Développement mobile avancé
Laboratoire n°1

Protocoles applicatifs mobiles

18.02.2025

Introduction

Ce laboratoire propose d'illustrer plusieurs familles de protocoles applicatifs mobiles, en particulier nous allons : 1) explorer différentes manières de sérialiser, compresser et multiplexer des échanges de données avec un endpoint *REST* ; 2) mettre en place un échange basé sur le protocole *GraphQL* et 3) se familiariser avec un service de *push* de messages, en particulier *Firebase Cloud Messaging*. Ce laboratoire vous permettra de réaliser trois manipulations permettant de mettre en œuvre ces différents types de fonctionnalités sur une application *Android*, ainsi que de répondre à des questions théoriques en lien avec les manipulations.

Environnement

Pour ce laboratoire nous vous fournissons plusieurs éléments, en particulier :

- Le squelette d'une application que vous compléterez pour les différentes manipulations ;
- Un serveur applicatif, accessible sur https://mobile.iict.ch/, qui offre une API *REST* https://mobile.iict.ch/api avec laquelle vous échangerez des données ainsi qu'une console permettant d'avoir un retour sur les requêtes traitées par le serveur. Pour les échanges au format *Protocol Buffers*, nous vous fournissons le fichier *.proto* ;
- Le serveur met également à disposition un endpoint spécifique permettant d'effectuer des requêtes *GraphQl*, https://mobile.iict.ch/graphql. Une documentation présentant le protocole et les données vous est fournie dans l'Annexe 1 ;
- Pour l'envoi de message *push*, nous vous mettons à disposition un outil, sous la forme d'un *jar* exécutable, permettant de vous faciliter l'authentification auprès des services de *Google* et l'utilisation de *Firebase* pour l'envoi de messages *push*, veuillez consulter l'Annexe 2.

Manipulations

1. Echange d'objets avec le serveur

Cette première manipulation vous propose de mettre en place un protocole applicatif permettant d'échanger des objets avec un serveur distant. Les objets échangés correspondent à des séries de

mesures (température, humidité, pression atmosphérique, précipitations) réalisées sur le mobile, on souhaite les faire parvenir au serveur. Le serveur va vérifier chacune de ces mesures par rapport à une plage de valeurs plausibles et vous donnera en retour un nouveau statut (« OK » ou « ERROR ») pour chacune d'entre elles. La Fig. 1 vous présente un exemple d'échange avec le serveur au format *json*, les mesures sont sérialisées en *json* et envoyées, en les multiplexant à plusieurs par requête, au serveur. Le serveur retourne un tableau de messages d'*acknowledgments* indiquant, pour chaque mesure identifiée par son *id*, son nouveau *statut*.

Requête

```
POST mobile.iict.ch/api
Content-Type: application/json;charset=UTF-8

[{
    "id": 23, "type": "HUMIDITY",
    "status": "NEW", "value": 48.4,
    "date": "2023-01-23T12:08:56.235+01:00"
 },{
    "id": 24, "type": "TEMPERATURE",
    "status": "NEW", "value": 125.3,
    "date": "2023-01-23T12:09:12.125+01:00"
}]
```

Réponse

```
200 OK
Content-Type: application/json;charset=UTF-8

[{
    "id": 23,
    "status": "OK"
 },{
    "id": 24,
    "status": "ERROR"
}]
```

Figure 1 - Exemple d'un échange http/json avec le serveur applicatif (tous les en-têtes ne sont pas représentés)

Le squelette de l'application fournie, sous la forme d'un projet *Android Studio* est composée de trois *Fragments* accessibles au travers d'une navigation à onglets, chaque *Fragment* représente une des trois manipulations de ce laboratoire. Nous allons nous intéresser dans un premier temps à l'onglet « Measures » qui correspond au *SendFragment*, la Fig. 2 représente visuellement le *Fragment* fourni.

