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

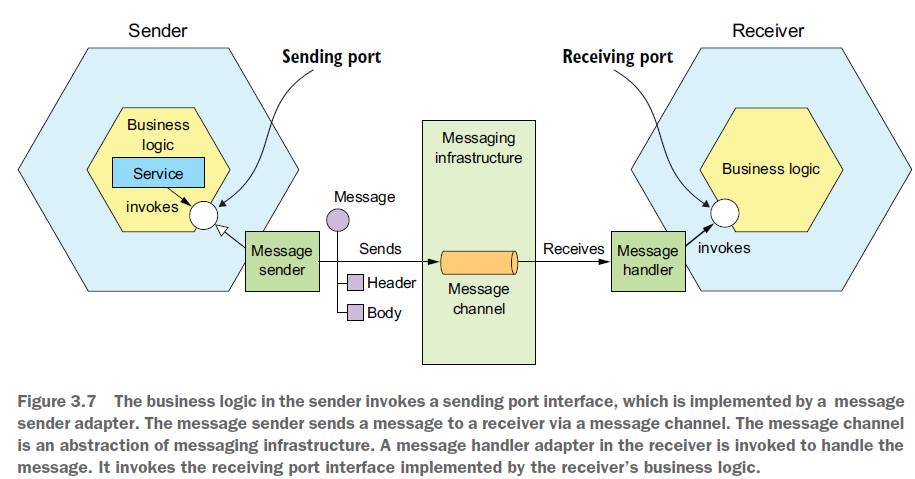
As figure 3.7 shows, messages are exchanged over channels ([www.enterpriseintegrationpatterns.com/MessageChannel.html](http://www.enterpriseintegrationpatterns.com/MessageChannel.html)).

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.**

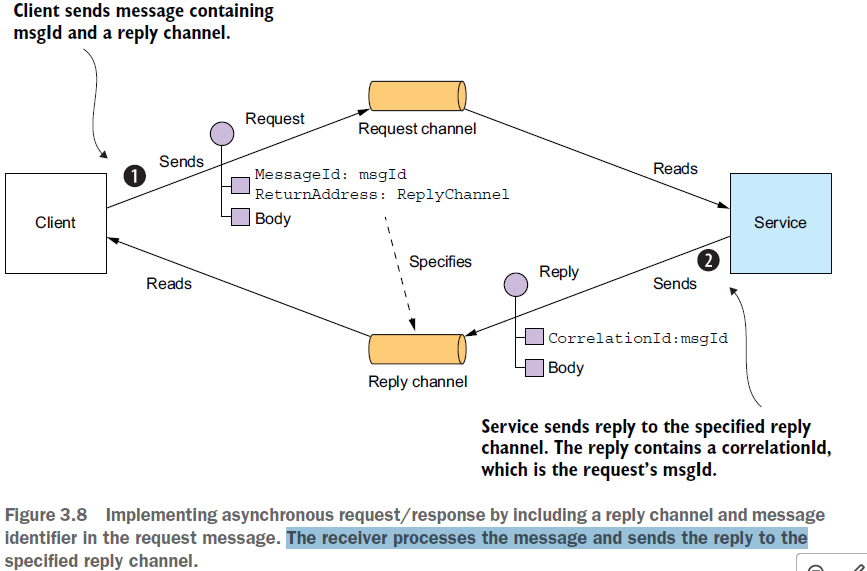
The difference between the two interaction styles is that with request/response the client expects the service to respond immediately, whereas with asynchronous request/response there is no such expectation.

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.

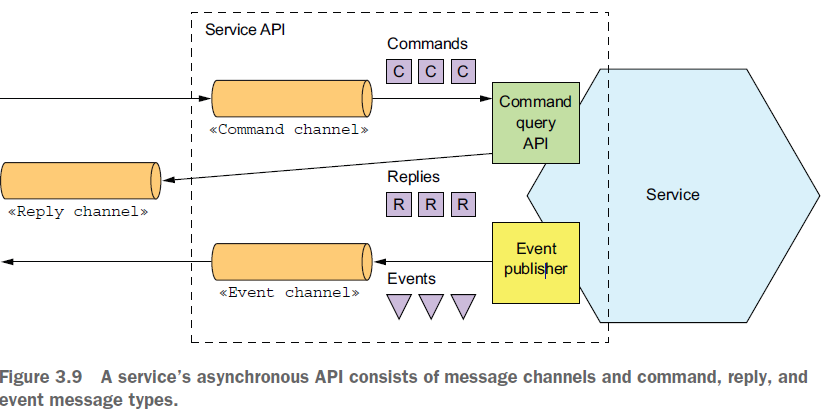
Each service in your application that has an asynchronous API will use one or more of these implementation techniques. A service that has an asynchronous API for invoking operations will have a message channel for requests. Similarly, a service that publishes events will publish them to an event message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

