5

Synchronous Communication

In this chapter, we are going to cover the most common way of communicating between microservices – synchronous communication. In Chapter 2, we already implemented the logic for communicating between microservices via the HTTP protocol and returning results in JSON format. In Chapter 4, we illustrated that the JSON format is not the most efficient in terms of data size, and there are many different formats providing additional benefits to developers, including code generation. In this chapter, we are going to show you how to define service APIs using the Protocol Buffers format and generate both client and server code for them.

By the end of this chapter, you will understand the key concepts of synchronous communication between microservices and will have learned how to implement microservice clients and servers.

The knowledge you gain in this chapter will help you to learn how to better organize the client and server code, generate the code for serialization and communication, and use it in your microservices. In this chapter, we will cover the following topics:

* Introduction to synchronous communication
* Defining a service API using Protocol Buffers
* Implementing gateways and clients
* Synchronous communication best practices

Technical requirements

To complete this chapter, you will need Go 1.18 or above and a Protocol Buffers compiler <https://grpc.io/docs/protoc-installation/>.

You will also need a Go gRPC plugin that you can install by running the following command:

go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest

export PATH="$PATH:$(go env GOPATH)/bin"

Additionally, you will need to install grpcurl tool: https://github.com/fullstorydev/grpcurl.

You can find the GitHub code for this chapter at <https://github.com/PacktPublishing/Microservices-with-Go---Second-Edition>/tree/main/Chapter05.

Introduction to synchronous communication

In this section, we are going to cover the basics of synchronous communication and introduce you to some additional benefits of Protocol Buffers that we are going to use for our microservices.

Synchronous communication is the way of interaction between network applications, such as microservices, where services exchange data using a request-response model. The process is illustrated in the following diagram:

Client

Server

request

response

Figure 5.1 – Synchronous communication

Requests and responses can either take a form of individual messages, or be represented as sequences of messages, called streams. Consider a video streaming service: a client would be sending a request to get a long video and get the response from a server as a sequence of messages, each containing a small chunk of video data.

There are many protocols for synchronous communication between services. HTTP is one of the most popular among them.

Each communication protocol defines its own data model for sending request and response data. In case of HTTP, this includes the following:

* URL parameters: In the case of the https://www.google.com/search?q=portugal URL, q is a URL parameter.
* Headers: Each request and response includes optional key-value pairs called headers, allowing you to propagate additional contextual metadata, such as the client or browser name; for example, User-Agent: Mozilla/5.0.
* Request and response body: The request and response can include a body that contains arbitrary data. For example, when a client uploads a file to a server, the file contents are usually sent as a request body.

When a server cannot handle a client request due to an error or the request is not received due to network issues, the client receives a specific response indicating an error. In the case of the HTTP protocol, there are two types of errors:

* Client error: This error is caused by the client. Examples of such errors include invalid request arguments (such as an incorrect username), unauthorized access, and access to a resource that is not found (for example, a non-existing web page).
* Server error: This error is caused by the server. This could be an application bug or an error with an upstream component, such as a database.

In Chapter 2, we implemented our API handlers by sending the result data as an HTTP response body in JSON format. We achieved this by using the Go JSON encoder:

if err := json.NewEncoder(w).Encode(details); err != nil {

    log.Printf("Response encode error: %v\n", err)

}

As discussed in the previous chapter, the JSON format is not the most optimal in terms of data size. Also, it does not offer useful tools, such as the cross-language code generation of data structures, that are provided by the formats, such as Protocol Buffers. Additionally, sending requests over HTTP and encoding the data manually is not the only form of communication between the services. There are some existing remote procedure call (RPC) libraries and frameworks that help to communicate between multiple services and offer some additional features to application developers:

* Automated serialization and deserialization: RPC libraries take responsibility of serializing and deserializing your communication data, as well as returning correct error codes in case if message data is invalid.
* Client and server code generation: Developers can generate the client code for connecting and sending data to other microservices, as well as generate the server code for accepting incoming requests.
* Authentication: Most RPC libraries and frameworks offer authentication options for cross-service requests — we are going to review them in Chapter 10.
* Context propagation: This is the ability to send additional data with requests, such as traces, which we are going to cover in Chapter 13.
* Documentation generation: Thrift can generate HTML documentation for services and data structures.

