Département de génie informatique et de génie logiciel

INF3995

Projet de conception d'un système informatique

Rapport final de projet

Chaîne de blocs pour dossiers d'étudiants

Équipe No. 4

Imrane Belhadia
Édouard Bochin
Ioana Bruj
Aurèle Chanal
Omar Talbi
Alexandre Vu

Avril 2020

Table des matières

Objectifs du projet **3**

Description du système **3**

Le serveur (Q4.5) 3

Les mineurs 6

Tablette (Q4.6) 9

Application sur PC 11

Fonctionnement général (Q5.4) 13

Résultats des tests de fonctionnement du système complet (Q2.4) **14**

Déroulement du projet (Q2.5) **15**

Travaux futurs et recommandations (Q3.5) **16**

Apprentissage continu (Q12) **17**

Conclusion (Q3.6) **18**

Références **18**

1. Objectifs du projet

L'objectif de ce document est de revenir sur le déroulement du projet Édu-Dossier dont le cadre a été défini précédemment dans la réponse à un appel d'offre d'un consortium constitué de l'École Polytechnique Montréal et de plusieurs autres entreprises. Nous préciserons l'architecture effective du système et de chacun de ses composants individuellement. Il s'agira ensuite de détailler les résultats obtenus et ce qui reste à faire.

L'objectif final du projet était d'offrir une solution permettant aux employeurs potentiels des étudiants et des gradués de Polytechnique Montréal d'accéder aux informations académiques des personnes concernées de manière simple et fiable. Les contraintes du projet ont été établies dans la réponse à l'appel d'offre conformément aux exigences du consortium à l'origine de celui-ci. La première exigence alors définie était de mettre l'accent sur la simplicité d'utilisation du système qui a pour interface une application android pour les usagers et une application PC pour l'administration. Il y avait également une grande importance accordée à la performance et la fiabilité. Pour ces raisons nous avons défini une solution avec quatre composantes tirant avantage de la redondance et d'une distribution des tâches entre les différents noeuds, soit un serveur web pour la gestion des requêtes HTTPS et des comptes des utilisateurs ainsi que trois mineurs pour la production et l'accès à l'information. De plus, pour ce qui est de la fiabilité, il a été défini que le système devrait être tolérant à la perte d'un ou de plusieurs mineurs. Chaque mineur devait être déployé sur un système embarqué de type FPGA avec un processeur ARM.

Des circonstances particulières ont empêché l'équipe de développement d'avoir accès à l'environnement initialement convenu dans l'appel d'offre et ont également entraîné un perte de deux semaines de développement. Face à ce changement de circonstances, les exigences du projet ont été ajustées par le consortium, réduisant ainsi les fonctionnalités requises lors de la livraison du projet. Le système n'a donc pas été déployé sur carte FPGA et la gestion de la perte de mineurs a été mise de côté pour la portée de ce projet. De plus, il a été nécessaire d'effectuer les arrangements nécessaires pour assurer une utilisation à distance de certains composants. Nous avons cependant conservé la présence de trois livrables, comme prévu initialement, qui ont été correctement délivrés avec les fonctionnalités prévues incluant les changements annoncés pour le livrable final.

2. Description du système

2.1 Le serveur (Q4.5)

Pour le serveur, comme dans la réponse à l'appel d'offre, nous avons implémenté l'interface REST avec la librairie RestBed [1]. Celle-ci avait l'avantage d'offrir une documentation claire, précise et détaillée avec laquelle on pouvait, avec quelques lignes de code C++, lancer un serveur sur un port d'écoute.

L'implémentation des gestionnaires de route HTTP que la librairie proposait était également claire et concise. Ainsi, dans le *main* du serveur, on retrouve les déclarations de toutes les ressources et la publication de ces ressources sur le service qui va lancer l'interface REST. Une ressource est un objet Restbed qui spécifie une route HTTP, par exemple `/admin/login`. On lui spécifie donc le chemin HTTP sur lequel elle est responsable de répondre, le type REST de la requête (GET, POST, PUT, DELETE, etc…), et le gestionnaire qui contient le code à exécuter. Un service est responsable de la gestion de toutes ces ressources. A l'aide de la méthode *publish*, on associe à un service toutes les ressources que l'on a déclaré. On démarre ensuite le service à l'aide d'un objet *settings*, qui va contenir le port d'écoute du service. Le serveur peut finalement être lancé.

