6

Asynchronous Communication

In the previous chapter, we illustrated how services can communicate with each other using a synchronous request-response model. There are other communication models that provide various benefits to the application developer, such as asynchronous communication, which we are going to cover in this chapter. RatingEventTypePut

In this chapter, you are going to learn the basics of asynchronous communication and some common techniques for using it, as well as some benefits and challenges it brings to microservice developers. We will cover a popular piece of asynchronous communication software, Apache Kafka, and illustrate how to use it for establishing communication between our microservices.

In this chapter, we are going to cover the following topics:

* Asynchronous communication basics
* Using Apache Kafka for messaging
* Asynchronous communication best practices

Technical requirements

To complete this chapter, you need Go 1.18+ or above, similarly to the previous chapters. Additionally, you will need Docker, which you can download at <https://www.docker.com>. You will need to register on the Docker website in order to test service deployments in this chapter.

You can find the GitHub code for this chapter here: [https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition](https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition/tree/main/Chapter06)/tree/main/Chapter06

Asynchronous communication basics

Asynchronous communication is communication between a sender and one or multiple receivers, where a sender does not necessarily expect an immediate response to their messages. In the synchronous communication model, which we covered in Chapter 5, the caller sending the request would expect an immediate (or nearly immediate, considering network latency) response to it. In asynchronous communication, it may take an arbitrary amount of time for the receiver to respond to the request, or to not respond at all (for example, when receiving a no-reply notification).

We can illustrate the differences between the two models using two examples. An example of synchronous communication is a phone call – two people having a phone conversation are in direct and immediate communication with each other, and they expect to hear the responses in real time. An example of asynchronous communication is sending mail to people. It can take time to respond to such mail, and the sender does not expect an immediate response to their messages.

It does not mean, however, that asynchronous communication is necessarily slower than the synchronous model. In most cases, asynchronous processing is as quick as synchronous processing and often can be even faster: asynchronous processing is often much less interruptive and leads to higher processing efficiency. It’s like replying to 10 emails, one by one, compared to switching between 10 parallel phone calls — the latter example of synchronous processing can sometimes be way slower due to context switching and frequent interruptions.

In the following section, we are going to review common benefits and challenges of asynchronous communication, that should help you to understand when to use it in microservice applications.

Benefits and challenges of asynchronous communication

The asynchronous communication model comes with its own benefits and challenges. Developers need to consider both in order to make a decision on whether to use this model. What are the benefits of using asynchronous communication? Let’s find out:

* More streamlined approach to processing messages: Imagine you have a service whose purpose is to process data and report the status to another service. The reporting service does not necessarily need to wait for any response back from the service it is reporting to, as it would do in the synchronous model. In asynchronous mode, it just needs confirmation that the status message was sent successfully. This is similar to sending a large number of postcards to your relatives – if you send a dozen postcards, you don't want to wait until each card gets delivered before sending the next one!
* Ability to de-couple the sending and processing of requests: Imagine a caller requesting a server to convert a large video file to a different format. In a synchronous model, the caller would be waiting in real time until the entire video is processed. This could easily take minutes and sometimes even hours, making such waiting very inefficient. Instead, such a task could be performed in an asynchronous way, where the caller would send the task to a server, get an acknowledgment that the task was received, and perform any other activity until an eventual notification of completion (or processing failure) is received.
* Better load balancing. Certain applications can have uneven request loads and are prone to sudden spikes of requests. If communication is synchronous, the server needs to answer every request in real time, and this can easily overload it. Imagine responding to 10 messages from your friends simultaneously: it is much easier to do this sequentially, one by one.

