4 A Guide 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. 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:

14 The following paper discusses of issues: J. Barzagry 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
4.2 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. Fig. 9 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 then goes on to describe each in detail:

- 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 multi-threaded 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.

Fig. 9. Classifying middleware technologies

4.3 Distributed Objects

Distributed object technology is a venerable member of the middleware family. Best characterized by CORBA\textsuperscript{15}, distributed object-based middleware has been in use since the earlier 1990's. As many readers will be fa-
The server process must create an instance of the servant and make it callable through the ORB:

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

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.

```java
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 servers and multi-threading 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.

![Fig. 11. Integration through messaging](image)

How message-oriented middleware can be used within an organization is illustrated in Fig. 11. The MOM creates a software bus for integrating home grown applications with legacy applications, and connecting local applications with the business systems provided by business partners.

4.4.1 Message-Oriented Middleware Basics

Message-oriented middleware 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. Fig. 12 depicts this basic send-receive mechanism.

![Fig. 12. MOM basics](image)

MOM is often implemented as a server that can handle messages from multiple concurrent clients. 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. 13.

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 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.

---

16 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.
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 to 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 so that can be recovered and subsequently delivered after a system failure. This is depicted in Fig. 14. 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%-80% 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.

![Diagram]

**Fig. 14. Guaranteed message delivery in message oriented middleware**

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 com-
plete 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, con-
secutive transactions sent from the same system to the same queue will ar-
rive in the order they were committed, and each message will be delivered
to the application exactly once for each committed transaction.

![Diagram]

**Fig. 15. Transactional messaging**

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 op-
erations are rolled back and appear never to have happened.

```
Begin transaction
1 update database record
2 put message on queue
3 commit transaction
4 get message from queue
5 update database record
6 commit transaction
```

A pseudo-code example of integrating messaging and database updates
is shown in Fig. 15. The sender application code uses transaction demarca-
tion 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 en-
terprise 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. 16).

Exactly how clustering works is product dependent. However, the
scheme in Fig. 16 is typical. Multiple instances of MOM servers are con-
figured 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.

![Fig. 16. Clustering MOM servers for reliability and scalability](image)

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.\(^\text{17}\)

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.

\(^{17}\) An application that needs to receive messages in the order they are sent is not suitable for operating in this a clustering mode.

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. 17.

![Fig. 17. Request-Reply messaging](image)

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.

4.4.3 Publish-Subscribe

Message oriented middleware 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 Diagram]

Fig. 18. Publish-Subscribe messaging

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. 18.

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 non-persistent, 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.

![Multicast Delivery Diagram]

Fig. 19. Multicast delivery for publish-subscribe
The underlying protocol a MOM technology uses for message delivery can profoundly affect performance. By default, most use straightforward point-to-point 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. 19, 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 addresses 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, 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 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. 20 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.

---

18 And the wide area network doesn’t support multicast.
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 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.

Application servers are widely used to build internet-facing applications. Fig. 22 shows a block diagram of the classic N-tier architecture adopted by many web sites.

---

**Fig. 22. N-Tier architecture for web applications**

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 J2EE, .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. 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 J2EE. This is a representative example of application server technology.

### 4.5.1 Enterprise JavaBeans

The Enterprise JavaBeans (EJB) architecture defines a standard programming model for constructing server-side Java applications. A J2EE-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. Fig. 23 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 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 EJB version 1.1 specifications defines two main types of EJB components, namely session beans and entity beans.

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 architecture because they encapsulate the business logic of a three-tier architecture.

![Fig. 23. J2EE application server, EJB container and associated services](image)

Entity beans are typically used for representing business data objects. The data members in an entity bean map directly to some data items stored in an associated database. Entity beans are usually accessed by a session bean that provides the business level services to the client.
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. 24.

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.

Fig. 24. 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 persistence 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.

There are also two types of entity beans, called Container Managed Persistence (CMP) entity beans and Bean Managed Persistence (BMP) entity beans. Persistence in this context refers to the way in which the data (usually a row in a relational database table) associated with the entity bean is read and written.

In the CMP entity bean case, the data that the bean represents is mapped automatically to the associated persistent data store (e.g. database) by the container. The container is responsible for loading the data into the bean instance, and writing changes back to the persistent data store at appropriate times, such as the start and end of a transaction. CMP relies on container-provided services and requires no application code as the container automatically generates the data access code. CMP is hence easy to implement, and supported for relational databases accessed using SQL.

In the case of a BMP entity bean, the bean code itself is responsible for accessing the persistent data it represents. This is typically done using handcrafted JDBC calls, or calls to a proprietary database or application API. Bean-managed persistence gives the bean developer the flexibility to perform persistence operations that are too complicated for the container to generate, or to use a data source that is not supported by the container, for example a custom or legacy database or an FTP site. While BMP requires more effort on the programmer's behalf to implement, it can sometimes provide opportunities to optimize data access, and in such cases may provide better performance than CMP.

4.5.3 EJB Programming

An EJB depends on the EJB container for everything it needs. If an EJB needs to access a JDBC connection or another bean, it uses the container services to achieve this.

To create an EJB component, the developer must provide two interfaces that define a bean's business and lifecycle management methods, plus the
actual bean implementation class. The two interfaces are called the remote and home interfaces, and have different purposes.

The home interface contains the lifecycle methods of the EJB. These provide clients with services to create, destroy, and find bean instances. In contrast, the remote interface contains the business methods offered by the bean. These are of course application specific. In order to use the methods in the bean's remote interface, clients must use the bean's home interface to obtain a reference to the remote interface.

A simple home interface is shown below. The home interface must inherit from EJBHome, and in this example, the interface contains a method to create an EJB of type Broker.

