




# **Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions**



# Chapter 1: Solving Integration Problems Using Patterns

# Introduction

- The Need for Integration
- Integration Challenges
- How Integration Patterns Can Help?
  - The “patterns” are not copy-paste code samples or shrink-wrap components, but rather nuggets of advice that describe solutions to frequently recurring problems.
  - Used properly, the integration patterns can help fill the wide gap between the high-level vision of integration and the actual system implementation.

# Tight coupling vs Loosely Coupled

- Connect the two systems is through the TCP/IP protocol

```
String hostName = "www.eaipatterns.com";
int port = 80;

IPHostEntry hostInfo = Dns.GetHostByName(hostName);
IPAddress address = hostInfo.AddressList[0];

IPEndPoint endpoint = new IPEndPoint(address, port);

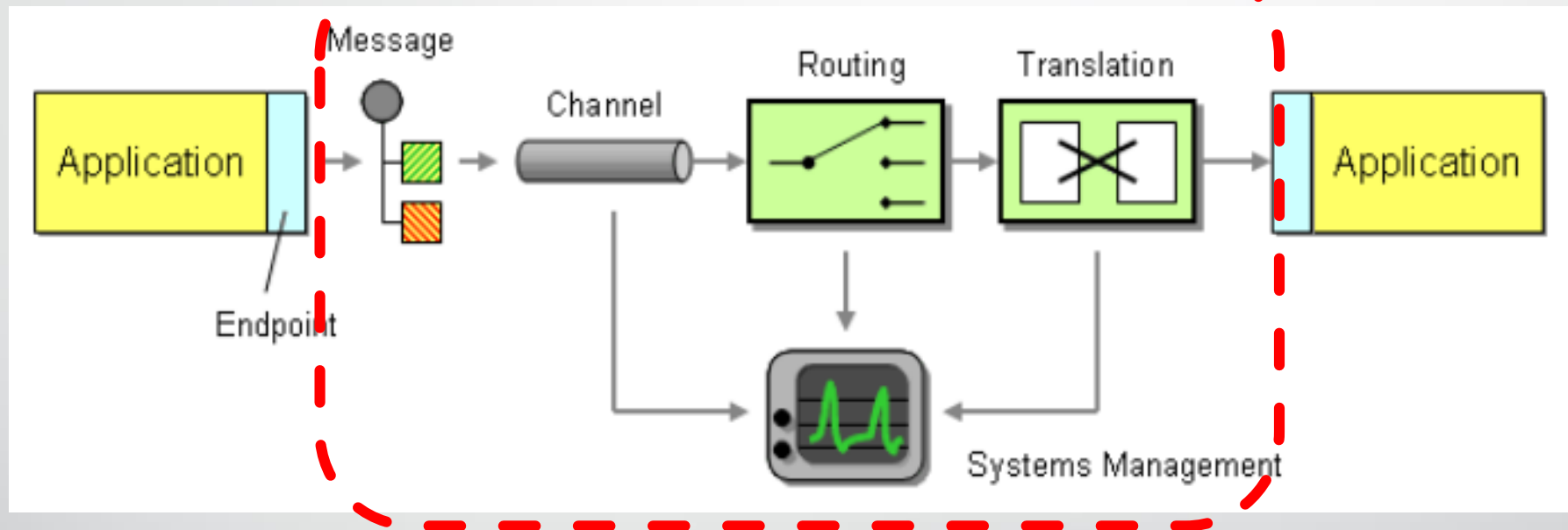
Socket socket = new Socket(address.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
socket.Connect(endpoint);
byte[] amount = BitConverter.GetBytes(1000);
byte[] name = Encoding.ASCII.GetBytes("Joe");
int bytesSent = socket.Send(amount);
bytesSent += socket.Send(name);

socket.Close();
```

# Tight coupling

- In summary, our minimalist integration solution is fast and cheap, but it results in a very brittle
- solution because the two participating parties make the following assumptions about each other:
  - Platform Technology – internal representations of numbers and objects
  - Location – hard-coded machine addresses
  - Time – all components have to be available at the same time
  - Data Format – the list of parameters and their types must match

# A Loosely Coupled Integration Solution



In order to connect two systems via an integration solution, a number of things have to happen. These things make up what we call middleware – the things that sit between applications.

# A Loosely Coupled Integration Solution (cont'd)

- Communications **channel** that can move information from one application to the other. This channel could be a series of TCP/IP connections, a shared file, a shared database or a floppy disk being carried from one computer to the next.
- A snippet of data that has an agreed-upon meaning to both applications that are to be integrated. This piece of data can be very small, such as the phone number of a single customer that has changed, or very large, such as the complete list of all customers and their associated addresses. We call this piece of data a **message**.


# A Loosely Coupled Integration Solution (cont'd)

- Because the internal data format of an application can often not be changed the middleware needs to provide some mechanism to convert one application's data format in the other's. We call this step **translation**.
  - Like Customer Full Name vs Customer First Name and Last Name
- What happens if we integrate more than two systems? Where does the data have to be moved?
  - Things would be a lot easier if the middleware could take care of sending messages to the correct places. This is the role of a **routing** component such as a message broker.



# A Loosely Coupled Integration Solution (cont'd)

- In order to have any idea what is going on inside the system we need a **systems management** function. This subsystem monitors the flow of data, makes sure that all applications and components are available and reports error conditions to a central location.
- Most packaged and legacy applications and many custom applications are not prepared to participate in an integration solution. We need a message **endpoint** to connect the system explicitly to the integration solution. The endpoint can be a special piece of code or a Channel Adapter provided by an integration software vendor.



# Chapter 2: Integration Styles

# Application Integration Options

- **File Transfer**

- An enterprise has multiple applications that are being built independently, with different languages and platforms.
- To have any chance of getting your head around it, you must minimize what you need to know about how each application works
- Produce the files at regular intervals according to the nature of the business.
- The modern fashion is to use XML.
- ❖ Updates tend to occur infrequently, as a result systems can get out of synchronization

# Application Integration Options (cont'd)

- **Shared Database**

- The enterprise needs information to be shared rapidly and consistently.
- If a family of integrated applications all rely on the same database, then you can be pretty sure that they are always consistent all of the time.
- ❖ Coming up with a unified schema that can meet the needs of multiple applications is a very difficult exercise
- ❖ Human conflicts between departments often exacerbate this problem.
- ❖ cause performance bottlenecks and even deadlocks as each application locks others out of the data

# Application Integration Options (cont'd)

- **Remote Procedure Invocation**

- The enterprise needs to share data and processes in a responsive way
- What is needed is a mechanism for one application to invoke a function in another application
- ❖ There are big differences in performance and reliability between remote and local procedure calls
- ❖ The applications are still fairly tightly coupled together

# Application Integration Options (cont'd)

- **Messaging**

- File Transfer and Shared Database enable applications to share their data, but not their functionality.
- Remote Procedure Invocation enables applications to share functionality, but tightly couples them in the process
- Asynchronous messaging is fundamentally a pragmatic reaction to the problems of distributed systems
- Sending a message does not require both systems to be up and ready at the same time.