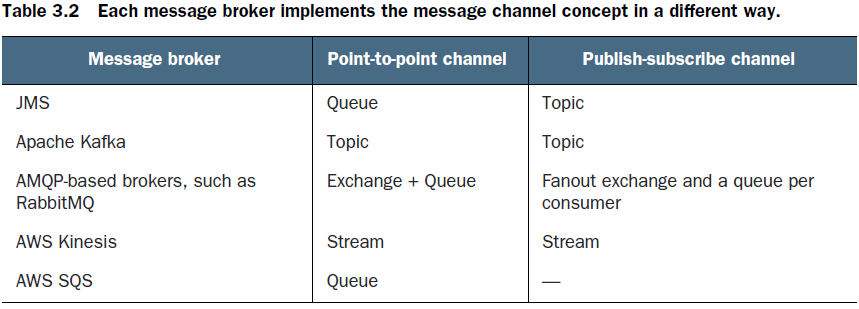
* For example, a very low-latency broker might not preserve ordering, make no guarantees to deliver messages, and only store messages in memory.
* A messaging broker that guarantees delivery and reliably stores messages on disk will probably have higher latency.

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

**Loose coupling**—A client makes a request by simply sending a message to the appropriate channel. The client is completely unaware of the service instances. It doesn’t need to use a discovery mechanism to determine the location of a service instance.

***Message buffering***—The message broker buffers messages until they can be processed. With a synchronous request/response protocol such as HTTP, both the client and service must be available for the duration of the exchange. With messaging, though, messages will queue up until they can be processed by the consumer. This means, for example, that an online store can accept orders from

customers even when the order-fulfillment system is slow or unavailable. The messages will simply queue up until they can be processed.

 ***Flexible communication***—Messaging supports all the interaction styles described earlier.

 ***Explicit inter-process communication***—RPC-based mechanism attempts to make invoking a remote service look the same as calling a local service. But due to the laws of physics and the possibility of partial failure, they’re in fact quite different. Messaging makes these differences very explicit, so developers aren’t lulled into a false sense of security.

### Drawbacks of Broker-Based Messaging

 *Potential performance bottleneck*—There is a risk that the message broker could be a performance bottleneck. Fortunately, many modern message brokers are designed to be highly scalable.

 *Potential single point of failure*—It’s essential that the message broker is highly available—otherwise, system reliability will be impacted. Fortunately, most modern brokers have been designed to be highly available.

 *Additional operational complexity*—The messaging system is yet another system component that must be installed, configured, and operated.

# Challenges when using Message-Based Communication

## Competing Receivers and Message Ordering

One challenge is how to **scale out message receivers while preserving message ordering**.

* It’s a common requirement to have multiple instances of a service in order to process messages concurrently.
* Moreover, even a single service instance will probably use threads to concurrently process multiple messages.

Using multiple threads and service instances to concurrently process messages increases the throughput of the application. But the challenge with processing messages concurrently is ensuring that each message is processed **once and in order**.

For example, imagine that there are three instances of a service reading from the same point-to-point channel and that a sender publishes Order Created, Order Updated, and Order Cancelled event messages **sequentially**.

A simplistic messaging implementation could concurrently deliver each message to a different receiver. Because of delays due to network issues or garbage collections(**????**), messages might be processed out of order, which would result in strange behavior. In theory, a service instance might process the Order Cancelled message before another service processes the Order Created message!

**A common solution**, used by modern message brokers like Apache Kafka and AWS Kinesis, is to **use *sharded* (partitioned) channels**.

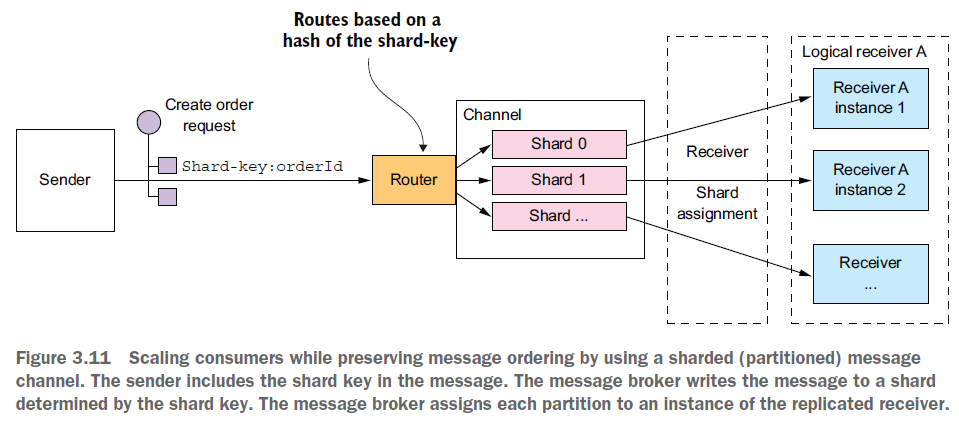
There are three parts to the solution:

