PYOROALB

Geste de savoir

gRPC

22 décembre 2022

Table des matières

1. gRPC . . . 1
   - 1.1. gRPC . . . 1
   - 1.2. RPC . . . 1
   - 1.3. HTTP/2 . . . 2
   - 1.4. Protobuf . . . 2
2. Un retour en arrière? . . . 2
3. Des outils pour travailler avec gRPC . . . 2
   - 3.1. Des lanceurs de requettes . . . 3
   - 3.2. Dans le navigateur . . . 3
   - 3.3. Garder du http . . . 3
   - 3.4. Dans les IDE . . . 3
4. Un projet avec gRPC . . . 3
   - 4.1. Le serveur . . . 4
   - 4.2. Un client . . . 6
5. Conclusions . . . 8

Contenu masqué . . . 8

1. gRPC

1.1. gRPC

gRPC est un framework RPC de 2016. Son but est de connecter des services.

Il possède de nombreux avantages tels que le streaming entre le client et le serveur. Qu'il soit bidirectionnel, côté serveur, ou côté client.

Ce framework est aussi multilangage . Il supporte la plupart des langages populaires aujourd'hui.
En plus d'être multilangage, il est aussi disponible pour de multiples plateformes. Il est disponible pour les applications backend, web et Android. Il sera aussi bientôt disponible pour Flutter et iOS.

gRPC est open source (le dépôt est ici). Il est initialement développé par google.

1.2. RPC

RPC signifie *Remote Procedure Call*. C'est un protocole requête → réponse (comme http) qui permet la délocalisation de l'exécution du code.

Il existe beaucoup de variations d'implémentation du protocole RPC. Parfois, ces implémentations ne sont pas compatibles entre elles.

1.3. HTTP/2

gRPC utilise HTTP/2 comme protocole de transport.

HTTP/2 est une version améliorer du protocole HTTP. C'est un standard du web qui permet de nouvelles fonctionnalités et comporte des améliorations telles que le *server push*.

1.4. Protobuf

Pour séréaliser les données à échanger, gRPC utilise par défaut Protocol Buffers .
C'est un format de sérialisation de donnée comme l'est le JSON, le XML, le YAML, … Protobuf est cependant un format binaire. C'est à dire que contrairement au JSON on ne peut pas le lire directement (nous devons passer par un programme).

Protobuf est compatible avec la plupart des langages populaires aujourd'hui.

Tout comme gRPC, protobuf est initialement développé par google et open source (le dépot est ici)

2. Un retour en arrière ?

Ici nous avons un nouveau protocole avec un nouveau format de sérialisation. Cela peut faire penser aux anciens formats de données et protocoles propriétaire qui pouvaient être compatible avec seulement un langage de programmation (ici on parle de ce qui pouvait exister avant le JSON ou le XML pour les échanges entre services).

gRPC n'est pas un retour en arrière sur ce niveau-là.
Il n'est pas propriétaire, il est ouvert et son implémentation est multi langage. Faire un projet avec gRPC ne contraint pas tous vos services à utiliser un langage de programmation.
De plus, gRPC est basé sur HTTP/2 qui est un standard du web.
Le framework permet aussi de choisir le format de sérialisation pour les échanges de données.

3. Des outils pour travailler avec gRPC

Bien que gRPC soit récent, il existe déjà de multiples outils pour travailler avec ce framework et protobuf.

3.1. Des lanceurs de requettes

Dans les logicielles pour envoyer des requêtes il y a Postman qui est connu notamment pour faire des requêtes http. Il y a aussi Kreya et Insomnia . Kreya est conçu avec pour premier but d'envoyer des requêtes gRPC. Insomnia et plus simpliste d'utilisation pour des petits projets.

3.2. Dans le navigateur

Il existe aussi plusieurs extensions de navigateur pour observer le comportement d'une application web qui utilise gRPC.

Des décodeurs sont disponibles en ligne pour convertir les données binaires protobuf en une représentation que nous, humains, pouvons lire.

