Création d'un serveur grpc

Mathieu Tourrette[1], Benjamin Alonzo[2], Abdelouahid Behar[3], Anys Nait-Zerrad[4], and Mohamad Boukrit[5]

[1]APPING 2

12 juin 2022

Table des matières

Résumé

Choix du sujet 2 : Création d'un serveur gRPC qui implémente la méthode x –> x+1

À partir de ce sujet que nous avons décidé de traiter nous prendrons le rôle d'une entreprise prestataire intervenant auprès d'un client afin de repenser et restructurer l'ensemble de ces ressources d'infrastructure de stockage. Le format de rendu du projet sera rédigé intégralement sur latex. Nous mettons aussi dans le rapport le lien de 2 vidéos hébergées sur Youtube ainsi que celui renvoyant vers le github de notre projet.

https://github.com/SeaJag/Grpc$_{s}erveur$

1 Contexte

L'entreprise cliente BKCE fait appel à notre entreprise EPICrypto afin de moderniser son architecture et passer d'une application monolithique à une infrastructure de micro service. Il nous est demandé d'investiguer le problème de dialogue entre les services et de chiffrer les communications pour plus de sécurité. Nous implémentons donc un POC afin de montrer le fonctionnement du serveur gRPC que l'on met en place.

2 Introduction

L'entreprise de notre client souhaite moderniser son architecture et passer d'une application monolithique à une infrastructure de micro service. Cette application est implémentée de telle manière qu'elle demande la présence d'un serveur ayant la capacité d'interpréter plusieurs flux de données. En effet, ces flux proviennent d'un certain nombre de services distribués qui appartiennent à d'autres clients partenaires, qui utilisent donc chacun leur propre langage. De plus, le client souhaite aussi apporter à son application un service permettant de comptabiliser les entrées serveur avec une méthode x->x+1 sur celle-ci fonctionnant en concomitance avec tous les autres services.

La problématique première du client est donc d'avoir accès à un serveur distant pouvant interpréter ces flux (peu importe le langage) tout en conservant l'intégrité et la sécurité de toutes les données qui y transitent et y sont stockées. Lors de cette étude nous allons accompagner ce client à mettre en place un serveur gRPC qui pourra répondre à toutes ces exigences en termes de distribution, rapidité et sécurité. Les solutions proposées pour cela sont donc l'implémentation d'un serveur gRPC pour l'hébergement des données et l'échange avec diverses micro-services ainsi que d'un système d'authentification TLS/SSL.

Dans un premier temps, nous allons présenter au client pourquoi la solution de la mise en place d'un serveur grpc combiné à une authentification SSL/TLS est la solution la plus adaptée à sa demande en lui expliquant son fonctionnement et ses atouts.

Dans un second temps, nous allons réaliser un cas pratique qui démontre comment nous avons mis en place ce système ainsi que son fonctionnement suivi de la realisation d'un microservice implémentant la méthode x -> x + 1. Nous utiliserons des vidéos et des captures d'écrans afin d'illustrer nos propos.

3 Partie théorique

3.1 Serveur gRPC

Afin de répondre aux attentes de l'entreprise cliente et pour la mise en place d'un tel projet il faut exposer quelques notions pour la compréhension de l'étude. Qu'est ce qu'un serveur gRPC, en quoi ce type de serveur va répondre à nos besoins et quelle fonctionnalité peut-il nous offrir ?

gRPC est un Framework d'appel de procédure distante (RPC) indépendant du langage et de haute performance permettant une communication particulièrement efficace au sein d'architecture client/serveur. Tout comme son prédécesseur le RPC, il fonctionne au niveau des processus. La caractéristique de la communication inter-processus via gRPC est le principe de transparence, une caractéristique similaire à la communication locale entre plusieurs processus d'une même machine.

Ces relations entre instances (partiellement) éloignées sont tout autant étroites et simples. Il est donc très intéressant pour y faire tourner de multiples services, Il agit comme une API “gateway” ou interfaces pour toutes les autres APIs/services. Elles se doit donc d’être robuste et efficace.

