**What Is Messaging?**

Messaging is a method of communication between software components or applications.

A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is *loosely coupled*. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only what message format and what destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application’s methods.

Messaging also differs from electronic mail (e-mail), which is a method of communication between people or between software applications and people.

Messaging is used for communication between software applications or software components.

An enterprise messaging system (EMS) or messaging system in brief is a set of published enterprise-wide standards that allows organizations to send semantically precise messages between computer systems.

Messaging makes applications loosely coupled by communicating asynchronously, which also makes the communication more reliable because the two applications do not have to be running at the same time. Messaging makes the messaging system responsible for transferring data from one application to another, so the applications can focus on what data they need to share as opposed to how to share it.

**Basic Messaging Concepts**

**Channels**—Messaging applications transmit data through a *Message Channel* , a virtual pipe that connects a sender to a receiver. A newly installed messaging system typically doesn’t contain any channels; you must determine how your applications need to communicate and then create the channels to facilitate it.

**Messages**— *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. The message system will try repeatedly to deliver the message (e.g., transmit it from the sender to the receiver) until it succeeds.

**Pipes and Filters**—In the simplest case, the messaging system delivers a message directly from the sender’s computer to the receiver’s computer. However, certain 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. For example, the message may have to be validated or transformed because the receiver expects a message format different from the sender’s. The *Pipes and Filters* architecture describes how multiple processing steps can be chained together using channels.

**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. The route a message must follow may be so complex that the original sender does not know what channel will get the message to the final receiver. Instead, the original sender sends the message to a *Message Router*, an application component that takes the place of a filter in the *Pipes and Filters*  architecture. The router then determines how to navigate the channel topology and directs the message to the final receiver, or at least to the next router.

**Transformation**—Various applications may not agree on the format for the same conceptual data; the sender formats the message one way, but the receiver expects it to be formatted another way. To reconcile this, the message must go through an intermediate filter, a *Message Translator* which converts the message from one format to another.

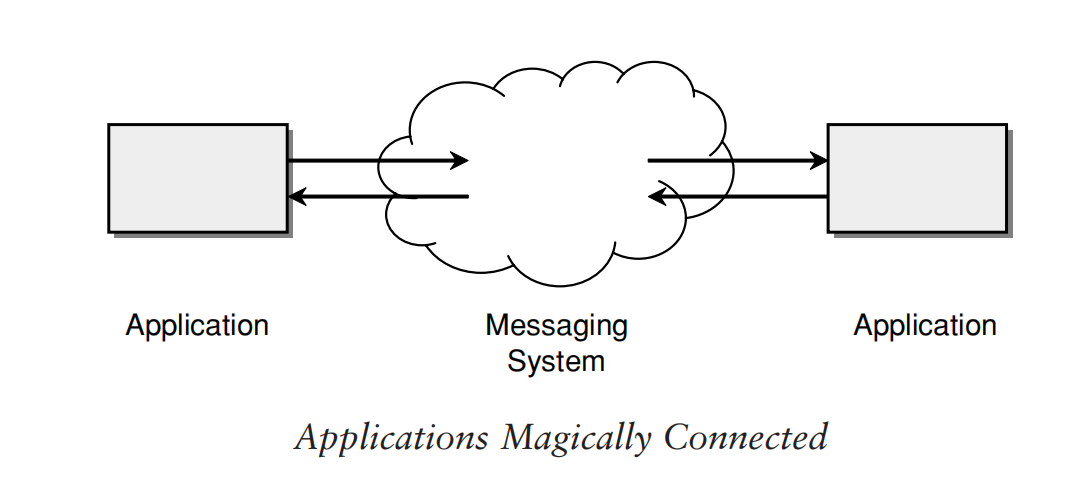
**Endpoints**—Most applications do not have any built-in capability to interface with a messaging system. Rather, they must contain a layer of code that knows both how the application works and how the messaging system works, bridging the two so that they work together. This bridge code is a set of coordinated *Message Endpoint*s that enable the application to send and receive messages.

