Systèmes mobiles
Laboratoire n°2 : Protocoles applicatifs

06.10.2020

Ce laboratoire propose d'illustrer l'utilisation de protocoles de communication applicatifs, basés sur http, dans le cadre d'applications mobiles.

1. Introduction

Ce laboratoire entend proposer une introduction aux techniques de programmation réparties asynchrones. Beaucoup plus complexes à maîtriser que les techniques synchrones, la programmation asynchrone est connue surtout dans le monde des interfaces utilisateurs, bien que ces dernières abusent trop souvent de dialogues dits "modaux" pour éviter la complexité induite par les interactions asynchrones.
Dans le monde du développement web, historiquement les sites internet étaient basés sur un modèle synchrone : l'utilisateur attend que la page soit entièrement chargée avant de pouvoir interagir avec. Depuis environs une douzaines d'année, l'*AJAX* (asynchronous Javascript and XML) permet de réaliser des applications web, de plus en plus complexes, permettant de charger leur contenu de manière asynchrone. Sans pour autant que les spécificités du monde mobiles ne soient prises en compte (connexion temporairement indisponible, mauvais débit, gestion d'énergie, etc.). Dans ce laboratoire nous allons illustrer l'utilisation de différentes techniques de protocoles asynchrones pour du mobile.

1.1 Prérequis

- Serveur applicatif accessible sur http://sym.iict.ch/. Plusieurs services sont définis sur ce serveur, entre autre le plus simple (http://sym.iict.ch/rest/txt) qui implémente un service écho, vous lui POSTez du texte (Content-Type: text/plain) et il vous l'envoie en retour accompagné de certaines informations sur le serveur. Une console est accessible depuis la page http://sym.iict.ch/, celle-ci vous permettra de voir ce que le serveur reçoit lorsque vous faites une requête.

1.2 Protocole et service

Un protocole comporte toujours 2 aspects, comme le modèle OSI le montre de manière très claire :

- Les couches réseau, internet et transport : cet aspect n'est que peu maîtrisable par le développeur d'applications, puisqu'il dépend d'éléments n'étant pas sous contrôle.
- L'aspect application : dans ce cas, le développeur a beaucoup plus de choix (et donc de responsabilités) sur le comportement de son applicatif. Nous allons nous intéresser à ce dernier car il va permettre à notre application de fonctionner de manière parfaitement asynchrone relativement au protocole réseau, sur lequel nous n'avons que peu d'influence.

1.3 Fonctionnement en l'absence de réseau

Une application professionnelle doit, dans la mesure du possible, permettre un travail aussi en l'absence de réseau. On voit mal un médecin se trouver dans l'incapacité de dicter des notes ou une ordonnance pour un patient simplement parce qu'il se trouve dans une campagne trop isolée pour intéresser les opérateurs de télécommunications. De manière similaire, un fonctionnaire des douanes ne peut pas interrompre le déchargement d'un avion-cargo parce qu'une indisponibilité de réseau l'empêche de documenter les produits qui sont extraits des cales.

Le service applicatif se doit donc de comporter une fonction de cache efficace, sûre et aussi transparente que possible.

2. Le serveur

Le serveur est moins conditionné par cette notion de synchronisme ou asynchronisme du service, car généralement il n'a que cela à faire : attendre une communication et y répondre.

Là encore, cette vision des choses est peut-être un peu étroite. Elle se réfère à un modèle Web où la requête du client est de nature consultative. Je veux consulter le programme TV de ce soir, ou consulter les horaires de chemin de fer : une requête HTTP se traduit en une autre requête SQL, qui voit son résultat traduit en une page HTML ou de données (ex : JSON) que le client pourra analyser. Ce genre de requête, très simple, peut être aisément traitée par un serveur basé sur des langages de script, du type JavaScript ou php.

Lorsque le client devient producteur d'informations, le traitement par le serveur peut devenir non trivial, et corollairement, le temps nécessaire à la constitution de la réponse peut également devenir trop grand, surtout en cas de surcharge du serveur. La réponse ne venant pas, le client conclura à un time-out de la communication, ce qui aura pour effet de répéter la procédure quelque temps après, contribuant à la surcharge du serveur, etc...

