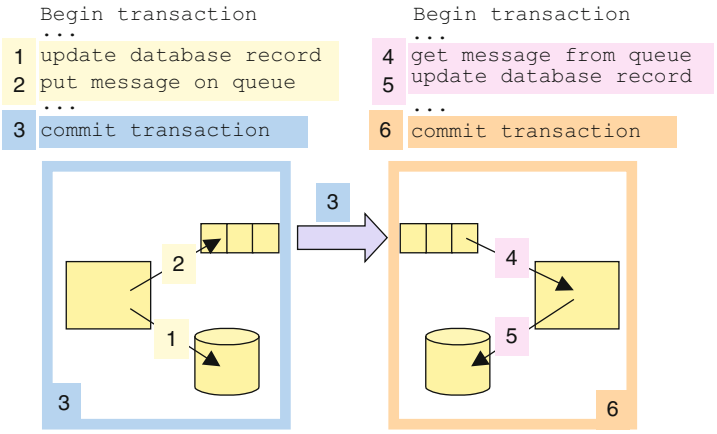

Fig. 4.7 Transactional messaging

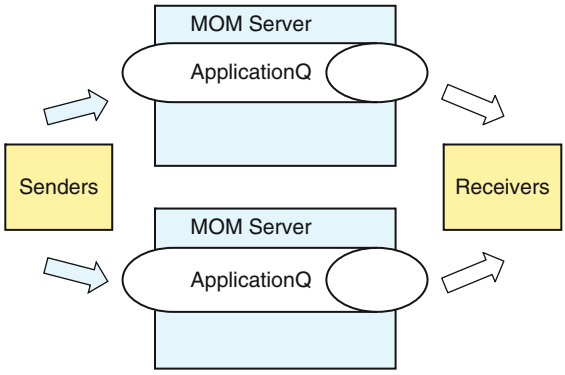

Fig. 4.8 Clustering MOM servers for reliability and scalability

Exactly how clustering works is product dependent. However, the scheme in Fig. 4.8 is typical. Multiple instances of MOM servers are configured in a logical cluster. Each server supports the same set of queues, and the distribution of these queues across servers is transparent to the MOM clients. MOM clients behave exactly the same as if there was one physical server and queue instance.

When a client sends a message, one of the queue instances is selected and the message stored on the queue. Likewise, when a receiver requests a message, one of the queue instances is selected and a message removed. The MOM server clustering implementation is responsible for directing client requests to individual queue instances. This may be done statically, when a client opens a connection to the server, or dynamically, for every request.[4]

[4]An application that needs to receive messages in the order they are sent is not suitable for operating in this a clustering mode.

A cluster has two benefits. First, if one MOM server fails, the other queue instances are still available for clients to use. Applications can consequently keep communicating. Second, the request load from the clients can be spread across the individual servers. Each server only sees a fraction (ideally 1/[number of servers] in the cluster) of the overall traffic. This helps distribute the messaging load across multiple machines, and can provide much higher application performance.

4.4.2.4 Two-Way Messaging

Although MOM technology is inherently asynchronous and decouples senders and receivers, it can also be used for synchronous communications and building more tightly coupled systems. In this case, the sender simply uses the MOM layer to send a request message to a receiver on a request queue. The message contains the name of the queue to which a reply message should be sent. The sender then waits until the receiver sends back a reply message on a reply queue, as shown in Fig. 4.9.

This synchronous style of messaging using MOM is frequently used in enterprise systems, replacing conventional synchronous technology such as CORBA. There are a number of pragmatic reasons why architects might choose to use messaging technology in this way, including:

- Messaging technology can be used with existing applications at low cost and with minimal risk. *Adapters* are available, or can be easily written to interface between commonly used messaging technologies and applications. Applications do not have to be rewritten or ported before they can be integrated into a larger system.
- Messaging technologies tend to be available on a very wide range of platforms, making it easier to integrate legacy applications or business systems being run by business partners.
- Organizations may already have purchased, and gained experience in using, a messaging technology and they may not need the additional features of an application server technology.