**Message Channel**

An enterprise has two separate applications that need to communicate by using

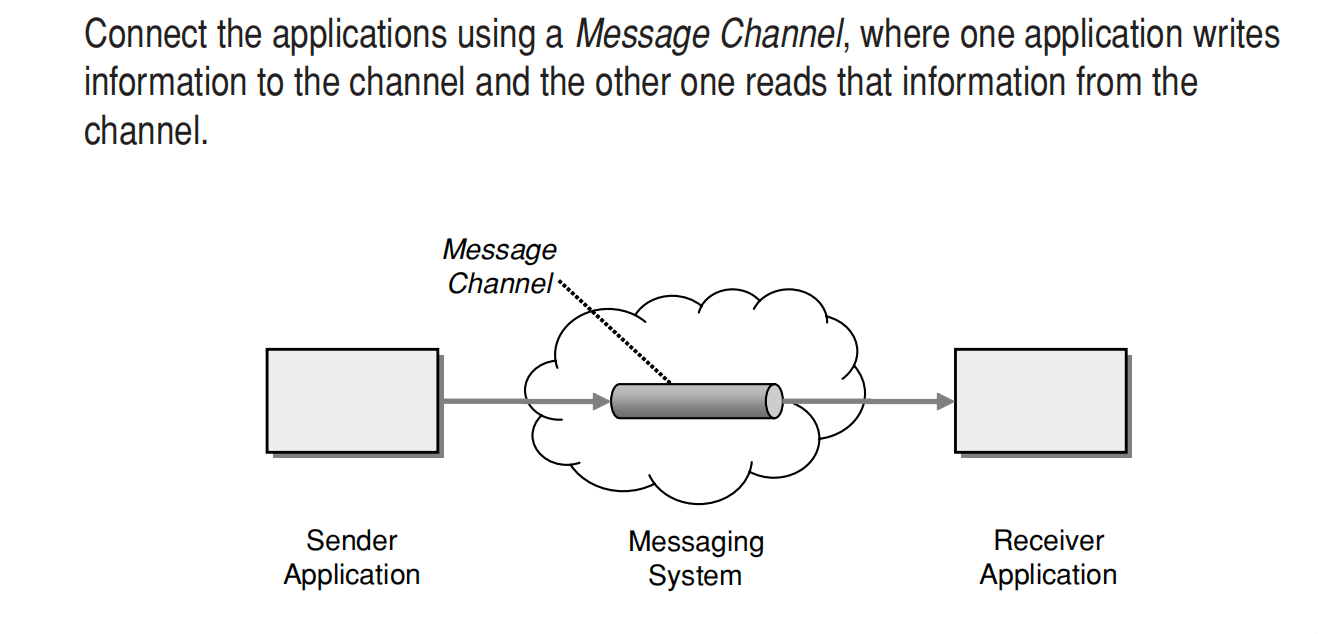
*Messaging* Once a group of applications has a messaging system available, it’s tempting to think that any application can communicate with any other application anytime you want it to. Yet, the messaging system does not magically connect all of

the applications



Likewise, it’s not as though an application just randomly throws out information into the messaging system while other applications just randomly grab whatever information they run across. (Even if this worked, it wouldn’t be very efficient.) Rather, the application sending out the information knows what sort of information it is, and the applications that would like to receive information aren’t looking for just any information but for particular types of information they can use. So 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 enables applications to communicate by transmitting information in predetermined,

predictable ways.



information into the messaging system but 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*.

The application sending information doesn’t necessarily know what particular application will end up retrieving it, but it can be assured that the application that retrieves the information is interested in the information. This is because the messaging system has different *Message Channel*s for different types of information the applications want to communicate. When an application sends information, it doesn’t randomly add the information to any channel available; it adds it to a channel whose specific purpose is to communicate that sort of information. Likewise, an application that wants to receive particular information doesn’t pull info off some random channel; it selects what channel to get information from based on what type of information it wants.

Channels are logical addresses in the messaging system. How they’re actually implemented depends on the messaging system product and its implementation.

A messaging system doesn’t automatically come pre configured with all of the message channels the applications need to communicate. Rather, the developers designing the applications and the communication between them have to decide what channels they need for the communication. Then the system administrator who installs the messaging system software must also configure it to set up the channels that the applications expect. Although some messaging system implementations support creating new channels while the applications are running, this isn’t very useful because other applications besides the one that creates the channel must know about the new channel so they can start using it too. Thus, the number and purpose of channels available tend to be fixed at deployment

time.

There are two different kinds of message channels: *Point-to-Point Channel*s and *Publish-Subscribe Channel*s. Mixing different data types on the same channel can cause a lot of confusion; to avoid this, use separate *Datatype Channels* . *Selective Consumer*  makes one physical channel act logically like multiple channels. Applications that use messaging often benefit from a special channel for invalid messages, an *Invalid Message Channel*. Applications that wish to use *Messaging* but do not have access to a messaging client can still connect to the messaging system using *Channel Adapter*s A well-designed set of channels forms a *Message Bus*  that acts like a messaging API for a whole group of applications.

**Example:** *Stock Trading*

When a stock trading application makes a trade, it puts the request on a *Message Channel* for trade requests. Another application that processes trade requests will look for those it can process on that same message channel. If the requesting application needs to request a stock quote, it will probably use a different *Message Channel*, one designed for stock quotes, so that the quote requests stay separate from the trade requests.

