### **Communicating using the Asynchronous messaging pattern**

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.

**Pattern: Messaging**

A client invokes a service using asynchronous messaging. See <http://microservices.io/patterns/communication-style/messaging.html>

**Overview of messaging**

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). 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:

Text, application

Description automatically generated

**About message channels**

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.

The business logic in the sender invokes a sending port interface, which is implemented by a message sender adapter. The message sender sends a message to a receiver via a message channel. The message channel is an abstraction of messaging infrastructure. A message handler adapter in the receiver is invoked to handle the message. It invokes the receiving port interface implemented by the receiver’s business logic.

Diagram

Description automatically generated

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

Text

Description automatically generated

**Implementing the interaction styles using messaging**

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

Let’s look at how to implement each interaction style, starting with request/response and asynchronous request/response.

**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. 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.

**Implementing asynchronous request/response by including a reply channel and message identifier in the request message. The receiver processes the message and sends the reply to the specified reply channel.**

**Diagram

Description automatically generated**

The client must tell the service where to send a reply message 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, which contains a correlation id that has the same value as message identifier, to the reply channel. The client uses the correlation id to match the reply message with the request.

Because the client and service communicate using messaging, the interaction is inherently asynchronous. 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 notifications**

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.

**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.

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. 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 responses

##### 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.

##### Creating an API specification for a messaging-based service API

##### The specification for a service’s asynchronous API must, specify 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 message channels and command, reply, and event message types:

##### Diagram Description automatically generated

##### 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:

##### Graphical user interface, text, application Description automatically generated

##### 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

##### A messaging-based application typically uses a message broker, an infrastructure service through which the service communicates. But a broker-based architecture isn’t the only messaging architecture. You can also use a brokerless-based messaging architecture, in which the services communicate with one another directly.

##### The two approaches, have different trade-offs, but usually a broker-based architecture is a better approach.

##### The services in brokerless architecture communicate directly, whereas the services in a broker-based architecture communicate via a message broker:

##### Diagram, shape, polygon Description automatically generated

##### Brokerless messaging

##### In a brokerless architecture, services can exchange messages directly. ZeroMQ ([http://zeromq.org](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:

##### 

##### 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

##### 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, request/response.

##### 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 chose from. Examples of popular open source message brokers include the following:

* ActiveMQ ([http://activemq.apache.org](http://activemq.apache.org/))
* RabbitMQ ([https://www.rabbitmq.com](https://www.rabbitmq.com/))
* Apache Kafka ([http://kafka.apache.org](http://kafka.apache.org/))

##### There are also cloud-based messaging services, such as AWS Kinesis (<https://aws.amazon.com/kinesis/>) and AWS SQS (<https://aws.amazon.com/sqs/>).

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

##### Text Description automatically generated

##### 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.

##### Implementing message channels using a message broker

##### 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.

##### Each message broker implements the message channel concept in a different way.

##### Table Description automatically generated

##### Almost all the message brokers described here support both point-to-point and publish-subscribe channels. The one exception is AWS SQS, which only supports point-to-point channels.

##### Now let’s look at the benefits and drawbacks of broker-based messaging.

##### Text, letter Description automatically generated

##### There are some downsides to using messaging:

##### Text Description automatically generated

##### 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.

##### Text, letter Description automatically generated

##### Scaling consumers while preserving message ordering by using a sharded (partitioned) message channel. The sender includes the shard key in the message. The message broker writes the message to a shard determined by the shard key. The message broker assigns each partition to an instance of the replicated receiver.

##### Diagram Description automatically generated

##### 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:

##### Text Description automatically generated with medium confidence

##### 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.

##### 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.

##### A consumer detects and discards duplicate messages by recording the IDs of processed messages in a database table. If a message has been processed before, the INSERT into the PROCESSED\_MESSAGES table will fail.

##### 

##### 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 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.

##### Transactional messaging:

A service often needs to publish messages as part of a transaction that updates the database. Services that publish domain events whenever they create or update business entities. 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 later, distributed transactions aren’t a good choice for modern applications. Moreover, many modern brokers such as Apache Kafka don’t support distributed transactions. As a result, an application must use a different mechanism to reliably publish messages.

**Using a database table as a message queue**

##### Let’s imagine that your 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.

##### 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.

##### A service reliably publishes a message by inserting it into an OUTBOX table as part of the transaction that updates the database. The Message Relay reads the OUTBOX table and publishes the messages to a message broker.

##### Diagram Description automatically generated

##### The OUTBOX table acts a temporary message queue. The MessageRelay is a component that reads the OUTBOX table and publishes the messages to a message broker.

**Pattern: Transactional outbox**

Publish an event or message as part of a database transaction by saving it in an OUTBOX in the database. See <http://microservices.io/patterns/data/transactional-outbox.html>.

##### You 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. We’ll look at each one.

##### Publishing events by using the Polling publisher pattern

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

##### 

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

##### A picture containing logo Description automatically generated

##### Pattern: Polling publisher

##### Publish messages by polling the outbox in the database. See <http://microservices.io/patterns/data/polling-publisher.html>.

##### 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.

##### Publishing events by applying the Transaction log tailing pattern

##### A sophisticated solution is for MessageRelay 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.

##### Diagram Description automatically generated

##### A service publishes messages inserted into the OUTBOX table by mining the database’s transaction log

##### 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.

##### Pattern: Transaction log tailing

##### Publish changes made to the database by tailing the transaction log. See <http://microservices.io/patterns/data/transaction-log-tailing.html>.

##### There are a few examples of this approach in use:

##### Debezium ([http://debezium.io](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.

##### 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.

##### Libraries and frameworks for messaging

##### A service needs to use a library to send and receive messages. One approach is to use the message broker’s client library, although there are several problems with using such a library directly:

##### The client library couples business logic that publishes messages to the message broker APIs.

##### A message broker’s client library is typically low level and requires many lines of code to send or receive a message. As a developer, you don’t want to repeatedly write boilerplate code.

##### The client library usually provides only the basic mechanism to send and receive messages and doesn’t support the higher-level interaction styles.