```
public interface BrokerHome extends EJBHome {
    // This method creates the EJB Object.
    Broker create() throws RemoteException, CreateException;
}
```

The (cut down) remote interface for this EJB looks like this:

```
public interface Broker extends EJBObject {
    public int newAccount(String name, String address, 
    int credit) throws RemoteException, SQLException;

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

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

An EJB client uses a bean's public interfaces to create and call an instance of the EJB. One instantiated by the client, the EJB implementation class, normally known as the bean class, becomes an accessible distributed Java object. Some example, rather simplified, client code is shown below:

```
Broker broker = null;
// find the home interface
Object _h = ctx.lookup("EntityStock.BrokerHome");
BrokerHome home = (BrokerHome) javax.rmi.PortableRemoteObject.narrow(_h,
BrokerHome.class);
// Use the home interface to create the Broker EJB Object
broker = home.create();
// execute requests at the broker EJB
broker.updateAccount(accno, 200000);
broker.buyStock(accno, stockID, 5000);
// we're finished
broker.remove();
```

EJB clients may be standalone Java applications, servlets, applets, or even other EJBs. All clients use the server bean's home interface to obtain a reference to an instance of the server bean. This reference is associated with the class type of the server bean's remote interface. Therefore the client interacts with the server bean entirely through the methods defined in the bean's remote interface.

In this example, the Broker bean is a stateless session bean that handles client requests. Internally, it actually uses the services of a number of entity beans to perform the data access logic. An example of one of the Broker methods, the updateAccount method, is included below:

```
public void updateAccount(int accno, int credit) 
throws RemoteException {
    try {
        Account account = accountHome.findByPrimaryKey(
        (new AccountPK(accno))).
        account.update(credit);
    }
    catch (Exception e) {
        throw new RemoteException(e.toString());
    }
}
```

This method uses an entity bean called Account. The entity bean encapsulates all the detailed manipulation of the application's data, in this case, exactly how an account record is updated. The code in the updateAccount method uses an entity bean finder method called findByPrimaryKey, which is provided by the Account bean in its home interface. The finder method takes the primary key for the account, and accesses the underlying database. If an account record is found in the database with this primary key, the EJB container creates an Account entity bean. The entity bean methods, in this example the update method, can then be used to access the data in the account record.