**1** A sharded channel consists of two or more shards, each of which behaves like a channel.

**2** The sender specifies a shard key in the message’s header, which is typically an arbitrary string or sequence of bytes. The message broker uses a shard key to assign the message to a particular shard/partition. It might, for example, select the shard by computing the hash of the shard key modulo the number of shards.

**3** The messaging broker groups together multiple instances of a receiver and treats them as the same logical receiver. **Apache Kafka, for example, uses the term *consumer group***. The message **broker assigns each shard to a single receiver.** **It reassigns shards when receivers start up and shut down. (LEARN MORE ABOUT THIS ASSIGNMENT PROCESS, HOW DOES KAFKA KNOWS IF A CONSUMER HAS SHUT DOWN?)**

The figure below shows how consumers of Order events can scale out while preserving the order of events for each Order item using sharding. In this example, each Order event message has the **orderId as its shard key**. Each event for a particular order is published to the same shard, which is read by a single consumer instance. As a result, **these messages are guaranteed to be processed in order.**



## Handling Duplicate Messages

Another challenge you must tackle when using messaging is dealing with duplicate messages. A message broker should ideally deliver each message only once, but guaranteeing exactly-once messaging is usually too costly. Instead, **most message brokers promise to deliver a message *at least* once.**

When the system is working normally, a message broker that guarantees at-least-once delivery will deliver each message only once. But a failure of a client, network, or message broker can result in a message being delivered multiple times. Say a client crashes after processing a message and updating its database—but before acknowledging the message. The message broker will deliver the unacknowledged message again, either to that client when it restarts or to another replica of the client.

Ideally, you should use a message broker that preserves ordering when redelivering messages. Imagine that the client processes an Order Created event followed by an Order Cancelled event for the same Order, and that somehow the Order Created event wasn’t acknowledged. The message broker should redeliver both the Order Created and Order Cancelled events. If it only redelivers the Order Created, the client may undo the cancelling of the Order.

There are a couple of different ways to handle duplicate messages:

 Write idempotent message handlers.

 Track messages and discard duplicates.

Let’s look at each option.

### Writing Idempotent Message Handlers

If the application logic that processes messages is idempotent, then duplicate messages

are harmless.

Application logic is *idempotent* if calling it multiple times with the same input values has no additional effect. For instance, cancelling an already-cancelled order is an idempotent operation. So is creating an order with a client-supplied ID.

**An idempotent message handler can be safely executed multiple times, provided that the message broker preserves ordering when redelivering messages.**

Unfortunately, **application logic is often not idempotent**. **Or you may be using a message broker that doesn’t preserve ordering when redelivering messages. Duplicate or out-of-order messages can cause bugs. In this situation, you must write message handlers that track messages and discard duplicate messages.**

**I add:**

**On the Producer side you also might send messages multiple times to the broker, so you have to be careful about preserving orders both when you send messages for the first time and potentially multiple times.**

### Tracking Messages and Discarding Duplicates

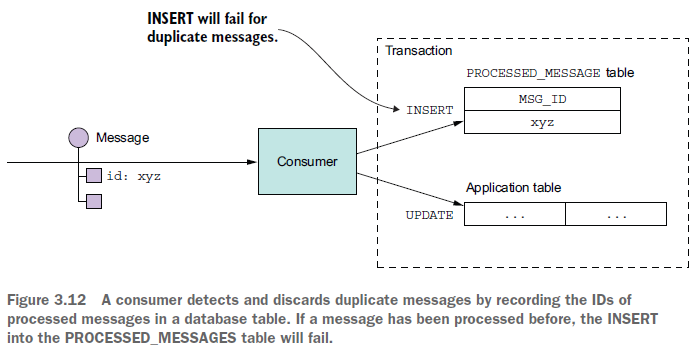
Consider, for example, a message handler that authorizes a consumer credit card. It must authorize the card exactly once for each order. This example of application **logic has a different effect each time it’s invoked.** If duplicate messages caused the message handler to execute this logic multiple times, the application would behave incorrectly.

The message handler that executes this kind of application logic must become idempotent

by detecting and discarding duplicate messages.

