Report on Ori Industries Assignment

Yiorgos Marinellis

January 22, 2020

Abstract

This text will try to present a proposed solution to the assignment by analysing the procedure that was followed. Throughout the text several assumptions that led to specific decisions will be described. In the end a basic evaluation of some trade-offs will be presented.

Having no professional experience in Go, I did my best to follow best development practices, but I still can argue that the efficiency of the code is not in it's best.

1 Purpose

The scope of this exercise is to create a basic service (defined as microservice) which exposes a couple of methods through a gRPC interface.

The development must be test driven and include an integration with a Continuous Integration/Continuous Delivery platform and a basic Command Line client application in order to access the methods.

Kubernetes deployment files in order to expose the service functionality must as well be created.

2 Cloud native

By definition from Cloud Native Computing Foundation) cloud native computing uses an open source software stack and is

- Containerized
- Dynamically orchestrated
- Microservice Oriented

2.1 Containerization

To facilitate reproducibility and resource isolation a container solution must be used. Docker was selected for this role which nowadays is the top trend in containerization. The main benefit of this approach is that the application becomes independent of the environment and that the container is highly portable.

2.2 Container Orchestration

Containers are actively scheduled and managed to optimize resource utilization using Kubernetes. It provides load balancing (with Services) between the active application instances (replicas on a Deployment). With ConfigMap and Secrets can share configuration and authentication secrets between your containers. Deployment to a K8S cluster can easily be done using the k8s cli inteface, `kubectl`. Deploying is fast, clean and easy. To my preference for more robust deployments I prefer to use Helm, currently not used for the scope of this exercise. Summarizing, Kubernetes implements all set of DevOpts for our application.

2.3 Microservices Architecture

Microservices Design Patterns implies a system of multiple, relatively small applications. They work together to provide the overall functionality of your system. They have well defined boundaries and can autonomously developed by small teams. They provide simplicity over functionality and easy horizontal scaling.

Microservices remove some complexity from the services themselves but they define a distributed system with more complexity. The distributed nature of the system also makes it a lot harder to monitor and manage taking into account that for each service there might be several instances that run in parallel.

3 Communication Protocol

The protocol `gRPC` is an implementation for the Remote Method Invocation schema. Such schemas are used in distributed systems when a computer program wants to execute a function in a remote address space. The service exposes a skeleton for the methods defined to be called remotely with their parameters and return types. The client creates a stubs in order to access the methods on its side. The stub performs type checking and marshalls the called type and input parameters into a request message. It then sends the message over the network.

3.1 Data exchange

By default gRPC uses `Protocol Buffers` as serialization mechanism. Data format is defined in `.proto` files. Protocol buffer data is structured as messages. The compilation of `.proto` files creates the necessary classes for our data, to be used from the applications we develop. Moreover, in gRPC exposed methods are as well defined in the `.proto` files with their parameters and return types.

Below is the `.proto` file I used to define message payloads and 3 methods that are exposed from the microservice.

```
syntax = "proto3";

package pb;

// Straight-forward declaration of payloads
message Value {
     int64 v = 1;
}

message Point {
     int64 x = 1;
     int64 y = 2;
}

message Ret {
     int64 ret = 1;
    string msg = 2;
}

message RetSummary {
    repeated int64 ret = 1;
     int64 totalPoints = 2;
    string msg = 3;
}

service OriService {

    // Simple RPC
    //
    // Receives a Point and returns the sum of its elements
    //
    // A sum of zero is returned when an empty Point is provided
    rpc Sum(Point) returns (Ret) {}

    // Client-to-server streaming RPC
    //
    // Receives a stream of Points and returns a summary with the GCD
    // for each point. Empty Point returns 0.
    rpc Gcd(stream Point) returns (RetSummary) {}

    // Server-to-client streaming RPC
    //
    // Receives value and streams back three-times the  SQRT of its
    rpc Sqrt(Value) returns (stream Ret) {}

}
```

4 12factor app

4.1 Codebase

The source code for the assignment is hosted on a repository Github. I usually prefer the monorepo approach or the use of multiple repositories with submodules. On the case both server and client code are located in the same repo but they are treated as to individuals applications with separate Dockerfiles.

The CI/CD file provides the commands stages to create multiple deploys for the application.

4.2 Dependencies

The dependency handling in go is ,as I consider, still an open problem. I decided to go on with `dep` as it provides explicit dependency/version declaration and is a project with a big community in Github. Other solutions that I came by was `Glide` which was simple in usage but late in resolving the dependency tree and `Go Mod` which seem to be as well as dep a really interesting project.