The home and remote interface for Account look like this:
public interface AccountHome extends EJBHome

public Account create(String name, String address, int credit) throws CreateException, RemoteException;

/**
 * Finds an Account by its primary Key (Account ID)
 */
public Account findByPrimaryKey(AccountPK key)
throws FinderException, RemoteException;

public interface Account extends EJBOBJECT

public void deposit(int amount) throws RemoteException;
public void withdraw(int amount)
throws AccountException, RemoteException;
// Getter/setter methods on Entity Bean fields
public String getSubName() throws RemoteException;
public void setSubName(String name) throws RemoteException;

The bean class for the entity bean contains the implementation of the remote methods. The code for the update method is included below. Note it is very simple, in fact a single line of Java.

public class AccountBean implements EntityBean {
    // Container-managed state fields
    public int sub_account;
    public String sub_name;
    public String sub_address;
    public int sub_credit;
    // lots missing...
    public void update(int amount) {
        sub_credit = amount;
    }
}

This simplicity is due to the fact that the entity bean is using "container managed persistence". The EJB container knows (we'll see how it knows soon) that there is a correspondence between the data members in the Account bean and the fields in an account table in the database the application is using.

Using this information, the container tools can generate the SQL statements needed to implement the finder method, and the queries needed to automatically read/write the data from/to the entity bean at the beginning/end of a transaction.

In this example, at the end of the Broker session bean's updateAccount method, the data items in the Account entity bean will be written back to the database, making the changes to the sub_credit persistent field. All this is done transparently, without explicit control from the programmer.

4.5.4 Deployment Descriptors

One of the major attractions of the EJB component model is the way in which it achieves a separation of concerns between business logic and infrastructure code. This separation of concerns refers to the fact that EJB's are mostly concerned with executing pure business logic. The EJB container becomes responsible for handling environmental and infrastructure issues like transactions, bean lifecycle management and security. This makes the code in the bean components simpler, as they aren't littered with code to handle all these additional complexities.

Beans inform the container of the services they require through deployment descriptors. A deployment descriptor is an XML document associated with an EJB. When a bean is deployed into a container, the container reads the deployment descriptor to find out how transactions, persistence (for entity beans), and access control should be handled. Hence deployment descriptors provide a declarative mechanism describing how these issues are handled.

The beauty of this mechanism is that the same EJB component can be deployed with different deployment descriptors suited to different application environments. If security is an issue, the component can specify its access control needs. If security is not an issue, no access control is specified in the deployment descriptor. In both cases the code in the EJB is identical. From a software engineering perspective, this is really nice to have.

The deployment descriptor example below is specified in an XML Document Type Definition (DTD). The deployment descriptor describes the type of bean (session or entity) and the classes used for the remote, home, and bean classes. It also specifies the transactional attributes of every method in the bean, which security roles can access each method (access control), and whether persistence in the entity beans is handled automatically by the container or is performed explicitly by the bean code.

The deployment descriptor for the Broker bean used in the example above is shown below. In addition to the attributes described above, the deployment descriptor specifies that this bean is stateless session bean, and that a container managed transaction is required to execute each method in the bean (these attributes are in boldface for ease of reading). As an example, we could simply change the <session-type> field in the XML to read Stateful, and the container would manage the bean very differently indeed.
As another example, let's look at the deployment descriptor for the Account entity bean. It looks like this:

```xml
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>EntityStock.BrokerHome</ejb-name>
      <home>db.entitystock.BrokerHome</home>
      <remote>db.entitystock.Broker</remote>
      <ejb-class>db.entitystock.BrokerBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>EntityStock.BrokerHome</ejb-name>
          <method-intf>Remote</method-intf>
          <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
      </container-transaction>
      </assembly-descriptor>
  </enterprise-beans>
</ejb-jar>
```

As well as the deployment attributes we've already seen, the above tells the container the following (bolded in the XML):

- it must manage persistence for beans of this type
- where to find the JDBC data source for the database
- what the primary key and data items are that must be mapped between the database and the entity bean

### 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 constraints. It also provides notification callbacks at the start and end of each transaction involving a bean instance.
- It manages the persistence of selected fields of CMP entity 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 home and 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.