The benefits that we just described are quite significant, and in many cases, asynchronous communication is the only way to perform certain types of tasks or to provide better system performance. Examples of problems for which asynchronous communication is a good fit include the following:

* Long-running processing tasks: Long-running tasks, such as video processing, are often better done asynchronously. The caller requesting such processing would not necessarily need to wait until it is completed and would eventually get notified of the final result.
* Send once, processed by multiple components: Certain types of messages, such as status reports, can be processed by multiple independent components. Imagine a system where multiple employees need to receive the same message – instead of sending it to each one independently, the message can get published to a component that can be consumed by everyone interested.
* High-performance sequential processing: Certain types of operations (for example, big data processing) are more efficient when performed sequentially and/or in batches. For such scenarios, asynchronous processing offers great performance improvements compared to more interactive and interruptive synchronous communication because the receiver of such requests can control the processing speed and process tasks one after another.

While the described benefits of asynchronous communication may seem appealing, it is important to note that it often brings some difficult challenges to developers:

* More complex error handling: Imagine sending a message to your friend and not receiving a response back. Was it because the friend did not receive the message? Did something happen during this time? Did the response get lost? In synchronous communication, such as a phone call, we would immediately know if the friend is not available and would be able to call back. In the case of an asynchronous scenario, we would need to think about more possible issues, such as the ones we described.
* Reliance on additional components for message delivery: Certain asynchronous communication use cases, such as the publisher-subscriber or message broker models described in the next section, require additional software for delivering messages. Such software often performs additional operations, such as message batching and storing, bringing additional complexity to the system in exchange for additional features it provides.
* Asynchronous data flow may seem non-intuitive to many developers and be more complex: Unlike the synchronous request-response model, where each request is logically followed by a response to it, asynchronous communication may be unidirectional (no responses are received at all) or may require the caller to perform additional steps in order to receive a response (for example, when the response is sent as a separate notification). Because of this, data flow in asynchronous systems may be more complex than in synchronous request-response interactions.

Now, let's cover some asynchronous communication techniques and patterns that can help you in organizing your services and establishing asynchronous communication between them.

Techniques and patterns of asynchronous communication

There are various techniques that help to make the asynchronous interaction between multiple services more efficient in various scenarios, such as sending a message to multiple recipients. In this section, we are going to describe multiple patterns that help to facilitate such interactions.

Message broker

A message broker is an intermediary component in the communication chain that can play multiple roles:

* Message delivery: It performs the delivery of a message to one or multiple receivers
* Message transformation: It transforms an incoming message into another format that can be later consumed by receivers
* Message batching: It combines multiple messages into a single one for more efficient delivery or processing
* Message routing: It routes incoming messages to the appropriate destination based on pre-defined rules

When you send a postcard to your friend or a relative, the post office plays the role of a message broker, playing an intermediary role in delivering it to the destination. In this example, the main benefit of using the message broker would be the convenience of sending the message (postcard, in our example) without any need to think about how to deliver it. Another benefit of using message brokers is delivery guarantees. A message broker can provide various levels of guarantees for message delivery. Examples of these guarantees include the following:

* At-least-once: The message gets delivered at least once, but may be delivered multiple times in case of failures
* Exactly-once: The message broker guarantees that the message gets delivered and it will be delivered exactly once
* At-most-once: The message can be delivered 0 or 1 time

The exactly-once guarantee is often harder to achieve in practice than at-least-once and at-most-once. In the at-least-once model, a message broker can just re-send the message in case of any failure (such as a sudden power loss or a restart). In the exactly-once model, the message broker needs to perform additional checks or store extra metadata to ensure the message is never re-sent to the receiver in any possible case.

Another classification of message brokers is based on the possibility of them losing messages:

* Lossy: A message broker that can occasionally (for example, in case of failures) lose messages
* Lossless: A message broker that provides a guarantee of not losing any messages

The at-most-once guarantee is an example of a lossy message broker, and at-least-once and exactly-once brokers are examples of lossless ones. Lossy message brokers are faster than lossless ones because they don't need to handle extra logic for guaranteeing message delivery, such as persisting messages.

The publisher-subscriber model

The publisher-subscriber model is a model of communication between multiple components (such as microservices) where every component can publish messages and subscribe to the relevant ones.