**Message**

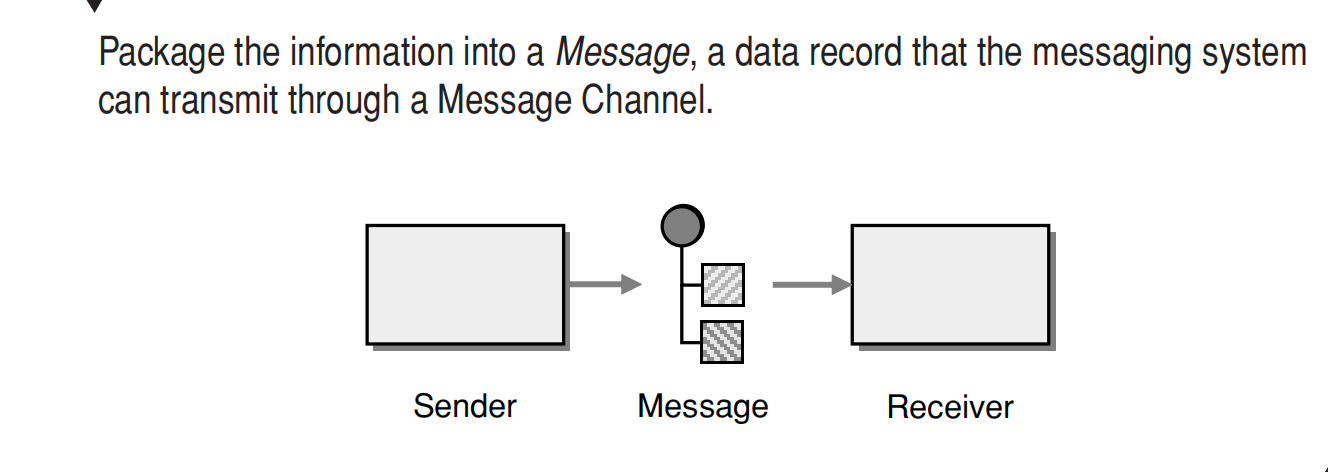
An enterprise has two separate applications that are communicating via *Messaging* using a *Message Channel*  that connects them.

***How can two applications connected by a Message Channel exchange a piece of***

***information?***

A *Message Channel* (can often be thought of as a pipe, a conduit from one application to another. It might stand to reason then that data could be poured into one end, like water, and it would come flowing out of the other end. But most application data isn’t one continuous stream; it consists of units, such as records, objects, database rows, and the like. So a channel must transmit units of data.

***What does it mean to “transmit” data?*** In a function call, the caller can pass a parameter by reference by passing a pointer to the data’s address in memory; this works because both the caller and the function share the same memory heap.Similarly, two threads in the same process can pass a record or object by passing a pointer, since they both share the same memory space.



Thus, 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.

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, which is generally ignored by the messaging system and simply transmitted as is.

This concept is not unique to messaging. Both postal service mail and e-mail send data as discrete mail messages. An Ethernet network transmits data as packets, as does the IP part of TCP/IP such as the Internet. Streaming media on the Internet is actually a series of packets.

To the messaging system, all messages are the same: some body of data to be transmitted as described by the header.

**Pipes and Filters**

In many enterprise integration scenarios, a single event triggers a sequence of processing steps, each performing a specific function.

For example, let’s assume a new order arrives in our enterprise in the form of a message. One requirement may be that the message is encrypted to prevent eavesdroppers from spying on a customer’s order. A second requirement is that the messages contain authentication information in the form of a digital certificate to ensure that orders are placed only by trusted customers. In addition, duplicate messages could be sent from external parties (remember all the warnings on the popular shopping sites to click the Order Now button only once?). To avoid duplicate shipments and unhappy customers, we need to eliminate duplicate messages before subsequent order processing steps are initiated. To meet these requirements, we need to transform a series of possibly duplicated, encrypted messages containing extra authentication data into a series of unique, simple plain-text order messages without the extraneous data fields.