# Chapter 3: Messaging Systems

# Introduction

- Messaging makes applications loosely coupled by communicating asynchronously
- Also makes the communication more reliable because the two applications do not have to be running at the same time



# Basic Messaging Concepts

- **Channels** — Messaging applications transmit data through a Message Channel, a virtual pipe that connects a sender to a receiver.
- **Messages** — A Message is an atomic packet of data that can be transmitted on a channel. Thus to transmit data, an application must break the data into one or more packets, wrap each packet as a message, and then send the message on a channel. Likewise, a receiver application receives a message and must extract the data from the message to process it.

# Basic Messaging Concepts (cont'd)

- **Multi-step delivery** — In the simplest case, the message system delivers a message directly from the sender's computer to the receiver's computer. However, actions often need to be performed on the message after it is sent by its original sender but before it is received by its final receiver.
- **Routing** — In a large enterprise with numerous applications and channels to connect them, a message may have to go through several channels to reach its final destination.

# Basic Messaging Concepts (cont'd)

- **Transformation** — Various applications may not agree on the format for the same conceptual data; the sender formats the message one way, yet the receiver expects it to be formatted another way.
- **Endpoints** — An application does not have some built-in capability to interface with a messaging system. Rather, it must contain a layer of code that knows both how the application works and how the messaging system works, bridging the two so that they work together.

# Message Channel

- The messaging system isn't a big bucket that applications throw information into and pull information out of. It's a set of connections that enable applications to communicate by transmitting information in predetermined, predictable ways.
- When an application has information to communicate, it doesn't just fling the information into the messaging system, it adds the information to a particular *Message Channel*.
- An application receiving information doesn't just pick it up at random from the messaging system; it retrieves the information from a particular Message Channel.

# Message Channel (cont'd)

- The messaging system has different Message Channels for different types of information the applications want to communicate. When an application sends information, it doesn't randomly add the info to any channel available; it adds the info to a channel whose specific purpose is to communicate that sort of information.
- Channels are logical addresses in the messaging system

# Message Channel (cont'd)

Alternatively, you can create the queue using code:

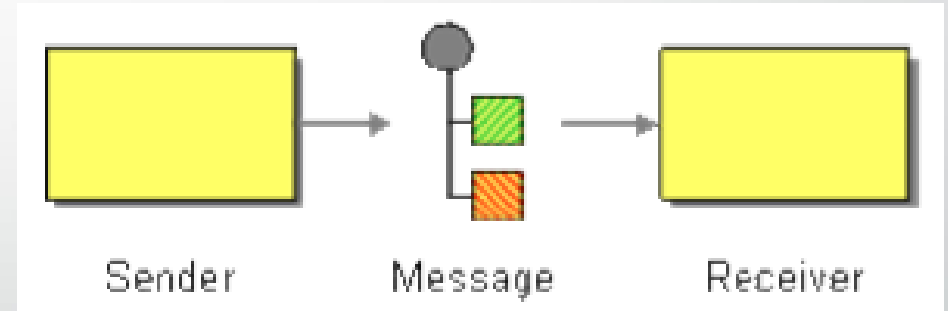
```
using System.Messaging;  
...  
MessageQueue.Create("MyQueue");
```

Once the queue is created, an application can access it by creating a `MessageQueue` instance:

```
MessageQueue mq = new MessageQueue("MyQueue");
```

# Message

- Data isn't one continuous stream; it is units, such as records, objects, database rows, and the like.
- Channel must transmit units of data
- Any data that is to be transmitted via a messaging system must be converted into one or more messages that can be sent through messaging channels.



# Message (cont'd)

- A message consists of two basic parts:
  1. **Header** – Information used by the messaging system that describes the data being transmitted, its origin, its destination, and so on.
  2. **Body** – The data being transmitted; generally ignored by the messaging system and simply transmitted as-is.

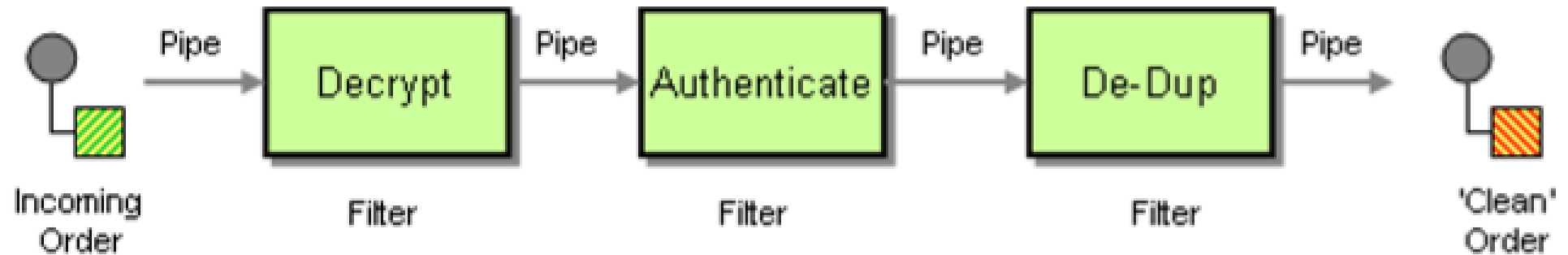


# Pipes and Filters

- In many enterprise integration scenarios, a single event triggers a sequence of processing steps, each performing a specific function.
  - Authentication, Decryption, ...
- A component can send a message to another component for further processing without waiting for the results.
- Using this technique, we could process multiple messages in parallel, on inside each component.

