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 (I add: /Query)***—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

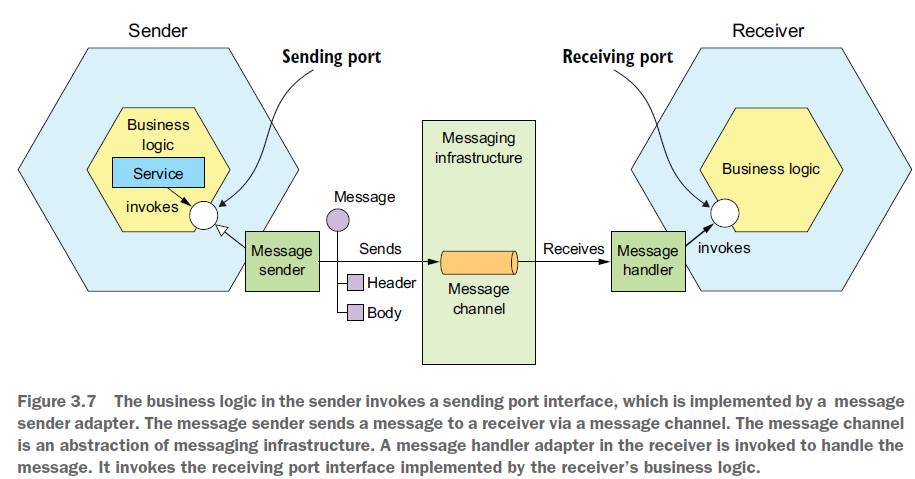
As figure 3.7 shows, messages are exchanged over channels ([www.enterpriseintegrationpatterns.com/MessageChannel.html](http://www.enterpriseintegrationpatterns.com/MessageChannel.html)).

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.**

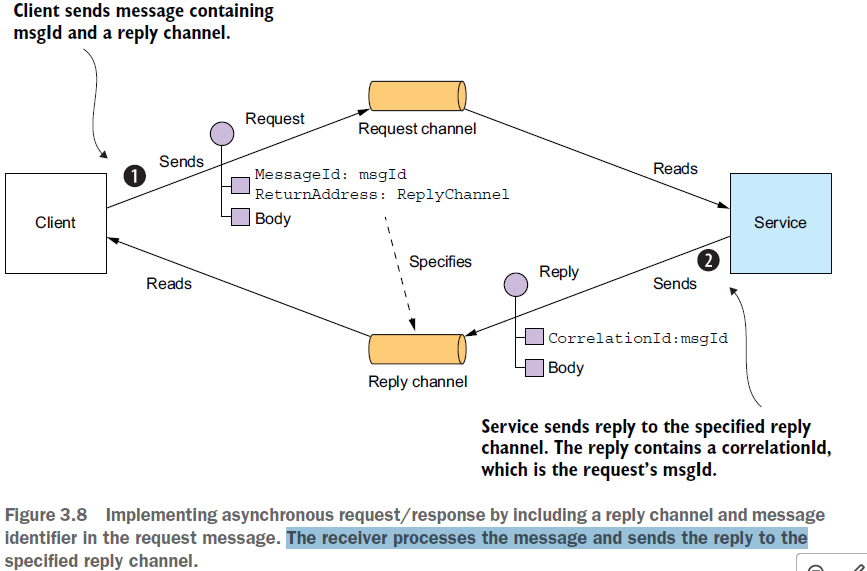
The difference between the two interaction styles is that with request/response the client expects the service to respond immediately, whereas with asynchronous request/response there is no such expectation.

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

The client and service implement the **asynchronous request/response** style interaction **by exchanging a pair of messages.**

As figure 3.8 shows,

* the client sends a command message, which specifies **the operation to perform**, **and parameters**, **to a point-to-point messaging channel owned by a service**.
* **The service processes the requests** and **sends a reply message**, which **contains the outcome**, **to a point-to-point channel owned by the client.**



* **The client must tell the service where to send a reply message,** it does that by sending a message that contains a **replyChannel header** (return address)
* **The client must also match incoming reply messages to the requests it has sent**, this problem is addressed by the client sending a **messageIdentifier header** along with the message, and the service sends a reply message to the reply channel along with a **correlationId header** that matches the original request’s messageId.
* and must match reply messages to requests. Fortunately, solving these two problems isn’t that difficult. The client sends a command message that has a reply channel header. The server writes the reply message,

In theory, a messaging client could block until it receives a reply, but in practice the client will process replies asynchronously.

**What’s more, replies are typically processed by any one of the client’s instances.**

## Implementing One-Way Notification

Implementing one-way notifications is straightforward using asynchronous messaging.