- **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.

### 4.5.6 Some Thoughts

This section has given a brief overview of J2EE 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. Also, EJB version 3.0 19 is attempting to simplify a lot of the housekeeping code that is required, so when this version is widely available, building EJB applications should be even easier.

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 Message Brokers

Basic messaging using MOM and publish-subscribe technologies suffices for many applications. It's a simple, effective and proven approach that can deliver high levels of performance and reliability.

MOM deployments start to get a little more complex though when message formats are not totally agreed amongst the various applications that communicate using the MOM. This problem occurs commonly in the domain of enterprise integration, where the basic problem is building business applications from large, complex legacy business systems that were never designed to work together and exchange information.

Enterprise integration is a whole field of study in itself (see Further Reading). From the perspective of this book however, enterprise integration has spawned an interesting and widely used class of middleware technologies, known as message brokers.

Let’s introduce message brokers by way of a motivating example. Assume an organization has four different legacy business systems that each hold information about customers. Each of these four stores some common data about customers, as well as some unique data fields that others do not maintain. In addition, each of the applications has a different format for a customer record, and the individual field names are different across each (e.g. one uses ADDRESS, another LOCATION, as a field name for customer address data). To update customer data, a proprietary API is available for each legacy system.

Whilst this is conceptually pretty simple, it’s a problem that many organizations have. So, let’s assume keeping the data consistent in each of these four applications is a problem for our hypothetical organization. Hence they decide to implement a web site that allows customers to update their account information to be distributed to all four systems.

20 Duplicate data holdings like this are very common in enterprises. For example, my bank still manages to send my credit card statement and credit card rewards points statement to different addresses.
their own details online. When this occurs, the data entered into the web page is passed to a web component in the web server (e.g., a servlet or ASP.NET page). The role of this component is to pass the updated data to each of the four legacy applications, so they can update their own customer data correctly.

The organization uses MOM to communicate between applications. Consequently, the web component formats a message with the new customer data, and uses the MOM to send the message to each legacy system\textsuperscript{1}. The message format, labeled In-format in Fig. 25, is an agreed format that the web component and all the legacy applications understand.

![Diagram of MOM communication](image)

**Fig. 25.** Using MOM to communicate a customer data update to 4 legacy systems.

Each legacy system has a queue interface component that can read messages from the queue, and using the data in the message, create a call to the customer data update API that the legacy system supports. In this example, the interface component would read the message from the queue, extract the specific data fields from the message that it needs to call its legacy system’s API, and finally issue the API call. As shown in Fig. 26, the interface component is basically performing a transformation from the In-format to a format suitable for its associated legacy system.

So, for each legacy application, there is a dedicated component that executes the logic to transform the incoming message into a correctly formatted legacy system API call. The transformation is implemented in the program code of the component.

![Diagram of message transformation](image)

**Fig. 26.** Message transformation from common to a legacy-specific format.

This solution has some interesting implications:

- If the common In-format message format changes, then the web component and every legacy system component that executes the transformation must be modified and tested.
- If any legacy system API changes, then only the transformation for that system must be modified and tested.
- Modifying any of the transformations most likely requires coordinating with the development team who are responsible for the upkeep of the legacy system(s). These development teams are the ones who know the intimate details of how to access the legacy system API.

Hence, there is a tight coupling between all the components in this architecture. This is caused by the need for them to agree on the message format that is communicated. In addition, in large organizations (or even harder, across organizational boundaries), communicating and coordinating changes to the common message format can be slow and painful. It's the sort of thing you'd like to avoid if possible.

The obvious alternative solution is to move the responsibility for the message format transformation to the web component. This would guarantee that messages are sent to each legacy system interface component in the format they need to simply call the legacy API. The transformation complexity is now all in one place, the web component, and the legacy system interface component becomes simple. It basically reads message from the queue and calls the associated API using the data in the message. Changes to the In-format message do not cause changes in legacy interface...
components, as only the web component needs modifying and testing. Changes to any legacy API though require the specific legacy system development team to request a new message format from the web component development team.