Pour ce qui est du code de l'interface REST dans son entièreté, nous avons choisi de placer ça dans un dossier nommé RestAPI. À l'intérieur de ce dernier, on retrouve les gestionnaires communs à admin et à usager dans le fichier `EntryPoint.cpp`, les gestionnaires spécifiques à admin dans `PcAppAPI.cpp`, et les gestionnaires spécifiques aux routes usager dans `AndroidAppApi.cpp`. Pour les requêtes de connexion, de déconnexion et de changement de mot de passe, nous avons décidé d'utiliser les mêmes gestionnaires que ce soit pour la route admin ou usager puisque ces requêtes sont traitées de la même manière. Ainsi, pour différencier l'administrateur et un utilisateur dans EntryPoint.cpp, nous vérifions la route et le json contenus dans la requête; si par exemple la route est /admin/login, s'assurer que le nom d'utilisateur dans le json est admin. Nous regroupons donc deux routes différentes dans un même gestionnaire, et ce, pour les trois types de requêtes.

Pour ce qui est du système d'authentification, nous avons réussi à implémenter l'authentification avec les JSON Web Tokens. Lors d'une connexion, si un utilisateur est authentifié, le serveur va construire un jeton unique correspondant à cet utilisateur [2]. Le serveur va alors renvoyer ce jeton à l'utilisateur afin que ce dernier l'utilise pour toutes les requêtes subséquentes qu'il fera au serveur. Cette association, jeton/utilisateur, sera stockée dans un objet de type map nommé *loggedUsers*, que l'on retrouve dans la classe `UtilityClass.cpp`. Cette classe regroupe notamment toutes les méthodes utilitaires et les constantes. Ainsi à chaque fois que l'on reçoit une requête subséquente de l'utilisateur supposé authentifié, le serveur vérifie si le jeton stocké dans la map correspond à celui envoyé par l'utilisateur.

Quant à la base de donnée, celle-ci a été implémentée à l'aide de SQLite, grâce à sa librairie SQLite Amalgamation. Ainsi, dans `DatabaseManager.cpp`, on retrouve toutes les méthodes d'interaction avec la base de données du serveur, comme l'insertion, la création, la récupération, ou encore la suppression d'un compte. Nous avons pris soin de hasher les mots de passe dans la base de données avec l'algorithme SHA-256, utilisé également pour hasher les blocs des mineurs. Au début du main du serveur, la méthode `initializeDB`() est utilisée pour s'assurer que la base de donnée du serveur est présente à chaque démarrage de celui-ci. Cette méthode crée (ou vérifie la présence de) la base de données nommée `android.db` et y crée (ou vérifie la présence de) la table USERS avec les colonnes ID, EDITION, USERNAME,

PASSWORD. Ensuite, la méthode `initializeAdmin`() s'assure que le compte admin est bien dans la base de donnée.

Ensuite, nous avons la classe `AndroidAppAPI.cpp`. Elle contient les gestionnaires des routes usager pour la transaction d'un bloc, la récupération de l'information d'un cours, la récupération de l'information pour un étudiant, et la récupération des fichiers pdf. La communication serveur/mineur s'opère dans ces gestionnaires à travers des méthodes de la classe ServerZmq, qui s'occupe d'encapsuler toute la partie de communication avec la librairie ZMQ. On utilise deux sockets de cette librairie pour le serveur. Le premier est utilisé pour la synchronisation avec les mineurs et l'envoi de requêtes à un mineur spécifique alors que le deuxième permet d'envoyer simultanément la nouvelle information à miner aux trois mineur. L'algorithme utilisé pour contacter les mineurs est simplement à tour de rôle. On essaye de communiquer avec le mineur 0, puis 1, puis 2. La boucle s'arrête au premier mineur qui répond. Si aucun mineur ne répond, un code HTTP approprié et un message d'erreur sont envoyés au demandeur.

Pour ce qui est de la classe `PcAppAPI.cpp`, elle contient tous les gestionnaires spécifiques à la route admin: demande de récupération de logs, demande de récupération de la chaîne de blocs, et suppression ou création d'un compte.