L'asynchronisme du serveur devient alors un paramètre important à considérer dans le design applicatif ; mais cet asynchronisme n'est pas pour autant évident à implémenter. Deux difficultés se présentent :

- Le protocole HTTP est synchrone ; il faut donc répondre avant d'en avoir terminé avec le traitement effectif. Plusieurs options se présentent :
    - Valider la requête ("202 Accepted"/"400 Bad Request") avant d'en avoir effectué le traitement effectif et renvoyer cette réponse immédiatement, éventuellement accompagnée d'un numéro de ticket qui permettra au client de suivre ultérieurement l'avancement du traitement. Ceci est faisable dans la majorité des cas, pour autant que le protocole d'application soit conçu dans cette optique ; par exemple, une syntaxe XML s'appuyant sur une DTD ou un schéma permet de valider le document avant d'avoir traité activement le contenu.
    - L'application peut être conçue de manière à ne requérir aucune validation ; c'est souvent le cas pour les saisies simples (logging, par exemple) ou pour les applications de traçage semi-automatique. Dans ce cas, une réponse constante "202 Accepted" permet d'assurer le client que les données ont été reçues et de vider cette information de son cache.

- Les serveurs d'application traditionnels (PHP, par exemple) ne sont pas conçus pour le travail parallèle ; même s'il n'est pas impossible de faire du multitraitement quasi-parallèle en PHP, ce n'est pas simple pour autant, parce que l'infrastructure n'est pas conçue pour ce faire (pas de multi-threading natif). Il faut donc se rabattre vers des serveurs applicatifs plus sophistiqués, comme .NET ou JavaEE ; mais ces derniers sont souvent nettement plus complexes à mettre en œuvre, et beaucoup moins bien supportés par les hébergeurs de service Internet. De son côté, *Node.js*, n'a mis en place que très récemment une nouvelle architecture permettant de réaliser des tâches lourdes en arrière-plan[1].

2.1 Transmission d'objets

Dans une application répartie, il est souvent nécessaire d'échanger des objets complexes, pas seulement des textes. Transmettre un objet implique que l'instance de l'objet en question soit **sérialisable** (puisse être représentée sous forme d'une suite de bits). Lorsque le protocole applicatif se fonde sur HTTP (un protocole permettant principalement la transmission de texte), il est recommandé de convertir l'objet en un texte (qui lui est facilement sérialisable). Plusieurs techniques sont à notre disposition dans le cadre des services web, où ce type de transmission est majoritairement utilisé, on trouve par exemple SOAP, XML-RPC, GraphQL ou REST-JSON.

3. Manipulation

Pour réaliser cette manipulation, il faudra au minimum une petite application avec une première activité proposant 5 boutons permettant de lancer 5 activités indépendantes implémentant les 5 points suivants. Vous pouvez aussi bien entendu, si vous le désirez, profiter de ce laboratoire pour prendre en main un des templates par défaut d'applications mis à disposition par Android Studio (Bottom Navigation, Navigation Drawer, Tabbed, etc.).

La manipulation proposée ici implique une communication asynchrone avec un serveur, sur la base d'une méthode synchrone comme le protocole HTTP (à implémenter) qui aurait, par exemple, pour signature :

```
public String sendRequest(String url, String request) throws Exception
```

Attention ! De nombreux exemples que l'on peut trouver sur le web utilisent les classes de la librairie HttpComponents de apache (`HttpClient`, `HttpPost`, `HttpGet`, `HttpResponse`, etc.); ces classes sont dépréciées (*deprecated*) depuis Android 6.0 (API 23); l'envoi de requêtes HTTP doit dorénavant se faire en utilisant les classes standards du package `java.net` de type `HttpUrlConnection`.

Il existe aussi des librairies permettant de simplifier le développement de la communication HTTP, sur *Android,* comme par exemple *Volley*[2]. Mais dans le cadre de ce laboratoire, nous souhaitons mettre en avant l'asynchronisme et les difficultés associées, nous vous demandons donc de ne pas utiliser de librairie « clé-en-main ». Celles-ci permettent souvent de faciliter la mise en œuvre des cas « simples », mais dans les cas plus complexes ou sortant de l'ordinaire (par exemple la compression des requêtes sortantes) elles peuvent avoir tendance à compliquer le problème.

