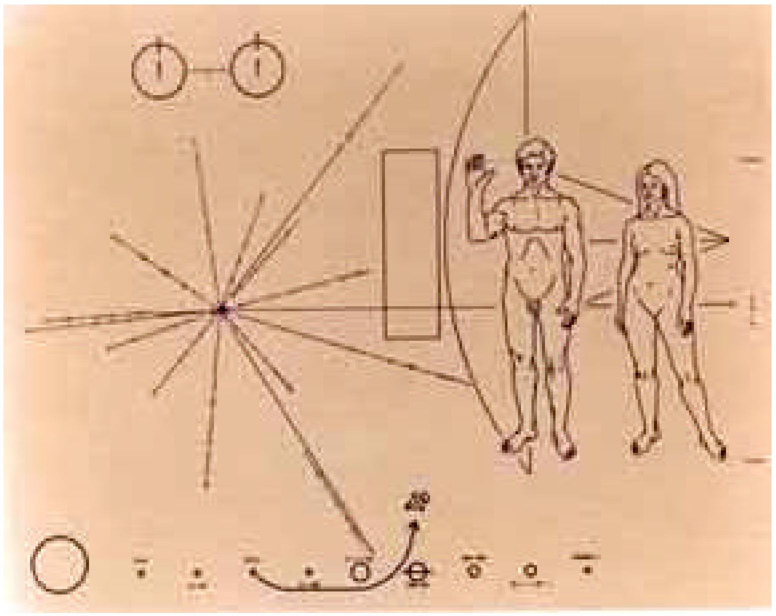
Patterns of System Integration with Enterprise Messaging

Bobby Woolf
Independent Consultant
woolf@acm.org

Kyle Brown
Sr. Technical Staff Member, IBM
brownkyl@us.ibm.com



The Pioneer Plaque by Dr. Carl Sagan, a message to extraterrestrial life forms

Table of Contents

1. Introduction

For almost as long as developers have been writing computer programs, they've needed a way for multiple programs to exchange information and work together. As computer systems have become more sophisticated and widespread, the need to be able to integrate individual applications into enterprise systems has grown. For example, it's no longer enough for an accounts-receivable program to do its job well, it must also serve the broader enterprise by working with other applications that manage payroll, order management, shipping and receiving, and so on. This is a paper about how to integrate enterprise systems using a technology called messaging. Messaging is a key technology that helps make separate applications work together as one to serve the enterprise.

1.1 Program-to-Program Communication

The general problem of program-to-program communication has been a constant and ongoing source of research, development and heartache almost since the dawn of the computer age. In particular, five different ways of program integration have risen to the forefront as the most commonly implemented solutions to this problem.

1. *File Transfer* – This was the original means of program-to-program communication, and is still the basis of many mainframe systems today. In it each program works on a physical file (stored perhaps on a hard disk, a tape drive, or a stack of punched cards) and then another program will take that file as its input and produce a new file, or modify the file if the storage medium allows it. While this has proven effective for solving many problems, issues of latency and resource contention have usually made it unattractive for today's high-speed, high-volume applications.
2. *Shared Database* – Another popular mechanism derived from the File transfer mechanism is the shared database system. In this solution database software handles some of the resource contention issues by providing mechanisms for locking and unlocking the data appropriately, and also provides standard mechanisms for creating, deleting, searching and updating information. However, the latency issue remains even in this solution – before one program can use information it must be written to a database (a physical file) by the other program.
3. *Raw Data Transfer* – In this scheme different programs use a mechanism like a network data transfer protocol (like TCP/IP sockets) or a physical transfer mechanism like Shared Memory to communicate information between different programs. The drawback of this solution (which is again still used in millions of programs) is that it requires synchronous communication – each program must wait on the other to complete its request before processing a response. While it is possible to temporally disconnect the systems, this involves adding significant complexity to the overall system, and involves programming issues that few programmers are competent to deal with – for instance, it is up to the programmer to decide how to guarantee that a message is properly sent and received; the developer must provide retry logic to handle all the cases where the network link is severed, or the request or response was lost in transmission.

4. *RPC* – The RPC (Remote Procedure Call) mechanism is a way of reducing the complexity of the Raw Data Transfer approach by wrapping the network protocols within a layer of code libraries such that it appears to the calling and called programs that a normal procedure call had taken place. Again, RPC is extremely popular, and is the basis of modern systems like CORBA, RMI and EJB. However, the basic issues of synchronicity and guaranteed delivery still remain.
5. *Messaging* – Messaging provides high-speed, asynchronous, program-to-program communication with guaranteed delivery. This particular solution is often implemented as a layer of software called Message Oriented Middleware (MOM). The design of systems to use messaging is the subject of this set of patterns.

As compared to the other four communication mechanisms, relatively few developers have had exposure to messaging and MOM's, and developers in general are not familiar with the idioms and peculiarities of this communications platform. As a result, we have seen many programmers try to use messaging in an inappropriate way, or to develop systems that do not take advantage of the capabilities and strengths of messaging.

1.2 Messaging Systems (MOMs)

A simple way to understand what messaging does is to consider voice mail (as well as answering machines) for phone calls. Before voice mail, when someone called, if the receiver could not answer, the caller hung up and had to call back later to see if the receiver would answer at that time. With voice mail, when the receiver does not answer, the caller can leave him a message; later the receiver (at his convenience) can listen to the messages queued in his mailbox. Voice mail enables the caller to leave a message now so that the receiver can listen to it later, which is often a lot easier than trying to get the caller and the receiver on the phone at the same time. Voice mail bundles (at least part of) a phone call into a message and queues it for later; this is essentially how messaging works.

In enterprise computing, messaging makes communication between processes reliable, even when the processes and the connection between them are not so reliable. There are two reasons processes may need to communicate:

1. One process has data that needs to be transmitted to another process.
2. One process needs to remotely invoke a procedure in another process.

There are two main defacto messaging standards for enterprise computing:

1. The Java 2 Platform, Enterprise Edition (J2EE) includes the Java Message Service API (JMS). [JMS-1]
2. The Microsoft .NET Framework SDK (.NET) includes the `System.Messaging` namespace for accessing MSMQ (see below). [NET-1]

There are a number of products available for embedding messaging into and between applications:

1. One of the oldest and best-known messaging products is IBM's MQSeries. [MQS-1] Its Java client implements the JMS API. [MQS-3]

2. Besides MQSeries, many other products implement the JMS API. [JMS-2]
3. Microsoft's messaging product is Microsoft Message Queuing (MSMQ), which is built into Windows 2000 and later releases. [MSMQ-1]

1.3 Patterns for Messaging

The patterns in this pattern language describe some of the fundamental decisions that go into architecting a system to use messaging. The pattern language is split into three sections:

1. *Message Channel Patterns* describe those patterns that are implemented by most commercial messaging systems; they describe the fundamental attributes of a messaging system, and describe how the different features interrelate. Not all MOM systems implement all of these patterns, however, these patterns all share the same level of abstraction in that they are more "infrastructural" than application-level.
2. *Message Patterns* describe those patterns that describe the form and content of the messages that the messaging systems carry. Some of these patterns may be features implemented by a messaging system, while others refer to generic message formats that implementers of a messaging system will implement.
3. *Messaging* Application *Patterns* describes those patterns that are used by systems designers in architecting messaging systems. Commercial "add-on" products for messaging systems implement many of them, but just as often developers building their own software using messaging systems implement them.

2. Message Channel Patterns

The central pattern in our pattern language is *Messaging*, which describes a technique programs can use to reliably exchange information under unreliable circumstances. Messaging comes in two forms: *Point-to-Point*, which transmits a message that should only be consumed once by one consumer; and *Publish-Subscribe*, which transmits copies of a message to all interested consumers. *Data Type Channel* shows how a sender can transmit different types of data to a receiver such that the receiver will know what each type's data is. When a single channel may carry several types of messages, a *Message Selector* can be used to find only those messages that meet specific criteria. Finally, a *Malformed Message Channel* allows sets of communicating programs to gracefully handle messages that are incorrectly formed or misdirected.

Messaging

My application is distributed among separate processes that communicate via remote procedure calls (RPC's). However, RPC's sometimes fail because of communication problems between the processes.

> **How can we communicate between programs reliably even when neither the network nor the receiver are reliable?**

The simplicity of RPC's is that they're synchronous; a call happens all at once, the caller blocking while the receiver processes. But this is also the shortcoming of an RPC; if anything goes wrong, the whole thing fails. What might go wrong? An RPC consists of three participants, the caller process, the receiving process, and the network connecting the two processes (see Figure 1: Successful RPC).

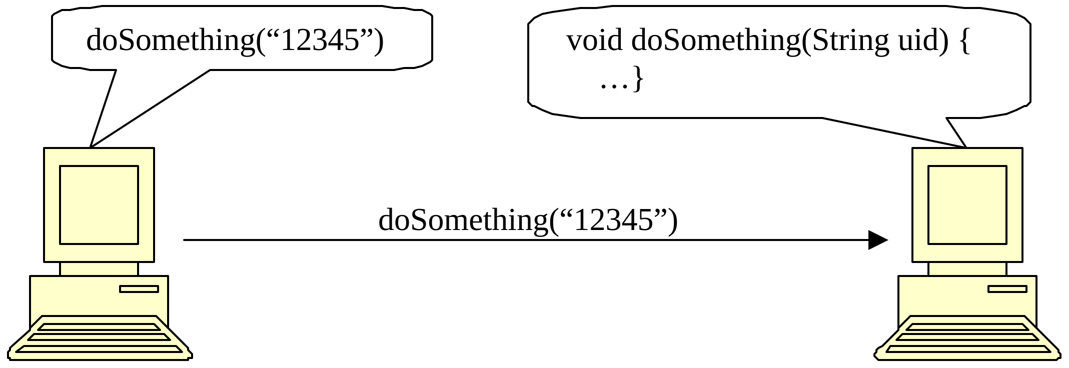


Figure 1: Successful RPC

All three participants have to be working properly, and all at the same time, for the RPC to work and invoke the procedure remotely. When the caller is ready to make the call, the receiver may not be ready (or able) to respond or the network may not be functioning reliably. If anything goes wrong, the RPC fails and is therefore not reliable (see Figure 2: RPC problems).

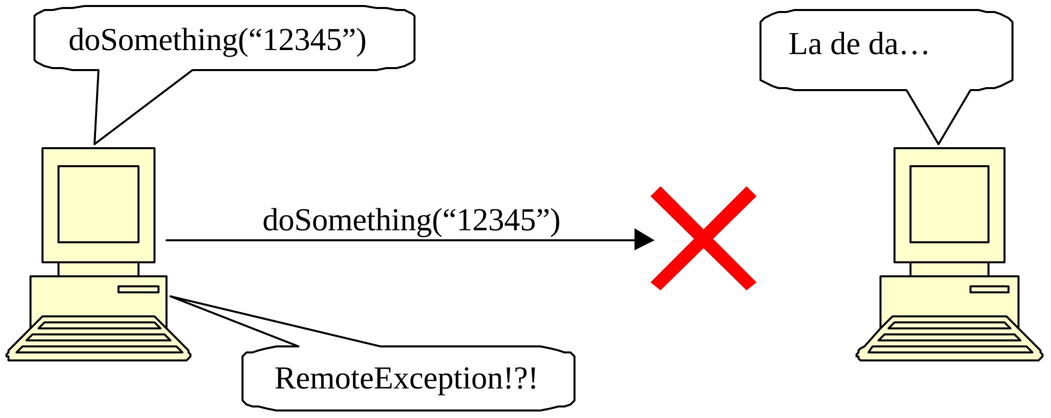


Figure 2: RPC problems

The caller can try to make up for an unreliable RPC by using a retry loop. Whenever an RPC fails, the caller retries it until it works. However, continuously retrying a call that has already failed repeatedly quickly becomes futile. After some number of retries, perhaps the caller should pause for a while before retrying again, in hopes that the problem will be fixed during the pause. But how will the source remember what RPC to try after the pause? It will need to somehow queue up the RPC, and then retry the queued RPC's periodically. And still, no matter how many times the caller pauses and retries the RPC, the call will only work if the caller, network, and receiver are all working when the call is made.

What if the call could be made even when the network and/or the receiver were not ready? Instead of queuing the RPC on the caller and retrying it repeatedly, what if the queued RPC call could be moved to the receiver and retried locally? This would make the call asynchronous, in that the caller would not know when the call actually got invoked on the receiver, just that it would happen eventually. The advantage would be that the call could be queued when the caller was ready, moved from the caller to the receiver when the network was ready, and invoked on the receiver when it was ready, even if the three parts were never ready at the same time.

Therefore:

> **Use *messaging* to make intra-program communication reliable, even when the network and the receiver program cannot be relied upon.**

By packaging the call as a message, the caller can queue the call to be delivered to and invoked on the receiver as soon as possible, whenever that may be. If delivery fails, it can be retried until it succeeds (or times out). This is illustrated in Figure 3: Queued communication.

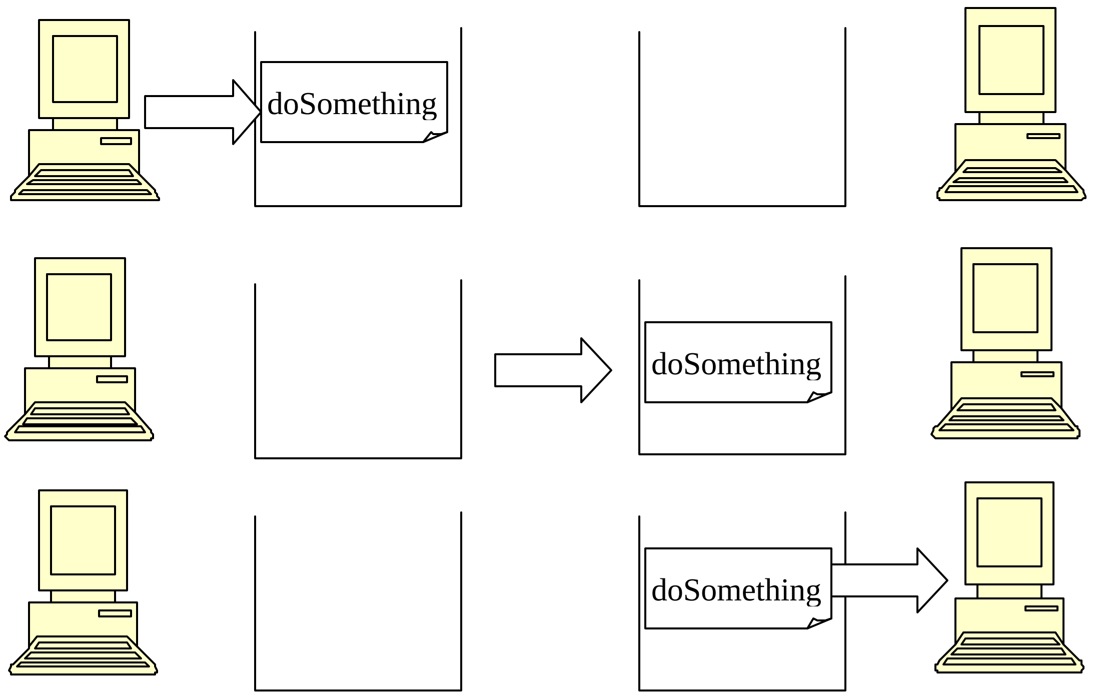


Figure 3: Queued communication

When an application uses messaging, the communication between the processes is made more reliable, and in addition, programs do not have to pause (or hang, thus occupying processor time in a useless "wait" loop) nor have to repeatedly call the receiver, again a futile and time-wasting exercise when either the network or receiver process are not ready. However, in this situation, the program will have to become a bit more complex. More importantly, the program may have to deal with asynchronicity, since the time between the original message transmission and reply receipt may be quite long – too long to wait for in a "busy loop".

Although you could probably write your own messaging system, there are many commercial ones already available. One of the oldest and best-known messaging products is IBM's MQSeries. Many messaging products (including the MQSeries Java client) implement the Java Message Service API (JMS). The .NET SDK contains the `System.Messaging` namespace, which provides access to Microsoft Message Queuing (MSMQ), the messaging product built into Windows.

JMS provides two modes for messaging, persistent and non-persistent. A message sent persistently receives the highest quality of service. When a persistent message is sent, the send action is not complete until the messaging system has stored the message in a non-volatile store (e.g., a database) so that the message will survive even if the messaging system crashes. Likewise, the messaging system must deliver a persistent message once and only once. The messaging system uses transactions internally to ensure this behavior (even if a client session is non-transactional). These persistent transactions will, of course, hurt performance, but are necessary overhead for reliable messaging. **This paper assumes that messages are sent persistently, thus ensuring the greatest level of reliability.**

There are two distinct approaches for callers to send messages to receivers: *Point-to-Point* and *Publish-Subscribe*. Either way, the caller will need to package the communication as a *Command Message*, *Document Message*, or *Event Message*. To implement a full remote procedure call, one that returns a result or at least notifies the caller when the call has completed successfully, the receiver will have to send the caller a *Reply Message*. If a message's transmission retries for too long without transmitting successfully, it may time out because of its *Message Expiration*.