A simple solution is for a message consumer to track the messages that it has processed using the **message id** and discard any duplicates.

It could, for example, store the message id of each message that it consumed in a database table. Figure 3.12 shows how to do this using a dedicated table.



When a consumer handles a message, it records the message id in the database table as part of the transaction that creates and updates business entities. In this example, the consumer inserts a row containing the message id into a PROCESSED\_MESSAGES table**. If a message is a duplicate, the INSERT will fail(why not just check the processed message table before processing instead of waiting for insert to fail ??) and the consumer can discard the message.**

Another option is for a message handler to record message ids in an application table instead of a dedicated table. This approach is particularly useful when using a NoSQL database that has a limited transaction model, so it doesn’t support updating two tables as part of a database transaction. Chapter 7 of the book shows an example of this approach.

## Transactional Messaging

**A service often needs to publish messages as part of a transaction that updates the database.**

Both the database update and the sending of the message must happen within a transaction. Otherwise, a service might update the database and then crash, for example, before sending the message. If the service doesn’t perform these two operations atomically, a failure could leave the system in an inconsistent state.

The traditional solution is to use a distributed transaction that spans the database and the message broker. But as you’ll learn in chapter 4, distributed transactions aren’t a good choice for modern applications. Moreover, many modern brokers such as Apache Kafka don’t support distributed transactions.(I think Kafka does in the current version)

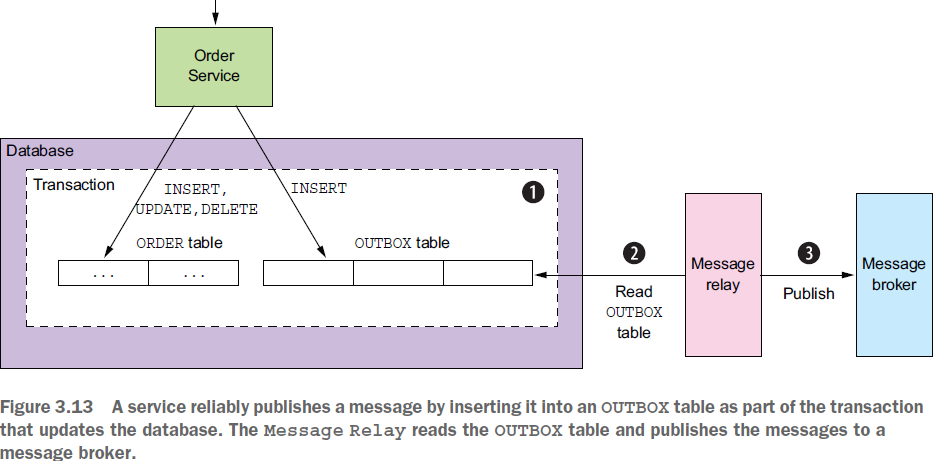
As a result, an application must use a different mechanism to reliably publish messages.

Let’s look at how that works.

### Using a Database Table as a Message Queue

**Be careful with:**

* **What happens if multiple instances of a message relay are reading/publishing the events?**
* **How do you guarantee that the order of published events matches the order in which they have been inserted into database to be processed?**
* **When inserting messages into the database, are you using multiple threads/instances? Are you preserving the order of events correctly?**



Let’s imagine that our application is using a relational database. A straightforward way to reliably publish messages is to apply the Transactional outbox pattern.

This pattern uses **a database table as a temporary message queue**. As figure 3.13 shows, a

service that sends messages has an OUTBOX database table. As part of the database transaction that creates, updates, and deletes business objects, the service sends messages by inserting them into the OUTBOX table. **Atomicity is guaranteed because this is a local ACID transaction.**

The OUTBOX table acts a temporary message queue. The Message Relay is a component

that reads the OUTBOX table and publishes the messages to a message broker.

*The next paragraph needs to be explored a little more:*

we can use a similar approach with some NoSQL databases. Each business entity stored as a record in the database has an attribute that is a list of messages that need to be published. When a service updates an entity in the database, it appends a message to that list. This is atomic because it’s done with a single database operation. The challenge, though, is efficiently finding those business entities that have events and publishing them.

There are a couple of different ways to move messages **from the database** **to the message broker**:

### Polling Publisher

If the application uses a relational database, a very simple way to publish the messages inserted into the OUTBOX table is for the Message Relay to poll the table for unpublished messages. It periodically queries the table:

SELECT \* FROM OUTBOX ORDERED BY ... ASC

Next, the Message Relay publishes those messages to the message broker, sending one to its destination message channel. Finally, it deletes those messages from the OUTBOX table:

BEGIN

DELETE FROM OUTBOX WHERE ID in (....)

