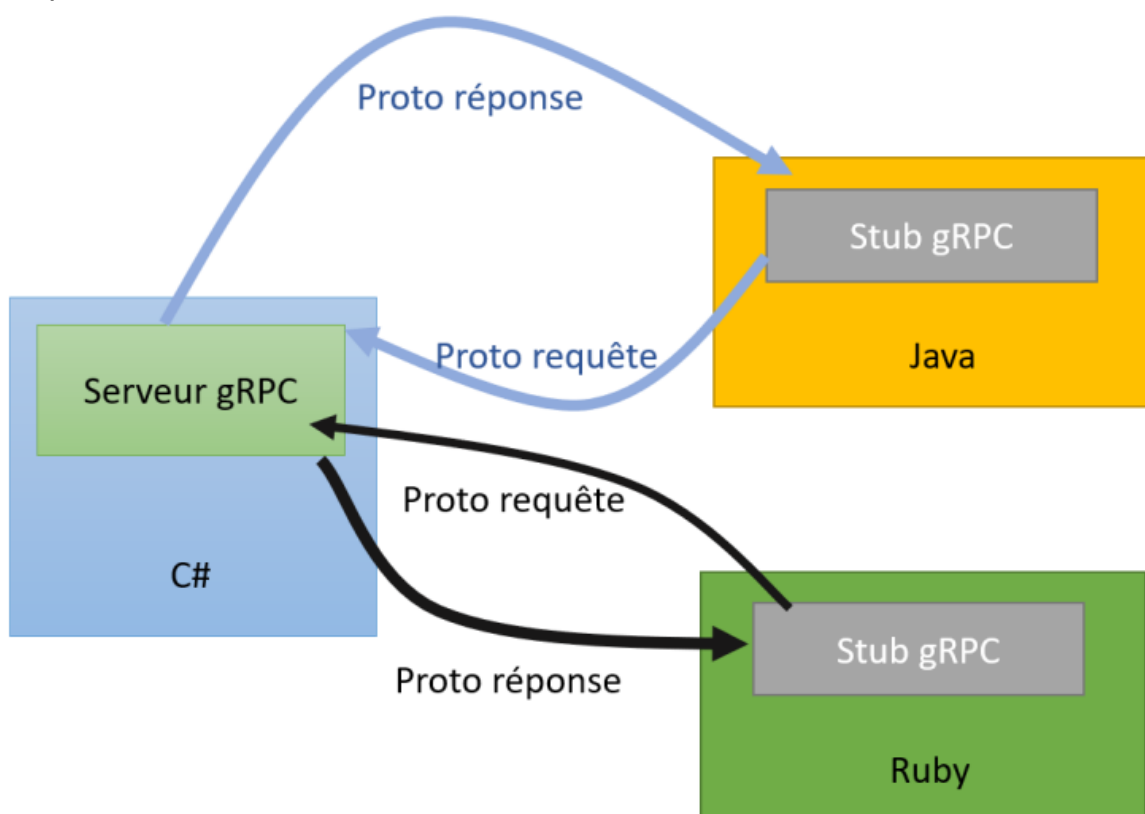


# GRPC SIMPLE SERVER

Sur ce projet, est utilisé le protocole réseau RPC (Remote Procedure Call).  
Ce protocole est utilisé dans le modèle client-serveur pour assurer la communication entre le client, le serveur et d'éventuels intermédiaires.  
Pour que celui-ci puisse être exploité, le DAM service utilise le framework GRPC.

Avec gRPC, on peut définir un service une seule fois dans un fichier **protobuf** .proto et générer des clients et des serveurs dans n'importe quel langage supporté par gRPC (dont Go), qui à leur tour peuvent être exécutés dans des environnements. Les interfaces sont décrites par un langage appelé **Protocol Buffer** qui sert à décrire les fichiers protobuf. Ce sont ces fichiers qui seront ensuite transtypés dans le langage voulu.

Exemple d'utilisation :



N'ayant aucune connaissance sur ces technologies, j'ai créé un mini-serveur dont le but est de représenter une simulation d'envoi d'un message afin de mieux comprendre le fonctionnement du framework.

## Mini-serveur GRPC - Serveur

Mon fichier protobuf appelé "Invoicer" est composé de plusieurs parties :

Un service qui définit 2 fonctions, qui seront implémentés dans mon serveur :

- Create
- GetInformations

N.B. : ces fonctions peuvent être appelées dans n'importe quel langage qui supporte GRPC

```
service Invoicer {  
    rpc Create(CreateRequest) returns (CreateResponse);  
    rpc GetInformations(CreateRequest) returns (RequestDest);  
}
```

On peut constater que ces fonctions ont un paramètre d'entrée et une valeur de retour.

La fonction Create prend en paramètre une requête "CreateRequest" et retourne une réponse "CreateResponse".

Tandis que la fonction GetInformations prend en paramètre une requête "CreateRequest" et retourne une réponse "RequestDest".