Tout d'abord, la demande de récupération de logs effectuée par l'application PC permet de retourner les logs du serveur ou d'un des mineurs au compte admin. Avant toute chose, les logs sont créés à partir d'une classe qui se nomme `logManager`. Cette classe contient deux principales méthodes qui sont *appendLog()* et *sendLogs()*. La méthode *appendLog()* permet d'enregistrer un log dans un vecteur, ce qui nous permet de créer une liste qui contient tous les logs de la session en cours. Cette méthode est utilisée dans toutes les principales requêtes, ce qui nous permet de créer et de sauvegarder un log lorsque l'une de ses instances est appelée. La méthode *sendLogs()* permet d'envoyer des logs. Pour ce faire, nous créons un fichier contenant la valeur de notre liste de logs et nous rajoutons des paramètres afin que le fichier obtenu contienne le bon format JSON. Finalement, nous lisons le fichier obtenu et nous mettons son contenu dans une variable pour la retourner là où nécessaire. Pour récupérer les logs du serveur, nous utilisons la méthode *sendLogs()*. Nous devons entrer l'ID du dernier log reçu et le serveur renverra tous les logs présents sur le serveur, du dernier ID envoyé par l'application PC jusqu'à la fin. Si l'administrateur entre l'ID "0" pour le dernier log reçu, cela signifie qu'il n'a encore rien reçu. De ce fait, il pourra recevoir uniquement les 20 derniers logs présents sur le serveur. En ce qui concerne les logs des mineurs, le compte admin doit envoyer, comme pour les logs serveur, l'ID du dernier log reçu en utilisant la méthode *sendLogs()*, mais aussi l'ID du mineur qu'il souhaite consulter. Ainsi, les mineurs enverront des logs de la même manière et dans le même format que le serveur.

Concernant la demande de récupération de la chaîne de blocs, celle-ci s'effectue de la même manière que la requête des logs. Les mineurs enregistrent leur chaîne de blocs dans un fichier JSON qui pourra être lu et enregistré dans une variable afin de la retourner au besoin. L'administrateur peut donc, comme pour les logs, entrer l'ID du

dernier bloc reçu ainsi que l'ID du mineur qu'il souhaite consulter. Le serveur envoie l'ID du bloc au mineur correspondant à ce que l'admin veut. Le mineur concerné va créer la variable contenant la chaîne de blocs de l'ID reçu jusqu'à la fin. Finalement, il pourra envoyer cette variable au serveur qui se chargera de la faire passer à l'application PC.

Enfin, concernant la création et la suppression de compte, l'administrateur envoie une requête JSON contenant le nom du compte, le mot de passe ainsi que le mode d'édition pour la création et une requête contenant uniquement le nom du compte pour la suppression. Le serveur récupère les informations de la requête et utilise les méthodes de la database pour ajouter ou enlever un compte.

Finalement, un fichier `RestAPI.h` incluant tous les fichiers cités précédemment est présent afin de faciliter l'inclusion de cette interface dans le main.

À propos des changements effectués sur le serveur par rapport à l'appel d'offre, nous avons décidé de ne pas utiliser la librairie BOOST car il ne semblait pas pertinent de se servir des fonctions que proposait cet outil. De plus, nous avons utilisé ngrok pour la communication HTTPS. Au départ, nous voulions utiliser Restbed pour ce type de connection, mais nous avons eu de nombreuses complications et de ce fait nous avons décidé d'utiliser cet outil. Il nous permet de faire un "pont" entre les adresses HTTP que nous avons sur le serveur et une adresse HTTPS que nous donne ngrok. Cela nous a permi d'utiliser cette adresse HTTPS dans l'application PC et mobile pour faire de requêtes entre le serveur et les différentes applications sans être dans le même sous réseau.

2.2 Les mineurs

Pour les mineurs, nous avons assez bien appliqué l'architecture mentionnée dans la réponse à l'appel d'offre. Les mineurs communiquent et se synchronisent entre eux et avec le serveur Web grâce aux sockets de la librairie ZeroMQ [3]. Par contre, avec ces sockets, un mineur peut gérer 8 types de requêtes plutôt que 3 types de requêtes. Six de ces requêtes proviennent du serveur et deux requêtes proviendront des autres mineurs. Toutes les requêtes mentionnées sont gérées à travers une boucle infinie sous forme de *polling* [4].

Tout d'abord, le serveur peut faire des requêtes afin de changer la difficulté du minage des mineurs pour la création d'un hash, demander les informations contenues dans la base de données d'un mineur, voir le contenu de la chaîne de blocs, ou recevoir un fichier PDF, qui contiendra des informations reliées à un cours. Le serveur peut aussi faire une requête vers les mineurs pour récupérer de l'information sur un étudiant ou un cours, envoyer les informations d'un cours dans un trimestre avec une liste d'élèves, afin de les sauvegarder dans les mineurs. Les mineurs sauvegardent les données envoyées par le serveur dans leurs chaînes de blocs et dans leur base de données locale, et utilisent ces bases de données afin de rechercher les données demandées par le serveur. Pour sauvegarder les informations envoyées par le serveur, un mineur créé un hash pour insérer les informations d'un cours dans sa chaîne de bloc locale, contrairement à notre architecture de l'appel d'offre qui créait des hashs reliés aux informations des étudiants. Ensuite, le mineur enverra une requête afin d'avertir les

autres mineurs qu'il vient de rajouter un nouveau bloc dans sa chaîne de blocs locale et de vérifier si son nouveau bloc est valide. La création d'information se fait à travers un ensemble de sockets *publish*/*subscribe* [5] où les mineur reçoivent l'information. Ce choix d'architecture a pour avantage que le serveur ne dépend pas du bon fonctionnement des mineurs et simplifie grandement la synchronisation avec les trois mineurs lors de la production d'information. Il n'y a cependant pas de réponse à travers le publisher du server pour le minage. C'est une conséquence de l'architecture à laquelle il faut s'adapter.