Key:
Message = [ ]

![Diagram](image)

---

A message broker solution is attractive because it completely decouples the web component and the legacy interface components. The web component simply assembles and emits a message, and the broker transforms the message into the necessary format for each legacy system. It then sends an output message to the legacy system interface components in the precise format they desire.

A further attraction is the simplification of all the components in the system, as they now do not have to be concerned with message format transformation. The message transformation logic is localized within the message broker, and becomes the responsibility of the middleware development group to maintain. Consequently, if changes are needed in the web or legacy system message formats, the development team responsible only need liaise with the middleware development group, whose job it is to correctly update the transformations.

It's not a massive job to implement the broker pattern in conjunction with a standard MOM platform\(^{23}\). Such a solution would still have the disadvantage of defining the transformation logic in the program code. For simple transformations, this is no big deal, but many such applications involve complex transformations with fiddly string formatting and concatenations, formulas to calculate composite values, and so on. Nothing too difficult to write, but if there were a better solution that made creating complex transformations simple, I doubt many people would complain.

Message broker technologies begin to excel at this stage, because they provide specialized tools for:

- Graphically describing complex message transformations between input formats and output formats. Transformations can be simple in terms of moving an input field value to an output field, or they can be defined using scripting languages (typically product specific) that can perform various formatting, data conversions and mathematical transforms.
- High performance message transformation engines that can handle multiple simultaneous transformation requests.
- Describing and executing message flows, in which an incoming message can be routed to different transformations and outputs depending on the values in the incoming message.

An example of a message mapping tool is shown in Fig. 28. This is Microsoft's BizTalk Mapper, and is typical of the class of mapping technologies. In BizTalk, the mapper can generate the transformations necessary to move data between two XML schemas, with the lines depicting the mapping between source and destination schemas. Scripts (not shown in the

---

\(^{22}\) See Buschmann reference in Further Reading, Chapter 1

\(^{23}\) The solution is left as an exercise to the reader!
Managed centrally, allowing a team responsible for application integration to coordinate and test changes.

Executed by a high performance, multi-threaded transformation engine.

Fig. 29. Message routing and processing

Importantly, message brokers operate on a per message level. They receive an input message, transform it according to the message routing rules and logic, and output the resulting message or messages to their destinations. Brokers work best when these transformations are short-lived and execute quickly, in for example a few milliseconds. This is because they are typically optimized for performance, and hence try to avoid overheads that would slow down transformations. Consequently, if a broker or its host machine crashes, it relies on the fact that failed transformation can simply be executed again from the beginning, meaning expensive state and transaction management is not needed.\(^{24}\)

\(^{24}\) Many message brokers do optionally support transactional messaging, and even allow the broker to modify databases during transformation execution. These transactions are coordinated by an ACID transaction manager, such as the one supplied with the underlying MOM technology.
For a large class of application integration scenarios, high-speed transformation is all that's required. However, many business integration problems require the definition of a series of requests flowing between different applications. Each request may involve several message transformations, reads and updates to external database systems, and complex logic to control the flow of messages between applications and potentially even humans for off-line decision-making. For such problems, message brokers are insufficient and well, you guessed it, even more technology is required. This is described in the next section.

Before moving on though, it should be emphasized that message brokers, like everything in software architecture and technologies, do have their downsides. First, they are proprietary technologies, and this leads to vendor lock-in. It’s the price you pay for all those sophisticated development and deployment tools. Second, in high volume messaging applications, the broker can become a bottleneck. Most message broker products support broker clustering to increase performance, scalability and reliability, but this comes at the cost of complexity and dollars.

### 4.7 Business Process Orchestration

Business processes in modern enterprises can be complex in terms of the number of enterprise applications that must be accessed and updated to complete the business service. As an example, Fig. 30 is a simple depiction of a sales order business process, in which the following sequence of events occurs.