## Pipes and Filters (cont'd)

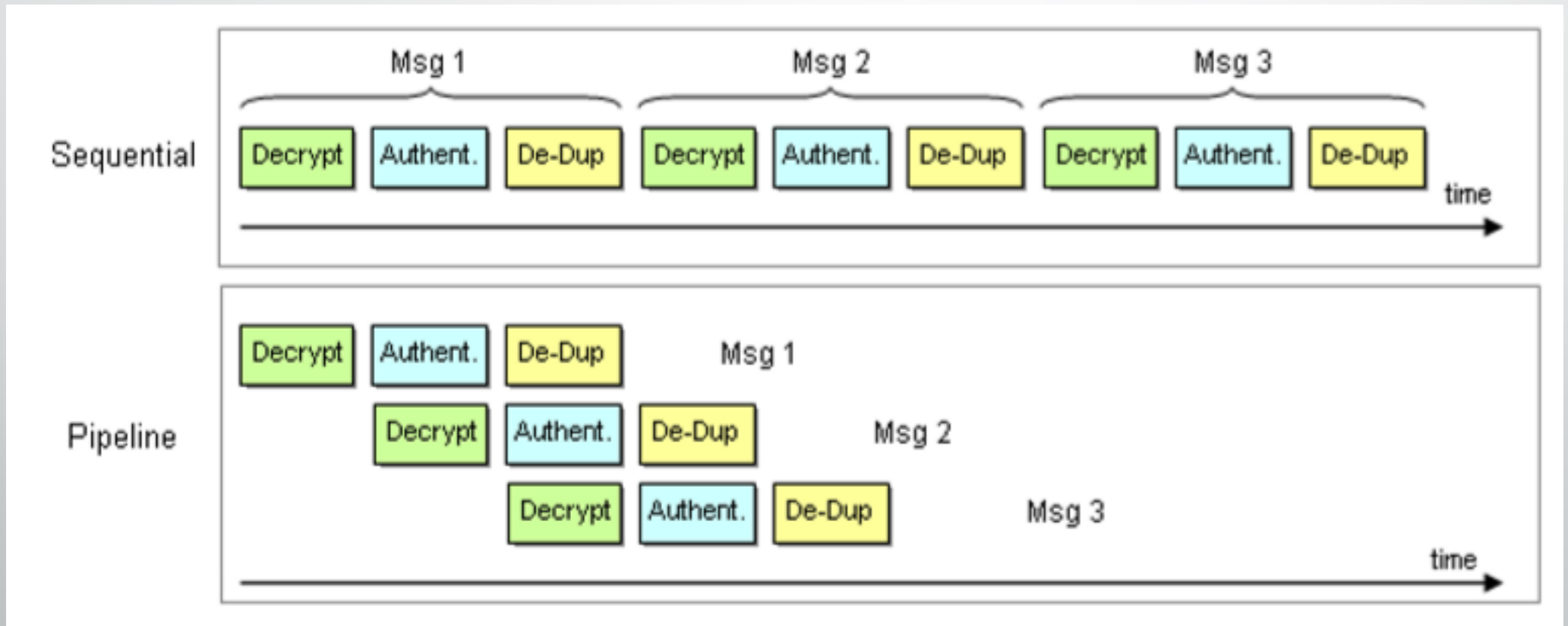
Use the *Pipes and Filters* architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes).



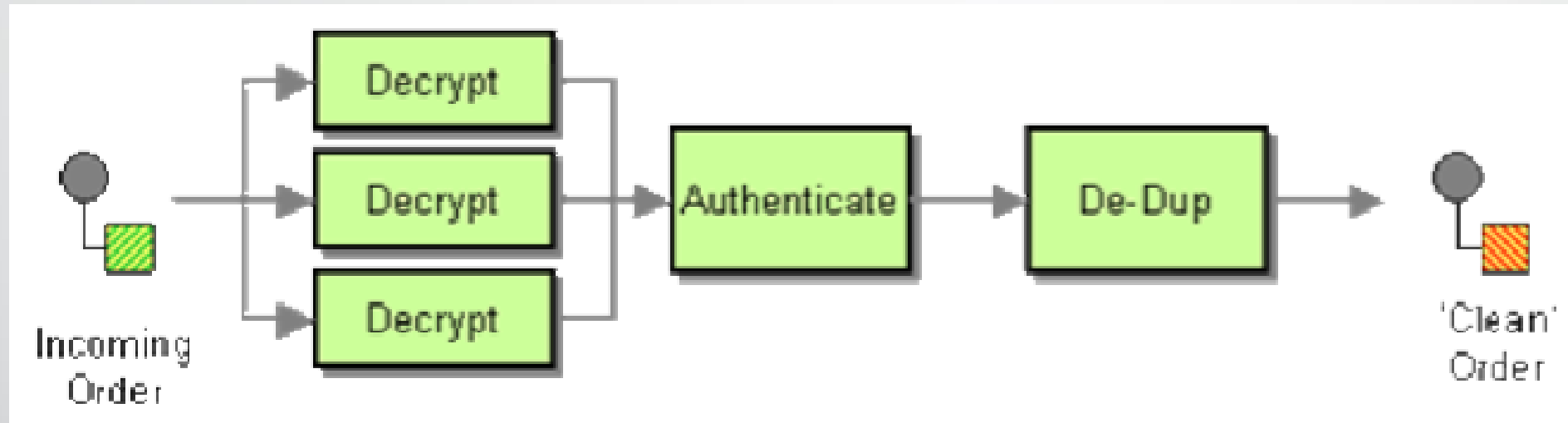
# Pipes and Filters (cont'd)

- Each filter exposes a very simple interface: it receives messages on the inbound pipe, processes the message, and publishes the results to the outbound pipe.
- We can add new filters, omit existing ones or rearrange them into a new sequence -- all without having to change the filters themselves.
- Many patterns, e.g. routing and transformation patterns, are based on this Pipes and Filters architectural style.