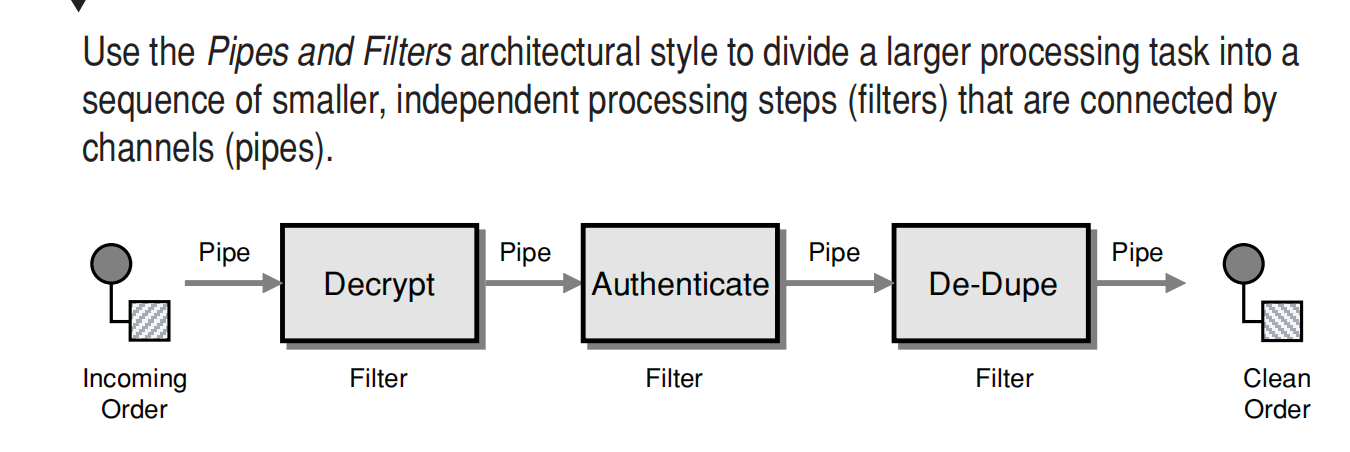
***How can we perform complex processing on a message while maintaining independence and flexibility?***

One possible solution would be to write a comprehensive “incoming message massaging module” that performs all the necessary functions. However, such an approach would be in flexible and difficult to test. What if we need to add a step or remove one? For example, what if orders can be placed by large customers who are on a private network and do not require encryption?

Implementing all functions inside a single component also reduces opportunities for reuse. Creating smaller, well defined components allows us to reuse them in other processes. For example, order status messages may be encrypted but do not need to be duped because duplicate status requests are generally not harmful. Separating the decryption function into a separate module allows us to reuse this function for other messages. Integration solutions typically connect a collection of heterogeneous systems.

As a result, different processing steps may need to execute on different physical machines, such as when individual processing steps can only execute on specific systems. For example, it is possible that the private key required to decrypt incoming messages is only available on a designated machine and cannot be accessed from any other machine for security reasons. This means that the decryption component has to execute on this designated machine, whereas the other steps may execute on other machines. Likewise, different processing steps may be implemented using different programming languages or technologies that prevent them from running inside the same process or even on the same computer.

If we use asynchronous messaging, we should take advantage of the asynchronous aspects of sending messages from one component to another. For example, 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, one inside each component.



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. The pipe connects one filter to the next, sending output messages from one filter to the next. Because all components use the same external interface, they can be *composed* into different solutions by connecting the components to different pipes. We can add new filters, omit existing ones, or rearrange them into a new sequence—all without having to change the filters themselves. The connection between filter and pipe is sometimes called a *port*. In the basic form, each filter component has one input port and one output port. When applied to our example problem, the *Pipes and Filters* architecture

results in three filters connected by two pipes. We need one additional pipe to send messages to the decryption component and one to send the clear-text order messages from the duper to the order management system. This makes a total of four pipes.

*Pipes and Filters* describes a fundamental architectural style for messaging systems: Individual processing steps (filters) are chained together through the messaging channels (pipes). Many patterns in this and the following sections, such as routing and transformation patterns, are based on this *Pipes and Filters* architectural style. This lets you easily combine individual patterns into larger solutions.

The *Pipes and Filters* style uses abstract pipes to decouple components from each other. The pipe allows one component to send a message into the pipe so that it can be consumed later by another process that is unknown to the component.