A customer places an order through a call center. Customer data is stored in a customer relationship management package (Siebel). Once the order is placed, the customer’s credit is validated using an external credit service, and the accounts payable database is updated to record the order and send an invoice to the customer.

Placing an order causes a message to be sent to Shipping, who update their inventory system and ship the order to the customer. When the customer receives the order, they pay for the goods and the payment is recorded in the accounts system. All financial data is periodically extracted from the accounts system and stored in an Oracle database for management reporting and archiving.

Implementing such business processes has two major challenges. First, from the time an order is placed to when the payment is received might take several days or weeks, or even longer. Somewhere then, the current state of the business process for a given order, representing exactly what stage it is up to, must be stored, potentially for a long time. Losing this state, and hence the status of the order, is not a desirable option.

Second, exceptions in the order process can cause the state of the order to fail and rollback. For example, an order is taken for some stock item. Let’s assume this stock is not available in the warehouse and when it is reordered, the supplier tells the warehouse that the old stock is now obsolete, and that a newer, more expensive model will replace it. The customer is informed of this, and they decide to cancel the order. Canceling requires the order data to be removed from the warehouse, accounts payable and Siebel systems. This is potentially a complex task to reliably and correctly perform.

This style of rollback behavior can be defined by the process designer using a facility known as a compensating transaction. Compensating transactions allow the process designer to explicitly define the logic required to undo a failed transaction that partially completed.

In long-running business processes such as sales order processing, standard ACID transactions, which lock all resources until the transaction completes, are not feasible. This is because they lock data in the business systems for potentially minutes, hours or even weeks in order to achieve transaction isolation. Locked data cannot be accessed by concurrent transactions, and hence lock contention will cause these to wait (or more likely fail through timing out) until the locks are released. Such a situation is not likely to produce high-performance and scalable business process implementations for long-running business processes.
Transactional behavior for long running processes is therefore usually handled by grouping a number of process activities into a long-running transaction scope. Long-running transactions comprise of multiple process activities that do not place locks on the data items they touch in the various business systems. Updates are made and committed locally at each business system. However, if any activity in the transaction scope fails, the designer must specify a compensating function. The role of the compensator is to undo the effects of the transaction that have already committed. Essentially this means undoing any changes the transaction had made, leaving the data in the same state as it was before the transaction commenced.

Long-running transactions are notoriously difficult to implement correctly. And sometimes they are somewhat impossible to implement sensibly - how do you compensate for a business process that has sent an email confirming an order has been shipped, or has mailed an invoice? So, technology solutions for compensating transactions don’t eradicate any of these fundamental problems. But they do provide the designer with a tool to make the existence of a long running transaction explicit, and an execution framework that automatically calls the compensator when failures occur. For many problems this is sufficient for building a workable solution.

As Fig. 31 illustrates, business process orchestration (BPO) platforms are designed to make implementing these long running, highly integrated business processes relatively straightforward.

BPO platforms are commonly built as a layer on top of a message broker. They extend message brokers by adding:

- **State management**: the state of an executing business process is stored persistently in a database. This makes it resilient to BPO server failure. Also, once the process state is stored in the database, it does not consume any resources in the BPO engine.
- **Development tools**: visual process definition tools are provided for defining business processes.
- **Deployment tools**: these enable developers to easily link logical business process steps to the underlying business systems using various types of connectivity, including message queues, web protocols and file systems.

An example from Microsoft’s BizTalk technology is shown in Fig. 32. This shows the design of a simple business process for the ordering exam-
ple in Fig. 30. Messages are sent and received by activities in the process using ports. Ports basically connect to the business systems using a port-defined transport mechanism, for example HTTP, a message queue, or a file. All messages handed inside an orchestration must be defined by XML schemas. Activities can be carried out in sequence, or in parallel as shown in the example.

BPO engines are the most recent addition to the IT middleware stack. The need for their functionality has been driven by the desire to automate more and more business processes that must access numerous independent business applications. There seems little doubt that this trend will continue as enterprises drive down costs by better integrating and coordinating their internal applications, and seamlessly connecting to external business partners.