Lorsque les mineurs reçoivent une requête de validation de hash de la part des autres mineurs, les mineur reçoivent avec la requête un bloc miné avec son hash. Ceci leur permet d'utiliser les attributs du bloc pour miner un hash et vérifier que le hash du bloc est bien valide. Une fois que les mineurs ont vérifié le bloc en question, ils ajoutent ce bloc dans leurs chaînes de blocs respectives et renvoient un message de confirmation vers le mineur qui a miné le bloc pour l'informer de la validité du bloc. Puis, chaque mineur augmentera le niveau de validation du bloc selon le nombre de mineurs qui ont vérifié que ce bloc avait un bon hash, et s'assure que chaque mineur a le même degré de vérification en envoyant une requête de mise à jour de la vérification d'un bloc. Une fois cette étape faite, les mineurs sauvegardent leurs chaînes de blocs dans des fichiers JSON et les informations envoyées par le serveur dans leur base de données locale.

Pour les librairies du mineurs, nous avions utilisés les librairies OpenSSL, SQLite et ZeroMQ afin de mettre en place le minage. OpenSSL est une librairie *open source* qui nous permet de créer le hash des blocs facilements, sans avoir à implémenter une méthode de création de hash nous-mêmes, l'algorithme utilisé est sha256. De plus les messages qui n'était pas de type string était sérialisé sous forme de Json pour l'envoie avec la même librairie que pour le serveur. Enfin, nous utilisons la librairie sqlite afin de créer les bases de données SQL locales des mineurs.

Pour ce qui est du code, nous avons trois structures. La classe *MinerCommunication*, pour les communications vers l'extérieur du mineur, l'ensemble de classes relatives à la chaîne de blocs et la partie qui interface avec la base de donnée. *MinerCommunication* se charge uniquement de l'implémentation de la structure de l'ensemble des sockets ZMQ sur TCP, ainsi que de fournir des méthodes pour envoyer différents types de messages aux autres mineurs et au serveur. Les opération de plus bas niveau concernant la gestion des messages avec ZeroMQ sont faites à travers un service partagé avec le serveur (*zmqMessageHandler*). Pour la partie de la chaîne de blocs, la première classe, la classe *Block*, représente un bloc individuel et fournit les fonctions de minage et de vérification associé du bloc. La classe Blockchain permet d'ajouter des blocs dans la chaîne de blocs ou d'effectuer la vérification du bloc, en appelant les méthodes de la classe *Block*. Pour un étudiant ou pour un cours, elle permet de s'assurer que les informations dans la base de données sont les mêmes que dans la chaîne de blocs en utilisant le mécanisme de la chaîne de blocs. Le fichier principal du mineur, *main.cpp*, sert à recevoir les requêtes par les sockets ZeroMQ, de créer des *thread* afin lancer le minage en parallèle avec la réception des requêtes d'informations, et de traiter les requêtes avec la chaîne de blocs. La chaîne de block est

un singleton dans chaque mineur, situé dans le fichier *BlockchainSingleton.hpp*. Ce fichier permet de gérer les différentes requêtes provenant du serveur et des autres mineurs, ainsi que d'enregistrer les informations de la chaîne de blocs dans la base de données SQL locale du mineur. De plus, ce fichier vérifie aussi que les informations prisent dans la base de données ne sont pas corrompues et qu'elles sont identiques à celles enregistrées dans la chaîne de blocs. Finalement, nous avons le fichier *MinerDbManager.cpp* qui crée la base de données des mineurs, enregistre les informations envoyés par le serveur et sélectionne les informations d'un étudiant ou d'un cours demandé par le serveur pour le renvoyer vers le singleton de la chaîne de blocs, qui vérifiera que l'information est valide et qui renvoie l'information vers le fichier *main.cpp,* qui renverra ensuite l'information vers le serveur. Pour finir nous avons également un service de lecture de fichier de configuration pour l'information définie à l'exécution on peut ainsi exécuter le même code pour les trois mineur avec des ports différent définie dans ce fichier de configuration. Il se trouve dans le code commun et peut etre modifier pour le serveur.

Cet ensemble est intégré dans un une partie qui jouait le rôle d'interface entre le main et les deux api développées qui implémente la gestion des requêtes des éléments externe au mineur. Ce dernier avait uniquement pour rôle de rediriger les différentes requêtes entre celles du serveur et des mineurs au bon gestionnaire de requête et de faire la gestion du thread chargé d'effectuer le minage de bloc.

La communication entre les deux fils d'exécutions est elle aussi gérée avec ZeroMQ , conformément au indication de la documentation de cette libraire. Le type de socket ZMQ utilisé est «pair»[3]. Une fois le minage lancé à travers le singleton de la chaîne de blocs, aucune opération de contrôle n'est faites. À la fin du minage dans le thread, celui-ci reçoit un message lui indiquant si son bloc sera ne doit pas être utilisé. L'interruption du minage de manière asynchrone par un thread extérieur dépasse le cadre de ce projet car elle est, en générale, déconseillée en c++.

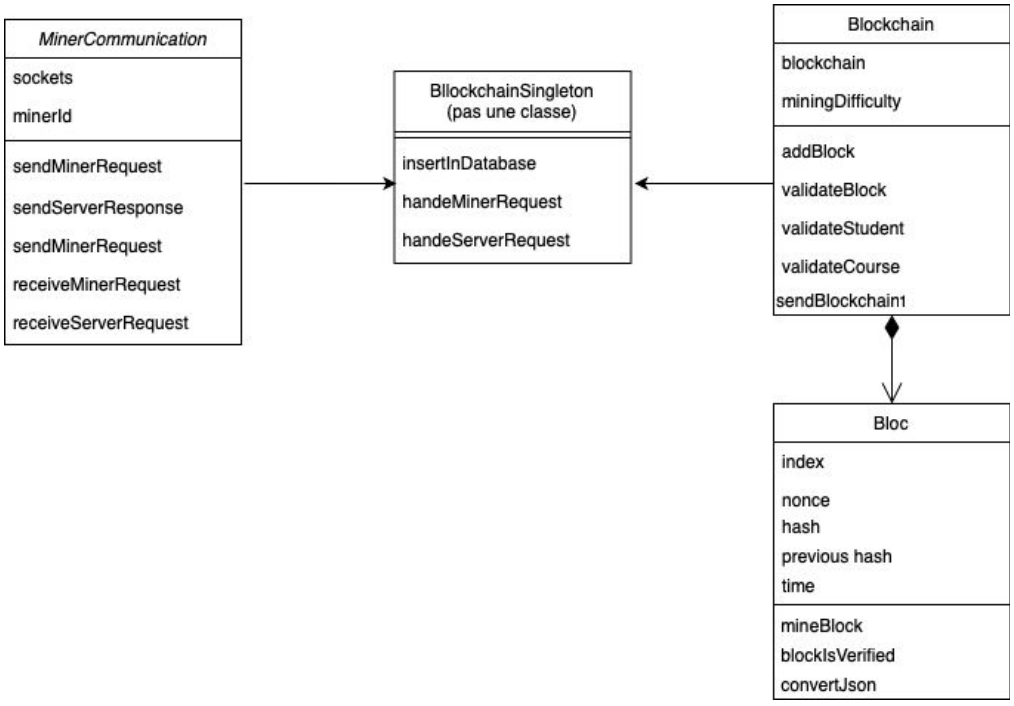

Diagramme 1 : Interaction des différents composants d'un mineur

2.3 Tablette (Q4.6)

Pour ce qui est de l'application mobile, certaines divergences de l'architecture initiale proposée ont été nécessaires. Nous nous sommes assurés d'utiliser des activités et des fragments lorsqu'approprié afin de tirer avantage des propriétés que ceux-ci offrent. L'application a cinq activités, qui ont toutes des interfaces graphiques très différentes les unes des autres:

- Activité de connexion (`MainActivity.kt`): permet d'entrer les informations de l'utilisateur et de se connecter pour accéder au reste de l'application. Lance la requête HTTP `/usager/login` en envoyant le nom d'usager et le mot de passe, et en retournant le JSON Web Token associé au compte si celui-ci existe.
- Activité de choix d'action (`OptionsActivity.kt`): permet à l'utilisateur de choisir l'action souhaitée dans l'application, notamment changer son mot de passe, consulter des informations, se déconnecter, et, si en mode d'édition, ajouter des informations. Permet de lancer la requête HTTP `/usager/logout` en choisissant l'option de déconnexion ou en faisant *back* sur le bouton natif Android.
- Activité d'édition de mot de passe (`UserActivity.kt`): permet de changer de mot de passe et valide l'entrée du nouveau mot de passe. Lance la requête `/usager/motdepasse` en envoyant l'ancien mot de passe et le nouveau.
- Activité de visualisation de PDF (`PdfActivity.kt`): permet à l'utilisateur de consulter les PDF nativement. Est lancé lorsque la requête `/fichier/notes` est réussie. Cette activité peut être lancée soit à partir du bouton sur la page de consultation d'un cours, ou sur chacun des cours d'un étudiant dans la page de consultation d'un étudiant. Dans les deux cas, le sigle et le trimestre du cours

sont envoyés en paramètre, et le string Base64 correspondant est reçu et reconstitué.