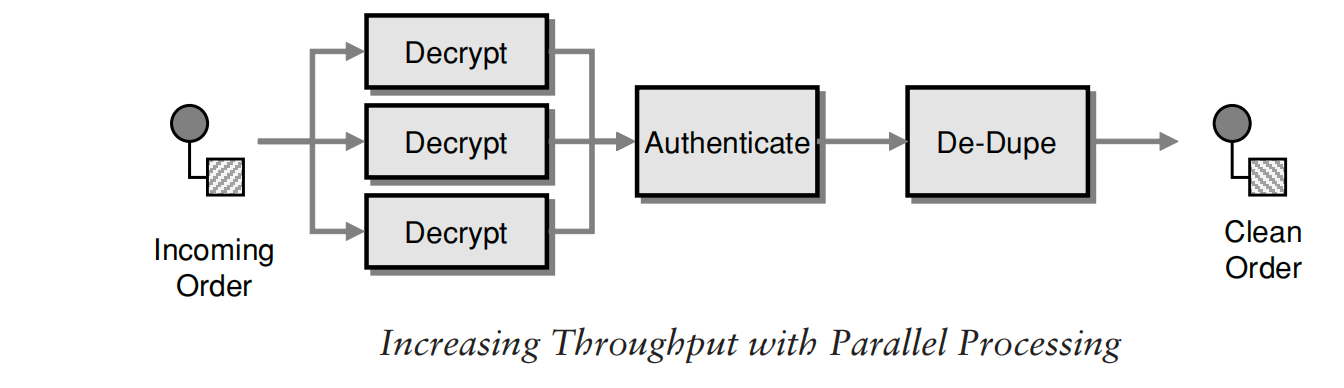
**Pipeline Processing**

Connecting components with asynchronous *Message Channel*s allows each unit in the chain to operate in its own thread or its own process. When a unit has completed processing one message, it can send the message to the output channel and immediately start processing another message. It does not have to wait for the subsequent components to read and process the message. This allows multiple messages to be processed concurrently as they pass through the individual stages. For example, after the first message has been decrypted, it can

be passed on to the authentication component. At the same time, the next message can already be decrypted. We call such a configuration a *processing pipeline* because messages flow through the filters like liquid flows through a pipe. When compared to strictly sequential processing, a processing pipeline can significantly increase system throughput.

**Parallel Processing**

However, the overall system throughput is limited by the slowest process in the chain. We can deploy multiple parallel instances of that process to improve throughput. In this scenario, a *Point-to-Point Channel*  with *Competing Consumers* is needed to guarantee that each message on the channel is consumed by exactly one of *N* available processors. This allows us to speed up the most time-intensive process and improve overall throughput. We need to be aware, though, that this figuration can cause messages to be processed out of order. If the sequence of messages is critical, we can run only one instance of each component or we must use a *Re sequencer*



**Message Router**

***How can you decouple individual processing steps so that messages can be passed***

***to different filters depending on a set of conditions?***

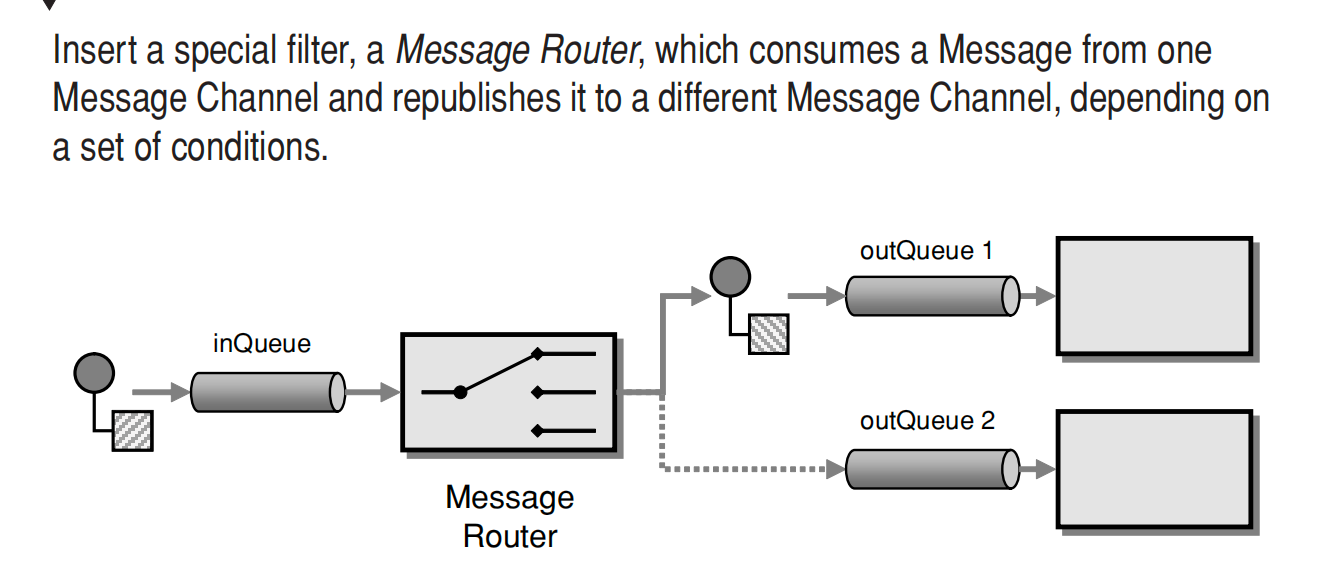
For example, a compiler will always execute the lexical analysis first, the syntactic analysis second, and the semantic analysis last. Message-based integration solutions, on the other hand, deal with individual messages that are not necessarily associated with a single, larger data set. As a result, individual messages are more likely to

require a different series of processing steps.

