Software Architecture Design II
Distributed System Architecture

Matthew Dailey

Computer Science and Information Management
Asian Institute of Technology

Readings

Readings for these lecture notes:

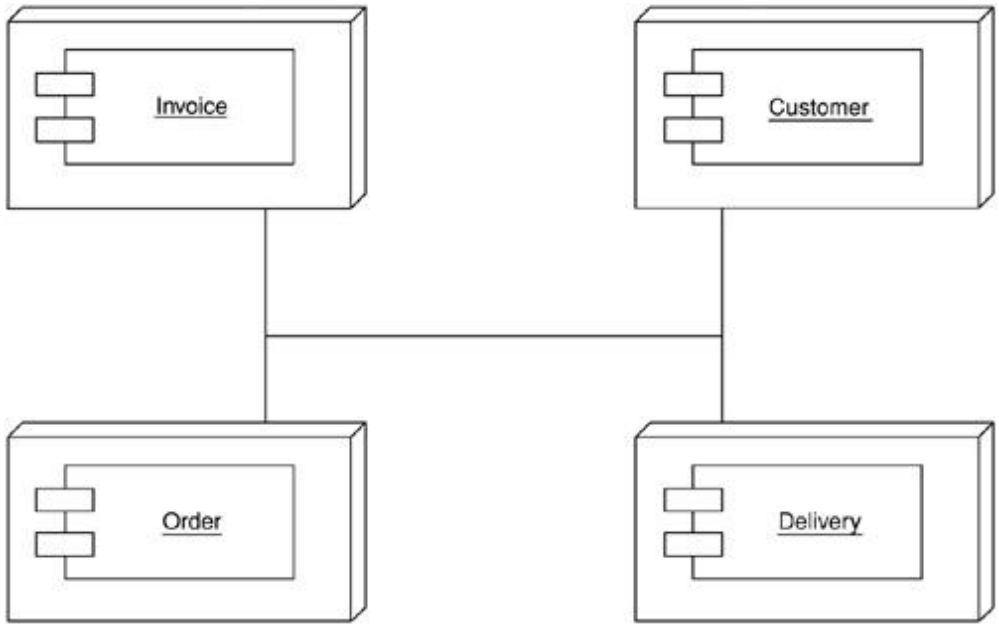
- Fowler (2002), *Patterns of Enterprise Application Architecture*, Addison-Wesley.
- Hohpe and Woolf (2004), *Enterprise Integration Patterns*, Addison-Wesley.

Outline

Distribution strategies

First law of distributed objects

Fowler's First Law of Distributed Objects: don't distribute objects! Here is an example bad design:



Fowler (2002), Fig. 7.1

Why? Because the distribution is based on domain classes, and performance is going to be terrible.

Distribution strategies

Local vs. remote interfaces

Some rules of thumb:

- A procedure call is fast.
- A procedure call between two separate processes is orders of magnitude slower.
- A call between separate processes on separate machines is another order of magnitude slower.

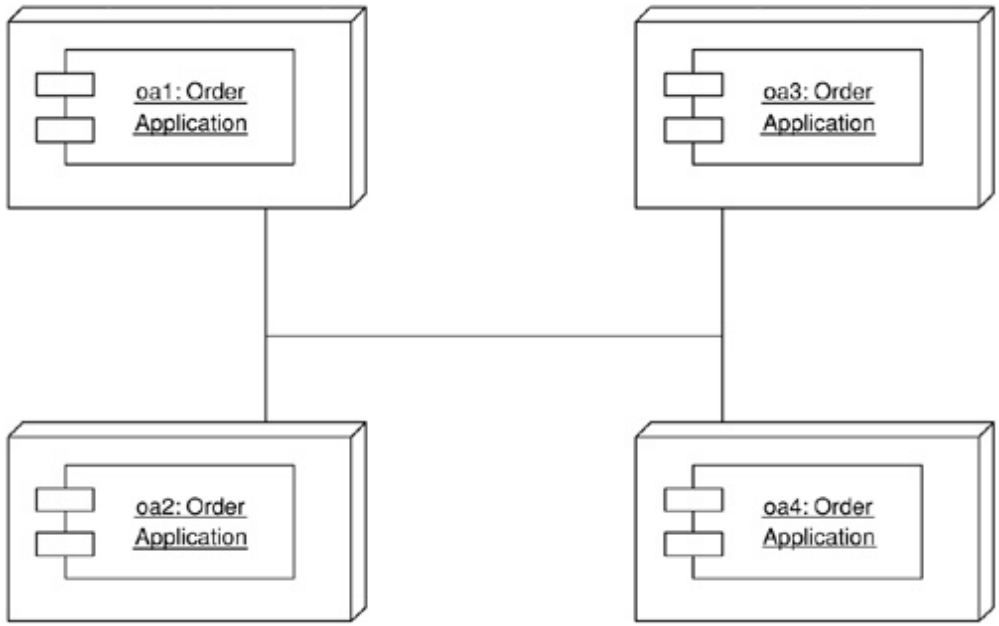
Local interfaces are best when they are fine-grained. This is very consistent with the OO philosophy where we have many small objects responsible for their own small operations.

When we take an interface and make it remote, fine-grainedness is not good. Rather than 3 calls to fetch three attributes of an object, we'd better get all three attributes in one call, to avoid tripling the network overhead.

Distribution strategies

Coarse-grained interfaces and clustering

So any object that can be invoked over the network must have a coarse-grained interface. Any object with a fine-grained interface should never be invoked over the network.



Fowler (2002), Fig. 7.2

Here is a better design based on clustering. We can do fine-grained interfaces between domain classes which is what we like, at the small cost of having the entire application replicated at each node.

Distribution strategies

When must you distribute?

Clustering can only take you so far. We do have to separate processes sometimes. Here are the common reasons:

- Clients and servers: it's impractical to replicate the entire shared data store across every PC in the enterprise. If we don't want to use mainframes, we must have clients connecting to the data store over a network.
- Application server/Database server: not absolutely required but typical. For sure in an enterprise system the DB will be a separate process, and that gives most of the performance gap. Luckily SQL is coarse-grained by design, to allow remote access.

Distribution strategies

When must you distribute?

Other times when distribution may be necessary:

- Web server/application server: try to run the application in the same process as the server, but sometimes it's not possible.
- Vendor differences: any third-party application you're using will normally run in its own process. This will normally be a coarse-grained interface, though, so it should be OK.
- Split application server software: avoid this at all costs! If it cannot be avoided, you have to divide your software into coarse-grained components with remote interfaces.

Distribution strategies

The distribution boundary

Some rules of thumb for the boundary between two components with a remote interface:

- At the boundaries, use coarse-grained objects that, if necessary, have a lot of attributes and operations doing a lot.
- Inside each component, stick with normal fine-grained objects.
- The coarse-grained boundary object should simply act as a facade for the fine-grained objects inside.
- Clients that know they're making local calls to your component can use the fine-grained interface directly.

The Remote Facade pattern formalizes this approach.

Data Transfer Objects (DTOs) go hand-in-hand with Remote Facades. They bundle together all the data from the relevant domain objects that need to be transferred over the remote interface.

What kind of interface to use for DTO transfer, RPC, and asynchronous messaging?

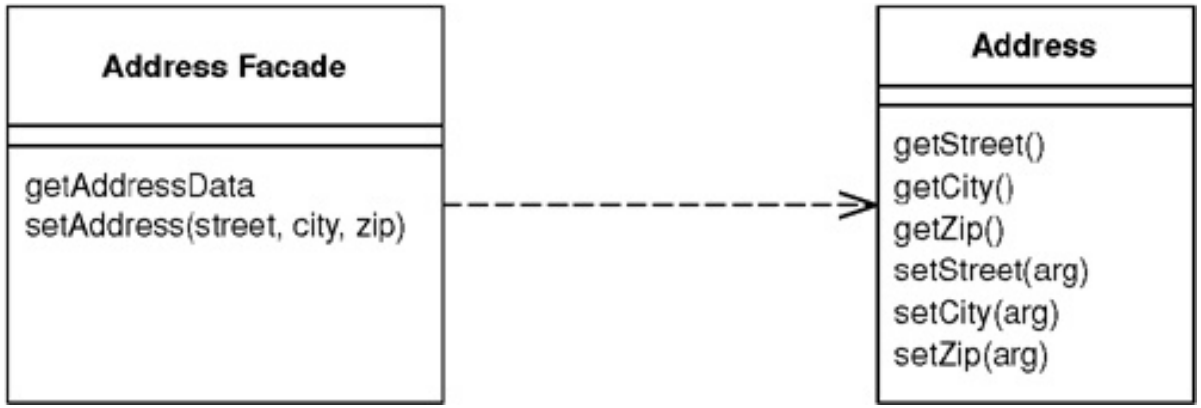
- When interoperability is paramount, use XML over HTTP (SOAP/WSDL). It is currently the most flexible method we have.
- When interoperability is not a concern, use native interfaces such as Java RMI and JMS.

Don't get stuck with synchronous remote calls. For the sake of performance, wherever possible, use asynchronous calls.

Distribution strategies

Remote Facade

A Remote Facade provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.

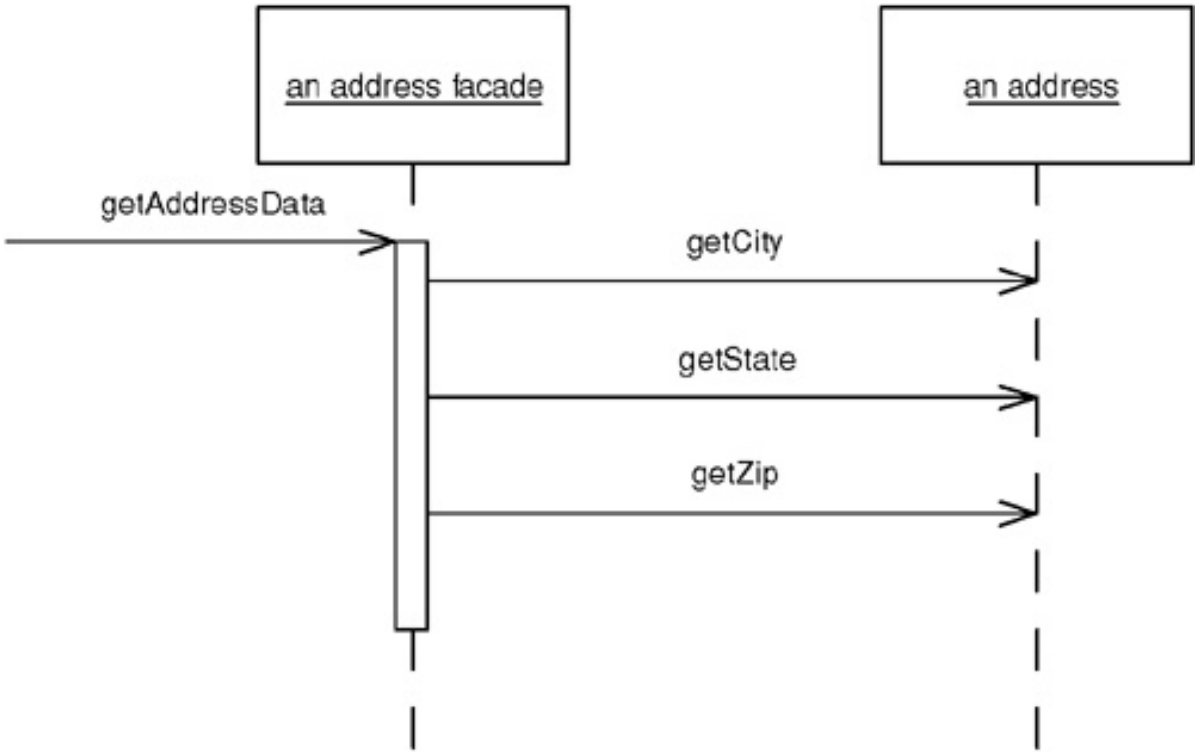


Fowler (2002), p. 388

This pattern is based on GoF's Coarse-Grained Facade.

The facade contains no business logic — it just translates the coarse-grained methods to underlying fine-grained objects.

The facade replaces individual getters and setters with bulk accessors:



Fowler (2002), Fig. 15.1

Distribution strategies

Remote Facade

The facade will often be an interface to a complex web of fine-grained objects — on one client call, it might interact with and integrate data from multiple fine-grained objects.

For data transfer, if the classes are identical on both ends, and they are serializable, then we can simply serialize and transfer them.

Any Remote Facade can be stateful or stateless. One of the three main session state patterns will be needed in this stateful case.

Remote Facades are good places to apply security (authentication and authorization) and transaction control.

Remote Facade has no domain logic!! A good way to tell if you've violated this law is to see if the application can run entirely locally without the remote facade.

Example: Using an EJB session bean as a Remote Facade

EJB session beans can have remote or local interfaces and can be stateful or stateless.

Here we look at POJOs running inside the EJB container that are accessed remotely through a session bean designed as a Remote Facade.

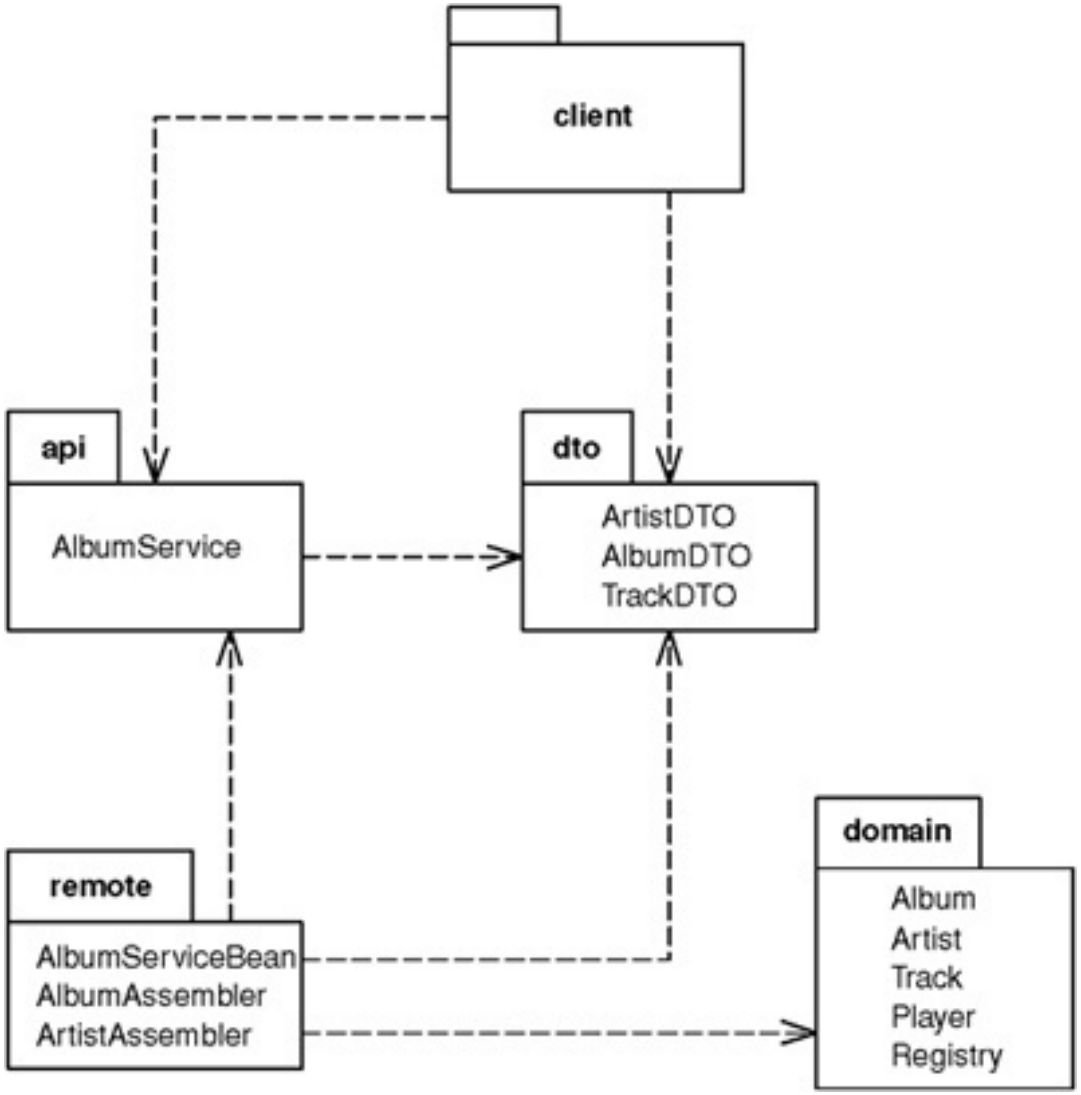
The example is organized into 4 packages:

- Domain Model objects go in package `domain`.
- Objects with remote interfaces go in package `remote`.
- Data Transfer Objects (DTOs) go in package `dto`.
- Interfaces go in package `api`.

[I've modified Fowler's example for EJB 3.0 by dropping the Home interface.]

Distribution strategies

Remote Facade example



Adapted from Fowler (2002), Fig. 15.2

Distribution strategies

Remote Facade example

The remote interface:

```
@Remote
public interface AlbumService {
   String play(String id);
   String getAlbumXml(String id);
   AlbumDTO getAlbum(String id);
   void createAlbum(String id, String xml);
   void createAlbum(String id, AlbumDTO dto);
   void updateAlbum(String id, String xml);
   void updateAlbum(String id, AlbumDTO dto);
   void addArtistNamed(String id, String name);
   void addArtist(String id, String xml);
   void addArtist(String id, ArtistDTO dto);
   ArtistDTO getArtist(String id);
}
```

Note: albums and artists are appearing in the same interface, and there are multiple versions of the same method (accessors working on XML or DTO representations of the domain objects).

Distribution strategies

Remote Facade example

The methods in the facade are simple to implement — each just delegates to another object. Note there is no domain logic!

```
class AlbumServiceBean...

   public AlbumDTO getAlbum(String id) {
         return new AlbumAssembler().writeDTO(Registry.findAlbum(id));
   }
   public String getAlbumXml(String id) {
         AlbumDTO dto = new AlbumAssembler().writeDTO(Registry.findAlbum(id));
         return dto.toXmlString();
   }
   public void createAlbum(String id, AlbumDTO dto) {
         new AlbumAssembler().createAlbum(id, dto);
   }
   public void createAlbum(String id, String xml) {
         AlbumDTO dto = AlbumDTO.readXmlString(xml);
         new AlbumAssembler().createAlbum(id, dto);
   }
   ...
```

Distribution strategies

Remote Facade example

```
public void updateAlbum(String id, AlbumDTO dto) throws RemoteException {
      new AlbumAssembler().updateAlbum(id, dto);
}
public void updateAlbum(String id, String xml) throws RemoteException {
      AlbumDTO dto = AlbumDTO.readXmlString(xml);
      new AlbumAssembler().updateAlbum(id, dto);
}
```

Distribution strategies

Remote Facade example

Some test code using JUnit (note for EJB3 we should use EJB3Unit):

- This case goes from domain object to DTO to XML back to DTO.
- We don't need to deploy to the EJB container — we instantiate the session bean directly.

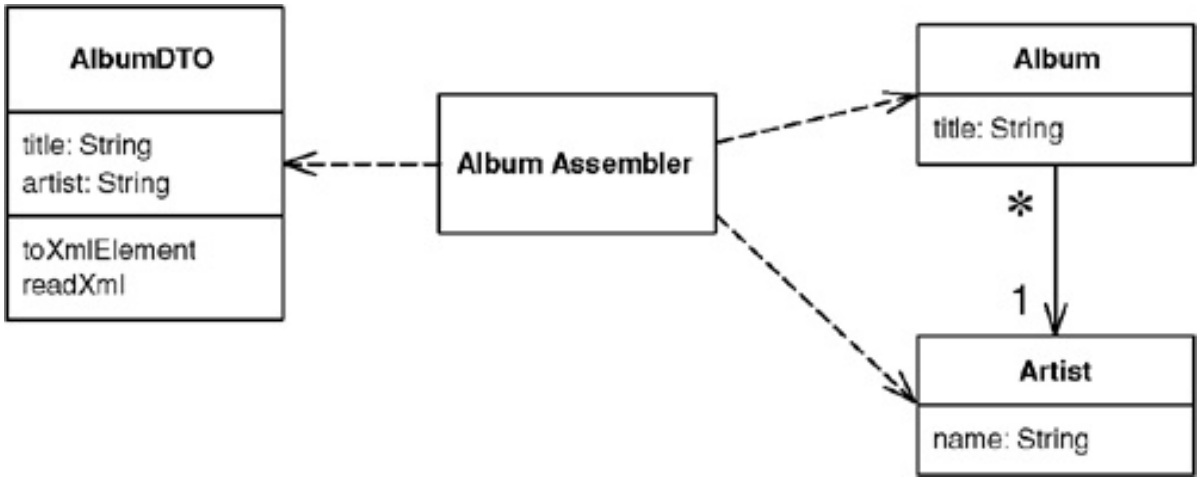
```
class XmlTester...

   private AlbumDTO kob;
   private AlbumDTO newkob;
   private AlbumServiceBean facade = new AlbumServiceBean();
   protected void setUp() throws Exception {
      facade.initializeForTesting();
      kob = facade.getAlbum("kob");
      Writer buffer = new StringWriter();
      kob.toXmlString(buffer);
      newkob = AlbumDTO.readXmlString(new StringReader(buffer.toString()));
   }
   public void testArtist() {
      assertEquals(kob.getArtist(), newkob.getArtist());
   }
```

Distribution strategies

Data Transfer Object

A Data Transfer Object *is an object that carries data between processes in order to reduce the number of method calls.*



Fowler (2002), p. 401

Each call to a remote interface is expensive, so we try to minimize the number of needed calls by globbing data into one class.

Distribution strategies

Data Transfer Object

Some properties/guidelines for a Data Transfer Object (DTO):

- It should hold all the data for a call.
- It must be serializable (not too complex).
- It should only have attributes, getters, and setters.
- It often wraps more data than is really needed: sending too much data in one call is better than multiple calls.
- The attributes need not be cohesive: a DTO often combines more than one domain object.

Distribution strategies

Data Transfer Object

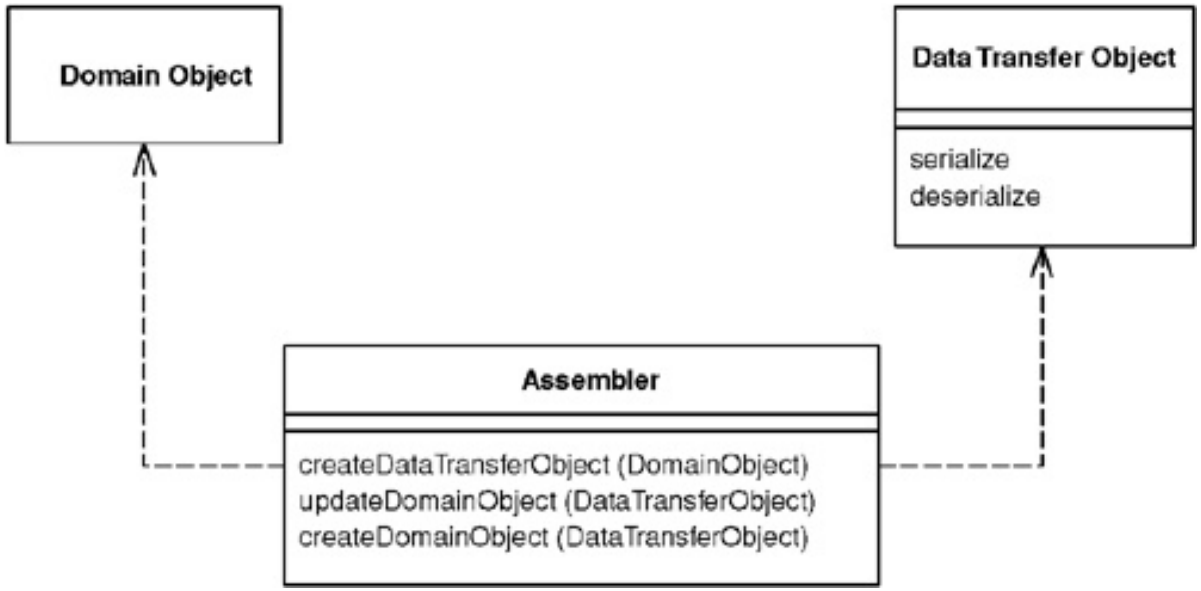
More properties of a DTO:

- Attribute types should be primitives or other DTOs (but keep the graph simple).
- The DTO classes must be present on both sides of the channel.
- The DTOs might be different for different clients even when they're transferring the same objects (e.g. XML vs. binary serialization).
- You might use the same DTO for several related requests.
- Some folks prefer DTOs to be immutable but it's not necessary. A mutable DTO is good when you want to build the object up gradually.

Record Set is a pretty common DTO. It's the DTO for a SQL database, and as we saw it works well with Table Modules and frameworks like .NET.

XML serialization will usually be the best choice, unless performance is a problem. Java has built-in binary serialization, and .NET has both binary and XML serialization.

Use an assembler to decouple the domain objects from the DTOs:

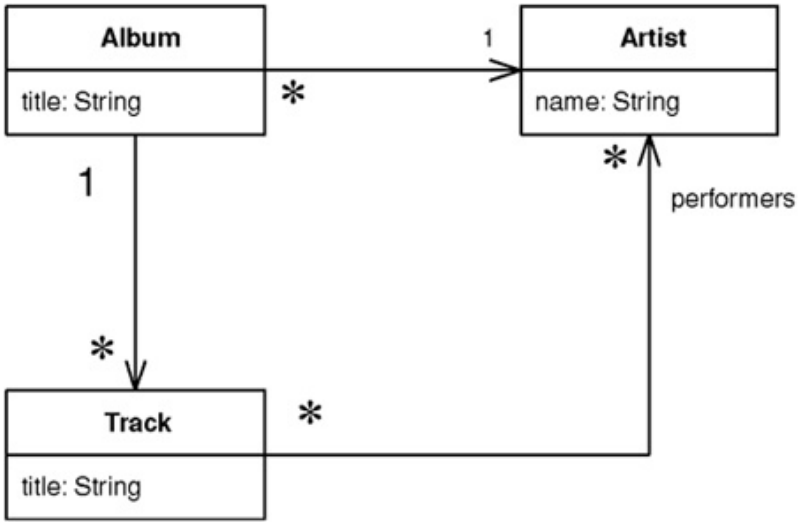


Fowler (2002), Fig. 15.4

Distribution strategies

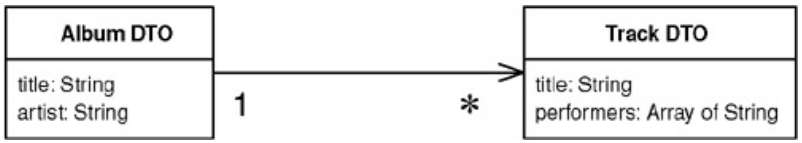
Data Transfer Object example

Example: transferring information about albums. Suppose we have this domain model:



Fowler (2002), Fig. 15.5

We'll transfer them using this scheme:



Fowler (2002), Fig. 15.6

Distribution strategies

Data Transfer Object example

The Assembler code:

```
class AlbumAssembler...

   public AlbumDTO writeDTO(Album subject) {
      AlbumDTO result = new AlbumDTO();
      result.setTitle(subject.getTitle());
      result.setArtist(subject.getArtist().getName());
      writeTracks(result, subject);
      return result;
   }
   private void writeTracks(AlbumDTO result, Album subject) {
      List newTracks = new ArrayList();
      Iterator it = subject.getTracks().iterator();
      while (it.hasNext()) {
         TrackDTO newDTO = new TrackDTO();
         Track thisTrack = (Track) it.next();
         newDTO.setTitle(thisTrack.getTitle());
         writePerformers(newDTO, thisTrack);
         newTracks.add(newDTO);
      }
      result.setTracks((TrackDTO[]) newTracks.toArray(new TrackDTO[0]));
   }
```

Distribution strategies

Data Transfer Object example

```
private void writePerformers(TrackDTO dto, Track subject) {
   List result = new ArrayList();
   Iterator it = subject.getPerformers().iterator();
   while (it.hasNext()) {
      Artist each = (Artist) it.next();
      result.add(each.getName());
   }
   dto.setPerformers((String[]) result.toArray(new String[0]));
}
```

Distribution strategies

Data Transfer Object example

To go the other way, to update a model item from the DTO, is more difficult. If we're creating new ones:

```
class AlbumAssembler...

   public void createAlbum(String id, AlbumDTO source) {
      Artist artist = Registry.findArtistNamed(source.getArtist());
      if (artist == null)
         throw new RuntimeException("No artist named " + source.getArtist());
      Album album = new Album(source.getTitle(), artist);
      createTracks(source.getTracks(), album);
      Registry.addAlbum(id, album);
   }
   private void createTracks(TrackDTO[] tracks, Album album) {
      for (int i = 0; i < tracks.length; i++) {
         Track newTrack = new Track(tracks[i].getTitle());
         album.addTrack(newTrack);
         createPerformers(newTrack, tracks[i].getPerformers());
      }
   }
```

Distribution strategies

Data Transfer Object example

```
private void createPerformers(Track newTrack, String[] performerArray) {
   for (int i = 0; i < performerArray.length; i++) {
      Artist performer = Registry.findArtistNamed(performerArray[i]);
      if (performer == null)
         throw new RuntimeException("No artist named " + performerArray[i]);
      newTrack.addPerformer(performer);
   }
}
```

Note that we assume the artists are already in the Registry.

To update an existing album:

```
class AlbumAssembler...

   public void updateAlbum(String id, AlbumDTO source) {
      Album current = Registry.findAlbum(id);
      if (current == null)
         throw new RuntimeException("Album does not exist: " + source.getTitle());
      if (source.getTitle() != current.getTitle())
         current.setTitle(source.getTitle());
      if (source.getArtist() != current.getArtist().getName()) {
         Artist artist = Registry.findArtistNamed(source.getArtist());
         if (artist == null)
            throw new RuntimeException("No artist named " + source.getArtist());
         current.setArtist(artist);
      }
      updateTracks(source, current);
   }
```

Distribution strategies

Data Transfer Object example

```
private void updateTracks(AlbumDTO source, Album current) {
   for (int i = 0; i < source.getTracks().length; i++) {
      current.getTrack(i).setTitle(source.getTrackDTO(i).getTitle());
      current.getTrack(i).clearPerformers();
      createPerformers(current.getTrack(i), source.getTrackDTO(i).getPerformers(
   }
}
```

Distribution strategies

Data Transfer Object example

The album object is updated, not replaced, but the artists and tracks are simply replaced. The decision here depends on whether other objects refer to the objects in question.

The serialization in this case is binary. To avoid problems when the DTO object fields are changed or reodered, we can use a map:

```
class TrackDTO...

   public Map writeMap() {
      Map result = new HashMap();
      result.put("title", title);
      result.put("performers", performers);
      return result;
   }
   public static TrackDTO readMap(Map arg) {
      TrackDTO result = new TrackDTO();
      result.title = (String) arg.get("title");
      result.performers = (String[]) arg.get("performers");
      return result;
   }
```

Distribution strategies

Data Transfer Object example

This can be handled automatically using a reflective routine in the superclass:

```
class DataTransferObject...

   public Map writeMapReflect() {
      Map result = null;
      try {
         Field[] fields = this.getClass().getDeclaredFields();
         result = new HashMap();
         for (int i = 0; i < fields.length; i++)
            result.put(fields[i].getName(), fields[i].get(this));
      }catch (Exception e) {throw new ApplicationException (e);
      }
      return result;
   }
```

Distribution strategies

Data Transfer Object example

```
public static TrackDTO readMapReflect(Map arg) {
   TrackDTO result = new TrackDTO();
   try {
      Field[] fields = result.getClass().getDeclaredFields();
      for (int i = 0; i < fields.length; i++)
         fields[i].set(result, arg.get(fields[i].getName()));
   }catch (Exception e) {throw new ApplicationException (e);
   }
   return result;
}
```

The text has an out-of-date example of XML serialization. Nowadays folks use XStream or other tools.

Distribution strategies

Conclusion

Remember Fowler's first law of distribution: don't distribute!

When you must distribute, use coarse-grained interfaces.

That's the essence of synchronous distributed systems.

Next we consider asynchronous distributed systems in more detail.

Outline

Messaging Introduction

What is messaging?

The synchronous RPC techniques we've seen so far are quite limiting, especially for application integration.

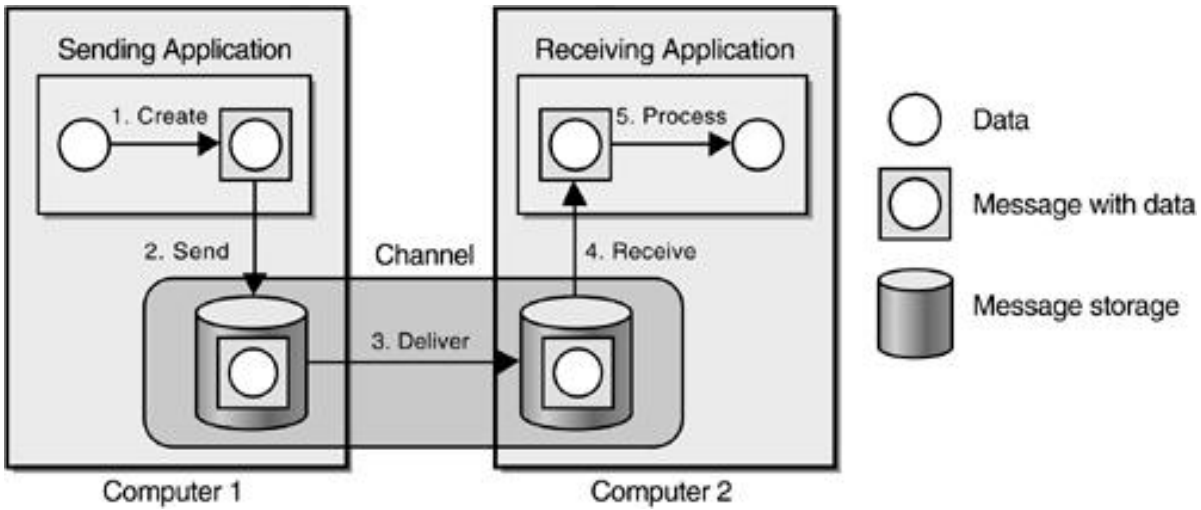
Asynchronous communication has many benefits over RPC.

The basic pattern for asynchronous communication is Messaging:

- Programs communicate with each other using packets of data called messages.
- Channels (or queues) connect programs and allow them to transmit messages.
- A sender or producer puts messages into a channel.
- A receiver or consumer reads (and deletes) messages from a channel.
- A message is composed of a header containing metadata and a body containing application data.

A messaging system or message-oriented middleware (MOM) provides communication services to applications in the same way that DBMSs provide persistence services to applications.

Similar to databases, messaging systems need to be configured by an administrator with specific channels.



Hohpe and Woolf (2004), p. xxxii

Steps:

1. Create: sender creates and populates a message.
2. Send: sender adds the message to a channel.
3. Deliver: the messaging system moves the message from the sender's system to the receiver's system.
4. Receive: the receiver reads and deletes the message from the channel.
5. Process: the receiver extracts the data from the message.

Compared to synchronous communication, asynchronous communication with a messaging system offers many advantages:

- The send and forget paradigm allows more efficient use of resources.
- Store and forward means that messages can take arbitrarily complicated paths to reach the destination.
- The sender and receiver are less coupled.
- Serialization is handled by the messaging system.
- Flexible message formats allow cross-platform integration.
- The producer's speed doesn't depend on the consumer's speed.

More benefits:

- Message consumers don't get overloaded — they process incoming messages at their own pace.
- Reliability is increased, thanks to store-and-forward.
- Processing can continue when disconnected from the network.
- The messaging system acts as a mediator or hub, increasing deployment flexibility.
- Since no application threads block waiting for responses, overall resource consumption is decreased.

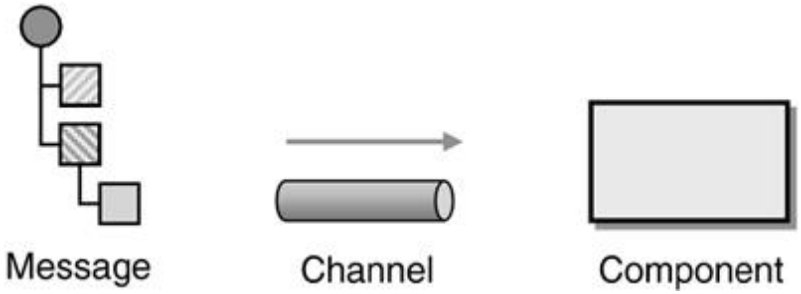
Asynchronous messaging does come with a price, however:

- Debugging in an event-driven model can be confusing.
- Messages can get out of sequence.
- Not all problems can be solved with asynchronous techniques (sometimes the user expects an immediate response).
- Message encoding and decoding adds overhead.
- Commercial messaging systems often don't interoperate.
- Vendor lock-in is a danger, even with standards such as JMS.

Different notation and terminology are used by different groups:

Enterprise Integration Pattern	Java Message Service (JMS) Term
Message Channel	Destination
Point-to-Point Channel	Queue
Publish-Subscribe Channel	Topic
Message	Message
Message Endpoint	MessageProducer, MessageConsumer

Instead of UML, Hohpe and Woolf use their own visual notation:



Hohpe and Woolf (2004), p. xliv

Messages can be complex hierarchically-structured objects.

Components include applications, routers, translators, and so on.

Outline

Most organizations run hundreds or thousands of applications.

Consider just a couple simple scenarios:

- A customer calling in to change address and check whether the last payment was received might require interaction with both a CRM system and a billing system.
- A customer calling into place an order might require interaction with separate CRM, accounting, inventory, shipping, and billing systems.

In even the most "simple" business processes, we are faced with many applications that need some form of integration.

Integration Introduction

Challenges

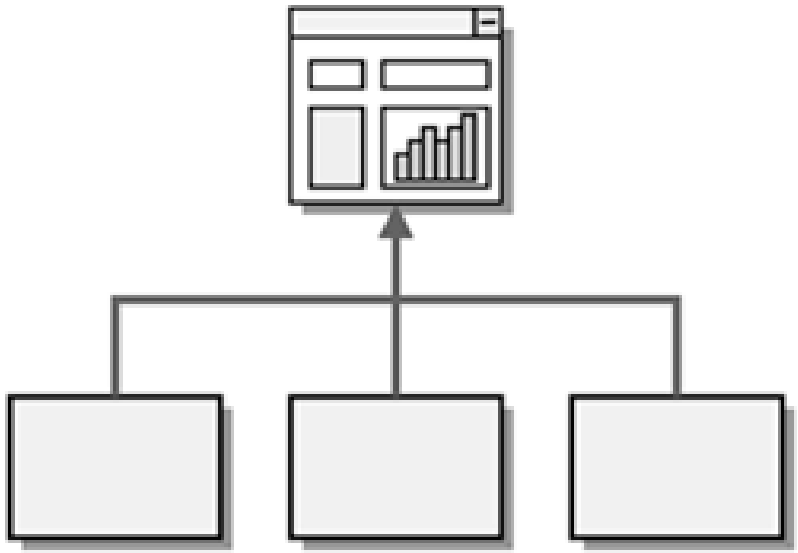
Integration engineers are faced with many difficulties:

- Corporate politics may get in the way of integration. It is difficult to connect applications "owned" by groups that are not used to talking with each other.
- Once online, the integration solution becomes a critical resource. Failures of the the system will be very bad for profits and customer relationships.
- Proprietary packaged applications may not offer any way to support integration.
- Although standards based on XML are a big step towards interoperability, vendor-specific extensions cause fragmentation and a new form of vendor lock in.
- Semantic differences between applications and the way they treat data makes integration difficult.
- Maintenance of an integration solution can be a nightmare.