Let's take Twitter as an example. Any user can publish messages to their feeds, and other users can subscribe to them. Similarly, microservices can communicate by publishing the data that other services can consume. Imagine that we have a set of services that process various types of user data, such as user photos, videos, and text messages. If a user deleted their profile, we would need to notify all services about this. Instead of notifying each service one by one, we could publish a single event that would indicate that a user profile is deleted, and all services could consume it and perform any relevant actions, such as archiving user data.

The relationship between the publishers, the subscribers, and the data produced by the publisher is illustrated in the following diagram:

Published data

Publisher

Subscriber

Subscriber

Subscriber

Figure 6.1 – The publisher-subscriber model

The publisher-subscriber model provides a flexible solution for sending and delivering data in a system where messages can be processed by multiple components. Each publisher can publish their messages without caring about the delivery process and any difficulties in delivering messages to an arbitrary number (even a very large one) of receivers. Each subscriber can subscribe to the relevant messages and get them delivered without needing to contact the publisher directly and check if there is any new data to consume.

Now, as we have covered some high-level asynchronous communication models, let's move on to the practical side of the chapter and illustrate how you can implement asynchronous communication in your microservices.

Using Apache Kafka for messaging

In this section, we are going to introduce you to Apache Kafka, a popular message broker system that we are going to use to establish asynchronous communication between our microservices. You will learn the basics of Kafka, how to publish messages to it, and how to consume such messages from the microservices we created in the previous chapters.

Apache Kafka basics

Apache Kafka is an open source message broker system that provides the ability to publish and subscribe to messages containing arbitrary data. Originally developed at LinkedIn, Kafka has become perhaps the most popular open source message broker software and is used by thousands of companies around the world.

In the Kafka model, a component that publishes messages is called a producer. Messages are published in sequential order to objects called topics. Each message in a topic has a unique numerical offset in it. Kafka provides APIs for consuming messages (the component for consuming messages is called a consumer) for the existing topics. Topics can also be partitioned to allow multiple consumers to consume from them (for example, for parallel data processing).

We can illustrate the Kafka data model in the following diagram:

Topic

Producer

Consumer

...

Figure 6.2 – The Apache Kafka data model

Having such a seemingly simple data model, Kafka is a powerful system that offers lots of benefits to its users:

* High sequential write and read throughput: Kafka is optimized for highly performant write and read operations. It achieves this by doing as many sequential writes and reads as possible, allowing it to efficiently make use of hardware such as hard disk drives, as well as sequentially sending large amounts of data over the network.
* Scalability: Developers can leverage topic partitioning provided by Kafka to achieve more performant parallel processing of their data.
* Flexible durability: Kafka allows users to configure the policies for storing data, such as message retention. Messages can be stored for a fixed amount of time (for example, for 7 days) or indefinitely until there is enough space on the data storage.

Note

While Kafka provides many benefits for developers, it is important to note that it is a fairly complex infrastructure component that may be nontrivial to manage and maintain. We are going to use it in this chapter for illustrative purposes, especially taking into account its wide adoption and popularity in the developer community. In this chapter, we will avoid the difficulties of setting up a Kafka cluster by using its Docker version, but for production use cases you may need to get familiar with the relevant Kafka maintenance documentation, available at https://kafka.apache.org/documentation/.

Let’s explore how we can leverage the benefits offered by Kafka for the microservices we developed in the previous chapters.

Adopting Kafka for our microservices

Let’s get back to the rating service example from the previous chapters. The service provided a synchronous API for inserting rating records, allowing its callers to call an endpoint and get an immediate response from the service. Such an API would be useful in many practical use cases, including one where the user submits a rating from a user interface or a web form.

Now consider a scenario where we work with a data provider who frequently publishes rating records (for example, movie ratings from a popular movie database, such as IMDb) that we can use in our rating service. Here, we would need to consume such records and ingest them into our system, so we could use them in addition to the data that was created through our API. The publisher-subscriber model that we described earlier in this chapter would be a great fit for this use case – the publisher would be the data provider that provides the rating data, and the subscriber would be a part of our application (such as a rating service), which would consume the data.