In the next section, we are going to cover some of the RPC libraries that you can use in your Go services, along with the features they provide.

Go RPC frameworks and libraries

Let's review some popular RPC frameworks and libraries that are available for Go developers.

Apache Thrift

We already covered Apache Thrift in Chapter 4 and mentioned its ability to define RPC services – sets of functions provided by an application, such as a microservice. Here is an example of a Thrift RPC service definition:

service MetadataService {

  Metadata get(1: string id)

}

The Thrift definition of a service can be used to generate client and server code. The client code would include the logic for connecting to an instance of a service, as well as making requests to it, serializing and deserializing the request and response structures. The advantage of using a library such as Apache Thrift over making HTTP requests manually is the ability to generate such code for multiple languages: a service written in Go could easily talk to a service written in Java, while both would use the generated code for the communication, removing the need of implementing serialization/deserialization logic.

gRPC

gRPC is an RPC framework that was created at Google. gRPC uses HTTP/2 as the transport protocol and Protocol Buffers as a serialization format. Similar to Apache Thrift, it provides an ability to define RPC services and generate the client and server code for the services. In addition to this, it offers some extra features, such as the following:

* Authentication
* Context propagation
* Documentation generation

gRPC protocol separates communication into four distinct types:

* Unary: A client sends a single message as a request and gets a single message as a response.
* Client streaming: A client sends a sequence of messages (for example, small chunks of a large file during the upload process) and gets a single message as the response (in the file upload case, this could include the identifier of the uploaded file).
* Server streaming: A client sends a single message as a request and gets a response as a sequence of messages (for example, a sequence of file chunks when downloading a large file).
* Bi-directional streaming: Both client and server send sequences of messages to each other.

Note

Streaming adds some extra overhead, such as increased memory and CPU usage for keeping client and server connections active, so should be used for specific limited use cases, such as large file uploads or media streaming. We are going to provide some streaming-specific examples in Chapter 16.

gRPC adoption is much higher than for Apache Thrift, and its support of the popular Protocol Buffers format makes it a great fit for microservice developers. In this book, we are going to use gRPC as a framework for synchronous communication between our microservices. In the next section, we are going to illustrate how to leverage the features provided by Protocol Buffers to define our service APIs.

Defining a service API using Protocol Buffers

Let's demonstrate how to define a service API using the Protocol Buffers format and generate the client and server gRPC code for communication with each of our services using a proto compiler. This knowledge will help you to establish a foundation for both defining and implementing APIs for your microservices using one of the industry's most popular communication tools.

Let's start with our metadata service and write its API definition in the Protocol Buffers schema language.

Open the api/movie.proto file that we created in the previous chapter and add the following to it:

service MetadataService {

    rpc GetMetadata(GetMetadataRequest) returns (GetMetadataResponse);

    rpc PutMetadata(PutMetadataRequest) returns (PutMetadataResponse);

}

message GetMetadataRequest {

    string movie\_id = 1;

}

message GetMetadataResponse {

    Metadata metadata = 1;

}

The code we just added defines our metadata service and its GetMetadata endpoint. We already have the Metadata structure from the previous chapter that we can reuse now.

Let's note some aspects of the code we just added:

* Request and response structures: It's a good practice to create a new structure for both a request and a response. In our example, they are GetMetadataRequest and GetMetadataResponse.
* Naming: You should follow consistent naming rules for all your endpoints. We are going to prefix all request and response functions with the function name.