# Pipeline Processing



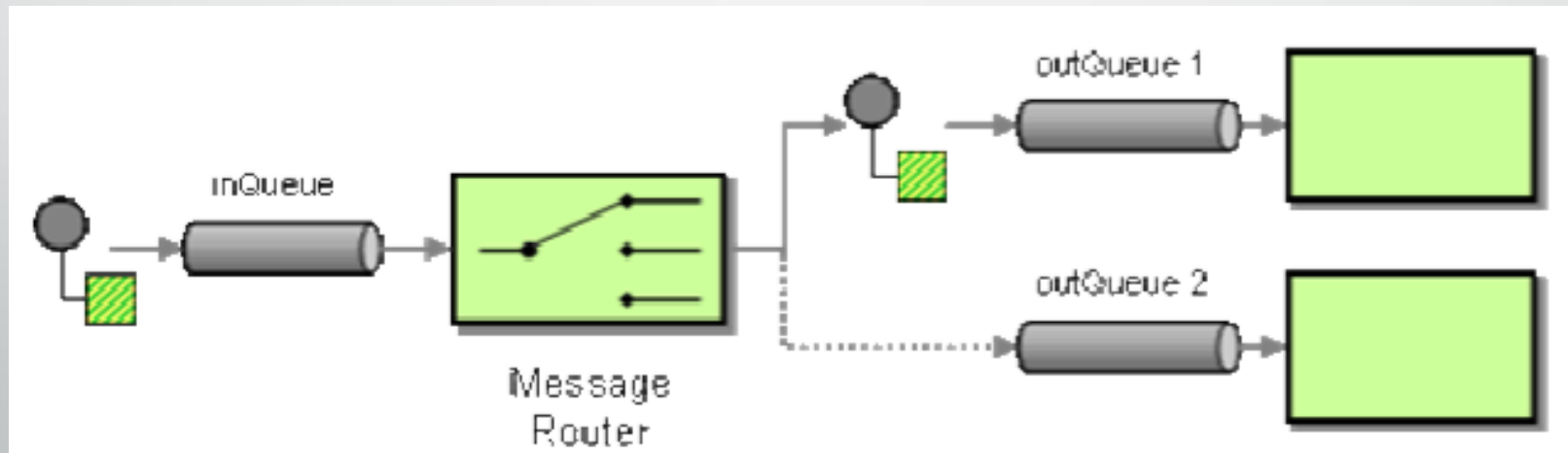
# Parallel Processing



- This configuration can cause messages to be processed out of order.
- Parallelizing filters works best if each filter is stateless

# Message Router

- How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?
- Insert a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel depending on a set of conditions.



# Message Router (cont'd)

- Thanks to the Pipes and Filters architecture the components surrounding the Message Router are completely unaware of the existence of a Message Router.
- Message Router does not modify the message contents. It only concerns itself with the destination of the message.
- The key benefit of using a Message Router is that the decision criteria for the destination of a message are maintained in a single location.

# Message Router (cont'd)

- It can degrade performance, a performance bottleneck.
  - By using multiple routers in parallel or adding additional hardware, this effect can be minimized.



# Message Router Variants

- Fixed router
  - Only a single input channel and a single output channel are defined
  - Useful to intentionally decouple subsystems
- Content-Based Router
  - Decide the message's destination only on properties of the message itself
- Context-Based Routers
  - Decide the message's destination based on environment conditions
  - used to perform load balancing
  - Test or failover functionality
- Stateless vs Stateful

# Message Translator

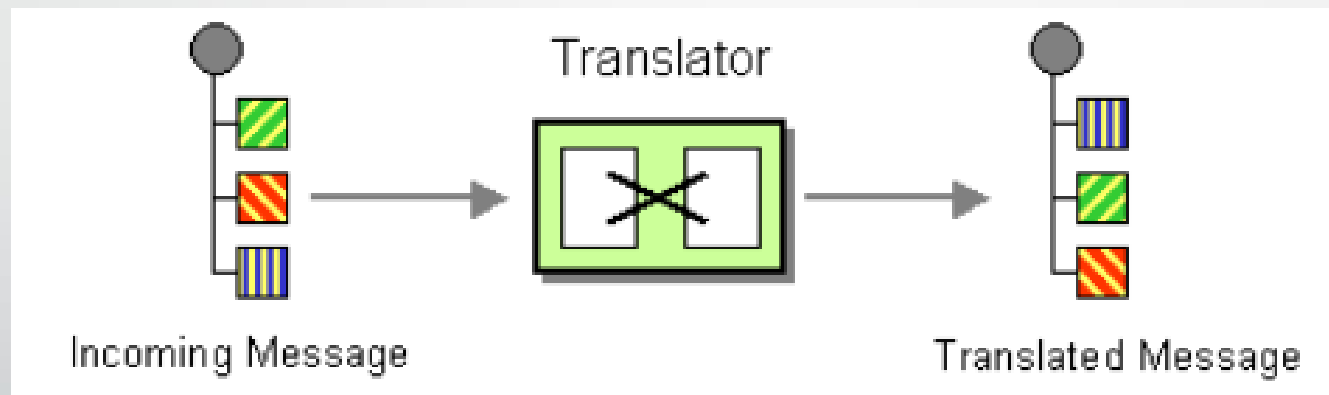
- How can systems using different data formats communicate with each other using messaging?
- We could avoid having to transform messages if we could modify all applications to use a common data format!!! (Difficult)
  - Changing an application's data format is risky, difficult, and requires a lot of changes to inherent business functionality
  - Data format changes are simply not economically feasible
  - **Modifying one application to match another application's data format would violate loose coupling because it makes two applications directly dependent on each other's internal representation**

# Message Translator (cont'd)

- We could incorporate the data format translation directly into the Message Endpoint.
  - Requires access to the endpoint code
  - Hard-coding the format translation to the endpoint would reduce the opportunities for code reuse

# Message Translator (cont'd)

Use a special filter, a Message Translator, between other filters or applications to translate one data format into another.



# Levels of Transformation

This is the domain of entity-relationship diagrams and class diagrams.

The application used by one department may divide the country into 4 regions: Western, Central, Southern and Eastern, identified by the letters 'W', 'C', 'S' and 'E'. Another department may differentiate the Pacific Region from the Mountain Region and distinguishes the Northeast from the Southeast. Each region is identified by a unique two-digit number. What number does the letter 'E' correspond to?

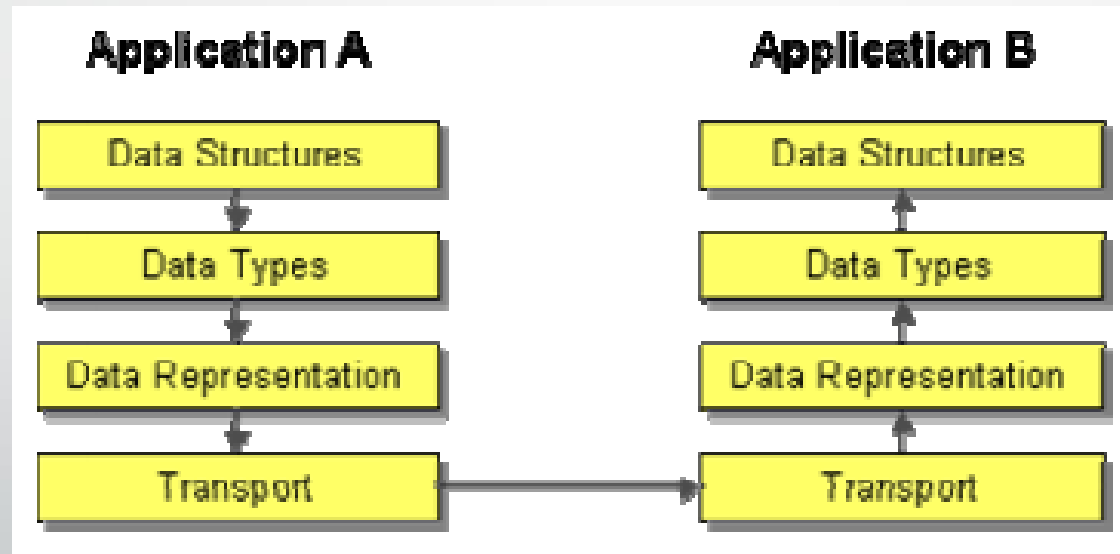
In order to interface systems with different data representations, data may have to be decrypted, uncompressed and parsed, then the new data format rendered, and possibly compressed and encrypted as well.