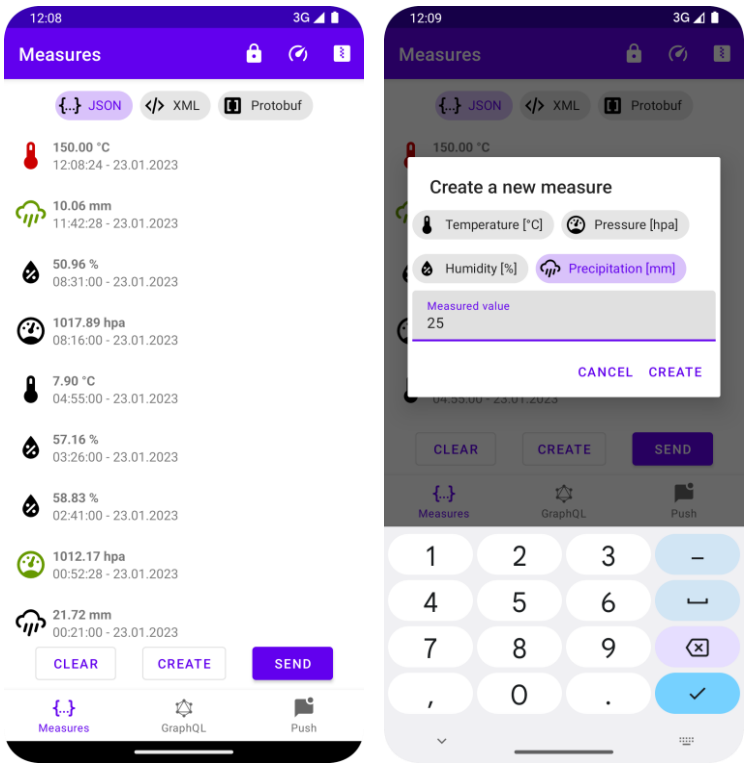

Figure 2 – Visuel du Fragment permettant la création et l'envoi de mesures vers le serveur

Vous noterez en particulier la présence de trois entrées dans le menu, celles-ci permettent à l'utilisateur de choisir l'utilisation d'un endpoint chiffré ou non-chiffré, de sélectionner une vitesse de transfert (simulation par le serveur du débit des différentes technologies de réseaux mobiles), ou encore de compresser ou non les échanges. Juste en dessous, on retrouve trois « Chips » (widget *material* permettant de faire une sélection), ceux-ci permettent de choisir la méthode de sérialisation des données à utiliser pour communiquer avec le serveur : *json*, *xml* ou, *protobuf*. Le corps du *Fragment* représente la liste des mesures stockées en mémoire, la couleur de leur icône est liée à leur statut. Dans la partie inférieure, nous trouvons trois boutons permettant de : supprimer les données en mémoire ; créer (clic simple) une nouvelle mesure ; créer (clic long) plusieurs nouvelles mesures et finalement d'envoyer la liste de mesures au serveur, en utilisant les paramètres sélectionnés par l'utilisateur (type de sérialisation, chiffrement, compression et débit simulés).

Le serveur offre un unique endpoint, http(s)://mobile.iict.ch/api acceptant sur une requête *POST* les trois types de sérialisation avec les différentes options en fonction des en-têtes reçues :

- *Content-Type* (soit *application/json*, *application/xml*, ou *application/protobuf*) **obligatoire**
  Indication du format des données échangées, le serveur va répondre avec le même format de sérialisation, qu'il précisera dans les en-têtes de la réponse
- *X-Network* (*CSD*, *GPRS*, *EDGE*, *UMTS*, *HSPA*, *LTE*, ou *NR5G*)
  Il est possible de préciser le débit de la connexion (simulé par le serveur), si le choix n'est pas précisé dans la requête, le serveur sélectionnera aléatoirement une catégorie de débit et la précisera dans sa réponse
- *X-Content-Encoding* (*NONE*, ou *DEFLATE*)
  Si l'en-tête est absente ou définie à NONE, le serveur traitera les données sans les décompresser. La réponse à une requête compressée sera également compressée
- *X-Error*
  Utilisé par le serveur pour communiquer d'éventuelles erreurs