COMMIT

* Polling the database is a simple approach that works reasonably well at low scale.
* The downside is that frequently polling the database can be expensive.
* **Also, whether you can use this approach with a NoSQL database depends on its querying capabilities. That’s because rather than querying an OUTBOX table, the application must query the business entities, and that may or may not be possible to do efficiently.**

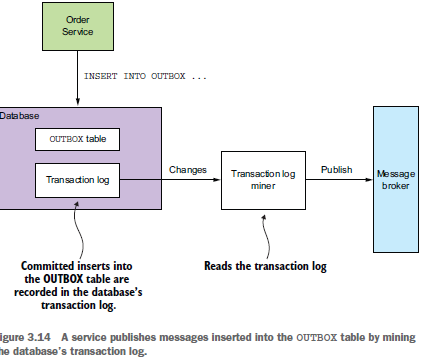
Because of these drawbacks and limitations, it’s often better—and in some cases, necessary—to

use the more sophisticated and performant approach of tailing the database transaction log.

### Transaction Log Tailing

A sophisticated solution is for Message Relay to *tail* the database transaction log (also called the commit log). Every committed update made by an application is represented as an entry in the database’s transaction log.

A transaction log miner can read the transaction log and publish each change as a message to the message broker. Figure 3.14 shows how this approach works.



The Transaction Log Miner reads the transaction log entries. It converts each relevant log entry corresponding to an inserted message into a message and publishes that message to the message broker. This approach can be used to publish messages written to an OUTBOX table in an RDBMS or messages appended to records in a NoSQL database.**(How exactly for NoSQL??)**

I add: *Using this method you have guarantee that messages are published to the broker in order they were inserted into the database.*

Although this approach is obscure, it works remarkably well. The challenge is that implementing it requires some development effort. You could, for example, write low-level code that calls database-specific APIs.

Alternatively, you could use an open-source framework such as Debezium that publishes changes made by an application to MySQL, Postgres, or MongoDB to Apache Kafka. **The drawback of using Debezium is that its focus is capturing changes at the database level and that APIs for sending and receiving messages are outside of its scope. That’s why I created the Eventuate Tram framework, which provides the messaging APIs as well as transaction tailing and polling.**

**I think he refers to API for sending messages to brokers and managing sagas and other features that the Eventuate Tram offers.**

Examples of technologies that help you implement this:

 *Debezium* (http://debezium.io)—An open-source project that publishes database changes to the Apache Kafka message broker.

 *LinkedIn Databus* (https://github.com/linkedin/databus)—An open-source project that mines the **Oracle transaction log** and publishes the changes as events. LinkedIn uses Databus to synchronize various derived data stores with the system of record.

 *DynamoDB streams* (http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html)—DynamoDB streams contain the time-ordered sequence of changes (creates, updates, and deletes) made to the items in a DynamoDB table in the last 24 hours. An application can read those changes from the stream and, for example, publish them as events.

 ***Eventuate Tram***(<https://github.com/eventuate-tram/eventuate-tram-core)—Your> author’s very own open-source transaction messaging library that uses MySQL binlog protocol, Postgres WAL, or polling to read changes made to an OUTBOX table and publish them to Apache Kafka**.(Take a look at this)**

# Libraries and Frameworks for Messaging

Skipped this part, we will learn how to use he eventuate tram framework as we go further through the book.

# Using Asynchronous Messaging to Improve Availability

IPC mechanisms have different trade-offs. One particular trade-off is how your choice of IPC mechanism impacts availability. In this section, you’ll learn that **synchronous communication with other services as part of request handling reduces application availability**. As a result, you should design your services to use asynchronous messaging whenever possible.

Skipped this part, we will learn more about this in the next chapter and get back to this section with more insight.

# Key Take-Aways from the IPC documents

* Synchronous remote procedure invocation-based protocols, such as REST, are the easiest to use. But **services should ideally communicate using asynchronous messaging in order to increase availability.**
* In order to prevent failures from cascading through a system, a service client that uses a synchronous protocol must be designed to handle partial failures, which are when the invoked service is either down or exhibiting high latency. In particular, it must use timeouts when making requests, limit the number of outstanding requests, and use the Circuit breaker pattern to avoid making calls to a failing service.
* An architecture that uses synchronous protocols must include a service discovery mechanism in order for clients to determine the network location of a service instance. The **simplest approach is to use the service discovery mechanism implemented by the deployment platform**: the Server-side discovery and 3rd party registration patterns.
* A good way to design a messaging-based architecture is to use the messages and channels model, which abstracts the details of the underlying messaging system. You can then map that design to a specific messaging infrastructure, which is typically message broker–based.