Not all integration problems are the same. Most applications fall into one of these categories:

- Information portals
- Data replication
- Shared business functions
- Service-oriented architectures
- Distributed business processes
- Business-to-business integration

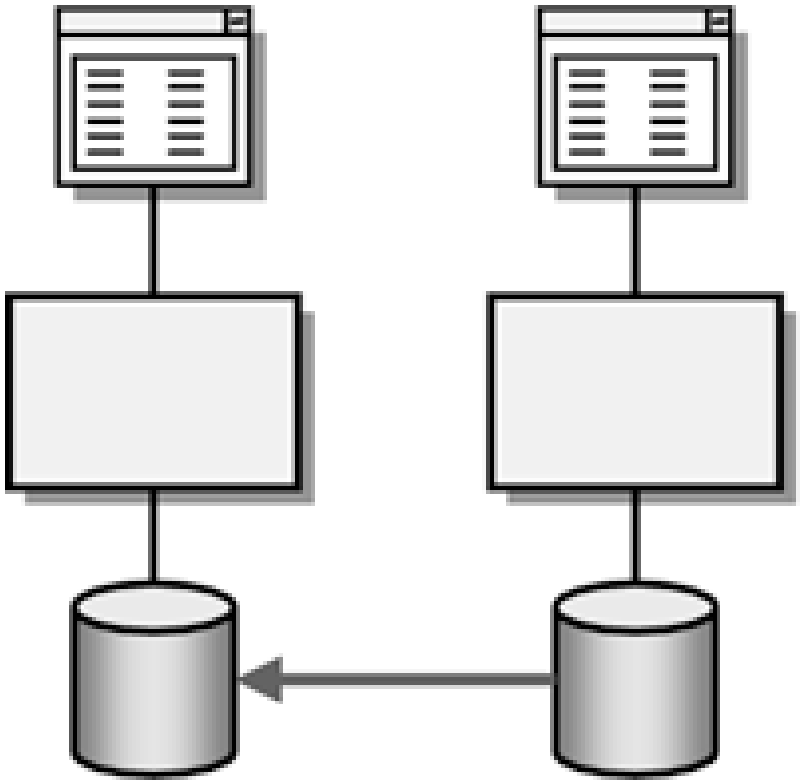
Information portals help us to avoid using multiple systems to get a job done. Typically divided screen with little interaction.



Hohpe and Woolf (2004), p. 6

Data replication moves data between applications at the data store level.

For example, a `Customer` object might exist in customer care, accounting, shipping, and billing systems.



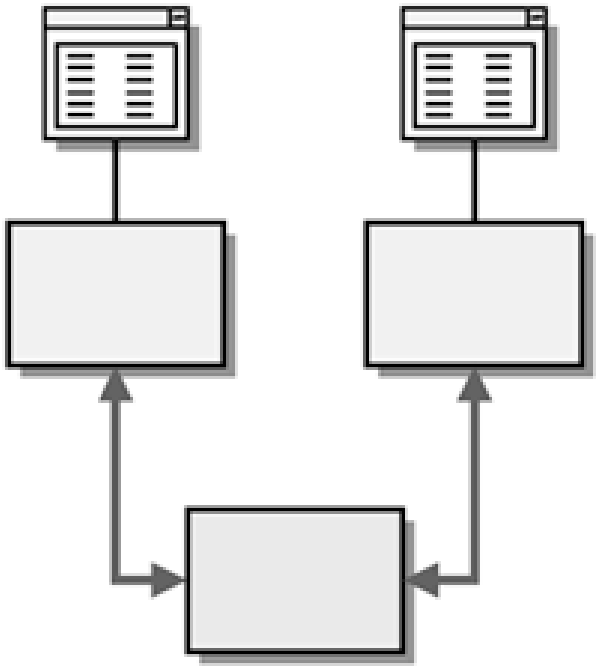
Hohpe and Woolf (2004), p. 6

Shared business functions are pieces of business logic/data that are needed by multiple applications.

Instead of replication, we keep the logic in one application.

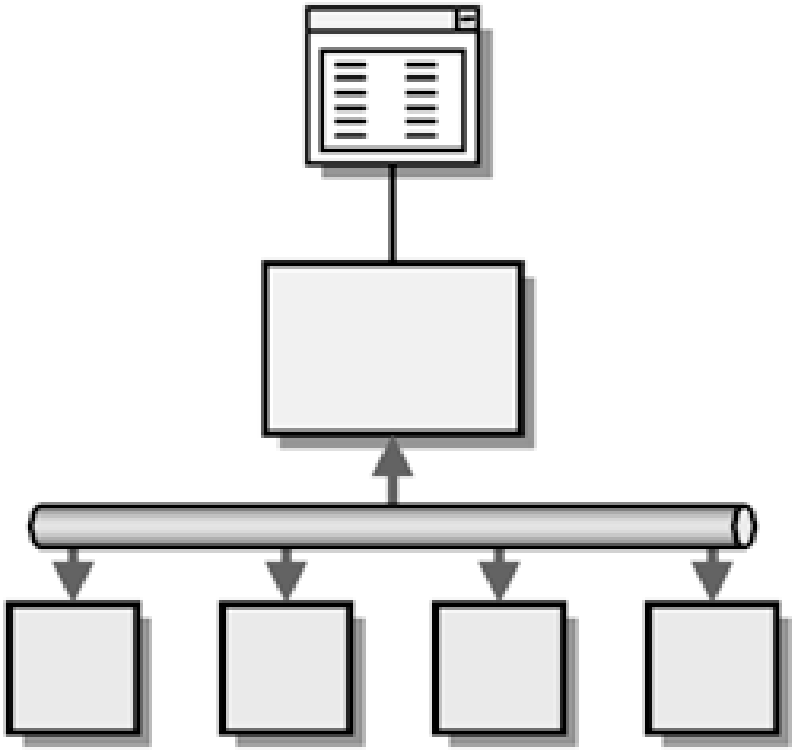
Example: postal code verification for an `Address`.

.



Hohpe and Woolf (2004), p. 7

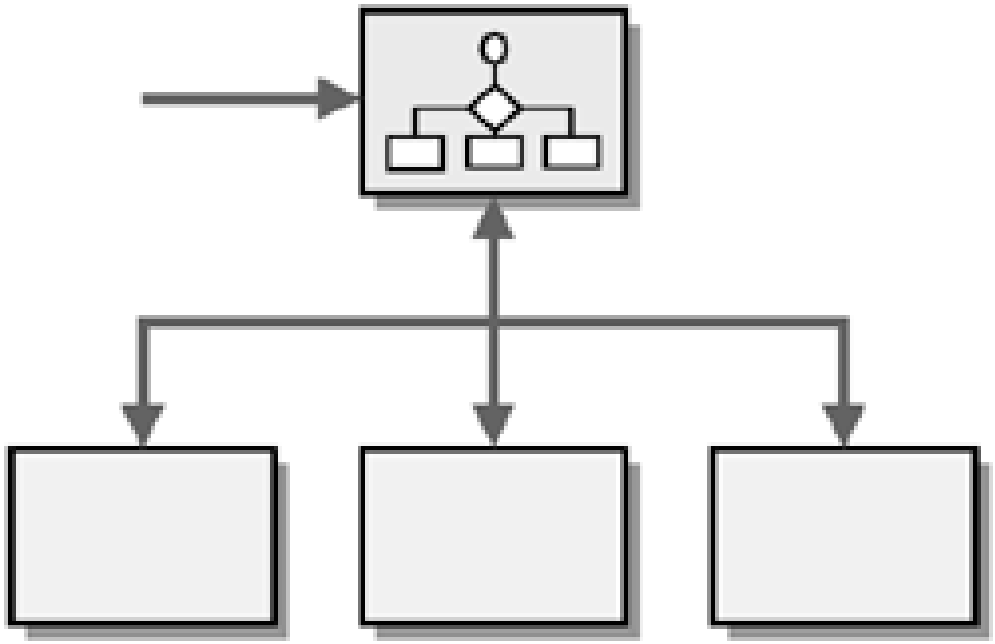
Service-oriented architectures provide a directory of services (shared business functions) and publish public contracts for those services.



Hohpe and Woolf (2004), p. 8

Distributed business processes automate a complex business process by coordinating multiple applications.

Oftentimes the applications' services are organized by a SOA.

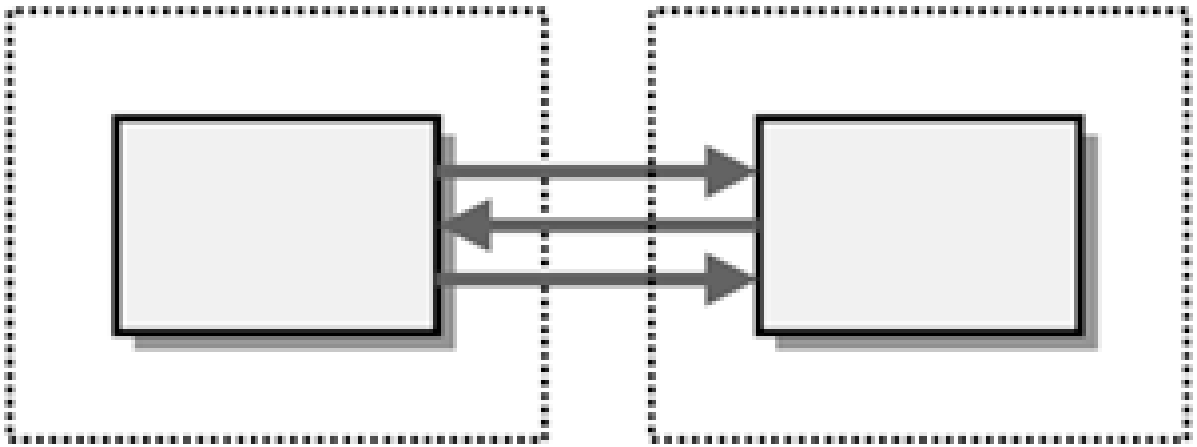


Hohpe and Woolf (2004), p. 8

Business-to-business applications involve coordination across corporate boundaries, usually over the Internet.

Examples include e-commerce systems that interact with a shipping company's system or a retailer interacting with a supplier.

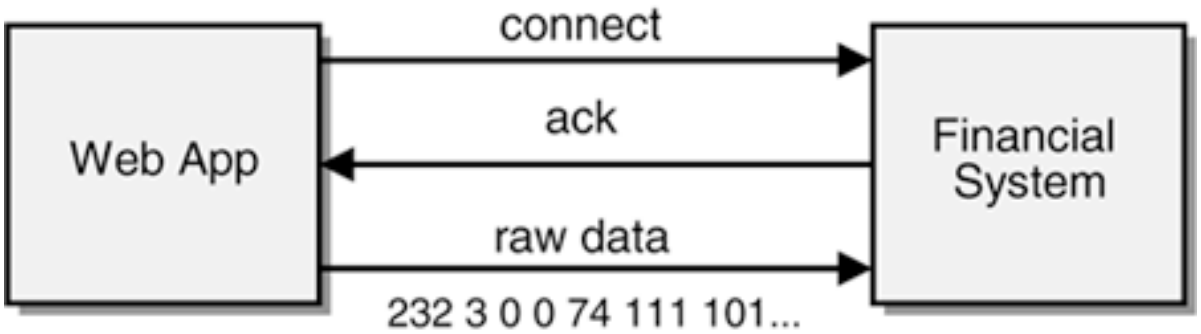
The considerations are the same as for the other styles, but now security and fault tolerance become even more critical.



Hohpe and Woolf (2004), p. 9

Why do we need complex messaging systems?

For example, every OS has TCP/IP built in. Why not just open a socket to the service provider/message consumer and send the information?



Hohpe and Woolf (2004), p. 13

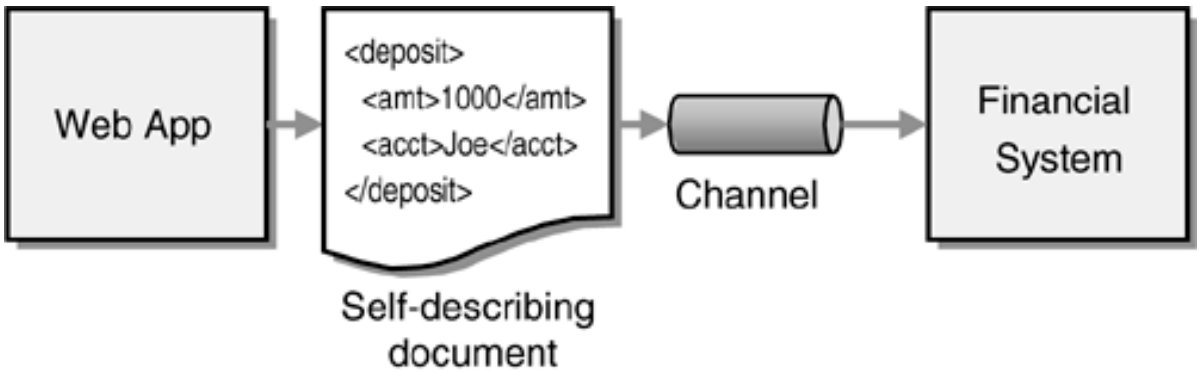
The problem is that over-simplistic integration approaches create several types of coupling:

- The participants are coupled by data representations (word size, big/little endianness, etc.).
- They are coupled by location (the sender needs to know the IP or hostname of the receiver).
- They are temporally coupled (both parties need to be available at the time of communication).
- They are coupled by data format (order and/or names of parameters, for example).

Coupling reduces maintainability, modifiability, and reuse.

We can reduce coupling at the cost of additional complexity:

- We introduce a self-describing, platform-independent representation of data such as XML.
- We send to a Message Channel instead of directly to the recipient.
- We use aynchronous message queues instead of synchronous communication.



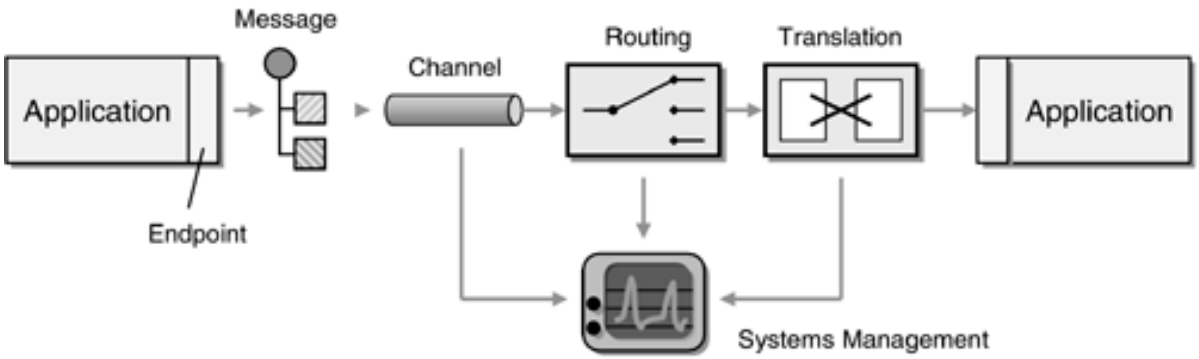
Hohpe and Woolf (2004), p. 14

Integration Introduction

Why simple solutions don't work

Just a message format and channel is not enough, however.

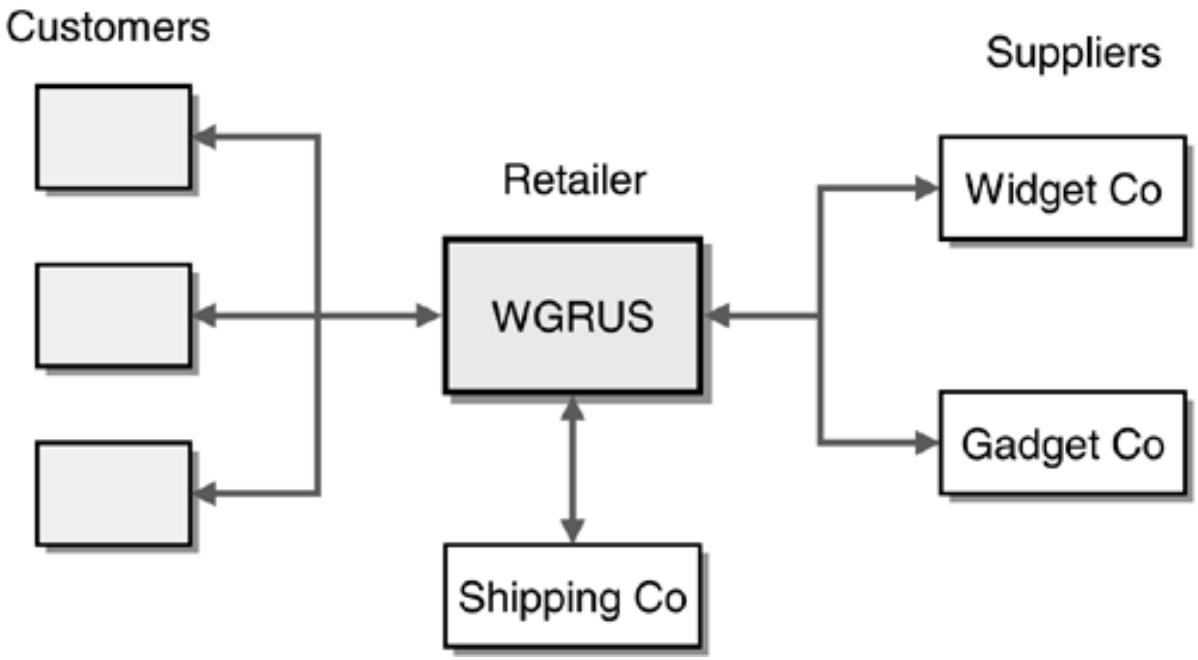
- When data formats disagree, we add message transformations.
- To decouple by location/IP, we add message routing.
- To keep the system running smoothly, we add systems management.



Hohpe and Woolf (2004), p. 15

We hope that the additional complexity can be mitigated through the use of message-oriented-middleware.

As a fictitious case study, consider the online retailer Widgets and Gadgets 'R Us:



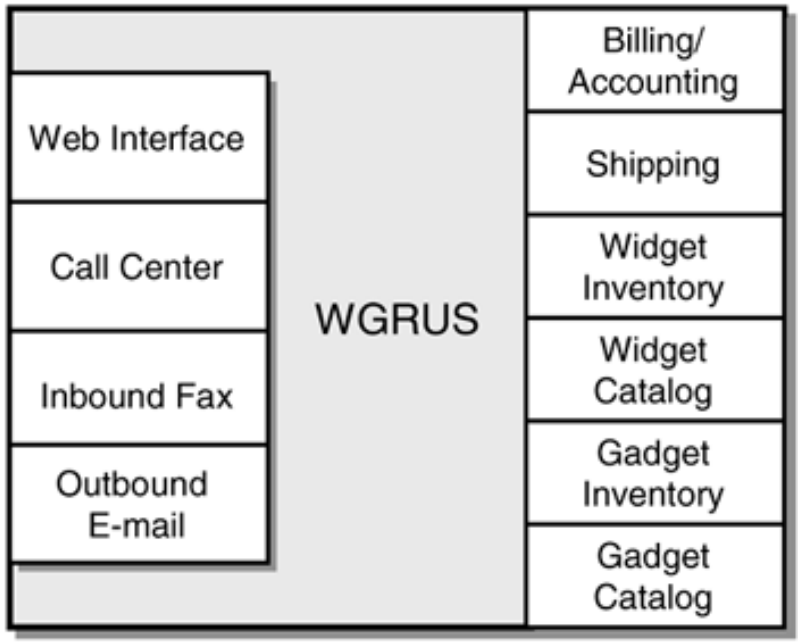
Hohpe and Woolf (2004), p. 17

WGRUS buys widgets and gadgets from retailers and sells them to customers.

Suppose we want to streamline the business processes for

- Take Orders
- Process Orders
- Check Status
- Change Address
- New Catalog
- Announcements
- Testing and Monitoring

Further suppose that we have to work within an existing IT infrastructure containing several existing applications:



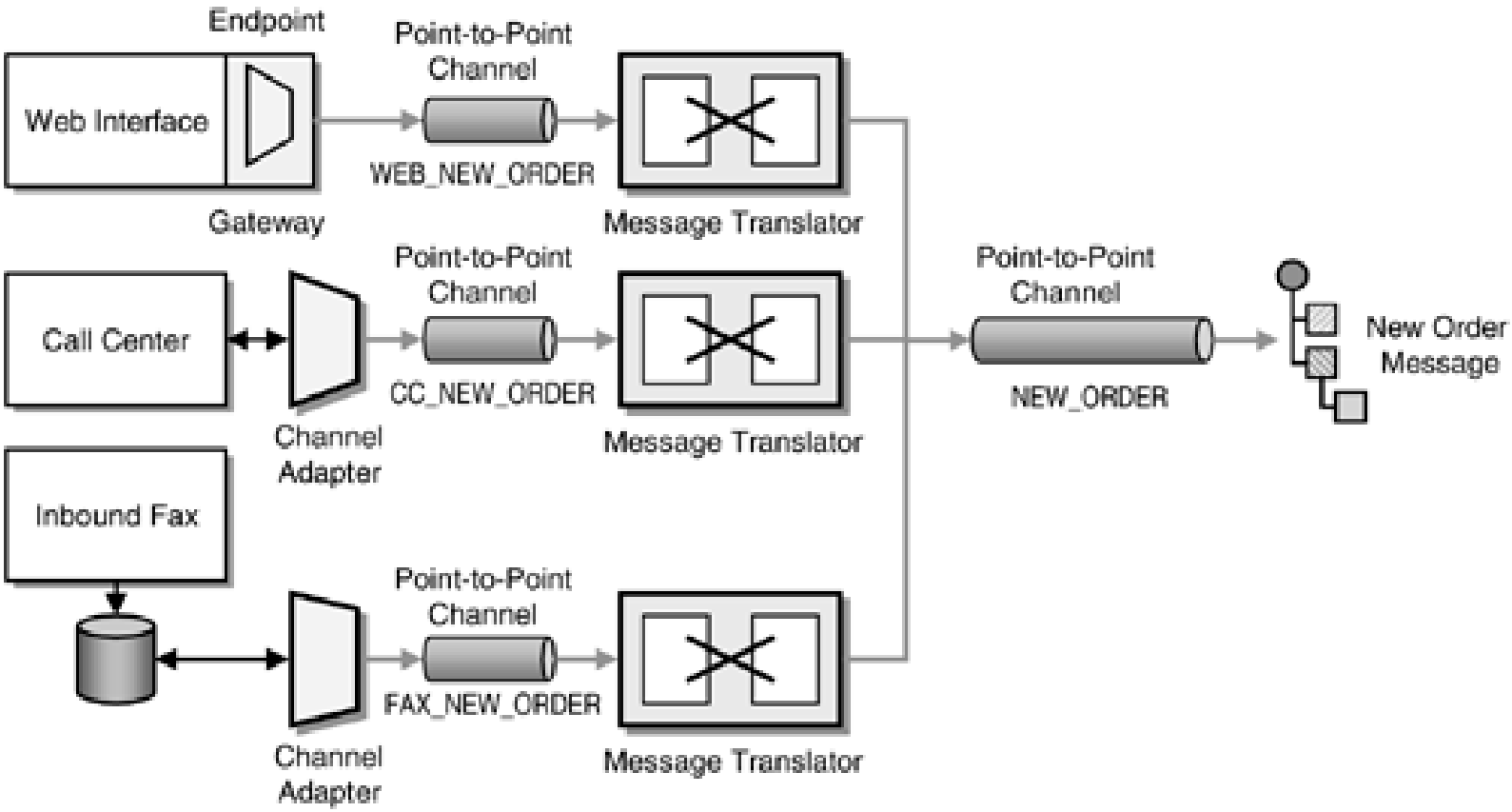
Hohpe and Woolf (2004), p. 18

The current Take Orders system requires 3 separate modalities that use existing applications:

- The Web interface is a custom J2EE application. Since we have control, we can use the Message Endpoint pattern via a Message Gateway, which isolates application code from the messaging system.
- The call center uses a packaged application. Since we have no control over this application, we use a Channel Adapter which attaches to the application and publishes events to a Message Channel as they occur.
- Inbound faxes are manually entered into a MS Access database. Here we use a Channel Adapter at the data store level.

Integration Introduction

Case study: Take Orders at WGRUS

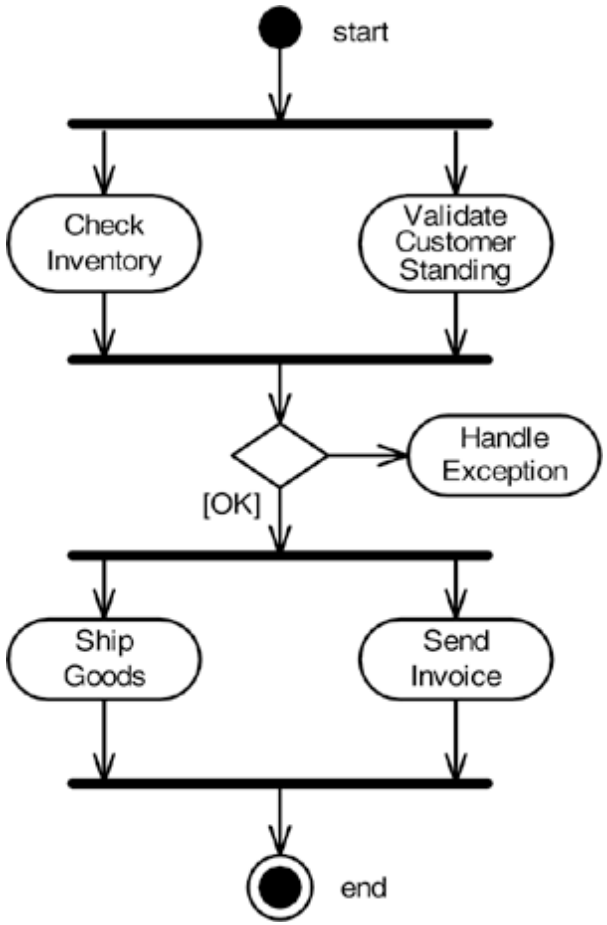


Hohpe and Woolf (2004), p. 19

We need a few more components to unify the three order entry applications:

- We need Message Translators to translate the application-specific message formats into a Canonical Data Model.
- Since we want to decouple the canonical New Order message format from the application message formats, each application-specific Message Channel is named for the application that publishes to it. They need to be Point-To-Point Channels because we want each produced message to be processed exactly one time.
- The `NEW_ORDER` Message Channel is likewise point-to-point. Since it only uses one kind of message, it is a Datatype Channel, and the New Order message is a Document Message because it describes something rather than requesting action.

The worflow for Process Orders can be described succinctly by a UML activity diagram:

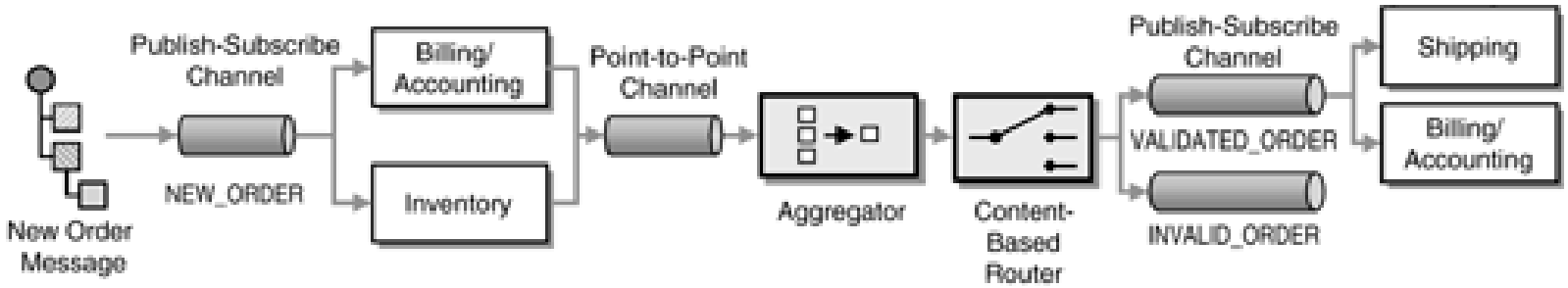


Hohpe and Woolf (2004), p. 21

The integration solution needs to communicate with the accounting system (customer validation, billing, and invoices), the inventory system, and the shipping system.

This is an example of a distributed business process.

The activity diagram can be translated directly into integration solution components:



Hohpe and Woolf (2004), p. 22

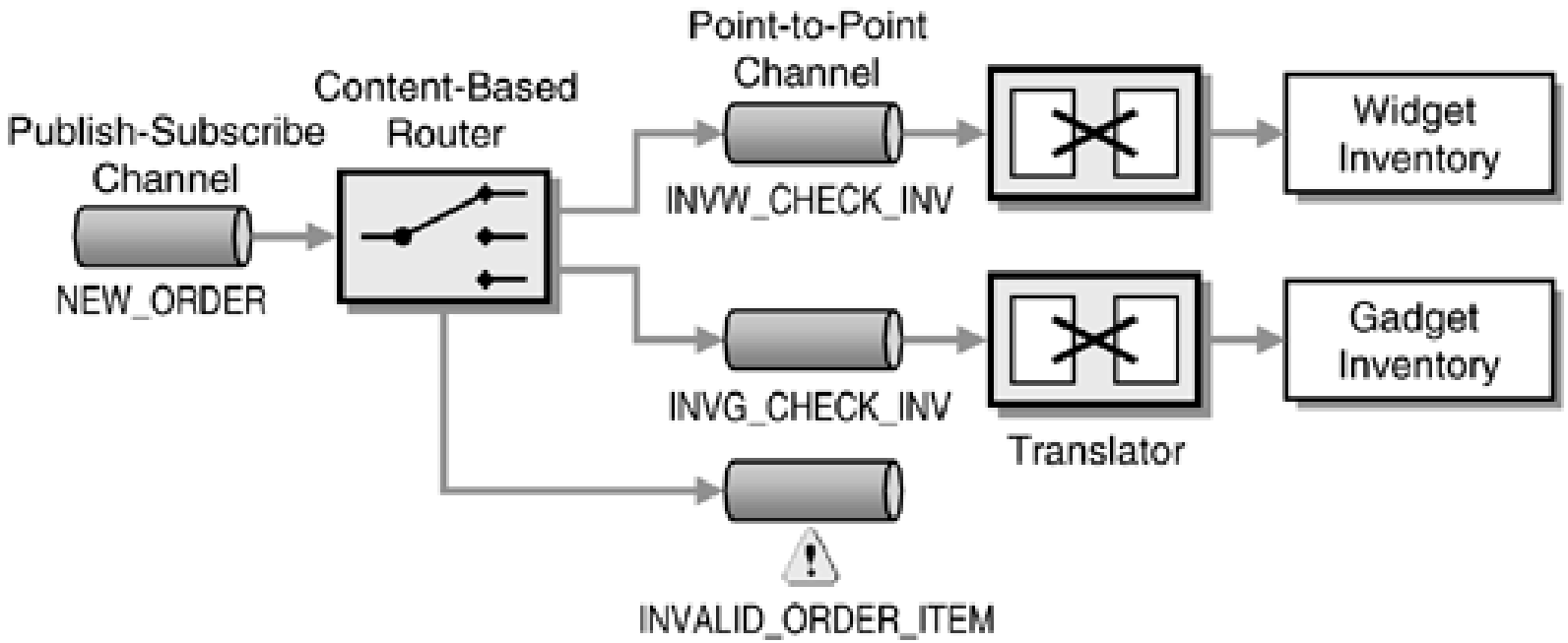
We use Publish-Subscribe Channels for forks and Aggregators for joins.

A Content-Based Router is an appropriate pattern for decision nodes in the workflow.

Within the inventory checking activity, suppose that Widget requests need to go to one inventory system and Gadget requests need to go to another.

This suggests another Content-Based Router along with Message Translators to transform the inventory checking commands into application-specific form.

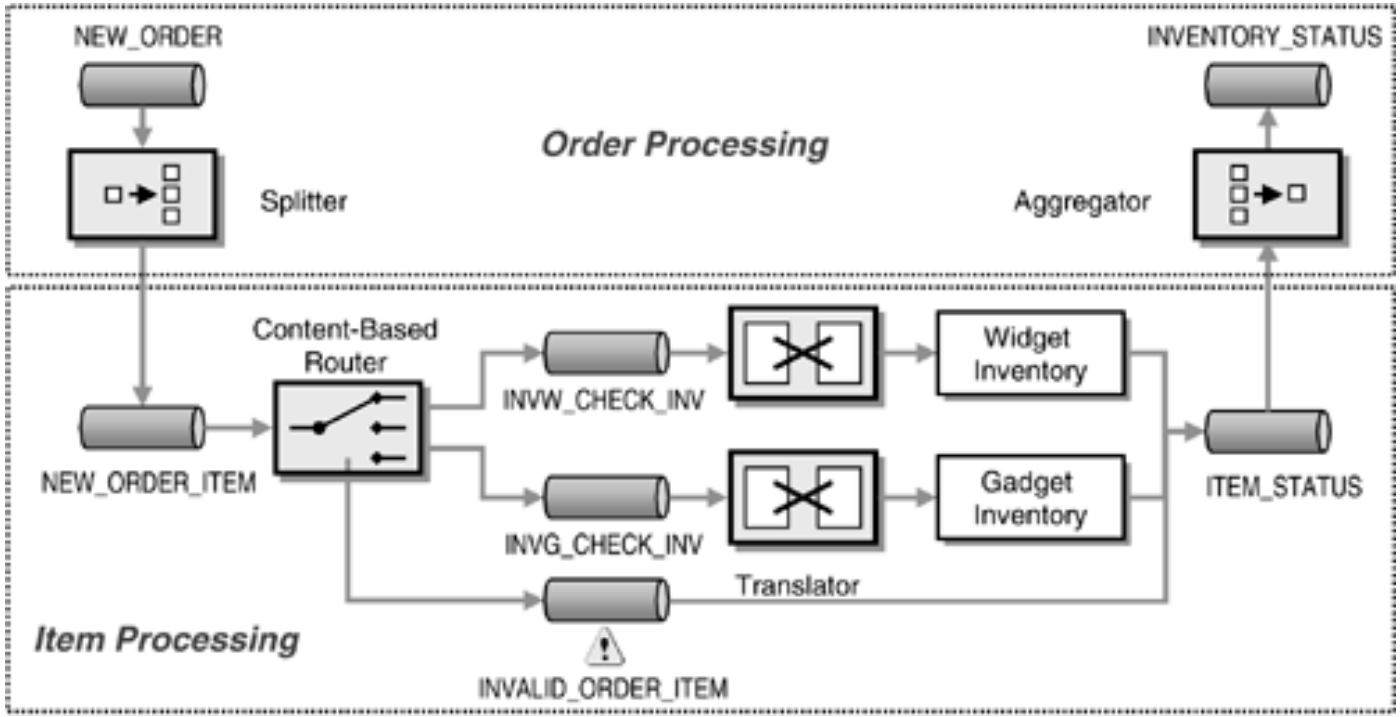
Note that the New Order Document Message is interpreted as a Command Message in the inventory checking message channels.



Hohpe and Woolf (2004), p. 23

The `INVALID_ORDER` and `INVALID_ORDER_ITEM` channels are examples of Invalid Message Channel.

To handle multiple items in one order, we can use a Splitter and an Aggregator.

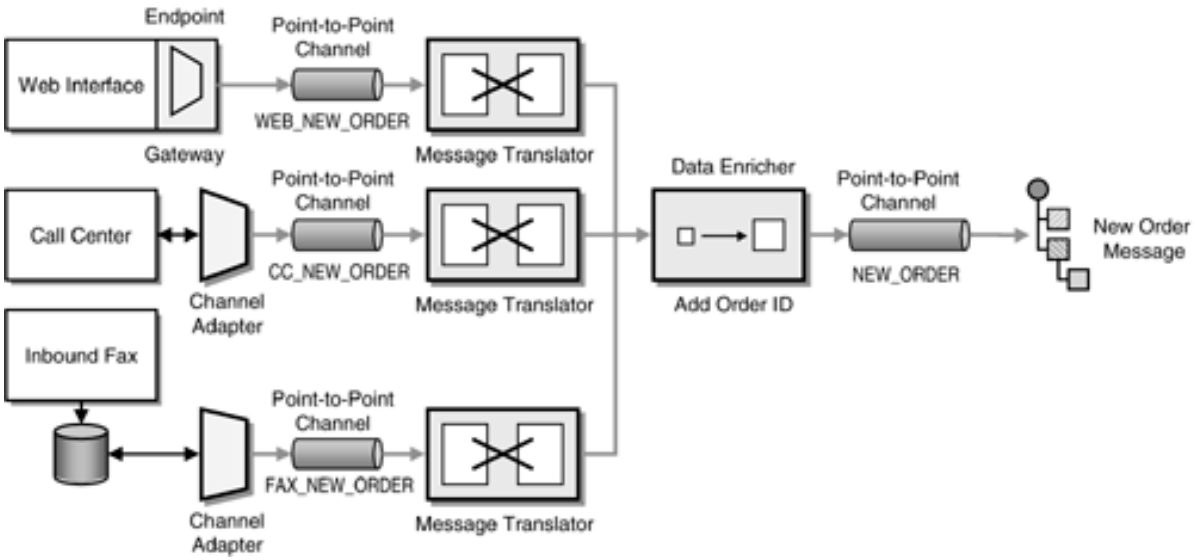


Hohpe and Woolf (2004), p. 24

The Aggregator needs to do correlation, completeness checking, and message combination.

To perform correlation, our aggregator needs some kind of order ID. Where does this come from?

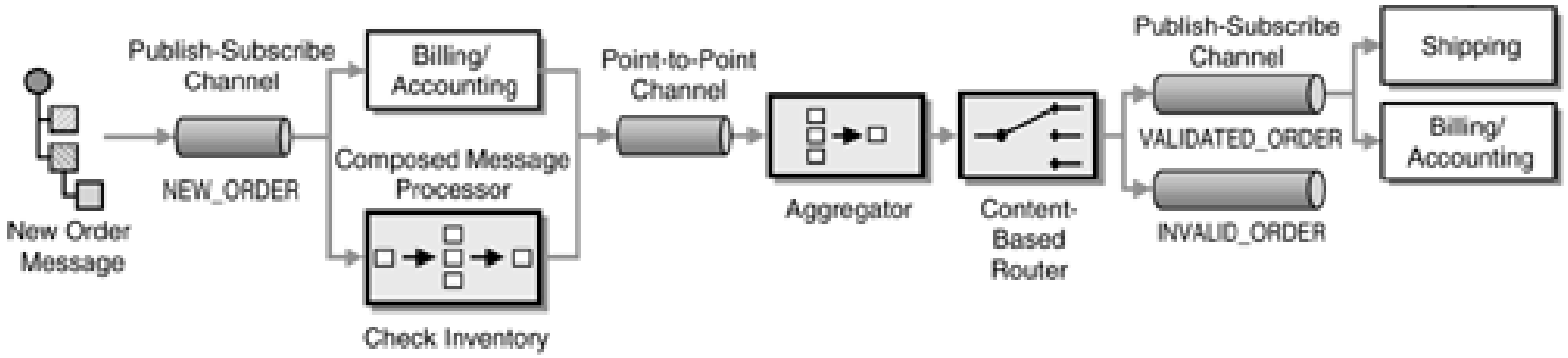
The order ID should be assigned in the Take Orders workflow when the New Order message is created. This is the DATA ENRICHER pattern:



Hohpe and Woolf (2004), p. 25

With a unique order ID for every New Order message, the Aggregator can easily aggregate each validated Item Status message into a composed Inventory Status message for the order.

Any time we have a Splitter, Router, and Aggregator, we have a case of Composed Message Processor.



