Berney Alec & Forestier Quentin & Herzig Melvyn

SYM – 06.11.2021

Protocoles applicatifs

Labo 2



Table des matières

[Introduction 2](#_Toc87108560)

[Démarrage 2](#_Toc87108561)

[Remarques préalables 2](#_Toc87108562)

[Manipulations 3](#_Toc87108563)

[Asynchrone 3](#_Toc87108564)

[Utilisation 3](#_Toc87108565)

[Mise en place 3](#_Toc87108566)

[Retardé 4](#_Toc87108567)

[Utilisation 4](#_Toc87108568)

[Mise en place 5](#_Toc87108569)

[Sérialisé 5](#_Toc87108570)

[Compressé 5](#_Toc87108571)

[GraphQL 5](#_Toc87108572)

# Introduction

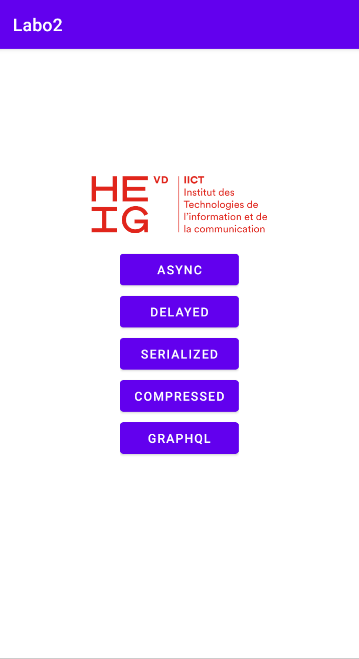
Ce laboratoire est une introduction aux techniques de programmation réparties asynchrones. Nous allons illustrer l’utilisation de cinq techniques de protocoles asynchrones sur plateforme mobile.

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 associées.

## Remarques préalables

Ce laboratoire est une introduction à la programmation asynchrone. En conséquence, certain aspect 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 mémoire. 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

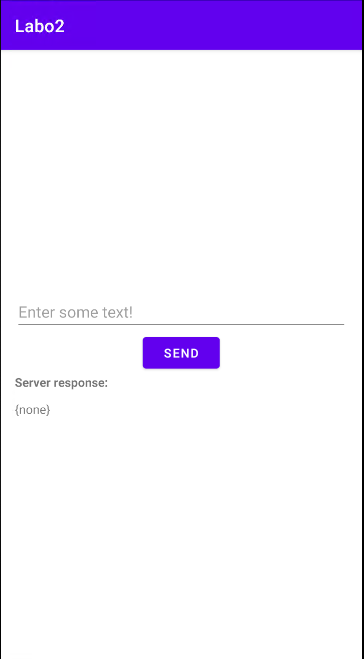
Comme présenté préalablement, ce laboratoire consiste en 5 manipulations majeures. Nous avons donc réalisé une activité par manipulation. Chaque activité, à la création, démarre son objet *SymComManager* qui est responsable de négocier les communications. Nous avons réussi à adapter le *SymComManager* 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 transmission de ce laboratoire sont effectuée 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 créer 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 principale

*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*

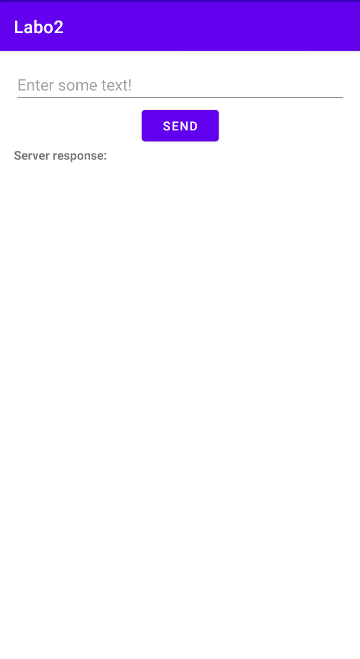
Finalement nous avons créé la classe **SymComThread** qui permet d’exécuter une requête *SymComRequest* sur autre thread que l’UI thread. Il est construit à partir d’une SymComRequest et d’une référence faible sur un *CommunicationEventListener.* Une fois la réponse reçue, le *SymComThread* retourne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

## Retardé

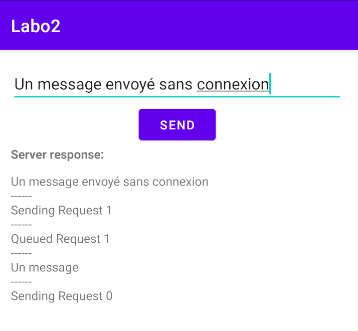
La mise en place de la communication retardée est sensiblement proche de la première manipulation.

### 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 message

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

L’envoie 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 paire 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 .

## Sérialisé

## Compressé

## GraphQL