Dans cette manipulation nous vous demandons de mettre en place les coroutines nécessaires pour assurer l'échange (requêtes et réponses) avec le serveur. Normalement vous avez uniquement la classe *MeasuresRepository.kt* à devoir modifier, en particulier dans le bloc *measureTimeMillis{}* de la méthode `sendMeasureToServer()`. Cette méthode est appelée avec tous les choix de l'utilisateur (chiffrement, compression, type de réseau et mode de sérialisation), votre implémentation devra être en mesure d'effectuer la requête et de traiter la réponse en fonction de tous les choix. Le serveur va retourner un tableau d'acknowledgments (id, statut) que vous devrez répercuter sur la liste locale de mesures.

Astuce : Avec la nouvelle infrastructure réseau de l'école les adresses IP affichées sont génériques. Pour filtrer vos requêtes sur l'interface du serveur, nous vous conseillons de modifier l'*user-agent* :

```
connection.setRequestProperty("User-Agent", "Labo_G01")
```

1.1. Format json

Vous mettrez en place la sérialisation et désérialisation des objets suivant la syntaxe présentée dans la Fig. 1, la requête et la réponse englobent un tableau de zéro, un ou plusieurs objets au format *json*.

1.2. Format XML

La sérialisation des données au format *XML* devra suivre la *DTD* disponible sur le serveur à l'adresse https://mobile.iict.ch/measures.dtd, le format des données sera vérifié par le serveur et une erreur

sera retournée si la *DTD* est absente ou n'est pas respectée. La Fig. 3 représente un exemple d'échange au format *xml* avec le serveur.

Nous vous conseillons d'utiliser la librairie *jdom2*[1] pour la sérialisation et désérialisation au format *xml*. Celle-ci permet d'ajouter assez facilement une *DTD* au document, cf. `DocType`.

Requête

```
POST mobile.iict.ch/api
Content-Type: application/xml;charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE measures SYSTEM "https://mobile.iict.ch/measures.dtd">
<measures>
    <measure id="23" status="NEW">
        <type>HUMIDITY</type>
        <value>48.4</value>
        <date>2024-01-23T12:08:56.235+01:00</date>
    </measure>
    <measure id="24" status="NEW">
        <type>TEMPERATURE</type>
        <value>125.3</value>
        <date>2024-01-23T12:09:12.125+01:00</date>
    </measure>
</measures>
```

Réponse

```
200 OK
Content-Type: application/xml;charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<measures>
    <measure id="23" status="OK" />
    <measure id="24" status="ERROR" />
</measures>
```

Figure 3- Exemple d'un échange http/xml avec le serveur applicatif (tous les en-têtes ne sont pas représentés)

1.3. Format Protocol Buffers

La dernière partie de cette manipulation concerne l'utilisation du format *Protocol Buffers* pour la sérialisation des données. Le format des messages échangés est défini dans le fichier *measures.proto*, celui-ci pourra être utilisé afin de générer automatiquement le code *Kotlin* et *Java* nécessaire à la sérialisation et la désérialisation.

Astuces : Il faut utiliser la version compilée correspondant à votre système d'exploitation, par exemple *protoc-**x.y**-win64.zip*, disponible sur https://github.com/protocolbuffers/protobuf/releases/. Le code *Kotlin* et *Java* se génère avec la commande suivante (*kotlin_dir* est le dossier de destination du code généré) :

protoc.exe --java_out=kotlin_dir --kotlin_out=kotlin_dir measures.proto

[1] http://www.jdom.org/, disponible sur maven https://mvnrepository.com/artifact/org.jdom/jdom2

Actuellement, pour utiliser *protobuf* en *Kotlin*, il est nécessaire de générer également le code *Java*. Seuls les formats de messages sont générés en *Kotlin*, le code s'occupant de la sérialisation et de la sérialisation reste en *Java*.