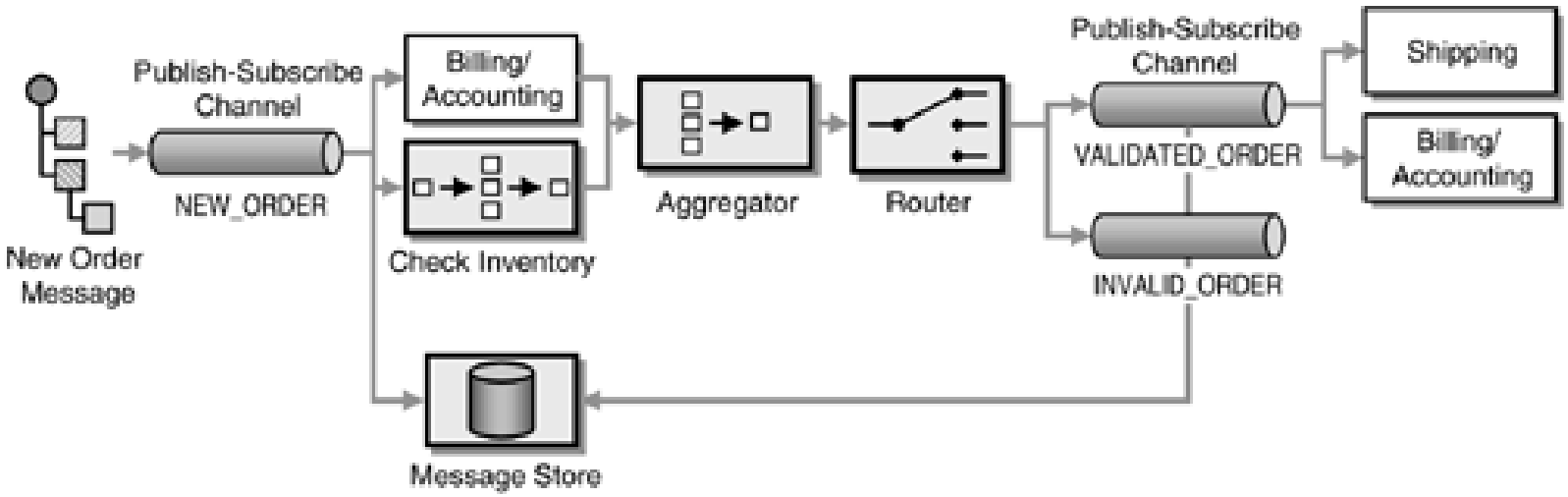
Hohpe and Woolf (2004), p. 25

Customers and managers will want to check the status of a long-running business process like Process Order.

To determine the state of an uncompleted process, what do we need to know?

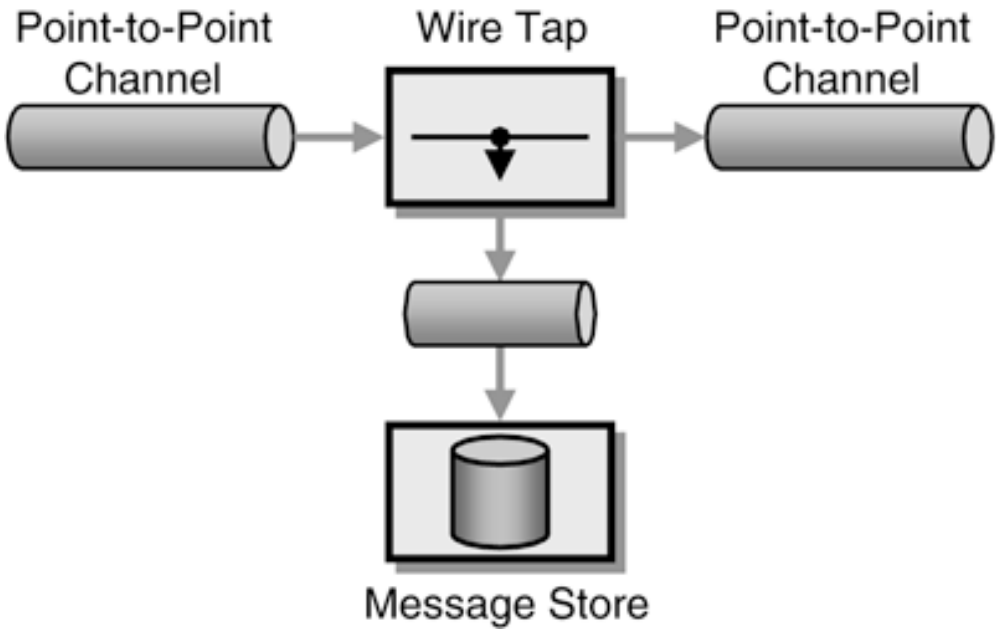
We need to know the last message related to the order.

To track the messages associated with an order, we introduce a Message Store as a subscriber to each Publish-and-Subscribe channel.



Hohpe and Woolf (2004), p. 26

In the case of Point-to-Point Channels, we the Message Store cannot subscribe; instead it can use a Wire Tap that forwards a message to two Point-to-Point Channels.



Hohpe and Woolf (2004), p. 27

Message Stores are convenient in that they prevent us from having to copy extraneous information into messages.

Example: if we store the New Order message in the Message Store at the beginning of Process Order, later stages can lookup the complete information about the order in the Message Store as needed.

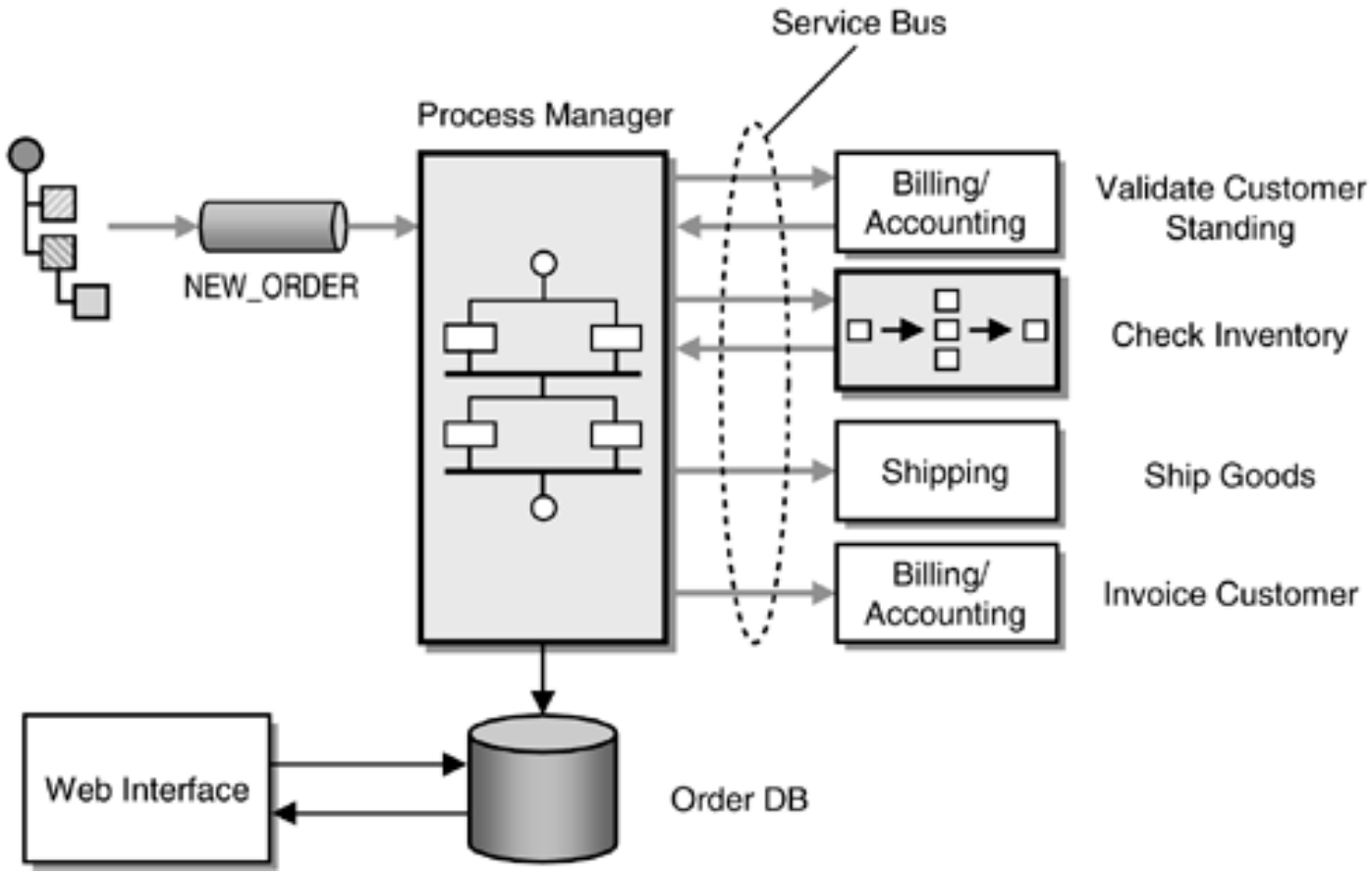
This is an example of Claim Check.

Taking the Message Store idea further, we can move the Aggregator and related logic into the Message Store.

Putting the glue logic in the Message Store turns it into a Process Manager that uses a Process Template to store intermediate process state and keep track of the progress of a process.

Integration Introduction

Case study: Check Status at WGRUS



Hohpe and Woolf (2004), p. 28

Process Manager lets us treat applications as reusable shared business functions connected to a service bus.

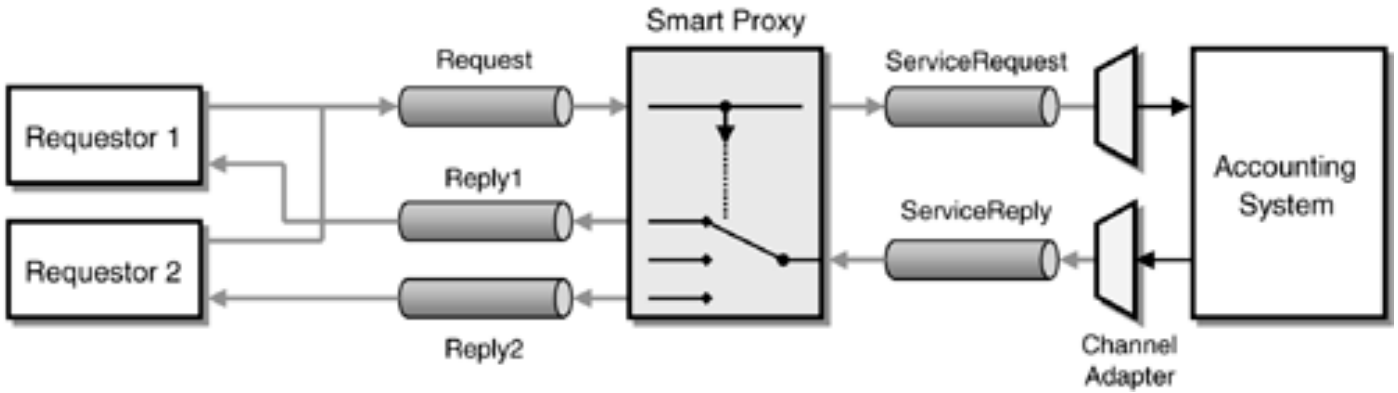
If we add discovery and public contracts, we have a SOA.

Note that request-reply services connected to the bus need to to know what channel to reply to. Producers invoking such services should provide a Return Address.

Finally, we need a way for the Web interface to gather the information needed to check order status. This is naturally accomplished using a Shared Database.

This couples the Process Manager and the Web interface by forcing them to use the same data schema, but this may be an acceptable risk.

Legacy systems may not support Return Address. If so, we may need to use a Smart Proxy to wrap the legacy system and route reply messages to the correct Return Address channel.



Hohpe and Woolf (2004), p. 29

Billing and shipping both need the customer's `Address` to perform their tasks.

How can they get an address?

- We can include the address in the New Order message.
- We can store address data in each system and replicate it.

How to pick a solution, and how can we handle updates?

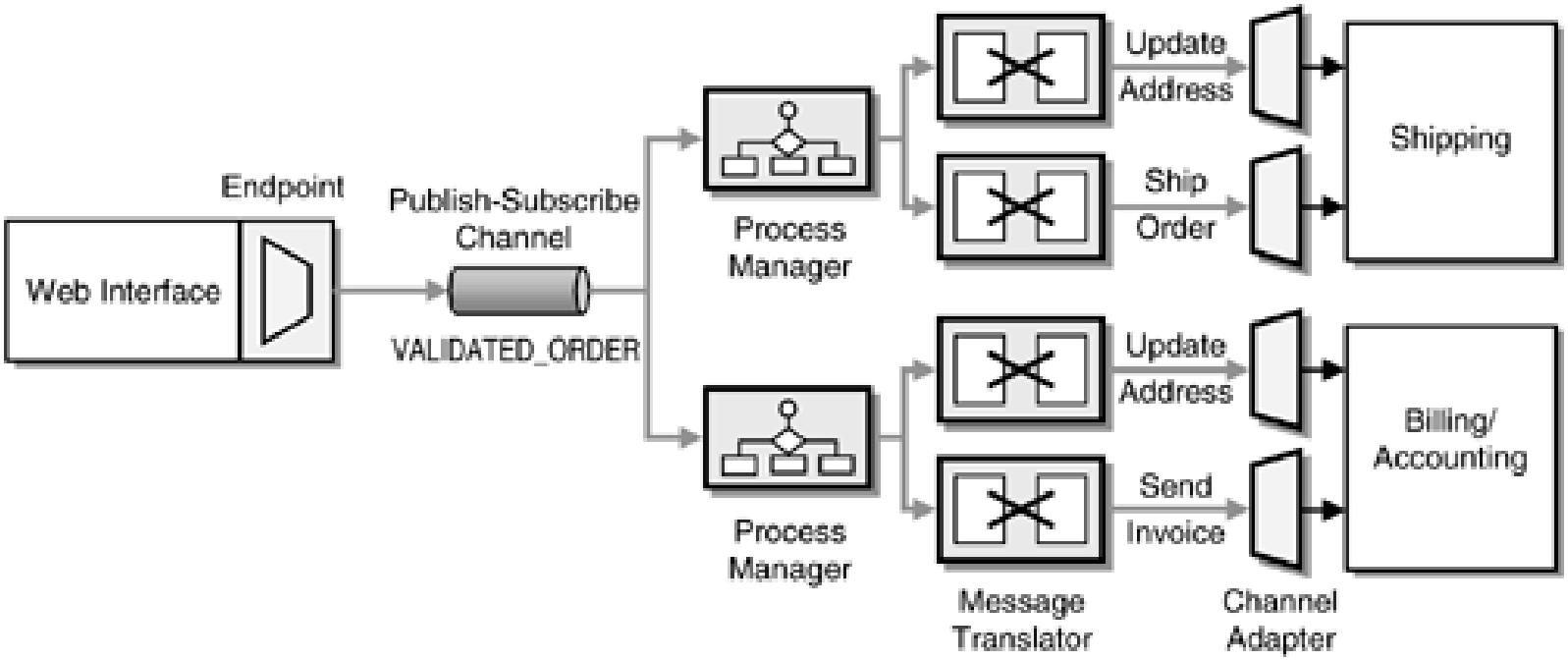
Solution 1: new addresses are included in the New Order message.

If the application does not support updating the address at the same time as the order is entered, we have to introduce sequential logic with the help of a Process Manager.

The New Order message is in canonical form, so it needs to be translated into private formats using Message Translators.

The Message Translators could go inside the Process Manager, but it is better to encapsulate the message-specific details.

Solution 1 (new address ships with the New Order message):



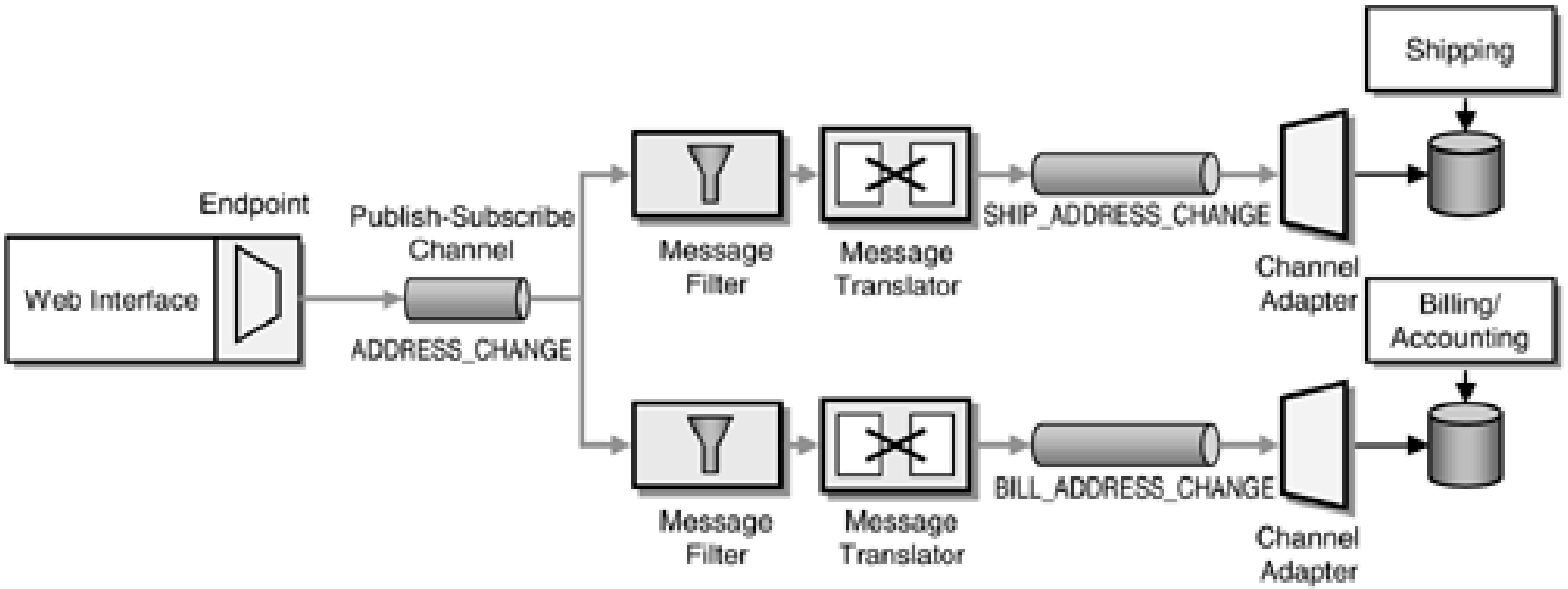
Hohpe and Woolf (2004), p. 30

Solution 2: we propagate address changes from the Web interface. Address changes are published to a Publish-and-Subscribe channel, and interested parties subscribe.

To avoid incorrect behavior like having the Shipping address change when the Billing address changes, we introduce Message Filters to select the right kind of address change messages.

As usual, Message Translators transfrom canonical formats into private message formats.

Solution 2 (address data is stored in each application):



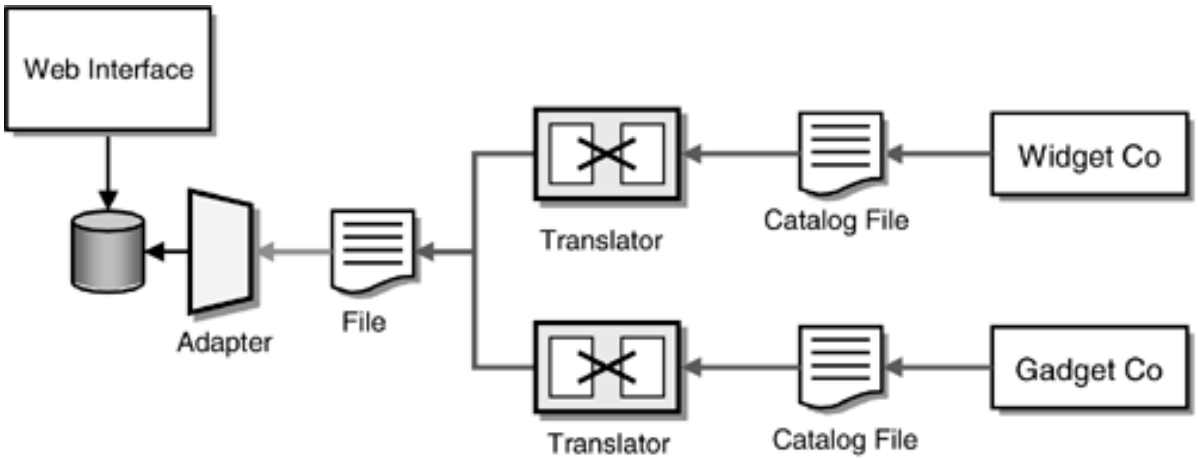
Hohpe and Woolf (2004), p. 31

Which of the methods should we use?

Usually, the application and the API it publishes will dictate the strategy.

Now, suppose WGRUS's suppliers update their catalogs every month.

We use File Transfer.



Hohpe and Woolf (2004), p. 33

File Transfer is very efficient for bulk updates.

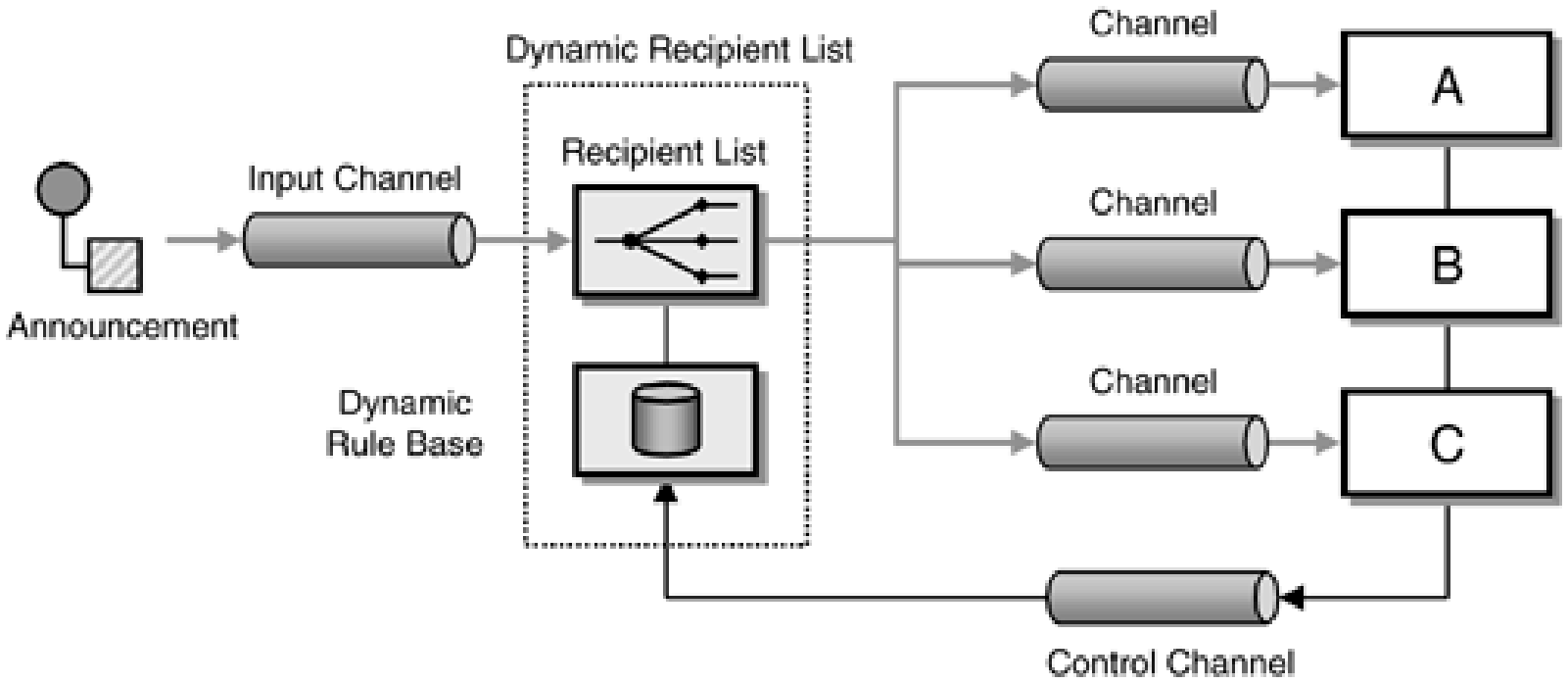
We want each customer to provide preferences on what product announcements to receive.

Publish-and-Subscribe Channels are not fine-grained enough for this task.

We use a Dynamic Recipient List combining Recipient List with Dynamic Router (a Router whose routing algorithm changes in response to control messages).

For email, each subscriber's email address is a channel.

For a SOAP Web service, on the other hand, each customer's URI would be a channel.



Hohpe and Woolf (2004), p. 34

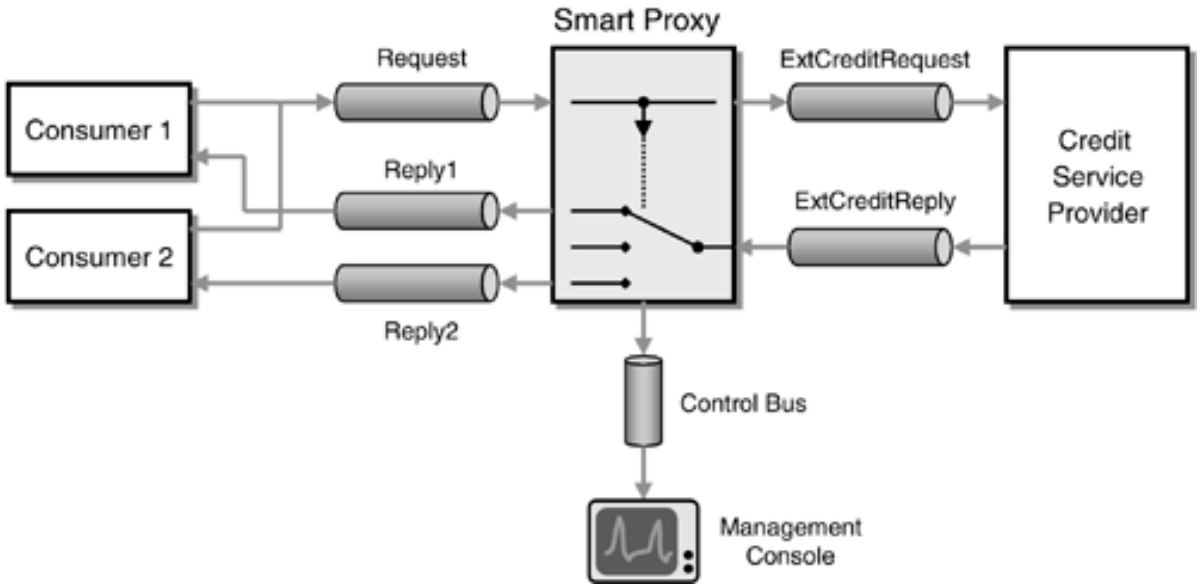
Monitoring the system is critical.

Suppose we access an external credit agency to check the creditworthiness of customers.

Suppose the contract is pay-per-use with a QoS guarantee that says we do not have to pay if the response time is too slow.

To verify an invoice from the credit service provider, we need to have precise records of our requests and the response time for each request.

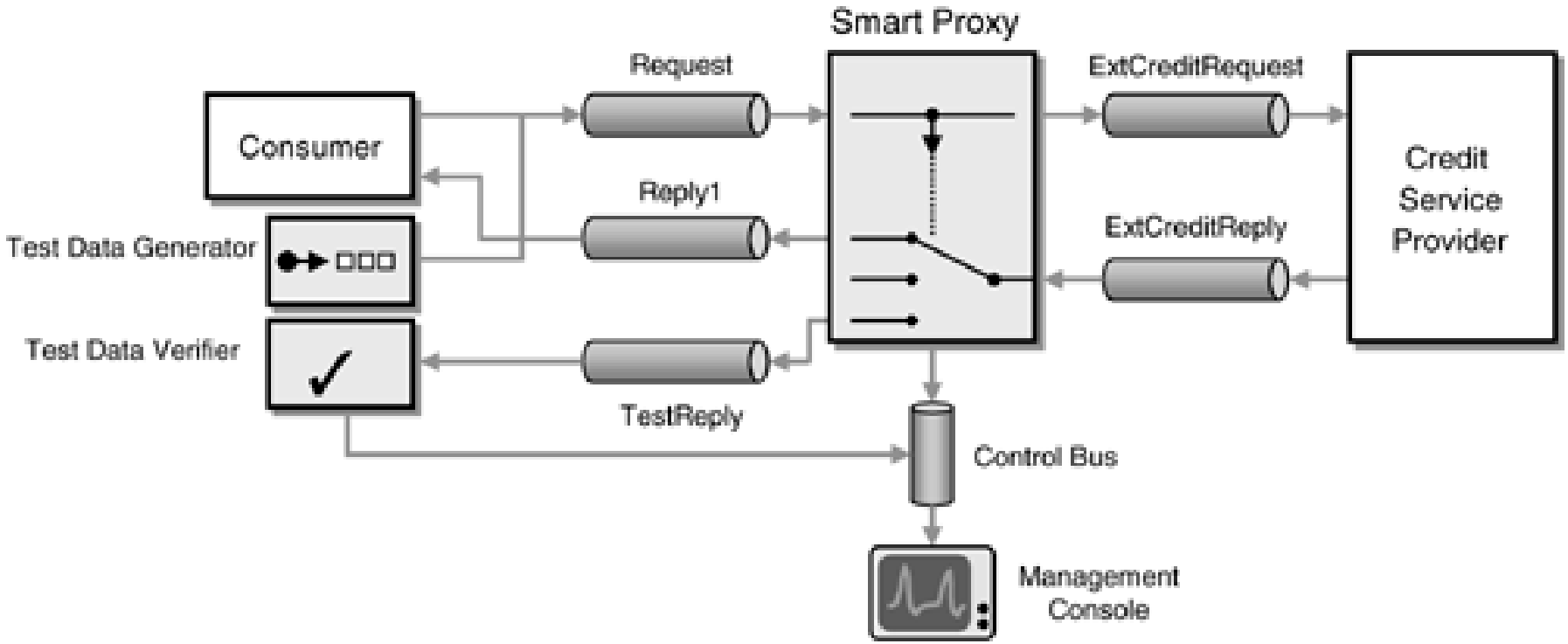
A Smarty Proxy is the place to insert testing and monitoring.



Hohpe and Woolf (2004), p. 35

The Smarty Proxy manages Return Address channels, measures response times, and publishes information to a Control Bus for monitoring at the mangement console.
Monitoring response time is not enough, however.

We also need to check the correctness of the external service.



Hohpe and Woolf (2004), p. 36

We see that integration solution diagrams with text explanations are effective at describing an integration solution elegantly in a vendor-neutral and technology-neutral way.

Outline

So far we have seen Messaging, File Transfer, Shared Database in use.

We also have Remote Procedure Invocation.

Here we'll consider the benefits and limitations of each style. First, what are the criteria we should consider in choosing an integration style?

Integration Styles

Criteria for choosing a style

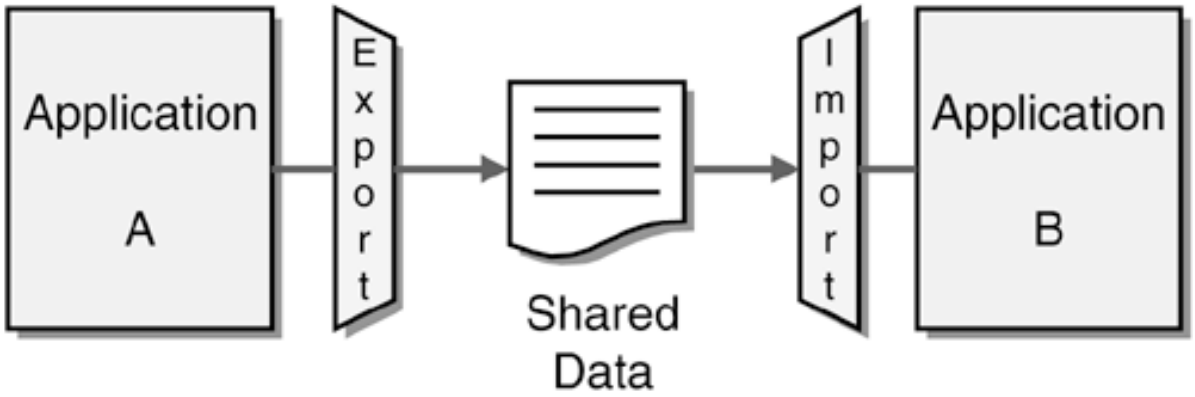
- Coupling: we want to reduce coupling between integrated applications as much as possible.
- Intrusiveness: where possible, we want to avoid modifying applications and avoid writing a lot of glue code.
- Technology selection: we must consider the cost and developer learning curve. (But note that from-scratch integration usually costs more in the long run.)
- Data format: we need a clear view of how difficult it will be to get applications to agree on data formats or introduce appropriate Message Translators.
- Data timeliness: we want to reduce the latency of updates to shared data, but we must also avoid too fine-grained updates.

[continued...]

- Data or functionality: Messaging and Remote Procedure Invocation allow sharing functionality as well as data. Is it worthwhile?
- Remote communication: we want to reduce the frequency of synchronous procedure calls as much as possible due to their high cost.
- Reliability: we need a suitable level of reliability for each application.

In general, we choose the best approach to each problem we face. We can mix strategies within one system as necessary.

File Transfer *has each application produce files that contain the information the other applications must consume. Integrators take the responsibility of transforming files into different formats. We produce the files at regular intervals according to the nature of the business.*



Hohpe and Woolf (2004), p. 44

Integration Styles

File Transfer

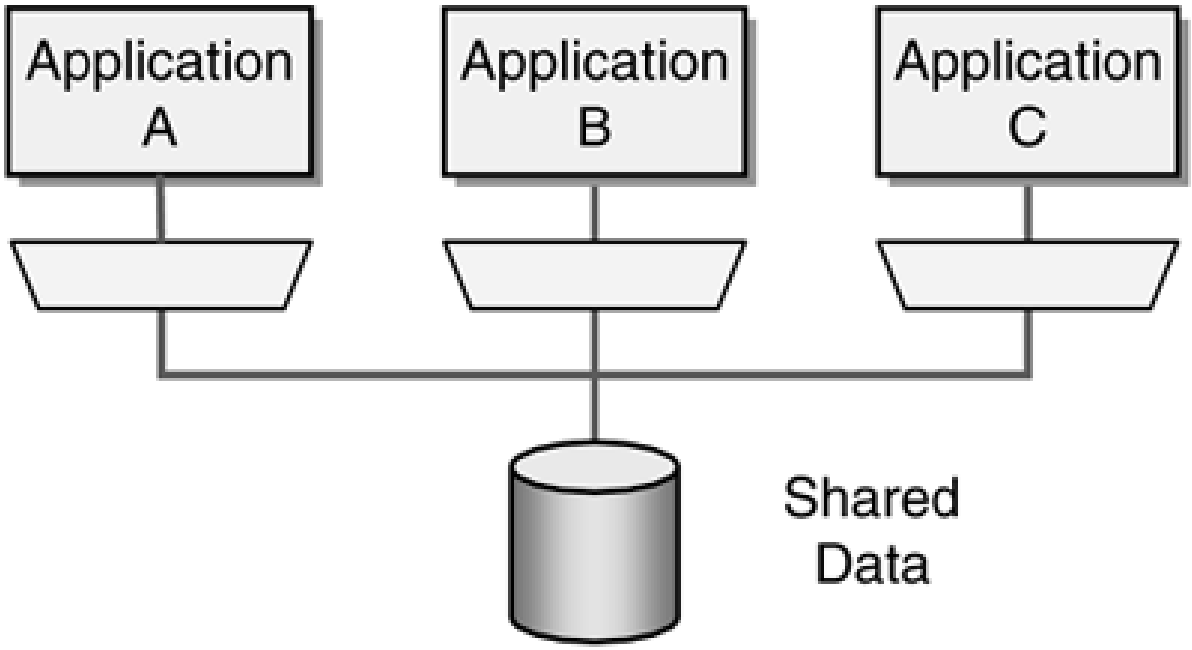
Application A provides the file in some format that is only sometimes negotiated with the integrators.

Integrators write the import functions for application B, C, ...

Main benefits are simplicity and low coupling.

The main problem is that staleness creates inconsistency if updates are infrequent.

Shared Database *integrates applications by having them store their data in a single shared database, and defines the schema of the database to handle all the needs of the different applications.*



Hohpe and Woolf (2004), p. 48

Integration Styles

Shared Database

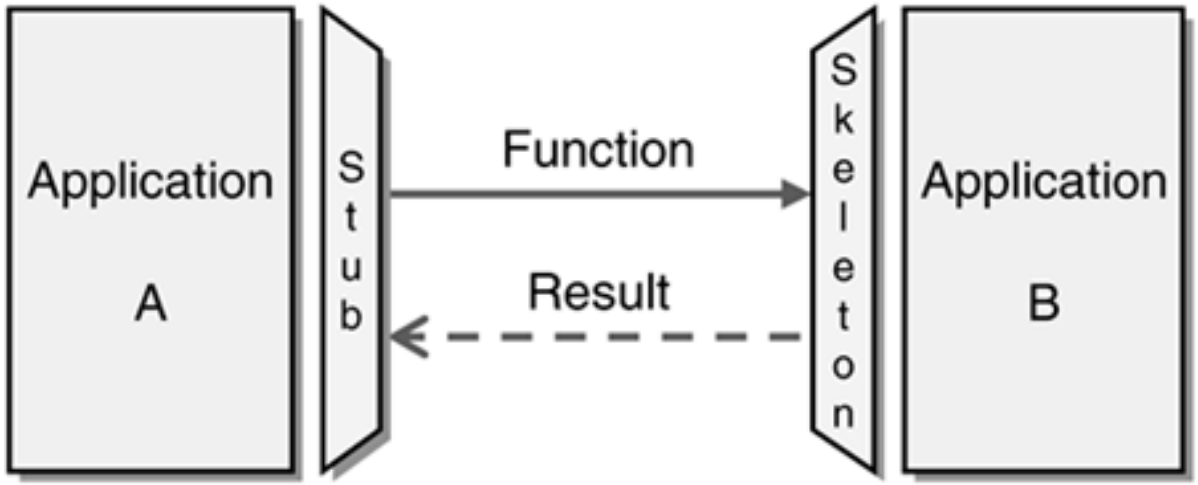
Benefits:

- More timely than File Transfer.
- Forces a shared schema across applications, avoiding semantic dissonance.
- Almost all applications have a RDBMS anyway.

Problems:

- It will be difficult to get departments to agree on a common schema.
- Packaged applications will already have their own schemas.
- Physically distributed databases may be difficult to maintain.
- If too many applications lock resources, performance will be poor.

Remote Procedure Invocation *Develops each application as a large-scale object or component with encapsulated data. It provides an interface to allow other applications to interact with the running application.*



Hohpe and Woolf (2004), p. 51

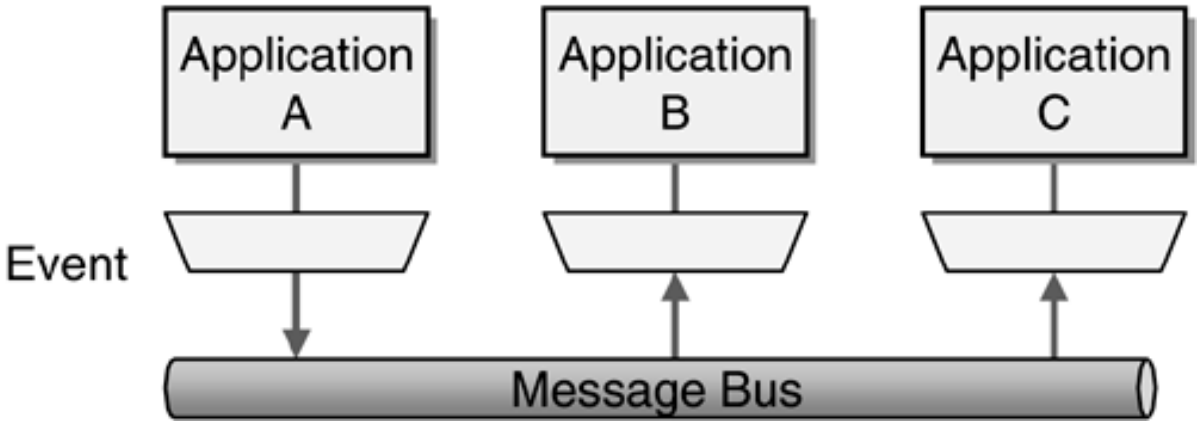
Benefits:

- Easy way to share functionality as well as data.
- Encapsulates data sources, making them easier to change.
- Web service protocols work better than messaging systems over the Internet (no problem with firewalls, for example).
- Easy to provide multiple views of the same data store.