We can illustrate the described model using the following diagram:

Rating data

Data provider

Rating service

...

publish

subscribe

Rating DB

Figure 6.3 – The publisher-subscriber model of rating ingestion from a data provider

The model of interaction between the data provider and the rating service is a perfect example of asynchronous communication – the rating service does not necessarily need to process the provider’s data immediately. It is up to us when and how to consume this data – our rating service could do this periodically (for example, once an hour, or once a day), or handle the new rating data as soon as it gets published. Let’s choose the second approach in this chapter.

The only missing piece in our model is the component that allows us to publish the rating data from the data provider and subscribe to it from our rating service. Apache Kafka, which we described earlier, is a great fit for this use case – it provides a performant, scalable, and durable solution for producing and consuming arbitrary data, allowing us to use it as a rating data message broker.

To illustrate the model that we have just described, let’s implement the following logic:

* A new example application that will produce rating data for Apache Kafka
* Logic in the rating service to consume the rating data from Apache Kafka and save it to our rating database

Before we proceed to implement both components, we need to decide which data serialization format to use between them. For simplicity, let’s assume the data provider provides us with the rating data in JSON format. An example of the provided rating data would be as follows:

[{"userId":"105","recordId":"1","recordType":1,"value":5,"providerId":"test-provider","eventType":"put"},{"userId":"105","recordId":"2","recordType":1,"value":4,"providerId":"test-provider","eventType":"put"}]

Let’s define a Go structure for such rating records. In the src/rating/pkg/model/rating.go file, add the following code:

// RatingEvent defines an event containing rating information.

type RatingEvent struct {

Rating

ProviderID string `json:"providerId"`

EventType RatingEventType `json:"eventType"`

}

// RatingEventType defines the type of a rating event.

type RatingEventType string

// Rating event types.

const (

RatingEventTypePut = RatingEventType("put")

RatingEventTypeDelete = RatingEventType("delete")

)

In our code, we used a technique called type embedding — we embedded a Rating type into a RatingEvent structure so we don’t need to re-define same fields as the Rating type has.

Now, let’s implement the example application that reads rating data from a provided file and produces it in Kafka. Create a cmd/ratingproducer directory and add a main.go file, containing the following code:

package main

import (

    "encoding/json"

    "fmt"

    "os"

    "time"

    "github.com/confluentinc/confluent-kafka-go/kafka"

    "movieexample.com/rating/pkg/model"

)

func main() {

    fmt.Println("Creating a Kafka producer")

    producer, err := kafka.NewProducer(&kafka.ConfigMap{"bootstrap.servers": "localhost"})

    if err != nil {

        panic(err)

    }

    defer producer.Close()

    const fileName = "ratingsdata.json"

    fmt.Println("Reading rating events from file " + fileName)

    ratingEvents, err := readRatingEvents(fileName)

    if err != nil {

        panic(err)

    }

    const topic = "ratings"

    if err := produceRatingEvents(topic, producer, ratingEvents); err != nil {

        panic(err)

    }

    const timeout = 10 \* time.Second

    fmt.Println("Waiting " + timeout.String() + " until all events get produced")

    producer.Flush(int(timeout.Milliseconds()))

}

In the code that we just added, we initialize a Kafka producer by calling kafka.NewProducer, read the rating data from a file, and produce rating events containing the rating data in Kafka. Note that we import the github.com/confluentinc/confluent-kafka-go/kafka Kafka library — a Kafka client made by Confluent, a company founded by the creators of Kafka. There are multiple popular open source Kafka libraries for Go, including github.com/Shopify/sarama, which is well maintained and is widely used across many Go projects. You can use either library in your projects depending on your preference.