[1] https://www.jesuisundev.com/comprendre-les-worker-threads-de-nodejs/
[2] https://github.com/google/volley

3.1 Service de transmission asynchrone

On implémentera un service de transmission asynchrone (appelons par exemple cette classe `SymComManager`), qui proposera les deux méthodes suivantes :

- `public void sendRequest(String request, String url) throws Exception`
  Permet d'envoyer le texte `request` vers le serveur désigné par `url`

- `public void setCommunicationEventListener (CommunicationEventListener l)`
  Permet de définir un listener `l` qui sera invoqué lorsque la réponse parviendra au client

Le but est de pouvoir écrire, au niveau de l'application, quelque chose qui ressemble à ceci :

```
// On suppose que votre classe d'accès est nommée SymComManager
SymComManager mcm = new SymComManager();
mcm.setCommunicationEventListener(
      new CommunicationEventListener(){
            public boolean handleServerResponse(String response) {
                  // Code de traitement de la réponse – dans le UI-Thread
            }
      }
);
mcm.sendRequest(…,…);
```

Avec la méthode présentée ci-dessus, le terminal n'attend pas forcement une réponse de la part du serveur ; mais s'il y a une réponse, il est possible de la faire passer à l'activité ou au service à l'origine de l'envoi.

Note : Pour ce laboratoire nous vous demandons de gérer la communication entre les différents threads à l'aide du mécanisme de *Handler*[3], vu durant le cours. Nous n'allons pas utiliser l'approche traditionnelle des *AsyncTasks* celle-ci ayant été dépréciée cet été, ni celle des coroutines qui feront l'objet d'exercices ultérieurement.

3.2 Transmission différée

Dans le cadre de l'argumentaire utilisé pour justifier la transmission asynchrone, on se propose d'implémenter une fonctionnalité de transmission différée. En l'absence de connexion avec le serveur, l'application fonctionne normalement, sans que l'utilisateur n'éprouve une gêne quelconque. Dès que la connexion avec le serveur est rétablie, les informations qui avaient été fournies par l'utilisateur sont transmises au serveur. Il n'est pas nécessaire pour cette manipulation d'utiliser l'interface `CommunicationEventListener` utilisée au point précédent.

On admet dans le cas de notre manipulation qu'il n'est pas utile d'opérer la transmission aussitôt que le serveur devient atteignable, une tâche exécutée à intervalles réguliers est suffisante. Egalement vous pouvez rester sur une solution simple « en mémoire », vous indiquerez toutefois dans votre rapport les limitations de cette façon de faire et proposerez des outils et techniques mieux adaptés (sans forcément réaliser l'implémentation).

[3] https://developer.android.com/reference/android/os/Handler

3.3 Transmission d'objets

Transmettre des objets est souvent indispensable dans une application distribuée, et donc dans une application mobile ; nous allons utiliser à nouveau notre service de transmission pour envoyer un objet sérialisé sous forme de texte (xml ou json) vers le serveur, et récupérer l'information qu'il nous renvoie (serveur miroir, contenu envoyé accompagné de quelques informations sur le serveur) pour la restituer sous forme d'instance d'objet. Vous devez pour cette partie implémenter les étapes de sérialisation et de désérialisation.

3.3.1 Format JSON

Le serveur http://sym.iict.ch/rest/json accepte que vous lui postiez un contenu au format json (Content-Type : application/json) et vous le retourne. Vous êtes libre de définir les objets que vous souhaitez envoyer.

3.3.2 Format XML

Le serveur http://sym.iict.ch/rest/xml accepte que vous lui postiez un contenu au format xml (Content-Type : application/xml) et vous le retourne. Le serveur s'attend à recevoir une liste de personnes avec leurs numéros de téléphones (annuaire), les documents xml échangés seront donc vérifiés par la DTD disponible à l'adresse suivante http://sym.iict.ch/directory.dtd.