Now, let's add the definition of the rating service to the same file:

service RatingService {

    rpc GetAggregatedRating(GetAggregatedRatingRequest) returns (GetAggregatedRatingResponse);

    rpc PutRating(PutRatingRequest) returns (PutRatingResponse);

}

message GetAggregatedRatingRequest {

    string record\_id = 1;

    int32 record\_type = 2;

}

message GetAggregatedRatingResponse {

    double rating\_value = 1;

}

message PutRatingRequest {

    string user\_id = 1;

    string record\_id = 2;

    int32 record\_type = 3;

    int32 rating\_value = 4;

}

message PutRatingResponse {

}

Our rating service has two endpoints, and we defined requests and responses for them in a similar way to the metadata service.

Note

As an alternative to using int32 type for storing record\_type fields, we could have used Protocol Buffers enumerations: <https://protobuf.dev/programming-guides/proto3/#enum>  
Both options have some upsides: our rating format is more generic and could be used for storing ratings for any types of records, while enumeration-based would be more specific, yet requires to keep supported values in sync between clients and servers.

Finally, let's add the definition of the movie service to the same file:

service MovieService {

    rpc GetMovieDetails(GetMovieDetailsRequest) returns (GetMovieDetailsResponse);

}

message GetMovieDetailsRequest {

    string movie\_id = 1;

}

message GetMovieDetailsResponse {

    MovieDetails movie\_details = 1;

}

Now our movie.proto file includes both our structure definitions and the API definitions for our services. We are ready to generate code for the newly added service definitions. In the src directory of the application, run the following:

protoc -I=api --go\_out=. --go-grpc\_out=. movie.proto

The preceding command is similar to the command that we used in the previous chapter for generating code for our data structures. However, it also passes a --go-grpc\_out flag to the compiler. This flag tells the Protocol Buffers compiler to generate the service code in gRPC format.

Let's see the compiled code that was generated as the output for our command. If the command is executed without any errors, you will find a movie\_grpc.pb.go file inside the src/gen directory. The file will include the generated Go code for our services. Let's take a look at the generated client code:

type MetadataServiceClient interface {

    GetMetadata(ctx context.Context, in \*GetMetadataRequest, opts ...grpc.CallOption) (\*GetMetadataResponse, error)

}

type metadataServiceClient struct {

    cc grpc.ClientConnInterface

}

func NewMetadataServiceClient(cc grpc.ClientConnInterface) MetadataServiceClient {

    return &metadataServiceClient{cc}

}

func (c \*metadataServiceClient) GetMetadata(ctx context.Context, in \*GetMetadataRequest, opts ...grpc.CallOption) (\*GetMetadataResponse, error) {

    out := new(GetMetadataResponse)

    err := c.cc.Invoke(ctx, "/MetadataService/GetMetadata", in, out, opts...)

    if err != nil {

        return nil, err

    }

    return out, nil

}

This generated code can be used in our applications to call our API from the Go applications. Additionally, we can generate such client code for other languages, such as Java, adding more arguments to the compiler command that we just executed. This is a great feature that can save us lots of time when writing microservice applications – instead of writing client logic for calling our services, we can use the generated clients and plug them into our applications.

In addition to the client code, the Protocol Buffers compiler also generates the service code that can be used for handling the requests. In the same movie\_grpc.pb.go file, you will find the following:

type MetadataServiceServer interface {

    GetMetadata(context.Context, \*GetMetadataRequest) (\*GetMetadataResponse, error)

    mustEmbedUnimplementedMetadataServiceServer()

}

func RegisterMetadataServiceServer(s grpc.ServiceRegistrar, srv MetadataServiceServer) {

    s.RegisterService(&MetadataService\_ServiceDesc, srv)

}

We are going to use both the client and server code that we just saw in our application. In the next section, we are going to modify our API handlers to use the generated code and handle requests using the Protocol Buffers format.

Implementing gateways and clients

In this section, we are going to illustrate how to plug the generated client and server gRPC code into our microservices. This will help us to switch communication between them from JSON-serialized HTTP to Protocol Buffers gRPC calls.