Problems:

- Coupling
- Looks too much like a normal procedure call, encouraging developers to treat them the same.

Messaging *transfers packets of data frequently, immediately, reliably, and asynchronously, using customizable formats.*



Hohpe and Woolf (2004), p. 54

Integration Styles

Messaging

Senders send data to receivers by sending a Message via a Message Channel that connects them.

If the sender doesn't know who to send to, it can use a Message Router.

When sender and receiver disagree on the data format, they can resolve it using a Message Translator.

Benefits:

- Almost as decoupled as File Transfer.
- Nearly as timely as Shared Database.

Problems:

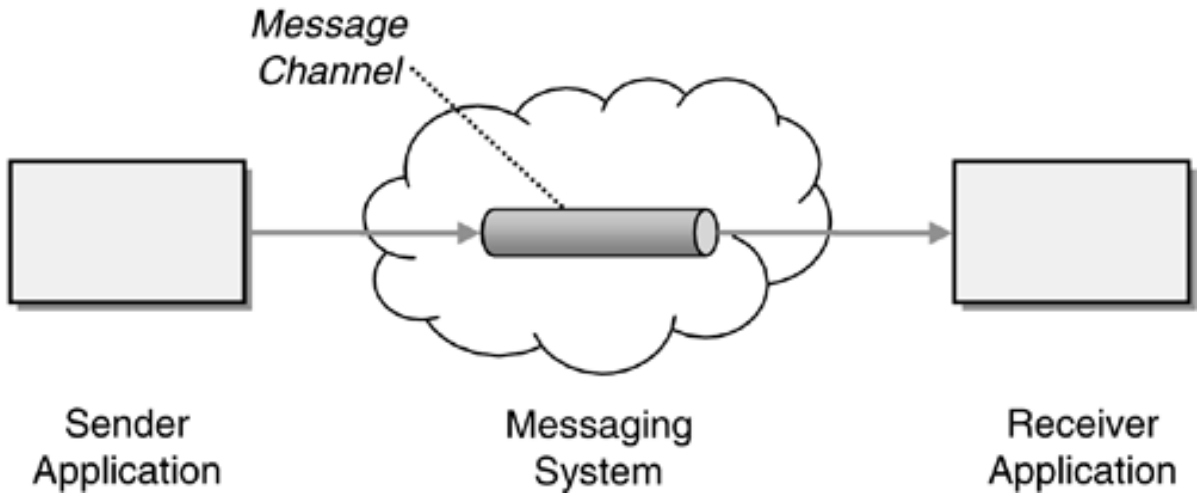
- Semantic dissonance
- Complexity

Outline

Now we'll go into more detail on the basic concepts of messaging systems.

- Channels are virtual pipes.
- Messages are atomic packets of data that the system tries repeatedly to deliver until successful.
- Pipes and Filters put multiple steps of processing together using Channels.
- Message Routers encapsulate rules for deciding what Channel a message should be delivered to, decoupling the sender from the destination.
- Message Translators are flters that translate between message formats.
- Endpoints form the layer of the application that talks to the messaging service.

A Message Channel *connects applications where one application writes information to the channel and the other one reads that information from the channel.*



Hohpe and Woolf (2004), p. 61

Message Channels are dedicated to particular sorts of information, so readers know they will be interested in the contents.

A channel is a logical address in the system.

In most systems, the number and purpose of the channels is fixed at deployment time.

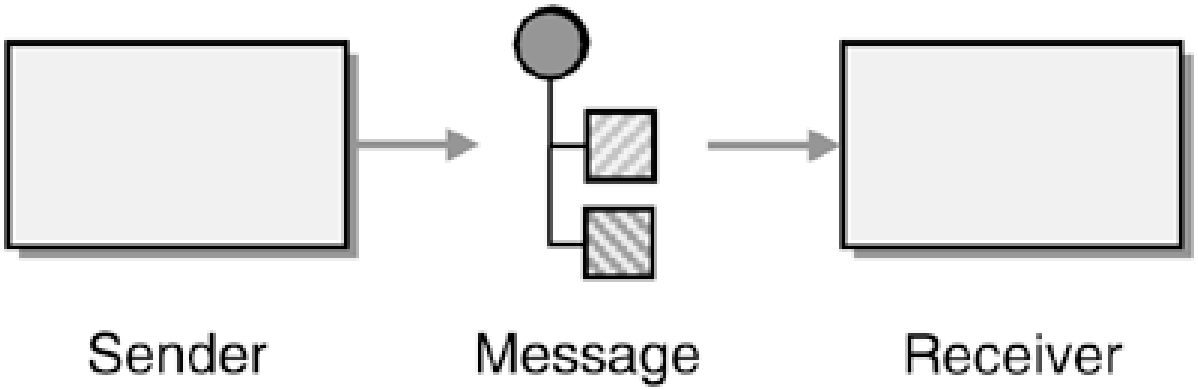
- Calling `createQueue()` or similar usually connecting to a pre-existing queue.

Terminology (producer, consumer, sender, receiver, client, server) varies. We will normally use the term endpoint.

There are many flavors of Message Channel:

- Point-to-Point Channel and Publish-Subscribe Channel.
- Datatype Channels only allow one datatype.
- Selective Consumer allows an endpoint to ignore messages not matching some selection criterion.
- Invalid Message Channel.
- Channel Adapters allow applications not designed for messaging to participate in messaging.
- A Message Bus is a well-designed set of channels acting like a messaging API.

A Message *packages a data record so that the messaging system can transmit it through a* Message Channel.



Hohpe and Woolf (2004), p. 67

Messaging Systems

Message

Message data has to be marshalled (serialized) and unmarshalled (deserialized).

Each message has a header used by the messaging system and a body used by the components.

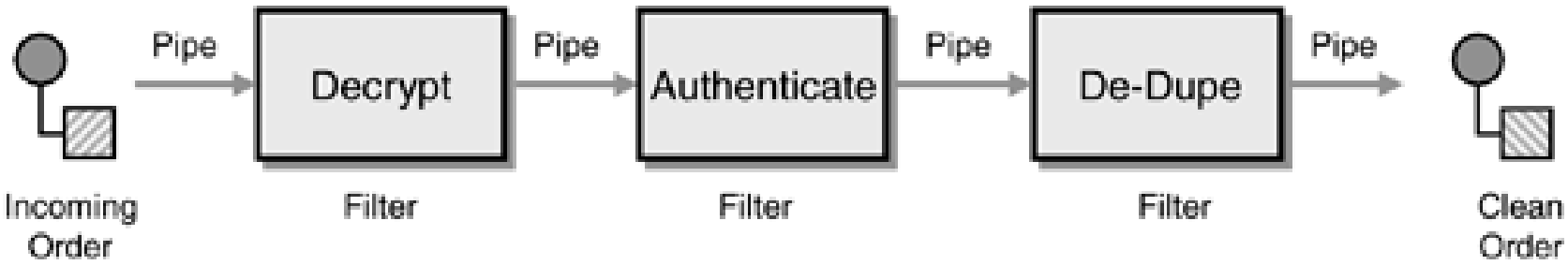
Types:

- A Command Message invokes a procedure.
- A Document Message contains a data set.
- An Event Message notifies receivers of some change.
- Request-Reply indicates there should be a return message.
- Use a Message Sequence when the data is too large.
- If the message is only valid for a length of time, we use Message Expiration.
- When possible, message formats should be specified in terms of a Canonical Data Model.

JMS message types:

- `TextMessage`: a literal string or XML.
- `BytesMessage`: raw bytes.
- `ObjectMessage`: a single serializable object.
- `StreamMessage`: a stream of primitives like `int`.
- `MapMessage`: key-value pairs similar to map. E.g. `getInt("numberOfItems")`.

Pipes and Filters *divides a larger processing task into a sequence of smaller, independent processing steps (filters) that are connected by channels (pipes).*



Hohpe and Woolf (2004), p. 71

Each filter uses the same interface, normally a single input pipe and a single output pipe.

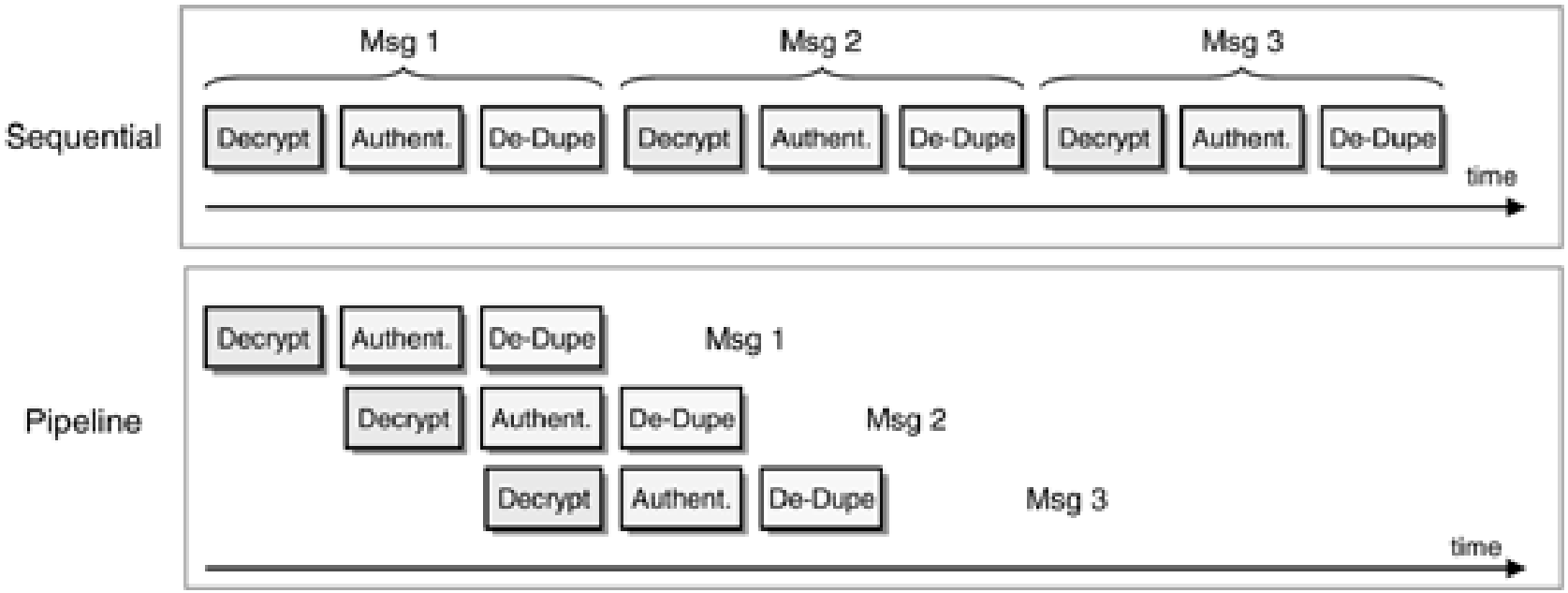
The pipes could be full-blown Message Channels, but oftentimes they will be lightweight in-memory queues.

Benefits: decoupling, reuse, and easy unit testing.

Problems: heavy use of resources if Message Channels are used.

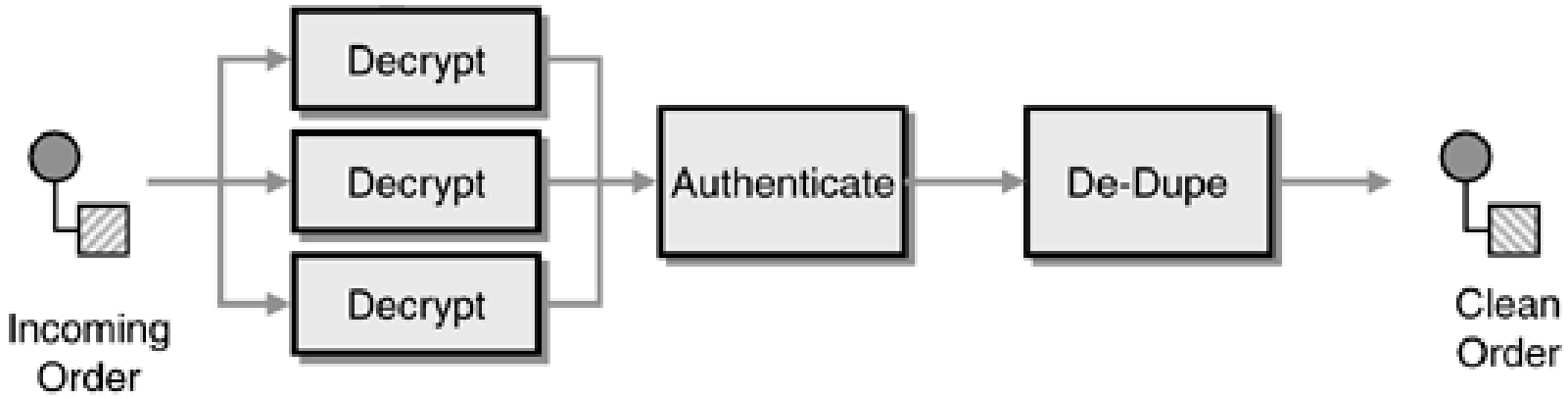
We could relax the single input/single output model to include Message Router.

If each filter runs its own thread and invokes the next filter in line with an asynchronous message, we achieve pipelined behavior:



Hohpe and Woolf (2004), p. 73

But although pipelining achieves parallelism, the pipeline is limited by the slowest filter. We might instead use parallel processing, especially if the filters are stateless:

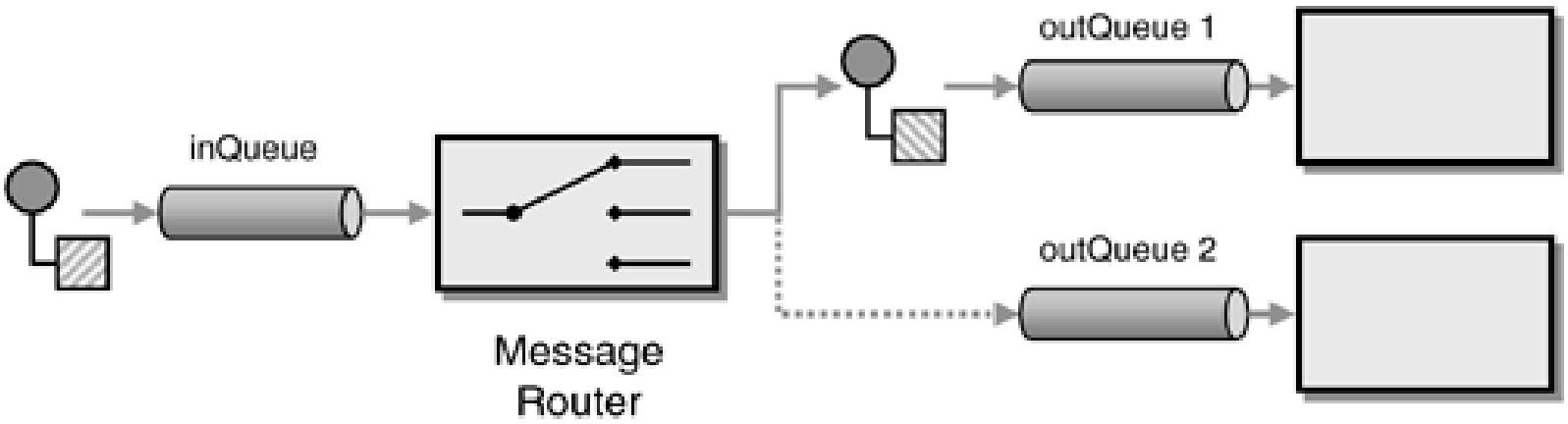


Hohpe and Woolf (2004), p. 74

This would use a Point-to-Point Channel with Competing Customers.

If output sequencing is important, we will need a Resequencer.

A Message Router *consumes a* Message *from one* Message Channel *and republishes it to a different* Message Channel*, depending on a set of conditions.*



Hohpe and Woolf (2004), p. 80

If a sending application has to decide which Message Channel to publish to, it is not fully decoupled from the receiver.

Pipes and Filters can help here, but it assumes the same process will be applied to every message.

Message Routers do not modify messages.

If destinations are changing frequently:

- Don't use Message Router (predictive routing).
- Use a Publish-Subscribe Channel and mulitple Message Filters (reactive routing).

Sometimes Message Routers make it difficult to understand the message flow in a system.

The simplest routing strategy is fixed, but this is rarely useful.

Content-Based Routers are common. They use properties of a message to decide where to deliver.

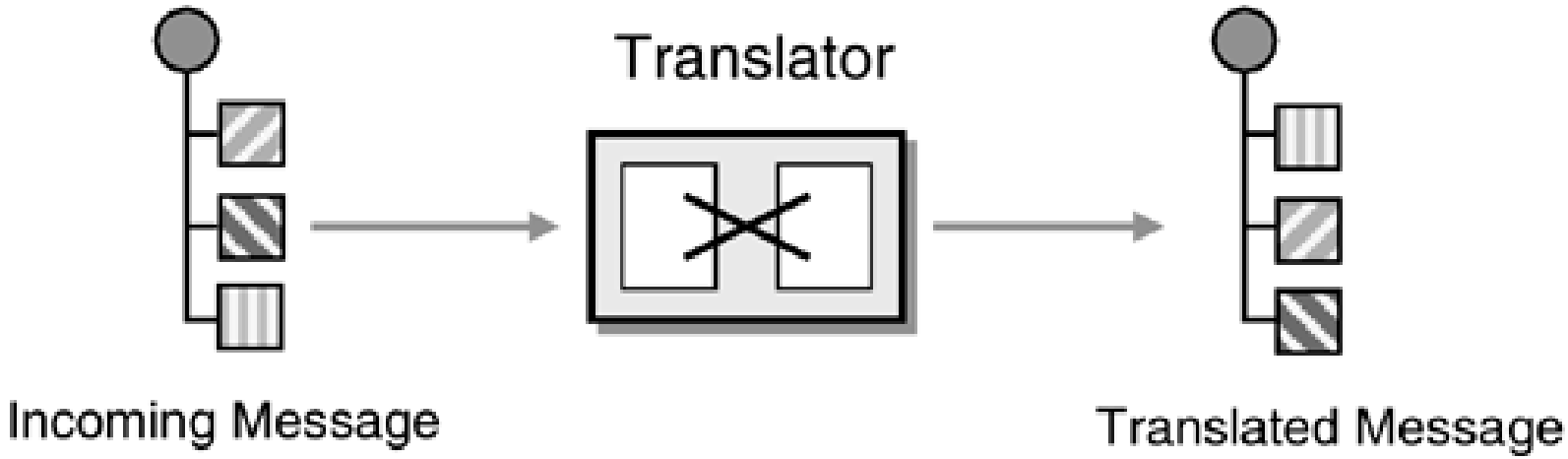
Routers are usually stateless, but some (e.g. a de-duper) are stateful.

Most routers use hard coded logic, but it can also connect to a Control Bus for configuration changes.

A Dynamic Router configures itself based on control messages from the recipients.

Commercial EAI Message Brokers do validation, transformation, and routing.

A Message Translator *is a filter sitting between other filters or applications to translate one data format into another.*



Hohpe and Woolf (2004), p. 86

We saw that Message Channel decouples senders and receivers.

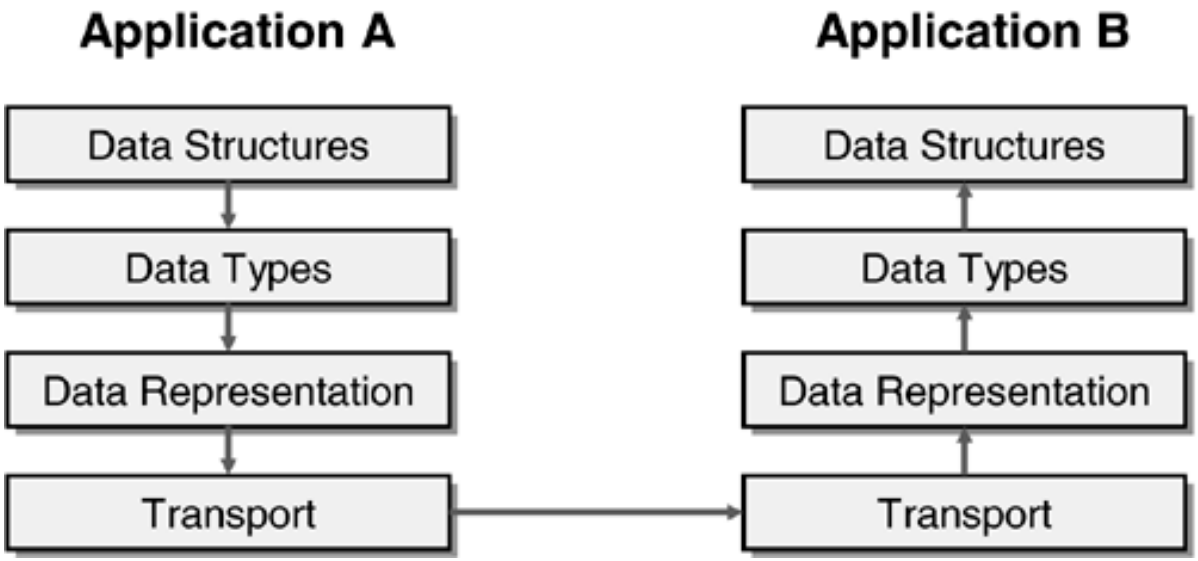
We saw that Message Router decouples senders from the destination channels.

Message Translators work at multiple layers of messaging: data structures, data types, data representations, and transport.

Layer	Deals With	Transformation Needs (Example)	Tools/Techniques
Data Structures (Application Layer)	Entities, associations, cardinality	Condense many-to-many relationship into aggregation.	Structural mapping patterns, custom code
Data Types	Field names, data types, value domains, constraints, code values	Convert ZIP code from numeric to string. Concatenate First Name and Last Name fields to single Name field. Replace U.S. state name with two-character code.	EAI visual transformation editors, XSL, database lookups, custom code

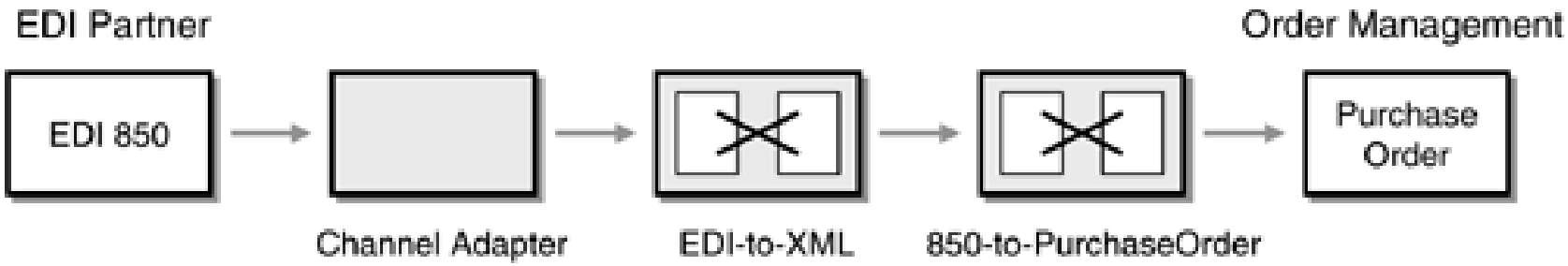
Layer	Deals With	Transformation Needs (Example)	Tools/Techniques
Data Representation	Data formats (XML, name-value pairs, fixed-length data fields, EAI vendor formats, etc.). Character sets (ASCII, UniCode, EBCDIC). Encryption/compression.	Parse data representation and render in a different format. Decrypt/encrypt as necessary.	XML parsers, EAI parser/renderer tools, custom APIs
Transport	Communications protocols: TCP/IP sockets, HTTP, SOAP, JMS, TIBCO RendezVous	Move data across protocols without affecting message content.	Channel Adapter, EAI adapters

It is possible to separate complex transformations by layers:



Hohpe and Woolf (2004), p. 89

Example: transforming from a standard Electronic Data Interchange format for a Purchase Order to an application's Purchase Order class.



Hohpe and Woolf (2004), p. 90

Transport: File Transfer Protocol to HTTP.
Representation: fixed field to XML.
Fields and structure: from the standard format to the object's format.

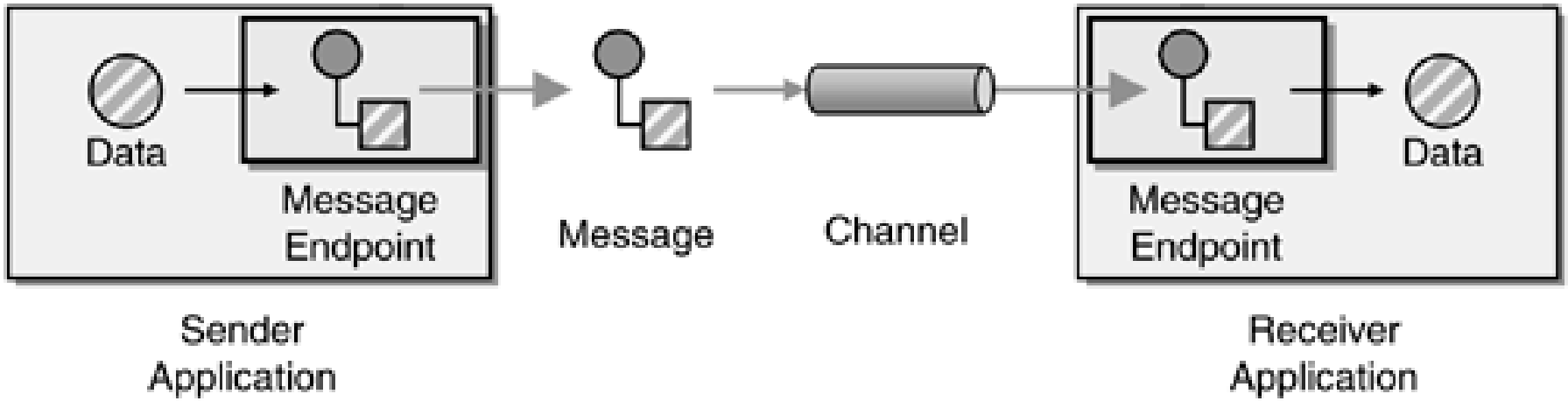
A few special types of of translators:

- Envelope Wrapper: converts a data structure into a message.
- Content Enricher: adds information to a message.
- Content Filter: removes information from a message.
- Claim Check: stores partial information from a message for later retrieval.
- Normaliser: makes a message consistent according to some criteria.

Translations can be done programmatically or with a transform technology like XSL. EAI products often contain visual XSL editors.

Clients of a messaging system want to use the specific messaging API.

A Message Endpoint *is a client of the messaging system that connects an application to a* Message Channel *that application can then use to send or receive* Messages.



Hohpe and Woolf (2004), p. 96

Message Endpoint is a special case of Channel Adapter in which the adapter is integrated with the application.

A well-designed endpoint is a Messaging Gateway that encapsulates the details of the messaging API from the rest of the application.

If the messages require transactions, we explicitly control them using Transactional Clients.

Receiver endpoints can be Polling Consumers, Event-Driven Consumers, Competing Consumers, Message Dispatchers, Selective Consumers, or Durable Subscribers.

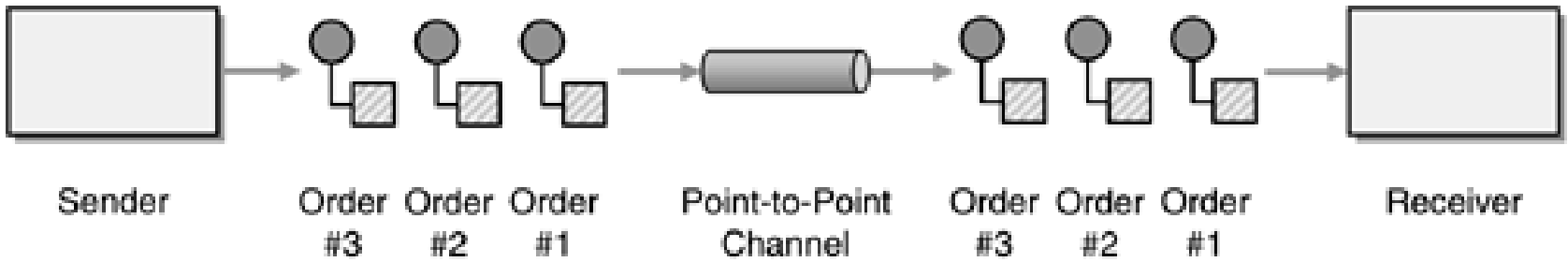
Outline

Next we consider the various types of Message Channel in detail.

Keep in mind that in most cases, the channels available need to be pre-defined.

Normally channels will be unidirectional. Two-way conversations should use Request-Reply.

A Point-to-Point Channel *ensures that only one receiver will receive a particular message.*



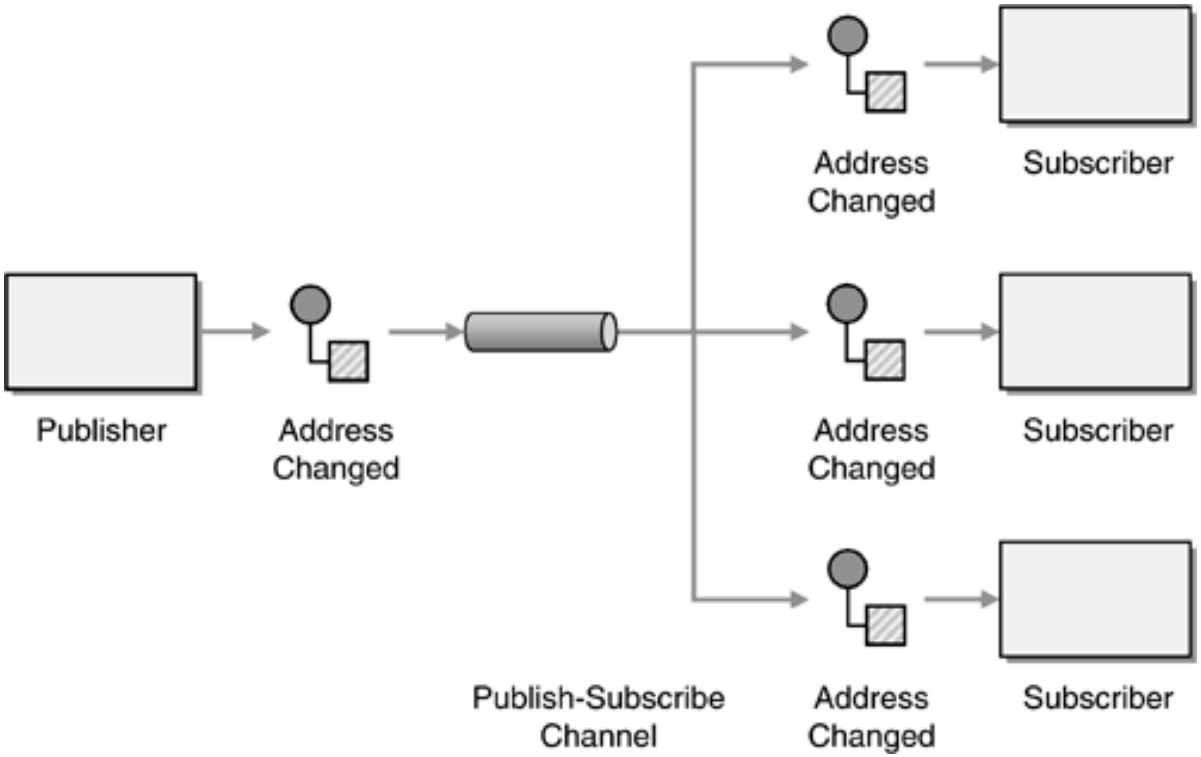
Hohpe and Woolf (2004), p. 103

When we attach multiple consumers to a Point-to-Point Channel, we have Competing Consumers.

To implement RPC, we can combine two Point-to-Point Channels with Request-Reply, a Command Message, and a Document Message.

Example in a stock trading application: requesting a stock trade.

A Publish-Subscribe Channel *delivers a copy of a particular event to each receiver.*



Hohpe and Woolf (2004), p. 107

Each subscriber of a Publish-Subscribe Channel receives each message exactly once.

We can think of each subscriber as having its own output channel.

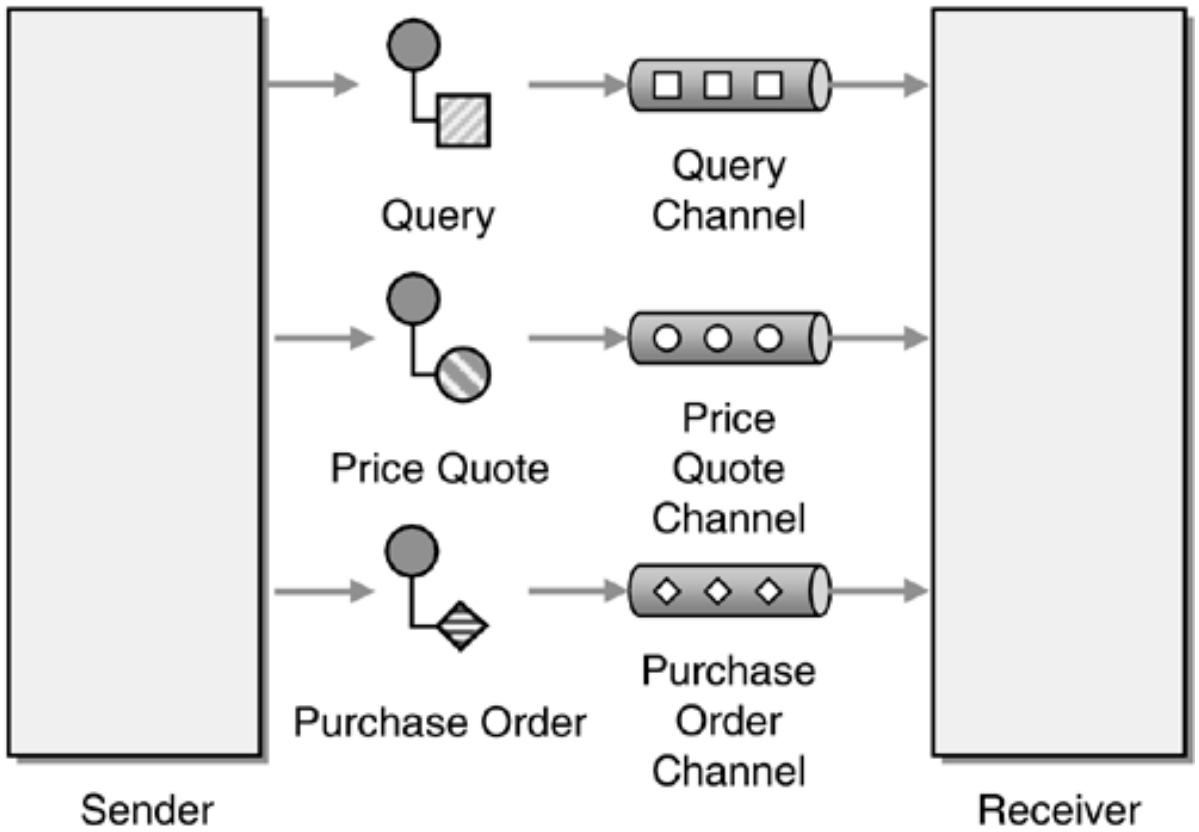
Subscriptions may be durable (Durable Subscriber) or non-durable.

When notifications need acknowledgment, use with Request-Reply.

Message Expiration is useful to avoid accumulation of messages for inactive subscribers.

Stock trading example: we might use a Publish-Subscribe Channel to get notifications of stock quotes and stock trades.

A Datatype Channel *is a channel in which all data is of the same type.*



Hohpe and Woolf (2004), p. 112

Motivation for Datatype Channel: the consumer is simplified if it knows what type of message to expect.

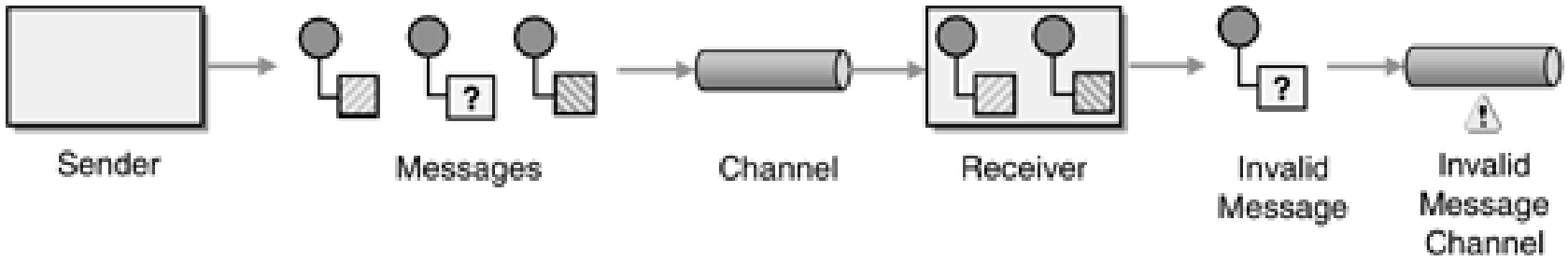
Alternatives include Selective Consumer or checking a Format Indicator in a case statement.

If we have a multiplexed channel and we want a Datatype Channel, we can apply Content-Based Router.

A related concept is a quality-of-service channel, in which we dedicate multiple channels to the same data type but use a different quality of service for the channels.

Example: quote requests, trade requests, and changes of address might all use separate Datatype Channels.

An Invalid Message Channel *is a special channel for messages that could not be processed by their receivers.*



Hohpe and Woolf (2004), p. 116

Useful when a message has an unrecognized format, when expected fields are missing, and so on.

An Invalid Message Channel acts like a log for the messaging system.

Note that invalid messages are different from invalid application requests.

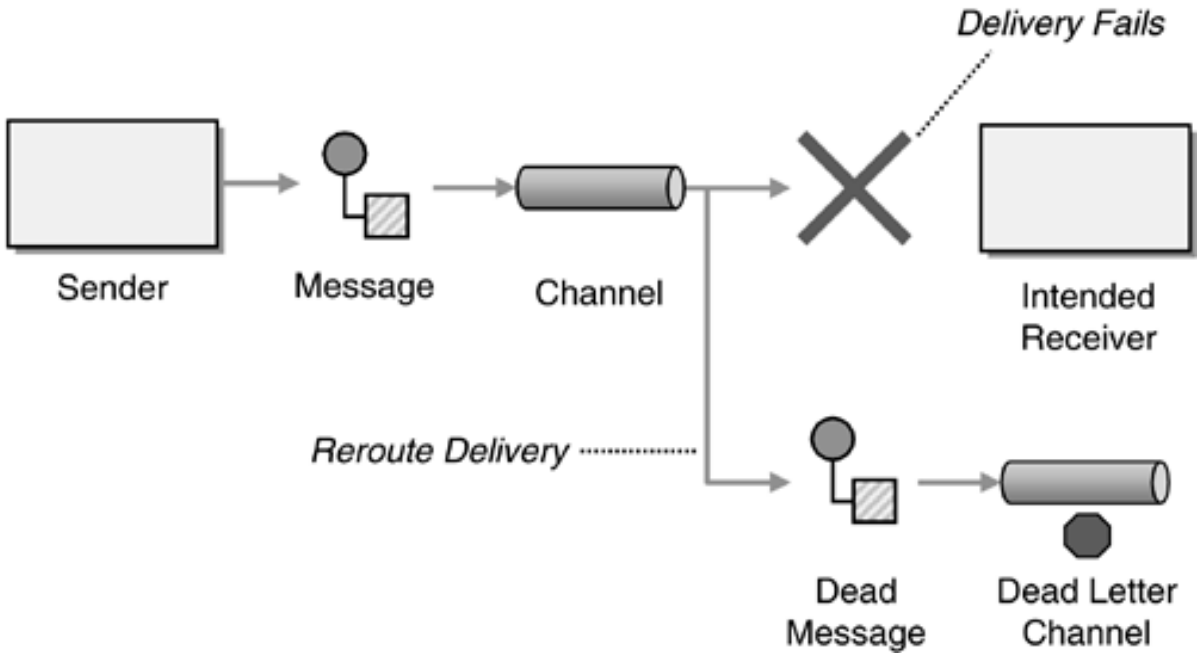
Example: a request to delete a non-existent record is an application error, not an invalid message.