Now, let’s add a function for reading rating events to the file we just created:

func readRatingEvents(fileName string) ([]model.RatingEvent, error) {

    f, err := os.Open(fileName)

    if err != nil {

        return nil, err

    }

    defer f.Close()

    var ratings []model.RatingEvent

    if err := json.NewDecoder(f).Decode(&ratings); err != nil {

        return nil, err

    }

    return ratings, nil

}

Finally, add a function for producing rating events:

func produceRatingEvents(topic string, producer kafka.Producer, events []model.RatingEvent) error {

    for \_, ratingEvent := range events {

        encodedEvent, err := json.Marshal(ratingEvent)

    if err != nil {

        return err

    }

    if err := producer.Produce(&kafka.Message{

TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},

Value:          []byte(encodedEvent),

}, nil); err != nil {

        return err

    }

    return nil

}

Let’s describe some parts of the code that we just wrote:

* We created a Kafka producer by calling a kafka.NewProducer function and providing localhost as the Kafka address for testing it locally.
* The program that we created is expected to read rating data from the ratingsdata.json file.
* When we produce events to Kafka using a Produce function, we specify a topic partition using a kafka.TopicPartition structure. In the structure, we provide the topic name (in our example, we call it ratings) and the topic partition (in our example, we use kafka.PartitionAny to produce a partition — we will cover this part later, in the Asynchronous communication best practices section).
* At the end of our main function, we call the Flush function to make sure all messages are sent to Kafka.

The function that we just created is using the github.com/confluentinc/confluent-kafka-go/kafka library, which we need to include in our Go module. Let’s do this by running the following code:

go mod tidy

Let’s also add a file containing the rating events. In the directory that we just used, create a ratingsdata.json file, containing the following code:

[{"userId":"105","recordId":"1","recordType":1,"value":5,"providerId":"test-provider","eventType":"put"},{"userId":"105","recordId":"2","recordType":1,"value":4,"providerId":"test-provider","eventType":"put"}]

Now, our application is ready. We have implemented the logic to read the rating data from a file and publish it to Apache Kafka for further consumption by the rating service. Let’s implement the logic in the rating service to consume the published data. Create a rating/internal/ingester/kafka directory and add an ingester.go file with the following contents:

package kafka

import (

    "context"

    "encoding/json"

    "fmt"

    "rating/pkg/model"

    "github.com/confluentinc/confluent-kafka-go/kafka"

    "movieexample.com/rating/pkg/model"

)

// Ingester defines a Kafka ingester.

type Ingester struct {

    consumer \*kafka.Consumer

    topic    string

}

// NewIngester creates a new Kafka ingester.

func NewIngester(addr string, groupID string, topic string) (\*Ingester, error) {

    consumer, err := kafka.NewConsumer(&kafka.ConfigMap{

        "bootstrap.servers": addr,

        "group.id":          groupID,

        "auto.offset.reset": "earliest",

    })

    if err != nil {

        return nil, err

    }

    return &Ingester{consumer, topic}, nil

}

Additionally, add this piece of code to it:

// Ingest starts ingestion from Kafka and returns a channel containing rating events

// representing the data consumed from the topic.

func (i \*Ingester) Ingest(ctx context.Context) (chan model.RatingEvent, error) {

fmt.Println("Starting Kafka ingester")

    if err := i.consumer.SubscribeTopics([]string{i.topic}, nil); err != nil {

        return nil, err

    }

    ch := make(chan model.RatingEvent, 1)

    go func() {

        for {

            select {

            case <-ctx.Done():

                close(ch)

                i.consumer.Close()

            default:

        }

        msg, err := i.consumer.ReadMessage(-1)

        if err != nil {

            fmt.Println("Consumer error: " + err.Error())

            continue

        }

        fmt.Println("Processing a message")

        var event model.RatingEvent

        if err := json.Unmarshal(msg.Value, &event); err != nil {

            fmt.Println("Unmarshal error: " + err.Error())

            continue

        }

        ch <- event

        }

    }()

    return ch, nil

}

In the code we just created, we have implemented a NewIngester function to create a new Kafka ingester, the component that will ingest rating events from it. The Ingest function starts message ingestion in the background and returns a Go channel with RatingEvent structures.