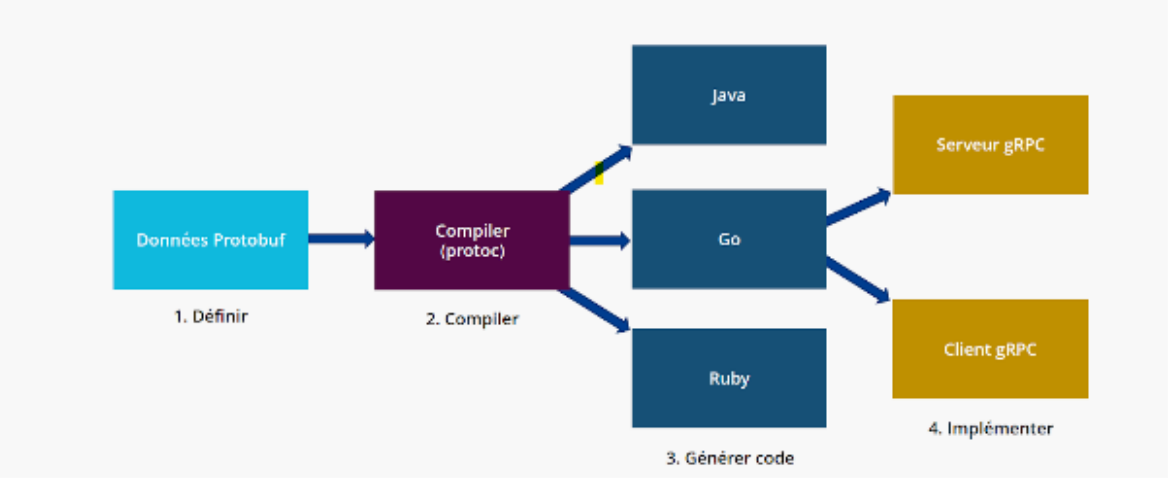
Rentrons dans les détails..

3.2 Comment cela fonctionne ?

Dans le gRPC, un client peut faire appel à une méthode via le serveur sur une différente machine comme s’il s’agissait d’un objet local. Le gRPC se base sur l’idée de définir un service, spécifier les méthodes qui peuvent être appelées à distance avec leur paramétrage et leur "return type”. Côté serveur, celui-ci implémente une interface qui va permettre de gérer les requêtes des différents clients, le client à lui un “stub” (interface client) qui possède les même méthodes que le serveur.

Il faut tout d’abord définir ce qu’est un Protocol Buffers ou protobuf afin de comprendre la suite des explications.

Les Protocol Buffers (Protobuf) remplissent plusieurs fonctions au sein du système gRPC : ils servent d’Interface Definition Language, décrivant une interface indépendamment de son langage. Ils ne sont donc pas liés à un langage de programmation spécifique (par exemple Java, C ou bien Python). Ils définissent en outre les services à utiliser et les fonctions mises à disposition dans un fichier “.proto”. Le fichier .proto, est par la suite compilé grâce à protoc (explication dans le point d’après) dans un langage supporté pour créer les différentes classes et méthodes afin d’implémenter le serveurs.



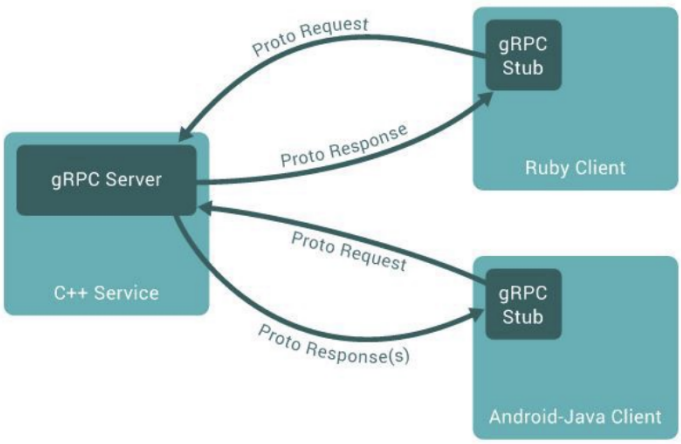
Définition du fichier .proto pour la communication inter-processus : détermination des services à appliquer, ainsi que des paramètres et types de réponse principales susceptibles d’être appelés à distance.

Génération du code gRPC à partir du fichier .proto : des compilateurs spéciaux (« protoc ») génèrent le code opérationnel de la classe correspondante d’un langage cible souhaité à partir des fichiers .proto enregistrés (par exemple C++, Java).

Implémentation du serveur dans le langage sélectionné.

Création du stub client (instance du client) pour l’appel du service ; la ou les requête(s) est/sont ensuite exécutée(s) via le serveur et le ou les client(s).

Toujours dans une optique de communication à distance entre différents systèmes, les Protobufs servent de format d’échanges de message de base, ils déterminent les structures, types et objets de messages. Il va tout simplement venir s’assurer que le client et le serveur se comprennent.



Voici (ci-dessus) une illustration du fonctionnement du serveur gRPC et comment se font les interactions clients serveurs.

Pour l'échange de flux de données entre machines (Proto Request et Proto Response), le protocole HTTP/2 est intégré à des protocoles réseau spéciaux, tels que TCP/IP ou UDP/IP. Les flux de données transfèrent les données binaires compactes générées lors de la sérialisation (Marshalling) typique des Remote Procedure Calls. Pour le traitement complet des structures de données abstraites côté client et côté serveur en plusieurs étapes, les flux transmis sont également désérialisés (Unmarshalling).

3.3 Protoc (Compilateur)

En examinant le man de la commande protoc on a la description suivante :

"protoc is a compiler for protocol buffers definitions files. It can generate C++, Java and Python source code for the classes defined in PROTO_FILE."

Protoc est donc un compiler qui va créer les classes et les méthodes dans un langage souhaité à partir d'un fichier ".proto". Ce compilateur est donc un atout majeur sur la mise en place d'un système acceptant le multilanguage.

3.4 La force du gRPC

Notre client à mentionner límportance d'avoir des échanges de requête rapide et sécurisé à son serveur. En lóccurrence gRPC offre des performances et une sécurité API jusqu'à 10 fois plus rapide que la communication REST+JSON car il utilise HTTP2.

HTTP2 améliore les performances grâce à :
- La poussée du serveur qui permet à HTTP2 de pousser le contenu du serveur vers le client avant d'être demandé.
- Le multiplexage qui élimine le blocage en tête de ligne.
- Une méthode de compression plus avancée pour réduire la taille des messages des en-têtes. Ce qui accélère le chargement.

Mais ce n'est pas tout, les forces du serveur gRPC résident aussi dans sa capacité de diffusion, en effet il prend en charge la sémantique de diffusion côté client ou côté serveur, déjà intégrée à la définition de service. Cela simplifie grandement la création de services ou de streaming. De plus, le service gRPC traite les différentes combinaisons de relais HTTP2 (notamment Diffusion de serveur à client, Streaming client-serveur).

La tolérance et la productivité qu'offre un serveur gRPC permet un fonctionnement transparent et intuitif. En effet, le format binaire permet aux applications de stocker et d'échanger facilement des données structurées. Ces programmes peuvent être codés dans des langages de programmation différents (Java, Ruby, C#, Python, Go, etc) et avec divers logiciels.

Ce type de serveur possède une bibliothèque conçue pour fonctionner avec une variété d'outils dissemblables et va donc permettre à notre client d'avoir une large flexibilité et interopérabilité quant à ces différentes sources de données. De plus, il dispose de bons outils pour générer du code passe-partout indispensable.

3.5 TLS/SSL, la solution pour nos communications sécurisés

SSL signifie Secure Sockets Layer, il sera au cœur de nos communications. Il s'agit d'une technologie standard destinée à la sécurité de la connexion Internet et à la protection des données sensibles qui sont transmises entre deux systèmes, empêchant des individus malveillants de lire et de modifier les informations transférées, y compris d'éventuelles informations personnelles. Les communications se font systématiquement entre deux systèmes, un client et un serveur ou bien deux serveurs.

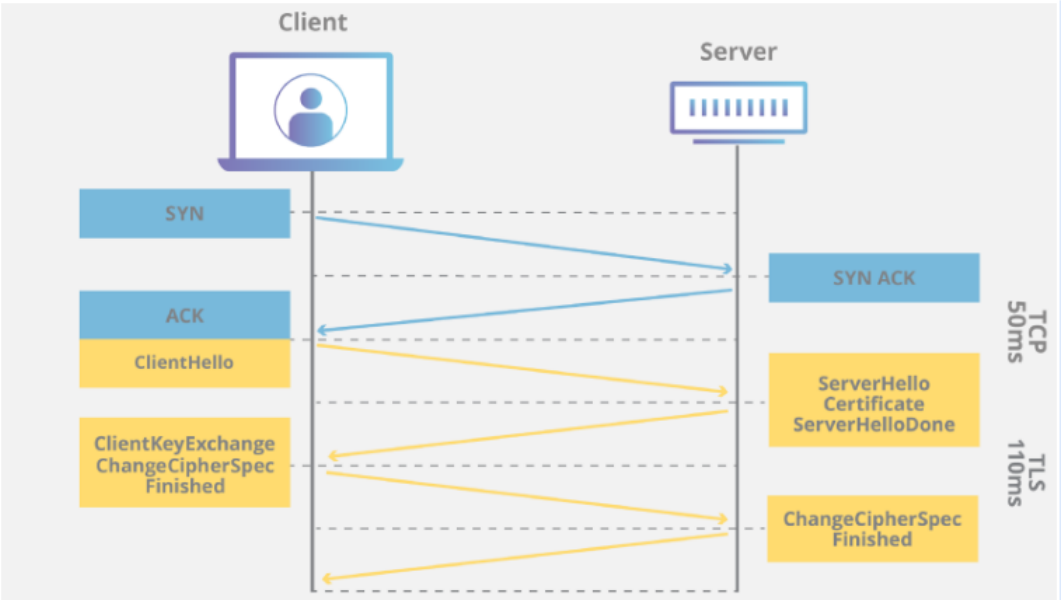
Il agit en veillant à ce que les données transférées entre les utilisateurs et les sites, ou entre les deux systèmes restent impossibles à lire. SSL utilise des algorithmes de chiffrement tels que AES, ou RSA bien SHA, pour brouiller les données en transit, empêchant des individus malveillants de les lire lorsqu'elles sont envoyées via la connexion. Étant donné la criticité des données qui peuvent être transmises tel que des informations personnelles, financières ou autres, il est évident d'accorder une telle importance sur les communications et la protection de ces données. TLS (Transport Layer Security) est simplement une nouvelle version, plus sûre, de SSL.

Voici un exemple :

Un site web qui met en œuvre le protocole SSL/TLS comporte un «HTTPS » dans son URL au lieu d'un « HTTP ». C'est pourquoi nous retrouvons systématiquement un petit logo avec un loquet sur les url de certains sites, comme les sites bancaires, administratifs ou gouvernementaux.

Comment se déroule la négociation SSL/TLS ?

La négociation TLS (également nommée handshake) est le processus qui amorce une session de communication utilisant le chiffrement TLS. Au cours de cette négociation, les deux parties communicantes s'échangent l'une l'autre des messages d'authentification et de vérification, établissent les algorithmes de chiffrement qu'elles utilisent et se mettent d'accord sur les clés de session. Cette négociation constitue un élément fondamental du fonctionnement du protocole HTTPS.



Voici les étapes d'une négociation TLS :

1- Message « Client Hello » : le client démarre la négociation en envoyant un message « Hello » au serveur. Le message inclut la version TLS prise en charge par le client, les algorithmes de chiffrement prises en charge et une chaîne d'octets aléatoires connue sous le nom de « client random » ou un nombre aléatoire client.

2- Message « Server Hello » : en réponse au message Client Hello, le serveur envoie un message contenant le certificat SSL (dans le certificat se trouve une clé publique transmis au client) du serveur, la suite de chiffrement choisie par le serveur (AES, RSA..) et le « Server random », une autre chaîne aléatoire d'octets générée par le serveur.

3- Authentification : le client vérifie le certificat SSL du serveur auprès de l'autorité de certification qui l'a émis. Cette opération confirme que le serveur est bien celui qu'il prétend être et que le client interagit avec le véritable propriétaire du domaine.

4- Secret pré-maître : le client envoie une autre chaîne d'octets aléatoire, le « premaster secret », ou secret pré-maître. Le secret pré-maître est chiffré à l'aide de la clé publique et ne peut être déchiffré par le serveur qu'avec la clé privée. (Le client obtient la clé publique dans le certificat SSL du serveur.)

5- Utilisation de la clé privée : le serveur déchiffre le secret pré-maître.

6- Création des clés de session : le client et le serveur génèrent des clés de session à partir du client random, du serveur random et du secret pré-maître. Ils doivent aboutir aux mêmes résultats.

7- Client prêt : le client envoie un message « Client Finished » chiffré à l'aide d'une clé de session.

8- Serveur prêt : le serveur envoie un message « Server Finished » chiffré à l'aide d'une clé de session.

9- Chiffrement symétrique sécurisé effectué : la négociation est terminée et la communication se poursuit à l'aide des clés de session.

3.6 La méthode x->x+1

Cette méthode à fait l'objet d'une demande expresse du client. En effet, le client souhaite mettre en place un système comptabilisant le nombre d'accès au serveur à travers l'ensemble des micro services mis en place.

Cette méthode nous servira d'étalon ainsi que de test de confirmation quant au fonctionnement de l'appel et du retour faite par le serveur. Basiquement il s'agira dans notre cas (essaie) d'envoyer un chiffre aléatoire au serveur et d'attendre en retour le même chiffre auquel a été ajouté 1.

4 Partie pratique

Après avoir eu un aperçu global du fonctionnement d'un tel serveur, nous allons pouvoir répondre aux demandes du client quant à sa nouvelle implémentation d'architecture. Le client souhaite avoir la possibilité d'avoir accès à un serveur acceptant le multilanguage, une haute disponibilité des ses ressources ainsi que de faire appel à une multitude de micros services. Son but est de pouvoir faire fonctionner ses nouvelles applications et qu'elles soient mises en concordance avec plusieurs prestataires travaillant en collaboration.

Comme nous avons pu le voir, la mise en place d'un serveur gRPC répond au mieux à cette problématique grâce à sa haute disponibilité et sa fluidité (multiplexage, compression, poussée du serveur). Il répond aussi aux problématiques liées aux échanges et aux requêtes. Sa capacité à interpréter n'importe quel langage avec son système protocol buffer, compilant via la commande protoc, permet une forte compliance.

Enfin la demande du client concernant la sécurité de ses données étant une requête importante nous utiliserons TLS/SSL pour le chiffrement et le système d'authentification via des certificats.

Nous allons maintenant avoir une approche plus pratique de la mise en place du gRPC ainsi que du système d'authentification TLS/SSL qui fonctionnera avec celui-ci.Lors de cette partie nous présenterons la façon dont on peut mettre en place cette solution.

4.1 Protobuf

Nous allons créer un fichier.proto décrivant le service avec une requête et une réponse. Les classes d'accès aux données sont automatiquement générées à partir du fichier proto où nous avons spécifié toutes nos méthodes et définitions de service.

```
1     syntax = "proto3";
2
3     option java_package = "mahb.grpc";
4
5     service TemplateGRPC{
6         rpc TemplateGRPCEndpoint(TemplateGRPCRequest)
7         returns (TemplateGRPCResponse);
8     }
9
10    message TemplateGRPCRequest{
11            int32 number = 1;
12    }
13
14    message TemplateGRPCResponse{
15            string reply = 1;
16    }
```

4.2 Creation du serveur

Pour notre serveur nous avons défini notre endpoint pour retourner le message quand ce point de terminaison est appelé.

```
1         def TemplateGRPCEndpoint(self, request, context):
2             response = templateGRPC_pb2.TemplateGRPCResponse()
3             response.reply = "Send: {}, Return: {}".format(
4                 request.number, request.number + 1)
5             return response
```

Dans un premier temps, nous créons un service avec une connexion non sécurisée avec le framework gRPC sous python.

```
        server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

        templateGRPC_pb2_grpc.add_TemplateGRPCServicer_to_server(
            POCService(), server)

        server.add_insecure_port('[::]:5000')
        server.start()
```

4.3 Création du client

Du côté client, nous pouvons faire l'objet de la classe de requête qui est générée automatiquement, et passer les paramètres requis que nous avons spécifiés dans la définition du fichier ".proto".

Ici, nous ouvrons un channel sur le port 5000 non sécurisé.

```
    channel = grpc.insecure_channel('localhost:5000')
    stub = templateGRPC_pb2_grpc.TemplateGRPCStub(channel)
```

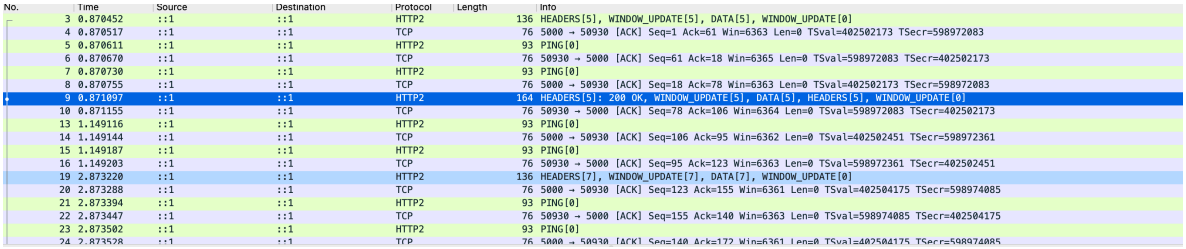
Ici, nous générons un nombre aléatoire dans le champ "number". Une fois la requête prête, il ne nous reste plus qu'à l'envoyer.

```
    request = templateGRPC_pb2.TemplateGRPCRequest(
        number=random.randint(0, 9999)
    )
```

Voici une vidéo explicative afin de vous présenter le fonctionnement des parties 4.1 , 4.2 et 4.3 : https://www.youtube.com/watch?v=QcJ15SAiixI

4.4 Observons le trafic

Afin de mieux comprendre comment le service fonctionne nous allons voir avec Wireshark quel paquet va être envoyé.



No.	Time	Source	Destination	Protocol	Length	Info
3	0.870452	::1	::1	HTTP2	136	HEADERS[5], WINDOW_UPDATE[5], DATA[5], WINDOW_UPDATE[0]
4	0.870517	::1	::1	TCP	76	5000 → 50930 [ACK] Seq=1 Ack=61 Win=6363 Len=0 TSval=402502173 TSecr=598972083
5	0.870611	::1	::1	HTTP2	93	PING[0]
6	0.870670	::1	::1	TCP	76	50930 → 5000 [ACK] Seq=61 Ack=18 Win=6365 Len=0 TSval=598972083 TSecr=402502173
7	0.870730	::1	::1	HTTP2	93	PING[0]
8	0.870755	::1	::1	TCP	76	5000 → 50930 [ACK] Seq=18 Ack=78 Win=6363 Len=0 TSval=402502173 TSecr=598972083
9	0.871097	::1	::1	HTTP2	164	HEADERS[5]: 200 OK, WINDOW_UPDATE[5], DATA[5], HEADERS[5], WINDOW_UPDATE[0]
10	0.871155	::1	::1	TCP	76	50930 → 5000 [ACK] Seq=78 Ack=106 Win=6364 Len=0 TSval=598972083 TSecr=402502173
13	1.149116	::1	::1	HTTP2	93	PING[0]
14	1.149144	::1	::1	TCP	76	5000 → 50930 [ACK] Seq=106 Ack=95 Win=6362 Len=0 TSval=402502451 TSecr=598972361
15	1.149187	::1	::1	HTTP2	93	PING[0]
16	1.149203	::1	::1	TCP	76	50930 → 5000 [ACK] Seq=95 Ack=123 Win=6363 Len=0 TSval=598972361 TSecr=402502451
19	2.873220	::1	::1	HTTP2	136	HEADERS[7], WINDOW_UPDATE[7], DATA[7], WINDOW_UPDATE[0]
20	2.873288	::1	::1	TCP	76	5000 → 50930 [ACK] Seq=123 Ack=155 Win=6361 Len=0 TSval=402504175 TSecr=598974085
21	2.873394	::1	::1	HTTP2	93	PING[0]
22	2.873447	::1	::1	TCP	76	50930 → 5000 [ACK] Seq=155 Ack=140 Win=6363 Len=0 TSval=598974085 TSecr=402504175
23	2.873502	::1	::1	HTTP2	93	PING[0]
24	2.873528	::1	::1	TCP	76	5000 → 50930 [ACK] Seq=140 Ack=172 Win=6361 Len=0 TSval=402504175 TSecr=598974085

La capture nous apprend déjà que notre serveur/client dialogue bien ensemble avec le protocole HTTP/2. De plus, la connexion entre notre client/serveur n'est pas encore chiffrée cela nous permet de voir plus simplement les trames. Une fois que notre client/serveur s'est mis d'accord sur leur méthode d'échange nous recevons un message avec le retour du serveur.

```
1e 00 00 00 60 08 0f 00  00 78 06 40 00 00 00 00   ····`···  ·x·@····
00 00 00 00 00 00 00 00  00 00 00 01 00 00 00 00   ········  ········
00 00 00 00 00 00 00 00  00 00 00 01 13 88 c6 f2   ········  ········
9e 90 91 5e fa 3f 41 92  80 18 18 db 00 80 00 00   ···^·?A·  ········
01 01 08 0a 17 fd b2 1d  23 b3 96 b3 00 00 03 01   ········  #·······
04 00 00 00 05 88 c0 bf  00 00 04 08 00 00 00 00   ········  ········
05 00 00 00 0d 00 00 1f  00 00 00 00 00 05 00 00   ········  ········
00 00 1a 0a 18 53 65 6e  64 3a 20 35 36 38 31 2c   ·····Sen  d: 5681,
20 52 65 74 75 72 6e 3a  20 35 36 38 32 00 00 01   Return:  5682···
01 05 00 00 00 05 be 00  00 04 08 00 00 00 00 00   ········  ········
00 00 00 08                                        ····
```

4.5 Chiffrement TLS

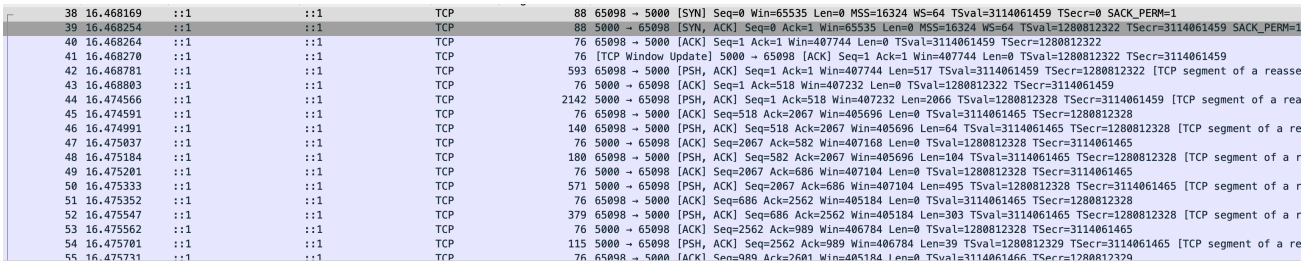
Pour l'instant, comme nous avons pu le voir, notre trafic est en clair sur le réseau. Regardons comment chiffrer notre connexion : Dans un premier temps il va falloir créer des clefs privées/publiques pour nos clients /server. Après cela nous intégrons nos clefs dans notre code.

```
    with open('ssl/server.key', 'rb') as f:
        private_key = f.read()
    with open(\'ssl/server.crt\', \'rb\') as f:
        certificate_chain = f.read()

    server_credentials = grpc.ssl_server_credentials(
        ((private_key, certificate_chain), ))

    server.add_secure_port('[::]:5000', server_credentials)
```

Maintenant, nous avons pu ajouter nos clefs à notre serveur. Il ne reste plus qu'à faire la même chose pour notre client en lui fournissant uniquement la clef publique de notre certificat.

```
    with open('ssl/server.crt', 'rb') as f:
        creds = grpc.ssl_channel_credentials(f.read())
    channel = grpc.secure_channel('localhost:5000', creds)
```

Si nous regardons à nouveau le trafic nous voyons bien le que le trafic est bien sécurisé avec la version TLSV1.3 nous retrouvons notre hiérarchie du TLS avec le SYN, ACK.



38	16.468169	::1	::1	TCP	88	65098 → 5000 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=3114061459 TSecr=0 SACK_PERM=1
39	16.468254	::1	::1	TCP	88	5000 → 65098 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16324 WS=64 TSval=1280812322 TSecr=3114061459 SACK_PERM=1
40	16.468264	::1	::1	TCP	76	65098 → 5000 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=3114061459 TSecr=1280812322
41	16.468270	::1	::1	TCP	76	[TCP Window Update] 5000 → 65098 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=1280812322 TSecr=3114061459
42	16.468781	::1	::1	TCP	593	65098 → 5000 [PSH, ACK] Seq=1 Ack=1 Win=407744 Len=517 TSval=3114061459 TSecr=1280812322 [TCP segment of a reasse
43	16.468803	::1	::1	TCP	76	5000 → 65098 [ACK] Seq=1 Ack=518 Win=407232 Len=0 TSval=1280812322 TSecr=3114061459
44	16.474566	::1	::1	TCP	2142	5000 → 65098 [PSH, ACK] Seq=1 Ack=518 Win=407232 Len=2066 TSval=1280812328 TSecr=3114061459 [TCP segment of a rea
45	16.474591	::1	::1	TCP	76	65098 → 5000 [ACK] Seq=518 Ack=2067 Win=405696 Len=0 TSval=3114061465 TSecr=1280812328
46	16.474991	::1	::1	TCP	140	65098 → 5000 [PSH, ACK] Seq=518 Ack=2067 Win=405696 Len=64 TSval=3114061465 TSecr=1280812328 [TCP segment of a re
47	16.475037	::1	::1	TCP	76	5000 → 65098 [ACK] Seq=2067 Ack=582 Win=407168 Len=0 TSval=1280812328 TSecr=3114061465
48	16.475184	::1	::1	TCP	180	65098 → 5000 [PSH, ACK] Seq=582 Ack=2067 Win=405696 Len=104 TSval=3114061465 TSecr=1280812328 [TCP segment of a r
49	16.475201	::1	::1	TCP	76	5000 → 65098 [ACK] Seq=2067 Ack=686 Win=407104 Len=0 TSval=1280812328 TSecr=3114061465
50	16.475333	::1	::1	TCP	571	5000 → 65098 [PSH, ACK] Seq=2067 Ack=686 Win=407104 Len=495 TSval=1280812328 TSecr=3114061465 [TCP segment of a r
51	16.475352	::1	::1	TCP	76	65098 → 5000 [ACK] Seq=686 Ack=2562 Win=405184 Len=0 TSval=3114061465 TSecr=1280812328
52	16.475547	::1	::1	TCP	379	65098 → 5000 [PSH, ACK] Seq=686 Ack=2562 Win=405184 Len=303 TSval=3114061465 TSecr=1280812328 [TCP segment of a r
53	16.475562	::1	::1	TCP	76	5000 → 65098 [ACK] Seq=2562 Ack=989 Win=406784 Len=0 TSval=1280812328 TSecr=3114061465
54	16.475701	::1	::1	TCP	115	5000 → 65098 [PSH, ACK] Seq=2562 Ack=989 Win=406784 Len=39 TSval=1280812329 TSecr=3114061465 [TCP segment of a re
55	16.475731	::1	::1	TCP	76	65098 → 5000 [ACK] Seq=989 Ack=2601 Win=405184 Len=0 TSval=3114061466 TSecr=1280812329

Maintenant nous avons bien le resulat de notre fonction qui retourne notre nombre avec plus 1.

```
reply: "Send: 9180, Return: 9181"

reply: "Send: 1296, Return: 1297"

reply: "Send: 9141, Return: 9142"
```

Voici une vidéo explicative afin de vous présenter le fonctionnement des parties 4.4 et 4.5 : https://www.youtube.com/watch?v=4Gdnev7v6Ic

5 Conclusion

Au cours de ce POC, nous avons dû répondre aux problématiques du client, celui-ci ayant un besoin d'une optimisation d'architecture au niveau de ses services, à cet égard nous lui avons proposé ce POC autour de cette problématique afin de répondre à ses besoins.

De notre réflexion est née l'utilisation d'un serveur gRPC afin de basculer sur une infrastructure de micro-service et donc de faciliter l'utilisation des différents services clients. Ce type de système a été une solution que nous avons jugé bon de présenter à notre client, de par ses fonctionnalités, sa capacité à accepter le multi-langage et sa disponibilité à user de plusieurs sources de données à distance.

De surcroît, nous avons estimé utile de sécuriser les communications entre services, celles-ci étant à distance, en utilisant le protocole SSL/TLS. Ce dernier permettra, en l'espèce, de chiffrer les données transmises dans les paquets TCP.

Lors de ce POC, nous avons donc pris le temps d'expliquer le fonctionnement de tous ces éléments à la fois d'un point de vue théorique, mais également pratique, afin que le client comprenne la perspective de ce changement d'architecture, mais aussi pour qu'il comprennent et envisage le large éventail de possibilités qu'offre ce type d'architecture.

6 Annexe

— https://www.digicert.com/how-tls-ssl-certificates-work
— https://www.cloudflare.com/fr-fr/learning/ssl/what-is-ssl/
— https://www.bortzmeyer.org/xml-rpc-for-blogs.pdf
— https://docs.servicestack.net/grpc
— https://main.qcloudimg.com/raw/document/intl/product/pdf/$1055_37406_en.pdf$
— https://github.com/SeaJag/$Grpc_{S}erveur$
— https://www.youtube.com/watch?v=QcJ15SAiixI
— https://www.youtube.com/watch?v=4Gdnev7v6Ic