However, this would require the message originators to be aware of the selection criteria for different processing steps in order to publish the message to the correct channel. It could also lead to an explosion

of the number of *Message Channel*s. Furthermore, the decision on which steps the message undergoes may not just depend on the origin of the message.

For example, we could imagine a situation where the destination of a message varies depending on the number of messages that have passed through the channel so far. No single originator would know this number and would therefore be unable to send the message to the correct channel.



The *Message Router* differs from the basic notion of *Pipes and Filters*  in

that it connects to multiple output channels (i.e., it has more than one output port). However, thanks to the *Pipes and Filters*  architecture, the components surrounding the *Message Router* are completely unaware of the existence of a *Message Router*. They simply consume messages off one channel and publish them to another. A defining property of the *Message Router* is that it does not modify the message contents; it concerns itself only 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. If new message types are defined, new processing components are added, or routing rules change, we need to change only the *Message Router* logic, while all other components remain unaffected. Also, since all messages pass through a single *Message Router*, incoming messages are guaranteed to be processed one by one in the correct order.

While the intent of a *Message Router* is to decouple filters from each other,

using a *Message Router* can actually cause the opposite effect. The *Message Router* component must have knowledge of all possible destination channels in order to send the message to the correct channel. If the list of possible destinations changes frequently, the *Message Router* can turn into a maintenance bottleneck.

**Message Translator**

The previous patterns show how to construct messages and how to route them to the correct destination. In many cases, enterprise integration solutions route

messages between existing applications such as legacy systems, packaged applications, homegrown custom applications, or applications operated by external partners. Each of these applications is usually built around a proprietary data model. Each application may have a slightly different notion of the Customer entity, the attributes that define a Customer, and other entities to which a Customer is related. For example, the accounting system may be more interested in

the customer’s taxpayer ID numbers, whereas the customer-relationship management (CRM) system stores phone numbers and addresses. The application’s

underlying data model usually drives the design of the physical database schema,

an interface file format, or an application programming interface (API)—those

entities with which an integration solution must interface. As a result, each

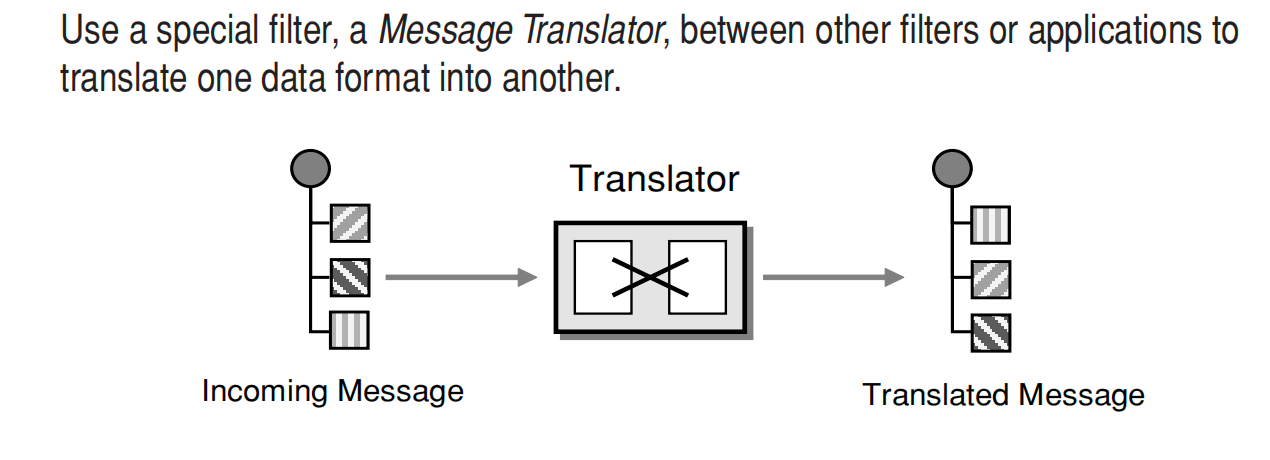
application typically expects to receive messages that mimic the application’s

internal data format.