provides data transfer between the different systems. It is responsible for complete and reliable data transfer across different network segments and deals with lost data packets and other network errors.

| 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<br>Custom code                                  |
| Data Types                          | Field names, data types, value domains, constraints, code values   | Convert zip code from numeric to string.<br>Concatenate <i>first name</i> and <i>last name</i> fields to single <i>name</i> field.<br>Replace US state name with two character code. | EAI visual transformation editors<br>XSL<br>Database lookups<br>Custom code |
| Data Representation                 | Data formats (XML, name-value pairs, fixed-length data fields etc., EAI vendor formats)<br>Character sets (ASCII, UniCode, EBCDIC)<br>Encryption / compression | Parse data representation and render in a different format. Decrypt/encrypt as necessary.  | XML Parsers, EAI parser / renderer tools<br>Custom APIs                     |
| Transport                           | Communications Protocols: TCP/IP sockets, http, SOAP, JMS, TIBCO Rendez Vous   | Move data across protocols without affecting message content.  | <a href="#"><i>Channel Adapter</i></a><br>EAI adapters                      |

# Chaining Transformations

Many business scenarios require transformations at more than one layer




# Message Endpoint

- How does an application connect to a messaging channel to send and receive messages?
- The messaging endpoint code takes command or data, makes it into a message, and sends it on a particular messaging channel. It is the endpoint that receives a message, extracts the contents, and gives them to the application in a meaningful way.
- Encapsulates the messaging system from the rest of the application

# Message Endpoint (cont'd)

- An endpoint is channel-specific, so a single application would **use multiple endpoints to interface with multiple channels**.
- An application may use more than one endpoint to interface to a single channel, usually to support multiple concurrent threads.





# Chapter 4: Messaging Channels

# Introduction

- By selecting a particular channel to send the data on, the sender knows that the receiver will be one that is looking for that sort of data by looking for it on that channel.
- In this way, the applications that produce shared data have a way to communicate with those that wish to consume it.

# Message Channel Themes

- If an application has data to transmit or data it wishes to receive, it will have to use a channel.
- The challenge is knowing what channels your applications will need and what to use them for.

# Message Channel Themes (cont'd)

- **Fixed set of channels**
  - When designing an application, a developer has to know where to put what types of data to share that data with other applications, and likewise where to look for.
- **Determining the set of channels**
- **Unidirectional channels**

# Message Channel Decisions

- **One-to-one or one-to-many**
  - To send the data to a single application, use a Point-to-Point Channel
  - If you want all of the receiver applications to be able to receive the data, use a Publish-Subscribe Channel. When you send a piece of data this way, the channel effectively copies the data for each of the receivers

# Message Channel Decisions (cont'd)

- **What type of data**
  - *Datatype Channel* is the principle that all of the data on a channel has to be of the same type. This is the main reason why messaging systems need lots of channels;
- **Invalid and dead messages**
  - The message system can ensure that a message is delivered properly, but it cannot guarantee that the receiver will know what to do with it
  - Put the strange message on a specially designated *Invalid Message Channel*, in hopes that some utility monitoring the channel will pick up the message and figure out what to do with it.
  - A *Dead Letter Channel* for messages which are successfully sent but ultimately cannot be successfully delivered

# Message Channel Decisions (cont'd)

- **Crash proof**
  - If the messaging system crashes or is shut down for maintenance, what happens to its messages?
- **Non-messaging clients**
  - Sometimes the "non-messaging client" really is a messaging client, just for a different messaging system. In that case, an application that is a client on both messaging systems can build a Messaging Bridge between the two, effectively connecting them into one composite messaging system.

# Message Channel Decisions (cont'd)

- **Communications backbone**

- A new application simply needs to know which channels to use to request functionality and which others to listen on for the results
- The messaging system itself essentially becomes a Message Bus, a backbone providing access to all of the enterprise's various and ever-changing applications and functionality