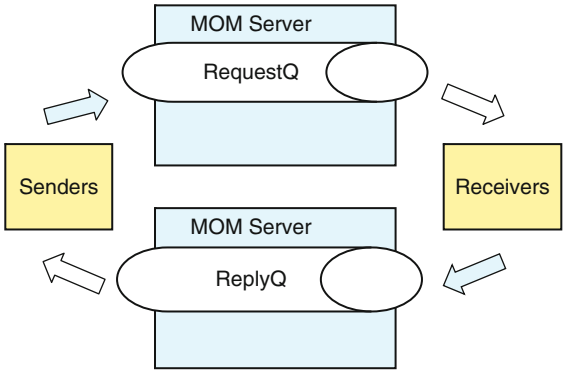

Fig. 4.9 Request–Reply messaging

4.4.3 Publish–Subscribe

MOM is a proven and effective approach for building loosely coupled enterprise systems. But, like everything, it has its limitations. The major one is that MOM is inherently a one-to-one technology. One sender sends a single message to a single queue, and one receiver retrieves that message for the queue. Not all problems are so easily solved by a 1–1 messaging style. This is where publish–subscribe architectures enter the picture.

Publish–subscribe messaging extends the basic MOM mechanisms to support *1 to many*, *many to many*, and *many to 1* style communications. Publishers send a single copy of a message addressed to a named *topic*, or *subject*. Topics are a logical name for the publish–subscribe equivalent of a queue in basic MOM technology. Subscribers listen for messages that are sent to topics that interest them. The publish–subscribe server then distributes each message sent on a topic to every subscriber who is listening on that topic. This basic scheme is depicted in Fig. 4.10.

In terms of loose coupling, publish–subscribe has some attractive properties. Senders and receivers are decoupled, each respectively unaware of which applications will receive a message, and who actually sent the message. Each topic may also have more than one publisher, and the publishers may appear and disappear dynamically. This gives considerable flexibility over static configuration regimes. Likewise, subscribers can dynamically subscribe and unsubscribe to a topic. Hence the subscriber set for a topic can change at any time, and this is transparent to the application code.

In publish–subscribe technologies, the messaging layer has the responsibility for managing topics, and knowing which subscribers are listening to which topics. It also has the responsibility for delivering every message sent to all active current subscribers. Topics can be persistent or nonpersistent, with the same effects on reliable message delivery as in basic point-to-point MOM (explained in the previous section). Messages can also be published with an optional "time-to-live" setting. This tells the publish–subscribe server to attempt to deliver a message to all active subscribers for the time-to-live period, and after that delete the message from the queue.

The underlying protocol a MOM technology uses for message delivery can profoundly affect performance. By default, most use straightforward point-to-point

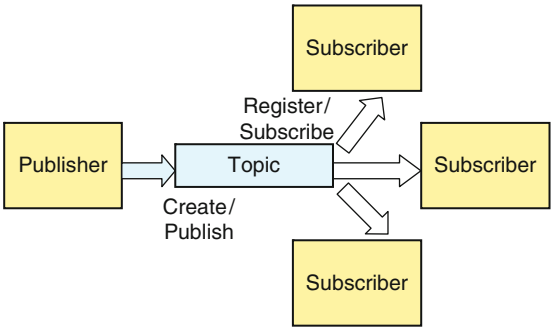

Fig. 4.10 Publish–Subscribe messaging

TCP/IP sockets. Implementations of publish–subscribe built on point-to-point messaging technology duplicate each message send operation from the server for every subscriber. In contrast, some MOM technologies support multicast or broadcast protocols, which send each message only once on the wire, and the network layer handles delivery to multiple destinations.

In Fig. 4.11, the multicast architecture used in TIBCO's Rendezvous publish–subscribe technology is illustrated. Each node in the publish–subscribe network runs a daemon process known as *rvd*. When a new topic is created, it is assigned a multicast IP address.

When a publisher sends a message, its local *rvd* daemon intercepts the message and multicasts a single copy of the message on the network to the address associated with the topic. The listening daemons on the network receive the message, and each checks if it has any local subscribers to the message's topic on its node. If so, it delivers the message to the subscriber(s), otherwise it ignores the message. If a message has subscribers on a remote network,[5] an *rvrd* daemon intercepts the message and sends a copy to each remote network using standard IP protocols. Each receiving *rvrd* daemon then multicasts the message to all subscribers on its local network.