Point-to-Point

My application is using *Messaging* to make remote procedure calls (RPC's) or transfer documents.

How can the caller be sure that only one receiver will receive the document or perform the call?

One advantage of an RPC is that it's invoked on a single remote process, so either that receiver performs the procedure or it does not (and an exception occurs). And since the receiver was only called once, it only performs the procedure once. But with messaging, once a call is packaged as a message and placed on a channel, potentially many receivers could see it on the channel and decide to perform the procedure.

The messaging system could prevent more than one receiver from monitoring a single channel, but this would unnecessarily limit callers that wish to transmit data to multiple receivers. All of the receivers on a channel could coordinate to ensure that only one of them actually performs the procedure, but that would be complex, create a lot of communications overhead, and generally increase the coupling between otherwise independent receivers. Multiple receivers on a single channel may be desirable so that multiple messages can be consumed concurrently, but any one receiver should consume any single message.

Therefore:

Configure the channel to use *point-to-point* messaging, which ensures that only one receiver will receive a particular message.

A point-to-point channel ensures that only one receiver consumes any given message. If the channel has multiple receivers, only one of them can successfully consume a particular message. If multiple receivers try to consume a single message, the channel ensures that only one of them succeeds, so the receivers do not have to coordinate with each other. The channel can still have multiple receivers to consume multiple messages concurrently, but only a single receiver consumes any one message.

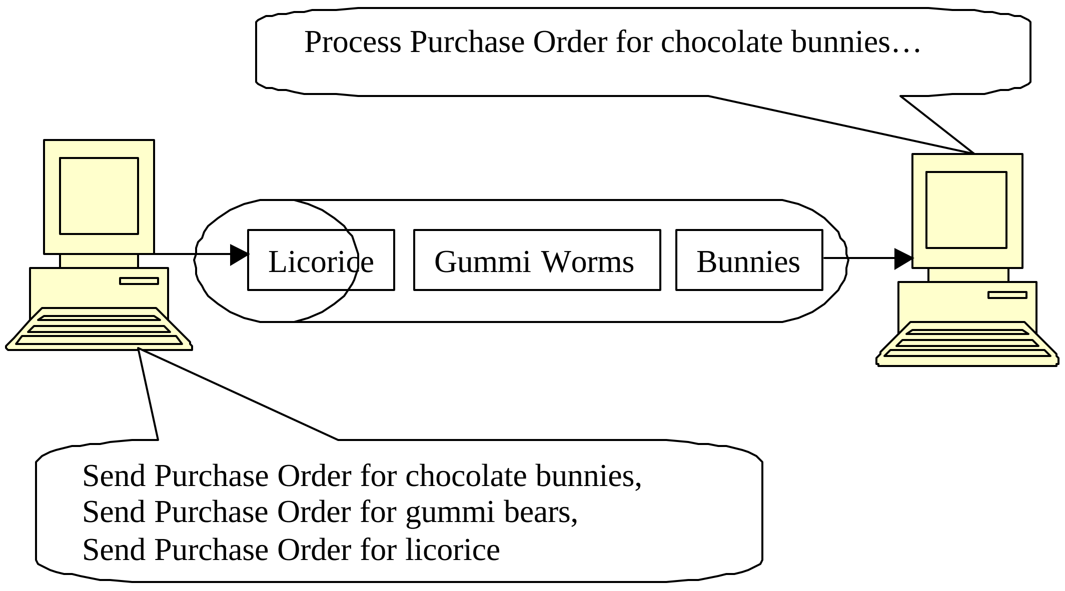

Process Purchase Order for chocolate bunnies…
Licorice
Gummi Worms
Bunnies
Send Purchase Order for chocolate bunnies,
Send Purchase Order for gummi bears,
Send Purchase Order for licorice

Figure 4: Point to Point connection

In JMS, a point-to-point channel implements the `Queue` interface. In .NET, the -`MessageQueue` class implements a point-to-point channel. MSMQ, which implements .NET messaging, only supported point-to-point messaging prior to version 3.0, so point-to-point is what .NET supports.

In a stock trading system, the request to make a particular trade is a message that should be consumed and performed by exactly one receiver, so the message should be placed on a point-to-point channel.

To implement an RPC using messaging, use a pair of point-to-point channels, one channel for the call message a reverse channel for a *Reply Message*. The call is a *Command Message* whereas the reply is a *Document Message*.

Publish-Subscribe

My application is using *Messaging* to communicate events.

How can the sender broadcast an event to all interested receivers?

Luckily, there are well-established patterns for implementing broadcasting. The Observer pattern [GHJV95] describes the need to decouple observers from their subject so that the subject can easily provide event notification to all interested observers no matter how many observers there are (even none). The Publisher-Subscriber pattern [BMRSS95] expands upon Observer by adding the notion of an event channel for communicating event notifications.

That’s the theory, but how does it work with messaging? The event can be packaged as a message so that messaging will reliably communicate the event to the observers (subscribers). Then the event channel is a messaging channel. But how will a messaging channel properly communicate the event to all of the subscribers?

Each subscriber needs to be notified of a particular event once, but should not be notified repeatedly of the same event. The event cannot be considered consumed until all of the subscribers have been notified. But once all of the subscribers have been notified, the event can be considered consumed and should disappear from the channel. Yet having the subscribers coordinate to determine when a message is consumed violates the decoupling of the observer pattern. Concurrent consumers should not be considered to compete, but should be able to share the event message, but only so long as the consumers are not somehow part of the same subscriber.

Therefore:

> **Configure the channel to use *publish-subscribe* messaging, which sends a copy of a particular message to each receiver.**

A publish-subscribe channel works like this: It has one input channel that splits into multiple output channels (a.k.a., event channels), one for each subscriber. When an event is published into the channel, the publisher-subscriber consumes the input message by placing a copy of the message into each of the event channels. Each event channel has only one subscriber, which is only allowed to consume a message once. In this way, each subscriber only gets the message once and consumed copies disappear from their channels.

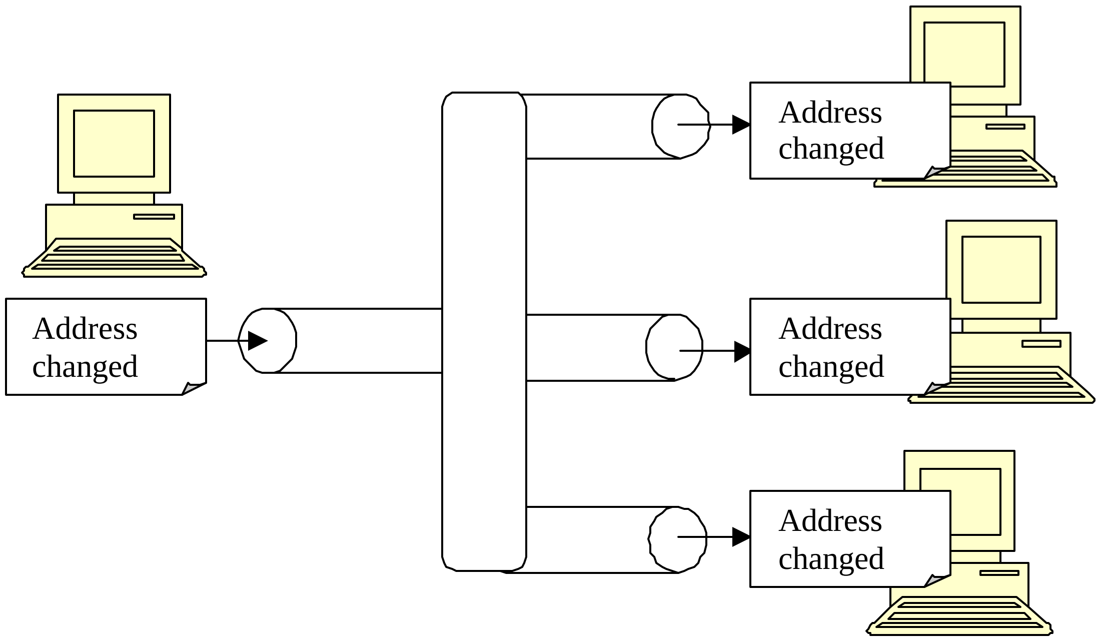


Figure 5: Publish-Subscribe communication

In JMS, a publish-subscribe channel implements the `Topic` interface. The sender uses a `TopicPublisher` to send messages; each receiver uses its own `TopicSubscriber` to receive messages. A subscription can be durable or non-durable—the difference is what the messaging system does with messages received while a subscriber does not have an open connection. With a durable subscription, when the subscriber does not have an open connection, the messaging system will queue messages until the subscriber reads them. With a non-durable subscription, if the messaging system has messages for the subscriber but the subscriber does not have an open connection for retrieving them, the messaging system will throw away that subscriber's copies of the messages. **This paper assumes that subscribers are durable, thus ensuring the greatest level of reliability.**

MSMQ 3.0 adds what it calls a "one-to-many messaging model," which has two different approaches:

1. *Real-Time Messaging Multicast* – This most closely matches publish-subscribe, but its implementation is entirely dependent on IP multicasting via the Pragmatic General Multicast (PGM) protocol.
2. *Distribution Lists and Multiple-Element Format Names* – A Distribution List enables the sender to explicitly send a message to a list of receivers (but this violates the spirit of the Observer pattern). A Multiple-Element Format Name is a symbolic channel specifier that dynamically maps to multiple real channels, which is more the spirit of the publish-subscribe pattern but still forces the sender to choose between real and not-so-real channels. [MSMQ-3]

In a stock trading system, many systems may need to be notified of the completion of a trade, so make them all subscribers of a publish-subscribe channel that publishes trade completions.

An event on a publish-subscribe channel is often an *Event Message* because multiple dependents are often interested in an event. If a subscriber wishes to acknowledge a notification, it can send a *Reply Message* back to the publisher (probably via a *Point-to-Point* channel).

Data Type Channel

My application is using *Messaging* to transfer different types of data, such as different types of documents.

> **How can I transmit a data item such that the receiver will know how to process it?**

The message contains data—a document of some kind—so obviously it's a *Document Message*. But there can be different types of data. If the receiver does not know what type of data it's receiving, it must test the data, using a case statement (which is inflexible and error-prone).

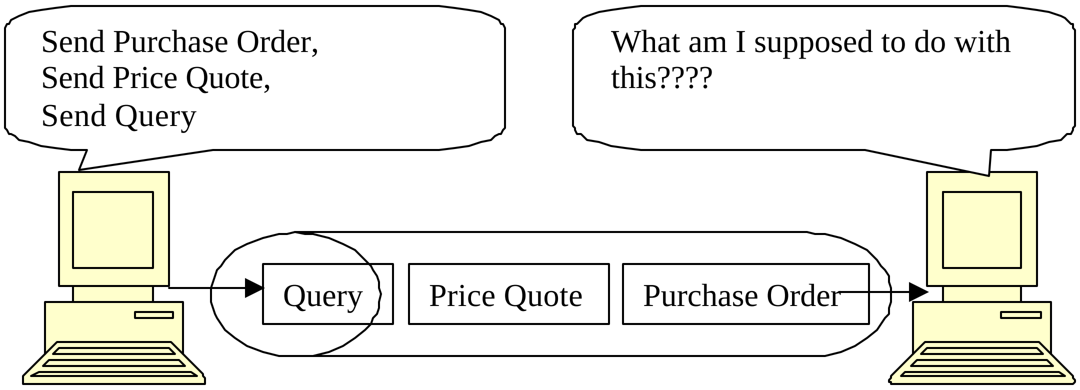


Figure 6: Mixed Data Types

The sender knows what type of data it's sending, so how can this be communicated to the receiver? The sender could put a flag in the message's header, but then the receiver will need a case statement again. The sender could wrap the data in a *Command Message* with a different command for each type of data, but that presumes to tell the receiver what to do with the data when all that the message is trying to do is transmit the data to the receiver.

A principle of messaging is that all of the messages on a channel must be of the same type. That way the receivers always know what to expect. Here the problem is that messages of different types are being sent on a single channel.

Therefore:

> **Use a separate *data type channel* for each data type, so that all data on a particular channel is of the same type.**

That way, all of the document messages on a given channel will contain the same type of data. The sender, knowing what type the data is, will need to select the appropriate channel to send it on. The receiver, knowing what channel the data was received on, will know what its type is.

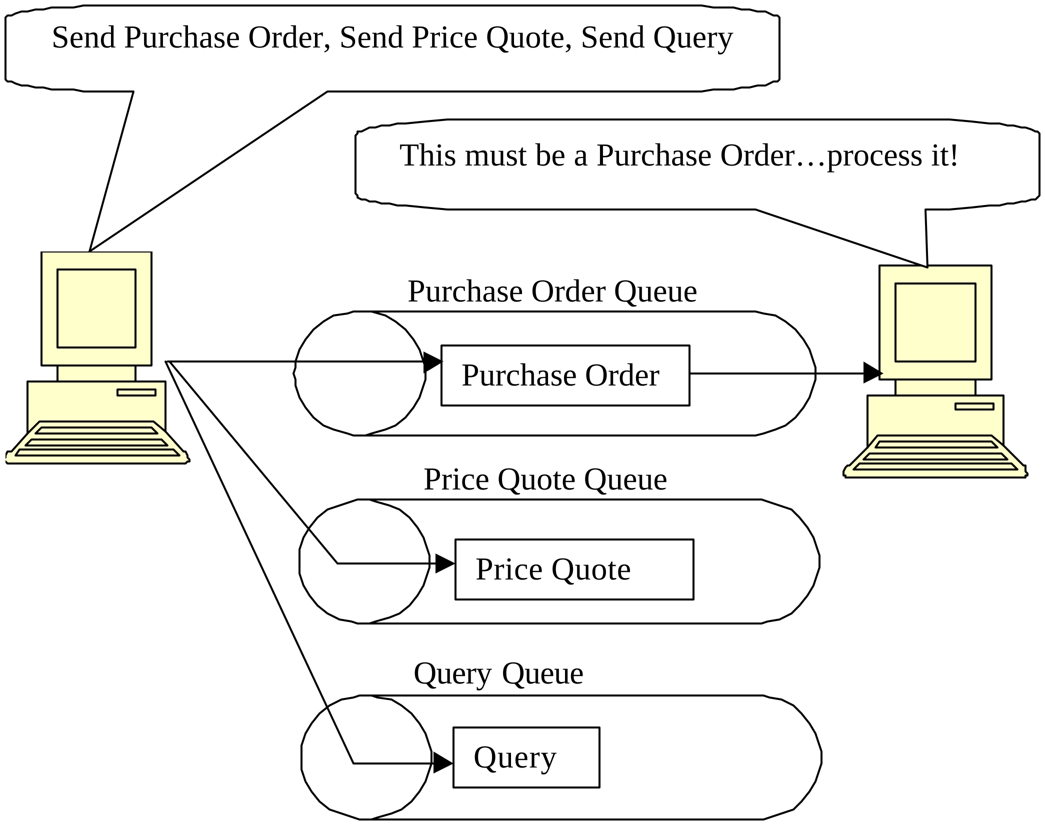

Send Purchase Order, Send Price Quote, Send Query
This must be a Purchase Order…process it!
Purchase Order Queue
Purchase Order
Price Quote Queue
Price Quote
Query Queue
Query

Figure 7: Data Type Channels

The message still looks like a document message, just that there are now multiple channels for these document messages, one for each data type. For example, the sender may need to transmit several types of documents to the receiver, such as: purchase order, price quote, and query. Thus the sender and receiver would need three channels for transmission: a purchase order channel, a price quote channel, and a query channel. If the sender needs to transmit a query, it must use the query channel. When the receiver gets a message on the query channel, it will know the message data is a query.

An application may need to transmit many different data types, too many to create a separate Data Type Channel for each. In this case, multiple data types can share a single channel by using a different *Message Selector* for each type. This makes a single channel act like multiple data type channels.

Malformed Message Channel

My application is using *Messaging* to communicate between processes.

> **How can a messaging receiver gracefully handle receiving a message that makes no sense?**

In theory, everything on a message channel is just a message and message receivers just process messages. However, the fact is that receivers and message formats are customized for their part of the application, so each receiver must make a lot of assumptions about the content of the messages it's processing. While it would be nice to assume that no sender would ever put a message on a channel with an incorrect format, the fact is that things can go wrong. As long as a valid message object is addressed to a valid channel, the messaging system will pass the message through, so the receiver needs to be prepared to defend itself against improperly formatted messages.

Worse, a malicious sender could purposely send badly formatted messages. Or a completely separate sender could decide to start using an existing channel for its own purposes, transmitting messages of a different format. Most messaging systems do not provide security features to control which senders and receivers can access which channels.