Invalid messages should be handled in the Message Gateway; application errors should be handled by the application.

Obviously, someone should monitor the Invalid Message Channels and take action.

Example: a request for a stock quote or a stock trade, without the specifiying the name of the stock, would be an invalid message.

A Dead Letter Channel *is a channel used by the messaging system for messages it cannot or should not deliver.*



Hohpe and Woolf (2004), p. 120

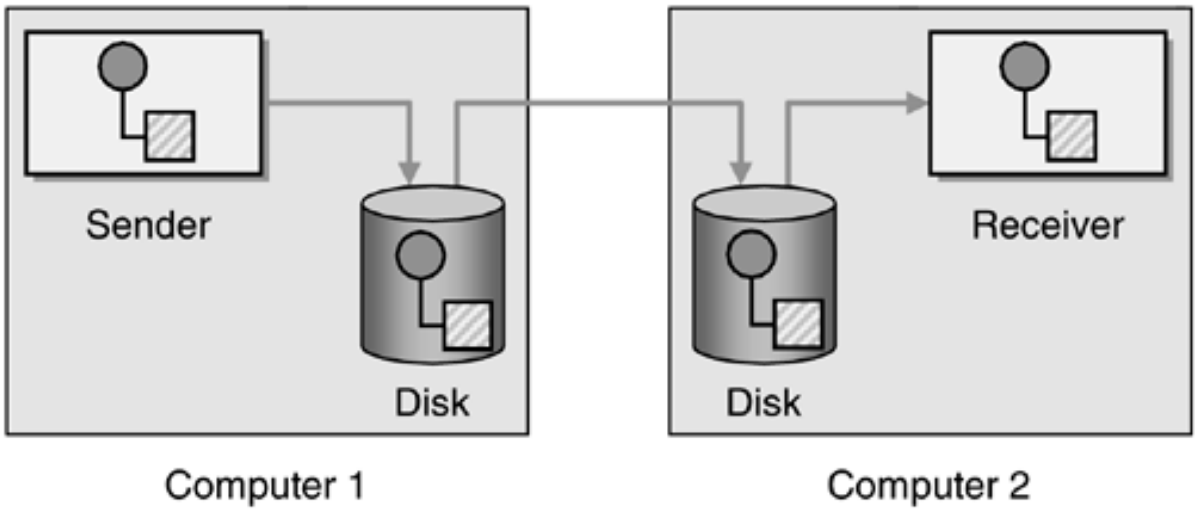
Dead Letter Channels are useful when a message channel is deleted while messages in the channel are yet undelivered.

Other undeliverable messages include expired messages or messages whose type is not processed by any Selective Consumer.

Note the difference with Invalid Message Channel: invalid messages are delivered but don't make sense to the application; dead messages are never delivered.

Stock trading example: a trade request should have a Message Expiration, after which messages should be moved to a Dead Letter Channel if not received.

Guaranteed Delivery *makes messages persistent so that they are not lost even if the messaging system crashes.*



Hohpe and Woolf (2004), p. 123

Store-and-forward is a key feature of messaging systems.

When we store, usually it is in memory.

If we want undelivered messages to survive crashes, we use Guaranteed Delivery.

During transmission, a message is never deleted until it is successfully stored at the receiving end.

Note that Guaranteed Delivery hurts performance.

Note that in IT we can never guarantee with 100% certainty!

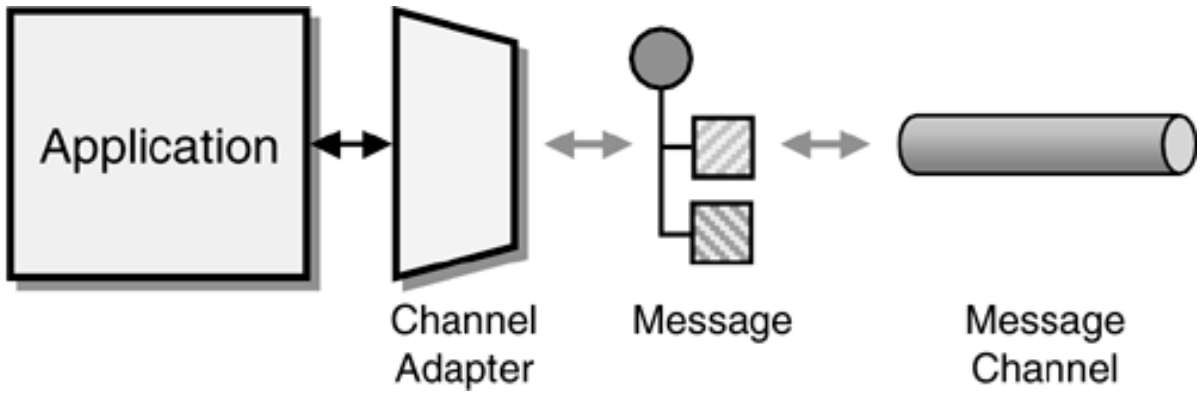
JMS uses Guaranteed Delivery by default, except that Publish-Subscribe Channel subscribers must be active to receive messages.

For guaranteed delivery to offline pub-sub subscribers in JMS, use Durable Subscriber.

JMS persistence and delivery can be configured on a per-message or per-producer basis.

Stock example: we might guarantee delivery for trades, confirmations, and changes of address, but not price change notifications. If some subscribers want guaranteed price change notifications, we would use the quality of service channel strategy.

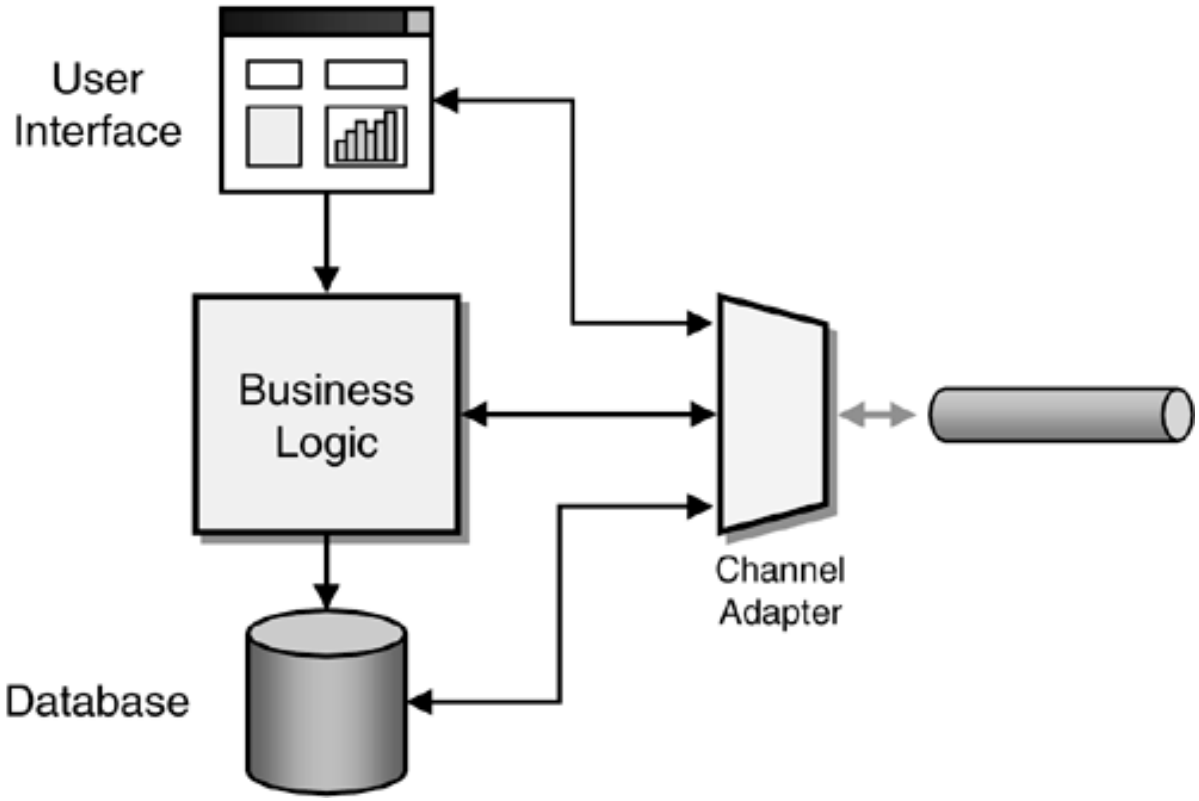
A Channel Adapter accesses the application's API or data to publish messages on a channel based on this data and likewise can receive messages and invoke functionality inside the application.



Hohpe and Woolf (2004), p. 128

A Channel Adapter is a client of the messaging system that listens to internal events and invokes the messaging system in response.

Channel Adapters work at one of three levels:



Hohpe and Woolf (2004), p. 128

- User interface adapters extract information from returned Web pages or dumb terminal screens. Brittle and slow.
- Business logic adapters access an API, e.g. through an EJB component or through a C++ library. This is always best if the API is stable and well-defined.
- Database adapters connect directly to the application's database. Simple. Best for read-only transactions, since updates could be dangerous. Can use triggers. Private schemas may change.

If an adapter is unidirectional (e.g. a Web screen scraper), we have to poll to find out about events. This is inefficient.

Channel Adapter is often combined with Transactional Client to make application and messaging operations atomic.

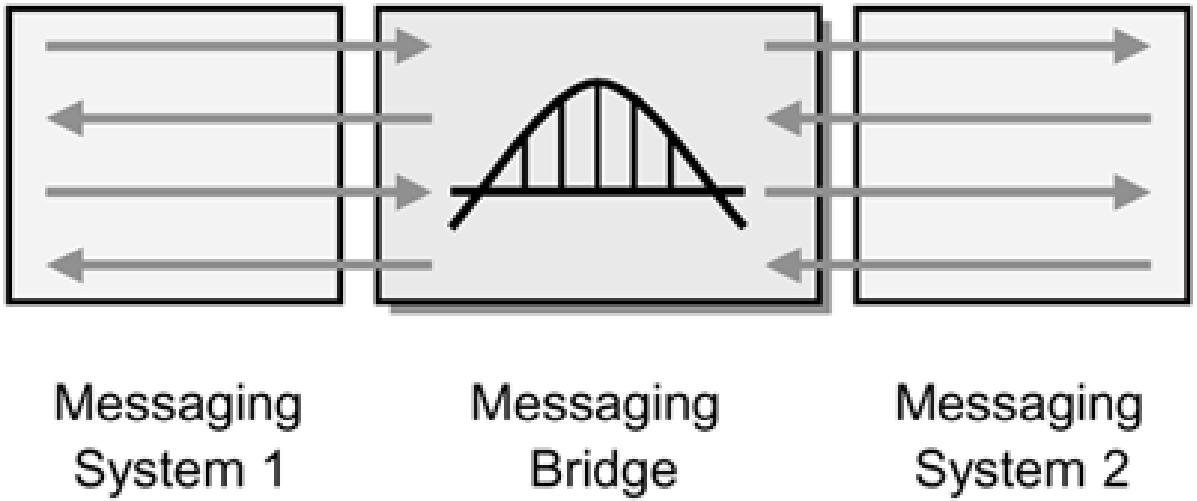
Many EAI vendors provide channel adapters for major applications.

A common pattern is a Channel Adapter to move SOAP Web service messages between HTTP and a messaging system.

Stock trading example: if we want to log all trade messages to a RDBMS, we would use a Channel Adapter to connect to the database.

We might use a UI Channel Adapter to extract stock quotes from the Web.

A Messaging Bridge *connects messaging systems by replicating messages between them.*



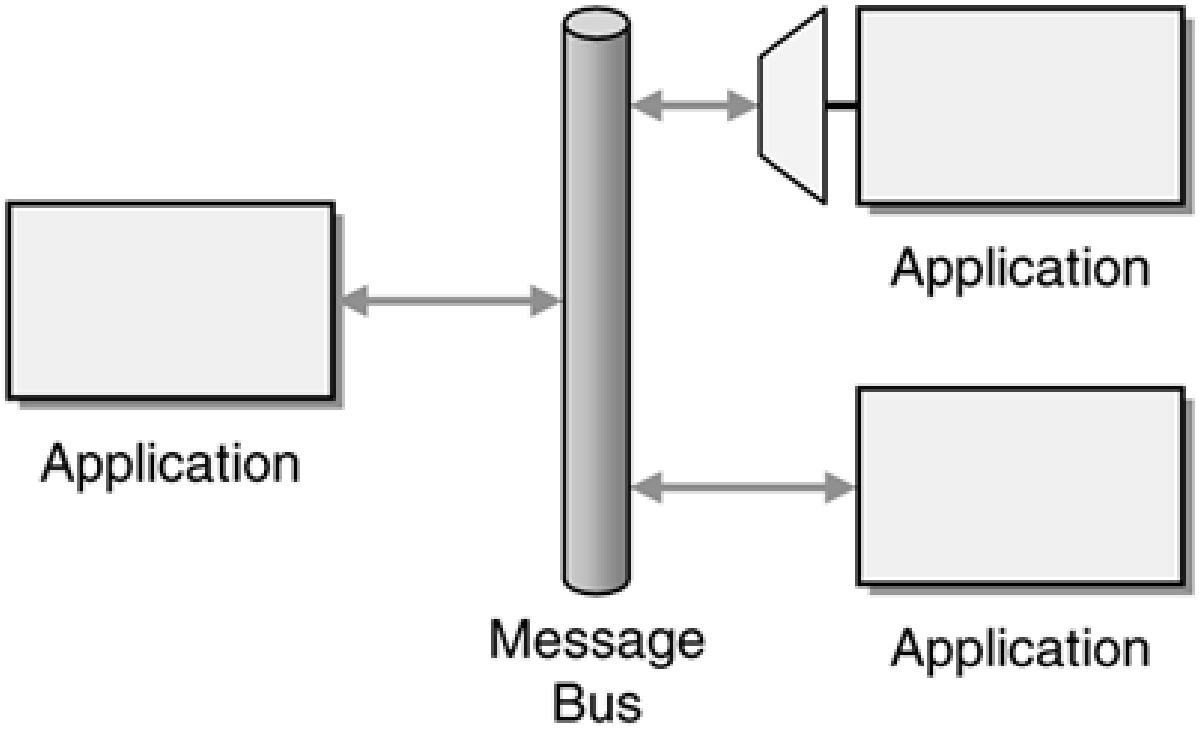
Hohpe and Woolf (2004), p. 134

A Messaging Bridge uses pairs of Channel Adapters to move messages between two corresponding channels managed by two different messaging systems.

Even with JMS compliance, different vendors' implementations may not be able to interoperate.

The need for messaging bridges is common when two companies merge or when formerly independent divisions' IT systems need to start communicating.

A Message Bus *structures the connecting middleware between applications to enable them to work together using messaging.*



Hohpe and Woolf (2004), p. 138

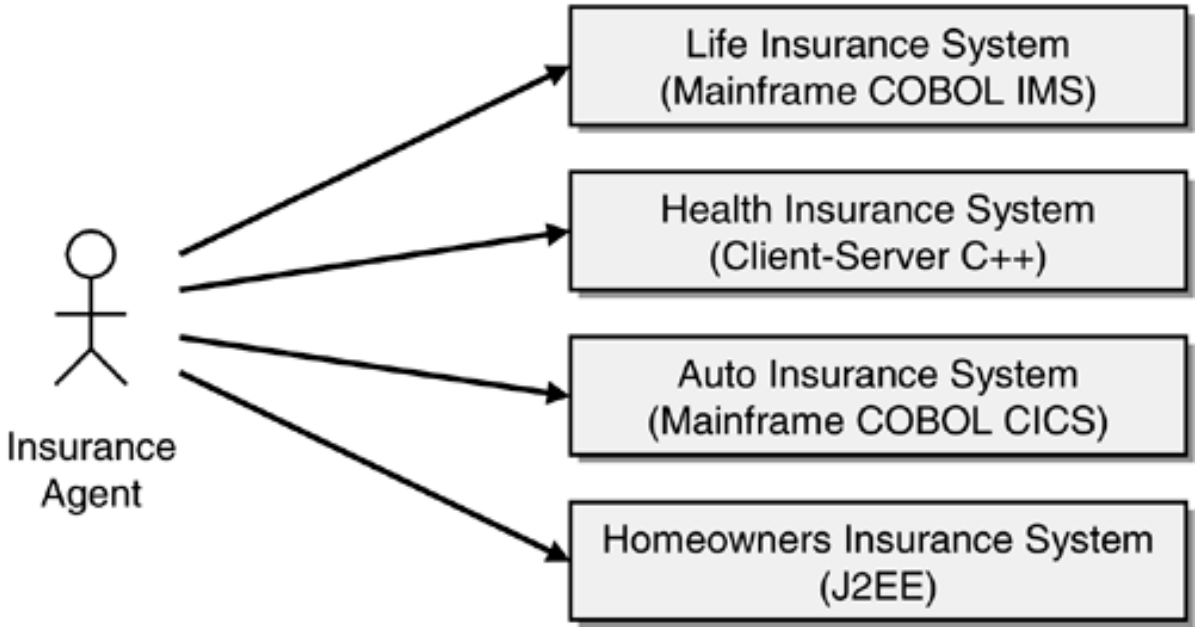
A Message Bus combines a Canonical Data Model, a common command set, and messaging.

The bus infrastructure will often use shared Message Routers and use Publish-Subscribe Channels so that anyone can receive messages.

Each application will use one or more Channel Adapters to connect to the bus.

The shared command set would be used by the Message Routers to determine where messages should go.

Example: imagine an insurance agent that needs to work with many different systems within the enterprise:

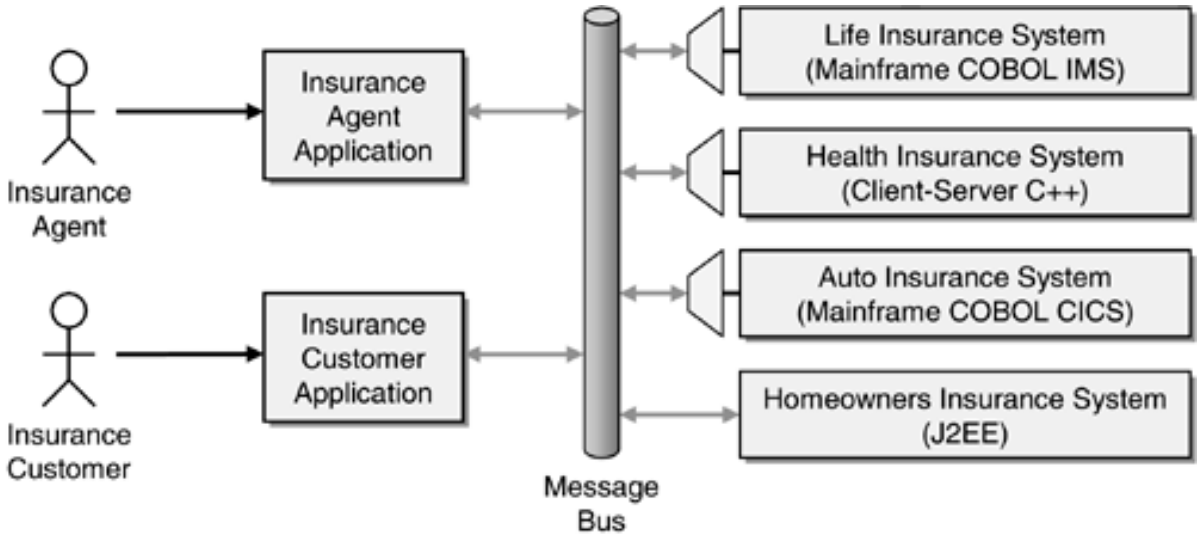


Hohpe and Woolf (2004), p. 137

Rather than creating a new application with point-to-point integration with the other applications, we develop a Canonical Data Model and connect each application to the Message Bus (next page).

Messaging Channels

Message Bus



Hohpe and Woolf (2004), p. 139

Now end-user applications become presentation layers that execute system operations by publishing Command Messages on the bus.

A Message Bus thus allows the construction of an enterprise-wide service oriented architecture.

Outline

Messages

Introduction

What kind of message should we send, and what do we need to put in it?

The decisions depend on what we want:

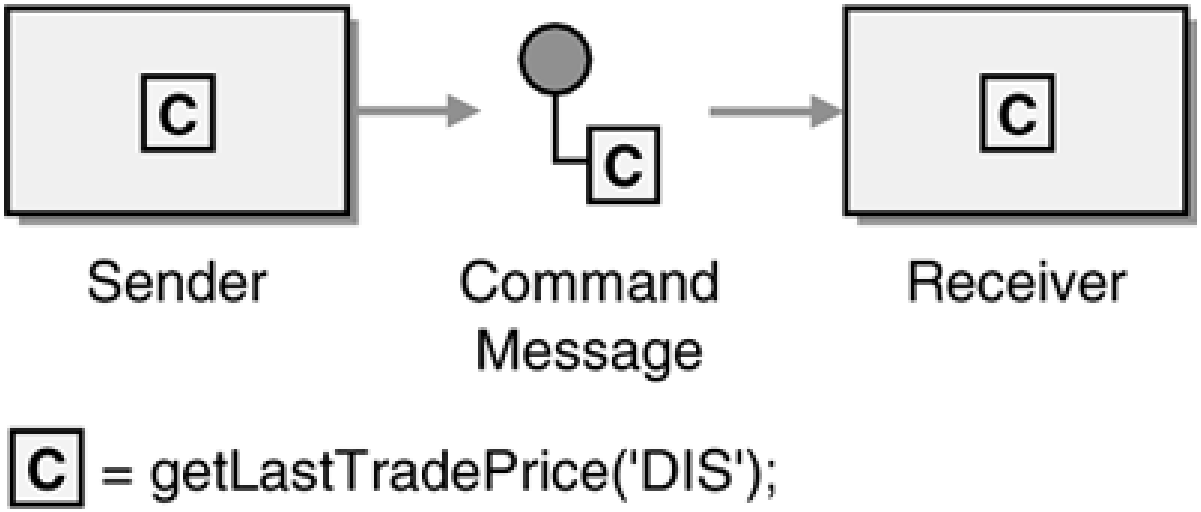
- Use a Command Message to invoke a function.
- Use a Document Message to transmit a data structure.
- Use Event Messages for event notification.

When a reply is required, use Request-Reply with a Return Address and a Correlation Identifier.

When the data is too large, use a Message Sequence.

When messages are no longer relevant after a period of time, use Message Expiration.

Use a Command Message *to reliably invoke a procedure in another application.*



Hohpe and Woolf (2004), p. 145

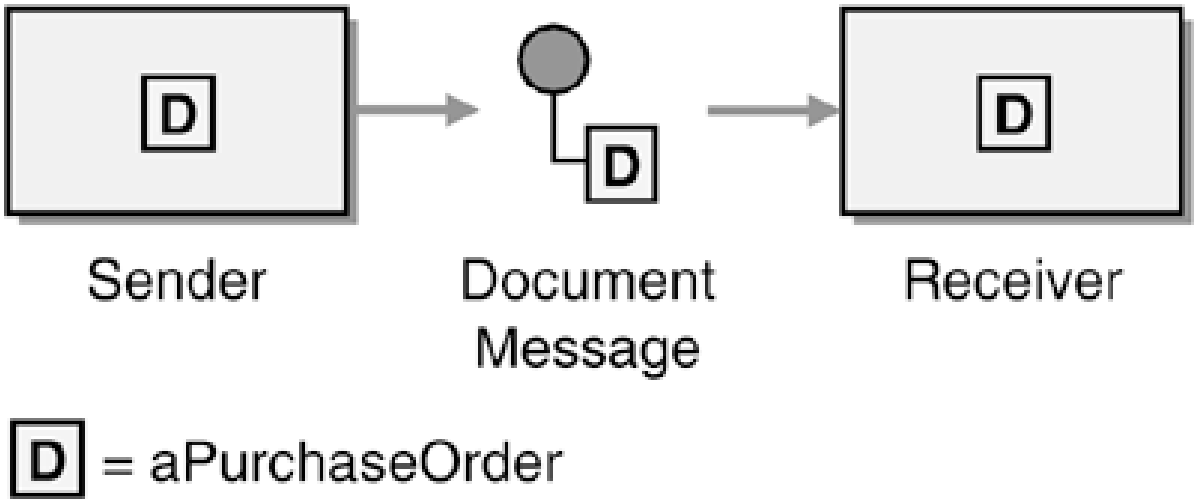
The idea is we want to invoke a procedure as in RPC but we want to do so asynchronously using messaging.

The GoF Command pattern shows how to turn a request into an object.

In JMS, any type of message, e.g. an `ObjectMessage` could wrap a command object, or a `TextMessage` could represent an XML description of a command and its arguments.

A SOAP request is another example of a command.

Use a Document Message *to reliably transfer a data structure between applications.*



Hohpe and Woolf (2004), p. 148

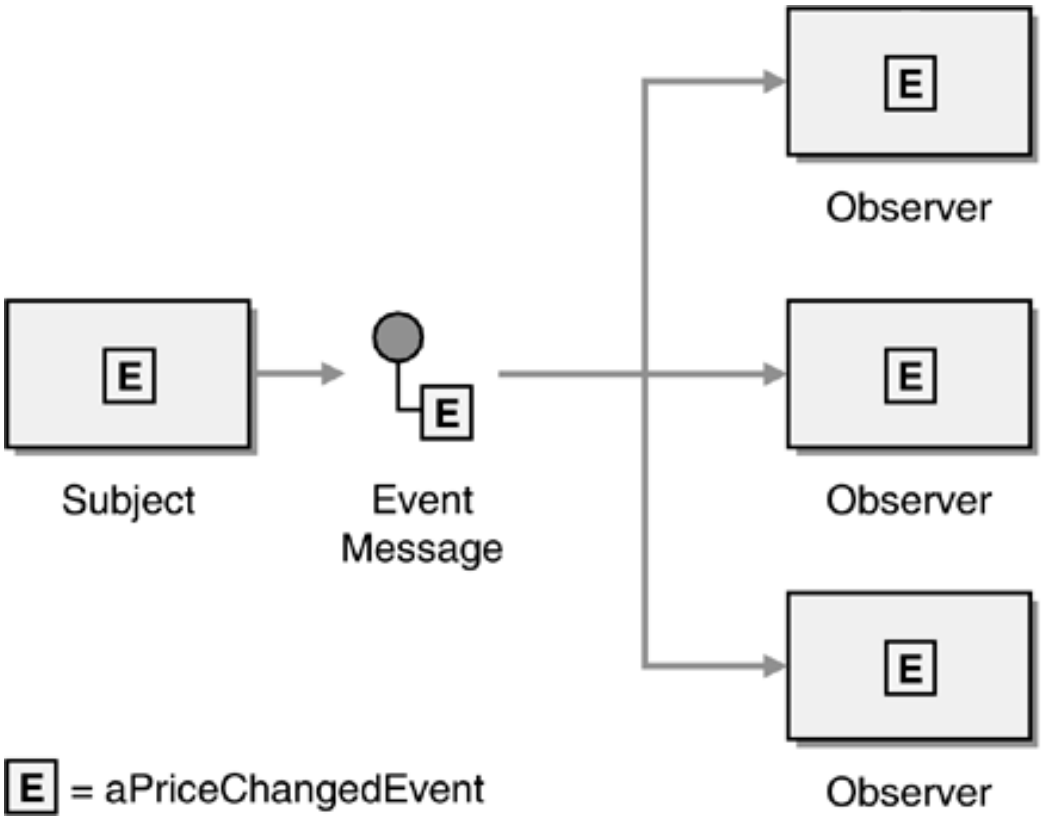
A Document Message simply contains an arbitrary data structure.

The pattern is similar to Event Message, but:

- The goal is the transfer of data, not informing of an event.
- Often used with Guaranteed Delivery.
- Usually not used with Message Expiration.
- Usually used with Point-to-Point Channel.

Could be any type, e.g. an `ObjectMessage` could contain an arbitrary serialized data structure, or a `TextMessage` could contain an arbitrary data structure in XML format.

Use an Event Message *for reliable, asynchronous event notification between applications.*



Hohpe and Woolf (2004), p. 152

Model-View Controller is a classic example of where Event Message is useful.

Example: prices changing in an e-commerce application.

Sender creates an event object then wraps in a message and publishes it.

Compared to Document Message:

- Contents are less important; timing is more important.
- Guaranteed Delivery is not very useful.
- Often used with Message Expiration to guarantee quick delivery or no delivery.
- Not usually used with Durable Subscriber.
- Usually used with Publish-Subscribe Channel.

Messages

Event Message

There are two main styles of event notification:

- In the push model we put the updates (e.g. new prices and product info) in the Event Message.
- In the pull model we send minimal information and let observers request additional information as necessary.

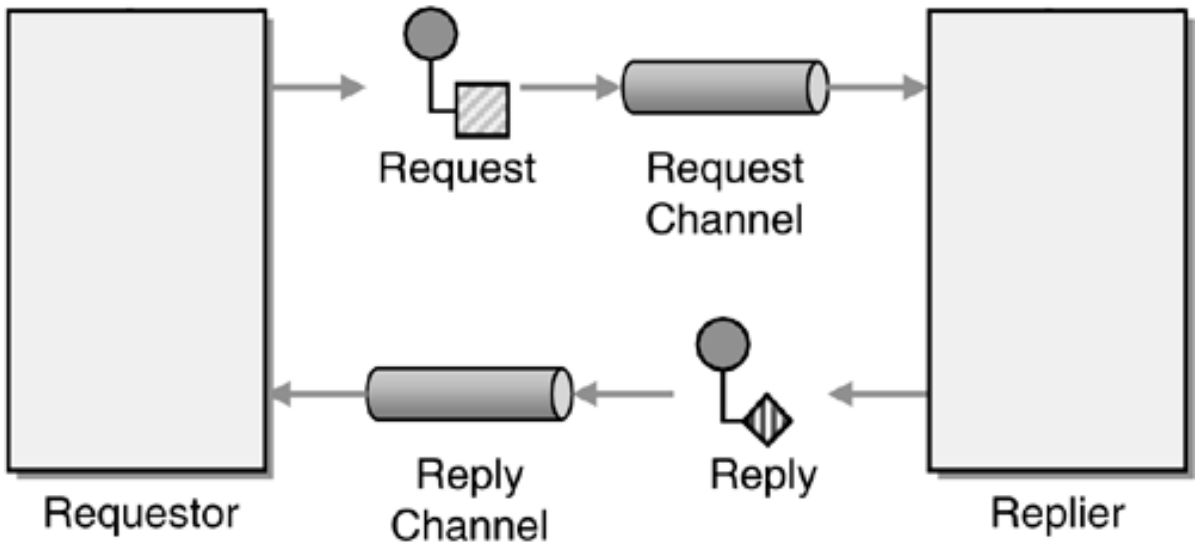
A pull-model event notification is comprised of three messages:

- Update: a pure Event Message sent to every observer.
- State request: a Command Message observers use to fetch state information.
- State reply: a Document Message the subject uses to send details to an observer.

[Think about advantages and disadvantages of the two main approaches.]

Oftentimes we need to perform two-way communication over one-way message channels.

Send a pair of Request-Reply *messages, each on its own channel.*



Hohpe and Woolf (2004), p. 155

Messages

Request-Reply

The requestor sends a request message and waits for a reply from the replier, who responds with a reply message.

Request channels could be Point-to-Point Channels or Publish-Subscribe Channels, but response channels are almost always point-to-point.

There are two basic ways for the requestor to wait:

- Synchronous block: a single thread sends, blocks for the reply, processes it. Simple, but what about crash recovery and the number of threads required?
- Asynchronous callback: caller thread sends, separate thread blocks for any reply message and invokes callbacks.

Messages

Request-Reply

There are three basic types of Request-Reply interactions:

- Messaging RPC: request is a Command Message; reply is a Document Message.
- Messaging query: a remote query. Request is a Command Message; reply is often a Message Sequence.
- Notify/acknowledge: request is an Event Message; reply could be a Document Message or a Command Message for events following a pull model.

And there are three types of replies:

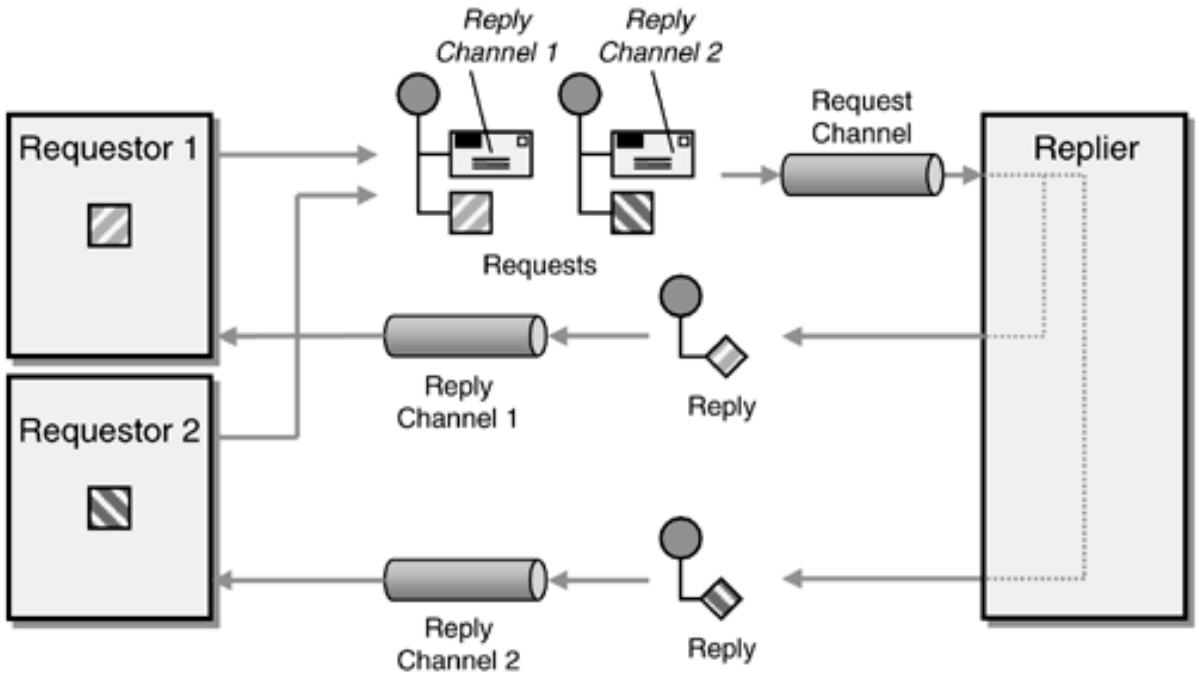
- Void: a simple notification that the request was received.
- Result value: a single return value object.
- Exception: an indication that the request was aborted.

The request will need a Return Address, and the reply may need a Correlation Identifier.

JMS implementation:

- We can dynamically create a `TemporaryQueue` or `TemporaryTopic` that lives for the lifetime of the corresponding JMS `Connection`.
- In the request message, we set the `reply-to` attribute.
- Alternatively, `QueueRequestor` creates a temporary channel, sets the `reply-to` attribute, sends the message, and blocks for the response.
- The main disadvantage of temporary queues is they do not support Guaranteed Delivery — we have to roll our own asynchronous callback listener (perhaps a message-driven bean in EJB) and use a permanent channel.

The request message should contain a Return Address *that indicates where to send the reply message.*



Hohpe and Woolf (2004), p. 160

Messages

Return Address

We want to avoid hard coding the reply channel.

Also, we sometimes want the reply receiver to be different from the requestor.

Return Address allows us to decouple the requestor from the reply receiver.

SOAP does not support decoupled reply receivers directly, but WS-Addressing augments SOAP with a `wsa:replyTo` header that a compliant replier will use instead of the requestor's TCP connection.

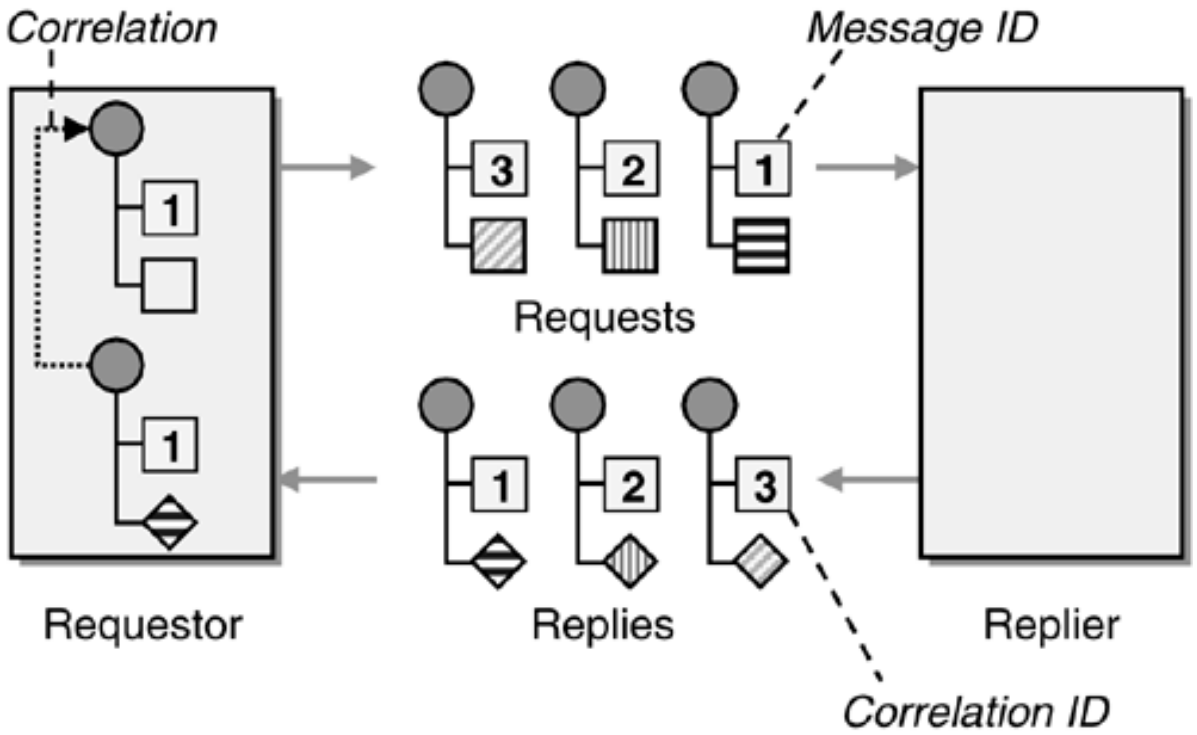
Example of Return Address in JMS for the requestor:

```
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);
```

Corresponding reply by the receiver:

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

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



Hohpe and Woolf (2004), p. 164

The Correlation Identifier pattern involves 6 concepts:

- Requestor: the application sending a request and waiting for a response.
- Replier: the application receiving the request, fulfilling it, and sending the reply.
- Request: the message containing a request identifier.
- Reply: the message containing the correlation identifier.
- Request identifier: a unique token for the request message.
- Correlation identifier: a token in the reply with the same value as the request identifier.

Messages

Correlation Identifier

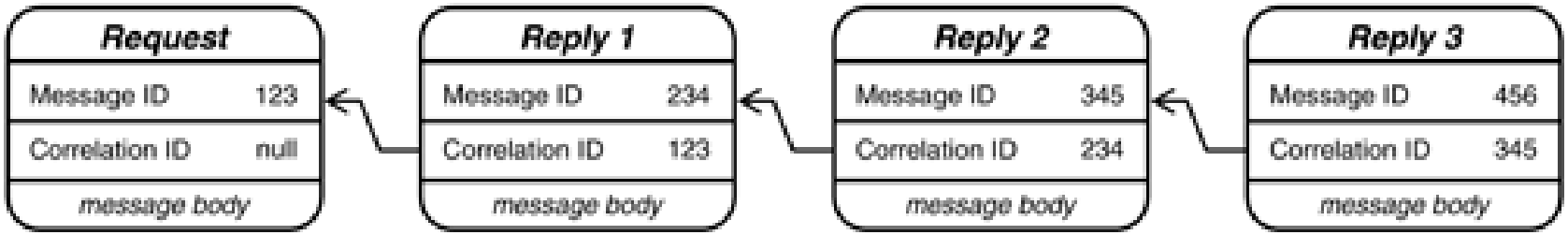
The requestor and replier need to agree on the name and type of the request identifier and correlation identifier.

The identifiers will normally be in the message header, not the body.

There are three main ways to create the IDs:

- Unique arbitrary ID: simple to create but difficult for reply processor to use.
- Business object ID: uses the ID of the relevant business object, e.g. an `Order` ID.
- Hybrid: the requestor keeps a map from request IDs to business objects.

When requests and replies are chained, we may have request IDs and correlation IDs in the same message:



Hohpe and Woolf (2004), p. 167

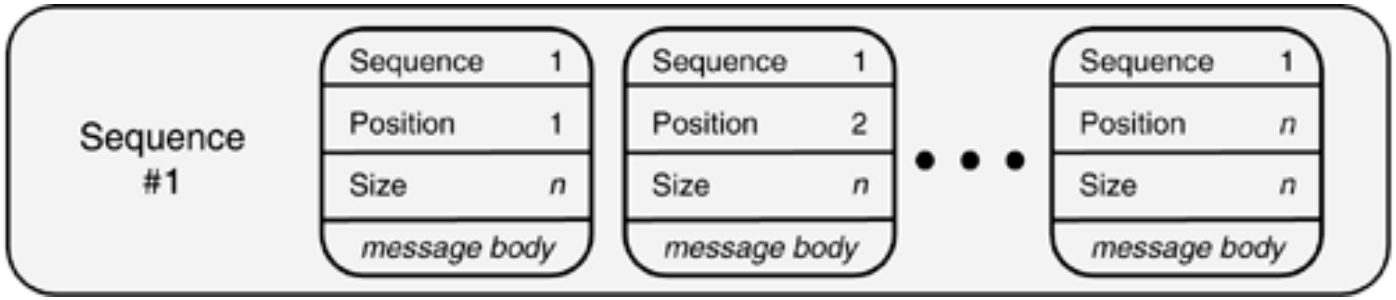
In SOAP, we can add fields to the envelope header.

JMS provides a Correlation Identifier attribute:

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

Oftentimes, there are practical limits on message size, and sometimes large messages hurt performance.

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



Hohpe and Woolf (2004), p. 171

Message Sequences need three fields:

- Sequence identifier: a unique ID for this cluster.
- Position identifier: an ID for a particular message indicating its position in the sequence.
- Size (integer) or End indicator (Boolean): allows receiver to know when all messages in a sequence have been received.

To avoid confusion, if a receiver expects a Message Sequence, we should always format it as a sequence, even when there is only one part.

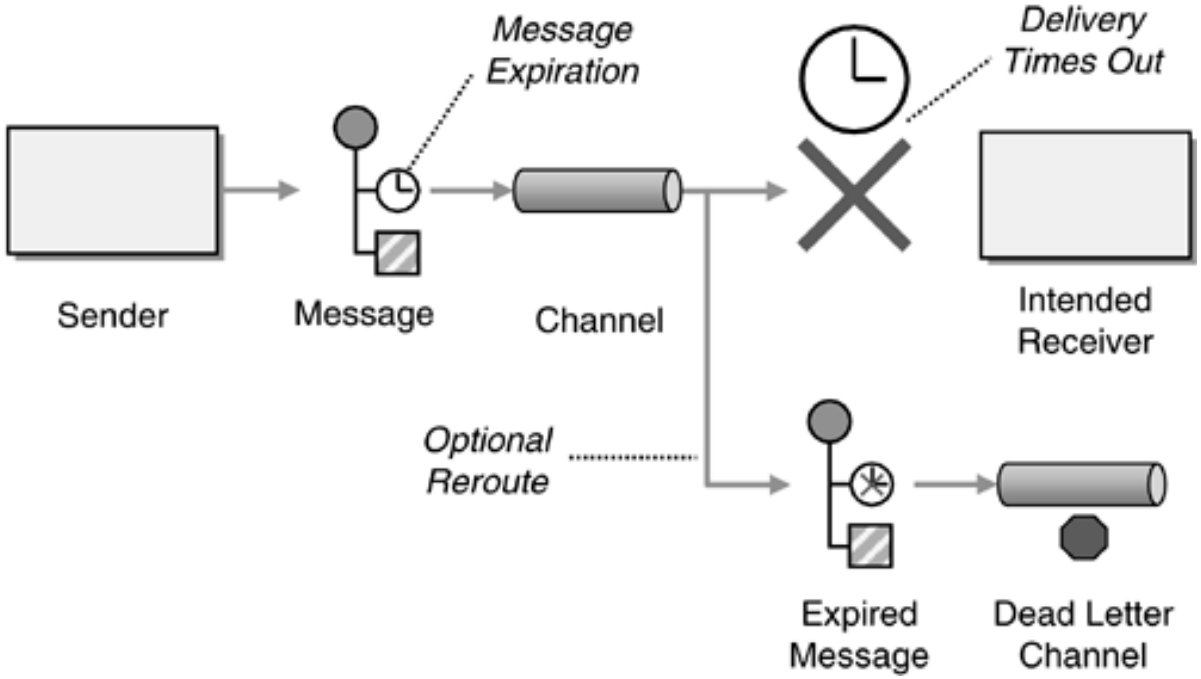
When message sequence is a reply to some request, the sequence ID and the correlation ID will normally be the same.

JMS does not support sequence IDs and position IDs directly, but JMS clients can always add arbitrary headers to a message.

Additional considerations for Message Sequence handling:

- If we get some but not all parts of a sequence, we should route to an Invalid Message Channel.
- We can use Message Sequence with a Transactional Client to ensure that no messages are delivered and processed until all messages are sent and received.
- Message Sequence is not compatible with Competing Consumers or Message Dispatcher.
- An alternative is Claim Check, in which we only pass around a key to the data that can be obtained from a database.
- The pattern is different from Splitter/Aggregator in that the large message never exists.

Set the Message Expiration *to specify a time limit for how long the message is viable.*



Hohpe and Woolf (2004), p. 177

Though messaging is reliable, it might take a long time to deliver a message.

Normally the messaging system will route any expired messages to a Dead Letter Channel.

We simply insert a timestamp into the message.

JMS supports Message Expiration with a time to live property:

- Use `MessageProducer.setTimeToLive(long)` to set the time to live (in ms) for all messages produced.
- Use `MessageProducer.send(Message message, int deliveryMode, int priority, long timeToLive)` to set the time for an individual message.

Message formats change over time.

In practice, a live system will have transition periods in which multiple versions of the same message exist simultaneously in the system.

Design a data format that includes a Format Indicator *so that the message specifies what format it is using.*

We allow different versions to share channels, but distinguish between the different formats.

Three main methods:

- Version number: the sender and receiver agree on the formats corresponding to the versions. Simple but difficult to maintain.
- Foreign key: we send a reference to a format document that has to be retrieved. Compact but more complex for recipient.
- Format document: the format is explained in each message. Good compromise but wastes bandwidth.

Outline

Routing patterns fall into three categories:

- Simple routers
- Composed routers
- Architectural patterns (architectural styles based on Message Routers)

Message Routing

Simple routers

We have 7 patterns for simple routers:

- Content-Based Router chooses output channels based on message content.
- Message Filter passes a message on to the output channel only if criteria are met.
- Dynamic Router allows reconfiguration via a control channel. Dynamic Router can be combined with most router patterns.
- Recipient List is a Content-Based Router that routes messages to more than one destination channel.
- Splitter splits a message into multiple messages.
- Aggregator receives a stream of messages and combines them into one stream. Necessarily stateful.
- Resequencer reorders messages into a proper sequence. Also stateful.

Message Routing

Composed routers and architectural patterns

There are four main patterns that combine multiple routing decisions:

- Composed Message Processor splits a message, routes it to parallel participants, and reassembles the replies.
- Scatter Gather is similar, but sends copies to the participants.
- Routing Slip specifies the path a message should take.
- Process Manager implements a routing path, but on each step, the results come back to the processor.

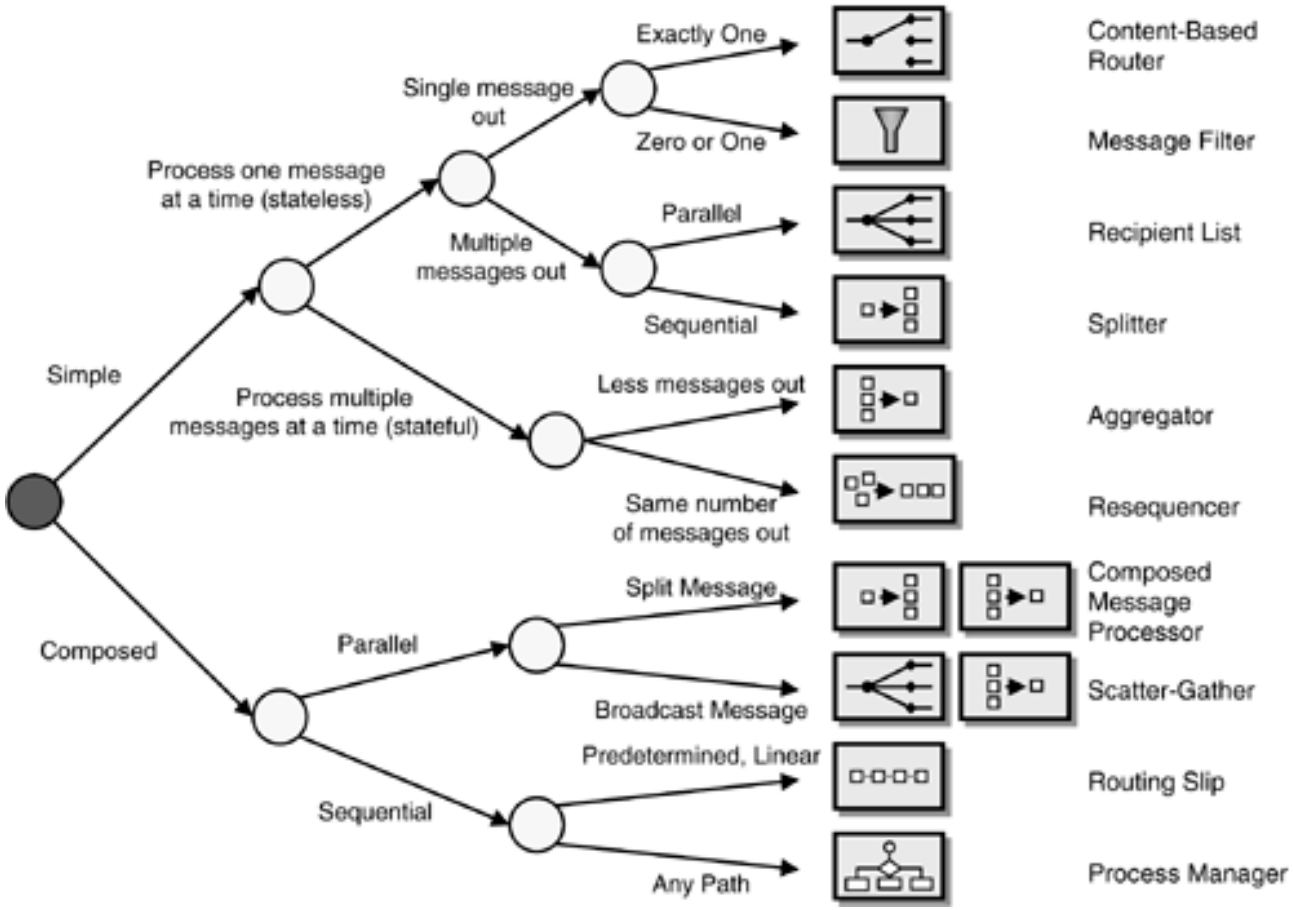
There is one main useful architectural style based on routers:

- Message Broker uses routing to organize the system into a hub and spoke architecture.

Message Routing

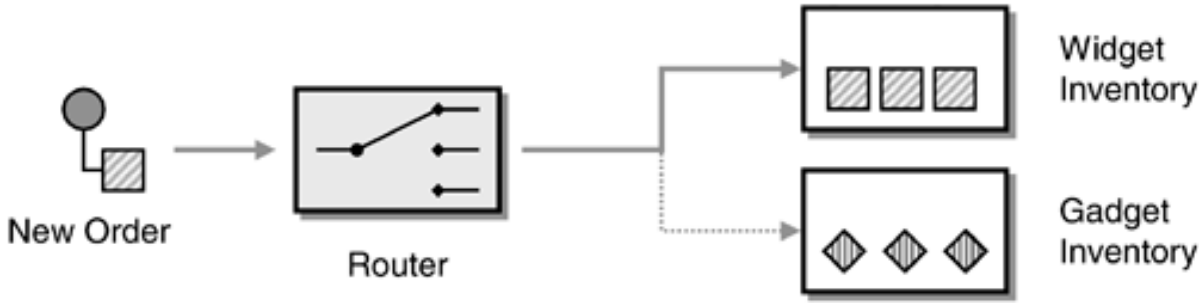
Road map to the router patterns

Overview of the situations to use each routing pattern in:



Hohpe and Woolf (2004), p. 229

Use a Content-Based Router to route each message to the correct recipient based on the message's content.



Hohpe and Woolf (2004), p. 232

Message Routing

Content-Based Router

The Content-Based Router is useful whenever we want to hide producers from the details of consumer business logic that is split across multiple system.

Usually delivers each message to exactly one output channel.

Criteria can be presence of a field, value of a field, etc.

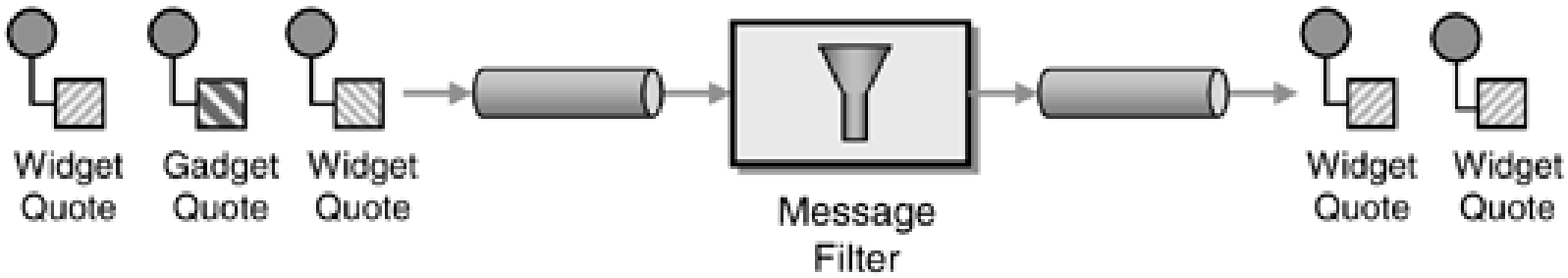
The routing rules are predictive in that the router knows about the capabilities of each downstream processor.

See text for example in C# and drag-drop functionality in TIBCO.

Message Routing

Message Filter

Use a special kind of Message Router, *a* Message Filter, *to eliminate undesired messages from a channel based on a set of criteria.*



Hohpe and Woolf (2004), p. 238

Message Routing

Message Filter

Sometimes Publish-Subscribe Channels are not fine grained enough.

We could use Content-Based Router in this situation, but it will be difficult to maintain if subscribers change preferences frequently.

Message Filter puts the subscriber in control of what it receives.

Usually stateless but some times stateful (e.g. a de-duper).

We already saw Selective Consumer in JMS. This is usually better than Message Filter but it's not always possible.

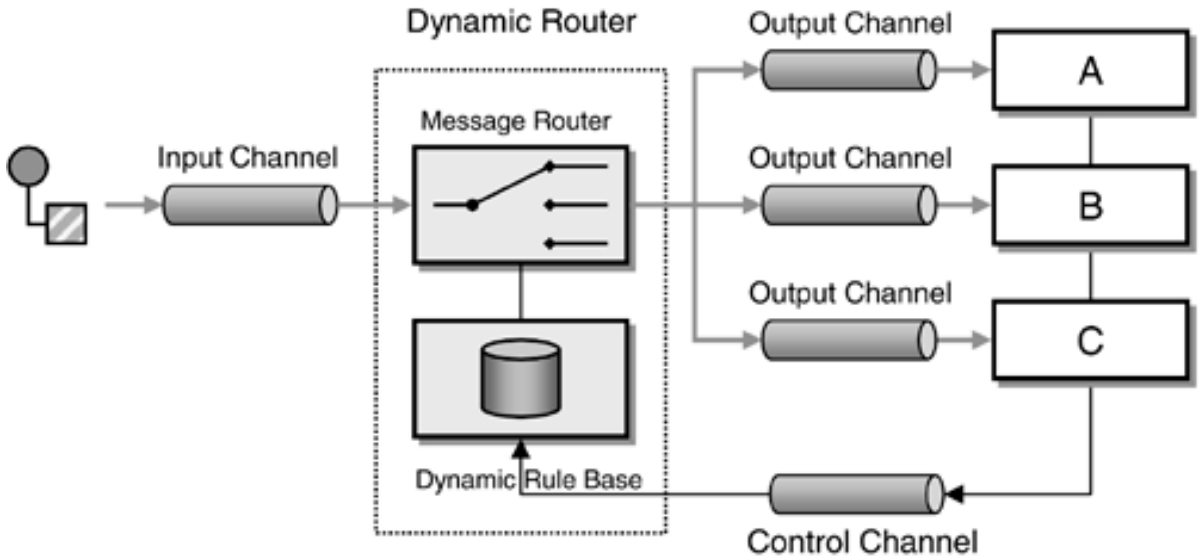
See C# example in text.

Message Routing

Message Filter vs. Content-Based Router

Content-Based Router	Publish-Subscribe Channel with Message Filters
Exactly one consumer receives each message.	More than one consumer can consume a message.
Central control and maintenance — predictive routing.	Distributed control and maintenance — reactive filtering.
Router needs to know about participants. Router may need to be updated if participants are added or removed.	No knowledge of participants required. Adding or removing participants is easy.
Often used for business transactions, e.g., orders.	Often used for event notifications or informational messages.
Generally more efficient with queue-based channels.	Generally more efficient with publish-subscribe channels.

Use a Dynamic Router*, a router that can self-configure based on special configuration messages from participating destinations.*



Hohpe and Woolf (2004), p. 244

Message Routing

Dynamic Router

Dynamic Router adds a control channel to any other routing scheme that allows participants to announce their preferences.

The main problem is conflicts between rules specifying who a message should be sent to. Possible strategies:

- Ignore conflicting control messages.
- Allow conflicting rules but only use the first applicable rule we find.
- Allow conflicting rules and simply send to all (Recipient List).

Dynamic Router is useful for service discovery in a SOA:

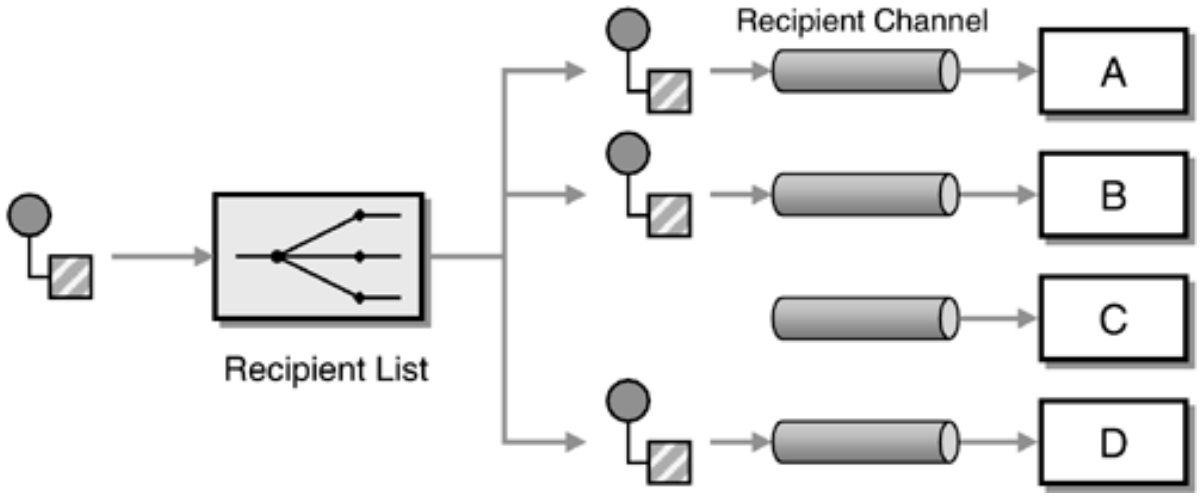
- Service providers register with the Dynamic Router.
- Requests from requestors only mention the service name; the Dynamic Router routes the message to the correct service provider.

See C# example in text.

Message Routing

Recipient List

Define a channel for each recipient. Then use a Recipient List *to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list.*



Hohpe and Woolf (2004), p. 250

Message Routing

Recipient List

In Content-Based Router, messages are only routed to one destination channel.

For multiple recipients, we could yse Publish-Subscribe Channel and Message Filter.

Having the recipient information split across many filters is difficult to maintain, however.

Recipient List provides a more centeralized management scheme.

The list of recipients can be attached to the message itself (less common) or computed by the Recipient List router logic (more common).

If we add Dynamic Router, the Recipient List becomes under control of the receivers.

How can we achieve reliability with Recipient List?

- Use a transaction.
- Keep the list persistent, track each failed delivery, and retry until successful.
- Use idempotent recievers that are not affected by duplicate messages, and restart whenever a failure occurs.

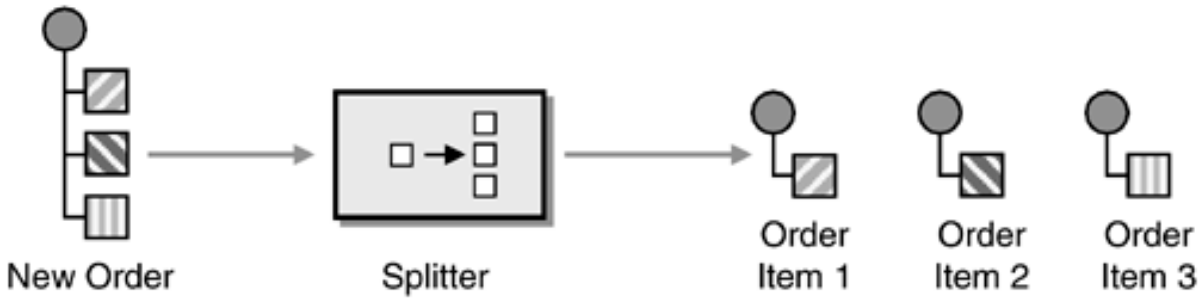
See C# example in text.

Recipient List	Publish-Subscribe Channel with Message Filters
Central control and maintenance — predictive routing.	Distributed control and maintenance — reactive filtering.
Router needs to know about participants. Router may need to be updated if participants are added or removed (unless using dynamic router, but at expense of losing control).	No knowledge of participants required. Adding or removing participants is easy.
Often used for business transactions, e.g., request for quote.	Often used for event notifications or informational messages.
Generally more efficient if limited to queue-based channels.	Can be more efficient with publish-subscribe channels (depends on infrastructure).

Message Routing

Splitter

Use a Splitter to break out the composite message into a series of individual messages, each containing data related to one item.

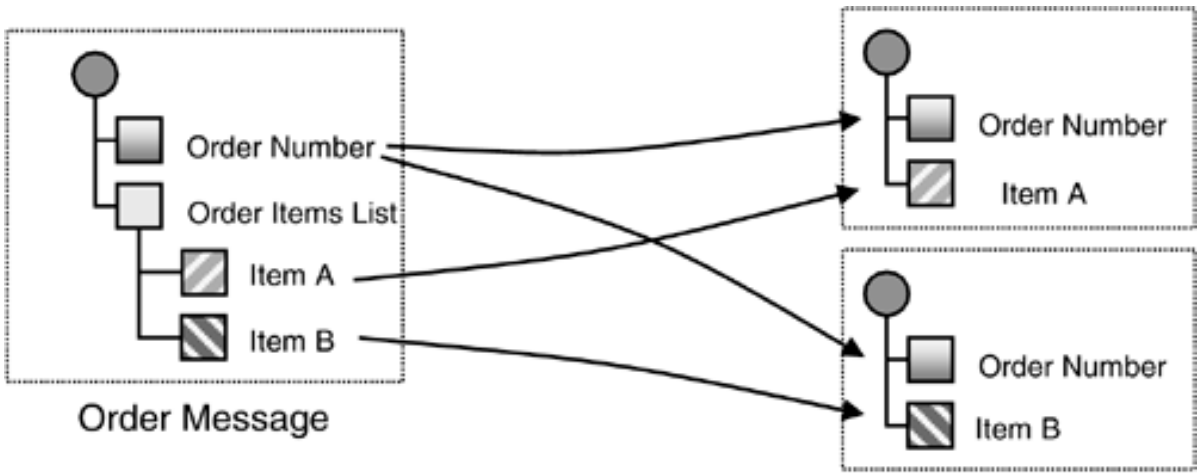


Hohpe and Woolf (2004), p. 260

Message Routing

Splitter

The split-up messages normally combine some common information with the pieces:



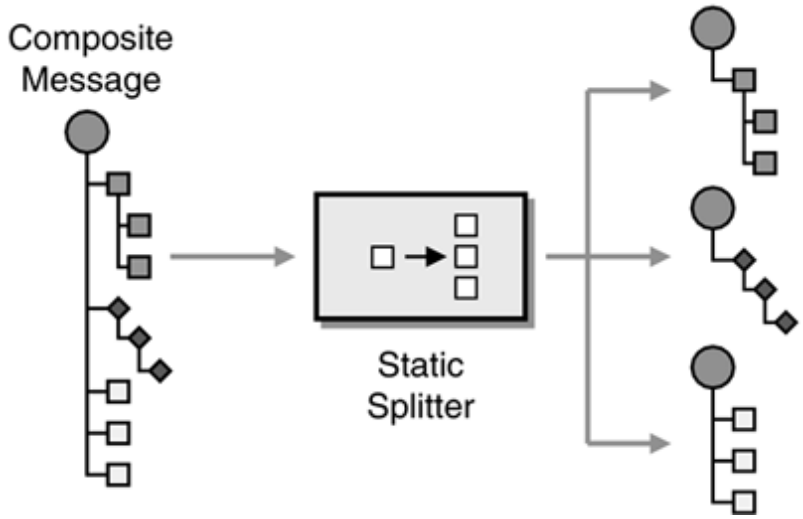
Hohpe and Woolf (2004), p. 260

Message Routing

Special cases of Splitter

There are two special cases of Splitter:

- When the Splitter pattern is applied recursively, we have an iterating Splitter (think XSL).
- When the Splitter uses a fixed strategy, rather than using repeating elements, we have a static Splitter:



Hohpe and Woolf (2004), p. 261

Oftentimes, the split-up messages will need to be processed independently then recombined by an Aggregator.

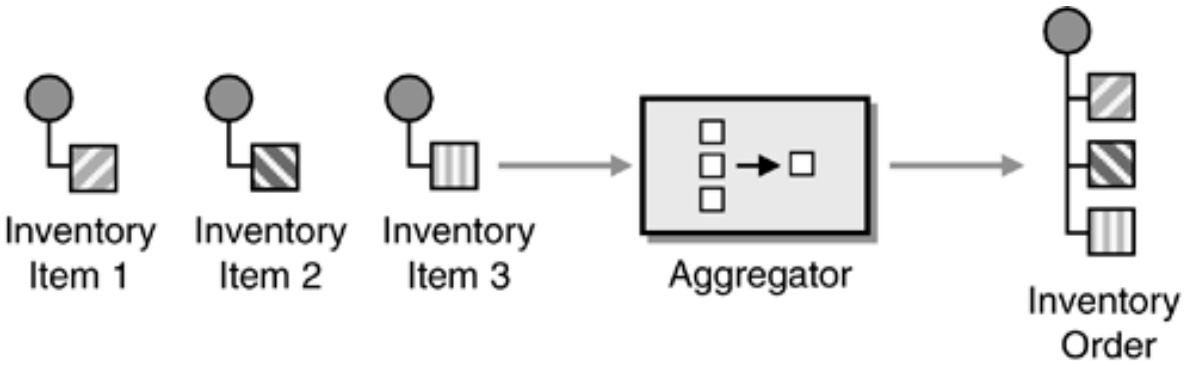
In this case, we need a Correlation Identifier.

If the message ordering is important, we need a sequence number in every message.

Message Routing

Aggregator

Use a stateful filter, an Aggregator*, to collect and store individual messages until it receives a complete set of related messages. Then, the* Aggregator *publishes a single message distilled from the individual messages.*



Hohpe and Woolf (2004), p. 269

Message Routing

Aggregator

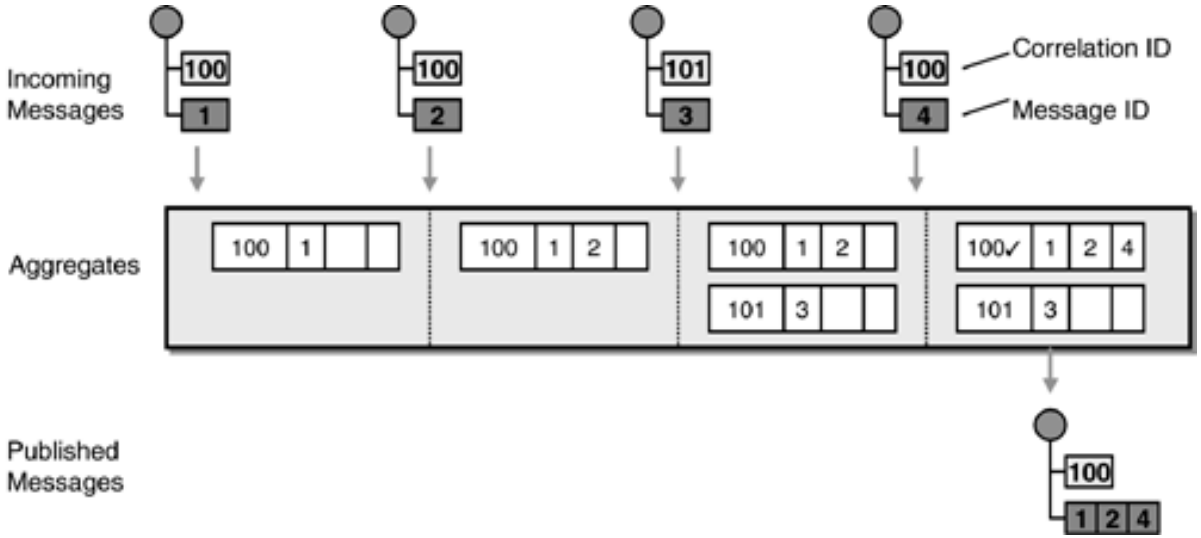
Aggregation requires solutions to three problems:

- Correlation: how to identify the messages to aggregate?
- Completeness criteria: how to decide when we're ready to publish the aggregate?
- Aggregation algorithm: how to combine the individual messages?

Basic algorithm:

- If a message with a new Correlation Identifier arrives, create a new aggregate and insert the message.
- If a message with an existing Correlation Identifier arrives, add the new message to the aggregate.
- Check completeness criteria. If complete, publish.

Illustration of the basic algorithm:



Hohpe and Woolf (2004), p. 271

Message Routing

Aggregator

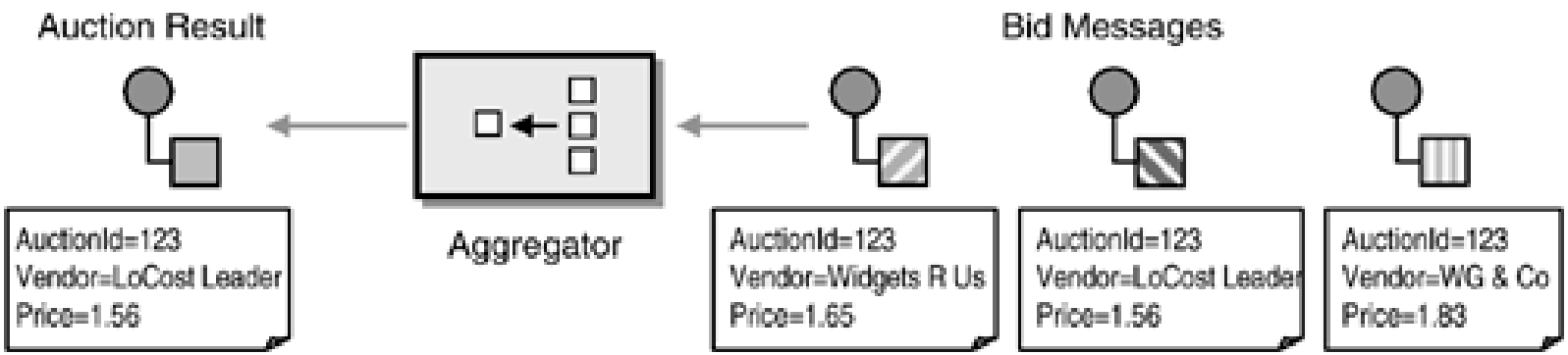
Different applications will require different approaches to the completeness criteria:

- Wait for all: simplest approach, but has many issues. What if a message is delayed or a processor crashes?
- Timeout: wait for some amount of time then process the results up to that time. Common in bidding-type applications.
- First best: only wait for one response. Ignore others. Also common in bidding and quotation systems.
- Timeout with override: wait until a timeout occurs or a preset minimum score is reached, whichever comes first.
- External event: stop aggregating when a particular Event Message arrives.

Different applications will also take different approaches to the aggregation algorithm:

- Best only: only report the data from the best message aggregated.
- Condense data: take means or other summary statistics of the aggregated messages.
- Collect data for later evaluation: just collect all replies into a single message and publish it.

See text for example JMS implementation for this situation:

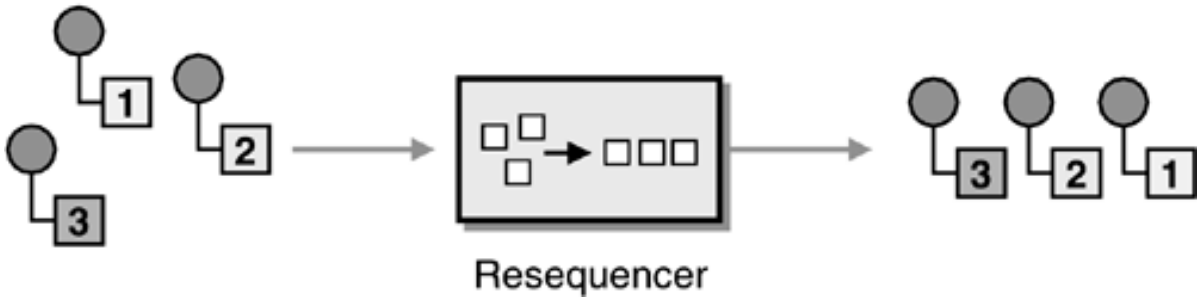


Hohpe and Woolf (2004), p. 276

Message Routing

Resequencer

Use a stateful filter, a Resequencer*, to collect and reorder messages so that they can be published to the output channel in a specified order.*



Hohpe and Woolf (2004), p. 284

Message Routing

Resequencer

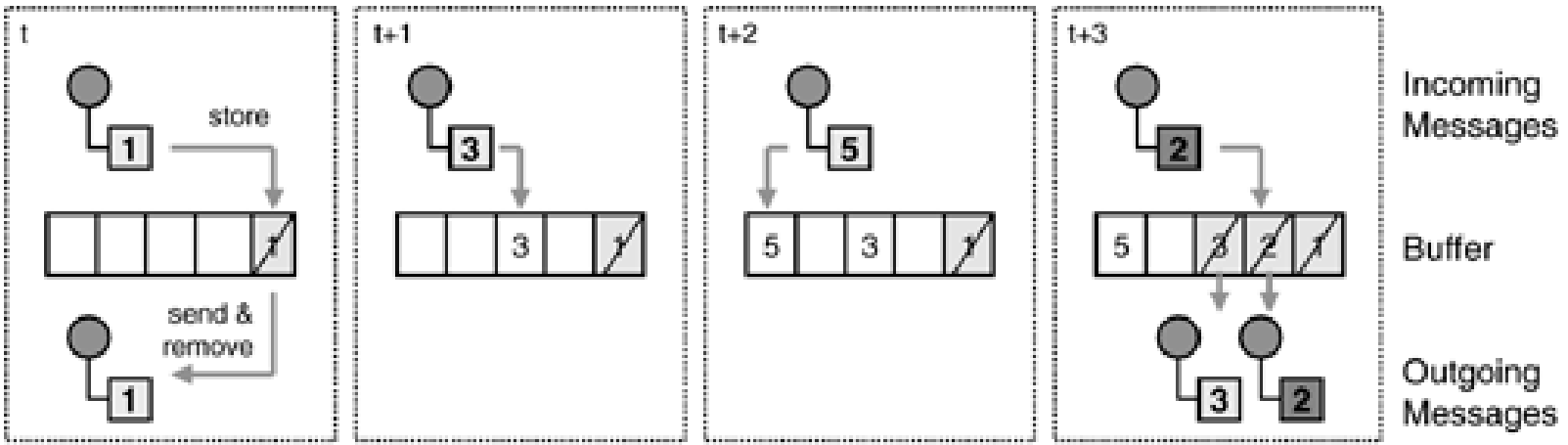
If we have parallel processing, messages can get out of order.

If ordering is important, we need a sequence number in each message. (Remember Message Sequence?)

If the parallel messages are being published by the same producer, e.g., a Splitter, it is easy to insert the sequence numbers.

In other cases, it could be tricky, and we may have to use a distributed sequence generation algorithm.

The basic idea is to delay forwarding messages if there is a gap in the sequence, until the gap is filled:



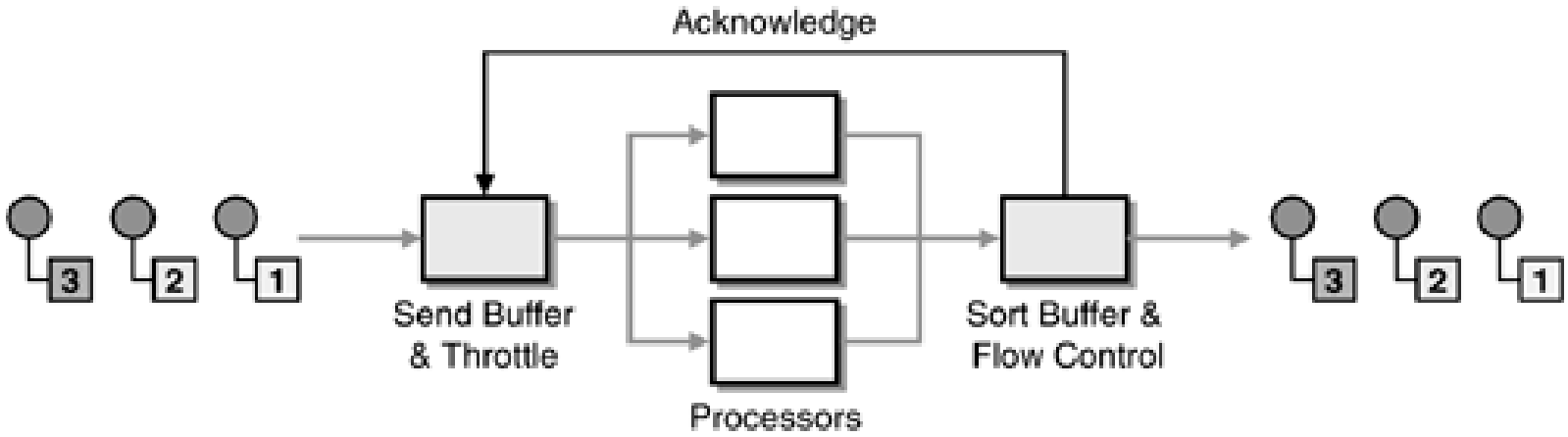
Hohpe and Woolf (2004), p. 287

Message Routing

Resequencer

The main problem is buffer overrun when one route fails or is slow.