Not surprisingly, solutions based on multicast tend to provide much better raw performance and scalability for best effort messaging. Not too long ago, I was

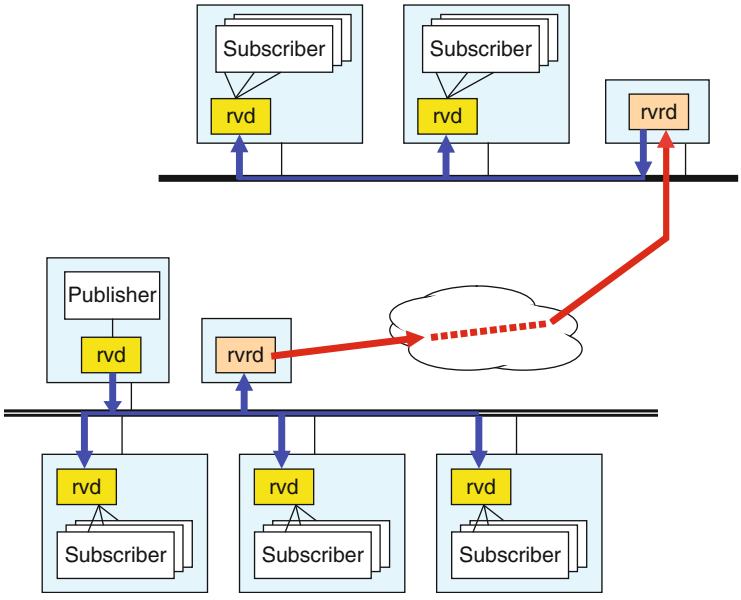

Fig. 4.11 Multicast delivery for publish–subscribe

[5]And the wide area network doesn't support multicast.

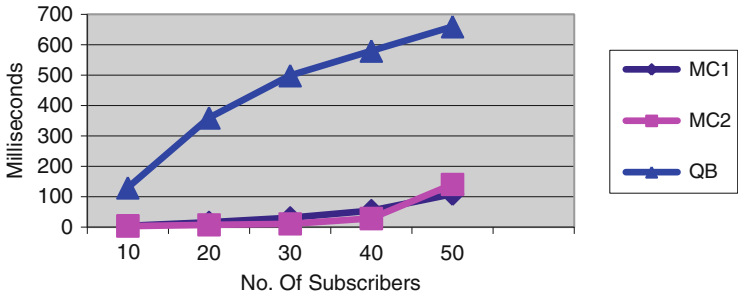

Fig. 4.12 Publish–subscribe best effort messaging performance: Comparing 2 multicast technologies (MC1, MC2) with a queue-based (QB) publish–subscribe technology

involved in a project to quantify the expected performance difference between multicast and point-to-point solutions. We investigated this by writing and running some benchmarks to compare the relative performance of three publish–subscribe technologies, and Fig. 4.12 shows the benchmark results.

It shows the average time for delivery from a single publisher to between 10 and 50 concurrent subscribers when the publisher outputs a burst of messages as fast as possible. The results clearly show that multicast publish–subscribe is ideally suited to applications with demands for low message latencies and hence very high throughput.

4.4.3.1 Understanding Topics

Topics are the publish–subscribe equivalent of queues. Topic names are simply strings, and are specified administratively or programmatically when the topic is created. Each topic has a logical name which is specified by all applications which wish to publish or subscribe using the topic.

Some publish–subscribe technologies support hierarchical topic naming. The details of exactly how the mechanisms explained below work are product dependent, but the concepts are generic and work similarly across implementations. Let's use the slightly facetious example shown in Fig. 4.13 of a topic naming tree.