4.8 Integration Architecture Issues

The difficulty of integrating heterogeneous applications in large enterprises is a serious one. While there are many issues to deal with in enterprise integration, at the core is an architectural problem concerning modifiability. The story goes like this.

Assume your enterprise has five different business applications that need integrating to support some new business processes. Like any sensible architect, you decide to implement these business processes one at a time (as you know a “big bang” approach is doomed to fail!).

The first process requires one of the business systems to send messages to each of the other four, using their published messaging interfaces. To do this, the sender must create a message payload in the format required by each business application. Assuming one-way messages only, this means our first business process must be able to transform its source data into 4 different message formats. Of course, if the other business systems decide to change their formats, then these transformations must be updated. What we’ve created with this design is a tight coupling, namely the message formats, between the source and destination business systems. This scenario is depicted in the left-side of Fig. 33.

With the first business process working, and with many happy business users, you go on to incrementally build the remainder. When you’ve finished, you find you’ve created an architecture like that in the right-side of Fig. 33. Each application sends messages to each of the other four, creating 20 interfaces, or dependencies, that need to be maintained. When one business application is modified, it’s possible that each of the others will need to update their message transformations to send messages in a newly required format.

This is a small scale illustration of a problem that exists in thousands of organizations. I’ve seen enterprise architectures that have three hundred point-to-point interfaces between forty or so standalone business applications. Changing an application’s message interface becomes a scary exercise in such enterprises, as so many other systems are dependent on it. Sometimes making changes is so scary, development teams just won’t do it. It’s simply too risky.

In the general case, the number of interfaces between N applications is (N^2-N). So as N grows, the number of possible interfaces grows exponentially, making such point-to-point architectures non-scalable in terms of modifiability.

Now it’s true that very few enterprises have a fully connected point-to-point architecture such as that on the right-side of Fig. 33. But it’s also true that many interfaces between two applications are two-way, requiring two transformations. And most applications have more than one interface, so in reality the number of interfaces between two tightly coupled applications can be considerably greater than one.

Another name for a point-to-point architecture is a “spaghetti architecture”, hopefully for obvious reasons. When using this term, very few people are referring to spaghetti with the positive connotations usually associated with tasty Italian food. In fact, as the discipline of enterprise integration blossomed in the late 1990’s, the emerging dogma was that spaghetti architectures should be avoided at all costs. The solution pro-
mented, for good reasons, was to use a message broker, as explained earlier in this chapter.

Let's analyze exactly what happens when a spaghetti architecture is transformed using a message broker, as illustrated in Fig. 34. Complexity in the end points, the business applications, is greatly reduced as they just send messages using their native formats to the broker, and these are transformed inside the broker to the required destination format. If you need to change an end point, then you just need to modify the message transformations within the broker that are dependent on that end point. No other business applications know or care.

![Diagram](image)

Fig. 34. Eliminating a point-to-point architecture with a message broker.

Despite all these advantages to introducing a message broker, the no free lunch\(^{22}\) principle, as always, applies. The downsides are:

- The spaghetti architecture really still exists. It's now resident inside the message broker, where complex dependencies between message formats are captured in broker-defined message transformations.
- Brokers are a potentially a performance bottleneck, as all the messages between applications must pass through the broker. Good brokers support replication and clustered deployments to scale their performance. But of course, this increases deployment and management complexity, and more than likely the license costs associated with a solution. Message broker vendors, perhaps not surprisingly, rarely see this last point as a disadvantage.

So message brokers are very useful, but not a panacea by any means for integration architectures. There is however a design approach that can be

\(^{22}\) http://en.wikipedia.org/wiki/Tanstaafl

utilized that possesses the scalability of a point-to-point architecture with the modifiability characteristics of broker-based solution.

The solution is to define an enterprise data model (also known as a canonical data model) that becomes the target format for all message transformations between applications. For example, a common issue is that all your business systems have different data formats to define customer information. When one application integrates with another, it (or a message broker) must transform its customer message format to the target message format.

