Asynchronous, Message-Based Communication in Microservices

When using messaging, services communicate by asynchronously exchanging messages.

A messaging-based application typically uses a *message broker*, which acts as an

intermediary between the services, although another option is to use a brokerless

architecture, where the services communicate directly with each other. A service client

makes a request to a service by sending it a message. If the service instance is expected

to reply, it will do so by sending a separate message back to the client. Because the

communication is asynchronous, the client doesn’t block waiting for a reply. Instead,

the client is written assuming that the reply won’t be received immediately.

I start this section with an overview of messaging. I show how to describe a messaging

architecture independently of messaging technology. Next I compare and contrast

brokerless and broker-based architectures and describe the criteria for selecting a

message broker. I then discuss several important topics, including scaling consumers

while preserving message ordering, detecting and discarding duplicate messages,

and sending and receiving messages as part of a database transaction. Let’s begin by

looking at how messaging works.

application has reached

# Overview of Messaging

A useful model of messaging is defined in the book Enterprise Integration Patterns (Addison-Wesley Professional, 2003) by Gregor Hohpe, and Bobby Woolf.

In this model, messages are exchanged over message channels. A sender (an application or service) writes a message to a channel, and a receiver (an application or service) reads messages from a channel. Let’s look at messages and then look at channels.

## About Messages

A message consists of **a header and a message body** ([www.enterpriseintegrationpatterns.com/Message.html](http://www.enterpriseintegrationpatterns.com/Message.html)).

The *header* is a collection of name-value pairs, metadata that describes the data being sent. In addition to name-value pairs provided by the message’s sender, the message header contains name-value pairs, such as a unique *message* *id* generated by either the sender or the messaging infrastructure, and **an optional *return address*, which specifies the message channel that a reply should be written to.**

The message *body* is the data being sent, **in either text or binary format**. There are several different kinds of messages:

 ***Document***—A generic message that contains only data. **The receiver decides how to interpret it**. The reply to a command is an example of a document message.

 ***Command***—A message that’s the **equivalent of an RPC request**. It specifies the operation to invoke and its parameters.

 ***Event***—**A message indicating that something notable has occurred in the sender**. **An event is often a domain event, which represents a state change of a domain object such as an Order, or a Customer.**

The approach to the microservice architecture described in this book **uses commands and events extensively.**

Let’s now look at channels, the mechanism by which services communicate.

## About Message Channels

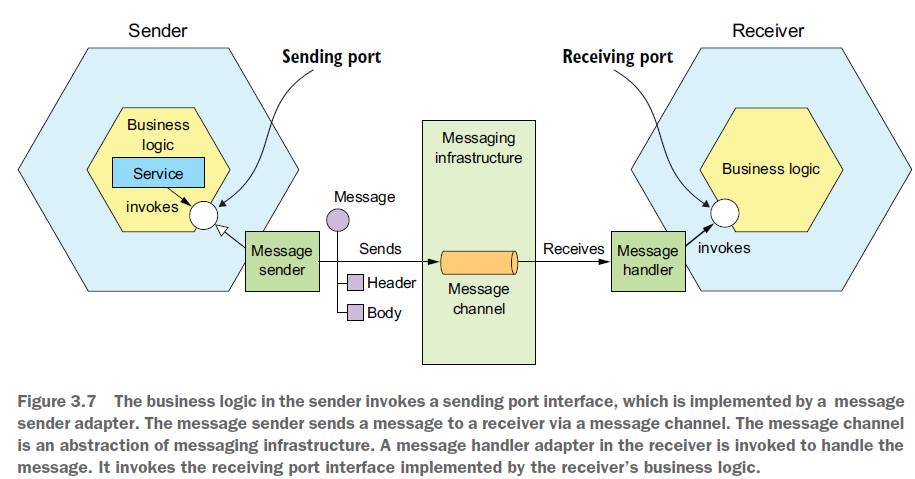
As figure 3.7 shows, messages are exchanged over channels ([www.enterpriseintegrationpatterns.com/MessageChannel.html](http://www.enterpriseintegrationpatterns.com/MessageChannel.html)).

The business logic in the sender invokes a *sending port* interface, which encapsulates the underlying communication mechanism. The *sending port* is implemented by a *message sender* adapter class, which sends a message to a receiver via a message channel.

**A *message channel* is an abstraction of the messaging infrastructure**.

A *message handler* adapter class in the receiver is invoked to handle the message. It invokes a *receiving port* interface implemented by the consumer’s business logic.

Any number of senders can send messages to a channel. Similarly, any number of receivers can receive messages from a channel.



There are two kinds of channels: point-to-point (www.enterpriseintegrationpatterns.com/PointToPointChannel.html) and publish-subscribe ([www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html](http://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html)):

*I add: I think this is what he meant by the channel being an abstraction of the messaging infrastructure, it can be a publish-subscribe or point-to-point channel.*

 **A *point-to-point***channel **delivers a message to exactly one of the consumers** that is reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier. For example, a command message is **often** sent over a point-to-point channel.

 **A *publish-subscribe***channel **delivers each message to all of the attached consumers**. Services use **publish-subscribe channels for the one-to-many interaction styles** described earlier. For example, an event message is **usually** sent over a publish-subscribe channel.

# Implementing the Interaction Styles Using Messaging

One of the valuable features of messaging is **that it’s flexible enough to support all the interaction styles described in section 3.1.1**.

Some interaction styles are directly implemented by messaging. Others must be implemented on top of messaging.

## Implementing Request/Response and Asynchronous Request/Response

When a client and service interact using either request/response or asynchronous request/response, the client sends a request and the service sends back a reply.

The difference between the two interaction styles is that with request/response the client expects the service to respond immediately, whereas with asynchronous request/response there is no such expectation.

Messaging is inherently asynchronous, so only provides asynchronous request/response. **But a client could block until a reply is received.**

….page88

………………don’t forget the intro section