Each box represents a topic name that can be used to publish messages. The unique name for each topic is a fully qualified string, with a "/" used as separator between levels in the tree. For example, the following are all valid topic names:

```
Sydney
Sydney/DevGroup
Sydney/DevGroup/Information
Sydney/DevGroup/Information/work
Sydney/DevGroup/Information/gossip
Sydney/SupportGroup
Sydney/SupportGroup/Information
Sydney/SupportGroup/Information/work
Sydney/SupportGroup/Information/gossip
```

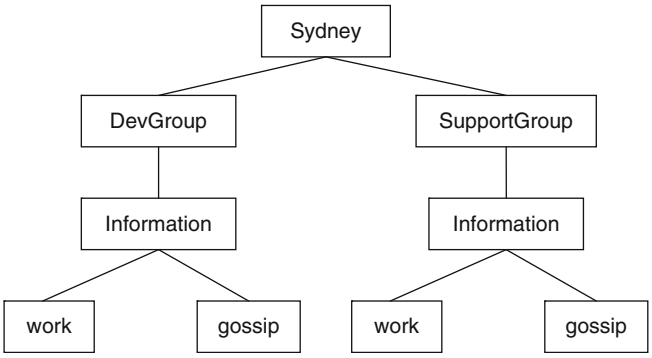

Fig. 4.13 An example of hierarchical topic naming

Hierarchical topic names become really useful when combined with topic wildcards. In our example, an "*" is used as a wildcard that matches zero or more characters in a topic name. Subscribers can use wildcards to receive messages from more than one topic when they subscribe. For example:

```
Sydney/*/Information
```

This matches both `Sydney/DevGroup/Information` and `Sydney/SupportGroup/Information`. Similarly, a subscriber that specifies the following topic:

```
Sydney/DevGroup/*/*
```

This will receive messages published on all three topics within the `Sydney/DevGroup` tree branch. As subscribing to whole branches of a topic tree is very useful, some products support a shorthand for the above, using another wildcard character such as "**", i.e.,:

```
Sydney/DevGroup/**
```

The "**" wildcards also matches all topics that are in `Sydney/DevGroup` branch. Such a wildcard is powerful as it is naturally extensible. If new topics are added within this branch of the topic hierarchy, subscribers do not have to change the topic name in their subscription request in order to receive messages on the new topics.

Carefully crafted topic name hierarchies combined with wildcarding make it possible to create some very flexible messaging infrastructures. Consider how applications might want to subscribe to multiple topics, and organize your design to support these.

4.5 Application Servers

There are many definitions for application servers, but all pretty much agree on the core elements. Namely, an application server is a component-based server technology that resides in the middle-tier of an N-tier architecture, and provides distributed communications, security, transactions and persistence. In this section, we'll use the Java Enterprise Edition[6] as our example.

Application servers are widely used to build internet-facing applications. Figure 4.14 shows a block diagram of the classic N-tier architecture adopted by many web sites.

An explanation of each tier is below:

- *Client Tier*: In a web application, the client tier typically comprises an Internet browser that submits HTTP requests and downloads HTML pages from a web server. This is commodity technology, not an element of the application server.
- *Web Tier*: The web tier runs a web server to handle client requests. When a request arrives, the web server invokes web server-hosted components such as servlets, Java Server Pages (JSPs) or Active Server Pages (ASPs) depending on the flavor of web server being used. The incoming request identifies the exact web component to call. This component processes the request parameters, and uses these to call the business logic tier to get the required information to satisfy the request. The web component then formats the results for return to the user as HTML via the web server.
- *Business Component Tier*: The business components comprise the core business logic for the application. The business components are realized by for example Enterprise JavaBeans (EJB) in JEE, .NET components or CORBA objects. The business components receive requests from the web tier, and satisfy requests usually by accessing one or more databases, returning the results to the web tier.

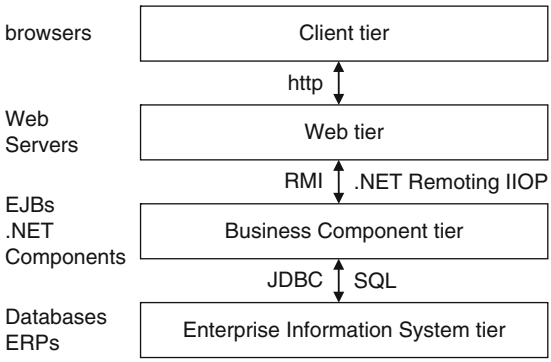

Fig. 4.14 N-Tier architecture for web applications

[6]The platform was known as *Java 2 Platform, Enterprise Edition* or *J2EE* until the name was changed to *Java EE* in version 5.