For a receiver to get its bearings on what to do with a message, it has to look for certain expected flags in the header, has to determine the message body's sub-type (bytes, text, etc.) and usually will perform some initial parsing of the body's data. Any of this can go wrong if the message format is not valid. This leads to a lot of if-then-else code in the receiver: if the message has some expected feature, then use that feature, however, if the feature is not there, then the resulting exception must be handled.

When the receiver finds that a message format is invalid, it could put the message back on the channel, but then the message will just be re-consumed by the receiver or another like it. The receiver could ignore messages without a valid message selector, but the ignored messages will clutter the channel and hurt performance, and this still does not help the receiver handle a message with a valid message selector but some other formatting problem. The receiver could just ignore the badly formatted message, consuming it from the queue but then throwing it away, but this could cause serious problems with the messaging to go undetected. What the system needs is a way to clean malformed messages out of channels and put them somewhere out of the way, yet a place where these malformed messages can be detected to diagnose problems with the messaging system.

Therefore:

> **Create a *malformed message channel* so that when a receiver discovers a badly formatted message, it can put that message on the malformed channel.**

The malformed channel will not be used for normal, successful communication, so its being cluttered with malformed messages will not be a problem. An error handler that wants to diagnose malformed messages can use a receiver on the malformed channel to detect messages as they become available.

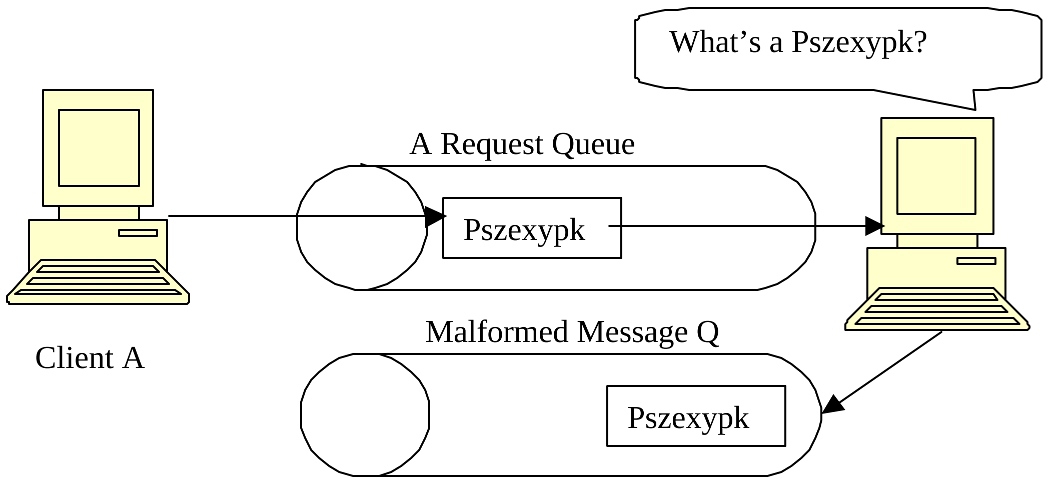


Figure 8: Malformed Message Processing

For example, if a receiver reads what is supposed to be a text message but finds that it's a byte message, it can put the message on the malformed channel. If a message is supposed to have a header flag such as a *Correlation Identifier*, *Message Sequence* identifiers, *Reply Channel*, etc., but the message is missing the flag, the receiver can move the message to the malformed channel.

In JMS, the specification suggests that if a `MessageListener` gets a message it cannot process, a well-behaved listener should divert the message to "to some form of application-specific 'unprocessable message' destination." [JMS02, p. 69] This unprocessable message destination is a malformed message channel.

Messaging systems have a similar concept called "dead message queue" [MHC01, p. 125] or "dead-letter queue." [IBM00, p. 57], [MSMQ-2] The messaging system considers a message dead when it cannot be delivered properly; reasons include a message whose destination channel no longer exists, a message that expires, etc. Whereas dead messages cannot be delivered successfully by the messaging system, malformed messages are delivered successfully, but the receiver considers them malformed. Thus the messaging system can route dead messages to the error queue automatically, but receivers of malformed messages must manually add them to the error queue.

3. Message Patterns

These patterns describe the form and format of messages that flow between parts of a system built with a messaging system. A *Command Message* enables procedure call semantics to be executed in a messaging system. A *Document Message* enables a messaging system to transport a document or information, such as the information that should be returned to a sender as a result of a command message. An *Event Message* uses messaging to perform event notification. A *Reply Message* handles the semantics of remote procedure call results, which require the ability to handle both successful and unsuccessful outcomes. A *Reply Specifier* enables a program making a request to identify the channel on which a reply should be sent. A *Correlation Identifier* enables a requesting program to associate a specific response with its request. When the data to be conveyed may span several messages, a *Message Sequence* enables the receiver to accurately reconstruct the original data. *Message Expiration* enables a sender to set a deadline by which the message should either be delivered or ignored. Finally, *Message Throttle* enables a receiver to control the rate at which it consumes messages.

Command Message

My application is using *Messaging* to make remote procedure calls (RPC's).

How can a remote procedure be called as a message?

Luckily, there's a well-established pattern for how to encapsulate a request as an object. The Command pattern [GHJV95] shows how to turn a request into an object that can be stored and passed around. If this object were a message, then it could be stored in and passed around through a messaging channel. Likewise, the command's state (if any) can be stored in the message's state.

Therefore:

Use a *command message* to package an RPC as a message that can be placed on a channel.

There is no specific message type for commands; a command message is simply a regular message that happens to contain a command. In JMS, the command message could be any type of message; examples include an `ObjectMessage` containing a `Serializable` command object, a `TextMessage` containing the command in XML form, etc. In .NET, a command message is a `Message` with a command stored in it. A Simple Object Access Protocol (SOAP) request is a command message.

For example, the SOAP protocol [SOAP-1] has a convention that the body of a SOAP request message consists of an XML element that corresponds to the name of a method that should be invoked on the receiving end. Likewise this element will contain a set of other elements that correspond to the parameters of this remote method. An example of this is shown below:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=
     "http://www.w3.org/2001/06/soap-envelope"
  SOAP-ENV:encodingStyle=
     "http://www.w3.org/2001/06/soap-encoding">

  <SOAP-ENV:Body>
     <m:GetExchangeRate xmlns:m="CurrencyIsUsURI">
        <country1>USA</country1>
        <country2>Japan</country2>
     </m:GetExchangeRate>
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Command messages are usually sent *Point-to-Point* so that each command will only be consumed and invoked once. A true RPC not only invokes the procedure but also returns a result, so a command message often has a matching *Reply Message* that is a *Document Message* containing the procedure's result or any exceptions.

Document Message

My application is using *Messaging* to communicate between processes.

How can messaging be used to transfer a document from one process to another?

This is a classic problem in distributed processing: One process has data another one needs. An RPC can be used to send the data, but then the caller is also telling the receiver—via the procedure being invoked—what to do with the data. Likewise, a *Command Message* would transfer the data, but would be overly specific about what the receiver should do with the data.

Yet we do want to use messaging to transfer the data. Messaging is more reliable than an RPC. *Point-to-Point* messaging can be used to make sure that only one receiver gets the data (no duplication), or *Publish-Subscribe* messaging can be used to make sure that any receiver who wants the data gets a copy of it. So the trick is to take advantage of messaging without making the message too much like an RPC.

Therefore:

Use a *document message* to reliably transfer a document between two processes.

Whereas a command message is like an RPC, a document message is like a single parameter of an RPC, a single unit of data (which may in turn contain smaller units of data). Whereas an RPC and command message tells the receiver what to do, a document message just transfers a unit of data to the receiver without telling the receiver what to do with it.

To the messaging system, a document message looks just like a command message; they're both just packets of data being sent through a messaging channel. In JMS, the document message may be an `ObjectMessage` containing a `Serializable` data object for the document, or it may be a `TextMessage` containing the data in XML form. In

.NET, a document message is a `Message` with the data stored in it. A Simple Object Access Protocol (SOAP) reply message is a document message.

The following example (drawn from the example XML schema in [Graham]) shows how a simple purchase order can be represented as XML and sent as a message using JMS.

```
// Assume that we have obtained a Session (session)
// and a Destination (dest) from JNDI lookups

//Create a sender
MessageProducer sender = session.createProducer(dest);

String purchaseOrder =
" <po id=\"48881\" submitted=\"2002-04-23\">
    <shipTo>
      <company>Chocoholics</company>
      <street>2112 North Street</street>
      <city>Cary</city>
      <state>NC</state>
      <postalCode>27522</postalCode>
    </shipTo>
    <order>
      <item sku=\"22211\" quantity=\"40\">
        <description>Bunny, Dark Chocolate, Large</description>
      </item>
    </order>
  </po>";

TextMessage message = session.createTextMessage();
message.setText(purchaseOrder);

//Send the message
sender.send(message);
```

Document messages are usually sent *Point-to-Point* to move the document from one process to another without duplicating it. A document message can be broadcast via *Publish-Subscribe*, but this creates multiple copies of the document that probably need to be read-only. A *Reply Message* is usually a document message—the result is intended for a single receiver and should not be ignored—where the result value is the document. When the “document” is a transient one describing an event, that’s an *Event Message*.

Event Message

My application is using *Messaging* to communicate between processes.

How can messaging be used to communicate events from one process to another?

An event is a special kind of data. It’s not a document with a lifetime in the sender process or the receiver. It exists just long enough for a subject to tell its observer what just happened (a la the Observer pattern [GHJV95]). Once the subject sends the event, it does not need the data anymore; once the observer receives and processes the event, it does not need the data anymore either. So this data is not a document to be read, modified, passed around, and saved for future use, it’s just a notification. Also, unlike a document, an event can often be ignored, such as when the observer is too busy to process the event.

In the Observer pattern, a subject notifies an observer of an event by calling the observer's `Notify()` method. Yet if the observer is in a different process from the subject, this call is an RPC that may not be reliable. Messaging is a good way to make RPC's reliable.

Therefore:

Use an *event message* for reliable event notification between processes.

When a subject has an event to announce, it will create an event object, wrap it in a message, and put it on a channel. The observer will receive the event message, get the event, and process it. Messaging does not change the event notification, just makes sure that the notification gets to the observer.

In Java, an event can be an object or data such as an XML document. Thus they can be transmitted through JMS as an `ObjectMessage`, `TextMessage`, etc. In .NET, an event message is a `Message` with the event stored in it.

There is usually no reason to limit an event message to a single receiver via *Point-to-Point*; the message is usually broadcast via *Publish-Subscribe* so that all interested processes receive notification. Whereas a *Document Message* needs to be consumed so that the document is not lost, a receiver of event messages can often ignore the messages when it's too busy to process them. Event message is a key part of implementing the Observer pattern using messaging; the complete design will be discussed in another chapter.

Reply Message

My application is using *Messaging*, specifically *Command Messages*, to make remote procedure calls (RPC's).

How does a caller get the result of a command message?

As explained in the Messaging pattern, an RPC is a synchronous call, so the caller blocks while the receiver runs the procedure. When the procedure finishes, its result is returned back to the caller. Messaging makes the call asynchronous, so the caller does not know when the receiver invokes the procedure, much less when the procedure finishes or what the result was. Yet even with messaging, the caller may still need to know when the procedure finishes, what the result value was, or what exception occurred.

The caller could somehow block synchronously waiting for the result, but that does not magically make the result travel back from the receiver. The caller might be able to use the messaging system in some synchronous way to block on sending the message until the receiver consumes the message, but that just tells the caller when the message was consumed, not what the procedure's result was. The receiver could add the result to the caller's message, but the caller will never see that because it does not consume the message, the receiver does. The caller could try to consume the message looking for a result, but then the caller is consuming its own messages, which is chaotic for the messaging system and will probably prevent the receiver from ever actually receiving the message.

Yet there must be some way for the caller to get the procedure's result. This is the problem that a *Document Message* solves: The receiver has data—the procedure result—that the caller needs. So do not try to add the result to the original message. Send the result back using a whole new message just for this purpose.

Therefore:

> **The receiver should use a *reply message* to send the procedure's result back to the caller.**

The result is just data, so the reply message will be a document message. Reply messages can also be used to transmit events back to the caller, such as acknowledging receipt of the original message, but the idea of any reply message is that it's closely associated one-to-one with a previous request message from the caller.

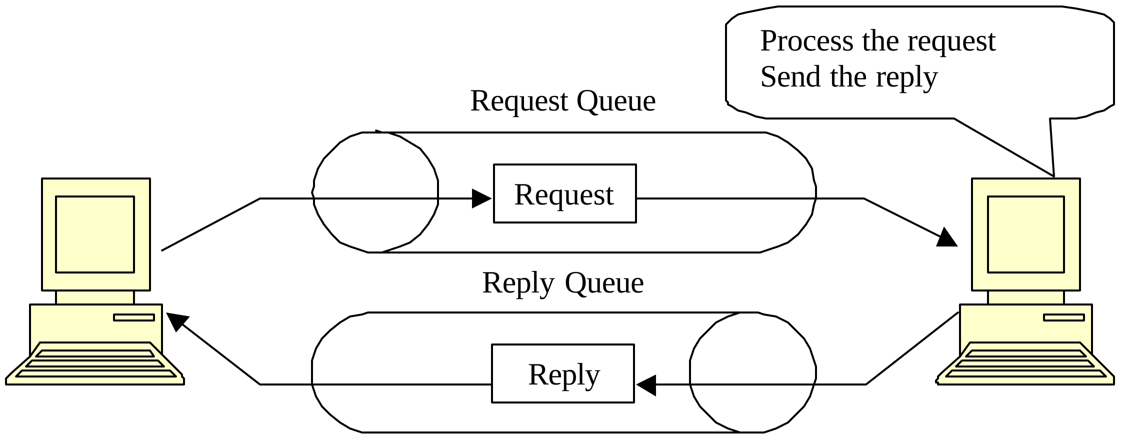


Figure 9: A receiver sending a reply

When a reply message indicates the result of a procedure call, it can be thought of as a "result message." In an object-oriented environment, a "procedure" (remote or otherwise) is really a message invocation, which can result in one of three ways:

1. Returns `void` – Simply notifies the caller that the message has finished so that the caller can stop blocking.
2. Returns a value – A single object the message uses as its result.
3. Returns (throws) an exception – A single exception object indicating that the message aborted before completing successfully; also indicates why.

Since these are the three things a message result can do, they're the three things a result message must be able to contain. The message must indicate success or failure, so there are two types of result messages, organized like this:

1. Success
    a. Returns `void` – Tells the caller that the message has finished and that it executed successfully.
    b. Returns a value – Tells the caller that the message has finished, that it executed successfully, and that this value was the result.

2. Failure
    a. Returns (throws) an exception – Tells the caller that the message has finished, but not successfully, and why it failed.

Note that the SOAP specification [SOAP-1] describes one particular implementation of this pattern, in that it shows how when SOAP is used to implement an RPC that the results returned can either be a call response message (which may or may not contain a return value), or a SOAP Fault, which corresponds to the failure scenario.

JMS provides for a variation of this pattern using a *temporary queue*. The requestor uses its JMS connection to create a temporary queue and sets that as the `JMSReplyTo` property of the request message (see *Reply Specifier*). The replier obtains access to the temporary queue from the request message and sends the reply to that queue. [MHC01, pp. 67-69] This offers a certain level of security in that only a consumer of the request can possibly send a message on the reply queue. However, this technique is not always reliable, because a temporary queue can only provide non-persistent messaging. The temporary queue's lifetime ends when the connection that created it closes, so if the requestor and its connection crash, all reply messages will be lost and no new ones can be sent. (The messaging system may put such lost messages in a dead letter queue; see *Malformed Message Channel*.) Thus, to transmit replies reliably, a permanent queue with persistent messaging is more appropriate.

The caller may wish to indicate a *Reply Specifier* in its request message so the receiver will know where to send the reply. A reply message may need a *Correlation Identifier* to identify the original request it's replying to.

Reply Specifier

My application is using *Messaging* to send a request message for which it expects a *Reply Message*.

How does the receiver of a request message know where to send the reply message?

Messages are often thought of as completely independent, such that any sender sends a message on any channel whenever it likes. However, messages are often associated, such as request-reply pairs, two messages which appear independent but where the reply message has a one-to-one correspondence with the request message that caused it. Thus the receiver that processes the request message cannot simply send the reply message on any channel it wants, it must send it on the channel the caller expects the reply on.

Each receiver could automatically know which channel to send replies on, but hard coding such assumptions makes the software less flexible and more difficult to maintain. Furthermore, a single receiver could be processing calls from several different callers, so the caller it should reply to is the one that sent the request.