Metadata service

In Chapter 2, we created our internal model structures, such as metadata, and in Chapter 4, we created their Protocol Buffers counterparts. Then, we generated the code for our Protocol Buffers definitions. As a result, we have two versions of our model structures – internal ones, as defined in metadata/pkg/model, and the generated ones, which are located in the gen package.

You might think that having two similar structures is now redundant. While there is certainly some level of redundancy in having such duplicate definitions, these structures practically serve different purposes:

* Internal model: The structures that you create manually for your application should be used across its code base, such as the repository, controller, and other logic.
* Generated model: Structures generated by tools such as the protoc compiler, which we used in the last two chapters, should only be used for serialization. The use cases include transferring the data between the services or storing the serialized data.

You might be curious why it's not recommended to use the generated structures across the application code base. There are multiple reasons for this, which are listed as follows:

* Unnecessary coupling between the application and serialization format: If you ever want to switch from one serialization format to another (for example, from Thrift to Protocol Buffers), and all your application code base uses generated structures for the previous serialization format, you would need to rewrite not only the serialization code but the entire application.
* Generated code structure could vary between different versions: While the field naming and high-level structure of the generated structures are generally stable between different versions of code generation tooling, the internal functions and structure of the generated code could vary from version to version. If any part of your application uses some generated functions that get changed, your application could break unexpectedly during a version update of a code generator.
* Generated code is often harder to use: In formats such as Protocol Buffers, all fields are always optional. In generated code, this results in lots of fields that can have nil values. For an application developer, this means doing more nil checks across all applications to prevent possible panics.

Because of these reasons, the best practice is to keep both internal structures and the generated ones and only use the generated structures for serialization. Let's illustrate how to achieve this.

We would need to add some mapping logic to translate the internal data structures and their generated counterparts. In the metadata/pkg/model directory, create a mapper.go file and add the following to it:

package model

import (

    "movieexample.com/gen"

)

// MetadataToProto converts a Metadata struct into a

// generated proto counterpart.

func MetadataToProto(m \*Metadata) \*gen.Metadata {

    return &gen.Metadata{

        Id:          m.ID,

        Title:       m.Title,

        Description: m.Description,

        Director:    m.Director,

    }

}

// MetadataFromProto converts a generated proto counterpart

// into a Metadata struct.

func MetadataFromProto(m \*gen.Metadata) \*Metadata {

    return &Metadata{

        ID:          m.Id,

        Title:       m.Title,

        Description: m.Description,

        Director:    m.Director,

    }

}

The code we just added transforms the internal model into the generated structures and back. In the following code block, we are going to use it in the server code.

Now, let's implement a gRPC handler for the metadata service that would handle the client requests to the service. In the metadata/internal/handler package, create a grpc directory and add a grpc.go file:

package grpc

import (

    "context"

    "errors"

    "google.golang.org/grpc/codes"

    "google.golang.org/grpc/status"

    "movieexample.com/gen"

    "movieexample.com/metadata/internal/controller"

    "movieexample.com/metadata/internal/repository"

    "movieexample.com/metadata/pkg/model"

)

// Handler defines a movie metadata gRPC handler.

type Handler struct {

    gen.UnimplementedMetadataServiceServer

    svc \*controller.MetadataService

}

// New creates a new movie metadata gRPC handler.

func New(ctrl \*metadata.Controller) \*Handler {

    return &Handler{svc: ctrl}

}

Let's implement the GetMetadataByID function:

// GetMetadataByID returns movie metadata by id.

func (h \*Handler) GetMetadata(ctx context.Context, req \*gen.GetMetadataRequest) (\*gen.GetMetadataResponse, error) {

    if req == nil || req.MovieId == "" {

        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty id")

    }

    m, err := h.svc.Get(ctx, req.MovieId)

    if err != nil && errors.Is(err, controller.ErrNotFound) {

        return nil, status.Errorf(codes.NotFound, err.Error())