A run-time environment known as a *container* accommodates the components. The container supplies a number of services to the components it hosts. These varying depending on the container type (e.g., EJB, .NET, CORBA), but include transaction and component lifecycle management, state management; security, multithreading and resource pooling. The components specify, in files external to their code, the type of behavior they require from the container at run-time, and then rely on the container to provide the services. This frees the application programmer from cluttering the business logic with code to handle system and environmental issues.

- *Enterprise Information Systems Tier*: This typically consists of one or more databases and back-end applications like mainframes and other legacy systems. The business components must query and interact with these data stores to process requests.

The core of an application server is the business component container and the support it provides for implementing business logic using a software component model. As the details vary between application server technologies, let's just look at the widely used EJB model supported by JEE. This is a representative example of application server technology.

4.5.1 Enterprise JavaBeans

The EJB architecture defines a standard programming model for constructing server-side Java applications. A JEE-compliant application server provides an EJB container to manage the execution of application components. In practical terms, the container provides an operating system process (in fact a Java virtual machine) that hosts EJB components. Figure 4.15 shows the relationship between an application server, a container and the services provided. When an EJB client invokes a server component, the container allocates a thread and invokes an instance

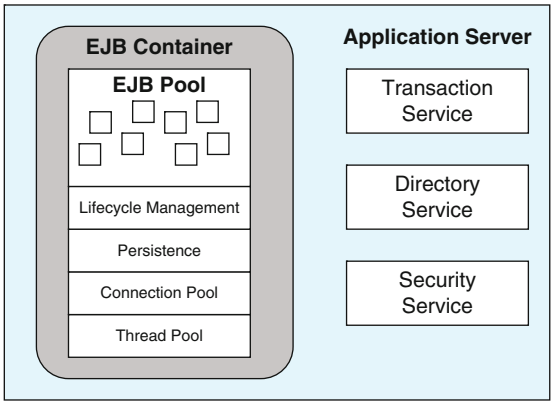

Fig. 4.15 JEE application server, EJB container and associated services

of the EJB component. The container manages all resources on behalf of the component and all interactions between the component and the external systems.

4.5.2 EJB Component Model

The EJB component model defines the basic architecture of an EJB component. It specifies the structure of the component interfaces and the mechanisms by which it interacts with its container and with other components.

The latest EJB specification (part of the Java™ Platform, Enterprise Edition (Java EE) version 5) defines two main types of EJB components, namely *session beans* and *message-driven beans*. Earlier JEE specifications also defined *entity beans*, but these have been phased out and replaced by the simpler and more powerful *Java Persistence API*[7]. This provides an object/relational mapping facility for Java applications that need access to relational databases from the server tier (a very common requirement, and one beyond the scope of this book).

Session beans are typically used for executing business logic and to provide services for clients to call. Session beans correspond to the controller in a model-view-controller[8] architecture because they encapsulate the business logic of a three-tier architecture. Session beans define an application-specific interface that clients can use to make requests. Clients send a request to a session bean and block until the session bean sends a response.

Somewhat differently to session beans, message-driven beans are components that process messages asynchronously. A message bean essentially acts as a listener for messages that are sent from a Java Message Service (JMS) client. Upon receipt of a message, the bean executes its business logic and then waits for the next message to arrive. No reply is sent to the message sender.

Further, there are two types of session beans, known as *stateless* session beans and *stateful* session beans. The difference between these is depicted in Fig. 4.16.

A stateless session bean is defined as not being *conversational* with respect to its calling process. This means that it does not keep any state information on behalf of any client that calls it. A client will get a reference to a stateless session bean in a container, and can use this reference to make many calls on an instance of the bean. However, between each successive bean invocation, a client is not guaranteed to bind to any particular stateless session bean instance. The EJB container delegates client calls to stateless session beans on *an as needed* basis, so the client can never be certain which bean instance they will actually talk to. This makes it meaningless to store client related state information in a stateless session bean. From the container's perspective, all instances of a stateless session bean are viewed as equal and can be assigned to any incoming request.