Le contenu minimum à envoyer au serveur est donc :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE directory SYSTEM "http://sym.iict.ch/directory.dtd">
<directory />
```

3.4 GraphQL – Format JSON

Le endpoint http://sym.iict.ch/api/graphql accepte que vous lui postiez une requête GraphQL au format JSON (Content-Type : application/json). Vous trouverez dans le document annexe *API – GraphQL* une petite documentation sur ce serveur et sur son utilisation.

Le but de cette manipulation est de réaliser une petite interface listant les auteurs disponibles en base de données, et lorsque l'utilisateur sélectionne un auteur, d'afficher la liste de ses postes. Vous veillerez à ne pas faire d'under-fetching ou d'over-fetching.

3.5 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 envoi variant entre la 2G et la 4G, 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.

Le protocole HTTP ne prévoit pas nativement l'envoi de contenu compressé, car selon le modèle historique, le client (navigateur web) produit uniquement de petites requêtes qui peuvent déboucher sur un contenu important envoyé en retour depuis le serveur, qui lui peut être compressé. Ce modèle ne s'applique pas bien aux applications mobiles. Notre serveur supporte le mode deflate (pour les méthodes d'accès utilisées précédemment : /text, /json et /xml), 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`. Attention, le contenu compressé ne devra pas être wrappé avec les en-têtes et le checksum ZLIB[4].

4. Questions

Veuillez répondre aux questions suivantes dans votre rapport. Pour celles-ci, il n'est pas nécessaire de rendre du code ou d'intégrer ces fonctionnalités à l'application que vous rendrez. Toutefois il vous sera certainement utile de faire un peu de code pour tester et mettre un extrait de code dans votre rapport peut vous aider à illustrer votre réponse.

4.1 Traitement des erreurs

Les classes et interfaces `SymComManager` et `CommunicationEventListener`, utilisées au point *3.1*, restent très (et certainement trop) simples pour être utilisables dans une vraie application : que se passe-t-il si le serveur n'est pas joignable dans l'immédiat ou s'il retourne un code HTTP d'erreur ? Veuillez proposer une nouvelle version, mieux adaptée, de ces deux classes / interfaces pour vous aider à illustrer votre réponse.

4.2 Authentification

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

4.3 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 ?

4.4 Ecriture différée

Lorsque l'on implémente l'écriture différée, il arrive que l'on ait soudainement plusieurs transmissions en attente qui deviennent possibles simultanément. Comment implémenter proprement cette situation ? Voici deux possibilités :

- Effectuer une connexion par transmission différée
- Multiplexer toutes les connexions vers un même serveur en une seule connexion de transport. Dans ce dernier cas, comment implémenter le protocole applicatif, quels avantages peut-on espérer de ce multiplexage, et surtout, comment doit-on planifier les réponses du serveur lorsque ces dernières s'avèrent nécessaires ?

Comparer les deux techniques (et éventuellement d'autres que vous pourriez imaginer) et discuter des avantages et inconvénients respectifs.

4.5 Transmission d'objets

a. 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 ?

[4] Voir : https://developer.android.com/reference/java/util/zip/Deflater.html#Deflater(int,%20boolean)

b. L'utilisation d'un mécanisme comme *Protocol Buffers*[5] est-elle compatible avec une architecture basée sur *HTTP* ? Veuillez discuter des éventuelles avantages ou limitations par rapport à un protocole basé sur JSON ou XML ?
c. 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.

4.6 Transmission compressée

Quel gain de compression (en volume et en temps) peut-on constater en moyenne sur des fichiers texte (xml et json sont aussi du texte) en utilisant de la compression du point 3.4 ? Vous comparerez plusieurs tailles et types de contenu.

5. Durée

- 10 périodes
- A rendre le dimanche **15.11.2020** à **23h55** au plus tard.

6. Rendu/Evaluation

Pour rendre votre code, nous vous demandons de bien vouloir zipper votre projet Android Studio, vous veillerez à bien supprimer les dossiers build (à la racine et dans app/) pour limiter la taille du rendu. En plus, vous remettrez un document **pdf** comportant au minimum les réponses aux questions posées.

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.

Bonne chance !

[5] https://developers.google.com/protocol-buffers/