    } else if err != nil {

        return nil, status.Errorf(codes.Internal, err.Error())

    }

    return &gen.GetMetadataResponse{Metadata: model.MetadataToProto(m)}, nil

}

Let's highlight some parts of this implementation:

* The handler embeds the generated gen.UnimplementedMetadataServiceServer structure. This is required by a Protocol Buffers compiler to enforce future compatibility.
* Our handler implements the GetMetadataByID function in exactly the same format as defined in the generated MetadataServiceServer interface.
* We are using the MetadataToProto mapping function to transform our internal structures into the generated ones.

Now we are ready to update our main file and switch it to the gRPC handler. Update the metadata/cmd/main.go file, changing its contents to the following:

package main

import (

"context"

"flag"

"fmt"

"log"

"net"

"time"

    "google.golang.org/grpc"

    "movieexample.com/gen"

    "movieexample.com/metadata/internal/controller"

    grpchandler "movieexample.com/metadata/internal/handler/grpc"

    "movieexample.com/metadata/internal/repository/memory"

    "movieexample.com/metadata/pkg/model"

)

func main() {

    log.Println("Starting the movie metadata service")

    repo := memory.New()

    svc := controller.New(repo)

    h := grpchandler.New(svc)

    lis, err := net.Listen("tcp", "localhost:8081")

    if err != nil {

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

    }

    srv := grpc.NewServer()

    gen.RegisterMetadataServiceServer(srv, h)

    if err := srv.Serve(lis); err != nil {

panic(err)

}

}

The updated main function illustrates how we instantiate our gRPC server and start listening for requests in it. The rest of the function is similar to the one we had before.

We are done with the changes to the metadata service and can now proceed to the rating service.

Rating service

Let's create a gRPC handler for the rating service. In the rating/internal/handler package, create a grpc directory and add a grpc.go file with the following code:

package grpc

import (

    "context"

    "errors"

    "google.golang.org/grpc/codes"

    "google.golang.org/grpc/status"

    "movieexample.com/gen"

    "movieexample.com/rating/internal/controller"

    "movieexample.com/rating/pkg/model"

)

// Handler defines a gRPC rating API handler.

type Handler struct {

    gen.UnimplementedRatingServiceServer

    svc \*controller.RatingService

}

// New creates a new rating gRPC handler.

func New(svc \*controller.RatingService) \*Handler {

    return &Handler{svc: ctrl}

}

Now, let's implement the GetAggregatedRating endpoint:

// GetAggregatedRating returns the aggregated rating for a

// record.

func (h \*Handler) GetAggregatedRating(ctx context.Context, req \*gen.GetAggregatedRatingRequest) (\*gen.GetAggregatedRatingResponse, error) {

    if req == nil || req.RecordId == "" || req.RecordType == "" {

        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty id/type")

    }

    v, err := h.svc.GetAggregatedRating(ctx, model.RecordID(req.RecordId), model.RecordType(req.RecordType))

    if err != nil && errors.Is(err, controller.ErrNotFound) {

        return nil, status.Errorf(codes.NotFound, err.Error())

    } else if err != nil {

        return nil, status.Errorf(codes.Internal, err.Error())

    }

    return &gen.GetAggregatedRatingResponse{RatingValue: v}, nil

}

Finally, let's implement the PutRating endpoint:

// PutRating writes a rating for a given record.

func (h \*Handler) PutRating(ctx context.Context, req \*gen.PutRatingRequest) (\*gen.PutRatingResponse, error) {

    if req == nil || req.RecordId == "" || req.UserId == "" {

        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty user id or record id")

    }

    if err := h.svc.PutRating(ctx, model.RecordID(req.RecordId), model.RecordType(req.RecordType), &model.Rating{UserID: model.UserID(req.UserId), Value: model.RatingValue(req.RatingValue)}); err != nil {

        return nil, status.Errorf(codes.Internal, err.Error())