[7] http://java.sun.com/javaee/reference/faq/persistence.jsp

[8] See http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

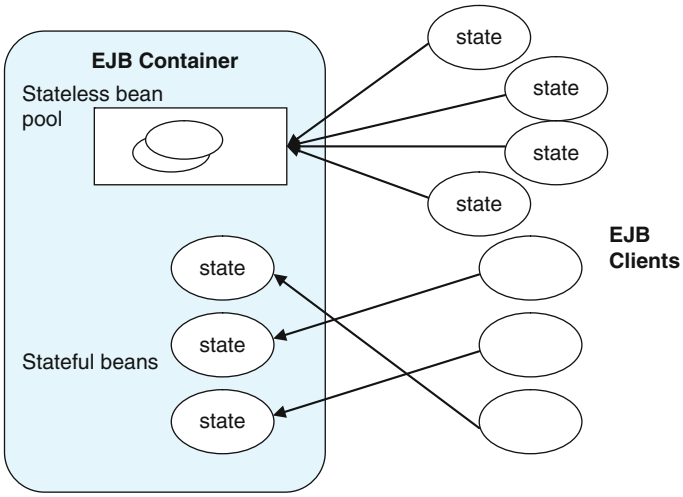

Fig. 4.16 Stateless versus stateful session beans

On the other hand, a stateful session bean is said to be conversational with respect to its calling process; therefore it can maintain state information about a conversation with a client. Once a client gets a reference to a stateful session bean, all subsequent calls to the bean using this reference are guaranteed to go to the same bean instance. The container creates a new, dedicated stateful session bean for each client that creates a bean instance. Clients may store any state information they wish in the bean, and can be assured it will still be there next time they access the bean.

EJB containers assume responsibility for managing the lifecycle of stateful session beans. The container will write out a bean's state to disk if it has not been used for a while, and will automatically restore the state when the client makes a subsequent call on the bean. This is known as *passivation* and *activation* of the stateful bean. Containers can also be configured to destroy a stateful session bean and its associated resources if a bean is not used for a specified period of time.

In many respects, message driven beans are handled by the EJB container in a similar manner to stateless session beans. They hold no client-specific conversational data, and hence instances can be allocated to handle messages sent from any client. Message beans don't receive requests directly from clients however. Rather they are configured to listen to a JMS queue, and when clients send messages to the queue, they are delivered to an instance of a message bean to process.

4.5.3 Stateless Session Bean Programming Example

To create an EJB component in EJB version 3.0, the developer must provide session bean class and a *remote* business interface. The *remote* interface contains the business

methods offered by the bean. These are of course application specific. Below is a (cut down) remote interface example for a stateless session bean. Note this is a standard Java interface that is simply decorated with *@Remote* annotation:

```
import javax.ejb.Remote;
@Remote
public interface Broker {
    public int newAccount(String name, String address,
        int credit)
        throws EJBException, SQLException;

    public void buyStock(int accno, int stock_id, int amount)
        throws EJBException, SQLException;

    public void updateAccount(int accno, int credit)
        throws EJBException, SQLException;
}
```

The class definition is again standard Java, and is simply annotated with *@Stateless*. The *@Stateless* annotation states that this class is a stateless session bean, and the business interface is used to invoke it.

```
import javax.ejb.Stateless;
@Stateless
public class BrokerBean implements Broker {
       // methods defined here …(not shown)
}
```

Accessing an EJB client in EJB 3.0 is very simple indeed, requiring the use of the *@EJB* annotation, namely:

```
@EJB BrokerBean broker;
Broker.updateAccount ( 99, 10000);
```

EJB clients may be standalone Java applications, servlets, applets, or even other EJBs. Clients interact with the server bean entirely through the methods defined in the bean's *remote* interface.

The story for stateful session beans is pretty similar, using the *@Stateful* annotation. Stateful session beans should also provide a bean specific initialization method to set up the bean's state, and a method annotated with *@Remove*, which is called by clients to indicate they have finished with this bean instance, and the container should remove it after the method completes.