You may notice that in our call to the ReadMessage function, we provided -1 as an argument. We specified a consumer offset — a checkpoint from which we should consume the messages from our topic. The value of -1 is specific to Kafka and means that we will always consume from the beginning of the topic, reading all existing messages.

Let’s use this structure in our rating service controller. In our rating/internal/controller/controller.go file, add the following code:

type ratingIngester interface {

    Ingest(ctx context.Context) (chan model.RatingEvent, error)

}

// StartIngestion starts the ingestion of rating events.

func (s \*RatingService) StartIngestion(ctx context.Context) error {

    ch, err := s.ingester.Ingest(ctx)

    if err != nil {

        return err

    }

    for e := range ch {

fmt.Printf("Consumed a message: %v\n", e)

        if err := s.PutRating(ctx, e.RecordID, e.RecordType, &model.Rating{UserID: e.UserID, Value: e.Value}); err != nil {

            return err

        }

    }

    return nil

}

In our code, we call the Ingest function and get back a Go channel containing rating events from the topic. We iterate over it using the for operator. It keeps returning us available rating events until the channel is closed (for example, when the Kafka client is closed on service shutdown).

Now, update the existing RatingService structure and the New function in this file to the following:

// RatingService encapsulates the rating service business logic.

type RatingService struct {

    repo     ratingRepository

    ingester ratingIngester

}

// New creates a rating service.

func New(repo ratingRepository, ingester ratingIngester) \*RatingService {

    return &RatingService{repo, ingester}

}

Now, our rating service is able to asynchronously consume rating events from Kafka, and execute the Put function for each one, writing it to the rating database. The remaining step for the rating service is to update its main.go file.

1. First, add an extra import of movieexample.com/rating/internal/ingester/kafka package to it.
2. Find the line where we initialize the ctrl variable and replace it with the following code:

ingester, err := kafka.NewIngester("localhost", "rating", "ratings")

if err != nil {

log.Fatalf("failed to initialize ingester: %v", err)

}

ctrl := rating.New(repo, ingester)

if err := ctrl.StartIngestion(ctx); err != nil {

log.Fatalf("failed to start ingestion: %v", err)

}

At this point, the rating service provides both a synchronous API for the callers that want to create ratings in real time and asynchronous logic for ingesting rating events from Apache Kafka.

Let’s run our rating service so we can test message ingestion. Navigate to rating/cmd directory and run its main.go file. You will see regular messages indicating that the service has started, but we haven’t produced any data to our topic yet — let’s do this now without stopping the rating service.

Create a file called docker-compose.yml in our cmd/ratingproducer directory, containing the following code:

version: '3'

services:

zookeeper:

image: wurstmeister/zookeeper

container\_name: zookeeper

ports:

- "2181:2181"

kafka:

image: wurstmeister/kafka

container\_name: kafka

ports:

- "9092:9092"

environment:

KAFKA\_ADVERTISED\_HOST\_NAME: localhost

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

In our code, we define Apache Kafka configuration required for running it locally in a Docker container. Note that the file also contains Zookeeper configuration — it is required by Kafka.

Credit

Big thanks to Jason Salas <https://github.com/jasonsalas> who shared this code example on GitHub after reading the first edition of this book.

Now, let’s run Kafka in a Docker container. Go to our project root directory and run the following commands:

docker-compose -d up

docker exec -it kafka /bin/sh

cd /opt/kafka\_<YOUR KAFKA VERSION>/bin

kafka-topics.sh --create --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic ratings

Then, let’s open a separate terminal and run the ratingproducer application:

cd cmd/ratingproducer

go run main.go

Our application produced rating events from a JSON file. Open the window with the running rating service and you should see the following messages in its output:

Consumed a message: {105 1 movie 5 test-provider put}

Consumed a message: {105 2 movie 4 test-provider put}

These new messages from our rating service confirm that we successfully received the data we just asynchronously produced by our ratingproducer application via Kafka. Now we are able to communicate between the services in both synchronous and asynchronous ways.