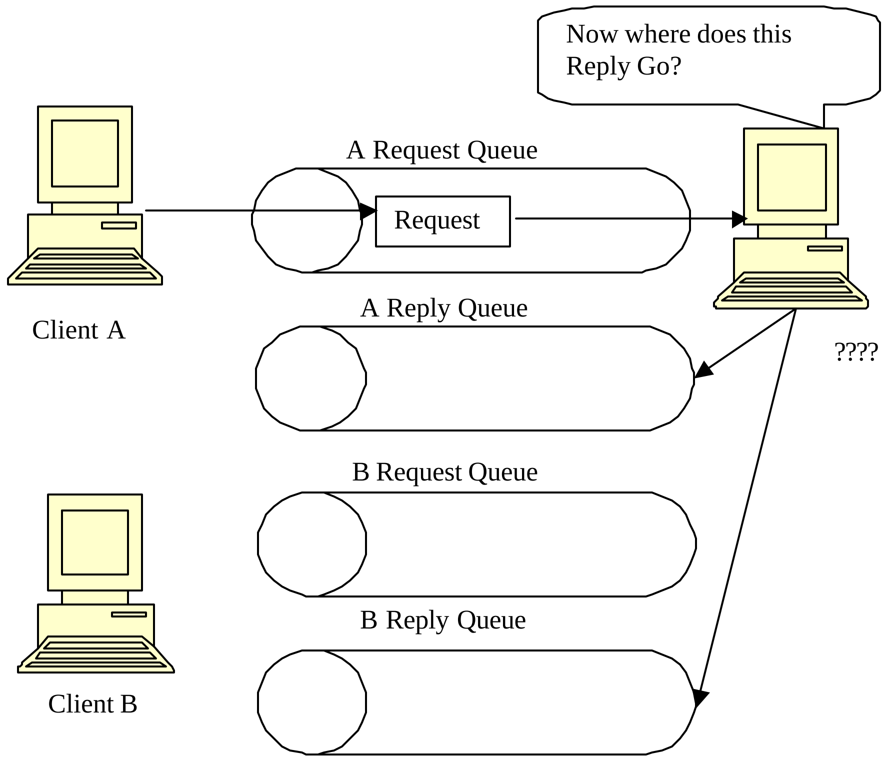


Figure 10: Unsure where to send the reply

A caller potentially may not want a reply sent back to itself. Rather, it may have an associated callback processor to process replies, and the callback processor may monitor a different channel than the caller does. The caller could have multiple callback processors such that replies for different requests from the same caller should be sent to different processors.

The reply channel will not necessarily transmit replies back to the caller; it will transmit them to whomever the caller wants to process the replies, because it's listening to the channel the caller specified. So knowing what caller sent a request or what channel it was sent on does not necessarily tell the receiver what channel to send the reply on. Even if it did, the receiver would still have to infer which reply channel to use for a particular caller or request channel. It's easier for the request to explicitly specify which reply channel to use.

What is needed is a way for the caller to tell the receiver where and how to send a reply back.

Therefore:

> **The request message should contain a *reply specifier* that indicates where to send the reply message.**

This way, the receiver does not need to know where to send the reply, it can just ask the request. If different messages to the same receiver require replies to different places, the receiver knows where to send the reply for each request. This encapsulates the knowledge of what channels to use for requests and replies within the caller so those decisions do not have to be hard coded within the receiver. A reply specifier is put in the header of a message because it's not part of the data being transmitted.

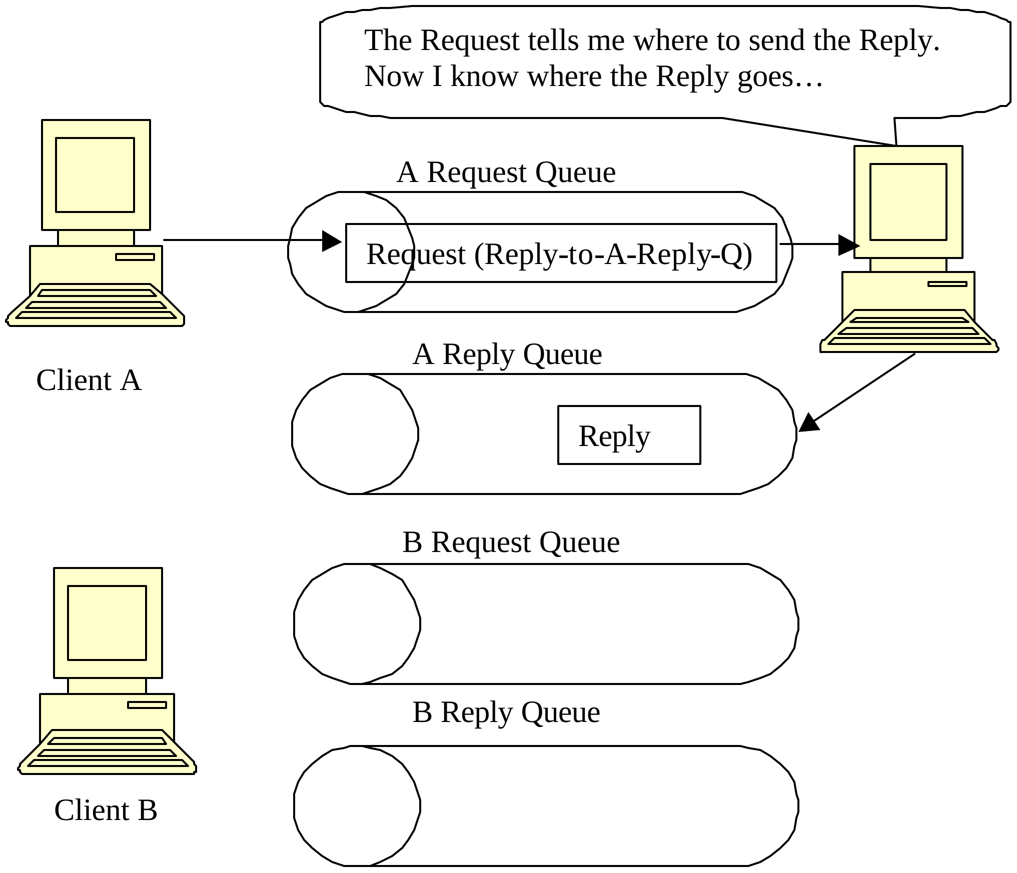

The Request tells me where to send the Reply.
Now I know where the Reply goes…
A Request Queue
Request (Reply-to-A-Reply-Q)
Client A
A Reply Queue
Reply
B Request Queue
B Reply Queue
Client B

Figure 11: Using Reply Specifier

A message's reply specifier is analogous to the reply-to field in an e-mail message. The reply-to e-mail address is usually the same as the from address, but the sender can set it to a different address to receive replies in a different account than the one used to send the original message.

JMS messages have a predefined property for reply specifiers, `JMSReplyTo`. Its type is a `Destination` (a `Topic` or `Queue`), rather than just a string for the destination name, which ensures that the destination (e.g., channel) really exists, at least when the request is sent.

A sender that wishes to specify a reply channel that is a queue would do so like this:

```
Queue requestQueue = // Specify the request destination
Queue replyQueue = // Specify the reply destination
Message requestMessage = // Create the request message
requestMessage.setJMSReplyTo(replyQueue);
MessageProducer requestSender =
  session.createProducer(requestQueue);
requestSender.send(requestMessage);
```

Then the receiver would send the reply message like this:

```
Queue requestQueue = // Specify the request destination
MessageConsumer requestReceiver =
  session.createConsumer(requestQueue);
Message requestMessage = requestReceiver.receive();
Message replyMessage = // Create the reply messageDestination
replyQueue = requestMessage.getJMSReplyTo();
MessageProducer replySender = session.createProducer(replyQueue);
replySender.send(replyMessage);
```

.NET messages also have a predefined property for reply specifiers, `ResponseQueue`. Its type is a `MessageQueue`, the queue that the application should send a response message to.

When the reply message is sent back the channel indicated by the reply specifier, it may also need a *Correlation Identifier*. The reply specifier tells the receiver what channel to put the reply message on; the correlation identifier tells the sender which request a reply is for.

Correlation Identifier

My application has received a *Reply Message*.

How does a sender that has received a reply know which request this is the reply for?

When one process invokes another via a Remote Procedure Call (RPC), the call is synchronous, so there is no confusion about which call produced a given result. But messaging is asynchronous, so from the caller's point of view, it makes the call, then sometime later a result appears. The caller may not even remember making the request, or may have made so many that it no longer knows which one this is the result for. With confusion like this, when the caller finally gets the result, it may not know what to do with it, which sort of defeats the purpose of making the call in the first place.

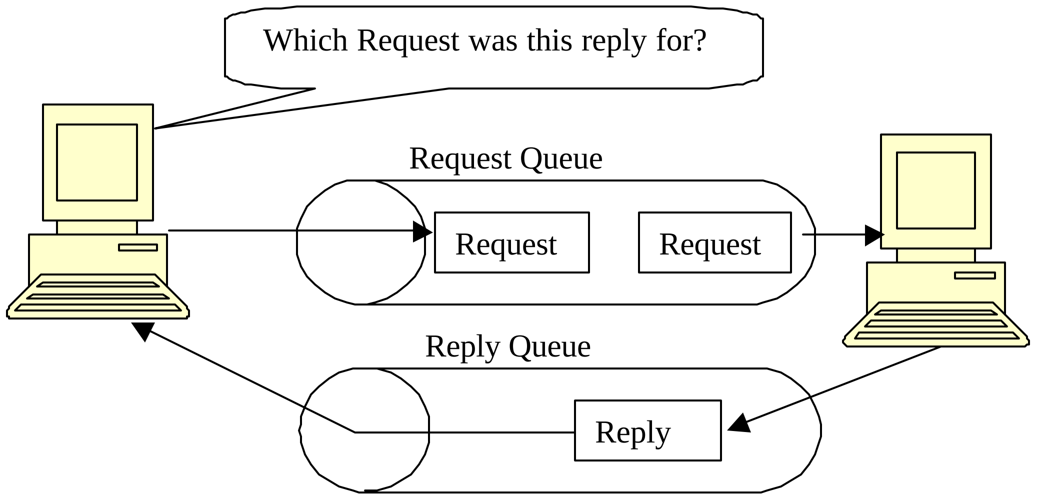


Figure 12

There are a couple of techniques the caller can use to avoid this confusion. It can make just one call at a time, waiting for a reply before sending another request. This will greatly slow the messaging process, however. The call could assume that it'll receive replies in the same order it sent requests, but messaging does not guarantee what order messages are delivered in and all requests may not take the same amount of time to process, so the caller's assumption would be faulty. The caller could design its requests such that they do not need replies, but this constraint would make messaging useless for many purposes.

What the caller needs is for the reply message to have a pointer or reference to the request message, but messages do not exist in a stable memory space such that they can be referenced by variables. However, a message could have some sort of foreign key, a unique identifier like a row in a relational database table. Such a unique identifier could be used to identify the message from other messages, clients that use the message, etc.

A full-blown unique identifier generator gets rather complex, making sure that identical identifiers are not generated simultaneously, guaranteeing that each identifier is unique forever and ever, etc. Reply messages can get by with something simpler. The caller just needs to ensure that of all the requests it has outstanding at any given time, each of them has a different identifier. This identifier may not be unique forever and ever, or for all contexts of all messages in all channels in the system, but just unique enough to identify which outstanding request this reply is for.

Therefore:

> **Each reply message should contain a *correlation identifier*, a unique identifier that indicates which request message this reply is for.**

This is how a correlation identifier works. When the caller creates a request message, it assigns the request an identifier that is different from those for all other currently outstanding requests (e.g., requests that do not yet have replies). When the receiver processes the request, it saves the identifier and adds the request's identifier to the reply. When the caller processes the reply, it uses the request identifier to know which request the reply is for. This is called a correlation identifier because of the way the caller uses the identifier to correlate (e.g., match; show the relationship) each reply to the request that caused it.

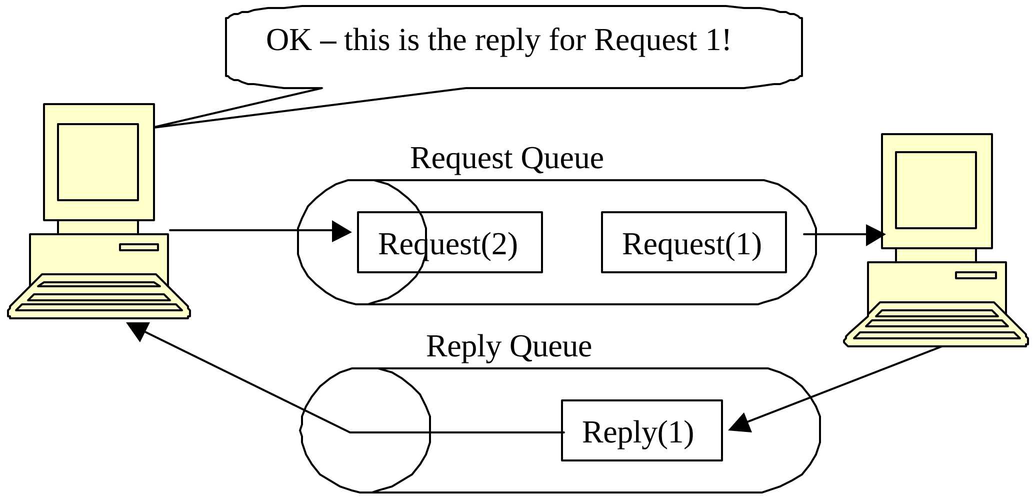


Figure 13

A correlation identifier is usually put in the header of a message rather than the body. The ID is not part of the command or data the caller is trying to communicate to the receiver. In fact, the receiver does not really use the ID at all; it just saves the ID from the request and adds it to the reply for the caller's benefit. Since the message body is the content being transmitted between the two systems, and the ID is not part of that, the ID goes in the header.

In practice, the application needs not only to relate a reply back to its request, but back to some business activity that caused the request in the first place. That business activity, such as needing to execute a stock trade or ship a purchase order, probably has its own unique business object identifier (such as a order ID), so that business activity's unique ID can be used as the request-reply correlation ID. Then when the receiver gets the reply and its correlation ID, it can bypass the request message and go straight to the business activity object that caused the request in the first place. In this case, rather than use the messages' built-in request message ID and reply correlation ID properties, you should use a custom business object ID in the request and the reply that identifies the business object this request-reply message pair represents.

In the request message, the ID can be stored as a correlation ID property or simply a message ID property. When used as a correlation ID, this can cause confusion as to which message is the request and which is the reply. So typically a request has a message ID but no correlation ID, then a reply has a correlation ID that is the same as the request's message ID. The reply may also have a message ID so that if it in turn has a reply to this reply, the second reply's correlation ID can be the same as the first reply's message ID. In this way, reply's can be chained indefinitely via a series of message ID's and correlation ID's.

JMS messages have a predefined property for correlation identifiers, `JMSCorrelationID`, which are typically used in conjunction with another predefined property, `JMSMessageID`. A reply message's correlation ID is set from the request's message ID like this:

```
Message requestMessage = // Get the request message
Message replyMessage = // Create the reply message
String requestID = requestMessage.getJMSMessageID();
replyMessage.setJMSCorrelationID(requestID);
```

Each `Message` in .NET has a `CorrelationId` member, a string in an acknowledgement message that is usually set to the `Id` of the original message. `MessageQueue` also has a special peek and receive methods, `PeekByCorrelationId(string)` and `ReceiveByCorrelationId(string)`, for peeking at and consuming the message on the queue (if any) with the specified correlation ID.

While a correlation identifier is used to match a reply message with its request, the request may also have a *Reply Specifier* that states what channel to put the reply on. Whereas a correlation identifier is used to match a reply message with its request, a *Message Sequence*'s identifiers are used to specify a message's position within a series of messages from the same sender.

Message Sequence

My application needs to send a huge amount of data to another process, more than may fit in a single message. Or my application has made a request whose reply contains too much data for a single *Reply Message*.

How can *Messaging* transmit an arbitrarily large amount of data?

It's nice to think that messages can be arbitrarily large, but there are practical limits to how much data a single message can hold. Some messaging implementations place an absolute limit on how big a message can be. Other implementations allow messages to get quite big, but large messages nevertheless hurt performance. Even if the messaging implementation allows large messages, the message producer or consumer may place a limit on the amount of data it can process at once. For example, many COBOL- and mainframe-based systems will only consume or produce data in 32 Kb chunks.

So how do you get around this? One approach is to limit your application to never need more data that what the messaging layer can handle. This is an arbitrary limit, though, which can prevent your application from producing the desired functionality. If the large amount of data is the result of a request, the caller could issue multiple requests, one for each result chunk, but that assumes the caller even knows how many result chunks will be needed. The receiver could listen for data chunks until there are not anymore (but how does it know there are not anymore?), then try to figure out how to reassemble the chunks into the original, large piece of data, but that would be error-prone.