4.3 Config

Application environment to store configuration values between deploys. For example environment variable `BIND_PORT` was used to store the port where server listens to and `SERVER_ADDRESS` defines the address where the server is accessible from the client.

4.4 Backing services

For the specific assignment no backing services (like a database, or a message passing system) were used. In any case these applications can be deployed with standalone Kubernetes deployments and accesses as services within the cluster. Otherwise, they can be used as SaaS cloud applications and treated as 3rd party backing services.

4.5 Build, release, run

For the scope of the application no release was made. According to Gitflow implementation, releases are sepate branches from the master (production branch). Gitlab CI manages releases using tags.

4.6 Processes

Both applications but mainly the server (microservice) does not hold any internal state. It runs statelessly, calculating and return values. No write or reads to the filesystem happen that can change the process's state. The filesystem must only be used as a cache.

4.7 Port binding

The gRPC service is accessible over the network on a port defined by the deployer. On a Kubernetes cluster this port refers to a Pod's port. A Service has as well to be created either for exposing within the cluster (ClusterIP) or outside (NodePort, LoadBalancer on cloud providers) which will bind the Pod's port to a port specified from the deployer as well.

4.8 Concurrency

Golang has a pretty well defined concurrency model using lightweight goroutines. The gRPC implementation defines that each RPC handler attached to a registered server will be invoked in its own goroutine. The same is true for service handlers for streaming RPCs. On the current implementation I used `errgroup` which provides a nice error propagation method and context cancellation for groups of goroutines. This will furthermore enables us the functionality to run other services (like an HTTP server, eventsource) on the same app.

4.9 Disposability

The server handles SIGTERM by canceling the current context and waiting until all function calls from the Go method have returned before going to shutdown. Unexpected hardware failures are not handled.

4.10 Dev/prod parity

The scope of this point is to keep development and production branches on a small gap. This can be achieved firstly by keeping the master and development repository branches as close as it can be and secondly by creating a common deployment mechanism for both production and development app. Assuming that we are deploying prod on a K8S cluster, we can

- deploy development on another K8S cluster keep as much as we can the same configuration
- deploy locally the MS app either by using docker-compose or Tilt

Before committing or creating a Merge Request or deploying all kind of tests (unit, component end-to-end) should be successful on the development branch.

4.11 Logs

Logs are currently forwarded to stdout and can be accessed by kubectl logs. On this stage the log level is statically defined to INFO but it can be easily configured as an environment variable as well. Moreover, log streams can be piped to a database like Elasticsearch in order to perform operations on them ,like aggregations, or provide visualizations (Kibana).

4.12 Admin processes

The straight forward declaration of the application didn't imply any administration tasks. Similar tasks ,like horizontal scaling, can be done on the cluster level using kubectl tool

5 Eventsource

We can run the http server handling an eventsource function on a goroutine (defined on the same errgroup). We can then use streams to push events to the eventsource server from another goroute (eg the gRPC goroutine). A client can then subscribe to the http enpoint in order to receive the events.

6 Monitoring

A monitoring system for the cluster is usually provided (or can be easily installed) from the Cloud Provider. Moreover every app can expose its metrics and store them to TSDB. On top of this setup an alert system and a visualization tool are needed. I personally favor `Prometheus`, `Alert Manager` and `Graphana`.

7 Functionality Access

The easyest way to access the service from outside the cluster would be to simply expose the Service with a NodePort of LoadBalancer from the cloud provider. In this case the SSL/TLS functionality (or an authentication mechanism) is necessary.

Another solution would be to create a service which works as a client for the gRPC server an exposes an HTTP REST API for the request/response. I would describe it as a translation proxy. The HTTP service can then be accessed with an Ingress.

The MS itself can as well run a web server on a separate goroutine to expose the functionality and gRPC only for internal consumption.

8 Conclusion

8.1 Notes

Some important notes.

- The directory layout used is not alligned with `go standards for package`.
- Their is no integration with a cloud provider
- The logging level of the applications is statically defined as Info
- No use of GoLint was used.
- No use of magefile
- Unittests for Gcd and Srqt server functions were not implemented
- Mock server only mocks Sum and Gcd
- Gitlab-Ci and Kubernetes files have not be tested on a real system
- Client was developed in quick 'n' dirty approach

8.2 Improvements

- Usage of Go Mod
- Better organization of code in packages and modules
- Write more unittests, an end-to-end test or even stress tests
- Use Helm
- Add ConfigMaps to pass configuration values to Kubernetes Pods
- Add SSL functionality to gRPC server
- Try to impove builds by compiling only the files that have changed
- Create a more generic and robust client using cobra