    }

    return &gen.PutRatingResponse{}, nil

}

Now, we are ready to update our rating/cmd/main.go file. Replace it with the following:

package main

import (

    "log"

    "net"

    "google.golang.org/grpc"

    "movieexample.com/gen"

    "movieexample.com/rating/internal/controller"

    grpchandler "movieexample.com/rating/internal/handler/grpc"

    "movieexample.com/rating/internal/repository/memory"

)

func main() {

    log.Println("Starting the rating service")

    repo := memory.New()

    svc := controller.New(repo)

    h := grpchandler.New(svc)

    lis, err := net.Listen("tcp", "localhost:8082")

    if err != nil {

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

    }

    srv := grpc.NewServer()

    gen.RegisterRatingServiceServer(srv, h)

    if err := srv.Serve(lis); err != nil {

panic(err)

}

}

The way we start the service is similar to the metadata service. Now, we are ready to link the movie service to both the metadata and rating services.

Movie service

In the previous examples, we created gRPC servers to handle client requests. Now, let's illustrate how to add logic for calling our servers. This will help us to establish communication between our microservices via gRPC.

First, let's implement a function that we can reuse in our service gateways. Create the src/internal/grpcutil directory, and add a file called grpcutil.go to it. Add the following code to it:

package grpcutil

import (

    "context"

    "math/rand"

    "pkg/discovery"

    "google.golang.org/grpc"

    "google.golang.org/grpc/credentials/insecure"

    "movieexample.com/pkg/discovery"

)

// ServiceConnection attempts to select a random service

// instance and returns a gRPC connection to it.

func ServiceConnection(ctx context.Context, serviceName string, registry discovery.Registry) (\*grpc.ClientConn, error) {

    addrs, err := registry.ServiceAddresses(ctx, serviceName)

    if err != nil {

        return nil, err

    }

    return grpc.Dial(addrs[rand.Intn(len(addrs))], grpc.WithTransportCredentials(insecure.NewCredentials()))

}

The function that we just implemented will try to pick a random instance of the target service using the provided service registry, and then it will create a gRPC connection for it.

Now, let's create a gateway for our metadata service. In the movie/internal/gateway package, create a directory called metadata. Inside it, create a grpc directory with a metadata.go file, containing the following code:

package grpc

import (

    "context"

    "google.golang.org/grpc"

    "movieexample.com/gen"

    "movieexample.com/internal/grpcutil"

    "movieexample.com/metadata/pkg/model"

    "movieexample.com/pkg/discovery"

)

// Gateway defines a movie metadata gRPC gateway.

type Gateway struct {

    registry discovery.Registry

}

// New creates a new gRPC gateway for a movie metadata

// service.

func New(registry discovery.Registry) \*Gateway {

    return &Gateway{registry}

}

Let's implement the function for getting the metadata from a remote gRPC service:

// Get returns movie metadata by a movie id.

func (g \*Gateway) Get(ctx context.Context, id string) (\*model.Metadata, error) {

    conn, err := grpcutil.ServiceConnection(ctx, "metadata", g.registry)

    if err != nil {

        return nil, err

    }

    defer conn.Close()

    client := gen.NewMetadataServiceClient(conn)

    resp, err := client.GetMetadataByID(ctx, &gen.GetMetadataByIDRequest{MovieId: id})

    if err != nil {

        return nil, err

    }

    return model.MetadataFromProto(resp.Metadata), nil

}

Let's highlight some details of our gateway implementation:

* We use the grpcutil.ServiceConnection function to create a connection to our metadata service.
* We create a client using the generated client code from the gen package.
* We use the MetadataFromProto mapping function to convert the generated structures into internal ones.