3.3. Garder du http

Il existe aussi des projets comme gRPC-gateway qui permette très facilement de mettre à disposition un reverse-proxi qui va traduire les appels HTTP et appel gRPC.
Cela permet de ne pas avoir un serveur http **et** un serveur gRPC à maintenir.

3.4. Dans les IDE

Afin d'aider les développeurs, il existe aussi de nombreuses extensions d'IDE pour gRPC et protobuf.

4. Un projet avec gRPC

Nous allons imaginer un projet simple avec gRPC. Pour cela nous allons crées un écho serveur (en Go) qui permettra aussi aux clients d'espionner ce que les gens lui disent.

Dans un premier temps nous allons rédiger un fichier `.proto` qui définit notre service ainsi que les messages qui vont être utilisés.

```
1  // ./proto/spying_echo.proto
2
3  syntax = "proto3";
4
5  package spyingecho;
6
7  option go_package = "./spyingechopb";
8
9  service SpyingEcho {
10     // Echo respond the thing you say
```

```
11      rpc Echo (EchoRequest) returns (EchoReply) {}
12      // Spy send all things that are say
13      rpc Spy (Empty) returns (stream EchoReply) {}
14  }
15
16  // EchoRequest regroup an used and his message
17  message EchoRequest {
18      string Name = 1;
19      string Msg = 2;
20  }
21
22  // EchoReply is the response of an user's request
23  message EchoReply {
24      string Msg = 1;
25  }
26
27  message Empty {}
```

Maintenant, nous allons utiliser la commande `protoc` qui permet de générer du code que nous allons utiliser pour notre serveur.
Je vais mettre la commande dans un makefile car elle comporte de multiples options.

```
1  # ./Makefile
2
3  .PHONY: grpc-go server server-run client client-run
4
5  grpc-go:
6          protoc --go_out=./grpc --go_opt=paths=import \
7                  --go-grpc_out=./grpc --go-grpc_opt=paths=import \
8                  ./proto/spying_echo.proto
```

Nous ne voyons pas l'installation de l'outil `protoc` ici. C'est volontaire, le site officiel vous expliquera en détaillé tout ce qu'il y a à installer.

A présent, la commande `make grpc-go` va nous générer des fichiers dans le répertoire `./grpc/spyingechopb/`. Il n'est pas nécessaire d'aller voir le code qui a été généré, cependant il est très intéressant afin de comprendre ce que ça faire ce code pour nous.

4.1. Le serveur

Maintenant, nous pouvons nous concentrer sur l'implémentation de notre serveur.

Nous allons créés une structure qui représentera notre serveur. Elle implémentera l'interface qui a été générée à partir de notre fichier `.proto`.

```go
// SypingEchoServer implements the  spyingechopb.SpyingechoServer interface
type SpyingechoServer struct {
        spyingechopb.UnimplementedSpyingEchoServer

        // spies regroup all spies that we'll send messages to
        spies Spies

        // ers is a channel where we put all messages to send to spys
        ers chan *spyingechopb.EchoRequest
}
```

Une fois notre structure définie, je crée une fonction qui va retourner un *serveur* initialiser et qui dispatchera les messages aux espions connectés.

```go
func NewSpyingechoServer() *SpyingechoServer {
        server := &SpyingechoServer{
                spies: newSpies(),
                ers:   make(chan *spyingechopb.EchoRequest),
        }

        go server.dispatch()
        return server
}
```

Il ne nous reste plusqu'à implémenté les routes rpc `Echo` et `Spy`.

```go
func (server *SpyingechoServer) Echo(ctx context.Context, echoRequest *spyingechopb.EchoRequest) (*spyingechopb.EchoReply, error) {
        server.ers <- echoRequest // Save the message to send it to all spies

        log.Printf("%s said: %s", echoRequest.GetName(), echoRequest.GetMsg())

        return &spyingechopb.EchoReply{Msg: fmt.Sprintf("You said: %s", echoRequest.GetMsg())}, nil
}
```