Ces requêtes et réponses sont initialisées avec un message en GRPC. Ce message est utilisé pour échanger des données entre le client et le serveur lors de l'appel d'une méthode spécifique du service.

```
message RequestDest {  
    Account from = 1;  
    Account to = 2;  
}  
  
message CreateRequest {  
    Account account = 1;  
    RequestDest request = 2;  
    Message message = 3;  
}  
  
message CreateResponse {  
    RequestDest request = 1;  
    Message message = 2;  
}
```

Ces messages se composent également d'autres messages :

- Account
- Message

```
message Account {  
    int64 id = 1;  
    string name = 2;  
}  
  
message Message {  
    string message = 1;  
}
```

Globalement, ces messages peuvent et vont être définis en structs/objets.  
Dans notre cas, lorsque nous appellerons nos fonctions plus tard, GRPC se chargera de convertir ces messages en struct/objet dans le langage utilisé (toujours Go dans notre cas).  
Ce qui pourrait donner cela en Go :

```
type Account struct {  
    id    int64  
    name string  
}  
  
type Message struct {  
    message string  
}  
  
type RequestDest struct {  
    from *Account  
    to   *Account  
}  
  
type CreateRequest struct {  
    account *Account  
    request *RequestDest  
    message *Message  
}  
  
type CreateResponse struct {  
    request *RequestDest  
    message *Message  
}
```

Maintenant que notre fichier protobuf est écrit, il faut générer le code GRPC spécifique à Go afin de pouvoir utiliser les fonctions instanciées dans le service :

```
protoc --proto_path=. --go_out=. --go-grpc_out=.  
--go-grpc_opt=require_unimplemented_servers=false *.proto
```

Cette commande nous génère 2 fichiers qui serviront au serveur et au client afin d'appeler les fonctions nécessaires.

Maintenant, on écrit un serveur simple en Go :

Tout d'abord, on implémente les fonctions définies dans notre fichier protobuf :

```
func (s myInvoicerServer) Create(  
    ctx context.Context,  
    req *invoicer.CreateRequest) (*invoicer.CreateResponse, error) {  
    response := &invoicer.CreateResponse{  
        Request: req.Request,  
        Message: req.Message,  
    }  
    return response, nil  
}  
  
func (s myInvoicerServer) GetInformations(  
    ctx context.Context,  
    req *invoicer.CreateRequest) (*invoicer.RequestDest, error) {  
    response := &invoicer.RequestDest{  
        From: req.Request.From,  
        To:    req.Request.To,  
    }  
    return response, nil  
}
```

Et ensuite, on réalise la fonction main qui se chargera de lancer notre serveur avec le protocole TCP sur le port 8000 et de s'occuper de recevoir les requêtes du client :

```
func main() {  
    lis, err := net.Listen("tcp", ":8000")  
    if err != nil {  
        log.Fatalf("cannot create listener : %s", err)  
    }  
  
    serverRegister := grpc.NewServer()  
    service := &myInvoicerServer{}  
  
    invoicer.RegisterInvoicerServer(serverRegister, service)  
    err = serverRegister.Serve(lis)  
    if err != nil {  
        log.Fatalf("impossible to serve : %s", err)  
    }  
}
```

On lance notre serveur :

```
go run main.go
```

Et il ne reste plus qu'à tester !

## Mini-serveur GRPC - Client/Test

Pour tester notre serveur GRPC, 2 solutions sont possibles :

Client.go

```
func main() {

    conn, err := grpc.Dial(":8000", grpc.WithInsecure())
    if err != nil {
        log.Fatalf("could not connect to server: %v", err)
    }
    defer conn.Close()

    client := invoicer.NewInvoicerClient(conn)

    req := &invoicer.CreateRequest{
        Account: &invoicer.Account{
            Id:    2,
            Name: "Alexandre",
        },
        Request: &invoicer.RequestDest{
            From: &invoicer.Account{
                Id:    2,
                Name: "Alexandre",
            },
            To: &invoicer.Account{
                Id:    3,
                Name: "Antonin",
            },
        },
    },
    Message: &invoicer.Message{
        Message: "Hello",
    },
}

    response, err := client.GetInformations(context.Background(), req)
    if err != nil {
        log.Fatalf("could not create invoice: %v", err)
    }
    log.Printf("Response: %v", response)
}
```

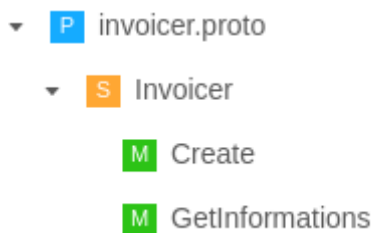
## BloomRPC

BloomRPC est un outil open source déprécié utilisé pour tester et interagir avec des services gRPC.

Il fournit une interface graphique conviviale qui permet aux développeurs de formuler des requêtes gRPC, d'envoyer des appels aux méthodes du service et de visualiser les réponses correspondantes.

Son utilisation est très simple :

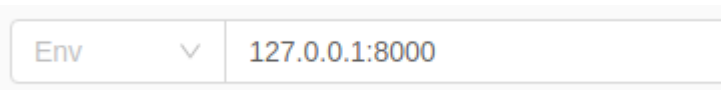
Après avoir lancé le serveur, il suffit d'importer le fichier protobuf pour voir apparaître les fonctions que l'on a définis :



En cliquant sur la fonction Create, on peut voir dans l'éditeur notre requête CreateRequest au format JSON dans laquelle on peut renseigner des valeurs :




Ensuite, il faut renseigner l'ip localhost (127.0.0.1) et le port dans la partie Env :



Puis, cliquer sur le bouton vert Play pour voir la réponse renvoyée par le serveur GRPC :

Response
<pre>{   "request": {     "from": {       "id": "20",       "name": "Alexandre"     },     "to": {       "id": "20",       "name": "Antonin"     }   },   "message": {     "message": "Hello"   } }</pre>



Cette réponse correspond au message CreateResponse du fichier protobuf.