At this point, we have covered the basics of asynchronous communication and can proceed to the final part of the chapter to see some best practices you should keep in mind while using this model.

Asynchronous communication best practices

In this section, we are going to cover the best practices of using the asynchronous communication model. You will learn some high-level recommendations for adopting the model in your applications and using it in a way that would maximize its benefits for you.

Versioning

Versioning is the technique of associating the format (or a schema) of the data with its version. Imagine you are working on a rating service, and you use a publisher-subscriber model for producing and consuming rating events. If at some point the format of your rating events gets changed, some of the events that are already produced will have an old data format, and some will have the new one. This situation may be hard to handle because the logic consuming such data would need to know how to differentiate between such formats and how to handle each one. Differentiating between two formats without knowing the data schema or its version could be a nontrivial task. Imagine that we have two JSON events:

{"recordID": "1", "rating": 5}

{"recordID": "2", "rating": 17, "userId": "alex"}

The second event has a userId field that is not present in the first. Is it because the producer did not provide it or because the data format did not have this field before?

Providing the schema version explicitly would help the data consumer handle this problem. Consider these updated examples:

{"recordID": "1", "rating": 5, "version": 1}

{"recordID": "2", "rating": 17, "userId": "alex", "version": 2}

In these examples, we know the versions of events and can now handle each one separately. For example, we may completely ignore events of a certain version (assume there was an application bug and we want to re-process events with an updated version instead) or use the version-specific validation (for instance, allow the records without a userId field for version 1, but disallow for the higher versions).

Versioning is very important to systems that can evolve over time because it makes dealing with different data formats easier. Even if you don’t expect your data format to change, consider using versioning to increase your system’s maintainability in the future.

Leveraging partitioning

In the code examples in the Adopting Apache Kafka for our microservices section, we implemented the logic for producing our data to message topics in Apache Kafka. The function for producing a message was as follows:

if err := p.Produce(&kafka.Message{

TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},

Value:          []byte(encodedEvent),

}, nil); err != nil {

    return err

}

In this function, we used the kafka.PartitionAny option. As we mentioned in the Apache Kafka basics section, Kafka topics can be partitioned to allow multiple consumers to consume different partitions of a topic. Imagine you have a topic with three partitions – you can consume each one independently, as illustrated in the following diagram:

Topic partition 0

Consumer 0

Topic partition 1

Consumer 1

Topic partition 2

Consumer 2

Figure 6.4 – A partitioned topic consumption example

You can control the number of topic partitions, as well as the partition for each message your services produce. Setting a partition manually may help you to achieve data locality — the ability to co-locate the data for various records, storing it together (in our use case, in the same topic partition). For example, you can partition the data using a user identifier, making sure the data for any user is stored on a single topic partition, helping you simplify the data search across the topic partitions.

It is important to note that topic partitioning might bring an extra challenge with message ordering. Messages within each partition will be ordered according to the time they were written to Kafka. However, if your data is produced into multiple partitions, order will be kept only within each partition. If you produce multiple messages A, B, C and all of them get written into different partitions of the same topic, you can get them in various orders, including A, C, B or even C, B, A. The exact order would depend of your producer and consumer configuration, as well as various factors like timing of the events and even consumption speed. Hence, to achieve correct ordering of events (for example, to make all messages for each movie come into the same partition), you can control which partition you write messages to, like in the following example:

const partitionCount := 3

partitionFunc := func(event model.RatingEvent) int32 {

h := fnv.New32a()

h.Write([]byte(event.RecordID))

return int32(h.Sum32()) % partitionCount

}

producer.Produce(&kafka.Message{

TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: partitionFunc(event)},

Value: []byte(encodedEvent),

}

In our example, we manually calculate a partition for input rating using RatingID field to make sure that we always get the same partition number for each distinct RatingID value.