- Activité d'ajout d'information (`EditActivity.kt`): permet à l'utilisateur d'ajouter des informations sur le cours et de choisir un PDF parmi ceux présents sur son apparel. Lance la requête `/transaction` en prenant en paramètres les informations requises sur un cours (sigle, nom, trimestre et fichier PDF) et les données sur un ou plusieurs étudiants qui ont pris le cours (prénom, nom, matricule et note).

Ces activités permettent d'accéder à des fragments dépendamment des actions prises par l'utilisateur. Lorsque l'utilisateur choisit de consulter des informations dans `OptionsActivity.kt`, selon s'il souhaite consulter une liste de cours ou une liste d'étudiants, `CourseSearchFragment.kt` ou `StudentSearchFragment.kt` sont ouverts, respectivement. Ces deux fragments permettent d'entrer les informations requises afin de faire les requêtes `info/cours` et `info/etudiant` respectivement. Lorsqu'un cours est recherché, grâce à son sigle et trimestre, une liste d'étudiants ayant pris ce cours à ce trimestre est générée. Pour la recherche d'un étudiant, le ou les cours pris par l'étudiant sont générés; il est possible de spécifier ou non le trimestre ou le sigle souhaité. Il y a également un fragment pour la page d'accueil, `LoginFragment.kt`, consistant présentement de l'interface de connexion. C'est un fragment afin de faciliter la modification ou l'ajout d'options à l'arrivée sur l'application, par exemple le choix d'une langue autre que le français. Finalement, il y aussi le fragment d'édition d'information, `InformationEditionFragment.kt`, ouvert à partir de l'activité d'ajout d'information, qui permet d'ajouter des informations sur un ou plusieurs étudiants pour le cours spécifié dans l'activité

Toutes les requêtes HTTP sont gérées par la classe `HttpHandler.kt`, qui regroupe tout ce qui est nécessaire aux interactions avec le serveur. Cette classe permet de gérer autant les requêtes réussies que celles retournées avec des erreurs. Lorsque la requête est réussie, l'utilisateur en est averti grâce à une fenêtre contextuelle ou une redirection vers la prochaine étape de l'application. Lorsqu'il y a un échec lors de la requête, l'utilisateur en est également averti grâce à des messages décrivant l'erreur et ce qui doit être fait afin d'y remédier. Il y a également une méthode qui permet de convertir une liste d'étudiants en liste JSON bien formée

Il y a également deux classes qui permettent l'utilisation d'un *RecyclerView* pour la visualisation des listes d'étudiants et de cours: `ListAdapter.kt` et `ListManager.kt`. Un *RecyclerView* a été choisi pour ses nombreux avantages, surtout du côté performance. En effet, lors du chargement d'une liste contenant de nombreux éléments, le *RecyclerView* ne charge que ceux qui sont visibles sur l'écran de l'utilisateur ainsi que quelques-uns au-dessus et en-dessous de l'écran [6]. Ceci est particulièrement utile lorsqu'il y a beaucoup d'objets dans une liste, chose qui est possible avec notre application. En effet, considérant qu'un seul cours peut avoir plus d'une centaine d'étudiants, et qu'il y a beaucoup de cours à Polytechnique, il est raisonnable de s'attendre à avoir des listes composées de plusieurs dizaines

d'étudiants. Attendre qu'ils soient tous chargés aurait un impact important sur la performance de l'application.

Finalement, il y a une classe regroupant les différentes interfaces et classes de données nécessaires à l'application, `DataClasses.kt`. C'est ici que sont les informations de la chaîne de blocs et de l'utilisateur. Ceci permet de créer des listes de blocs afin de faciliter la structure du code.

Plusieurs librairies ont été utilisées afin de rendre l'implémentation des fonctionnalités plus facile. En premier lieu, *Fuel* a été utilisée afin de faciliter l'implémentation des requêtes HTTP [7]. Cette librairie a été choisie à cause de sa documentation claire, des différents tutoriels disponibles sur l'internet ainsi que de l'extension *Fuel-Json*, qui permettait de traiter des réponses Json aux requêtes HTTP envoyées. De plus, Fuel offrait également une extension *Fuel-Android*, qui permettait d'optimiser les requêtes HTTP pour qu'elles fonctionnent sur Android, en redirigeant les *callback* sur le fil principal d'exécution Looper, qui a pour but d'aider l'exécution continue [8].