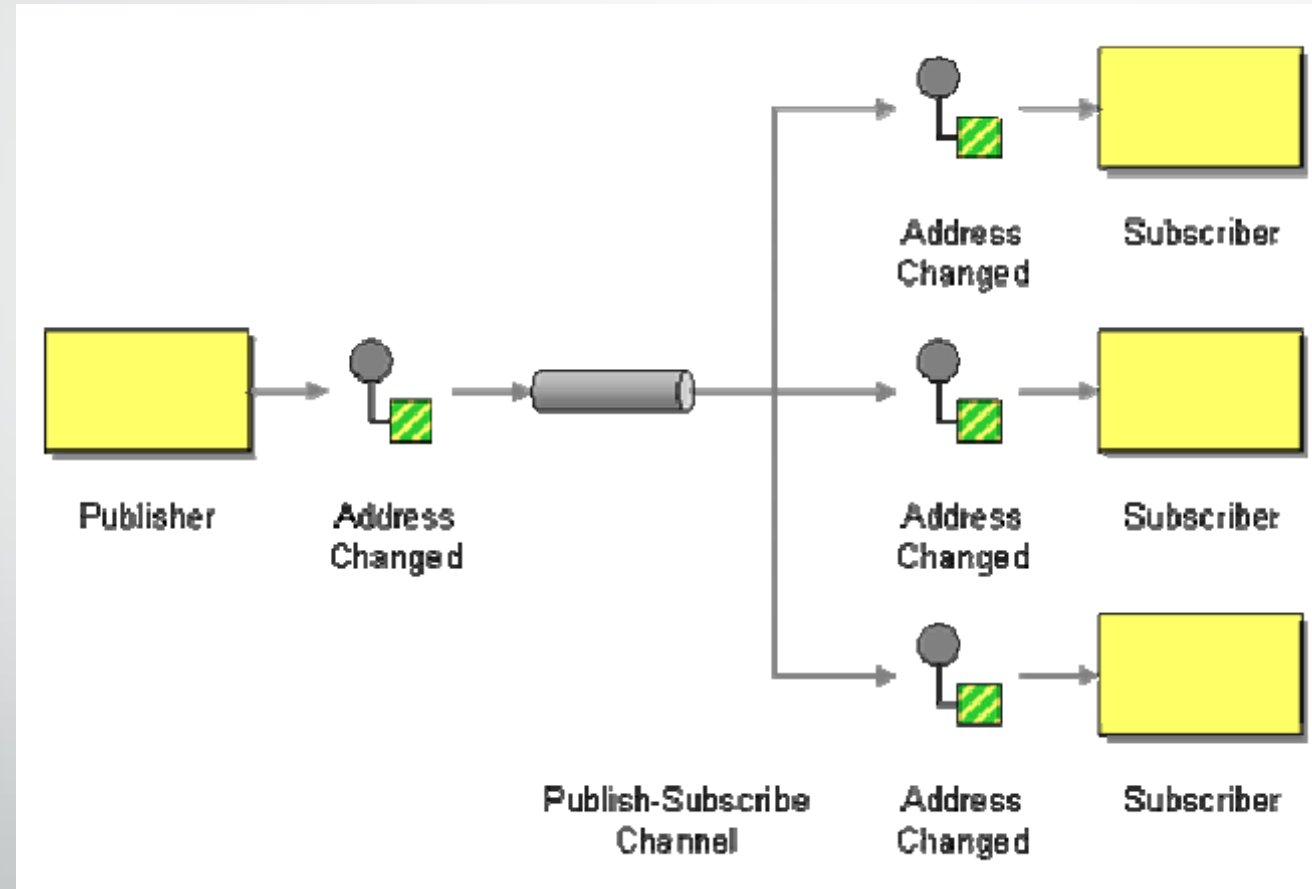
# Point-to-Point Channel

- How can the caller be sure that exactly one receiver will receive the document or perform the call?
- Once a call is packaged as a Message and placed on a Message Channel, potentially many receivers could see it on the channel and decide to perform the procedure.
- A Point-to-Point Channel ensures that only one receiver consumes any given message.

# Publish-Subscribe Channel

- How can the sender broadcast an event to all interested receivers?
- Each subscriber needs to be notified of a particular event once, but should not be notified repeatedly of the same event. The event cannot be considered consumed until all of the subscribers have been notified
- Send the event on a Publish-Subscribe Channel, which delivers a copy of a particular event to each receiver.

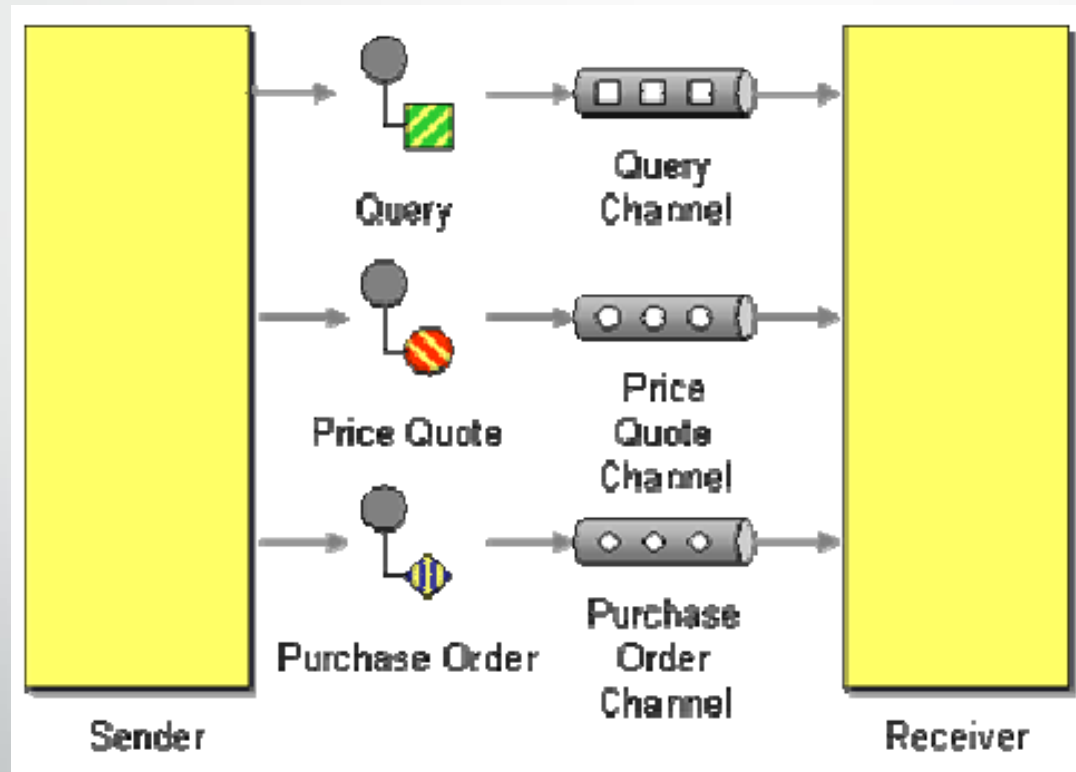
# Publish-Subscribe Channel (cont'd)



# Datatype Channel

- How can the application send a data item such that the receiver will know how to process it?
- All messages are just instances of the same message type, as defined by the messaging system, and the contents of any message are ultimately just a byte array.
- It is not specific enough for a receiver to be able to process a message's contents.
- The receiver must know what type of messages it's receiving, or it doesn't know how to process them

# Datatype Channel (cont'd)

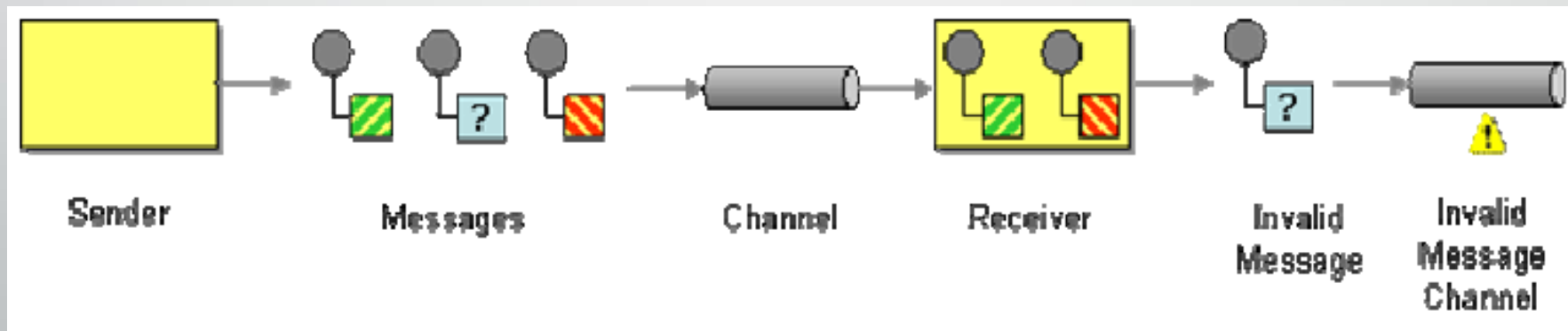


# Invalid Message Channel

- The message body may cause parsing errors, lexical errors, or validation errors. The message header may be missing needed properties. A sender might put a perfectly good message on the wrong channel. A malicious sender could purposely send an incorrect message just to mess-up the receiver.
- It could put the message back on the channel!
  - But then the message will just be re-consumed by the same receiver or another like it
- A way to clean improper messages out of channels and put them somewhere out of the way