The client sends a message, **typically a command message**, to a point-to-point channel owned by the service.

The service subscribes to the channel and processes the message. It doesn’t send back a reply.

**(Why does he use the term subscribe?)**

## Implementing Publish-Subscribe

Messaging has built-in support for the publish/subscribe style of interaction.

**A client publishes a message to a publish-subscribe channel that is read by multiple consumers.**

* As described in chapters 4 and 5, **services use publish/subscribe to publish domain events, which represent changes to domain objects.**
* **The service that publishes the domain events owns a publish-subscribe channel**, **whose name is derived from the domain class** (*He must mean the channel’s name*)**.**
* **For example, the Order Service publishes Order events to an Order channel, and the Delivery Service publishes Delivery events to a Delivery channel.**

**A service that’s interested in a particular domain object’s events only has to subscribe to the appropriate channel.**

## Implementing Publish/ Async Response

The publish/async responses interaction style is a higher-level style of interaction that’s implemented by combining elements of publish/subscribe and request/response.

* A client publishes a message that specifies a *reply channel* header **to a publish-subscribe channel.**
* A consumer writes a reply message containing a *correlation id* to the reply channel.
* The client gathers the responses by using the *correlation id* to match the reply messages with the request.

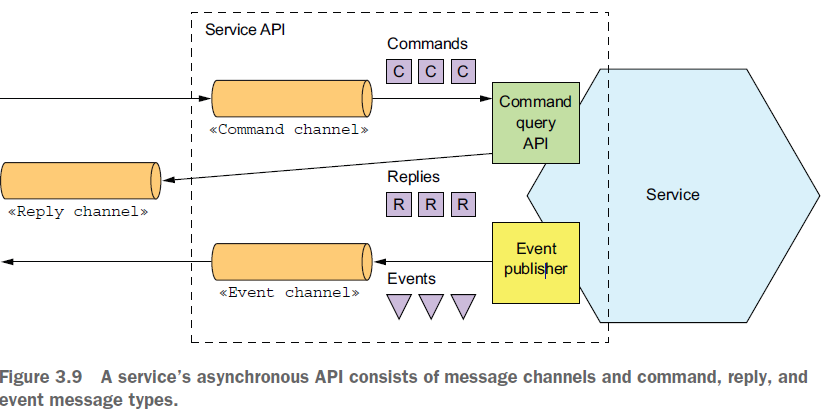
Each service in your application that has an asynchronous API will use one or more of these implementation techniques. A service that has an asynchronous API for invoking operations will have a message channel for requests. Similarly, a service that publishes events will publish them to an event message channel.

As described in section 3.1.2, it’s important to write an API specification for a service. Let’s look at how to do that for an asynchronous API.

# Creating an API Specification for a Messaging-Based Service API

The specification for a service’s asynchronous API must, as figure 3.9 shows, specify:

* The API itself can be operations invoked by clients or, events published by the service
* the names of the message channels
* the message types that are exchanged over each channel and their formats



You must also describe the format of the messages using a standard such as **JSON**, **XML**, or **Protobuf**. But **unlike with REST and Open API, there isn’t a widely adopted standard for documenting the channels and the message types.** Instead, you need to write an informal document.

A service’s asynchronous API consists of

* **operations, invoked by clients**
* **and events, published by the services.**

They’re documented in different ways. Let’s take a look at each one, starting with operations.

## Documenting Asynchronous Operations

**A service’s operations** can be invoked using one of two different interaction styles:

 ***Request/async response-style API***—This consists of:

* the service’s command message channel
* the types and formats of the command message types that the service accepts
* and the types and formats of the reply messages sent by the service.

 ***One-way notification-style API***—This consists of:

* the service’s command message channel
* and the types and format of the command message types that the service accepts.

**A service may use the same request channel for both asynchronous request/response and one-way notification.**

## Documenting Published Events

A service can also publish events using a publish/subscribe interaction style. The specification of this style of API consists of

* the event channel
* and the types and formats of the event messages that are published by the service to the channel.

**The messages and channels model of messaging is a great abstraction and a good way to design a service’s asynchronous API.** But in order to implement a service you need to choose a messaging technology and determine how to implement your design using its capabilities. Let’s take a look at what’s involved.

# Using a Message Broker

In a messaging-based application, we typically use a message broker, an infrastructure through which services communicate. You can also have a brokerless architecture in which services communicate directly. Both have their trade-offs but broker-based is usually a better approach and what we will be focusing on this approach after a quick look at the brokerless option as there might be scenarios in which it can be helpful.