Lorsque le serveur recevra un message, il le mettra dans une file pour l'envoyer aux expions avant de renvoyer qu client le texte qu'il à dit.

```go
func (server *SpyingechoServer) Spy(_ *spyingechopb.Empty, stream
    spyingechopb.SpyingEcho_SpyServer) error {
        // Add client to our spys list
        server.spies.Add(spy{stream: stream})
        log.Println("A new spy is connected")

        // Infinite loop to keep stream alive
        // Sending is manage by ses.dispatch method
        for {
                time.Sleep(time.Minute * 5)
        }
}
```

Quand un espion souhaite observer le serveur, nous l'ajoutons à notre liste d'espions. Une boucle infinie est en place afin de garder le stream toujours ouvert. Ce n'est pas cette fonction lit la file de message et qui envoie les messages aux espions. C'est la fonction suivante.

```go
func (server *SpyingechoServer) dispatch() {
        for msg := range server.ers {
                server.spies.Dispatch(msg)
        }
}
```

Contenu masqué n°1

Ici nous avons donc tout ce qu'il faut pour avoir notre serveur en action. Je ne suis volontairement pas allé plus loin dans le fonctionnement de l'application car on s'éloigne de gRPC.

4.2. Un client

Ici, nous implémenter un client qui espionnera le serveur.

```go
func main() {
        var opts []grpc.DialOption
        opts = append(opts, grpc.WithTransportCredentials(insecure
            .NewCredentials()))

        conn, err := grpc.Dial(fmt.Sprintf("%s:%d", HOST, PORT),
            opts...)
        if err != nil {
                log.Fatalf("fail to dial: %v", err)
        }
```

```
 9         defer conn.Close()
10
11         client := spyingechopb.NewSpyingEchoClient(conn)
12
13         spy(client)
14 }
```

Nous commençons par créer une connexion gRPC (lignes 2 à 9). Ensuite, nous créons notre client (ligne 11) et nous espionnons (ligne 13).

Voici l'implémentation de la fonction `spy`

```
 1 func spy(c spyingechopb.SpyingEchoClient) {
 2         ctx, cancel := context.WithCancel(context.Background())
 3         defer cancel()
 4
 5         stream, err := c.Spy(ctx, new(spyingechopb.Empty))
 6         if err != nil {
 7                 log.Fatal(err)
 8         }
 9
10         for {
11                 msg, err := stream.Recv()
12
13                 if err == io.EOF {
14                         break
15                 }
16                 if err != nil {
17                         log.Fatalf(
                               "error when receiving from stream: %s"
                               ,
                               err.Error())
18                 }
19
20                 fmt.Println(msg.GetMsg())
21         }
22 }
```

Nous créons un contexte que nous allons annuler à la sortie de la fonction.
On récupère notre stream en appelant la méthode `Spy` (nom de la route rpc définit dans le fichier `.proto`) sur le client.
Ensuite, avec notre stream, nous allons, dans une boucle infinie, récupérer ce que le serveur nous envoie pour l'afficher dans la console.

Contenu masqué n°2

On remarque qu'avec gRPC nous n'avons eu seulement à implémenter le comportement du serveur. Côté client, nous n'avons pas eu de sérialisation manuelle (conversion en JSON et gestion des possibles erreurs) à faire. De la même manière, en cas d'erreur réseau, nous récupérons directement une erreur dans le langage utilisé pour développer le projet.
De plus, je n'ai pas eu à recréer les mêmes structures de données dans le projet client et dans le projet serveur qui sont échangés par ces derniers. Je les utilise comme une dépendance depuis le code généré par `protoc`.

Le projet d'exemple complet se trouve ici .

5. Conclusions

gRPC présente de nombreux avantages. Un des principaux est qu'il permet une abstraction des appels réseaux et sur la sérialisation des objets qui sont échangés entre les services. Il permet aussi un usage très facile du streaming entre le client et le serveur.