Inspiration comes from the way a mail order company sometimes ships an order in multiple boxes. If there are three boxes, the shipper will mark them as "1 of 3," "2 of 3," and "3 of 3" so that the receiver will know which ones he's received and whether he has all of them. The trick is to apply the same technique to messaging.

Therefore:

Whenever a large set of data may need to be broken into message-size chunks, send the data as a *message sequence* and mark each message with three sequence identification fields.

The three sequence identification fields are:

1. *Sequence identifier* – Distinguishes this cluster of messages from others.
2. *Position identifier* – Uniquely identifies and sequentially orders each message in a sequence.
3. *Size* or *End field* – Specifies the number of messages in the cluster, or marks the last message in the cluster (whose position identifier then specifies the size of the cluster).

Let's say a set of data needs to be sent as a cluster of three messages. The sequence identifier of the three-message cluster will be some unique ID. The position identifier for each message will be different—either 1, 2, or 3. If the sender knows the total number of messages from the start, the sequence size for each message is 3. If the sender does not know the total number of messages until it runs out of data to send (e.g., the sender is streaming the data), each message will have a "sequence end" flag that is true for the final message in the sequence and false for all of the other messages. Either way, the position identifiers and sequence size/end field will give the receiver enough information to reassemble the parts back into the whole, even if the parts are not received in sequential order.

For example, let's say the user queries the application for all books by a certain author and there are ten matches. The messaging design might choose to return each match as a separate message. Then each message needs to indicate the query this reply is for, the message's position in the sequence, and how many messages total to expect.

Or imagine a sender needs to send an extremely large document to a receiver, so large that it must be broken into multiple messages. Again, each message needs to indicate its position in the sequence and indicate how many messages total to expect.

An application using JMS may wish to use a *Transactional Client* for sending and receiving sequences. The sender can send all of the messages in a sequence using a single transaction. This way, none of the messages will be delivered until all of them have been sent. Likewise, a receiver may wish to use a single transaction to receive the messages so that it does not truly consume any of the messages until it receives all of them. If any of the messages in the sequence are missing, the receiver can choose to rollback the transaction so that the messages can be consumed later.

Message sequence identifiers are often used in conjunction with *Correlation Identifiers,* but they solve different problems and can be used independently. If a reply is in multiple parts, each message will need the same correlation identifier to match the reply data with its request, but a different position identifier to indicate its position in the sequence. A single-message reply needs a correlation ID but no sequence ID. A multi-message document needs sequence ID's but no correlation ID.

Message sequences and *Competing Consumers* tend not to be compatible. If different receivers consume different messages in a sequence, none of the receivers will be able to reassemble the original data without exchanging message contents with each other. Thus a message sequence should be transmitted either via a *Publish-Subscribe* channel or a *Point-to-Point* channel with a single consumer.

Message Expiration

My application is using *Messaging*. If a message's data or request is not received by a certain time, it is useless and should be ignored.

> **How does a sender limit how much time can be used to transmit the message?**

Messaging virtually guarantees that the message will be delivered to the receiver eventually. What it cannot guarantee is how long the delivery will take. For example, if the network connecting the sender and receiver is down for a week, then it could take a week to deliver a message. Messaging is highly reliable, even when the participants (sender, network, and receiver) are not, but messages can take a very long time to transmit in unreliable circumstances.

Often, a message's contents have a practical limit for how long they're useful. A caller issuing a stock quote request probably looses interest if it does not receive an answer within a minute or so. The means the request should not take more than a minute to transmit, but also that the answer had better transmit back very quickly. A stock quote reply more than a few seconds old is probably too old and therefore unreliable.

Once the sender sends a message and does not get a reply, it has no way to cancel or recall the message. Likewise, a receiver could check when a message was sent and reject the message if it's too old, but different senders under different circumstances may have different ideas about how long is too long, so how does the receiver know which messages to reject?

What is needed is a way for the sender to specify the message's lifetime.

Therefore:

> **A message can contain a *message expiration* setting that specifies how long the message is viable.**

At the end of that time, the message will expire; an expired message is ignored and treated as if it where never sent in the first place.

A message expiration is a timestamp (date and time) that specifies how long the message will live or when it will expire. The setting can be specified in relative or absolute terms. An absolute setting specifies a date and time when the message will expire. A relative setting specifies how long the message should live before it expires; the messaging system will use the time when the message is sent to convert the relative setting into an absolute one.

The message expiration property has a related property, sent time, which specifies when the message was sent. A message's absolute expiration timestamp must be later than its sent timestamp (or else it will expire immediately). To avoid this problem, senders usually specify expiration times relatively, in which case the messaging system calculates the expiration timestamp by adding the relative timeout to the sent timestamp (expire = sent time + time to live).

JMS messages have a predefined property for message expiration, `JMSExpiration`, but a sender should not set it via `Message.setJMSExpiration(long)` because the JMS provider will override that setting when the message is sent. Rather, the sender should use its `MessageProducer` (`QueueSender` or `TopicPublisher`) to set the timeout for all messages it sends; the method for this setting is `MessageProducer.-setTimeToLive(long)`. Time-to-live is a relative setting specifying how long after the message is sent it should expire.

A .NET `Message` has two properties for specifying expiration: `TimeToBeReceived` and `TimeToReachQueue`. The *reach queue* setting specifies how long the message has to reach its destination queue, after which the message might sit in the queue indefinitely. The *be received* setting specifies how long the message has to be consumed by a receiver, which limits the total time for transmitting the message to its destination queue plus the amount of time the message can spend sitting on the destination queue. `TimeToBe-Received` is equivalent to JMS's `JMSExpiration` property. Both time settings have a value of type `System.TimeSpan`, a length of time.

When a message expires, the messaging system may simply discard it or may move it to a dead message queue (described in *Malformed Message Channel*). In *Publish-Subscribe* messaging, each subscriber gets its own copy of the message; some copies of a message may reach their subscribers successfully while other copies of the same message expire before their subscribers consume them. A *Reply Message* with an expiration may not work well—while the reply sender may wish to limit the message's lifetime, the reply receiver may become confused when it does not receive a reply because the message timed out. Thus the receiver has to be designed to handle the case where expected replies are never received.

4. Message Client Patterns

These patterns concern the behavior of messaging system clients. *Polling Consumer* describes a client that receives messages synchronously, whereas *Event-Driven Consumer* describes one that receives messages asynchronously. *Message Throttle* describes how a client can control the rate at which it accepts requests so as not to become overwhelmed. *Transactional Client* describes how a client can externally control the transactions used to send and receive messages. *Competing Consumers* describes a simple technique to process messages on a channel concurrently, whereas *Message Dispatcher* describes a more complex but more flexible technique for consuming messages concurrently.

Polling Consumer

A message consumer needs to consume messages so that it knows what to do, but needs to control when it consumes those messages.

How can a client explicitly control the rate at which it consumes messages?

Message consumers exit for one reason—to consume messages. The messages represent work that needs to be done, so the consumer needs to consume those messages and do the work.

But how does the consumer know when a new message is available? The easiest approach is for the consumer to repeatedly check the channel to see if a message is available. When a message is available, it consumes the message, and then goes back to checking for the next one. This process is called *polling*.

The beauty of polling is that the consumer can request the next message when it is ready for another message. So it consumes messages at the rate it wants to, rather than at the rate they arrive in the channel.

Therefore:

The client should use a *polling consumer*, one that explicitly makes a call when it wants to receive a message.

This is also known as a synchronous receiver, because the receiver thread blocks until a message is received.

A JMS `MessageConsumer` has three different receive methods:

1. `receive()` – Blocks until a message is available, then returns it.
2. `receiveNoWait()` – Checks once for a message, and returns it or null.
3. `receive(long)` – Blocks either until a message is available and returns it, or until the time-out expires and returns null.

A .NET `MessageQueue` client has several variations of receive. The two simplest are:

1. `Receive()` – Blocks until a message is available, then returns it.

2. `Receive(TimeSpan)` – Blocks either until a message is available and returns it, or until the time-out expires and throws `MessageQueueException`.

A consumer that is polling too much or blocking threads for too long can be redesigned as an *Event-Driven Consumer*. A set of polling consumers can be used to implement a *Message Throttle*.

Event-Driven Consumer

A message consumer needs to consume messages so that it knows what to do, but needs to control when it consumes those messages.

How can a client automatically consume messages as they become available?

The problem with *Polling Consumers* is that when the channel is empty, the consumer blocks threads and/or consumes process time polling for messages that are not there. This enables the client to control the rate of consumption, but wastes resources when there's nothing to consume.

Rather than continuously asking the channel if it has messages to consume, it would be better if the channel could tell the client when a message is available. For that matter, instead of making the consumer poll for the message to get the message, just give the message to the consumer as soon as the message becomes available.

Therefore:

The client should use an *event-driven consumer*, one that is automatically sent messages as they're added to the channel.

This is also known as an asynchronous receiver, because the receiver does not have a running thread until a callback thread delivers a message.

An event-driven consumer consists of two parts:

1. *Event Handler* – The code that detects the message-received event, gets the message, and hands it off to the performer.
2. *Performer* – The code that gets the message and processes it.

In JMS, the performer part of an event-driven consumer is a class that implements the `MessageListener` interface. This interface declares a single method, `onMessage(Message)`. The performer class implements `onMessage` to do whatever the consumer wants to do to process the message. Here is an example of a JMS performer:

```
public class MyMessageListener implements MessageListener {
  public void onMessage(Message message) {
     // Consume and process the message
  }
}
```

The event handler part of an event-driven client creates the desired performer object (which is a `MessageListener` instance) and associates it with the consumer:

```
Destination dest = // Get the destination
Session session = // Create the session
MessageConsumer consumer = session.createConsumer(dest);
MessageListener listener = new MyMessageListener();
consumer.setMessageListener(listener);
```

Now, when the destination receives a message, the JMS provider will call `MyMessageListener.onMessage` with the message as a parameter.

With .NET, the performer part of an event-driven client implements a method that is a `ReceiveCompletedEventHandler` delegate. This delegate method must accept two parameters: an `Object` that is the `MessageQueue`, and a `ReceiveCompletedEventArgs` that is the arguments from the `ReceiveCompleted` event. The method uses the arguments to get the message from the queue and process it. Here is an example of a .NET performer:

```
public static void MyReceiveCompleted(Object source,
  ReceiveCompletedEventArgs asyncResult)
{
  MessageQueue mq = (MessageQueue) source;
  Message m = mq.EndReceive(asyncResult.AsyncResult);
  // Consume and process the message
  mq.BeginReceive();
  return;
}
```

The event handler part of an event-driven client specifies that the queue should run the delegate method to handle a `ReceiveCompleted` event:

```
MessageQueue queue = // Get the queue
queue.ReceiveCompleted +=
  new ReceiveCompletedEventHandler(MyReceiveCompleted);
queue.BeginReceive();
```

Now, when the queue receives a message, it will issue a `ReceiveCompleted` event, which will run the `MyReceiveCompleted` method.

Event-driven consumers automatically consume messages as they become available. For more fine-grained control of the consumption rate, use a *Polling Consumer*. Event-driven consumers can be used to implement a *Message Throttle* if the total number of consumers is limited so as not to overwhelm the receiver application.

Message Throttle

When two applications are joined such that one calls the other, there is the possibility that the caller will make so many calls that they flood the receiver and cause it to crash.

How can a computer providing a service keep from becoming overwhelmed with an unreasonable number of requests?

When providing more users easier access to a legacy application, a very real concern is that the new users may well create far more load than the legacy application was designed to handle. Say a travel reservations app is designed to support a few hundred travel agents at a time, but a new web front-end enables anyone with a web browser to use this reservations system. If an airline announces a discounted fare, thousands of users may flood the website to buy tickets. The web front-end may be able to handle this load, but the legacy backend may not. If the web app forces too much load on the legacy app, the legacy app may crash.

The web app could try to control the load it places on the legacy app, but that can be difficult. The web app does not know how much load it's creating; what it can control is the number of simultaneous requests it makes. Yet with hundreds of threads making requests, even the web app may not know how many requests it's making. Even if the web app can control the load it causes, there could well be several other web apps also creating load, each thinking that it can use all of the legacy app's capacity.

What is needed is a way for the legacy app to control how many requests it tries to perform at once. If its load is low, it can take on more requests. If load is too high, it can slow the number of additional requests it accepts. Controlling load by adjusting to handle more or fewer requests is called *throttling*.

Therefore:

Use a *message throttle* to control the rate at which a receiver accepts requests.

This requires that the caller and receiver use *Messaging* to make requests. Then the receiver can control the rate at which it consumes request messages so that it does not try to handle too many requests at once. If the callers are sending request messages faster than the receiver is consuming them, the unconsumed messages will simply queue up while waiting to be consumed.

Messaging provides two ways to control message consumption: the receiver can be an *Event-Driven Consumer* or a *Polling Consumer*. Event-driven consumers will consume messages as fast as they become available. This conceivably could flood the legacy app, but probably will not because the messaging client only has so many threads and so many listener objects, so these will tend to limit the consumption rate. If event-driven consumption still might overwhelm the legacy app, the app can use a polling approach to more-explicitly control consumption. Either way, messaging's queuing allows communication to be paced so that the receiver will not be overwhelmed.

Throttling is typically used when consuming *Command Messages*, where the receiver cannot afford to loose a command but may become overwhelmed trying to process too many of them. Throttling can also be used when consuming *Document Messages,* again because they should not be lost but may require effort to consume.

Throttling is more typically used with *Point-to-Point* messaging, where the receiver needs to be sure not to loose the message. With *Publish-Subscribe* messaging, the messages are usually events, which can be easier to process than point-to-point command messages, and which a receiver may be able to ignore altogether.

Transactional Client

A messaging system, by necessity, uses transactional behavior internally. It may be useful for an external client to be able to control the scope of the transactions that impact its behavior.

How can a client control its transactions with the messaging system?

A messaging system must use transactions internally. A persistent message is not considered sent until the messaging system has persisted the message, so the send action and the internal persistence must be implemented as a single atomic operation.. The message that is sent must be persistent, so that if the system crashes and restarts, the persisted message will be sent correctly. Two *Competing Consumers* may attempt to read the same message; the messaging system must ensure that the read operation is atomic and that the consumers' two read operations are isolated so that only one of the consumers actually receives the message. When the messaging system moves a message from a physical copy of a queue on one computer to that on another, it uses a distributed transaction across the two computers to ensure that, at any given time, the message is in only one queue or the other, even if the messaging system crashes while moving the message. Thus the messaging system must always be in a consistent state, even if the system crashes. Transactions are a fundamental part of how a messaging system delivers messages reliably in unreliable circumstances.

A messaging client may not be aware of these internal transactions. All it knows is that the send and receive actions are atomic; a message is either sent or it is not, consumed or not consumed. Hiding transactions within calls like `send` and `receive` make messaging clients much simpler.

Yet hidden transactions are too simple for some messaging clients. A client may wish to explicitly control transaction boundaries.

Therefore:

Make the client's session with the messaging system transactional so that the client can specify transaction boundaries.

There are two reasons a client may wish to control its transactions with the messaging system:

1. *Batch message sends/receives* – The client wishes to send or receive several messages all at once, such that either all of them get sent/received, or none of them do. (Note that messages sent as a batch do not have to be received as a batch, and vice versa.)
2. *Coordinate transactions* – Sending/receiving a message may represent an action external to the messaging system, such as deleting a document being sent or consuming an event being received, and the external action needs to be coordinated with the message being manipulated such that either it all happens or none of it does. This often involves a distributed transaction, where transactions on two different data stores must be combined and coordinated using a two-phase commit.

Transactional clients using *Event-Driven Consumers* may not work as expected. The event handler typically must commit the transaction for receiving the message before passing the message to the performer. Then if the performer examines the message and decides it does not want to consume it, or if the performer encounters an error and wants to rollback the consume action, it cannot because it does not have access to the transaction. So an event-driven consumer tends to work the same whether or not its client is transactional.

In JMS, a client makes itself transactional when it creates its session.

```
Connection connection = // Get the connection
Session session =
  connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

Setting the first `createSession` parameter to true makes the session transactional.

When a client is using a transactional session, it must explicitly commit sends and receives to make them real.

```
Queue queue = // Get the queue
MessageConsumer consumer = session.createConsumer(queue);
Message message = consumer.receive();
```

At this point, the message has only been consumed in the consumer's transactional view. But to other consumers with their own transactional views, the message is still available.

```
session.commit();
```

Now, assuming that the commit message does not throw any exceptions, the consumer's transactional view becomes the message system's, which now considers the message consumed.

In .NET, each action on a queue (send, receive, etc.) is either transactional or not.