Now we are ready to create a gateway for our rating service. Inside the movie/internal/gateway package, create a rating/grpc directory and add a grpc.go file with the following contents:

package grpc

import (

    "context"

    "pkg/discovery"

    "rating/pkg/model"

    "google.golang.org/grpc"

    "movieexample.com/internal/grpcutil"

    "movieexample.com/gen"

)

// Gateway defines an gRPC gateway for a rating service.

type Gateway struct {

    registry discovery.Registry

}

// New creates a new gRPC gateway for a rating service.

func New(registry discovery.Registry) \*Gateway {

    return &Gateway{registry}

}

Add the implementation of the GetAggregatedRating function:

// GetAggregatedRating returns the aggregated rating for a

// record or ErrNotFound if there are no ratings for it.

func (g \*Gateway) GetAggregatedRating(ctx context.Context, recordID model.RecordID, recordType model.RecordType) (float64, error) {

    conn, err := grpcutil.ServiceConnection(ctx, "rating", g.registry)

    if err != nil {

        return 0, err

    }

    defer conn.Close()

    client := gen.NewRatingServiceClient(conn)

    resp, err := client.GetAggregatedRating(ctx, &gen.GetAggregatedRatingRequest{RecordId: string(recordID), RecordType: string(recordType)})

    if err != nil {

        return 0, err

    }

    return resp.RatingValue, nil

}

At this point, we are almost done with the changes. The last step is to update the main function of the movie service. Change it to the following:

package main

import (

    "context"

    "log"

    "net"

    "google.golang.org/grpc"

    "movieexample.com/gen"

    "movieexample.com/movie/internal/controller"

    metadatagateway "movieexample.com/movie/internal/gateway/metadata/grpc"

    ratinggateway "movieexample.com/movie/internal/gateway/rating/grpc"

    grpchandler "movieexample.com/movie/internal/handler/grpc"

"movieexample.com/pkg/discovery/static"

)

func main() {

    log.Println("Starting the movie service")

    registry := static.NewRegistry(map[string][]string{

        "metadata": {"localhost:8081"},

        "rating":   {"localhost:8082"},

        "movie":    {"localhost:8083"},

    })

    ctx := context.Background()

    if err := registry.Register(ctx, "movie", "localhost:8083"); err != nil {

        panic(err)

    }

    defer registry.Deregister(ctx, "movie")

    metadataGateway := metadatagateway.New(registry)

    ratingGateway := ratinggateway.New(registry)

    svc := controller.New(ratingGateway, metadataGateway)

    h := grpchandler.New(svc)

    lis, err := net.Listen("tcp", "localhost:8083")

    if err != nil {

        panic(err)

    }

    srv := grpc.NewServer()

    gen.RegisterMovieServiceServer(srv, h)

    if err := srv.Serve(lis); err != nil {

panic(err)

}

}

You might have noticed that the format hasn't changed, and we just updated the imports for our gateways, changing them from HTTP to gRPC.

We are done with the changes to our services. Now the services can communicate with each other using the Protocol Buffers serialization, and you can run them using the go run \*.go command inside each cmd directory. You can make a test gRPC request to one of our services by using a grpcurl tool:

grpcurl -plaintext -d '{"record\_id":"1", "record\_type":

"movie"}' localhost:8082 RatingService/GetAggregatedRating

You will get an error indicating that there is no data for the provided record, however this is currently expected since we don’t store any data yet — we will do this soon in Chapter 7, Persistence and Databases.

We have covered the main aspects of synchronous communication between microservices. Now, let’s review some of the synchronous communication best practices.

Synchronous communication best practices

Below if the list of synchronous communication best practices that will help you to improve maintainability and reliability of your synchronous communication, as well as clearly communicate any errors to service callers.