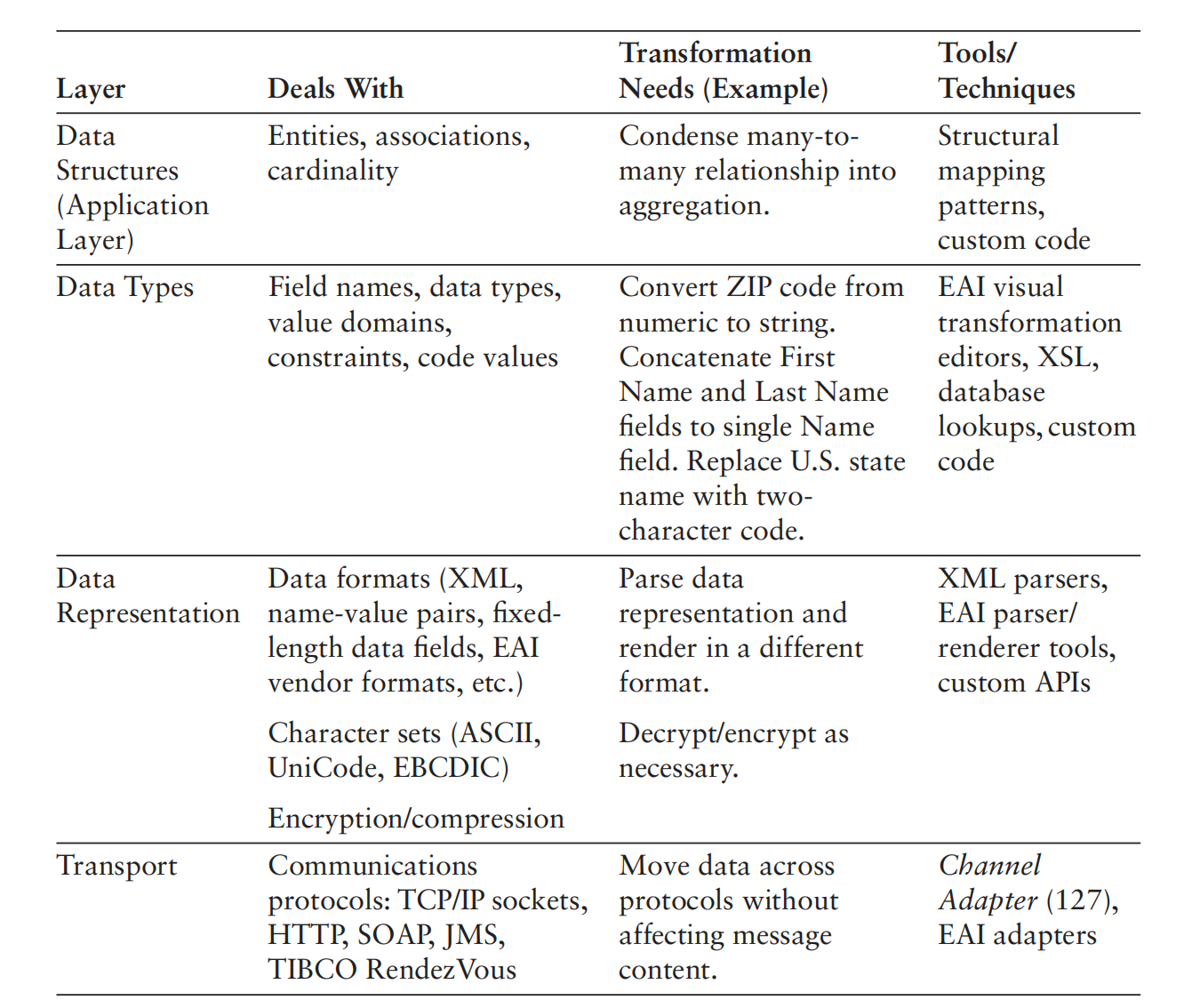
***How can systems using different data formats communicate with each other using***

***messaging?***

We could avoid having to transform messages if we could modify all applications to use a common data format. This turns out to be difficult for a number of reasons First, changing an application’s data format is risky, difficult, and requires a lot of changes to inherent business



An adapter converts the interface of a component into another interface so it can be used in a different context.



**Message Endpoint**

Applications are communicating by sending *Messages* (66) to each other via *Message Channel*s (60).