Use explicit message acknowledgement whenever necessary

Some message broker systems, such as Apache Kafka, provide an ability to control when each message is acknowledged as delivered. By default, many Kafka libraries, including confluent-kafka-go, mark all previously received messages as delivered every few seconds via a mechanism called offset commits. With this mechanism, for each topic partition consumed, the last consumed offset is periodically saved to avoid duplicate delivery. However such periodic acknowledgement of received messages has some downsides, such as possibility of message loss. Consider a scenario when you receive a message and run complex processing logic, that takes a long time to complete. In such a case, if message offset is committed before the message gets processed, there is a chance that any unexpected error might result in leaving that message as unprocessed. To avoid this, Apache Kafka allows to commit offsets manually. The following code illustrates this approach:

consumer, err := kafka.NewConsumer(&kafka.ConfigMap{

    "bootstrap.servers": addr,

    "group.id":          groupID,

    "auto.offset.reset": "earliest",

"enable.auto.commit": false,

"enable.auto.offset.store": false,

})

if err != nil {

return err

}

msg, err := i.consumer.ReadMessage(time.Minute)

if err != nil {

return err

}

// Process your message.

// ...

// Acknowledge message delivery.

\_, err = consumer.CommitMessage(msg)

If err != nil {

return err

}

In our example, we call the CommitMessage function only after we process the message, so we don’t lose track of it in case if we experience any processing error. Note that we also specified two configuration parameters to make this work: enable.auto.commit and enable.auto.offset.store should be set to false to make such explicit offset commits work.

Use a separate topic for unprocessed messages

One of the main challenges of asynchronous communication is error handling. Consider the following scenario: you read messages from a Kafka topic that contains updates to movie metadata. Most messages are processed successfully, however one message contains metadata for a movie that is not stored in our system (let’s assume its data is imported separately and the import has not happened yet). In such a scenario, you generally have two options:

1. Write an error and retry processing of the failed message.
2. Skip the failed message and proceed to the next one.

Option 1 might not result in anything: even after max number of retries, you might still experience the same error. In many cases, you also can’t retry indefinitely: when you process messages sequentially, retrying to process a single message blocks processing of all following ones. Option 2 would result in ignoring the failed message, and sometimes this might cause other issues, such as data inconsistency.

A common solution to such a problem is to introduce a separate topic for unprocessed messages: in case if we experience an unexpected error and can’t continue re-processing the same message after max number of retries, we would write it to a separate topic and try to re-process it later (for example, after we identify and fix the underlying problem). This approach is often called Dead Letter Queue (DLQ) and can be helpful in multiple other cases:

1. Message poisoning: Some messages might contain corrupted or unexpected metadata and require separate fixing.
2. Message expiry: Certain messages might take too long to be processed, so instead of blocking the execution, you might just store and re-process them separately.
3. Unexpected error handling: Imagine that you get millions of input messages and only one of them causes your service to panic. Moving the message to a separate topic for further inspection would help to avoid blocking message processing.

The list of best practices that we just described is not comprehensive. It does not cover all recommendations for using asynchronous communication in your microservices, but it provides some great ideas of what you should consider. Get familiar with the articles listed in the Further reading section for some additional ideas and recommendations.

Summary

In this chapter, we have covered the basics of asynchronous communication and illustrated how to use it in your microservices. You have learned the benefits of asynchronous communication and the common patterns, such as publisher-subscriber and message broker. In addition to this, we have covered the basics of the Apache Kafka tool and illustrated how to use it in our microservices by implementing the logic for producing and consuming data from it. This knowledge should help you establish efficient communication between your microservices in a wide variety of scenarios, including complex distributed environments.

In the next chapter, we are going to cover another important topic of microservice development – deployment and orchestration. You will learn how to build and deploy your application to use it in more advanced environments, such as cloud infrastructure.

Further reading

* Apache Kafka documentation: https://kafka.apache.org/documentation/
* Publish-subscribe pattern: https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe\_pattern
* Asynchronous message-based communication: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication>