Agrégation de canaux pour la communication directe sur mobile

Rayane Hamani

Agrégation de canaux pour la communication directe sur mobile

Rapport de Projet de Fin d'Études

Janvier-Février 2022

Auteur

Rayane Hamani

- Master 2 Informatique, spécialité Génie Logiciel, FST, Univ Lille
- rayane.hamani.etu@univ-lille.fr

Encadrants

Adrien Luxey

- Post-doctorant, équipe Spirals, laboratoire CRIStAL, Univ Lille
- adrien.luxey@inria.fr

Lauric Desauw

- Ingénieur de recherche, équipe Spirals, laboratoire CRIStAL, Univ Lille
- lauric.desauw@inria.fr

Sommaire

Introduction

Depuis l'aube des téléphones dits « intelligents » (avec le lancement de l'Apple iPhone 1 en juin 2007), jusqu'au dernier standard de communication cellulaire (dit « 5G », publié par la 3GPP en 2019), les partisans de l'informatique mobile promettent l'avènement imminent de la communication directe entre périphériques mobiles à proximité (en anglais *device-to-device communication*, abrégé D2D). La communication D2D permettrait en effet à de nouveaux services d'émerger : reconnaissant leurs environs, ces services seraient plus interactifs ; communiquant entre pairs, ils seraient plus collaboratifs et communautaires.
Néanmoins, nous sommes en 2022, et—plus que jamais—chacune de nos activités digitales s'effectue de manière centralisée, en ayant recours à cette cohorte bien physique de centres de données improprement appelée le nuage (*cloud*).

Il y a hélas de très bonnes raisons—et même urgentes—qui motivent le recours massif à la communication D2D dès que le cas d'usage le permet :

1. Réduire l'empreinte environnementale du numérique : Chaque connexion avec le *cloud* engendre un surcoût énergétique, une occupation de bande passante, et souvent un stockage de données sur des serveurs distants.
   Pour des usages tels qu'un partage de fichier entre particuliers, ces coûts sont tout simplement absents si l'on effectue l'échange directement entre les périphériques concernés.
2. Protéger la vie privée des usagers : De façon similaire, chaque connexion réseau laisse des traces (des métadonnées) qui sont simplement absentes dans le cas d'un échange D2D. Ces traces, bien qu'individuellement négligeables, sont le carburant du capitalisme de la surveillance, comme l'explique Shoshana Zuboff dans son livre « L'Âge du capitalisme de surveillance » (2020).

Il est grand temps de prendre le problème à la racine, en proposant des solutions logicielles permettant d'effectuer des tâches intrinsèquement locales de façon directe (sans recourir au *cloud*), et avec une bonne qualité de service. En particulier,

le projet que nous allons présenter aujourd'hui s'attaque au plus commun des usages : l'échange de document entre personnes physiquement proches.

Des solutions pour ce faire existent déjà sur les deux principaux systèmes d'exploitation (OS) mobiles : AirDrop sur iOS, et Nearby sur AndroidOS. Néanmoins, ces deux protocoles sont propriétaires : leur fonctionnement n'est pas publiquement documenté, et il est donc impossible pour ses usagers de savoir si la communication *via* ces protocoles s'effectue bien localement—sans recourir à un serveur tiers. En outre, ces deux protocoles ne sont pas inter-compatibles (impossible, donc d'échanger directement un fichier entre un appareil Android et un appareil sous iOS).

Le but de ce projet de fin d'études est de proposer un protocole similaire, ainsi qu'une application en démontrant l'usage. À terme, l'objectif est de publier, en licence libre, un protocole d'échange de fichiers D2D, qui sache utiliser tous les canaux de communication disponibles entre deux téléphones, de façon concurrente.
Notre plateforme cible est l'OS Android, car nous disposons de périphériques utilisant ce système. La contribution principale n'étant pas l'application, mais bien le protocole d'échange, ce choix n'entrave en rien la généralité du projet, puisqu'il sera toujours possible de développer une application utilisant le même protocole sur tout autre système d'exploitation.
Notre langage cible est Java, dont les API sont les plus proches du noyau Unix sur Android. En évitant ainsi les surcouches logicielles, notre code sera plus facilement généralisable à d'autres systèmes d'exploitation.

Nous ne cherchons pas à proposer une solution plus performante que ses homologues propriétaires (AirDrop & Nearby) ; nous cherchons plutôt une alternative à la fois transparente et conceptuellement simple. Transparente, car le protocole—qui sera publié une fois arrivé à maturité—n'emploie que des canaux de communication directs, par exemple Bluetooth, Wi-Fi ad hoc/Direct, NFC, et UWB. Conceptuellement simple, car l'application ne sera constituée que d'une interface graphique, et de canaux de communication dont l'usage est orchestré par un ordonnanceur.

Pour résumer : ce projet de fin d'études vise à proposer un protocole d'échange de fichiers reposant sur des canaux de communication directs entre périphériques mobiles, qui soit conceptuellement aussi simple que possible, et facilement implémentable sur tout système d'exploitation disposant d'API de communication sur lesdits canaux.

Tâches complétées

Ordonnanceur parent

L'ordonnancement des données est un point crucial de l'application. En effet, c'est ce qui va ultimement déterminer ses performances.

- D'un côté, le fichier d'échange va être sérialisé en nombreux petits fragments (ou « parties ») qui seront envoyés à travers plusieurs canaux. Ces canaux n'ont ni le même débit ni la même stabilité et l'adaptation de l'envoi des parties doit donc se faire en temps réel.
- De l'autre, le fichier d'échange va être reconstitué de part la réception des différents paquets.

Il y a donc deux tâches qui se distinguent : l'envoi et la réception. Afin de gérer ces tâches de façon optimale, deux ordonnanceurs ont été créés.

Ces ordonnanceurs partagent plusieurs choses en commun. Afin de regrouper leurs similarités et afin de mieux segmenter le code, une classe abstraite - parente de ces ordonnanceurs - a été créée.

Parmi les points communs des ordonnanceurs, on retrouve :

- Un état de fonctionnement (isInProgress)
- Une queue de message (messages)
- Une taille de chunk (CHUNK_SIZE)
- Une fonction de hash (checksumAlgorithm)
- Une base de données (database)
- etc…

L'état isInProgress représente si un fichier est en cours d'envoi / de réception. Lorsque c'est le cas, cela signifie que les ordonnanceurs sont déjà dans leur boucle d'exécution. A noter qu'ils ne s'occuperont que du fichier auquel ils sont attachés à ce moment-là et n'écouteront aucune demande de transmission pendant ce temps-là.

La queue messages est une queue de messages à traiter. L'ordonnanceur d'envoi n'attend que les accusés de réception de la part de l'autre appareil et l'ordonnanceur de réception n'attend que les fragments de fichier de la part de l'autre appareil.

La taille CHUNK_SIZE est une constante décrivant la taille d'un fragment de fichier. Les deux ordonnanceurs doivent manipuler des parties de même taille afin de faciliter l'échange et la reconstitution. L'échange, car sinon les canaux de réception ne sauront pas quelle taille de message lire. La reconstitution, car sinon l'ordonnanceur de réception aura des difficultés à déterminer où écrire les parties dans le fichier.

La fonction de hash checksumAlgorithm est une méthode utilisée pour déterminer la somme de contrôle en fin de message. Cette somme permet de valider qu'un message est arrivé sans erreur et peut être traité / est interprétable par un ordonnanceur.

La base de données database sert à garder une trace sur disque des parties restantes à envoyer / manquantes à recevoir. Cela permet également à l'ordonnanceur de réception d'être plus autonome et de savoir quand un fichier a été complètement reçu sans avoir besoin d'être notifié par l'ordonnanceur d'envoi.

Cette classe parent recense d'autres attributs et méthodes communs aux ordonnanceurs d'envoi et de réception. Elle possède aussi des prototypes de fonction comme la méthode execute qui est ensuite définie dans les classes des deux ordonnanceurs. Ceux-ci implémentent une interface Scheduler qui possède simplement la méthode execute des deux ordonnanceurs. Il s'agit donc de la boucle d'exécution des deux ordonnanceurs, celle qui s'occupera de faire les appels et les traitements qu'il faut.

Ordonnanceur d'envoi

L'ordonnanceur d'envoi est un des deux ordonnanceurs de gestion des données. Il est responsable des messages envoyés par chaque canal ainsi que de l'optimisation de leur queue de messages.

Lors de sa création, l'ordonnanceur d'envoi va informer l'ordonnanceur de réception de l'autre appareil qu'un fichier va être envoyé. Si celui-ci n'accuse pas réception dans les 30 secondes suivant la notification d'envoi, l'envoi est annulé et l'ordonnanceur se termine. Si celui-ci accuse réception, signifiant ainsi qu'il est prêt à recevoir, alors l'ordonnanceur d'envoi va former les messages contenant les parties et les donner aux différents canaux de communication. Dès lors qu'une partie a bien été reçue par le récepteur et que l'ordonnanceur d'envoi en a été informé, il supprime la partie de la base de données des parties à envoyer. **Ce comportement est à changer** mais ne devrait pas nécessiter beaucoup d'efforts à modifier. La fonction, dans PartDAO, de choix des parties à envoyer sera également à modifier en conséquence. Après avoir donné des messages à envoyer à un canal, l'ordonnanceur va traiter les accusés de réception qu'il aura reçu avant de redistribuer des messages à envoyer à un canal. Lorsque la base de données est vide et que toutes les parties ont été reçues par le récepteur, l'ordonnanceur d'envoi notifie l'autre appareil de la fin de la transmission puis se termine.

Au niveau de l'algorithme d'ordonnancement, celui-ci est inspiré de Selective Repeat ARQ. En effet, lorsqu'un message est envoyé, le canal qui l'a envoyé va le mettre en attente de son accusé de réception. Contrairement à Selective Repeat ARQ cependant, l'envoyeur n'attend aucun Non-ACK et si une partie n'est pas reçue par le receveur, elle devra impérativement attendre que toutes les autres parties soient envoyées pour être renvoyée. Si après cela le receveur n'a toujours pas accusé de réception cette partie, elle sera renvoyée une nouvelle fois et indéfiniment sans *timeout* jusqu'à réception.

L'ordonnanceur d'envoi optimise également la liste des messages à envoyer par les canaux. En effet, un canal qui a envoyé tous ses messages est en attente, ne fait

plus rien et ne contribue donc plus à l'application. Pour éviter ces temps morts, et aussi inversement pour ne pas surcharger un canal qui a du mal à tenir la cadence, la fonction optimiseNumberOfMessagesToSend va adapter le nombre de messages à envoyer dans un canal en fonction de ses performances. La fonction va regarder le nombre de messages envoyés par un canal, et en fonction de la quantité envoyée en pourcentage, va soit augmenter soit diminuer la quantité. Idéalement, le nombre de messages à envoyer devrait converger vers le nombre qu'est réellement capable d'envoyer un canal dans un laps de temps fixe. Les critères d'optimisation sont les suivants :

- Soit M le nombre total de messages envoyés par un canal en 1 seconde
- Si M < 75% de sa capacité totale, sa capacité totale /= 2
- Si M >= 100% de sa capacité totale, sa capacité totale *= 2
- Sinon sa capacité totale reste inchangée

Ordonnanceur de réception

L'ordonnanceur de réception est chargé de reconstruire le fichier envoyé par l'émetteur. Puisque ce fichier est envoyé par parties, l'ordonnanceur va :

- Créer le fichier avec la bonne taille sur l'appareil receveur,
- Écrire les parties aux bons endroits du fichier.

Avant d'écrire une quelconque partie dans le fichier, l'ordonnanceur attend un message de début de transmission. Ce message contient le nom et la taille du fichier, informations nécessaires au receveur pour créer un fichier de la bonne taille et le retrouver. Puisque la taille des données de chaque partie est fixe pour les deux ordonnanceurs (d'envoi et de réception) et que la taille du fichier est transmise, l'ordonnanceur de réception instancie également une base de données avec le numéro de toutes les parties qu'il attend. Cette base de données n'est pour l'instant utilisée que pour garder une trace des parties manquantes du fichier mais pourra apporter de nombreux avantages par la suite (enregistrer des métriques de performance, renseigner sur la progression de l'échange, reprendre une transmission, etc…).

Une fois qu'une transmission a débuté et que le fichier est créé et sa longueur fixée, l'ordonnanceur n'attend plus que deux types de messages messages :

- L'un contenant une partie,
- L'autre indiquant la fin d'une transmission.

Les messages sont vérifiés en amont par les canaux afin de s'assurer qu'aucun message non-interprétable n'atterrisse dans la queue de messages à traiter par l'ordonnanceur. Ce n'est qu'après qu'ils sont distribués par les canaux à l'ordonnanceur.

Pour chaque message contenant une partie, l'ordonnanceur va extraire la partie du message. Puisque chaque partie contient à la fois des données ainsi que la position où les écrire dans le fichier, l'ordonnanceur peut simplement aller au bon endroit dans le fichier et enregistrer les données. Après ça, l'ordonnanceur va préparer et donner à envoyer, un message d'accusé de réception à destination de l'émetteur afin qu'il sache que la réception et le traitement de la partie se sont déroulés avec succès. Enfin, il supprimera la partie reçue de la base de données des parties restantes à recevoir.

Lorsqu'un message de fin de transmission est reçu, l'ordonnanceur doit s'arrêter immédiatement. Pour le moment, il met juste à jour sa variable `isInProgress` et continue indéfiniment (dû au `while(true)` de la méthode d'exécution). **Il faut absolument changer ce comportement**. Idéalement, l'ordonnanceur ne devrait pas attendre ce genre de message et il serait de sa responsabilité d'indiquer qu'une transmission est terminée (lorsqu'un fichier est complètement reçu et formé). Les efforts nécessaires à la modification de ce comportement devraient être moyennement conséquents car il faut prévoir le cas où l'émetteur n'envoie plus rien (*crash* ou n'a plus rien à envoyer) et faire en sorte que l'ordonnanceur s'arrête de façon efficace en traitant tous les cas de figure.

Messages

Les différents canaux de communication permettent de s'échanger des données. Cependant, ces données n'ont de sens que lorsqu'elles respectent une certaine

structure, un certain protocole de communication. Afin que les appareils puissent communiquer entre eux et se comprendre, un protocole de communication a été mis en place sous la forme de messages générés avec Protocol Buffers (appelé par la suite protobuf pour faire plus court) dans sa version 3 (appelé proto3).

La structure de ces messages est la suivante :

- isAnAcknowledgement (protobuf : bool) (java : boolean)
- metaData / partData
- checksum (protobuf : bytes) (java : byteString)

Les champs metaData et partData sont des sous-messages englobés dans une instruction protobuf *oneof*. Cette instruction ne permet qu'à un seul des champs qu'elle englobe d'être initialisé. Un message ne peut donc posséder qu'un seul sous-message à la fois. Si un message possède un sous-message metaData par exemple, initialiser un sous-message partData écrasera le sous-message metaData. Un autre avantage de cette instruction est qu'elle met à disposition des outils de suivi de présence afin de déterminer lequel de ces champs a été initialisé.

Les champs du sous-message metaData sont les suivants :

- isInProgress (protobuf : bool) (java : boolean)
- filename (protobuf : string) (java : String)
- filesize (protobuf : int64) (java : long)

Les champs du sous-message partData sont les suivants :

- id (protobuf : int32) (java : int)
- data (protobuf : bytes) (java : byteString)

Dans proto2, il existe les instructions optional et required. Optional indique qu'un champ ne peut avoir que 0 ou 1 occurrence dans un message. Required indique qu'un champ doit avoir exactement 1 occurrence dans un message. Ces deux instructions ont été supprimées dans proto3 pour les raisons mentionnées dans ce lien : Pourquoi optional et required ont été supprimés dans proto3. Cependant, à cause de ces changements, il est devenu très compliqué voire impossible de déterminer la présence d'un champ dans un message. C'est d'ailleurs pour cette

raison que l'instruction optional a depuis fait son retour dans proto3 puisqu'elle aussi, tout comme l'instruction oneof, met à disposition des outils de suivi de présence. A noter que par défaut, les champs sans instruction dans proto3 sont des champs dits singular. Les champs singular sont identiques aux champs possédants l'instruction optional à la différence que les champs dits singular ne traquent pas la présence.

Les champs isAnAcknowledgement et checksum possèdent l'instruction optional. Puisqu'il n'est normalement pas possible de vérifier la présence d'un champ dans proto3, la seule façon de les rendre obligatoire dans chaque message est paradoxalement de leur donner l'instruction optional et de vérifier leur présence à chaque fois.

Canaux

Il y a trois canaux qui ont été envisagés pour le projet :

- Bluetooth
- Wi-Fi P2P
- NFC

Sur ces trois canaux, seul le canal Bluetooth a été implémenté pour le moment.

La gestion du canal Bluetooth est faite par l'objet BluetoothChannel. Cet objet va lancer tous les threads nécessaires à la connexion et à la transmission :

- AcceptThread (accepte une connexion)
- ConnectThread (demande une connexion)
- SendingThread (envoie des données)
- ReceiveThread (reçoit des données)

AcceptThread va ouvrir un BluetoothServerSocket et écouter les connexions entrantes. Tant qu'aucun appareil ne se connecte au BluetoothServerSocket et tant que celui-ci n'est pas fermé, le thread continuera d'attendre une connexion. Dès qu'une connexion arrive, un BluetoothSocket sera ouvert et le BluetoothServerSocket sera fermé (on ne veut communiquer qu'avec un seul

appareil, attendre d'autres connexions n'est donc pas pertinent). Les *threads* d'envoi et de réception seront ensuite lancés et utiliseront le BluetoothSocket fraîchement ouvert pour communiquer.

ConnectThread va ouvrir un BluetoothSocket et tenter de se connecter à l'appareil qui lui sera donné en paramètre de son constructeur. Tant que la tentative de connexion n'a pas réussi ou bien tant qu'elle n'échoue pas, le *thread* continuera d'essayer de se connecter à l'appareil renseigné.

SendingThread va se contenter d'envoyer les messages contenus dans une queue jusqu'à ce qu'on l'arrête. Après avoir envoyé un message, le message est mis dans une autre queue en attendant de recevoir son accusé de réception.

ReceivingThread va se contenter de recevoir des messages et vérifier leur validité jusqu'à ce qu'on l'arrête. Si un message est valide, il est transmis à l'ordonnanceur qui saura alors le traiter.

A noter que pour l'ordonnanceur d'envoi :

- Lorsque les canaux reçoivent un message, si ceux-ci sont valides, ils sont redirigés auprès de l'ordonnanceur. Seuls les accusés de réception sont attendus.
- Lorsque les canaux envoient un message, ils le mettent dans une nouvelle queue dans l'attente de la réponse. L'ordonnanceur regardera périodiquement cette nouvelle queue afin de mettre à jour les informations d'envoi (*early design* qui n'a pas encore totalement été instauré, il manque le ré-envoi de messages qui n'ont toujours pas accusé réception après certaines périodes de l'ordonnanceur).

A noter que pour l'ordonnanceur de réception :

- Lorsque les canaux reçoivent un message, si ceux-ci sont valides, ils sont redirigés auprès de l'ordonnanceur. Aucun accusé de réception n'est attendu.
- Lorsque les canaux envoient un message, ils ne les stockent pas car on n'attend aucune réponse (on envoie des accusés de réception).

Persistance

Une couche de persistance sur disque est absolument nécessaire dans l'éventualité d'une reprise sur panne. Théoriquement, dans le cas où l'application reste en avant-plan et n'a aucun problème, tout peut être fait en mémoire. Mais cette solution n'est pas envisageable tant il y a de variables d'interférence (perte de connexion, *kill* du système Android, etc…). Ultimement, il faut une couche de persistance sur disque ne serait-ce qu'afin d'assurer la complétude d'un échange. L'application peut aussi bénéficier d'autres avantages de part la mise en place d'une telle solution comme reprendre efficacement un échange, etc… Mais l'accent a été mis sur le fait d'assurer que toutes les parties d'un fichier soient bien reçues / envoyées.

De ce fait, pour le moment seule une base de données contenant les parties à échanger du fichier a été mise en place. Room, une librairie qui apporte une couche d'abstraction sur une base de données SQLite a été utilisé, de même qu'un schéma traditionnel :

- PartDatabase (la base de données)
- Part (l'entité)
- PartDao (le *Data Access Object* ou DAO)

PartDatabase contient les méthodes quant à l'instanciation de la base de données. Celle-ci est un singleton et ne sera manipulée que par les ordonnanceurs. À l'heure actuelle, PartDatabase supprime toutes les données contenues dans ses tables lors de sa création. Ce comportement sera à changer afin de supporter le cas où l'application doit reprendre un échange.

Part contient un id (int) et des données (byte[]). L'id référence le numéro d'une partie du fichier à échanger. Ce numéro, avec la CHUNK_SIZE de l'ordonnanceur, permet de récupérer les données d'une partie. Chaque partie ayant la même taille, les données d'une partie se trouvent donc à la position (ID-1) * CHUNK_SIZE du fichier. ID-1 car l'id de chaque partie est incrémental à l'instanciation et commence à 1. **A noter** que les données des parties ne sont pas

stockées dans la base de données : cela reviendrait à dupliquer le fichier et ce n'est pas souhaitable. Le champ résulte d'un *early design* qui doit être changé de même que dans les ordonnanceurs. Seul le champ PartData d'un message doit contenir les données du fichier. Les efforts à effectuer pour mettre à jour ce design devraient être faibles cependant, puisqu'au cours de la conception de l'application, il a été supposé que seul le champ PartData d'un message contienne les données.

PartDao contient les requêtes qui seront adressées à la table de Part de la base de données. Elle contient les habituelles opérations CRUD que l'on retrouve dans chaque DAO avec en plus une requête de compte du nombre d'entrées.

Interface utilisateur

L'interface utilisateur est composée de deux fragments :

- SendFragment
- ReceiveFragment

Un fragment est un composant indépendant pouvant être utilisé dans une activité ou dans un autre fragment. Il possède sa propre interface et son propre cycle de vie mais est attaché au contexte d'une activité. Dans le cas de l'application, les deux fragments sont attachés à l'activité principale MainActivity.

Le fragment SendFragment possède :

- Un fragment FileChooser pour choisir un fichier à envoyer. Ce fragment possède un bouton pour sélectionner un fichier et un textView pour afficher le chemin du fichier sélectionné,
- Un bouton FindDevice pour s'appairer avec un périphérique bluetooth découvrable aux alentours.

Le fragment FileChooser fonctionne comme attendu. Le bouton permet bien de choisir un fichier et son chemin est ensuite bien affiché dans le textView.

Le bouton FindDevice fonctionne également comme attendu en lançant une activité BluetoothDeviceListActivity. En revanche, cette activité rencontre quelques

problèmes dans le scan et la sélection de l'appareil avec lequel s'appairer. Le scan ne semble pas trouver d'appareils bien qu'il soit censé en trouver. La sélection d'un appareil dont l'appairage a déjà été fait dans le passé ne semble pas créer de lien bien qu'en en sélectionnant un l'activité retourne à la MainActivity. Ce comportement est donc à vérifier car il se peut que la sélection fonctionne comme attendu sans que l'on s'en rende compte. Le scan reste à débugger cependant.

Le fragment ReceiveFragment possède :

- Un bouton Receive pour être découvrable par les appareils aux alentours.

Le bouton Receive devrait idéalement être renommé en "Be Discoverable". Si l'appareil est déjà découvrable, alors le bouton n'est plus sélectionnable. Ce comportement est à étendre aux états STATE_CONNECTING et STATE_CONNECTED.

Discussion

Développement mobile

Le développement pour Android est assez particulier. Cela n'a pas été très difficile d'un point de vue du code car le projet a été fait en Java et qu'au final cela reste de la programmation orientée objet. Cependant, les différents types d'objet Android (Service, Foreground Service, Intent Service, Worker, AsyncTask, DownloadManager, etc…) sont assez désorientants pour un premier contact avec le développement mobile et bien qu'il y ait pas mal de documentations disponibles sur internet pour savoir quelle classe utiliser dans une situation donnée, cela remet pas mal en cause la conception d'une application.

Mis à part cela, le projet a été fait sur Android Studio qui est très pratique notamment au niveau des interfaces utilisateurs et du *build* / installation du programme sur un *device*. Il est basé sur IntelliJ et possède donc la plupart des avantages qu'IntelliJ possède (*linter*, auto-complétion, import de librairie, inclusion de Git, etc…).

Pour le coup, c'est vraiment penser et aboutir sur la meilleure conception de l'application qui a été le plus difficile dans le projet et pour un premier contact avec le développement sur Android. Quel type d'objet utiliser pour les ordonnanceurs ? Pour les canaux ? Comment se comporte un type d'objet dans tel scénario ? Quel impact peut avoir le système Android sur l'application et comment y remédier ? etc…

Ultimement, cela reste du Java et de la POO (Programmation Orientée Objet) donc une fois ces questions résolues, la programmation est assez intuitive.

Messages

Avant d'utiliser protobuf, un protocole similaire utilisant le standard ASN 1 a été essayé. Cependant, les différentes bibliothèques essayées étaient très compliquées à utiliser. A l'inverse, protobuf est beaucoup mieux documenté et beaucoup plus

simple à utiliser. Ultimement, les deux sont voués à faire la même chose mais protobuf a permis d'implémenter un protocole simple, clair et rapide (1 heure de travail) là où avec ASN 1 ce n'était pas le cas (plusieurs jours de travail…).

Un des avantages de protobuf est qu'il peut facilement être amélioré. Il suffit de modifier le fichier proto et de régénérer les fichiers dans le bon dossier. La commande pour générer les fichiers au bon endroit a été mise en commentaire dans l'en-tête du fichier proto.

Au niveau des performances, ça suit aussi. Voir benchmarks et messages benchmarkés. Les performances de sérialisation et désérialisation des messages sont très importantes dans ce projet. Puisque plusieurs canaux sont utilisés afin de paralléliser la transmission, on émet l'hypothèse que la vitesse d'échange s'en voit accrue. Il faut donc un protocole dont les performances physiques soient au moins aussi rapides que la vitesse d'échange afin de pouvoir suivre la cadence, que celui-ci ne soit pas le *bottleneck* de la liaison. Notre protocole semble plus simple que la plupart des messages benchmarkés. Bien que l'on ne peut faire d'affirmation pour le moment, les données que l'on peut trouver à ce sujet sont pour le moins rassurantes.

Système Android

Le système Android met pas mal de bâtons dans les roues. En effet, celui-ci va terminer les processus en arrière-plan au bout d'un certain temps et sous certaines conditions. Cela restreint fortement la conception de certaines applications comme la nôtre. Le système Android va terminer les processus en arrière-plan lorsqu'il manque de ressources ou bien si certains *threads* sont en marche depuis trop longtemps selon lui. En moyenne, Android va arrêter les Services toutes les 2 minutes, les Workers toutes les 10 minutes, etc… Cela afin de prémunir l'utilisateur de tâches qui tournent et consomment les ressources (batterie, mémoire, temps CPU, etc…) de son appareil sans qu'il le sache.

A ce jour, le seul moyen pour une tâche de se dérouler en fond indéfiniment est de notifier l'utilisateur. Cela prend la forme d'une notification qui doit être mise à jour le plus régulièrement possible afin de ne pas se faire épingler par le système Android. Cette gestion agressive des ressources et des applications est dûe au fait que chaque appareil Android est différent tant en termes de puissance que de capacité. On peut retrouver des systèmes Android avec 8 coeurs et +8 GB de mémoire vive comme avec 2 coeurs et 2 GB de mémoire vive. Cette disparité entre les systèmes (sans compter les modifications apportées par les surcouches) fait qu'Android doit utiliser au mieux ses ressources quoi qu'il arrive.

Pour les phases de tests de l'application, celle-ci peut simplement rester en avant-plan pour évaluer la transmission et éviter les problématiques de mise en arrière-plan.

Surcouche propriétaire

Il existe de nombreuses couches propriétaires pour Android (OneUI pour Samsung, OxygenOS pour OnePlus, etc…). Ces couches propriétaires se ressemblent mais restent différentes. Cela a un impact sur l'application. Normalement, celle-ci demande les permissions nécessaires à son fonctionnement mais certaines couches bloquent directement la permission faisant ainsi échouer ou mal fonctionner l'application.

Pour OneUI de Samsung par exemple, toutes les permissions ont été demandées et l'application, bien qu'incomplète, fonctionnait comme escomptée.
Pour le Pixel 4 et son interface Android native, certaines permissions n'ont pas été demandées et ont été bloquées directement. L'une des conséquences était que lorsqu'un appareil était sélectionné après un scan (donc un appareil déjà appairé, pour rappel le scan ne détecte pas encore les nouveaux appareils) l'application crashait.

Tâches restantes à réaliser

Application en arrière-plan

L'application utilise des objets pour les canaux et les ordonnanceurs. Cependant, lorsque l'application est mise en veille voire fermée, les objets sont détruits. Afin d'éviter ce comportement, il est nécessaire de passer par un Foreground Service. Les avantages à utiliser ce type de service sont :

- Il peut redémarrer autant de fois qu'il le faut jusqu'à ce que sa tâche soit terminée,
- Le système Android va éviter de tuer le processus tant que celui-ci met à jour une notification afin de prévenir l'utilisateur.

Ce type de service fonctionne également dans un *thread* différent du *thread* principal et permet également d'effectuer le transfert de fichier sans avoir l'application toujours ouverte. Cependant, s'il venait à être tué par le système Android, le processus devrait être capable de réinitialiser les objets et de reprendre là où il en était (responsabilité du programmeur).

Ordonnanceurs

Pour le moment, l'algorithme d'envoi des parties est assez naïf. Il se contente d'envoyer les parties dont la base de données fait mention. Il ne s'arrête que lorsque la base de données est vide, signifiant que tout le fichier a été envoyé et que le receveur a bien tout reçu. Il garde en mémoire les parties envoyées afin de ne pas les renvoyer à chaque tour de boucle mais ne les utilise pas à part ça. Il y a donc de la marge d'amélioration.

Un ordonnanceur pour les canaux peut également être créé. Il gèrerait toute la partie d'initialisation de la connexion. Si le canal Bluetooth peut effectivement *bootstrap* le canal Wi-Fi, on peut alors imaginer que d'autres canaux peuvent également bénéficier de la même solution. Si ce n'est pas le cas, d'autres solutions peuvent être envisagées comme utiliser un seul et même scan avec la borne radio 2,4 GHz, commune à Wi-Fi et Bluetooth 5.0 afin de récupérer le même appareil.

Il faut également s'assurer définitivement qu'un seul ordonnanceur (d'envoi ou de réception) soit instancié à la fois sur un appareil. Actuellement, les méthodes `getInstance` des ordonnanceurs s'assurent de ne créer qu'un seul ordonnanceur respectivement mais n'empêchent pas d'en créer un chacun. C'est à dire qu'il ne peut coexister deux ordonnanceurs du même type mais qu'il peut en revanche exister un ordonnanceur d'envoi et un ordonnanceur de réception instanciés en même temps. C'est un comportement à éviter à tout prix compte tenu de l'état actuel du projet.

Canaux

Parmi les canaux envisagés, seul Bluetooth a été implémenté pour le moment. Tout le trafic passe donc par ce canal. Afin d'améliorer les performances de l'application, répartir la charge de travail sur d'autres canaux est nécessaire. Les autres canaux envisagés et à implémenter sont :

- Wi-Fi P2P
- NFC

Avec le canal Bluetooth, il est possible de récupérer des informations sur l'appareil connecté tel que son adresse MAC. Ces informations peuvent être utiles pour prolonger une connexion via un autre canal. Par exemple, Wi-Fi P2P a besoin de l'adresse MAC d'un appareil pour s'y connecter. Puisque l'adresse MAC peut être retrouvée via le canal Bluetooth, cela peut permettre de se passer de la phase de *discovery* et de passer directement à la phase de connexion. On dit alors que le canal Bluetooth *bootstrap* le canal Wi-Fi. Il s'agit là d'une piste à étudier.

Appairage des appareils

Actuellement, l'appairage entre deux appareils ne semble pas fonctionner. Lorsqu'un appareil est choisi, l'activité censée lier les appareils semble retourner à l'activité principale sans rien faire. De plus, le scan ne découvre pas de nouveaux appareils. Le code est donc à vérifier. Celui-ci est repris du projet BluetoothChat du GitHub officiel de developer.android.com.

Evaluation des performances

L'évaluation des performances est une étape importante car elle permettra de répondre à la problématique du projet, à savoir de comparer la performance d'un protocole d'échange multi-canaux avec les solutions existantes. Elle permettra également de définir les axes d'amélioration de l'application. Une solution d'évaluation des canaux a été pensée mais retirée puisque le lien entre les canaux et les ordonnanceurs viendra peut-être à être modifié. Elle consistait simplement à garder une trace du dernier timestamp avec lequel `updateChannel` était appelé et à calculer la vitesse du canal pendant la période qui s'est écoulée. On peut aussi imaginer d'autres métriques à évaluer comme le nombre de messages arrivés avec au moins une erreur, la répartition de la charge des canaux avec l'augmentation de la distance entre les appareils, etc… Le tout étant d'avoir des données interprétables en direct par l'application, et utilisables *a posteriori* pour la production de graphes (*post-mortem*).

Reprise sur panne

Pour le moment, dans le cas où une transmission venait à être interrompue, elle devrait reprendre depuis le début. Il faudrait trouver une solution afin qu'un ordonnanceur puisse reprendre une transmission là où elle en était. Cela serait également utile pour l'utilisation de l'application en arrière-plan.

La base de données des parties est un bon début en ce sens. On pourrait continuer par exemple en créant une autre table regroupant des informations sur la transmission (adresse MAC de l'autre appareil, chemin du fichier d'échange, etc…). Ces informations seraient vérifiées à chaque liaison entre deux appareils et une reprise serait proposée à l'utilisateur lorsque cela est possible. Cela signifie aussi qu'une base de données serait créée par transmission.

Interface utilisateur

Les boutons FindDevice et Receive permettent de former la liaison entre les appareils. Cependant, les boutons qui lancent les ordonnanceurs n'ont pas encore

été créés. Il faudrait les créer et les rendre inactifs tant qu'il n'y a pas au moins un canal connecté et qu'aucun ordonnanceur n'est actif.

De même pour les boutons FindDevice et Receive, il faudrait les rendre inactifs si on est déjà dans un état en cours de connexion ou connecté ou qu'un ordonnanceur est déjà actif. Sinon, un utilisateur pourrait tenter de lancer plusieurs ordonnanceurs en même temps ou bien entamer plusieurs connexions en même temps, ce qui n'est pas un comportement souhaité par l'application.

Les ordonnanceurs respectent le patron de conception Singleton. Cependant, rien n'empêche que chaque ordonnanceur ait une instance fonctionnant. Il faut donc qu'il n'y ait qu'une seule référence à un ordonnanceur dans l'activité principale. C'est cette référence que les différents boutons devraient vérifier. Puisque les ordonnanceurs implémentent une interface appelée Scheduler et que celle-ci possède une méthode execution gérant tout l'envoi et la réception, le type de cette référence est l'interface Scheduler.

Il faudrait également afficher la progression de la transmission dans l'interface. Un champ pour cela a été créé dans le fragment Receive mais pas encore dans le fragment Send. Pour faire ça de la façon la plus propre, un entier appelé progression devrait être ajouté à ParentScheduler et à l'instantiation d'un l'ordonnanceur, un observateur observant cette valeur devrait être créé.

Conclusion

Le but du projet était de réaliser un protocole de communication s'appuyant sur plusieurs canaux. Afin d'évaluer ce protocole, une application l'utilisant pour échanger un fichier était nécessaire.

Le protocole a été pensé pour être le plus simple possible. Grâce à la solution Protocol Buffers, il est également clair et facilement compréhensible en regardant sa définition dans le fichier .proto à la racine du projet. Cette simplicité permet à la fois de facilement manipuler le protocole et de facilement le faire évoluer au besoin. Cependant, ce protocole n'a pas encore pu être testé car l'application n'est pas encore complète.

Le développement de l'application a été problématique au début (premier contact avec le développement mobile, quelle architecture adoptée, planification des séquences d'échanges entre l'émetteur et le récepteur, etc…). Les difficultés rencontrées ont certes ralenti le développement de l'application mais ont surtout permis de remonter des sujets intéressants et nécessaires à penser pour les communications multi-canaux (quelle méthode adopter en cas de reprise sur panne, comment profiter efficacement de la parallélisation de l'échange à travers les canaux, comment faciliter le pairage utilisateur, etc…). Ces éléments sont venus s'ajouter aux préoccupations initiales, traduisant ainsi tout le potentiel du projet et des communications multi-canaux. L'application n'est donc pas encore complète. Cependant, les bases ont déjà été implémentées et ne nécessiteront que peu de changement. L'application en est à un stade où il ne reste plus qu'à connecter les bouts entre eux et voir / débugger le résultat.

Le projet est donc proche d'un état fonctionnel.