Pour l'interface utilisateur, nous avons utilisé la librairie *Material Design*, de Google, afin d'uniformiser l'apparence de l'interface utilisateur et suivre les normes recommandées en design mobile [9]. En effet, Material Design préconise certains standards quant à l'apparence d'une interface utilisateur, et, par souci d'esthétisme et d'expérience utilisateur, nous avons choisi d'en utiliser les composantes lorsque possible, notamment pour les boutons et les icônes.

Nous avons également la librairie *Konform* afin de valider les entrées d'information et nous assurer que les données entrées par l'utilisateur et transmises au serveur sont dans le format attendu. Cette librairie a été choisie pour sa simplicité d'utilisation, sa documentation claire et exhaustive et les multiples exemples disponibles sur son site web [10]. Cette validation a été effectuée sur chacune des entrées de l'utilisateur, notamment le nom d'usager et le mot de passe lors de la connexion, l'entrée des informations des cours et des étudiants pour les transactions, et l'entrée d'informations lors de la recherche d'informations. Cela nous a permis de nous assurer qu'il n'y a pas de caractères interdits dans la base de donnée, sécurisant alors celle-ci, et de garantir également l'uniformité des données.

Finalement, nous avons aussi utilisé la librairie *AndroidPdfViewer* afin d'afficher nativement les PDF téléchargés lors de la consultation d'informations. La librairie a été choisie car elle supporte bien le chargement de PDF à partir d'un *byte array*, dans lequel nous avons transformé la Base64 reçue en réponse à la requête HTTP [11]. De plus, la librairie offrait plusieurs options pour l'affichage des PDF, avait de la documentation claire et permettait d'avoir une performance relativement bonne.

2.4 Application sur PC

Pour l'application PC, comme nous l'avions prévu initialement, nous avons utilisé Python afin d'optimiser notre productivité et d'avoir accès à des outils de haut niveau tel que la librairie Tkinter. Cette dernière vient d'une adaptation de la bibliothèque

graphique Tk écrite pour TCL et nous a permis la création de nos interfaces graphiques pour notre application PC [12].

En ce qui concerne la communication entre l'application PC et le serveur, nous avons utilisé la librairie *requests* qui est la librairie standard utilisé pour les requêtes HTTP en Python. Cette dernière permet notamment de réduire la complexité des requêtes HTTP grâce à un API très simple d'utilisation [13].

De plus, les données transitant de l'application PC au serveur via les requêtes HTTP devaient respecter le format JSON. Ainsi, pour respecter ce format de données, nous avons utilisé la librairie *json*. Cette librairie, par le biais d'un API simple, nous a notamment permis de convertir toutes sortes de structure de données dans un format JSON et inversement. Cela fut donc très utile lors de l'envoie et la réception de données car la flexibilité que nous offre cette librairie nous permit une manipulation facile des données à traiter [14].