# Invalid Message Channel (cont'd)

- The receiver should move the improper message to an Invalid Message Channel, a special channel for messages that could not be processed by their receivers.



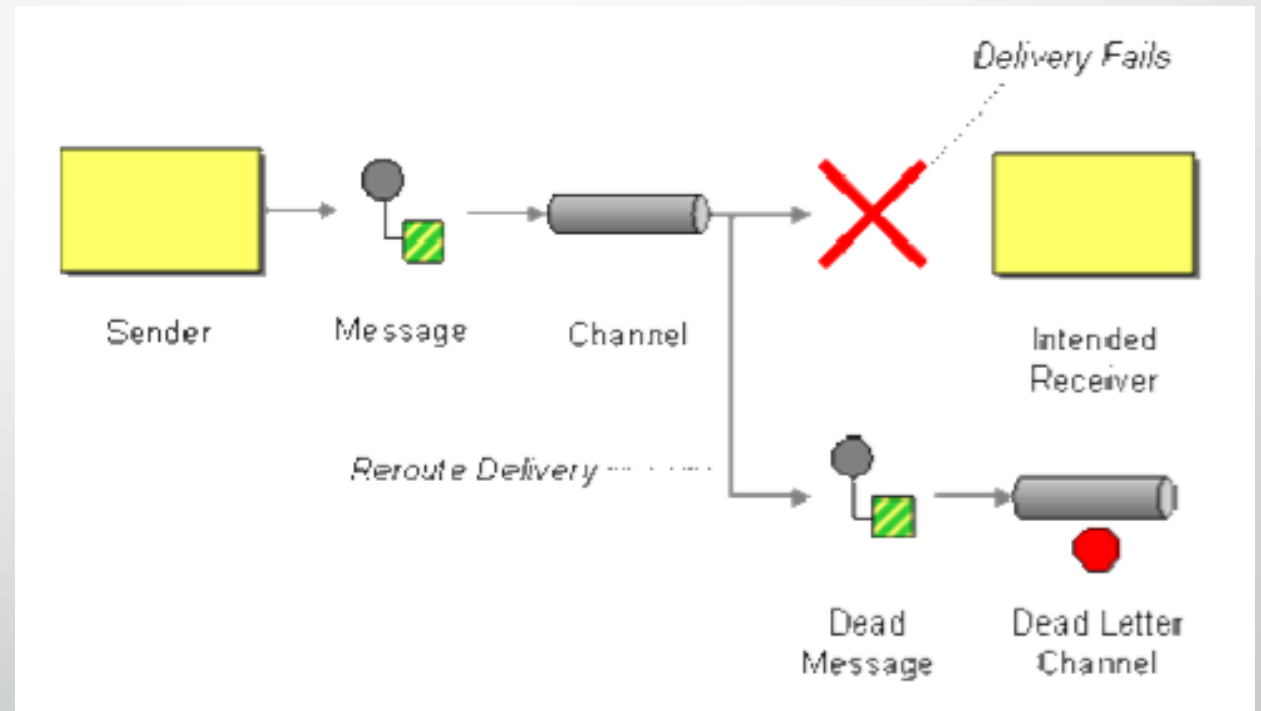
# Dead Letter Channel

- What will the messaging system do with a message it cannot deliver?
  - The message's channel may be deleted after the message is sent
  - The message may expire before it can be delivered
  - A message with a Selective Consumer that everyone ignores will never be read and may eventually die



## Dead Letter Channel (cont'd)

- When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a Dead Letter Channel.

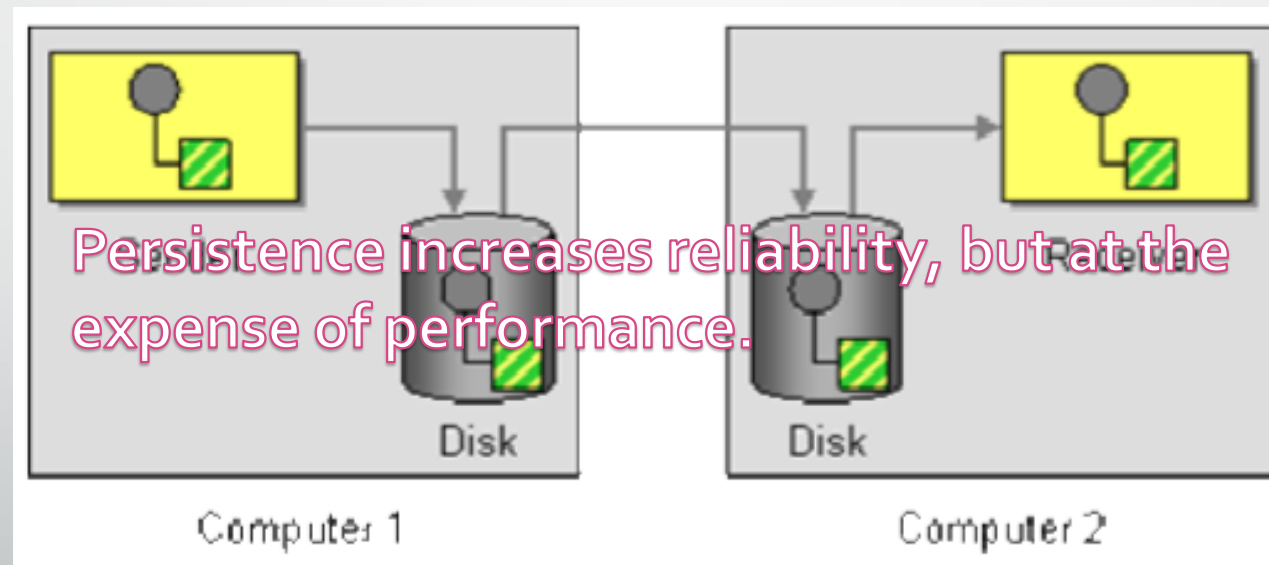


# Guaranteed Delivery

- How can the sender make sure that a message will be delivered, even if the messaging system fails?
- The store and forward process that messaging is based on
  - So where should the message be stored before it is forwarded? in memory?!!
  - If the messaging system crashes (for example, because one of its computers loses power or the messaging process aborts unexpectedly), all of the messages stored in memory are lost.