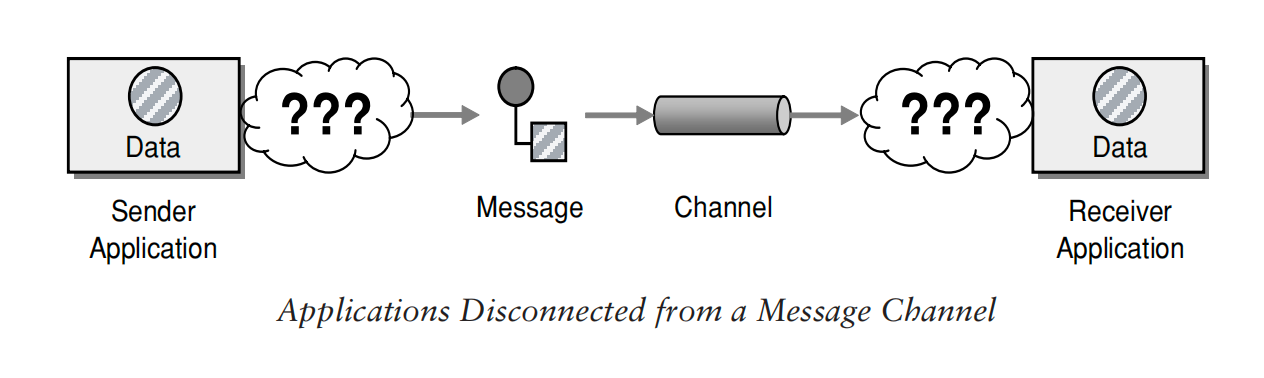
***How does an application connect to a messaging channel to send and receive***

***Messages?***

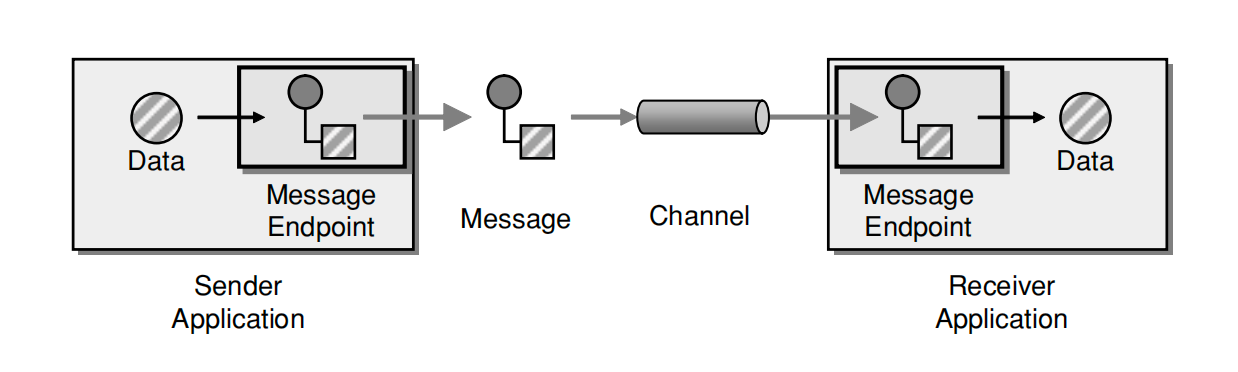
The application and the messaging system are two separate sets of software.

The application provides functionally for some type of user, whereas the messaging system manages messaging channels for transmitting messages for communication. Even if the messaging system is incorporated as a fundamental part of the application, it is still a separate, specialized provider of functionality,

much like a database management system or a Web server. Because the application and the messaging system are separate, they must have a way to connect



A messaging system is a type of server, capable of taking requests and responding to them. Like a database accepting and retrieving data, a messaging server accepts and delivers messages. A messaging system is a messaging server. A server needs clients, and an application that uses messaging is a client of the messaging server. But applications do not necessarily know how to be messaging clients any more than they know how to be database clients. The messaging server, like a database server, has a client API that the application can use to interact with the server. The API is not application specific but is domain- specific, where the domain is messaging. The application must contain a set of code that connects and unites the messaging domain with the application to allow the application to perform messaging.



*Message Endpoint* code is custom to both the application and the messaging system’s client API. The rest of the application knows little about message formats, messaging channels, or any of the other details of communicating with other applications via messaging. It just knows that it has a request or piece of data to send to another application, or is expecting those from another application. It is the messaging endpoint code that takes that 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.

The *Message Endpoint* encapsulates the messaging system from the rest of the application and customizes a general messaging API for a specific application and task. If an application using a particular messaging API were to switch to another, developers would have to rewrite the message endpoint code, but the rest of the application should remain the same. If a new version of a messaging system changes the messaging API, this should only affect the message endpoint code. If the application decides to communicate with others via some

means other than messaging, developers should ideally be able to rewrite the

message endpoint code but leave the rest of the application unchanged.

A *Message Endpoint* can be used to send messages or receive them, but one

instance does not do both. An endpoint is channel specific, so a single application would use multiple endpoints to interface with multiple channels.