```
MessageQueue queue = // Get the queue
MessageQueueTransaction transaction =
  new MessageQueueTransaction();
transaction.Begin();
Message message = queue.Receive(transaction);
transaction.Commit();
```

Although the client had received the message, the messaging system did not make the message unavailable on the queue until the client committed the transaction successfully.

A client sending or receiving a *Message Sequence* may use a single transaction to ensure sending or receiving all of the messages in the sequence or none of them. A client executing a *Command Message* may wait until it executes the command and sends the *Reply Message* before committing the transaction that receives the command and sends the reply. (The requestor needs two transactions, however—one for sending the request and another for receiving the reply. If the requestor tries to send the request but will not commit the transaction until it receives the reply, it'll wait forever because the request does not really get sent until the transaction is committed, so there never will be a reply.) A client consuming a *Document Message* may wish to persist the document before committing to consuming the message and persisting the data. A client consuming an *Event Message* may wish to process the event before committing the receive action.

Competing Consumers

My application is using *Messaging*. However, it cannot process messages as fast as they're being added to the channel.

How can a messaging client process multiple messages concurrently?

Messages arrive through a channel sequentially, so the natural inclination of a consumer is to process them sequentially. However, sequential consumption may be too slow and messages may pile up on the channel, which makes the messaging system a bottleneck and hurts overall throughput of the application. This can happen either because of multiple senders on the channel or because each message takes significantly more effort to consume and perform than it does to send.

The application could use multiple channels, but one channel might become a bottleneck while another sits empty, and a sender would not know which one of equivalent channels to use. Multiple channels would have the advantage, however, of enabling multiple consumers (one per channel), processing messages concurrently. Even if this worked, though, the number of channels the application defined would still limit the throughput.

What is needed is a way for a channel to have multiple consumers.

Therefore:

> **Create multiple *competing consumers* on a single channel so that the consumers can process multiple messages concurrently.**

This solution only works with *Point-to-Point* channels; multiple consumers on a *Publish-Subscribe* channel just create more copies of each message. But with point-to-point, each consumer processes a different message concurrently, so the bottleneck becomes how quickly the channel can feed messages to the consumers instead of how long it takes a consumer to process a message. A limited number of consumers may still be a bottleneck, but increasing the number of consumers can alleviate that constraint as long as there are available computing resources.

For competing consumers to work properly, each consumer must have its own session and message consumer. When the consumers are running concurrently, they will each have their own thread, and the transactions in a session are not guaranteed to work correctly if two concurrent threads share a single session. [JMS02, pp. 26-27]

A sophisticated messaging system will detect competing consumers on a channel and internally provide a *Message Dispatcher* that ensures that each message is only delivered to a single consumer. This helps avoid conflicts that would arise if multiple consumers each thought they were the consumer of a single message.

Competing consumers may not work well with *Transactional Clients*. A messaging system with a good locking scheme should prevent multiple consumers from trying to simultaneously consume a single message, but there is no guarantee that a particular messaging system implementation will provide this behavior or that the behavior will be equivalent from one implementation to the next. Thus code dependent on this behavior may not be portable and should be thoroughly tested on each messaging system implementation.

The coordination of competing consumers depends on each messaging system's implementation. If the client wants to implement this coordination itself, it should use a *Message Dispatcher*.

Message Dispatcher

My application is using *Messaging*. However, it cannot process messages as fast as they're being added to the channel. I considered using *Competing Consumers* but that would not work well.

How can a messaging client process multiple messages concurrently?

Competing consumers is a simple solution to this problem, but its specific behavior is largely dependent on the internal implementation of the messaging system. It may not work well with transactional clients, and does not work at all with *Publish-Subscribe* channels. These circumstances require a single consumer.

Yet a single consumer can become a bottleneck to consuming messages. When there are a lot of messages to consume or each one takes a long time to perform, consumption needs to take a minimal amount of time and the messages need to be performed concurrently. The channel wants a single consumer but multiple consumers need to run concurrently.

Therefore:

Create a *message dispatcher* on a channel to consume messages and distribute them to performers.

A message dispatcher consists of two parts:

1. *Dispatcher* – The object that detects a newly available message, gets the message, and hands it off to a performer.
2. *Performer* – The object that gets the message and processes it.

A dispatcher acts as a one-to-many connection between a single channel and a group of performers. The performers do most of the work; the dispatcher just acts as a matchmaker, matching each message with an available performer. The dispatcher receives the message, and then sends it to a performer to process it. Because the dispatcher does relatively little work, it can dispatch messages as fast as the messaging system can feed them and thus avoids becoming a bottleneck.

A dispatcher makes the performers work much like *Event-Driven Consumers*, even though the dispatcher could be an event-driven consumer or a *Polling Consumer*. As such, implementing a dispatcher as part of a *Transactional Client* can be difficult. If the client is transactional, ideally the dispatcher should allow the performer to process a message before completing the transaction. Then, only if the performer is successful should the dispatcher commit the transaction. If the performer fails to process the message, the dispatcher should rollback the transaction. Since each performer may need to rollback its individual message, the dispatcher needs a session for each performer and must use that performer's session to receive the performer's message and complete its transaction. Since event-driven consumers often do not work well with transactional clients, the dispatcher should not be an event-driven consumer, but rather should be a polling consumer.

In JMS, it is helpful to implement the performer as a `MessageListener`. A message listener has one method, `onMessage(Message)`; it accepts a message and performs whatever processing necessary. This forms a clean separation between the dispatcher and the performer. Likewise, in .NET, the performer should be a `ReceiveCompletedEventHandler` delegate, even though the dispatcher will not really issue `ReceiveCompleted` events.

Message Selector

My application is using *Messaging* to transmit a large number of data types to a large number of receivers, which requires a very large number of channels.

How can I make one channel act like several separate channels?

We like to think of a messaging system as if it has an unlimited number of channels, but no computer resource is truly unlimited. At the very least, more hardware is required. With a messaging system, the messaging servers have to be deployed across a greater number of computers. One messaging system may have a hard limit to the number of channels it can support; to have more, multiple messaging systems have to be integrated to work as one, which is not so easy to do.

So channels are a relatively scarce resource that should not be wasted. *Data Type Channel* says that you need a channel per data type to be transmitted. When a sender is trying to communicate with a particular receiver or type of receiver, it needs a dedicated channel for this use. In general, when a channel has multiple receivers, any one of them has equal access to any message on the channel, even if this is not what the sender intended. What is needed is some way to mark a message to specify which receiver should consume it.

Therefore:

Use a *message selector* to specify a message's type and receive only messages of a particular type.

The sender sets a message's selector value, and then each receiver filters for a certain selector value. A receiver will only receive the messages whose selector matches the receiver's filter. A channel should have at least one receiver filtering for each valid selector value or the messages with that selector will be ignored, like a channel with no receivers.

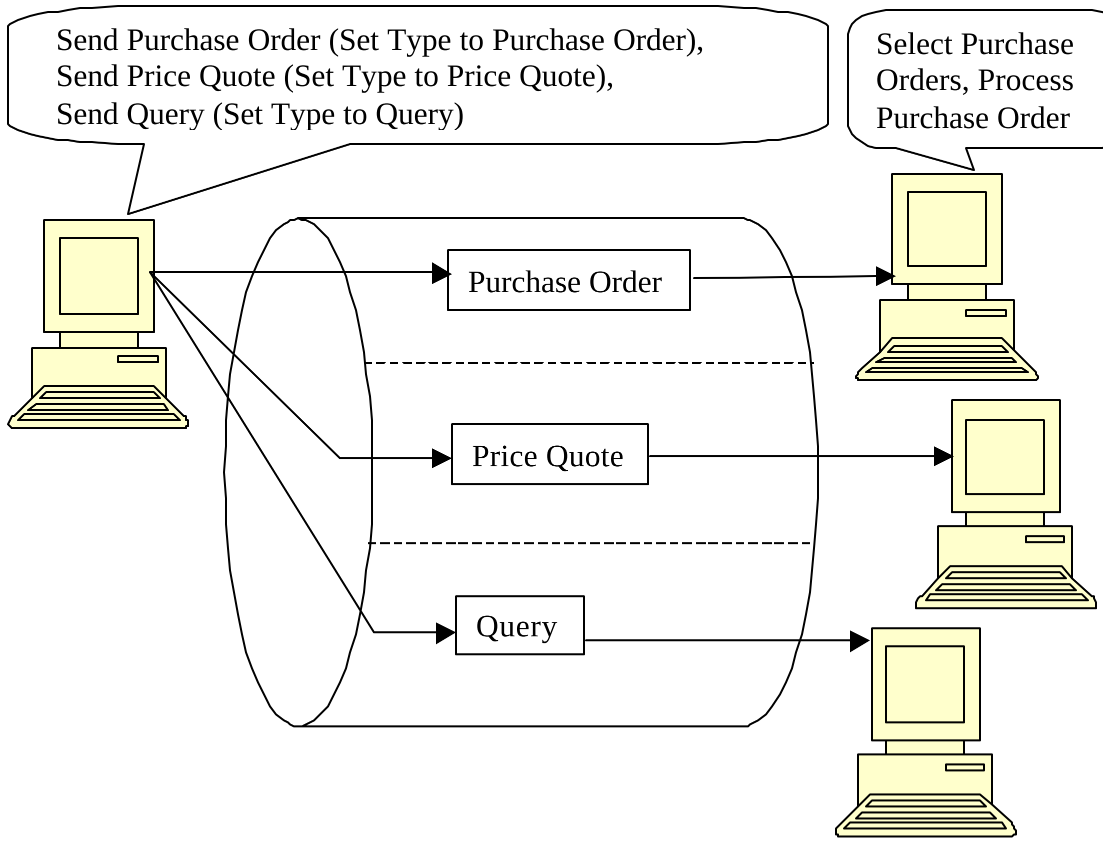

Send Purchase Order (Set Type to Purchase Order),
Send Price Quote (Set Type to Price Quote),
Send Query (Set Type to Query)
Select Purchase Orders, Process Purchase Order
Purchase Order
Price Quote
Query

Figure 14: Queue with Message Selectors

For example, a stock trading system with a limited number of channels might need to use one channel for both quotes and trades. The receiver for performing a quote is very different from that for trading, so the right receiver needs to be sure to consume the right message. So the sender would set the message selector on a quote message to `QUOTE`, and the quote receiver would only consume messages with `QUOTE` message selectors. Trade messages would have their own `TRADE` message selector that their senders and receivers would use. In this way, two message types can successfully share a single channel.

In JMS, a `MessageConsumer` (`QueueReceiver` or `TopicSubscriber`) can be created with a message selector string that filters messages based on their property values. First, a sender would set the value of a property in the message that the receiver could filter by:

```
TextMessage message = session.createTextMessage();
message.setText("<quote>SUNW</quote>");
message.setStringProperty("req_type", "quote");
Destination destination = //get the destination
MessageProducer producer = session.createProducer(destination);
producer.send(message);
```

Second, a receiver set its message selector to filter for that value:

```
Destination destination = //get the destination
String selector = "req_type = 'quote'";
MessageConsumer consumer =
  session.createConsumer(destination, selector);
```

This receiver will ignore all messages whose request type property is not set to “quote” as if those messages were never added to the destination at all.

In .NET, `MessageQueue.Receive` does not support message selectors per se. Rather, what a receiver can do is use `MessageQueue.Peek` to look at a message. If it meets the desired criteria, then it can use `MessageQueue.Receive` to read it from the queue.

Message selectors make a single channel act like multiple *Data Type Channels*.

5. Messaging Application Patterns

The following patterns are unique in that they are all really solve the same general problem, within the same architectural framework, but each relate to a specific part of that general problem. Our basic problem is this: messages form the core of our architecture; however, not all programs can handle messages (e.g. not all programs are built with a messaging system in mind) and even when a program is built to attach to a messaging system, it might not be capable of handling a specific message format that another program is capable of generating.

In this section of patterns, *Pipes and Filters* shows how you can architect a system that uses messaging to handle multiple sequential processing steps that can be carried out for different requests simultaneously. *Message Router* demonstrates how specific business logic can be used to make routing decisions and redirect processing among several potential steps. *Message Translator* and *Message Bridge* describe the ways in which systems can convert messages to other forms and transports to allow systems to communicate freely.

To facilitate the communication between disparate systems, a *Canonical Message Data Format* should be adopted. Whichever message format is chosen, it should exhibit *Data Format Flexibility* in order to make it clear how programs written at different times in a system lifecycle should communicate. Finally, *Message Bus* integrates several of the previous patterns into a joint architecture that solves several outstanding business problems.

Pipes and Filters Messaging

Computers often perform tasks as a series of steps, but this is more complicated when the tasks must be performed by different processes.

How can the steps of a process be performed when different steps must be performed on different machines?

Just about any task worth automating with computer software requires that a series of steps to be performed. At the line-of-code level, to average two numbers, the computer must first add them, and then divide by two. At the use case level, a document (such as a purchase order, insurance claim, etc.) must be received, processed, and then handed to the next processor.

A computer performs a task in a divide-and-conquer style, by splitting the task into a series of steps and performing one step at a time. This is so common that there's an architectural pattern for it, the Pipes and Filters pattern [BMRSS95]. A set of data is processed by a filter (a program that manipulates the data in some way), then passed through a pipe to another filter for more processing, and so forth for all of the steps in the task.

In its simplest form, a pipes-and-filters architecture runs all in one process with data in one memory space that's shared by all of the pipes-and-filter steps. With distributed computing, however, the various filters are often running in different processes. This distributes the processing and provides the opportunity for load balancing, fault tolerance, etc., but requires a more complex pipe to transmit the data from a filter in one process to another filter in a separate process. What is needed is a pipe designed to connect separate processes. As luck would have it, *Messaging* provides just such a pipe.

Therefore:

> **Use *pipes and filters messaging* to perform several steps in sequence even when different steps run in different processes.**

Each step becomes a filter that manipulates the data; the pipe is the messaging connection that transmits data from one step to the next.

For example, consider processing an insurance claim. The claim must be entered, approved, and then a check is issued. All three steps could be performed in one application running on one machine, but probably there's a specialized application for each step: A web page for customers to enter claims, a native GUI for claims adjusters to approve claims, and a program for writing and printing checks that will issue claims checks. Since these are three separate applications, they must be linked, and messaging is a good way to do so.

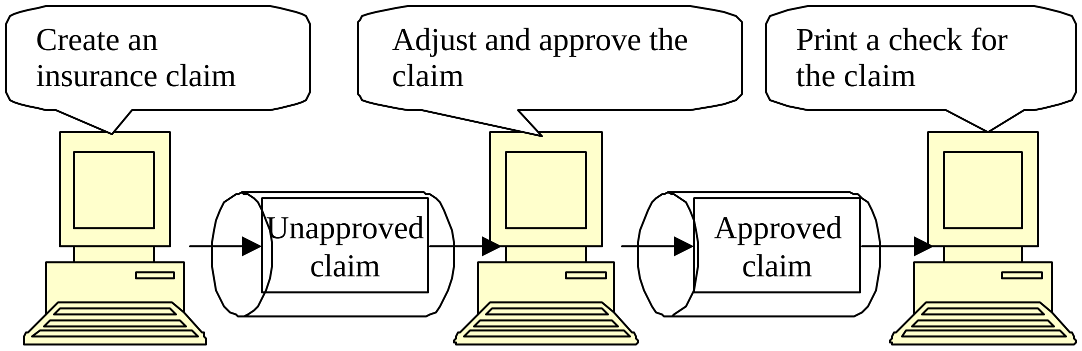

Create an insurance claim
Adjust and approve the claim
Print a check for the claim
Unapproved claim
Approved claim

Figure 15: Pipes and filter messaging for processing an insurance claim

More often than not, the messages that are carried on the pipes in a *Pipes and Filters Messaging* system are Document Messages. Thus, the pipes are most commonly *Point-to-Point* connections.

With a pipes and filters messaging design, data can be processed in several different ways. A *Message Router* can use various pipes to choose which filter to apply next. A *Message Translator* can convert the data format output by one filter into the data format expected by the next filter. If a filter cannot connect to a messaging pipe, use a *Message Bridge* to connect this filter to the messaging system.

Message Translator

My application implements a *Pipes and Filters Messaging* architecture, but different filters expect messages with different formats.

> **How can systems using different data formats work together in the same architecture?**

When two applications are built independently, they rarely agree on the same data format. Yet when we try to integrate them in an enterprise system, they need to agree on the data formats they're exchanging. For example, a company may develop an order processing application that uses a particular COBOL data format, and independently develop an order entry application that uses an XML data format. When the company decides to integrate these two applications together, which data format should the applications use?

The company integrating the applications may wish to avoid modifying either application, especially a change as fundamental as changing an application's data format. Changing an application's data format is risky, difficult, and will discourage integrating more applications in the future.

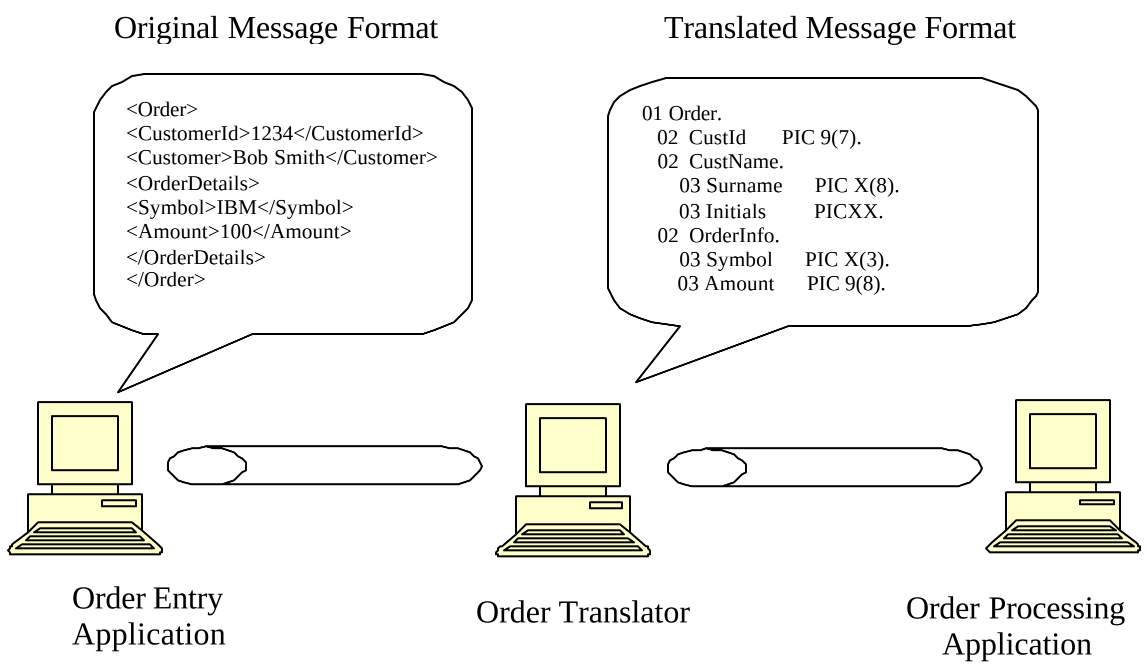
Messaging can be used to integrate the applications pipe-and-filter style, where each application acts like a big filter. But for two filters to pass data through a pipe, they have to agree on the data format being exchanged. What is needed is an intermediary application that will translate the data formats between the two existing applications.

Therefore:

> **Insert a special filter, a *message translator*, between two other filters to translate one's output format into the other's input format.**

A message translator is a specialized kind of filter that takes a message in one format and translates it to another format. There are a number of commercial products that do this, like Extricity and MQ-Si.

For example, the order entry application can send a message using its XML data format, the message goes to the translator which translates the data from the order entry format to the order processing format, and then the message goes to the order processing application in its COBOL format.