De plus, gRPC est multilangage. L'utiliser de nous enferme pas dans un langage de programmation entre tous les services.

Ce framework est basé sur un standard du web. Les risques que le protocole soit abandonné d'un jour à l'autre est assez faible.
Il est aussi ouvert et conçu par google. En utilisant gRPC on profite donc du support de google et de la communauté qui y contribue.

Contenu masqué

Contenu masqué n°1

Voici donc le fichier complet.

```go
// ./server/spying_echo_server.go

package main

import (
        "context"
        "fmt"
        "log"
        "time"

        "github.com/albdewilde/spying_echo/grpc/spyingechopb"
)

```

Contenu masqué

```go
// SypingEchoServer implements the  spyingechopb.SpyingechoServer interface
type SpyingechoServer struct {
        spyingechopb.UnimplementedSpyingEchoServer

        // spies regroup all spies that we'll send messages to
        spies Spies

        // ers is a channel where we put all messages to send to spys
        ers chan *spyingechopb.EchoRequest
}

func NewSpyingechoServer() *SpyingechoServer {
        server := &SpyingechoServer{
                spies: newSpies(),
                ers:   make(chan *spyingechopb.EchoRequest),
        }

        go server.dispatch()
        return server
}

func (server *SpyingechoServer) Echo(ctx context.Context, echoRequest *spyingechopb.EchoRequest) (*spyingechopb.EchoReply, error) {
        server.ers <- echoRequest

        log.Printf("%s said: %s", echoRequest.GetName(), echoRequest.GetMsg())

        return &spyingechopb.EchoReply{Msg: fmt.Sprintf("You said: %s", echoRequest.GetMsg())}, nil
}

func (server *SpyingechoServer) Spy(_ *spyingechopb.Empty, stream spyingechopb.SpyingEcho_SpyServer) error {
        // Add client to our spys list
        server.spies.Add(spy{stream: stream})
        log.Println("A new spy is connected")

         // Infinite loop to keep stream alive
        // Sending is manage by ses.dispatch method
        for {
                time.Sleep(time.Minute * 5)
        }
}

func (server *SpyingechoServer) dispatch() {
        for msg := range server.ers {
```

```
57                  server.spies.Dispatch(msg)
58          }
59  }
```

Retourner au texte.

Contenu masqué n°2

Voici le fichier complet.

```
1   package main
2
3   import (
4           "context"
5           "fmt"
6           "io"
7           "log"
8
9           "github.com/albdewilde/spying_echo/grpc/spyingechopb"
10          "google.golang.org/grpc"
11          "google.golang.org/grpc/credentials/insecure"
12  )
13
14  const (
15          HOST = "0.0.0.0"
16          PORT = 10000
17  )
18
19  func main() {
20          var opts []grpc.DialOption
21          opts = append(opts, grpc.WithTransportCredentials(insecure
                .NewCredentials()))
22
23          conn, err := grpc.Dial(fmt.Sprintf("%s:%d", HOST, PORT),
                opts...)
24          if err != nil {
25                  log.Fatalf("fail to dial: %v", err)
26          }
27          defer conn.Close()
28
29          client := spyingechopb.NewSpyingEchoClient(conn)
30
31          spy(client)
32  }
33
34  func spy(c spyingechopb.SpyingEchoClient) {
35          ctx, cancel := context.WithCancel(context.Background())
```

Contenu masqué

```
36          defer cancel()
37
38          stream, err := c.Spy(ctx, new(spyingechopb.Empty))
39          if err != nil {
40                  log.Fatal(err)
41          }
42
43          for {
44                  msg, err := stream.Recv()
45
46                  if err == io.EOF {
47                          break
48                  }
49                  if err != nil {
50                          log.Fatalf(
                                "error when receiving from stream: %s"
                                ,
                                err.Error())
51                  }
52
53                  fmt.Println(msg.GetMsg())
54          }
55  }
```

Retourner au texte.