Perform excessive request validation and return correct error codes

Earlier in this chapter, we mentioned that there are two basic types of errors during synchronous communication: client and server errors. All communication protocols, including HTTP, gRPC and Thrift, offer support for differentiating between different error types in client and server code. For example, in our metadata server code we used the following code to return different types of errors to our callers:

func (h \*Handler) GetMetadata(ctx context.Context, req \*gen.GetMetadataRequest) (\*gen.GetMetadataResponse, error) {

    if req == nil || req.MovieId == "" {

        return nil, status.Errorf(codes.InvalidArgument, "nil req or empty id")

    }

    m, err := h.svc.Get(ctx, req.MovieId)

    if err != nil && errors.Is(err, controller.ErrNotFound) {

        return nil, status.Errorf(codes.NotFound, err.Error())

    } else if err != nil {

        return nil, status.Errorf(codes.Internal, err.Error())

    }

    return &gen.GetMetadataResponse{Metadata: model.MetadataToProto(m)}, nil

}

Our code returns three different types of error codes:

1. codes.InvalidArgument: Client error indicating incorrect request data.
2. codes.NotFound: Client error indicating that the requested data is not found.
3. codes.Internal: Server error indicating some internal error while handling the request.

Each error code provides a hint to our clients on what exactly went wrong during the request processing. For example, our a client might retry a request when a server returns the codes.Internal error code, while the InvalidArgument error would indicate that the problem is likely on the side of our client and requires a change in the client code before the request can be processed successfully.

Leveraging specific error codes will help to investigate issues more easily, as well as to add various automations to your code (for example, automatic retries of requests in case of internal server errors, that we will implement in Chapter 10). Additionally, excessive request validation in your server code will help to avoid lots of possible issues, included code panics (for example, when some request parameter has a nil value).

Ensure idempotency

Idempotency is one of the critical concepts in synchronous communication and API design. In simple terms, idempotency means that some operation, such a network request, can be handled multiple times without any unintended side effects. Let’s assume we have a system where a server processes payment requests from one of its clients. Now assume that the client just submitted a payment request, but immediately after this the client’s network connection becomes unavailable. If server guarantees idempotency, the client can safely retry the same request without worrying about getting charged twice for the same payment.

How to achieve idempotency? There are some common techniques that might be helpful for this:

* Assign unique identifiers to requests: In our payment example, a server might require the client to provide an extra argument (for example, transaction\_id) that would act as a unique identifier of the request. Such identifiers are often called idempotency keys. Then, a server would guarantee that requests having the same idempotency key would be idempotent, hence safe to retry on the client side.
* Design APIs with idempotency in mind: When implementing service APIs, follow some basic rules for achieving idempotency. First, read operations (for example, GetMetadata requests) should not perform extra modifications of the underlying data. Second, various update operations (for example, HTTP PUT and DELETE calls) should also produce same results without causing any unexpected effects. This article provides a good summary on the importance of building idepmpotent APIs: https://stripe.com/blog/idempotency

Also be explicit about idempotency and all aspects of request handling in your API documentation. Your service clients should not be manually testing how the system would behave in various error cases: instead, you should provide clear description on how each endpoint works and what are expected inputs and outputs are, as well as which operations are idempotent and how such idempotency is achieved. We are going to cover this more in Chapter 10, Documentation.

This brief section summarizes our best practices on synchronous communication. You might find some other useful tips in the Further Reading section of this chapter.

Summary

In this chapter, we covered the basics of synchronous communication and learned how to make microservices communicate with each other using the Protocol Buffers serialization format and gRPC communication framework. We illustrated how to define our service APIs using the Protocol Buffers schema language and generate code that can be reused in microservice applications written in Go and other languages. Then, we implemented gRPC client and server components, and discussed some of the best practices of establishing synchronous communication.

The knowledge you gained in this chapter should help you write and maintain the existing services using Protocol Buffers and gRPC, as well as serve as an example of how to use code generation for your services.

In the next chapter, we are going to continue our journey into different ways of communication by covering another model, asynchronous communication.

Further reading

* gRPC: https://grpc.io
* HTTP/2 detailed overview: <https://web.dev/performance-http2>
* Idempotent REST APIs: https://restfulapi.net/idempotent-rest-apis/