Lorsque vous mettrez le code dans votre application, il faudra également ajouter la dépendance suivante, 3.**X.Y** doit être la même version que celle utilisée pour générer le code :

```
implementation("com.google.protobuf:protobuf-kotlin:3.X.Y")
```

Bien qu'il existe un plugin pour *Gradle*[2], dans le cadre de ce laboratoire nous vous conseillons plutôt de générer le code comme décrit ci-dessus, la configuration du plugin vous prendra trop de temps pour une unique génération de code.

1.4. Transmission compressée

Vous aurez certainement remarqué lors des points précédents que si vous échangiez une quantité importante de données, certaines requêtes pouvaient mettre du temps à se terminer. Le serveur simule aléatoirement une vitesse de réception et d'envoi variant entre la 2G et la 5G, le mode utilisé est indiqué dans l'en-tête `X-Network` de la réponse reçue. Pour cette partie vous forcerez l'utilisation du mode le plus lent en ajoutant l'en-tête `X-Network: CSD` aux requêtes que vous enverrez (vous pouvez le faire depuis le menu de l'app fournie).

Le protocole *HTTP* ne prévoit pas nativement l'envoi de contenu compressé dans les requêtes, car selon le modèle historique, le client (un navigateur web) produit uniquement de petites requêtes qui peuvent déboucher sur un contenu plus important envoyé en réponse depuis le serveur, qui elle peut être compressée. Ce modèle ne s'applique pas bien aux applications modernes, en particulier mobiles. Notre serveur supporte le mode *deflate* (quel que soit le format de sérialisation utilisé), vous devez ajouter l'en-tête `X-Content-Encoding: deflate` et compresser le corps de votre *POST* avec un `DeflaterOutputStream` (package `java.util.zip`), le contenu que le serveur vous retournera sera aussi accompagné de l'en-tête `X-Content-Encoding` correspondante et vous devrez décompresser le contenu à l'aide d'un `InflaterInputStream` avant de pouvoir le désérialiser.

Attention, le contenu compressé ne devra pas être wrappé avec les en-têtes et le checksum ZLIB[3]

1.5. Question théorique d'approfondissement

Quel gain, en volume et en temps, peut-on constater en moyenne sur les données échangées (*xml*, *json* et aussi *protobuf*) en utilisant la compression mise en place au point 1.4 ? Vous comparerez plusieurs tailles de contenu, plusieurs vitesses de transmission et plusieurs méthodes de sérialisation. Vous pouvez vous aider des valeurs « Received Size » (taille en bytes du contenu transféré) et «Payload Size» (taille en bytes du contenu après décompression) indiquées dans l'interface de logs du serveur. Est-ce utile de compresser dans tous les cas ? Veuillez élaborer votre réponse avec des chiffres et des exemples.

2. Requêtes au format *GraphQL*

L'endpoint http://mobile.iict.ch/graphql accepte que vous lui postiez une requête *GraphQL* au format *json* (Content-Type : application/json). Vous trouverez dans l'annexe *API – GraphQL* une petite documentation sur ce endpoint, son utilisation et le format des données.

[2] Protobuf Plugin for Gradle : https://github.com/google/protobuf-gradle-plugin
[3] Voir : https://developer.android.com/reference/java/util/zip/Deflater.html#Deflater(int,%20boolean)

Pour cette manipulation nous vous fournissons un *Fragment* composé d'un *Spinner* qui accueillera la liste des auteurs ; lors de la sélection d'un auteur sur le *Spinner*, la *RecyclerView* affichera la liste de ses livres après l'avoir chargée depuis le serveur. Le code fourni est quasiment fonctionnel, vous devez uniquement mettre en place les méthodes permettant d'effectuer une requête sur le serveur et de mettre les données reçues dans les *MutableLiveData* *_authors* et *_books* disponibles dans la classe *GraphQLRepository*, en particulier en implémentant les méthodes `loadAllAuthorsList()` et `loadBooksFromAuthor(author: Author)`. Vous veillerez à éviter l'*under-fetching* et l'*over-fetching* en concevant vos requêtes. L'API mise à disposition ne permet pas la compression des échanges, ni la simulation de différents débits.