4.5.4 Message-Driven Bean Programming Example

Message-driven beans are pretty simple to develop too. In the most common case of a message driven bean receiving messages from a JMS server, the bean implements the `javax.jms.MessageListener` interface. In addition, using the

`@MessageDriven` annotation, the developer specifies the name[9] of the destination from which the bean will consume messages.

```
import javax.jms.MessageListener;
@MessageDriven(mappedName="jms/BrokerQ")
public class BrokerMessageBean implements MessageListener {
        public void onMessage(Message msg) {
                TextMessage stockMessage =
                        (TextMessage)  msg;
                // process message
        }
}
```

4.5.5 Responsibilities of the EJB Container

It should be pretty obvious at this stage that the EJB container is a fairly complex piece of software. It's therefore worth covering exactly what the role of the container is in running an EJB application. In general, a container provides EJB components with a number of services. These are:

- It provides bean lifecycle management and bean instance pooling, including creation, activation, passivation, and bean destruction.
- It intercepts client calls on the remote interface of beans to enforce transaction and security (see below) constraints. It also provides notification callbacks at the start and end of each transaction involving a bean instance.
- It enforces session bean behavior, and acts as a listener for message-driven beans.

In order to intercept client calls, the tools associated with a container must generate additional classes for an EJB at deployment time. These tools use Java's introspection mechanism to dynamically generate classes to implement the *remote* interfaces of each bean. These classes enable the container to intercept all client calls on a bean, and enforce the policies specified in the bean's deployment descriptor.

The container also provides a number of other key run-time features for EJBs. These typically include:

- *Threading*: EJB's should not explicitly create and manipulate Java threads. They must rely on the container to allocate threads to active beans in order to provide a concurrent, high performance execution environment. This makes EJBs simpler to write, as the application programmer does not have to implement a threading scheme to handle concurrent client requests.
- *Caching*: The container can maintain caches of the entity bean instances it manages. Typically the size of the caches can be specified in deployment descriptors.

[9]Specifically, the annotation contains a `mappedName` element that specifies the JNDI name of the JMS queue where messages are received from.

- *Connection Pooling*: The container can manage a pool of database connections to enable efficient access to external resource managers by reusing connections once transactions are complete.

Finally, there's also some key features and many details of EJB that haven't been covered here. Probably the most important of these, alluded to above, are:

- *Transactions*: A transaction is a group of operations that must be performed as a unit, or not at all. Databases provides transaction management, but when a middle tier such as an EJB container makes distributed updates across multiple databases, things can get tricky. EJB containers contain a transaction manager (based on the Java Transaction API specification), that can be used to coordinate transactions at the EJB level. Session and message driven beans can be annotated with transaction attributes and hence control the commit or rollback of distributed database transactions. This is a very powerful feature of EJB.
- *Security*: Security services are provided by the EJB container and can be used to authenticate users and authorize access to application functions. In typical EJB style, security can be specified using annotations in the EJB class definition, or be implemented programmatically. Alternatively, EJB security can be specified externally to the application in an XML deployment descriptor, and this information is used by the container to override annotation-specified security.

4.5.6 Some Thoughts

This section has given a brief overview of JEE and EJB technology. The EJB component model is widely used and has proven a powerful way of constructing server-side applications. And although the interactions between the different parts of the code are at first a little daunting, with some exposure and experience with the model, it becomes relatively straightforward to construct EJB applications.

Still, while the code construction is not difficult, a number of complexities remain. These are:

- The EJB model makes it possible to combine components in an application using many different architectural patterns. This gives the architect a range of design options for an application. Which option is *best* is often open to debate, along with what does *best* mean in a given application? These are not always simple questions, and requires the consideration of complex design trade-offs.
- The way beans interact with the container is complex, and can have a significant effect of the performance of an application. In the same vein, all EJB server containers are not equal. Product selection and product specific configuration is an important aspect of the application development lifecycle.

For references discussing both these issues, see the further reading section at the end of this chapter.

4.6 Summary

It's taken the best part of 20 years to build, but now IT architects have a powerful toolkit of basic synchronous and asynchronous middleware technologies to leverage in designing and implementing their applications. These technologies have evolved for two main reasons:

1. They help make building complex, distributed, concurrent applications simpler.
2. They institutionalize proven design practices by supporting them in off-the-shelf middleware technologies.

With all this infrastructure technology available, the skill of the architect lies in how they select, mix and match architectures and technologies in a way that meets their application's requirements and constraints. This requires not only advanced design skills, but also deep knowledge of the technologies involved, understanding what they can be reliably called on to do, and equally importantly, what they cannot sensibly do. Many applications fail or are delivered late because perfectly good quality and well built middleware technology is used in a way in which it was never intended to be used. This is not the technology's fault – it's the designers'. Hence middleware knowledge, and more importantly experience with the technologies in demanding applications, is simply a prerequisite for becoming a skilled architect in the information technology world.