## Brokerless Messaging

In a brokerless architecture, services can exchange messages directly. ZeroMQ (http://zeromq.org) is a popular brokerless messaging technology. It’s both a specification and a set of libraries for different languages. It supports a variety of transports, including **TCP, UNIX-style domain sockets, and multicast.**

The brokerless architecture has some benefits:

 Allows lighter network traffic and better latency, because messages go directly from the sender to the receiver, instead of having to go from the sender to the message broker and from there to the receiver

 **Eliminates the possibility of the message broker being a performance bottleneck or a single point of failure**

 Features less operational complexity, because there is no message broker to set up and maintain

As appealing as these benefits may seem, brokerless messaging **has significant drawbacks:**

 Services need to know about each other’s locations and must therefore use one of the discovery mechanisms described earlier in section 3.2.4.

 **It offers reduced availability, because both the sender and receiver of a message must be available while the message is being exchanged.**

 **Implementing mechanisms, such as guaranteed delivery, is more challenging.**

In fact, some of these drawbacks, such as reduced availability and the need for service discovery, are **the same as when using synchronous, response/response.**

Because of these limitations, **most enterprise applications use a message broker-based architecture**. Let’s look at how that works.

## “Overview” of Broker-based Messaging

A message broker is an intermediary through which all messages flow. A sender writes the message to the message broker, and the message broker delivers it to the receiver.

* **An important benefit of using a message broker is that the sender doesn’t need to know the network location of the consumer**.
* Another benefit is that a message broker buffers messages until the consumer is able to process them.

There are many message brokers to choose from. Examples of popular open-source message brokers include the following:

 ActiveMQ (http://activemq.apache.org)

 RabbitMQ (https://www.rabbitmq.com)

 Apache Kafka (<http://kafka.apache.org>)

There are also cloud-based messaging services, such as

 AWS Kinesis (https://aws.amazon.com/kinesis/)

 AWS SQS (<https://aws.amazon.com/sqs/>).

When selecting a message broker, you have various factors to consider, including the following:

 *Supported programming languages*—You probably should pick one that supports a variety of programming languages.

 *Supported messaging standards*—Does the message broker support any standards, such as AMQP and STOMP, or is it proprietary?

 ***Messaging ordering***—Does the message broker preserve ordering of messages?

 ***Delivery guarantees***—What kind of delivery guarantees does the broker make?

 ***Persistence***—Are messages persisted to disk and able to survive broker crashes?

 ***Durability***—If a consumer reconnects to the message broker, will it receive the messages that were sent while it was disconnected?

 ***Scalability***—How scalable is the message broker?

 ***Latency***—What is the end-to-end latency?

 ***Competing consumers***—Does the message broker support competing consumers?

Each broker makes different trade-offs:

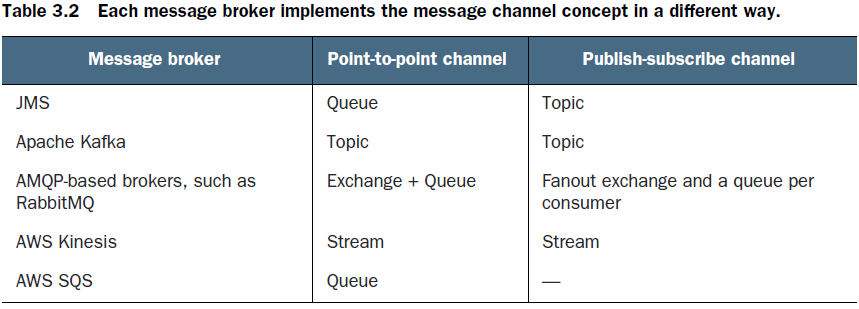
* For example, a very low-latency broker might not preserve ordering, make no guarantees to deliver messages, and only store messages in memory.
* A messaging broker that guarantees delivery and reliably stores messages on disk will probably have higher latency.

Which kind of message broker is the best fit depends on your application’s requirements. **It’s even possible that different parts of your application will have different messaging requirements**. It’s likely, though, that messaging ordering and scalability are essential.

### Message Channels, and Interaction Styles in Different Message Brokers

Each message broker implements the message channel concept in a different way:

* **JMS message brokers** such **as ActiveMQ** have **queues and topics**.
* **AMQP-based** message brokers such as **RabbitMQ** have **exchanges and queues**.
* **Apache Kafka** has **topics**
* AWS Kinesis has streams
* and AWS SQS has queues. What’s more
* some message brokers offer more flexible messaging than the message and channels abstraction described in this chapter.



### Benefits of Broker-Based Messaging

### Drawbacks of Broker-Based Messaging

………………don’t forget the intro section, and why is messaging inherently async, and also go over messaging styles again.