# Guaranteed Delivery (cont'd)

- Use files and databases to persist data to disk so that it survives system crashes.

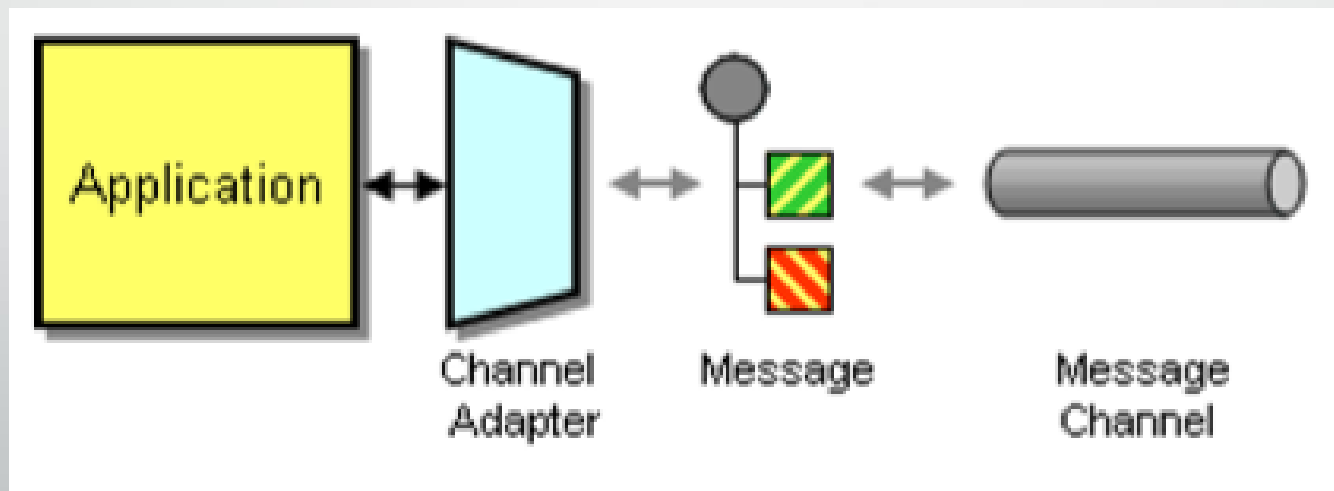


# Channel Adapter

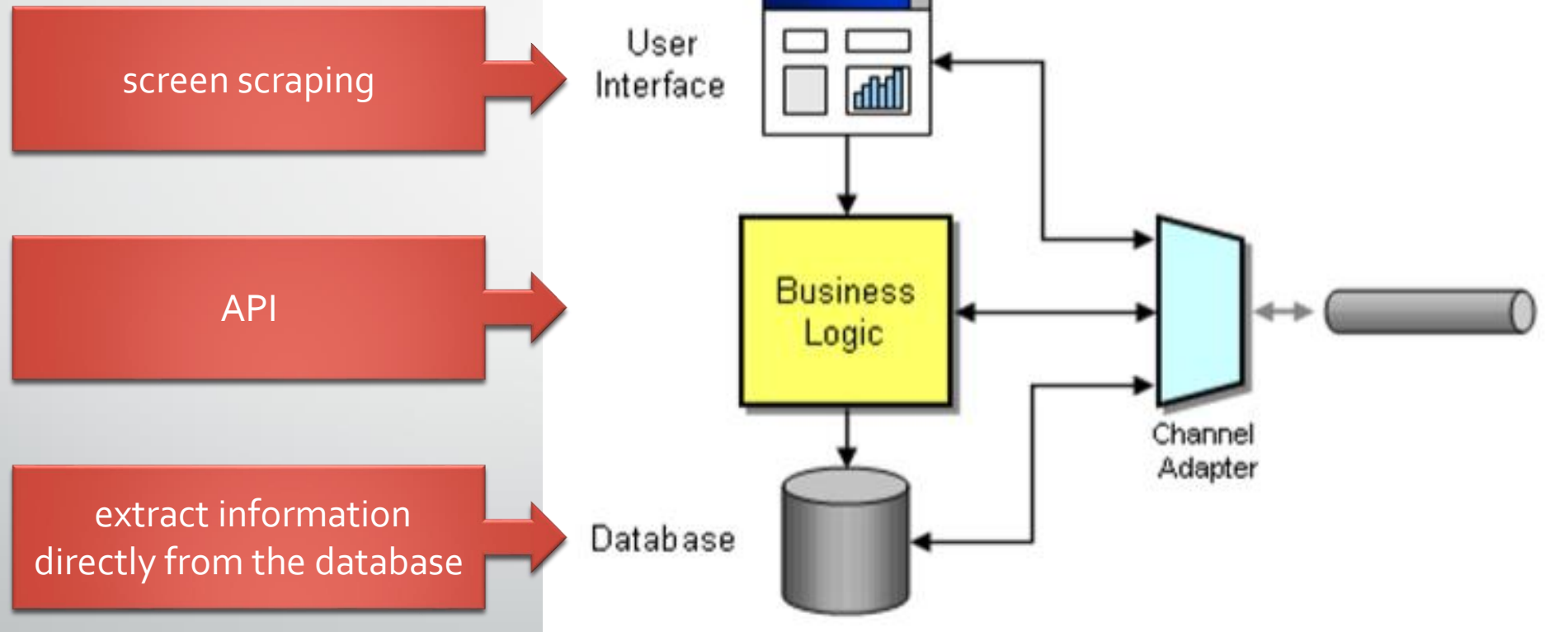
- How can you connect an application to the messaging system so that it can send and receive messages?
- Most applications were not designed to work with a messaging infrastructure.

# Channel Adapter (cont'd)

- Use a Channel Adapter that can access the application's API or data and publish messages on a channel based on this data, and that likewise can receive messages and invoke functionality inside the application.



# Channel Adapter (cont'd)



# Messaging Bridge

- How can multiple messaging systems be connected so that messages available on one are also available on the others?

Use a Messaging Bridge, a connection between messaging systems, to replicate messages between systems.