To make life more complex, it's rare that just a single architecture and technology solution makes sense for any given application. For example, simple messaging or an EJB component-based design might make sense for a particular problem. And these logical design alternatives typically have multiple implementation options in terms of candidate middleware products for building the solution.

In such situations, the architect has to analyze the various trade-offs between different solutions and technologies, and choose an alternative (or perhaps nominate a set of competing alternatives) that meets the application requirements. To be honest, I'm always a little suspicious of architects who, in such circumstances, always come up with the same architectural and technology answer (unless they work for a technology vendor – in that case, it's their job).

The cause of this "I have a hammer, everything is a nail" style behavior is often a fervent belief that a particular design, and more often a favored technology, can solve any problems that arise. As it's the end of the chapter, I won't get on my soap box. But I'll simply say that open-minded, experienced and technologically agnostic architects are more likely to consider a wider range of design alternatives. They're also likely to propose solutions most appropriate to the quirks and constraints of the problem at hand, rather than enthusiastically promoting a particular solution that demonstrates the eternal "goodness" of their favorite piece of technology over its "evil" competitors.

4.7 Further Reading

There's an enormous volume of potential reading on the subject matter covered in this chapter. The references that follow should give you a good starting point to delve more deeply.

4.7.1 CORBA

The best place to start for all CORBA related information is the Object Management Group's web site, namely:

http://www.omg.org

Navigate from here, and you'll find information on everything to do with CORBA, including specifications, tutorials and many books. For specific recommendations, in my experience, anything written by Doug Schmidt, Steve Vinosky or Michi Henning is always informative and revealing.

Talking of Michi Henning, another very interesting technology represented by the approach taken in Internet Communications Engine (Ice) from ZeroC (http://zeroc.com/). Ice is open source, and there's a list of interesting articles at:

http://zeroc.com/articles/index.html

Particularly interesting are "A New Approach to Object-Oriented Middleware" (IEEE Internet Computing, Jan 2004) and The Rise and Fall of CORBA (ACM Queue, Jun 2006)

4.7.2 Message-Oriented Middleware

The best place to look for MOM information is probably the product vendor's documentation and white papers. Use your favorite search engine to look for information on IBM WebSphere MQ, Microsoft Message Queue (MSMQ), Sonic MQ, and many more. If you'd like to peruse the Java Messaging Service specification, it can be downloaded from:

http://java.sun.com/products/jms/docs.html

If you're interested in a very readable and recent analysis of some publish–subscribe technology performance, including a JMS, the following is well worth downloading:

Piyush Maheshwari and Michael Pang, *Benchmarking Message-Oriented Middleware: TIB/RV versus SonicMQ*, Concurrency and Computation: Practice and Experience, volume 17, pages 1507–1526, 2005

4.7.3 Application Servers

Again, the Internet is probably the best source of general information on applications servers. Leading product include WebLogic (BEA), WebSphere (IBM), .NET application server (Microsoft), and for a high quality open source implementation, JBoss. There's a good tutorial for JEE v5.0 at:

http://java.sun.com/javaee/5/docs/tutorial/doc/index.html

There's also lots of good design knowledge about EJB applications in:

F. Marinescu. EJB Design Patterns: Advanced Patterns, Processes, and Idioms. Wiley, 2002

D. Alur, D. Malks, J. Crupi. Core JEE Patterns: Best Practices and Design Strategies. Second Edition, Prentice Hall, 2003

Two excellent books on transactions in Java, and in general, are:

Mark Little, Jon Maron, Greg Pavlik, Java Transaction Processing: Design and Implementation, Prentice-Hall, 2004

Philip A. Bernstein, Eric Newcomer, Principles of Transaction Processing, Second Edition (The Morgan Kaufmann Series in Data Management Systems), Morgan Kaufman, 2009

The following discusses how to compare middleware and application server features:

I. Gorton, A. Liu, P. Brebner. *Rigorous Evaluation of COTS Middleware Technology*. IEEE Computer, vol. 36, no. 3, pages 50–55, March 2003