Now let's assume we define a canonical message format for customer information. This can be used as the target format for any business application that needs to exchange customer-related data. Using this canonical message format, a message exchange is now reduced to the following steps:

- Source application transforms local customer data in to canonical customer information format.
- Source sends message to target with canonical message format as payload.
- Target receives message and transforms the canonical format into its own local customer data representation.

This means that each end point (business application) must know:

- how to transform all messages it receives from the canonical format to its local format
- how to transform all messages it sends from its local format to the canonical format

As Fig. 35 illustrates, by using the enterprise data model to exchange messages, we get the best of both worlds. The number of transformations is reduced to 2^N (assuming a single interface between each end point). This gives us much better modifiability characteristics. Also, as there are now considerably fewer and less complex transformations to build, the transformations can be executed in the end points themselves. We have no need for a centralized, broker-style architecture. This scales well, as there's inherently no bottleneck in the design. And there's no need for additional hardware for the broker, and additional license costs for our chosen broker solution.

I suspect some of you might be thinking that this is too good to be true. Perhaps there is at least a low cost lunch option here?
I'm sorry to disappoint you, but there are real reasons why this architecture is not ubiquitous in enterprise integration. The main one is the sheer difficulty of designing, and then getting agreement on, an enterprise data model in a large organization. In a green field site, the enterprise data model is something that can be designed upfront and all end points mandated to adhere to. But green field sites are rare, and most organization's enterprise systems have grown organically over many years, and rarely in a planned and coordinated manner. This is why broker-based solutions are successful. They recognize the reality of enterprise systems and the need for building many ad hoc transformations between systems in a maintainable way.

There are other impediments to establishing canonical data formats. If your systems integrate with a business partner's applications over which you have no control, then it's likely impossible to establish a single, agreed set of message formats. This problem has to be addressed on a much wider scale, where whole industry groups get together to define common message formats. A good example is RosettaNet²⁶ that has defined protocols for automating supply chains in the semiconductor industry. As I'm sure you can imagine, none of this happens quickly.²⁷

![Fig. 35. Integration using an enterprise data model](image)

For many organizations, the advantages of using an enterprise data model can only be incrementally exploited. For example, a new business systems installation might present opportunities to start defining elements of an enterprise data model, and to build point-to-point architectures that exploit end point transformations to canonical formats. Or your broker

²⁶ www.rosettanet.org
²⁷ See http://www ebxml.org/ for examples of initiatives in this area.

### 4.9 Summary

It's taken the best part of fifteen years to build, but now IT architects have powerful toolkit of 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 pre-requisite 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 a message broker 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.

4.10 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.

4.10.1 CORBA

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

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.

4.10.2 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:


4.10.3 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 lots of good design knowledge about EJB applications in:

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


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


4.10.4 Integration Middleware

An excellent book by one of the leaders in enterprise integration is:


The following three books have broad and informative coverage of design patterns for enterprise integration and messaging.

5 A Software Architecture Process

5.1 Process Outline

The role of an architect is much more than simply carrying out a software design activity. The architect must typically:

- **Work with the requirements team**: The requirements team will be focused on eliciting the functional requirements from the application stakeholders. The architect plays an important role in requirements gathering by understanding the overall systems needs and ensuring that the appropriate quality attributes are explicit and understood.

- **Work with various application stakeholders**: Architects play a pivotal liaison role by making sure all the application's stakeholder needs are understood and incorporated into the design. For example, in addition to the business user requirements for an application, system administrators will require that the application can be easily installed, monitored, managed and upgraded.

- **Lead the technical design team**: Defining the application architecture is a design activity. The architect leads a design team, comprising system designers (or on large projects, other architects) and technical leads in order to produce the architecture blueprint.

- **Work with the project management**: The architect works closely with project management, helping with project planning, estimation and task allocation and scheduling.

In order to guide an architect through the definition of the application architecture, it's useful to follow a defined software engineering process. Fig. 36 shows a simple, three-step iterative architecture process that can be used to guide activities during the design. Briefly, the three steps are: