Berney Alec & Forestier Quentin & Herzig Melvyn

SYM – 21.11.2021

Protocoles applicatifs

Labo 2



Table des matières

[Introduction 2](#_Toc88388056)

[Démarrage 2](#_Toc88388057)

[Remarques préalables 2](#_Toc88388058)

[Manipulations 3](#_Toc88388059)

[Asynchrone 3](#_Toc88388060)

[Utilisation 3](#_Toc88388061)

[Mise en place 3](#_Toc88388062)

[Retardé 4](#_Toc88388063)

[Utilisation 4](#_Toc88388064)

[Mise en place 5](#_Toc88388065)

[Limitations 5](#_Toc88388066)

[Transmission d’objets 6](#_Toc88388067)

[Utilisation 6](#_Toc88388068)

[Mise en place 7](#_Toc88388069)

[XML 7](#_Toc88388070)

[JSON 7](#_Toc88388071)

[Protobuf 8](#_Toc88388072)

[Compressé 9](#_Toc88388073)

[Utilisation 9](#_Toc88388074)

[Mise en place 10](#_Toc88388075)

[GraphQL 11](#_Toc88388076)

[Utilisation 11](#_Toc88388077)

[Mise en place 11](#_Toc88388078)

[Réponse aux questions 12](#_Toc88388079)

[Traitement des erreurs 12](#_Toc88388080)

[Authentification 12](#_Toc88388081)

[Threads concurrents 13](#_Toc88388082)

[Écriture différée 13](#_Toc88388083)

[Transmission d’objets 14](#_Toc88388084)

[Transmission compressée 15](#_Toc88388085)

# Introduction

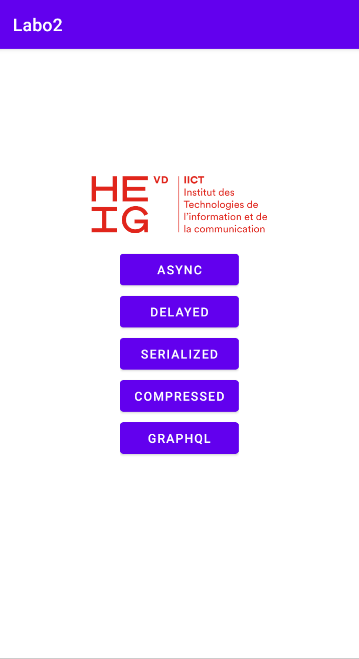
Ce laboratoire est une introduction aux techniques de programmation réparties asynchrones. Nous allons illustrer l’utilisation de cinq techniques de protocoles asynchrones sur plateforme mobile.

Les techniques seront :

* Asynchrone
* Différée
* Sérialisée
* Compressée
* GraphQL

# Démarrage

L’application démarre sur la page principale :



Chaque bouton crée une nouvelle activité qui déclenche une transmission selon la technique qui lui est respectivement associée.

## Remarques préalables

Ce laboratoire est une introduction à la programmation asynchrone. En conséquence, certains aspects de la programmation Android ne sont pas développés par manque de temps. C’est pourquoi les activités ne sauvegardent pas leur état en cas destruction.

De plus, il est aisé de créer des fuites de mémoire lorsque nous ne gérons pas correctement les références entre les objets. Encore une fois, ce laboratoire n’est pas dirigé sur la gestion des fuites mémoires. Cependant, nous avons tenté de limiter leur effet en travaillant aussi peu que possible avec des classes anonymes et lorsque cela a été nécessaire, nous avons travaillé avec des références faibles.

# Manipulations

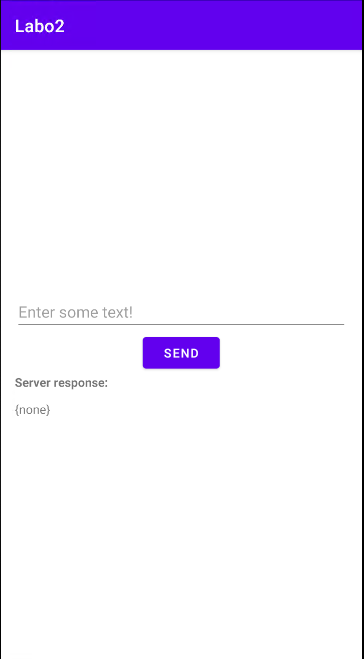
Comme présenté préalablement, ce laboratoire consiste en 5 manipulations majeures. Nous avons donc réalisé une activité par manipulation. Chaque activité, à la création, démarre son objet *SymComManager* qui est responsable de négocier les communications. Nous avons réussi à adapter le *SymComManager* pour qu’il fonctionne avec les 5 activités. Au cours des points suivants, nous allons décrire de manière incrémentale ce que nous avons mis en place pour arriver à l’état final de ce laboratoire.

## Asynchrone

Cette manipulation est le fondement de toutes les autres. Elle consiste en la mise en place du mécanisme principale de communication asynchrone.

### Utilisation

Nous avons créé l’activité *AsyncActivity.* Le layout est très basique.



L’utilisateur entre du texte dans le premier champ. Lorsqu’il clique sur « Send », une communication démarre avec un serveur qui se contente de renvoyer le corps de la requête. Une fois la réponse reçue, le contenu envoyé s’affiche en dessous de «Serveur response : »

### Mise en place

Comme les transmissions de ce laboratoire sont effectuées dans le web, nous avons commencé par implémenter un objet **SymComRequest.**

Le *symComRequest* est un objet simple qui prend 4 paramètres :¨

* Une URL de connexion :*url* en string
* Un body :*body* en string
* Un content type : *contentType* de type *ContentType*
* Une méthode : *requestMethod* de type *RequestMethod*

Concernant les deux derniers paramètres, nous avons donc créé deux énumérations :

*enum class ContentType(val value: String) { TEXT("text/plain) }  
enum class RequestMethod(val value: String) { GET("GET"), POST("POST") }*

Ensuite, nous avons mis en place le ***SymComManager***. Il offre deux méthodes principales

*fun setCommunicationEventListener(communicationEventListener: CommunicationEventListener)* qui s’occupe de récupérer un *communicationEventListener* et le stocke dans une référence faible à cause du fait que les *communicationEventListener* sont des classes anonymes au niveau des activités.

*fun sendRequest( request: SymComRequest )* qui récupère la requête et démarre un thread de communication en passant la *request* ainsi que une référence faible sur le *communicationEventListener*

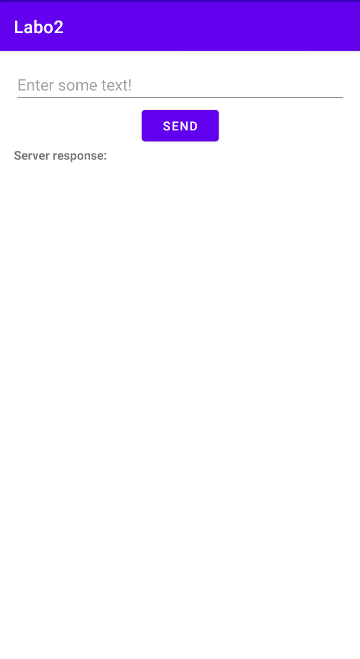
Finalement nous avons créé la classe **SymComThread** qui permet d’exécuter une requête *SymComRequest* sur autre thread que l’UI thread. Il est construit à partir d’une SymComRequest et d’une référence faible sur un *CommunicationEventListener.* Une fois la réponse reçue, le *SymComThread* retourne la réponse sur l’UI thread en créant un handler sur le *mainLooper*. La réponse est retournée grâce à la méthode post avec un *ResponseStringRunnable* qui est une classe interne statique pour permettre au thread de se terminer sans que le *runnable* ait préalablement été traité par le UI thread..

## Retardé

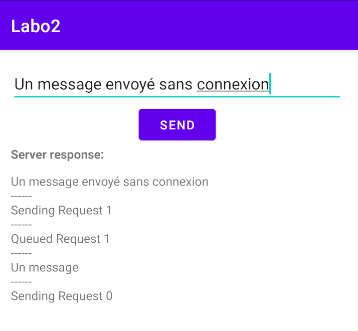
La mise en place de la communication retardée est sensiblement proche de la première manipulation.

### Utilisation

Au lancement le layout ressemble à ceci :



Le fonctionnement est similaire à la première manipulation. Toutefois le résultat des requêtes ne s’écrase pas et leur état s’affiche à la suite avec un logging.



Signification des messages

|  |  |  |  |
| --- | --- | --- | --- |
| Cause | Pas de connexion | Connexion, envoi | Réponse du serveur |
| Message | Queud Request <id> | Sending Request <id> | <input de l’utilisateur> |

L’envoi est déclenché automatiquement si la requête est envoyée avec de la connexion ou automatiquement (toutes les 10s) si la requête a été mise en file d’attente.

### Mise en place

Premièrement nous avons modifié le **SymComManager**. Nous avons ajouté une file de pairs de requête et de référence faible sur le *CommunicationEventListener*. Désormais, lors de *sendRequest,* si la communication internet n’est pas possible, la requête est placée dans la file.

Ensuite, à la création du *SymComManager,* nous démarrons un *Timer* qui vérifie toutes les 10secondes si la connexion est disponible. Si c’est le cas, il envoie les requêtes et les retire de la file.

Les requêtes mises en file d’attente sont pairées avec le *CommunicationEventListener* actif au moment de la tentative d’envoi. Ainsi, il est possible d’effectuer des requêtes avec différents *CommunicationEventListener.*

Finalement nous avons ajouté une méthode quit(). Cette méthode, arrête le time. Elle est appelée lors de l’appel à la méthode onDestroy() de l’activité.

Nous avons ajouté une nouvelle activité abstraite **BaseActivity**

abstract class BaseActivity : AppCompatActivity() {  
 protected lateinit var symComManager: SymComManager  
override fun onDestroy() {  
 super.onDestroy()  
 symComManager.quit()  
 }  
}

Cette nouvelle activité abstraite sera la base de toutes les activités passées, présentes et futures relatives aux manipulations .

### Limitations

* Les requêtes mises en cache sont stockées en mémoire. Si l’application doit être redémarrée, toutes les requêtes en attentes sont perdues. ¨
  + **Solution** : utiliser un stockage persistant comme une base de données locale avec Room ou simple fichier. Au démarrage l’application peut interroger ces stockages pour récupérer les requêtes non traitées.
* Nous utilisons un Timer afin de programmer l’envoi des requêtes en attentes. Concrètement les Timers sont des threads déguisés. Nous risquons d’avoir des problèmes de concurrence lors de l’accès à la structure de données conservant les requêtes non envoyées. De plus, la tâche est exécutée périodiquement et doit vérifier si la connexion est rétablie. Bien que cette manière de faire soit simple à mettre en place, ce n’est pas l’approche conseillée.
  + **Solution** : utiliser le *WorkManger* pour planifier notre tâche, tout en y ajoutant des contraintes d’exécution : batterie suffisante, connectivité… Ainsi nous utilisons une façon conseillée par Android et non pas une solution home made.

## Transmission d’objets

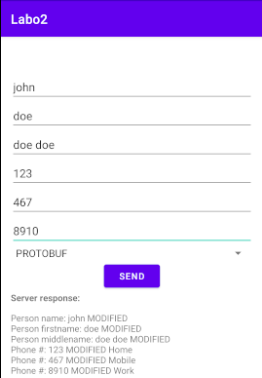
La mise en place de la communication sérialisée d’objet est différente des autres activités, car il est demandé d’envoyer des objets plus complexes qu’un champ texte.

### Utilisation

Au lancement le layout ressemble à ceci :

Une image contenant table

Description générée automatiquement

Le fonctionnement est le suivant, il faut remplir les 3 premiers champs (name, firstname et middlename), puis optionnellement remplir les numéros de téléphone voulus. Une fois ses champs remplis, il est possible de choisir le type d’envoi via la liste déroulante. Il est possible d’envoyer un message avec les formats suivants :

* Texte
* JSON
* XML
* Protocol Buffer

Une fois le format choisi, cliquer sur le bouton « Send » pour envoyer vos données au serveur.

La réponse reçue par le serveur sera ensuite affichée sous le label « Server response ».

### Mise en place

Premièrement nous avons modifié le **SymComThread** afin que ce dernier n’envoie et reçoive que du ByteArray.

Nous avons également créé des classes « Model » qui sont **Person** et **Phone** afin de stocker les données du formulaire et réalisé les transformations pour les différentes sérialisations directement depuis des objets. Ces classes font partie du package **models**. Chaque implémentation spécifique à un protocole sera décrite plus bas.

Nous avons aussi décidé de transformer la classe **SymComRequest** en classe abstraite et de créer 2 classes héritant de cette dernière :

* SymComBytesRequest
* SymComStringRequest

Ces 2 classes possèdent un type de « body » différent en fonction du type de donnée que l’on souhaite envoyer. Cependant, elles implémentent toutes les 2 obligatoirement la méthode **getBytesFromBody** permettant de retourner du ByteArray pour n’importe quel type de requête puisqu’il a été décidé d’envoyer et de recevoir uniquement du ByteArray.

Pour en finir avec les modifications de **SymComRequest**, nous avons ajouté des valeurs à notre enum **ContentType** afin de pouvoir utiliser d’autres formats :

*enum class ContentType(val value: String) { TEXT("text/plain)*, *JSON("application/json"), XML("application/xml"), PROTOBUF("application/protobuf")}*

L’activité **SerializedActivity** est structurée d’une façon à ce que chaque format d’envoie possède une fonction implémentant le **setCommunicationEventListener** et créant une requête à envoyer à **sendRequest**. Il est ainsi possible d’envoyer des requêtes différentes et de traiter les réponses différemment.

Nous avons également fait en sorte de créer un objet **Person** avec les informations du formulaire, qui sera ensuite utilisé part les méthodes d’envoi de données décrites ci-dessus.

### XML

Nous avons créé les méthodes toXML et fromXML dans la classe Person. La méthode toXML sérialise l’objet Person en XML grâce à la librairie Jackson. Cette libraire sérialise automatiquement l’objet Person, ainsi que tous les Phones qu’il possède en XML. Nous avons rajouté des décorations afin que la sérialisation corresponde à la DTD demandée.

La méthode fromXML est statique. Elle reçoit un string en paramètre, correspondant au XML de la personne à désérialiser. Il n’a pas été possible d’utiliser Jackson pour la désérialisation, car la gestion de l’enum dans Phone était très compliquée. Nous avons donc décidé de ne pas perdre trop de temps, et de désérialiser « à la main ».

### JSON

Les méthodes toJson et fromJson ont été créées dans la classe Person. Les 2 méthodes utilisent la librairie Gson pour sérialiser et désérialiser l’objet Person, ainsi que tous ses Phone. Il faut tout de même noter que l’API renvoie chaque champ texte tel que : "Jean" devient "Jean MODIFIED". L’enum pose à nouveau problème, et le type de Phone est donc null à la désérialisation.

### Protobuf

Tout d’abord, nous avons créé des méthodes (createProtobufPerson, createProtobufPhone) dans les classes **Phone** et **Person** afin de créer des objets de la classe générée automatiquement (DirectoryOuterClass) depuis nos objets à nous. Ces objets « protobuf person » seront ensuite encapsulés dans un **Directory**, puis transformés en ByteArray (via creatingByteArrayForProtobufData) et utilisés pour être envoyés au serveur.

Le même concept a été utilisé pour lire les valeurs retournées depuis le serveur. Nous avons tout d’abord utilisé la méthode **parsingDirectoryByteArrayData** pour créer parsée du **ByteArray** en **Directory**, qui pourra ensuite être transformé en String grâce à la méthode **protobufDirectoryToString**. Le String produit sera finalement utilisé pour afficher la réponse sur l’UI.

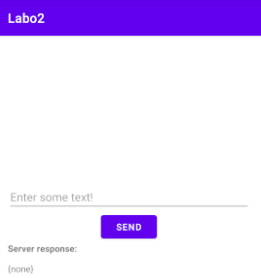
Dans la classe **Phone**, nous avons créé des méthodes afin de faire les correspondances entre les types Enum de la classe générée automatiquement, le type Enum de la classe **Phone** et le type **String**.

## Compressé

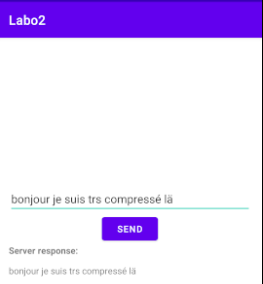
La mise en place de la communication compressée est sensiblement proche des premières manipulations (sauf celle de sérialisation).

### Utilisation

Au lancement le layout ressemble à ceci :



Le fonctionnement est similaire aux premières manipulations (sauf la sérialisée).



### Mise en place

Premièrement nous avons modifié les requêtes afin de pouvoir indiquer si ces dernières étaient compressées ou non. Pour ceci, nous avons ajouté l’attribut booléen **isCompressed** indiquant si la requête a pour but d’être compressée ou non avons l’envoi.Cette modification a affecté les 3 classes suivantes :

* **SymComRequest**
* **SymComStringRequest**
* **SymComBytesRequest**

Finalement, nous avons également dû adapter notre **SymComThread** afin qu’il supporte le mode compressé. Comme les requêtes HTTP compressées nécessitent des entêtes supplémentaires et des streams différents, nous avons fait en sorte que les streams et les entêtes supplémentaires soient affectés en fonction du type de requête.

Voici les entêtes ajoutés :

* connection.setRequestProperty("X-Network", "CSD")
* connection.setRequestProperty("X-Content-Encoding", "deflate")

Voici les streams utilisés :

* InflaterInputStream(connection.inputStream, Inflater(true))
* DeflaterOutputStream(connection.outputStream, Deflater(Deflater.DEFAULT\_COMPRESSION, true))

Ce qui est agréable avec notre infrastructure, c’est qu’il est possible de transformer toutes les activités / requêtes en format compressé très facilement (pratique pour la question 6 😉).

## GraphQL

### Utilisation

Cette activité permet de choisir un auteur dans un spinner en haut de la page. Ensuite, en fonction de l’auteur sélectionné, elle affiche les livres associés dans une *RecyclerView*.



### Mise en place

En termes de communication, cette activité n’a pas demandé de modification de l’infrastructure.

Les requêtes envoyées sont du simple JSON avec la méthode POST. Ces points ont déjà été traités lors de la manipulation d’objet au format JSON.

L’activité utilisée s’appelle *GraphqlActivity.* Elle tire parti du système de redéfinition de *CommunicationEventListener* exposé lors de la manipulation retardée.

Au démarrage, l’activité crée un premier *CommunicationEventListener* chargé de peupler le spinner grâce à la requête.

"{"query":"{findAllAuthors{id, name}}"}"

D’entrée, nous récupérons tous les 2000 auteurs. Nous récupérons leur identifiant ainsi que leur nom. En sachant que les int sont représentés sur 32 bits et que les caractères sont sur 16 bits, nous estimons les string long de 14 caractères en moyenne. Nous estimons la taille mémoire nécessaire à :

2000 \* (32 + 16 \* 14 ) / 8 = 64 000 bytes = 64 KB

Sachant que la heap size supporte plusieurs MB, nous avons accepté d’échanger de la mémoire pour diminuer les communications. De ce fait, il peut arriver que le chargement prenne plus ou moins de temps. C’est pourquoi nous notifions l’utilisateur avec un toast au démarrage et à la fin du chargement. Si la liste devait s’agrandir, il faudrait envisager d’implémenter une barre de recherche plutôt qu’un spinner.

Sur chaque entrée du spinner, nous avons placé un événement click qui se charge de repeupler le *RecyclerView* des livres. Contrairement aux auteurs qui sont tous chargés, les livres sont chargés à la volée et perdu lorsque la liste et repeuplée. La requête utilisée est la suivante :

"{"query\": "{findAuthorById(id: ${author.id}){books{title}}}"}"

Nous ne récupérons que les titres des livres de l’auteur en associé à la requête.

Ainsi à chaque requête, nous récupérons que ce dont nous avons besoin et nous évitons l’over-fetching et l’under-fetching.

Comme nous avons utilisé un *RecyclerView,* nous avons mis en place un adapteur *StringListAdapter* personnalisé qui s’occupe d’une liste de Strings (liste de titres). Nous associons à chaque élément une *TextView* définie dans le layout *recyclerview\_item.xml*.

# Réponse aux questions

## Traitement des erreurs

**Que se passe-t-il si le serveur n’est pas joignable dans l’immédiat ou s’il retourne un code HTTP d’erreur ?**

Si le serveur n’est pas joignable ou s’il retourne un code d’erreur, la requête est perdue. Comme chaque exception potentielle est immédiatement catchée et non traitée, le *SymComThread* est sans effet. Étant donné que nous travaillons dans un thread séparé, nous ne pouvons pas directement agir sur le UI thread pour afficher une erreur.

En conséquence, nous pourrions ajouter une méthode : *fun handleServerError(error : String, request SymComRequest)* dans le *CommunicationEventListener.* Le *SymComThread* pourra ainsi appeler cette méthode en cas d’échec.De cette façon, l’activité peut définir un comportement à adopter lorsqu’une requête échoue : affichage d’un toast pour signaler l’erreur reçue ou même temporiser et tenter un nouvel envoi dans quelques secondes.

## Authentification

**Si une authentification par le serveur est requise, peut-on utiliser un protocole asynchrone ? Quelles seraient les restrictions ?**

Oui, cela fonctionnerait. Cependant, il n'est pas judicieux d'utiliser ce processus. En effet, lors d'un login, que ferions-nous de l'utilisateur le temps que la réponse ne revienne ?

1. Si on attend la réponse, le protocole devient synchrone.
2. Si on redirige sur la page principale, puis on effectue une redirection quand le login est validé. Il est évident que l’utilisateur n’aurait pas une expérience optimale.
3. Si on donne l'accès protégé à l'utilisateur, sans connaitre la réponse. Il faudrait donc bien définir quelles données sont sensibles ou pas, mais le risque reste tout de même élevé au niveau de la sécurité .

Il serait cependant possible d'imaginer une application qui utilise la méthode du point 2, mais qui met en place un système limité tant que la réponse du login n'a pas été reçue. Par exemple, une simple consultation des données publiques.

On voit très rapidement les restrictions. Une application où l'utilisation d'un compte est centrale, comme n'importe quel réseau social, pourrait utiliser la méthode asynchrone, mais ce n'est probablement pas la bonne méthode.

**Peut-on utiliser une transmission différée ?**

Il est également possible d'utiliser une transmission différée, mais les problèmes cités pour l'asynchrone ci-dessus restent.

Dans le cas du protocole asynchrone ou de la transmission différée, si la question "Que faire de l'utilisateur pendant l'attente ?" se fait ressentir, il est probablement judicieux d'utiliser le protocole synchrone.

## Threads concurrents

**Lors de l'utilisation de protocoles asynchrones, c'est généralement deux threads différents qui se répartissent les différentes étapes (préparation, envoi, réception et traitement des données) de la communication. Quels problèmes cela peut-il poser ?**

Procéder avec des threads qui communiquent risque de générer des fuites de mémoire. Typiquement l’activité lance une requête dans un autre thread qui conserve une référence sur cette même activité. De cette façon, l’activité ne peut pas être libérée tant que thread conserve cette référence. Il est donc très important d’éviter les classes anonymes (car elle conserve une référence sur l’objet englobant) et de travailler avec des références faibles pour limiter le risque de fuite mémoire. Par exemple, dans notre *class SymComThread*, nous conservons une référence faible sur le *CommunicationEventListener* à notifier.

De plus, dans notre application, seul le UI Thread est capable de mettre la vue à jour. Si de nombreux threads tentent de transmettre des informations au UI thread pour qu’il se mette à jour, le UI thread risque momentanément de perdre en performance et devoir sauter quelques frames.

Finalement l’ordre de la réception des réponses peut ne pas être le même que l’ordre des envois. Dans certaines situations cela peut être non désiré. Nous pourrions imaginer un système ou les requêtes reçoivent un identifiant unique auto incrémenté qui sera transmis avec la réponse ce qui permettrait d’ordonner l’ordre du traitement des réponses. Une réponse serait « cachée » tant que toutes les réponses d’identifiant inférieur ne sont pas reçues (penser gérer correctement les requêtes qui auraient échoué),

## Écriture différée

**Comparer les deux techniques et discuter des avantages et inconvénients respectifs.**

Effectuer une connexion par transmission différée :

Le premier avantage est de ne pas envoyer toutes les requêtes en une seule. De ce fait si, une requête n’atteint pas le serveur uniquement celle-ci est retransmise. Ensuite, sitôt qu’une requête est traitée nous obtenons directement la réponse du serveur, pas besoin d’attendre que toutes les autres soient traitées.

Au contraire, effectuer une connexion par transmission différée peut potentiellement surcharger le serveur. De plus, si les requêtes doivent être traitées séquentiellement (par le client ou même le serveur) il faut mettre en place un moyen de les ordonner (voir question précédente avec les identifiants)

Multiplexer toutes les connexions vers un même serveur en une seule connexion de transport :

En réponse au problème de l’ordre du traitement des requêtes, multiplexer toutes les requêtes dans une seule permettrait de pallier ce souci. Le serveur traiterait les requêtes séquentiellement et concaténerait les réponses de la même manière. Dans la requête, les demandes sont ajoutées séquentiellement dans l’ordre dans lequel elles doivent être traitées. Le serveur parcourt et traite cette liste de requêtes dans l’ordre. Lorsqu’une réponse est nécessaire, il l’insère dans une liste qu’il ajoutera lors de la réponse au client.

Cependant, lorsque la transmission de la requête échoue, cela implique que toutes les demandes sont retransmises. Donc, comme la charge est potentiellement lourde, cela peut poser des problèmes de performances avec de faibles connexions. Si le réseau est faible, nous risquons d’avoir des requêtes qui n’aboutissent pas et pendant ce temps, d’autres demandes sont émises par l’utilisateur ce qui augmente la charge en entrainant un cercle vicieux. De plus, nous obtenons la réponse uniquement lorsque toutes les requêtes encapsulées auront été traitées.

## Transmission d’objets

1. **Quel inconvénient y a-t-il à utiliser une infrastructure de type REST/JSON n'offrant aucun service de validation (DTD, XML-schéma, WSDL) par rapport à une infrastructure comme SOAP offrant ces possibilités ? Est-ce qu’il y a en revanche des avantages que vous pouvez citer ?**

Voici une liste d’inconvénients d’utiliser une infrastructure REST/JSON :

* Le client ne peut pas vérifier que les données qu’il enverra au serveur sont valides / dans un format valide. Il est obligé de tester et vérifier que c’était bien le cas.
* Il est difficile pour le serveur d’interpréter des données qui ne sont pas structurées comme souhaité, même si le contenu est correct.

Voici les avantages que l’on peut trouver à ce même type d’infrastructure :

* Les transactions seront concises et par ce fait moins lourdes et plus rapides à traiter
* Il est beaucoup plus facile d’implémenter un protocole sans validation pour l’utilisateur si la structure est connue ou indiquée auparavant
  + Cela évite également les requêtes inutiles de validation
* L’infrastructure est également plus flexible et peut évoluer plus facilement

1. **Par rapport à l’API *GraphQL* mise à disposition pour ce laboratoire. Avez-vous constaté des points qui pourraient être améliorés pour une utilisation mobile ? Veuillez en discuter, vous pouvez élargir votre réflexion à une problématique plus large que la manipulation effectuée.**

Le nombre d’auteurs étant actuellement raisonnable, cela ne pose pas de soucis de renvoyer tous les auteurs et tous les livres dans la même requête. Il serait cependant très embêtant de récupérer tous les auteurs si ceux-ci atteignaient un nombre très important. Les transactions deviendraient longues et grosses en termes de volumes de données, ce qui utiliserait beaucoup de batteries.

Pour pallier ce problème, il serait intéressant de pouvoir indiquer le nombre d’auteurs ou de livres nous souhaiterions récupérer lors d’une requête.

Cela pourrait ressembler à quelque chose comme ceci :

*{query : getSomeAutors(nbAutors : Int !){[id name]}}*

En recevant, une liste plus courte d’auteurs, il serait ensuite plus facile d’implémenter un système de pagination au niveau du client. Le client enverrait uniquement une nouvelle requête s’il souhaitait voir afficher une seconde page. En connaissant les utilisateurs qui ne regardent pas souvent plus loin que la première page, cela éviterait de récupérer beaucoup de données. Cela apporterait :

* Un gain de temps pour le client et le serveur
* Une économie de batterie
* Des échanges plus légers

Dans la même idée, on pourrait également proposer des requêtes impliquant de filtre au niveau des livres et des auteurs. On souhaite souvent afficher uniquement les livres ou auteurs commençant par une certaine lettre de l’alphabet.

Cela pourrait ressembler à quelque chose comme ça :

*{query : getAutorsStartingWith(letter : Char !){[id name]}}*

Cela apporterait les mêmes bénéfices énumérés plus haut. On pourrait imaginer un certain nombre de filtres intéressants. On pourrait même penser à combiner les 2 principes (pagination et filtres).

On pourrait également imaginer une méthode de type recherche pour rechercher le nom d’un livre ou d’auteur spécifique.

## Transmission compressée

Pour tester la compression, nous avons effectué les tests en TEXT, JSON et XML. Nous avons rempli les champs avec une génération de Lorem Ipsum. Voici le paragraphe utilise :

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Non blandit massa enim nec dui nunc mattis enim ut. Sollicitudin aliquam ultrices sagittis orci a scelerisque. Lacus sed viverra tellus in hac habitasse. Tempor orci eu lobortis elementum nibh tellus molestie. Ipsum a arcu cursus vitae congue mauris rhoncus aenean. Proin sagittis nisl rhoncus mattis rhoncus urna. Quis vel eros donec ac odio. Sit amet purus gravida quis. Vitae elementum curabitur vitae nunc sed velit dignissim sodales. Viverra nam libero justo laoreet sit amet.

Nous avons obtenu les données suivantes :

|  |  |  |  |
| --- | --- | --- | --- |
|  | Avant compression | Après compression | Compression |
| Texte | 758 | 395 | 47.89% |
| 3806 | 1364 | 64.16% |
| 6450 | 1978 | 69.33% |
| XML | 823 | 295 | 64.16% |
| 2583 | 590 | 77.16% |
| 7107 | 649 | 90.87% |
| JSON | 674 | 220 | 67.36% |
| 2432 | 508 | 79.11% |
| 6951 | 572 | 91.77% |

On peut observer un taux de compression moyen de ~60% pour le texte, ~77% pour le XML et ~79% pour le JSON. Le taux moyen global est de ~72%. Ces valeurs paraissent très élevées. Nous pensons qu’il s’agit du fait que nous ayons utilisé un même paragraphe de Lorem Ipsum plusieurs fois lors de nos tests. De ce fait, le texte est répété et donc potentiellement plus facile à compresser.