In this case, a throttle may be appropriate:

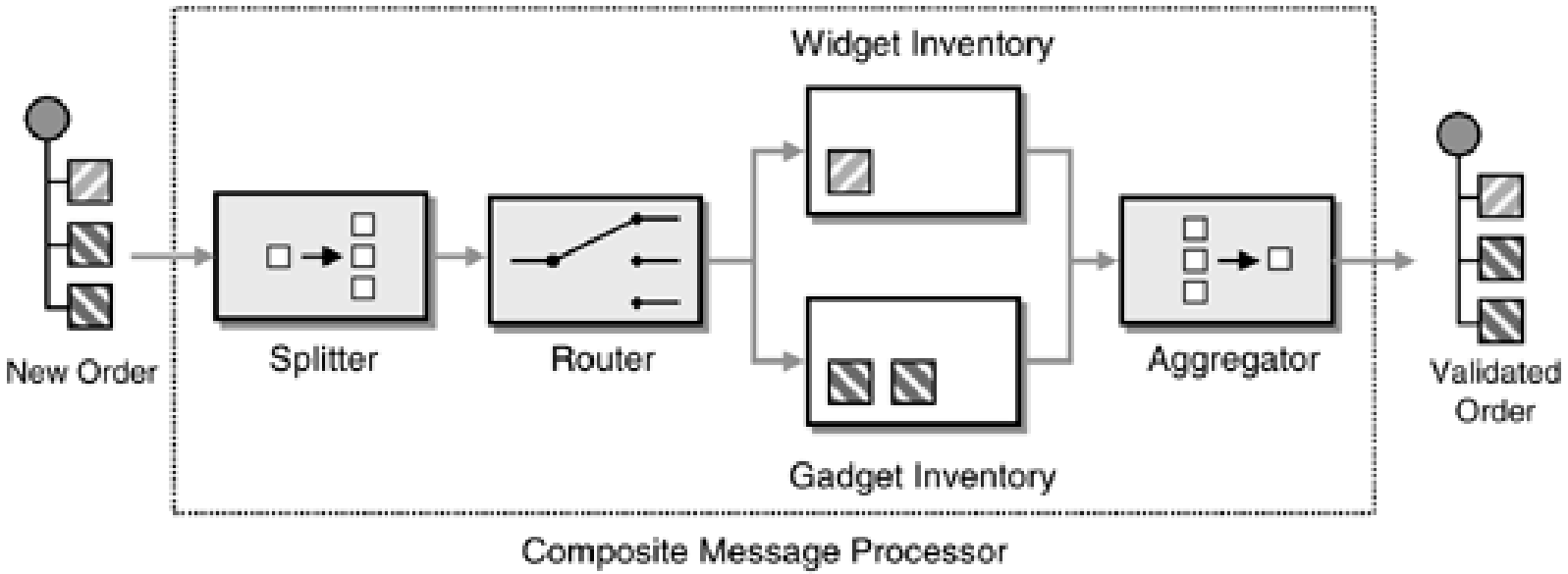


Hohpe and Woolf (2004), p. 287

The throttle tells the producer how many slots are available. (Recall the Bounded Buffer Producer-Consumer pattern from undergraduate operating systems!)

See the C# resequencer example in the text.

Use a Composed Message Processor *to process a composite message. The* Composed Message Processor *splits the message up, routes the submessages to the appropriate destinations, and reaggregates the responses back into a single message.*



Hohpe and Woolf (2004), p. 296

Whenever parts of a message require different processing, Composed Message Processor is appropriate.

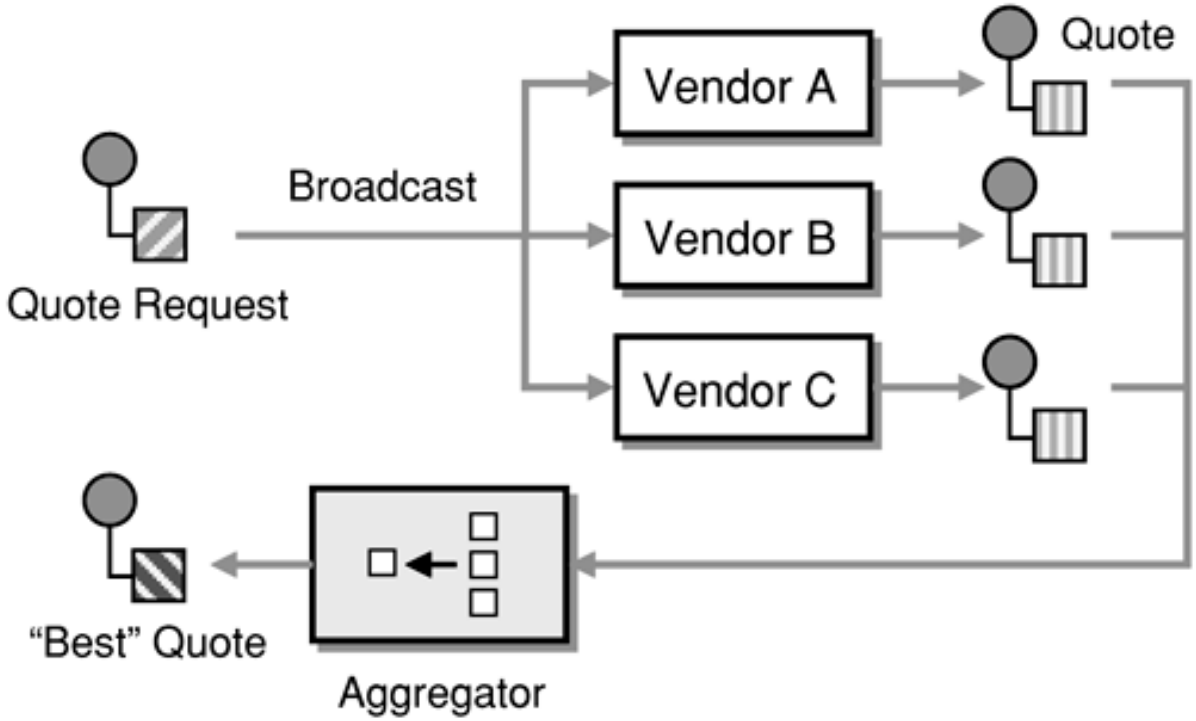
The pattern composes a Splitter, a Content-Based Router, and an Aggregator, all chained using Pipes and Filters.

The Aggregator still has the usual problems with delayed constituent messages.

Message Routing

Scatter-Gather

Use a Scatter-Gather *that broadcasts a message to multiple recipients and reaggregates the responses back into a single message.*



Hohpe and Woolf (2004), p. 298

Message Routing

Scatter-Gather

Scatter-Gather is appropriate when we want to ask the same question of different partners/applications.

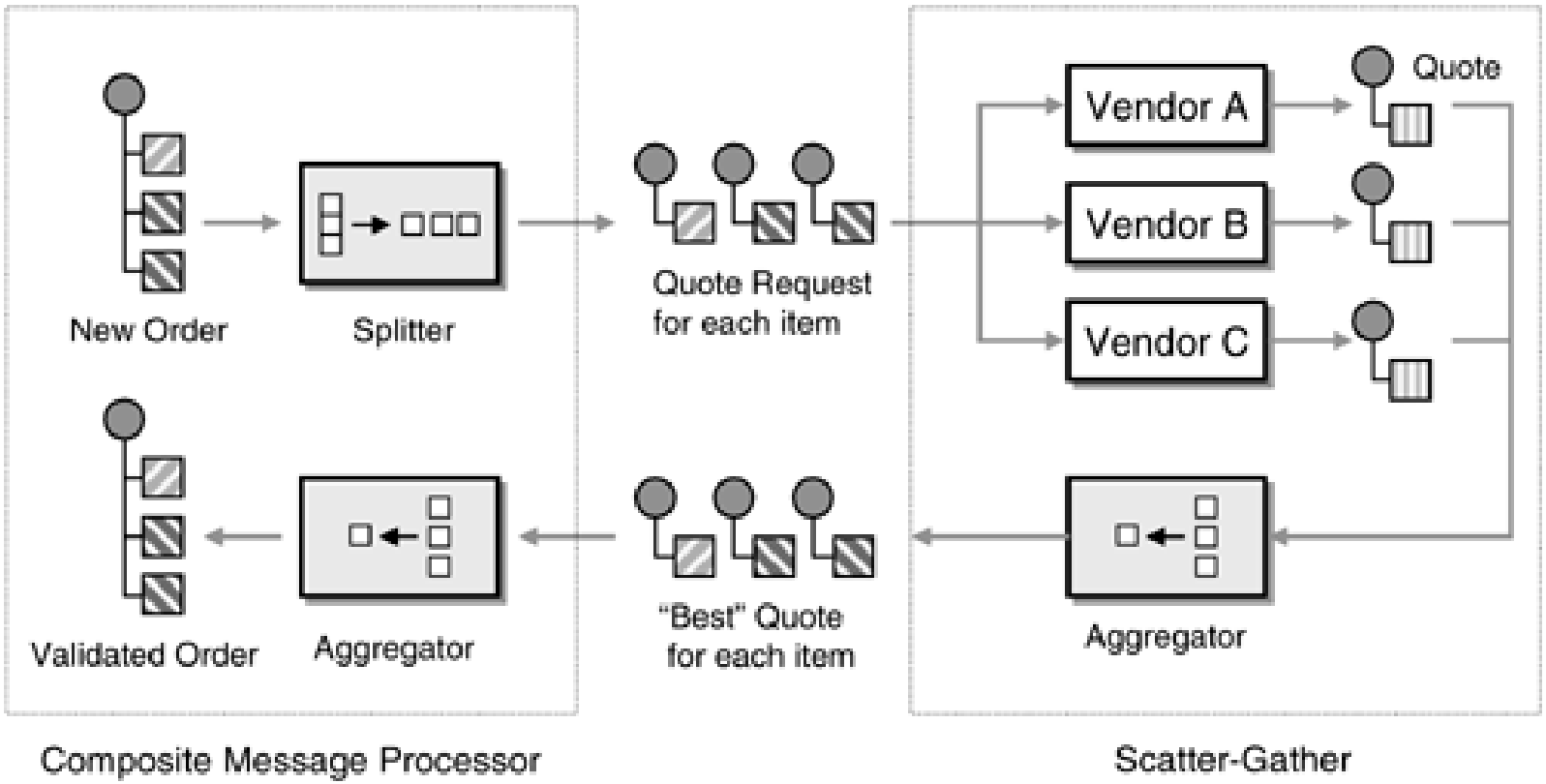
We may only receive responses from some of the recipients.

We use Aggregator to handle this kind of uncertainty.

There are two main variations:

- Distribution via a Recipient List gives Scatter-Gather control over recipients but causes coupling to their message channels.
- Auction-style control uses a single channel for broadcast, relinquishing control over the recipients but decoupling the the sender from the recipients.

A nice example combining Scatter-Gather with Composed Message Processor:

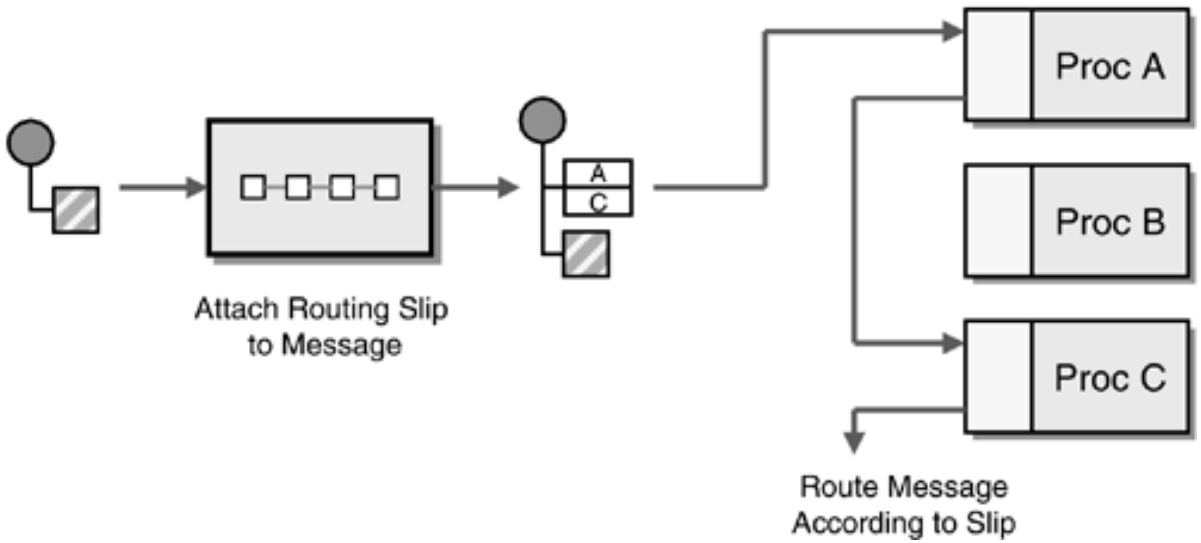


Hohpe and Woolf (2004), p. 300

Message Routing

Routing Slip

Attach a Routing Slip *to each message, specifying the sequence of processing steps. Wrap each component with a special message router that reads the Routing Slip and routes the message to the next component in the list.*



Hohpe and Woolf (2004), p. 305

Message Routing

Routing Slip

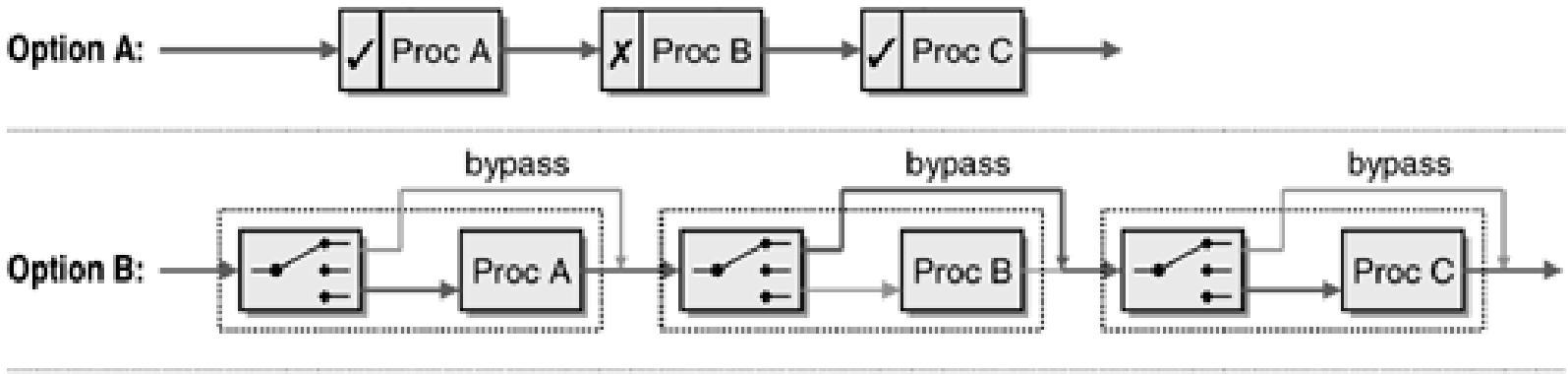
Pipes and Filters is great when we want the same thing to occur to every message.

But if the filtering depends on the type of message, we need some way to make the sequencing of filters more dynamic.

We need a solution with a few characteristics:

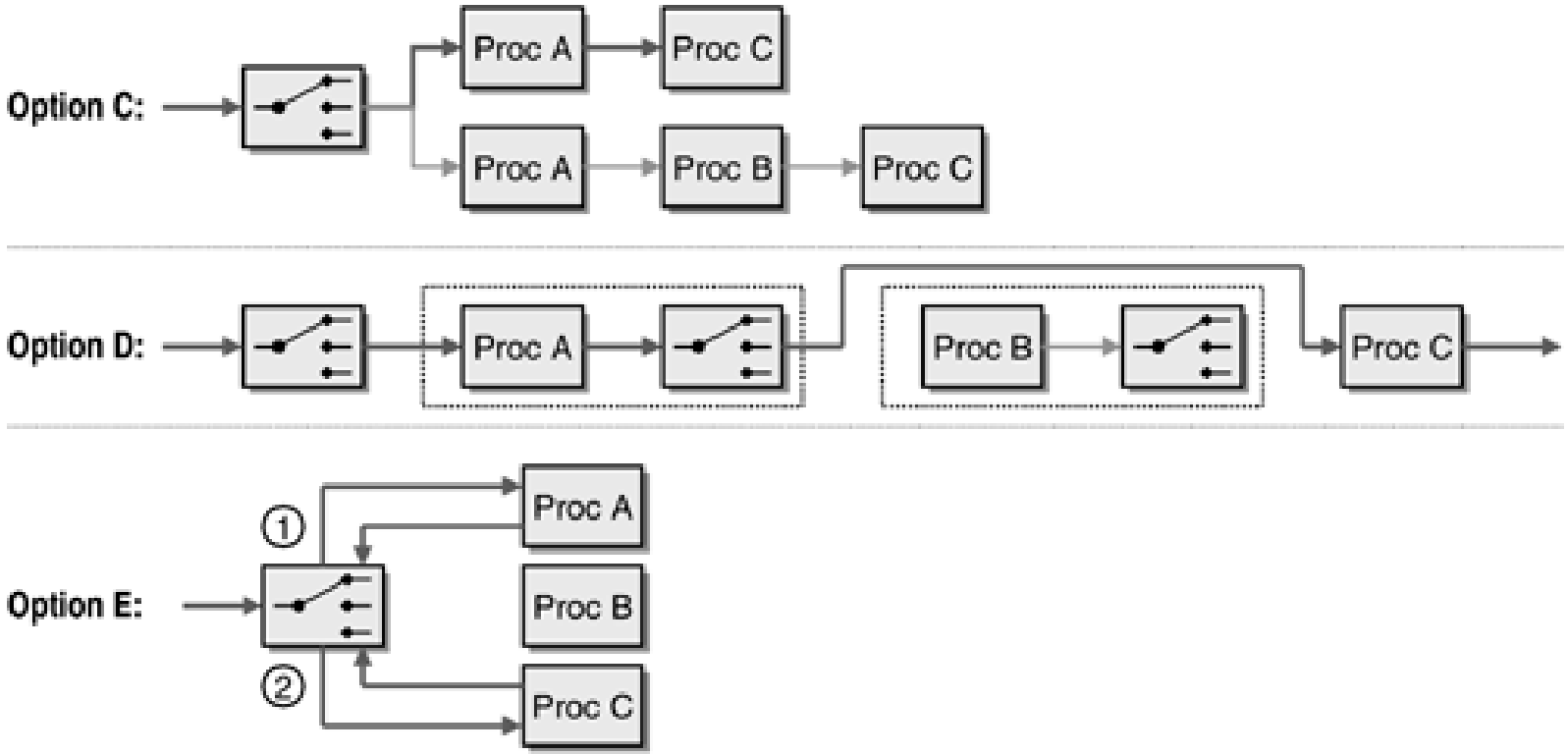
- Efficient message flow: we should allow message to bypass unnecessary components.
- Efficient use of resources: we should not use a huge number of channels, routers, and so on.
- Flexible: it should be easy to change a message's route.
- Simple to maintain: we should be able to introduce new message types and so on easily.

Here's a (weak) solution for three filters and a message that needs to skip filter B:



Hohpe and Woolf (2004), p. 302

Some other approaches, with different problems:



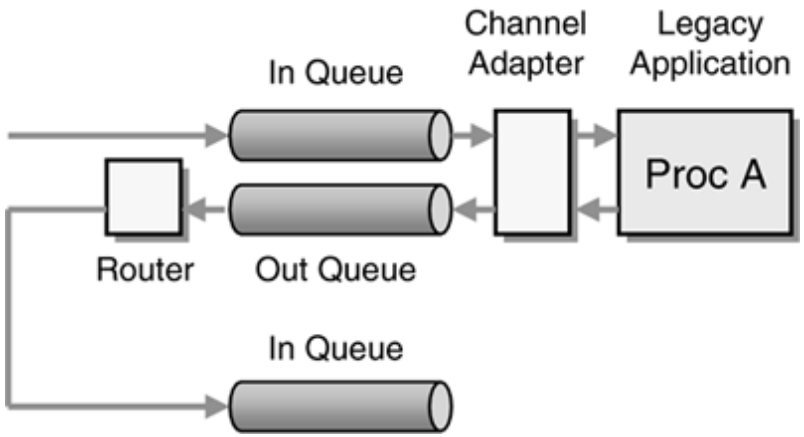
Hohpe and Woolf (2004), p. 303

Routing Slip combines the centralized control of option E with the with the efficiency of the hard coded straight Pipes and Filters of option C.

Message Routing

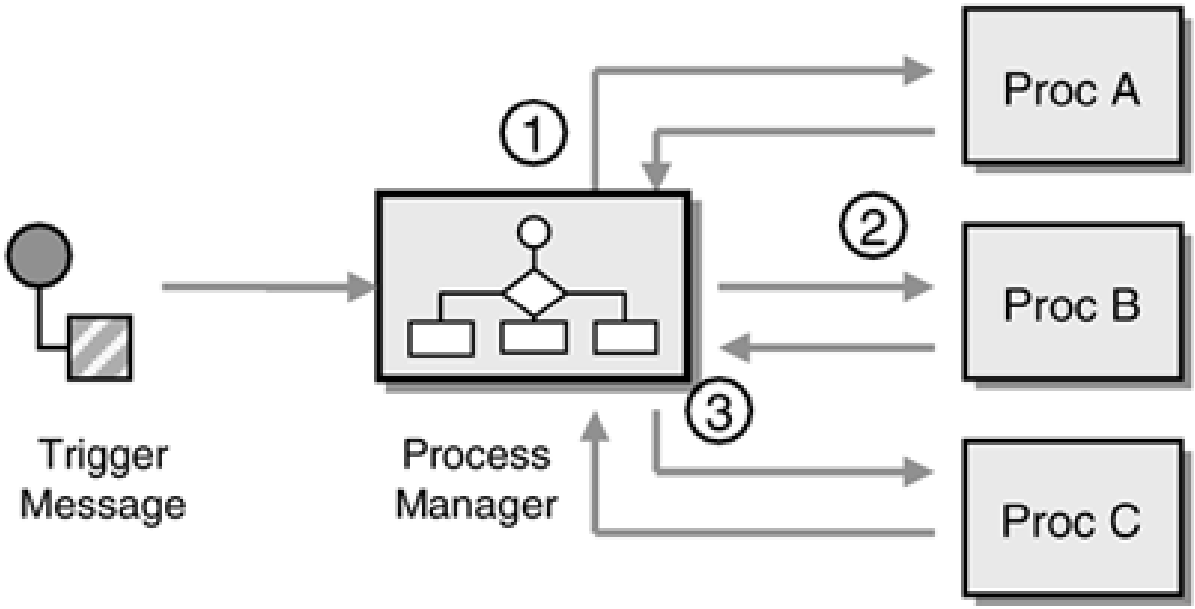
Routing Slip

When we are routing through a legacy application, we will not have the flexibility to add Routing List processing in the component itself, but rather in the Channel Adapter or a completely external router:



Hohpe and Woolf (2004), p. 307

Use a central processing unit, a Process Manager*, to maintain the state of the sequence and determine the next processing step based on intermediate results.*



Hohpe and Woolf (2004), p. 313

Routing Slip allows us to create a dynamically generated list of processing steps:

- But the steps are a linear list. No parallelism.
- The list of steps if fixed a priori. No change based on intermediate processing steps.

Process Manager achieves more flexible flow using a central processing unit responsible to route to the appropriate processing units.

This results in a hub-and-spoke architecture rather than a pipes-and-filters architecture.

The main cost of Process Manager is the increased complexity required to maintain process state:

- Where we are in the execution of the process.
- Intermediate data that will be needed (the process manager thus implements Claim Check).

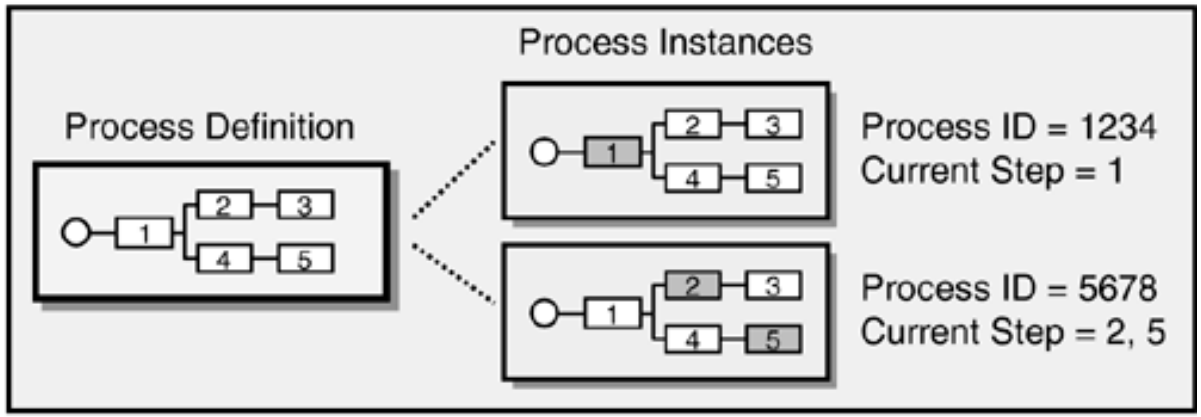
We prefer processing units that are stateless to simplify their design.

Reliability requires that Process Manager should store process state persistently.

Message Routing

Process Manager

The first message, a trigger message, causes creation of a process instance.



Hohpe and Woolf (2004), p. 315

Each process instance is an instance of some specific process definition (a template for the process execution).

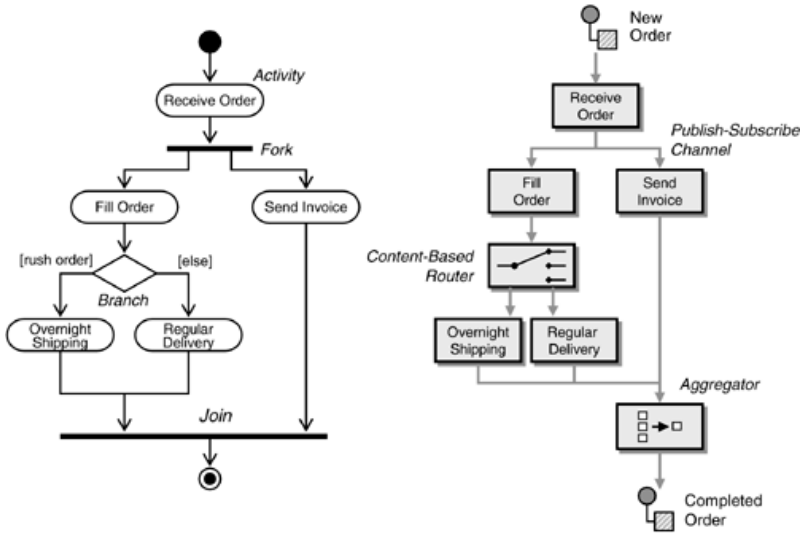
Incoming response messages are associated with a process instance using Correlation Identifier.

Message Routing

Process Manager

Process definitions are typically created using a visual business process modeling tools.

UML activity diagrams use activities, forks, joins, and branches that map directly to pipes-and-filters components.



Hohpe and Woolf (2004), p. 319

The visual model is typically converted into a vendor-specific language like JBPM or a standard language such as BPML or BPEL.

Message Routing

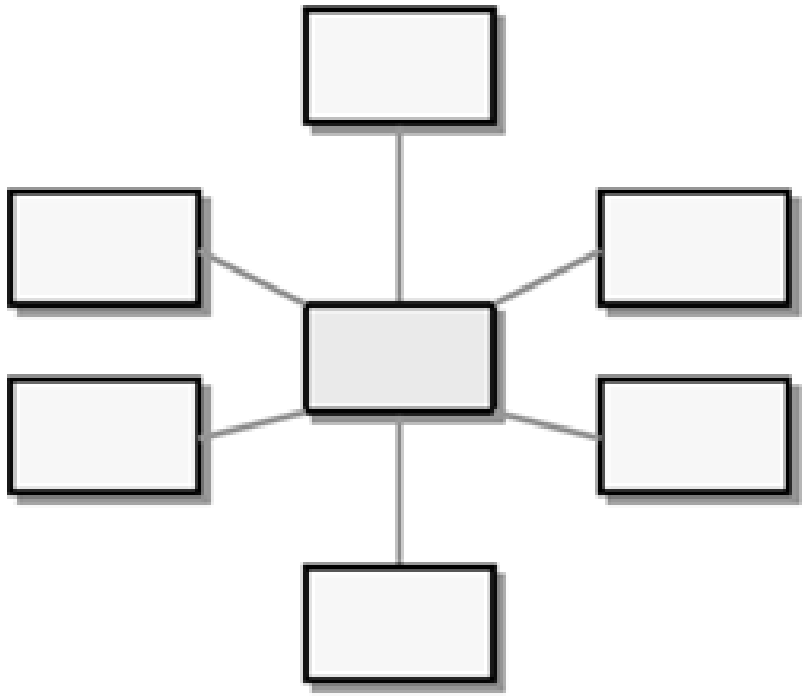
Process Manager

Distributed Pipes and Filters	Routing Slip	Central Process Manager
Supports complex message flow	Supports only simple, linear flow	Supports complex message flow
Difficult to change flow	Easy to change flow	Easy to change flow
No central point of failure	Potential point of failure (compute routing table)	Potential point of failure
Efficient distributed runtime architecture	Mostly distributed	Hub-and-spoke architecture may lead to bottleneck
No central point of administration and reporting	Central point of administration, but not reporting	Central point of administration and reporting

Message Routing

Message Broker

Use a central Message Broker *that can receive messages from multiple destinations, determine the correct destination, and route the message to the correct channel. Implement the internals of the Message Broker using other message routers.*



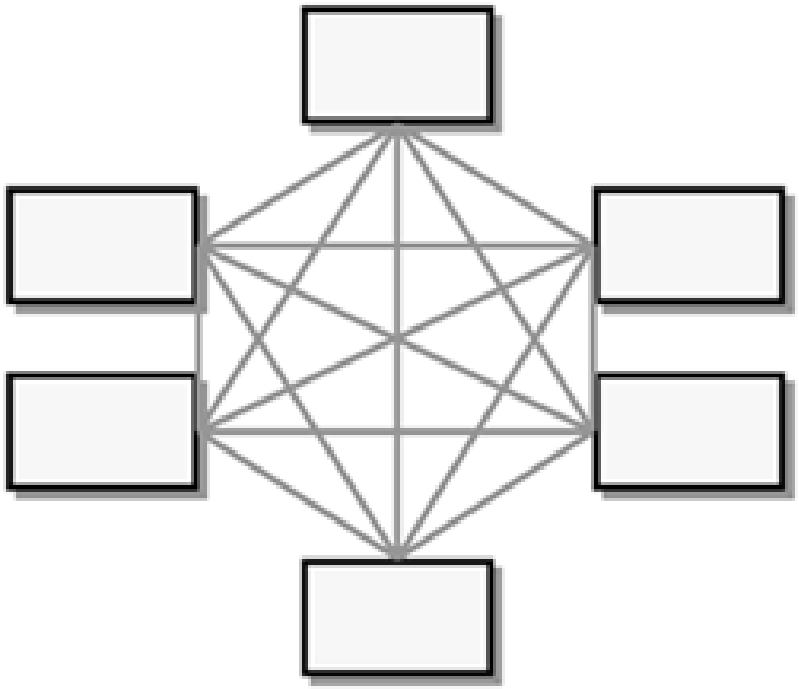
Hohpe and Woolf (2004), p. 324

Message Routing

Message Broker

Message Channel helps to decouple senders and receivers, but they are still coupled in that both parties need to know address of the channel.

Eventually we get "integration spaghetti:"



Hohpe and Woolf (2004), p. 322

Message Routing

Message Broker

Message Broker further decouples application by

- Routing messages to the correct destinations using any of the routing strategies we've already discussed.
- Performing appropriate message format translations between application-specific formats and a Canonical Data Model.

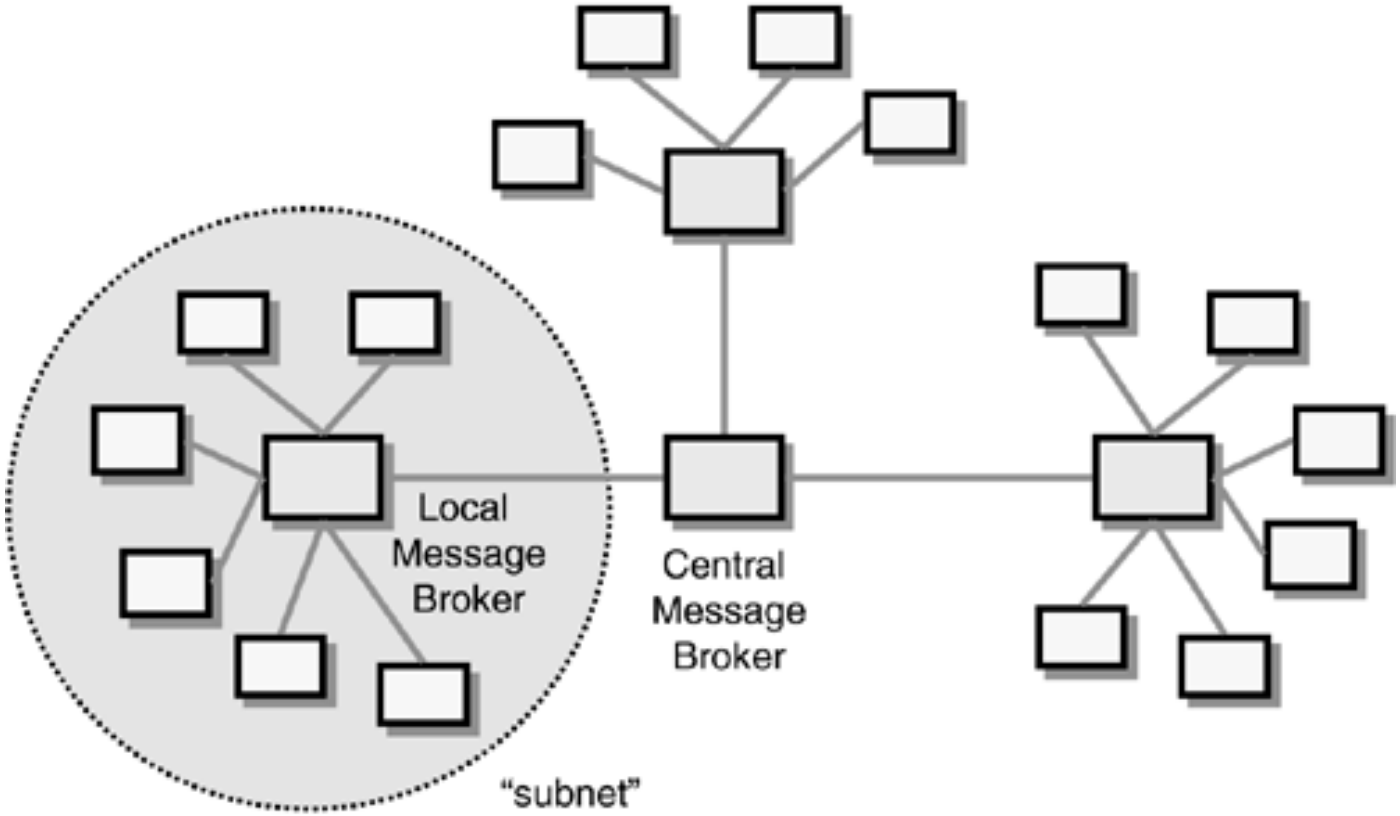
Message Broker moves the routing and transformation logic into the middleware rather than having separately administered channels and filters.

EAI packages provide visual tools for administering the message broker and its ruleset.

Message Routing

Message Broker

To prevent a single broker from becoming a bottleneck, we may use multiple brokers and organize them hierarchically:



Hohpe and Woolf (2004), p. 325

Outline

Message Transformation

Introduction

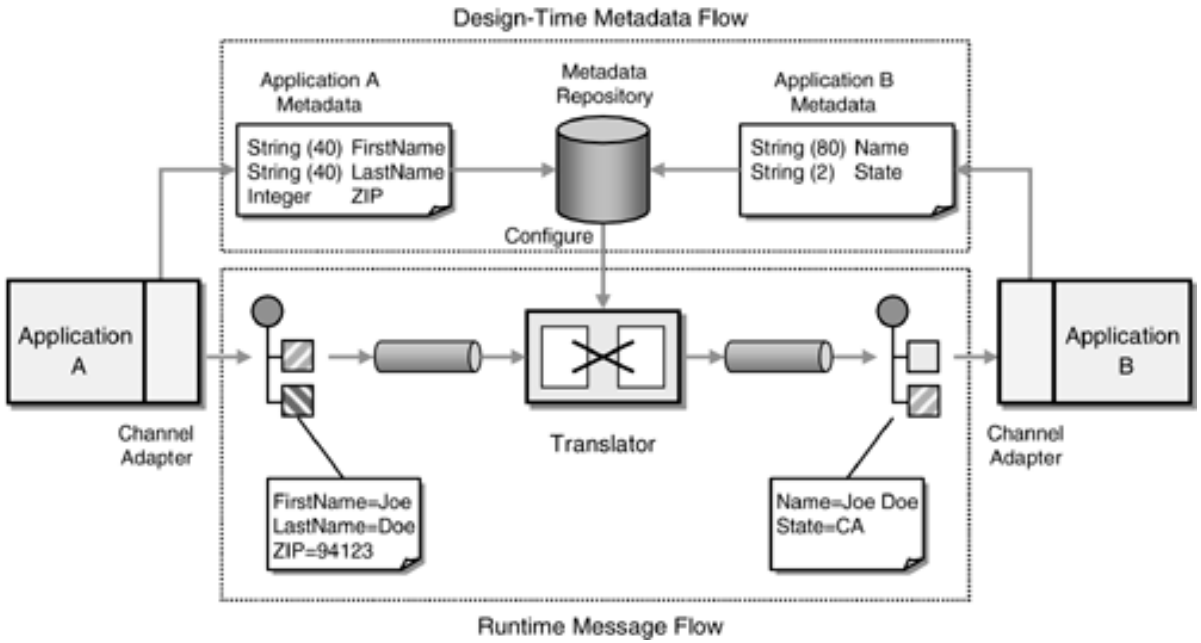
So far, our patterns are aimed mainly at address decoupling.

Without data format decoupling, however, we still have unmanageably strong dependencies between systems.

Here we consider a few special cases of Message Translator:

- Envelope Wrapper: wrap a payload in a messaging system compliant envelope.
- Content Enricher: add data to a message needed by recipeint but not produced by sender.
- Content Filter: remove unneeded data from a message.
- Claim Check: remove data from a message, store for later use.
- Normalizer: maps many formats to a single format.
- Canonical Data Model: map all message formats to a canonical data model and back to application-specific formats.

Message transformation benefits from the use of metadata that is extracted and stored separately from the data.