2.1. Question théorique d'approfondissement

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 en mettant en évidence les limitations de l'implémentation fournie, vous pouvez élargir votre réflexion à une problématique plus large que la manipulation effectuée.

3. Push de message (Firebase Cloud Messaging)

La dernière manipulation de ce laboratoire consiste à mettre en œuvre une fonctionnalité de *push* de message à l'aide du service *Firebase Cloud Messaging*. L'annexe 2 vous guidera pour la création d'un projet *Firebase*, son ajout à votre application *Android* ainsi que la création d'une clé d'authentification pour l'envoi des messages. Cette annexe documente également l'utilisation de l'outils *DMAMessagePusher.jar* fourni pour l'envoi des messages.

L'intégration de *Firebase Cloud Messaging* dans votre application *Android* se déroule principalement par la mise en place d'un *Service*, héritant de *FirebaseMessagingService*, qui sera appelé par le système lors de la réception d'un message destiné à notre app[4]. Votre service devra en particulier surcharger la méthode `onMessageReceived(message: RemoteMessage)`. Également ce *Service* devra très souvent surcharger la méthode `onNewToken(token: String)`, bien que celle-ci ne soit pas directement utilisée dans cette manipulation, elle est utile pour répondre à la question théorique ci-dessous. Veuillez dans un premier temps ajouter un message dans les logs à chaque fois que ces méthodes sont appelées. N'oubliez pas d'enregistrer votre *Service* dans le *Manifest* et d'ajouter l'*intent-filter*. Vous ajouterez également le code nécessaire dans l'*Activité* principale pour enregistrer votre app sur un ou plusieurs *topics*. Veuillez suivre les indications présentes dans l'Annexe 2 jusqu'à ce que les messages envoyés soient correctement reçus et affichés dans les logs de votre app.

Dans le squelette de l'application fournie, nous avons mis en place un onglet spécifique à cette manipulation, celui affiche le dernier message reçu à partir d'une base de données *Android Room*. Veuillez mettre en place dans votre *Service*, le stockage des messages reçus dans la base de données. Vous testerez que les messages sont bels et bien reçus, quel que soit l'état de votre app, *ouverte*, en *arrière-plan*, ou *fermée*. Vous essayerez également d'envoyer des messages lorsque le téléphone est éteint ou qu'il n'est pas connecté à *Internet*.

Nous souhaiterions également que vous puissiez mettre en place une fonctionnalité basique de *poll on request*, par exemple lors de la réception d'un message contenant la commande *clear*, l'application vide la base de données locales des messages. Vous pouvez changer les arguments de l'entité *Message* fournie si nécessaire.

[4] Documentation Firebase Messaging https://firebase.google.com/docs/cloud-messaging/android/client?hl=en

3.1. Question théorique d'approfondissement

Veuillez expliquer à quoi sert le *token* obtenu avec la méthode `onNewToken` du *Service Firebase* de réception des messages. Quand est-ce que ce *token* est généré ou regénéré, et qu'est-ce qu'une application doit faire avec celui-ci, vous illustrerez votre réponse en prenant l'exemple d'un service de messagerie, tel que *WhatsApp*. Si l'utilisateur dispose de son application de messagerie sur plusieurs appareils, par exemple sur son smartphone et sa tablette, comment doit-on gérer les *tokens* obtenus sur chaque device ?

Durée / Evaluation

- Durée de 8 périodes, à rendre le dimanche **16.03.2025** à **23h55** au plus tard.
- Pour rendre votre code, nous vous demandons de bien vouloir zipper votre projet Android Studio en veillant à bien supprimer les dossiers build (à la racine et dans app/) pour limiter la taille du rendu.
- Vous remettrez également un rapport au format **pdf** comportant au minimum vos explications sur la solution que vous proposez pour les 3 manipulations, celles-ci devront en particulier couvrir les différents points mentionnés dans la donnée.
- Merci de rendre votre travail sur *CyberLearn* dans un zip unique. N'oubliez pas d'indiquer vos noms dans le code, sur vos réponses et de commenter vos solutions.