```
<Order>
<CustomerId>1234</CustomerId>
<Customer>Bob Smith</Customer>
<OrderDetails>
<Symbol>IBM</Symbol>
<Amount>100</Amount>
</OrderDetails>
</Order>
```

```
01 Order.
  02  CustId      PIC 9(7).
  02  CustName.
     03 Surname      PIC X(8).
     03 Initials     PICXX.
  02  OrderInfo.
     03 Symbol     PIC X(3).
     03 Amount     PIC 9(8).
```

Figure 16: Order Translation

In this way, the Message Translator acts as a special filter in *Pipes and Filters Messaging*. It protects the systems on each end of the pipe from knowing about the details of the format the other expects, but at the same time, each of the systems are isolated from the Message Translator itself – if one or the other of the systems became able to speak the native format of the other, then the Message Translator could be removed from the topology altogether without changing the functioning of the system at all.

Message translators are often used in converting from an internal data representation (specific to a particular company) to a standard, external data representation. For example, an insurance company may translate requests made using one of the standard ACORD XML message formats[1] into an internal message format.

Another common use of this pattern we see today is that a group of programs decide on a common XML message format (a *Canonical Message Data Model*), and then use a message translator to interface with other (perhaps older) programs that do not natively speak that XML format.

Canonical Message Data Model

My application implements a *Pipes and Filters Messaging* architecture, but all of the filters expect different data formats. This has lead to a huge number of *Message Translator* filters that require maintenance effort and runtime resources.

[1] ACORD is a standards organization for the insurance industry. Its XML standards can be downloaded at http://www.acord.org/xml_frame.htm.

How can the pipes-and-filters data be designed to minimize translation?

As described in *Message Translator*, independently developed applications tend to use different data formats. They can be integrated with messages translators, but this can lead to an explosion of translators that become as difficult to maintain and run as the applications themselves. Eventually there comes a point where the translators are too complex and too numerous, and the applications need to be modified to work together better without translators.

The reason independently developed applications tend to use different data formats is that each format was designed with just one application in mind. What is needed is a format designed with all applications in mind, so that it will work with any application. Such a unified format is needed not just for a single data type, but for all data types that may be exchanged between applications. Once each application is modified to use these data formats, then all of the applications will be able to exchange data. As new applications are added, if they expect the unified data formats, then they too can be integrated without translators.

Therefore:

Develop a *canonical message data model* that all filters use so that no translation is necessary.

A canonical message data model is a unified data model for data types that will be passed through messaging. Designers should strive to make the unified model work equally well for all applications being integrated. Applications designed to produce and consume data conforming to the unified model do not require translators.

A canonical message data model is actually made up of two layered protocols (a) what data is sent between the parties and (b) how that data is represented. These two protocol layers are termed the *application layer* and *presentation layer* in the OSI reference model. There are a large number of presentation layer protocols to choose from, such as XDR, ASN.1, CDR, etc. A presentation layer is simply a common way of encoding or representing your data. For instance, you may say that your data is encoded in XML – this means that the data will follow the rules of the XML standard (purely text, with elements enclosed in tags that represent the structure of the data).

While, today the basis for a presentation layer protocol is often XML, but both ASN.1 and key-value pairs are also popular and useful. In this way all messages have the same top-level format, and each system translates from a well-known, canonical model to its own internal format on message receipt and vice versa before it sends a message.

However, even with a common presentation layer protocol you have to define common semantics too. For instance, you have to get all parties to agree on the definitions of all of your terms. This is the *application layer protocol* part of the reference model. This means that each of the systems in your application will need to agree on a common way to (for instance) identify Customers so that a Customer identifier on one system refers to the same Customer on another system and so on. In XML, DTD (Document Type Definition) and XML Schema are two different ways of specifying different application layer protocols for specific uses. So, within an organization, you may specify both that all data will be sent using XML (a presentation layer protocol choice) and using a specific XML Schema (an application layer choice).

Adopting a Canonical Message Format leads to requiring Data Format Flexibility. Usually, when adopting a canonical message data format, you will require *Message Translators* to convert from the canonical format to existing message formats, at least for some length of time while both formats must co-exist.

Data Format Flexibility

My application implements a *Pipes and Filters Messaging* architecture, perhaps using a *Canonical Message Data Model*, but realistically the message formats may need to change over time.

> **How can a message's data format be designed to allow for possible future changes?**

Let's take a worst-case scenario. Say you have a binary format and that in version 1.0 your first field in 32 bits. In version 2.0 that field is now 64 bits. The problem is that version 2.0 programs will try to read not only the first field, but part of the second field...chaos ensues. So a fixed binary format is not flexible enough to meet our needs.

Therefore:

> **Design a data format that includes meta-data to describe the structure of the data, a structure that may change over time.**

There are at least three template solutions to this problem.

1. Version Numbers – include a "version number" as the first field of the structure. Programs know immediately what they're dealing with and whether or not they can handle it.
2. “Self-describing” data. If your data is “key-value” or XML then your programs can more easily tolerate certain types of changes to the data structure since position does not matter (as much...). However, even XML documents often refer to a specific DTD – if the difference is too great, then a replacement DTD can declare its compatibility using a version numbering scheme like the one described above.
3. “For expansion” or “unused” fields. In positional (binary) forms you keep some space for future expansion, which older programs ignore and newer programs can use.

Which one you choose depends to a large extent upon the structure of the rest of your system. In most cases, a key-value or XML format will provide the most flexibility. However, many older systems (mostly those written in COBOL or C) may not be able to take advantage of an XML format as they do not have easily available XML parsers and generators for them. So, if you choose to use an XML format for flexibility, and you have a system like this, you must use a Message Translator to convert between the more flexible format and one of the other two binary cases.

Both Binary and XML formats can take advantage of the version number scheme. In an XML format, the version number is really carried through the internal reference to the DTD or Schema that the XML document was written to. If a program cannot process a particular version of a DTD or Schema, it can reject the document outright. On the other hand, in a binary format, literal version numbers are critical to understanding the structure of the remainder of the document. Since the size of a binary structure should often remain fixed (since older programs could suffer buffer overflow if a larger structure is passed to them than they expect) the “expansion” fields solution is often used in this situation. However, even in that situation, a version number must be used so the receiving program will know how to interpret the “expansion” fields by referring to a set of metadata that is keyed by version number indicating the meaning of each field.

Message Router

> **My application implements a *Pipes and Filters Messaging* architecture, but every filter may need to send data to any other filter, which will cause an explosion in the number of pipes needed. How can any filter send data to any other filter without the need for a pipe between every single pair of filters?**

For one of *N* filters to talk to every other filter requires *N – 1* pipes. Since message pipes are one way, not only does filter *A* need a pipe to send data to filter *B*, but then filter *B* needs another pipe to send data to filter *A*.

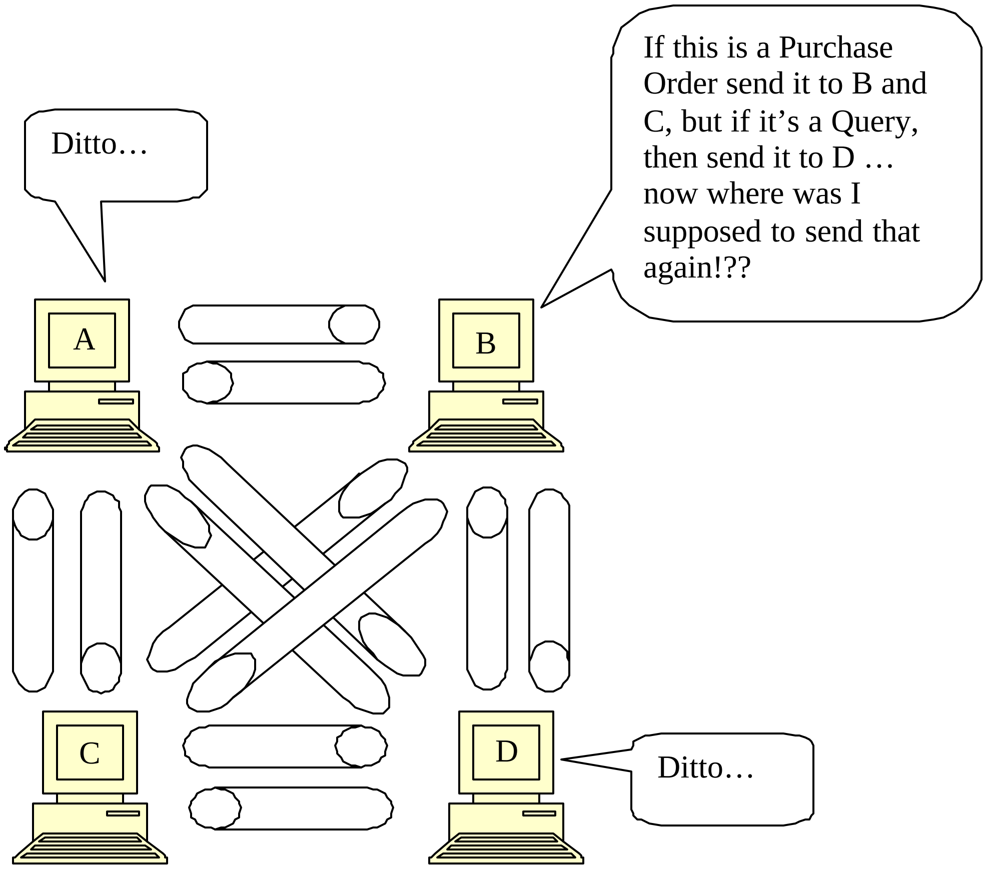

Ditto…
If this is a Purchase Order send it to B and C, but if it’s a Query, then send it to D … now where was I supposed to send that again!??
A
B
C
D
Ditto…

Figure 17

Publish-Subscribe may be a considered as a solution to this problem in that it reduces the total number of channels involved, but the semantics of publish-subscribe may not be appropriate for all cases; We may have the case where each system is buried under an avalanche of messages that it may not care about (necessitating *Message Selectors*), or we may end up in a case where only one of several similar systems may need to process a particular message – a complex problem to try address with *Publish-Subscribe*; one that would require very close coordination of the receiving systems.

As noted in *Message Selector*, no computing resource is truly unlimited, including message channels.

Therefore:

> **A filter should send its data to a central *message router* that will send the data to the next filter.**

A message router will read from one queue and then place its output on N (where N could be 1 or a number greater than 1) other queues based on information held inside each particular message's body or its headers.

The primary decision here is where the routing portion of your business logic lives. In a topology without a message router, often systems must duplicate routing logic in each of the potential senders and receivers of messages. A Message Router gathers together that routing logic into a single point, making it easier to update the routing when necessary.

This pattern is a realization of the pipes and filters pattern. A message router is a specialized kind of filter that takes input from one pipe and directs its output to 1 to N other pipes. Most Messaging router systems are rule-based, meaning that they can make simple decisions about which destinations a message will go to based on the content of the incoming messages.

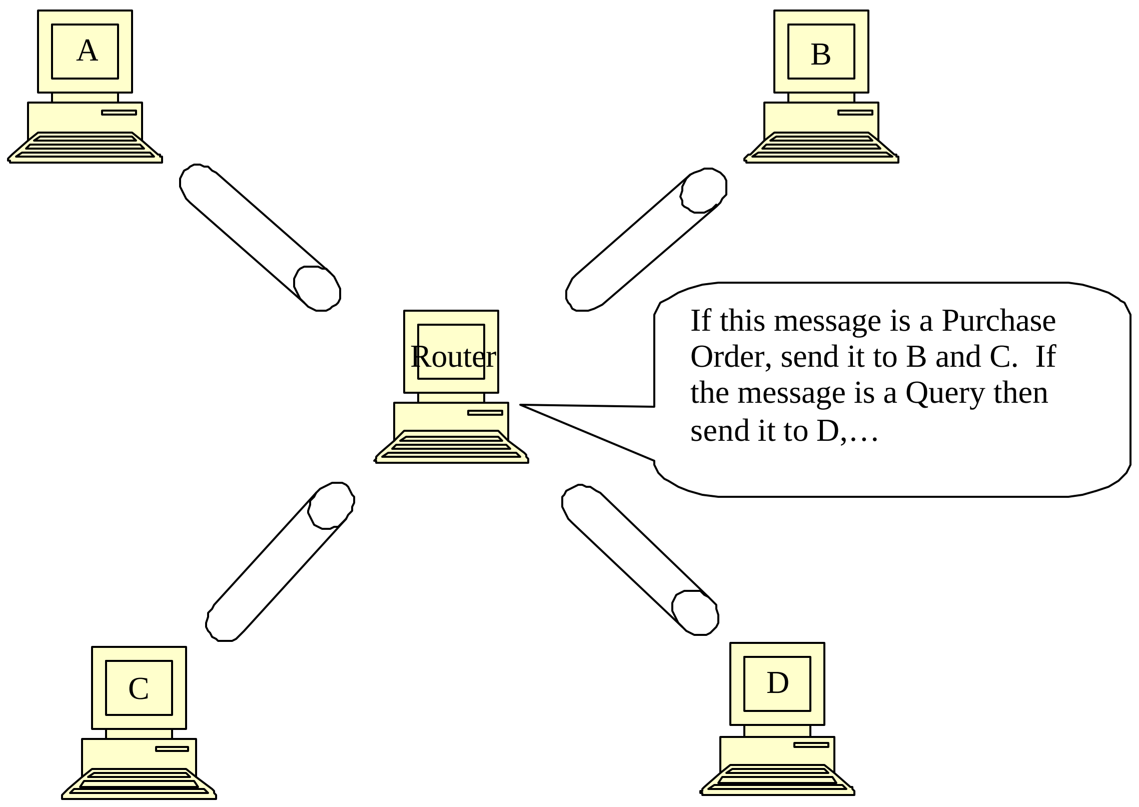

A
B
Router
If this message is a Purchase Order, send it to B and C. If the message is a Query then send it to D,…
C
D

Figure 18

As an example, IBM's MQSeries MQ Systems Integrator (MQSi) product acts as a message router [MQSi]. The following diagram shows the structure of the MQSi product. This is instructive because the basic steps used by the MQSi product (remove a message from a Queue or Topic, evaluate it against some set of rules, and then based on the decisions, place it on one or more other Queues or Topics) are the same that any Message Router will take.