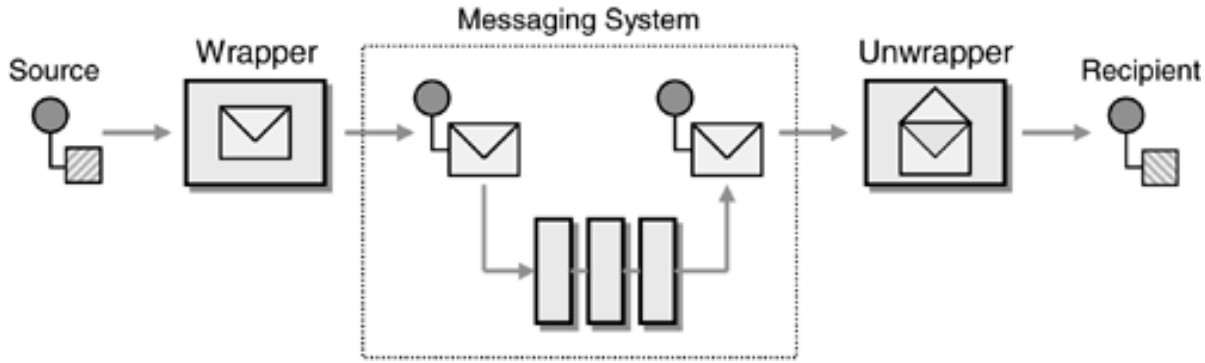
Hohpe and Woolf (2004), p. 328

Example: a set of XML schema definitions.

Message Transformation

Envelope Wrapper

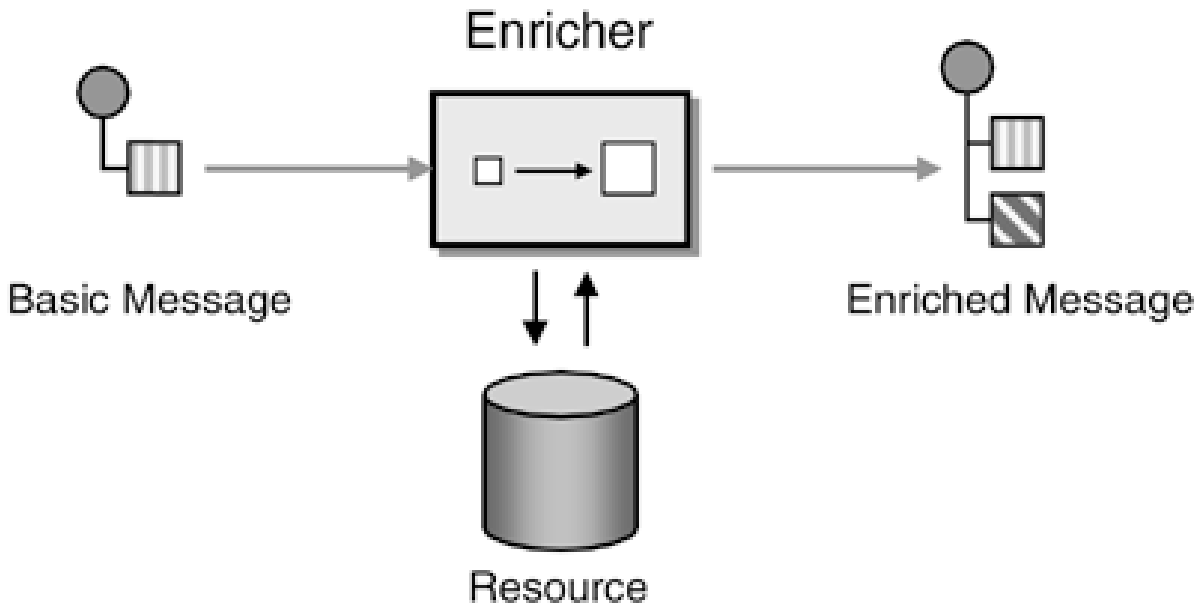
Use an Envelope Wrapper *to wrap application data inside an envelope that is compliant with the messaging infrastructure. Unwrap the message when it arrives at the destination.*



Hohpe and Woolf (2004), p. 331

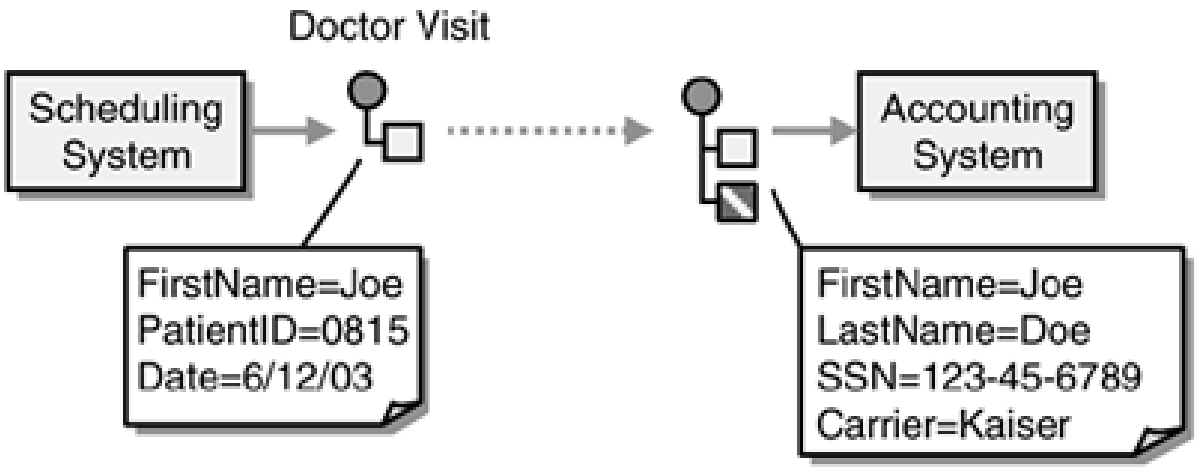
Good for encrytption and decryption, adding required header fields, adding security credentials, and so on.

Use a specialized transformer, a Content Enricher*, to access an external data source in order to augment a message with missing information.*



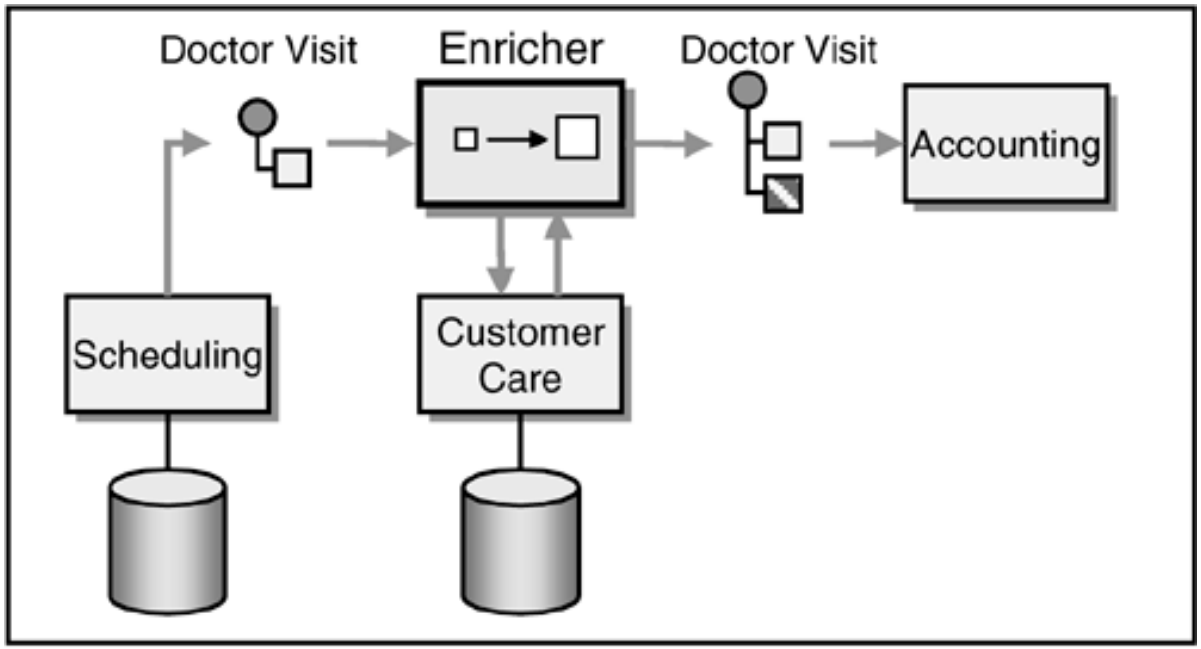
Hohpe and Woolf (2004), p. 339

Example: the accounting system in a doctor's office might require additional information:



Hohpe and Woolf (2004), p. 336

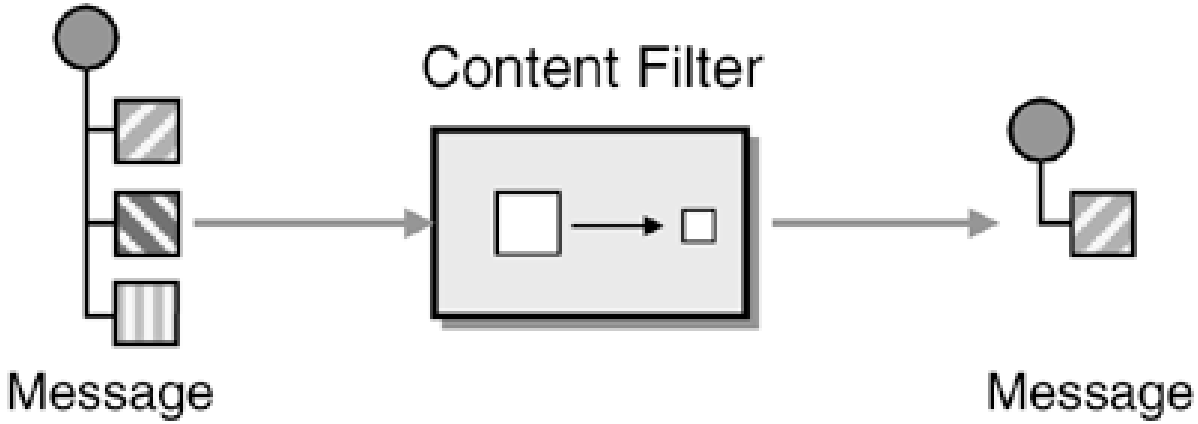
We allow the Content Enricher to query the customer care database to retrieve the desired data:



Hohpe and Woolf (2004), p. 340

In addition to data store retrieval, the Content Enricher may be able to compute the required data (e.g. a total) or extract it from the environment (e.g. a timestamp).

Use a Content Filter *to remove unimportant data items from a message, leaving only important items.*



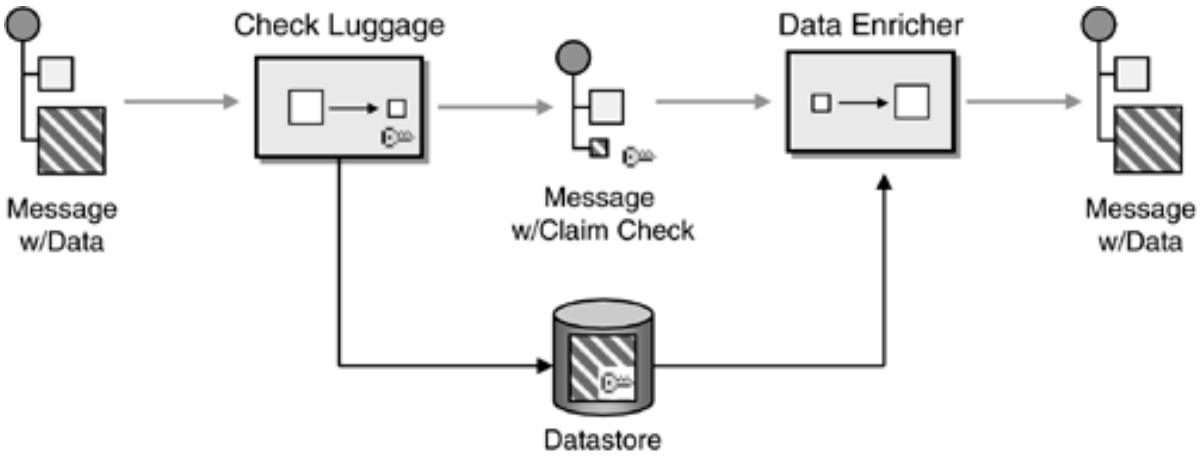
Hohpe and Woolf (2004), p. 343

Useful for security, simplification of recipient logic, and reduction of network traffic.

Message Transformation

Claim Check

Store message data in a persistent store and pass a Claim Check *to subsequent components. These components can use the Claim Check to retrieve the stored information.*



Hohpe and Woolf (2004), p. 347

Message Transformation

Claim Check

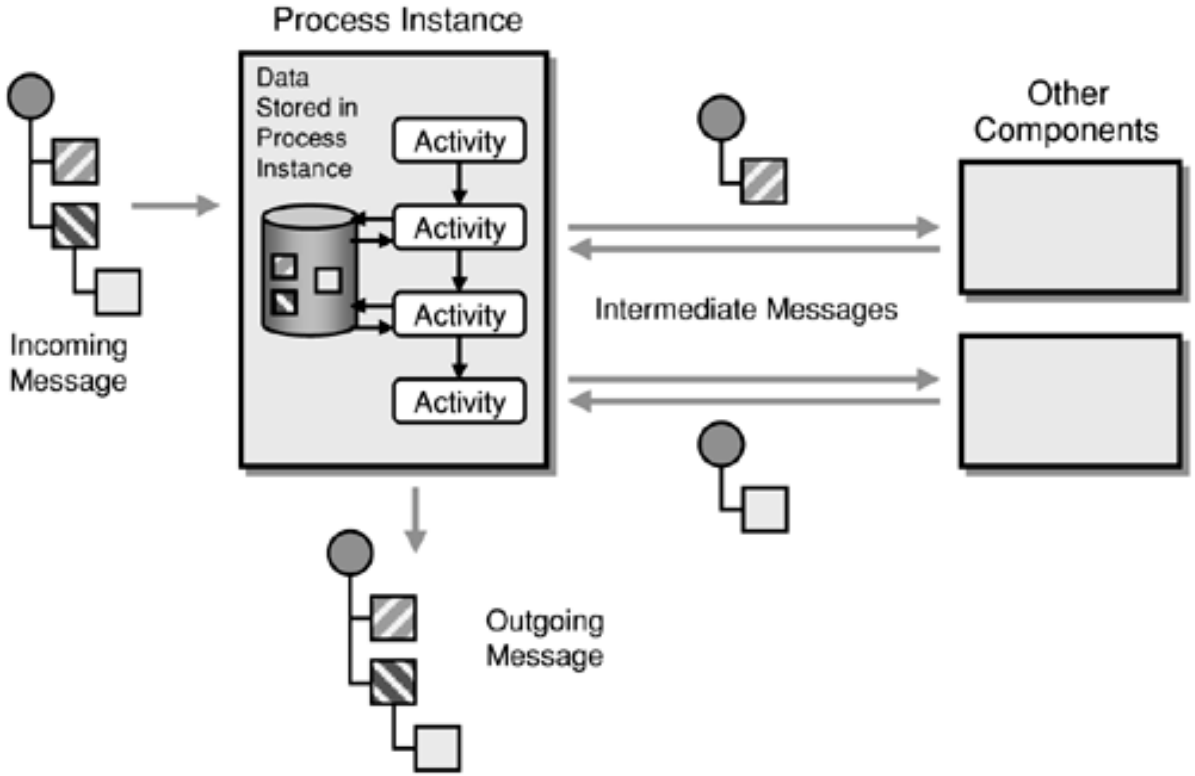
With Claim Check, we remove portions of a message, replace them with unique keys that can be used to query the data store as needed.

Useful for security, reduction of network traffic, and reduction of processing time.

Message Transformation

Claim Check

The state information stored by a Process Manager can be considered a case of Claim Check.

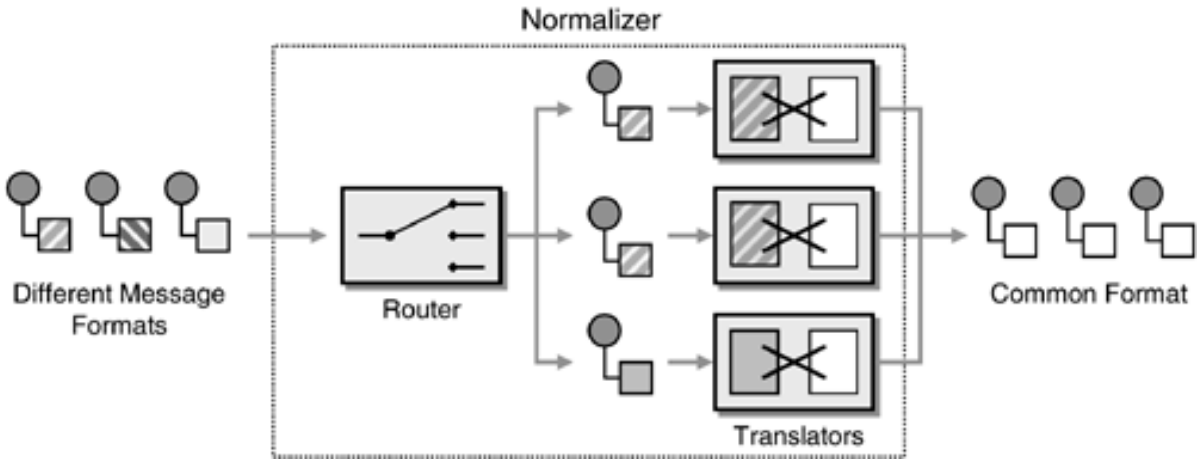


Hohpe and Woolf (2004), p. 351

Message Transformation

Normalizer

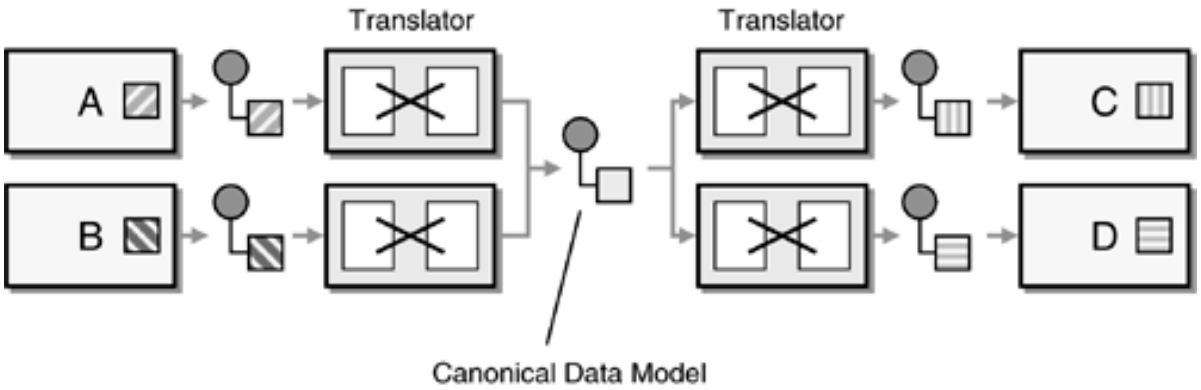
Use a Normalizer to route each message type through a custom Message Translator so that the resulting messages match a common format.



Hohpe and Woolf (2004), p. 353

Useful for dealing with multiple business partners that do not agree on a common format, or when we have multiple input channels (email, telephone, etc.).

Design a Canonical Data Model *that is independent from any specific application. Require each application to produce and consume messages in this common format.*



Hohpe and Woolf (2004), p. 356

Message Transformation

Canonical Data Model

Canonical Data Model does for message formats the same thing Message Broker does for message endpoints. It turns spaghetti into a hub-and-spoke architecture.

Canonical Data Model will normally require either external Message Translators or Messaging Mappers built in to the application domain layers.

Main problems:

- Very hard to design and make everyone happy.
- Overhead: instead of one point-to-point translation, in general, we have two translations for simple connections.

Like Message Broker, Canonical Data Model starts to pay off as the number of applications (and point-to-point connections) grows.

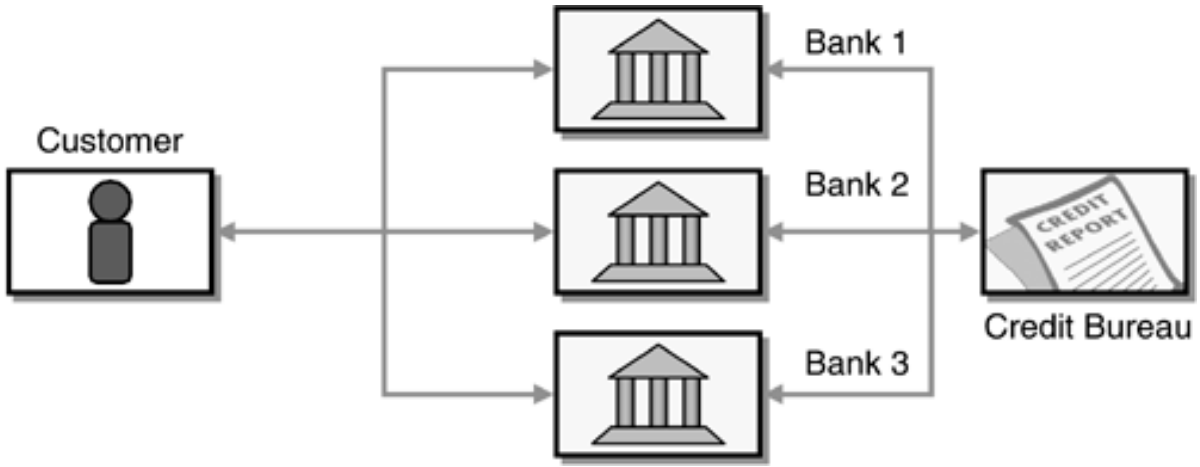
Outline

Case Study: Loan Broker

Introduction

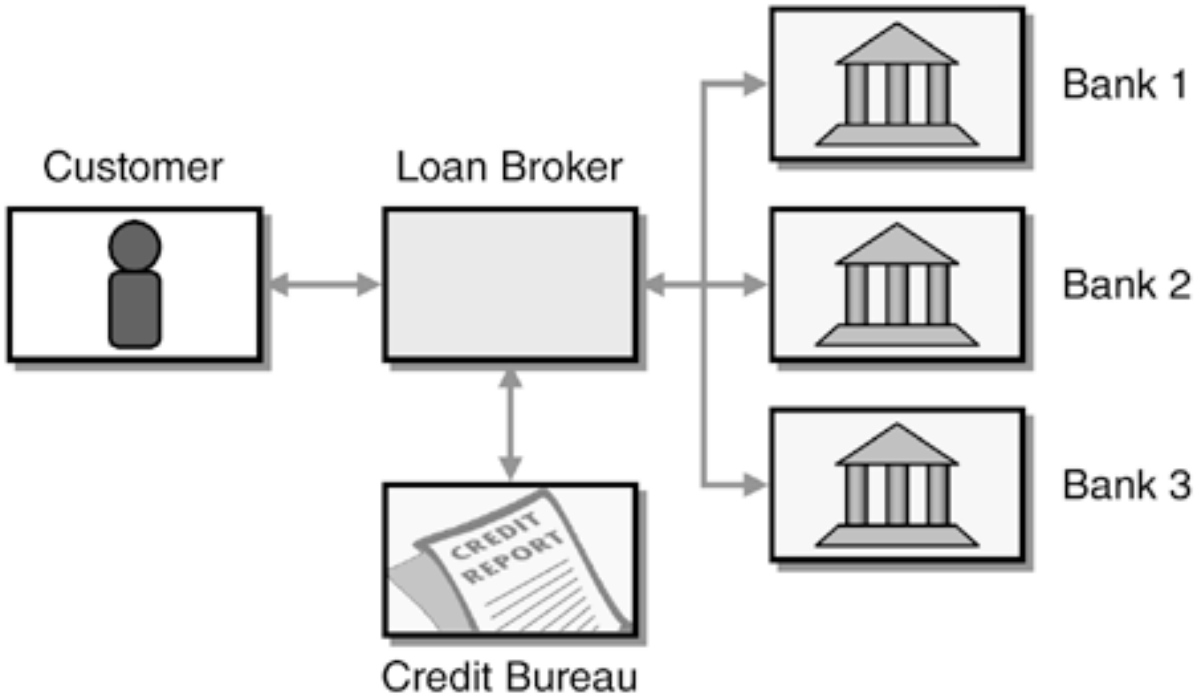
Customers looking for a loan often call several banks to get the best interest rate.

Each bank would normally gather some information from the customer, obtain the customer's credit report, then make a quote.



Hohpe and Woolf (2004), p. 362

A loan broker would make the process less tedious for the customer:



Hohpe and Woolf (2004), p. 362

Case Study: Loan Broker

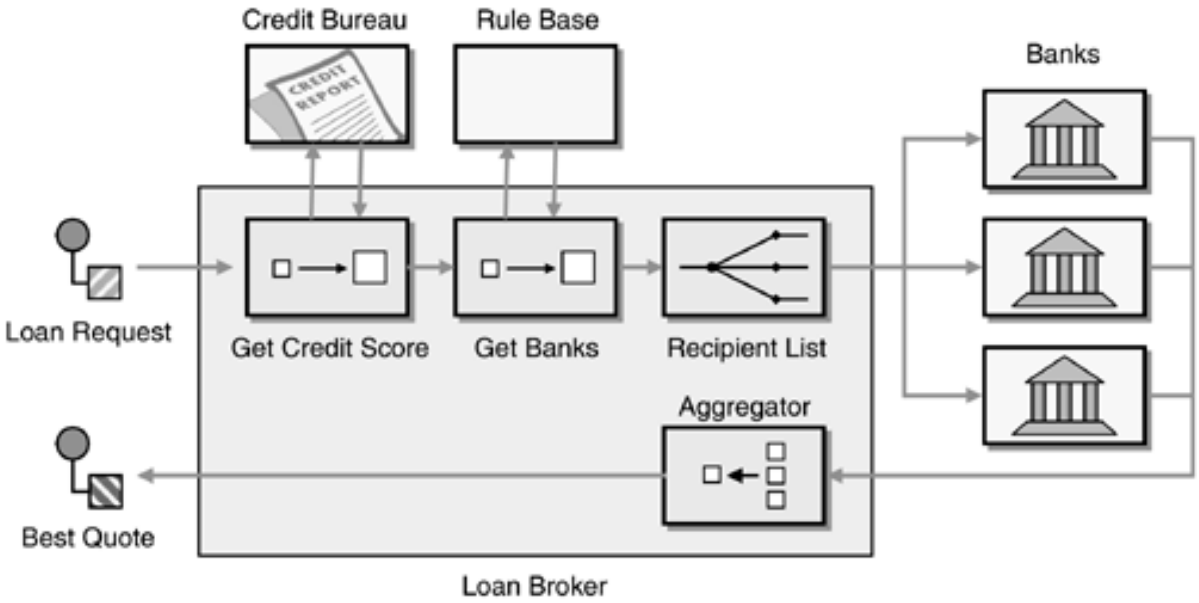
Process flow

The loan broker's internals should look something like this:

- Obtain a request for a quote: Message Channel.
- Get the customer's credit history: Content Enricher.
- Make a list of banks: Content Enricher.
- Send a request for quotation to each bank: Scatter-Gather, using Recipient List or Publish-Subscribe Channel.
- Collect the responses: Scatter-Gather, using Aggregator.
- Determine the best quote: Scatter-Gather, using Aggregator.
- Pass the result back to the customer: Message Channel.

Case Study: Loan Broker

Process flow

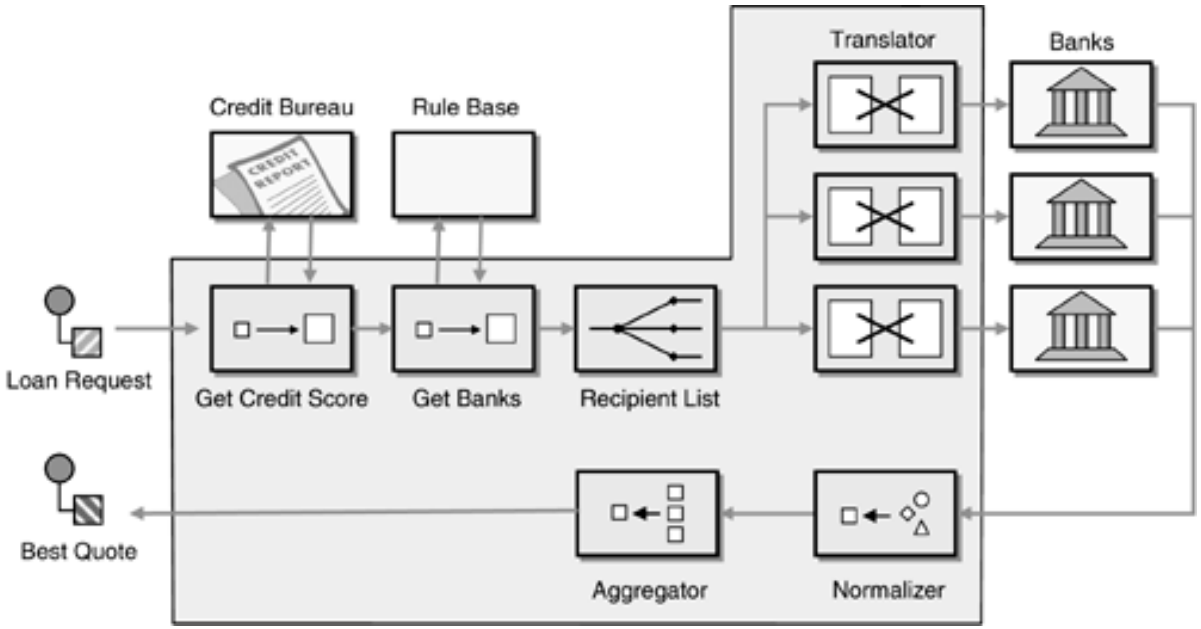


Hohpe and Woolf (2004), p. 363

Case Study: Loan Broker

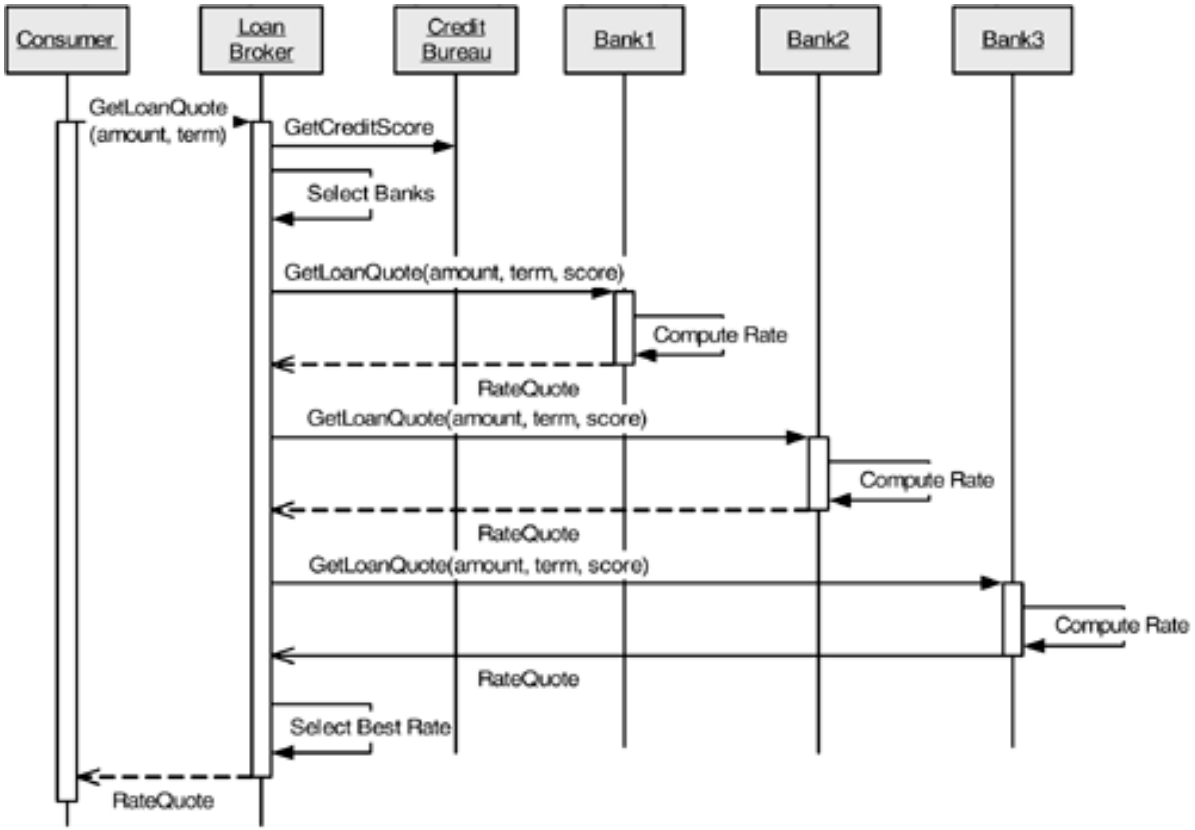
Process flow

The banks, however, probably do not agree on message formats so a set of Message Translators and a Normalizer will be required:



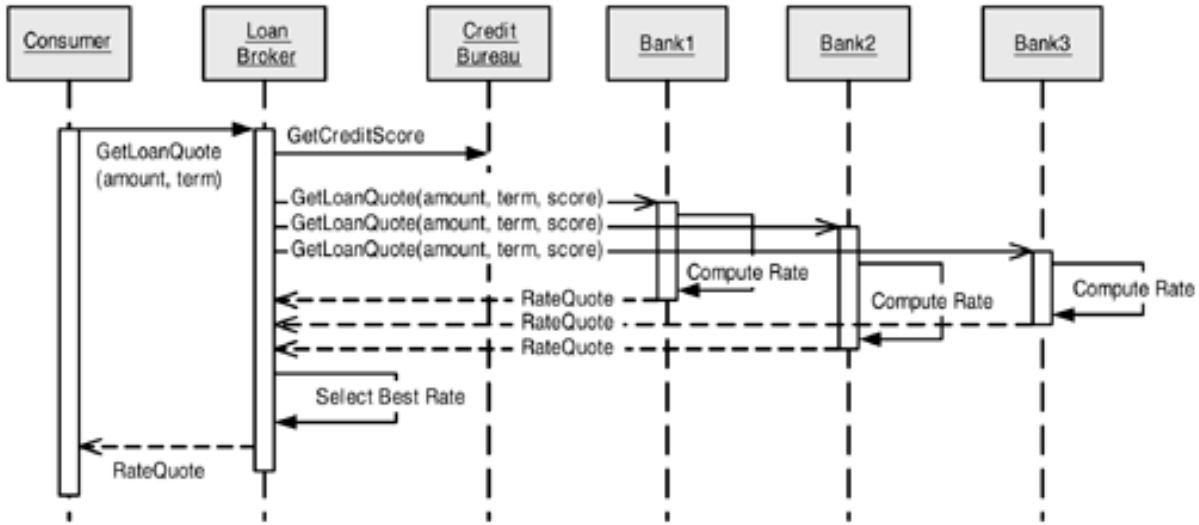
Hohpe and Woolf (2004), p. 364

How should we handle the sequence of requests to the banks? One possibility (with obvious problems) is synchronous invocation:



Hohpe and Woolf (2004), p. 365

A better solution is asynchronous invocation:



Hohpe and Woolf (2004), p. 366

Case Study: Loan Broker

Selecting banks and getting quotes

How to get the request for quote to the right banks?

- Fixed: always send to the same fixed set of banks.
- Distribution: the broker decides who to send the request to based on customer characteristics and terms of the request.
- Auction: the broker publishes to a Publish-Subscribe Channel and banks are free to subscribe and "bid" for each request.

Decision depends on many factors, especially the business relationship between the broker and the banks.

Case Study: Loan Broker

Concurrency

How to handle multiple concurrent requests?

- Multiple instances: one process is assigned to each request, usually from a pool of waiting instances.
- Single, event-driven instance: one process handles all requests concurrently, activated each time a message or request is received.

For the loan broker, since we spend most of our time blocking while waiting for responses to arrive from the credit agency or banks, the lightweight event-driven approach is preferred.

In either case, since message channels will be shared between concurrent requests, we need Correlation Indentifiers to associate messages with customer requests.

Case Study: Loan Broker

Concurrency

See Chapter 9 of the text for implementations with different approaches and messaging systems:

- Synchronous version with Java/Apache Axis (SOAP Web services)
- Asynchronous distribution list with point-to-point message queues using C#/Microsoft MSMQ
- Asynchronous auction with publish-subscribe channels using TIBCO Active Enterprise.

Outline

Messaging Endpoints

Introduction

Any application code that is interacting with the messaging system is part of a messaging endpoint.

Some of the endpoint patterns apply equally to message producers and message consumers.

As message receipt is more complex than message sending, many patterns apply only to message consumers.

These patterns apply equally well to message producers and message consumers:

- To encapsulate the messaging code and reduce coupling between the application and messaging system, use Messaging Gateway.
- For data translation between application data structures and message formats, use Messaging Mapper.
- To make a message send or receive part of an atomic transaction with other messaging events or database operations, use Transactional Client.

Messaging Endpoints

Message consumer patterns

Message consumption is more varied than message production:

- Is receipt synchronous or asynchronous? Use Polling Consumer or Event-Driven Consumer.
- Are messages assigned to recipients or grabbed by recipients? Use Competing Consumers or Message Dispatcher.
- Do we accept all messages or filter the messages we receive? Use Selective Consumer.
- Do we want our subscription to a Publish-Subscribe Channel to persist when we are disconnected? Use Durable Subscriber.
- Is there a possibility of being confused by receiving the same message twice? Use Idempotent Receiver.
- Do we want to provide access to existing synchronous services through the messaging system? Use Service Activator.

Outline

Debugging loosely coupled asynchronous systems is hard!

System management solutions keep track of things like throughput and latency, without any domain logic.

Business activity monitoring (BAM) solutions summarize business aspects of the system: total revenue, etc. BAM is closely related to data wharehousing.

Here we just briefly mention some of the basic system managment patterns.

How can we control multiple components in a distributed system?

A Control Bus provides management and monitoring functionality from a single console.

Some paths through the system may affect performance, so we may want to turn them on and off. Detour provides this functionality.

How can we observe and analyze message flow in a system?

Wire Tap inspects messages without altering the flow.

Message History tracks individual messages through system components.

Message Store keeps track of every message produced and consumed.

Smart Proxy allows tracking of the messages being send to and received from a request-reply service.

How can we do isolated testing of components in a distributed system?

A Test Message can be injected into the system.

A Channel Purger can be used to remove existing messages from a channel, so that testing can begin with a clean system.

Outline

See Chapter 12 for a case study on instrumenting the loan broker application with system management functionality.

See Chapter 13 for a real-world case study on a project to develop a bond trading system.

Conclusion

Industry trends

The industry is moving beyond "simple" EAI to emphasize designing systems from the beginning with loose coupling and asynchrony in mind.

Service oriented analysis and design adapts some of the analysis and design techniques we already know to modern service-oriented architectures.

We are moving beyond proprietary messaging systems to open standards based on SOAP.

- Main problem: HTTP is inherently unreliable, but asynchronous messaging solutions require reliable delivery.
- Standards such as WS-ReliableMessaging, WS-Coordination, and WS-Transaction aim to address this limitation.

Conclusion

Process management

Ultimately, the industry wants to allow business analysts to build process models in a visual tool then automatically generate and deploy a complete implementation.

Many EAI vendors provide proprietary notations for specifying business process implementations and process management engines to execute those processes.

BPEL is a cross platform specification language based on Web services, from Microsoft and IBM. It is still evolving.

Conclusion

Up and coming service oriented architecture technology

Service component architecture (SCA) and service data objects (SDO) are new standards from the Open SOA alliance (a group containing every major vendor except Microsoft).

SCA is a standard for how components (local or remote) can be put together to form a composite application.

Components could be implemented using C++, Java, COBOL, BPEL, etc. A component provides a set of services to clients.

Clients can bind to services using SOAP, JMS, EJB, and so on.

Components can be composed into composites whose services can likewise be accessed through a variety of bindings.