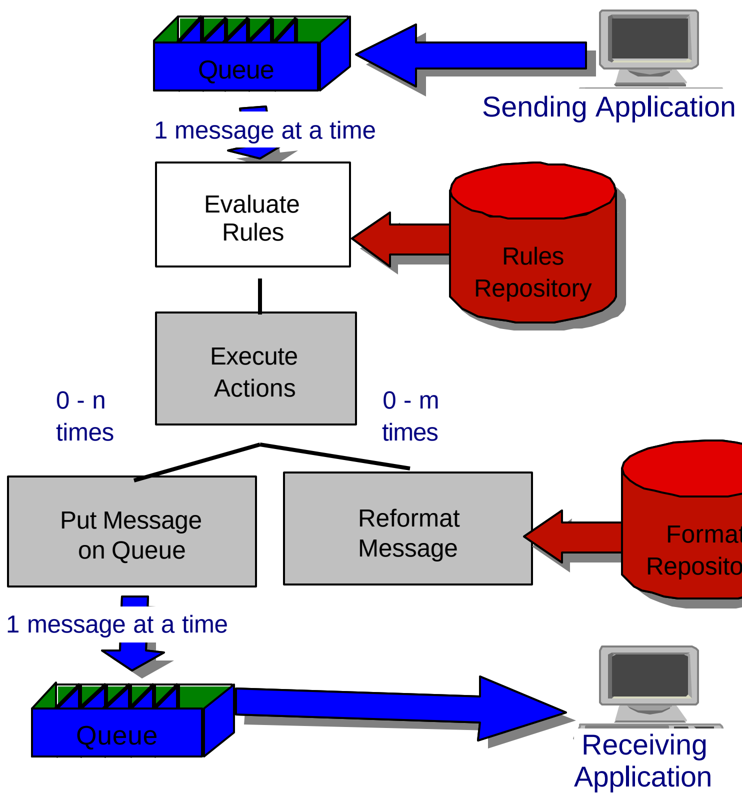


Figure 19: MQSi system structure

Of course, you do not need the flexibility of a product like MQSi in all cases. In many cases an individual Java program can act as a Message Router, invoking business logic triggered by the receipt of a JMS message, and then placing the output on one or more appropriate queues.

Message Bridge

My application implements a *Pipes and Filters Messaging* architecture, but some of the filters cannot use a messaging client and thus cannot put messages on a pipe implemented using a messaging system.

How can a process without a messaging client participate in messaging?

In many cases, existing systems are only capable of communicating via a simple protocol like HTTP or TCP/IP Sockets due to firewall restrictions or other problems like the unavailability of a messaging client for a particular programming language or operating system.

Another common case is where a company uses two different messaging systems; for instance IBM's MQ Series and Sonic Software's SonicMQ. The JMS standard does not mandate interoperability between JMS clients and servers; therefore you cannot (for instance) easily transfer a message from a Queue on MQ Series to another Queue on SonicMQ. So, as a result, in most EAI (Enterprise Application Integration) scenarios there are a set of cases where information cannot reach its intended destination because of protocol mismatches.

Therefore:

> **Use a *message bridge*, a gateway into the messaging system, to connect a non-messaging client to a messaging system.**

For instance, a common case is to build a bridge that accepts messages through HTTP and then sends them on to an external system through JMS (using a MOM like MQ Series) and vice versa.

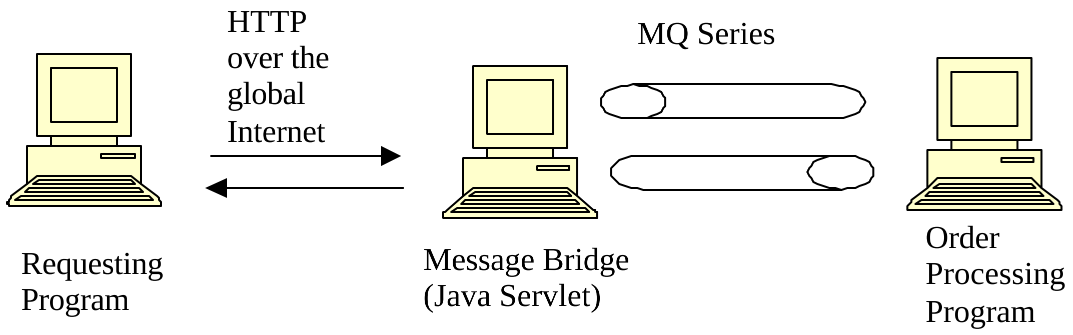


Figure 20: Message Bridge

Again, here the Message Bridge (the Java servlet in our example) is really just acting as a specific type of filter in Pipes and Filters Messaging. This sort of Gateway for protocols will commonly be used to allow connections from clients that cannot use the underlying MOM protocol for one reason or another. Often, MOM vendors include such gateways as a value-add to their products. For instance, WebSphere Application Server 5.0 contains a product called the “Web Services Gateway” that transparently bridges SOAP messages from HTTP to MQ Series in this way. This allows the SOAP messages to be conveyed over an intranet using the native MQ Series protocols, while messages can be carried to the global internet (outside the firewall) over HTTP. Another example of a message bridge of this sort can be found in the Apache SOAP engine, which includes a bridge for SOAP messages from SMTP to HTTP.

There are many concerns that must be dealt with in building a Message Bridge. Obviously, when you leave the protocol and software of a MOM behind, you leave the Qualities of Service that a MOM provides behind as well. You must also consider that different protocols have different properties – for instance, in the example above there is no easy way to “push” messages to the client system over HTTP unless that system listens over HTTP (e.g. contains a web server) as well. In many cases, this reduction in QoS is acceptable, but it is not a decision that should be taken lightly.

Message Bus

My enterprise contains several existing systems that must be able to share data and operate this shared data in response to a set of common business requests.

> **Is there an architecture that enables separate applications to work together, but in a decoupled fashion such that applications can be easily added or removed without affecting the others?**

When you are architecting a solution that must perform Enterprise Application Integration (EAI) you are often faced with the problem that you have multiple disparate systems, developed in several different languages and environments, which must communicate with each other in a controlled and orderly fashion. For instance, consider the following example: Let's say that you're an Insurance Company that sells different kinds of insurance products (life, health, auto, home, etc.). As a result of various corporate mergers and acquisitions, and the varying winds of change in IT development you are stuck with the following problem:

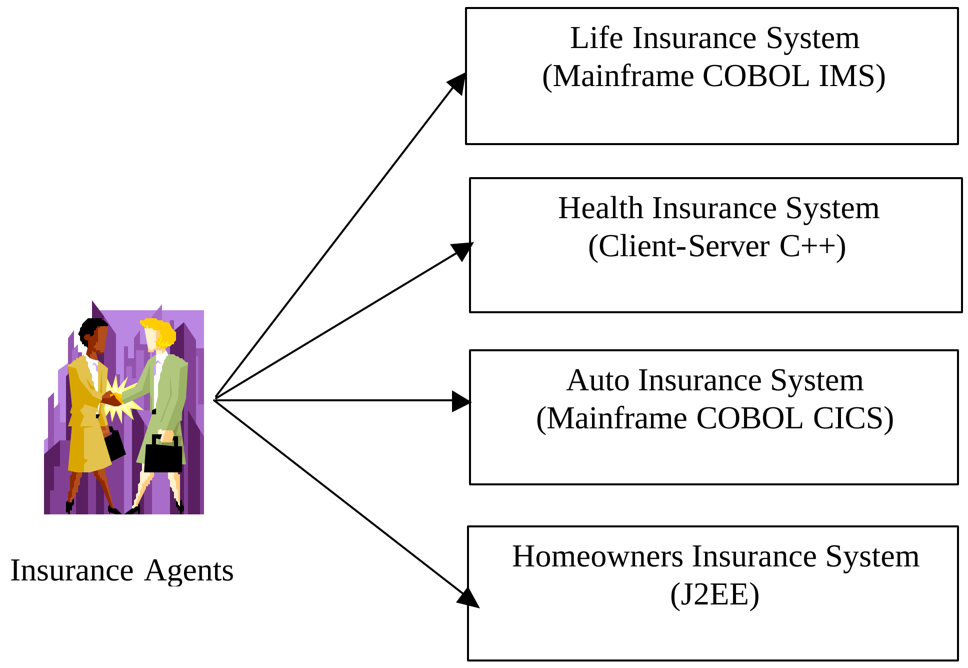


Figure 21: EAI Scenario

In this scenario, the different insurance products are all handled by different front-end systems, with varying business rules. However, what if an insurance agent wants to accomplish a simple task like changing the Home Address of a Client that has policies of all four types? In the worst-case scenario, the agent would have to use four different systems (with four different interfaces) in order to perform this update – a lengthy and time-consuming process that takes them away from their primary job function. What's more, the chances for error increase as the number of systems the agent must use increases.

So, what the agent would like to do is to be able to make this change once, and have the change propagate to all of the systems that need to be updated. However, in many cases, it's not feasible to scrap the existing systems and rewrite them with a single interface (due to the cost involved), nor is it feasible to "skip" the intervening system layers and go directly to the underlying databases where the data is stored (because of the business logic "edits" that run on the data before it is inserted or updated into the database). Writing a single program (such as a set of Java Servlets or EJBs that would use JCA) that acts as a front-end to all of the systems involved is often not feasible either, since managing the different interaction patterns of all of the systems involved would probably cause the front-end system to collapse under its own weight. So, what is needed is a way to use the existing systems, while still keeping them as independent components.

Therefore:

> **Structure the connecting middleware between these applications as a *message bus* that allows them all to work together strictly using messaging.**

A Message Bus is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a "least common denominator" set of interfaces.

The analogy here is to a communications bus in a computer system, which serves as the focal point for communication between the CPU, main memory, and peripherals. Just as in the hardware analogy, there are a number of pieces that come together to form the message bus:

? *Common communication infrastructure* – Just as the physical pins and wires of a PCI bus provide a common, well-known physical infrastructure for a PC, a common infrastructure must serve the same purpose in a message bus. Commonly, a MOM like MQ Series is chosen to serve as the physical communications infrastructure since they are cross-platform, cross-language, and available for most systems. The infrastructure may include *Message Router* capabilities to facilitate the correct routing of messages from system to system. Another common option is to use *Publish-Subscribe* to facilitate sending messages to all receivers.

? *Adapters* – The different systems must find a way to interface with the message bus. Most commonly, this is done with commercial or custom *Message Translators* that can handle things like invoking CICS transactions with the proper parameters, or representing the general data structures flowing on the bus in the specific and particular way they should be represented inside each system. This also requires a *Canonical Data Format* that all systems can agree on.

? *Common Command Structure* – Just like PC architectures have a common set of commands to represent the different operations possible on the physical bus (read bytes from an address, write bytes to an address), there need to be common commands that are understood by all the participants in the Messaging Bus. The *Command Message* pattern illustrates how this feature works. Another common implementation for this is the *Data Type Channel*, where a *Message Router* makes an explicit decision as to how to route particular messages (like Purchase Orders) to particular endpoints.

It is at the end that the analogy breaks down, since the level of the messages carried on the bus are much more fine-grained than the "read/write" kinds of messages carried on a physical bus.

In our EAI example, for instance, one instance of a command might be "Update an Address". The following diagram illustrates how this would work:

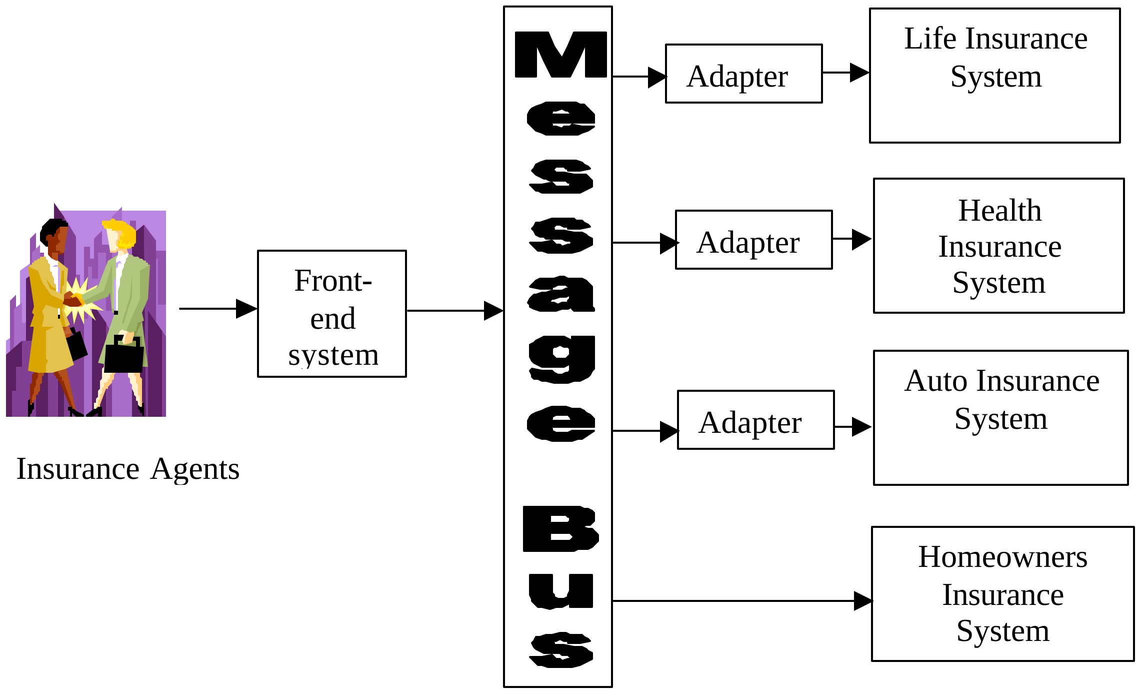


Figure 22: Message Bus

Here we have a single new GUI that only knows about the Message Bus – it is entirely unaware of the complexities of idiosyncrasies of the underlying systems. The Bus is responsible for routing Command messages to the proper underlying systems. In some cases, the best way to handle the Command messages is to build an Adapter to the system that interprets the Command and then communicates with the system in a way it understands (invoking a CICS transaction, for instance, or calling a C++ API). In other cases, it may be possible to build the Command-processing logic directly into the existing system as an additional way to invoke current logic.

6. Conclusions

In this pattern language, we've "opened a window" (so to speak) on the world of enterprise messaging. We've examined why enterprise messaging is useful, how it helps developers build systems that communicate reliably and asynchronously, and we've examined some architectural styles for building systems using enterprise messaging. However, we've barely scratched the surface of this topic – many more patterns exist in this space, and our journey to discover them has only begun.

7. Bibliography

[BMRSS95] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern Oriented System Architecture: A System of Patterns*, John Wiley & Sons, 1995.

[GHJV95] Erich Gamma, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley, 1995.

[Graham] Steve Graham, Simon Simeonov, Toufic Boubez, Glen Daniels, Doug Davis, Yuichi Nakamura, Ryo Nyeama, *Building Web Services with Java: Making Sense of XML, SOAP and UDDI*, SAMS Publishing, 2002.

[IBM00] IBM, *MQSeries Application Programming Guide*, Thirteenth edition (November 2000); ftp://ftp.software.ibm.com/software/ts/mqseries/-library/books/csqzal05.pdf.

[JMS-1] The Java Message Service API (JMS); http://java.sun.com/products/jms.

[JMS-2] JMS licensees; http://java.sun.com/products/jms/licensees.html.

[JMS02] *Java Message Service* (the Sun Java Message Service (JMS) 1.1 Specification); http://java.sun.com/products/jms/docs.html.

[MHC01] Richard Monson-Haefel and David A. Chappell, *Java Message Service*, O'Reilly, 2001.

[MQS-1] MQSeries from IBM; http://www.ibm.com/software/mqseries.

[MQS-3] MQSeries and Java; http://www.ibm.com/software/mqseries/api/mqjava.html.

[MQSi] *IBM MQSeries Integrator, Version 2 Technical Whitepaper*; ftp://ftp.software.ibm.com/software/ts/mqseries/library/whitepapers/mqsiv2twp.pdf.

[MSMQ-1] Microsoft Message Queuing (MSMQ); http://www.microsoft.com/msmq.

[MSMQ-2] Microsoft, "MSMQ Start Page," Platform SDK Release: November 2001; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msmq/msmq_overview_4ilh.asp.

[MSMQ-3] Microsoft, *Message Queuing in Windows XP: New Features*, 2001; http://www.microsoft.com/msmq/MSMQ3.0_whitepaper_draft.doc.

[NET-1] System.Messaging namespace in .NET; http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemMessaging.asp.

[SOAP-1] W3C Simple Object Acccess Protocol (SOAP) 1.1 Specification; http://www.w3.org/TR/SOAP/.