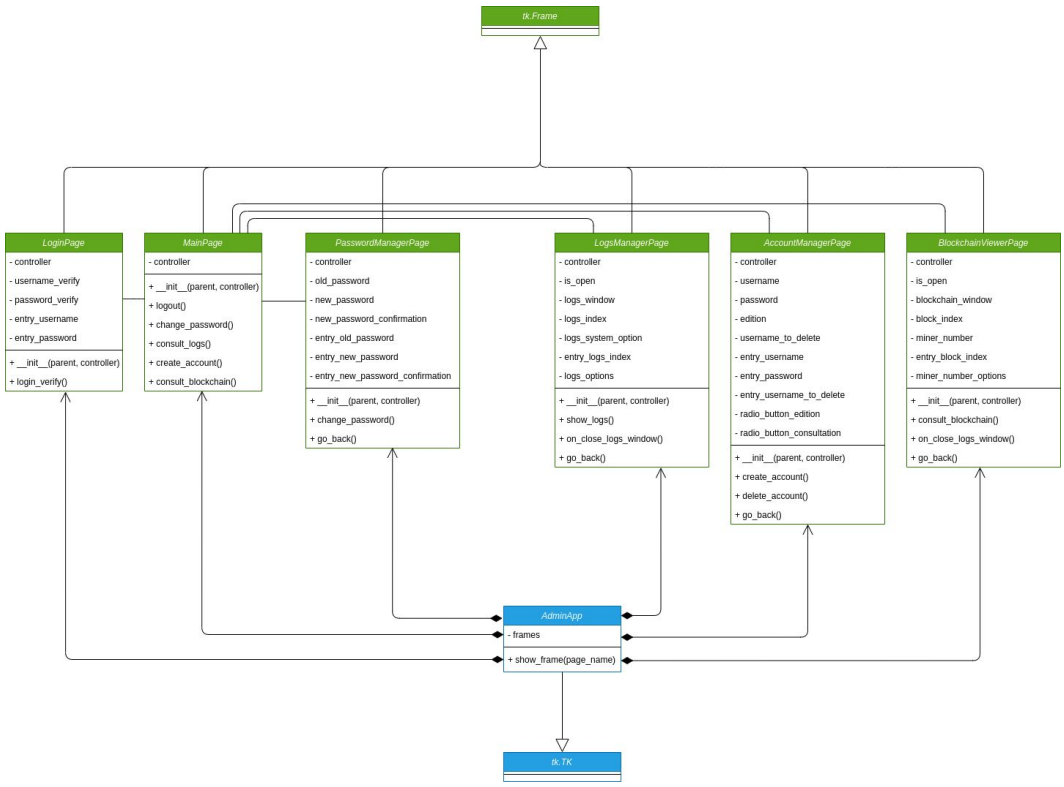

Figure 1 : Diagramme de classe de l'application PC.

Ci-dessus vous pouvez apercevoir une vue globale de l'architecture de l'application. En effet, notre application est composé majoritairement de classes qui interagissent entre elles. Tout d'abord, nous avons la classe *AdminApp* qui hérite de la classe tk.TK. Cette classe permet de générer la fenêtre principale de notre application PC et de définir et paramétrer toutes les pages (*frames)* que cette dernière contiendra. Ensuite, nous avons les différentes pages (*frames)* qui héritent toutes de la classe tk.Frame. Chacune de ces pages interagissent avec d'autres pages et la classe *AdminApp.*

Les différentes interactions entre les différentes classes de notre application PC sont les suivantes:

- Tout d'abord, l'*AdminApp* définit et paramétrise les cinq pages de l'application puis cette dernière affiche la page de connection *LoginPage* via la méthode *show_frame()*. De plus, chaque page est instanciée avec un *controller.* Ce *controller* est l'*AdminApp*; cela permettra donc à chaque pages de pouvoir changer de page grâce à la méthode show_frame() de *AdminApp.*
- Ensuite, une fois l'administrateur connecté, la classe *LoginPage* appelle la méthode *show_frame()* avec comme paramètre la classe *MainPage.* Depuis la *MainPage*, plusieurs options s'offrent à l'administrateur. L'administrateur a accès à la classe de gestionnaire des Logs, de gestionnaire de la chaîne de blocs, de gestionnaire de comptes et de gestionnaire de mots de passe.
- Finalement, une fois sur une de ces quatres pages de gestionnaires, l'administrateur peut retourner sur la *MainPage* grâce à un bouton qui appelle la méthode *go_back()* et se déconnecter via la méthode *logout()* de la *MainPage.*

Pour ce qui est de la découpe de l'application PC en plusieurs modules, nous avons un eu un souci. En effet, les différentes classes de l'application PC sont implémentés dans un unique fichier, non pas par choix mais par obligation. Notre implémentation devait normalement être découpé en 6 modules (une classe page par module) et un fichier main mais cela ne fut guère possible à cause de l'implémentation du *JSON Web Token*. Ce dernier doit être défini en tant que variable globale dans notre code pour qu'il soit partagé par chacune des classes pages et utilisé dans chacune des requêtes HTTP faites au serveur. Or si nous avions des modules séparés dans différents fichiers, nous aurions des problèmes de *import* en boucle qui causeraient un dysfonctionnement de l'application PC. Pour pallier à ces problèmes, il aurait fallu repenser à l'architecture de l'application PC.

2.5 Fonctionnement général (Q5.4)

Considérons que le correcteur n'a uniquement accès qu'au code sur le répertoire git du projet. La première étape consiste à mettre à jour les différents *submodules* de notre système grâce à la commande: *$git submodule update --init --recursive*. Ensuite, le correcteur peut effectuer la compilation et le lancement de toutes les composantes du système.

Pour le *backend*, un script de compilation (`compile.sh`) utilisant cmake est disponible dans le répertoire backend. Dans le but de diminuer le temps de compilation, il est conseillé d'utiliser les deux commandes suivantes: *$./compile.sh miner* et *$./compile.sh server* plutot que simplement *$./compile.sh* car le système complet est plus long à compiler. Une fois compilé, l'exécutable du mineur se trouve dans le répertoire *backend/cmake-build/projects/miner* et celui du serveur dans *backend/cmake-build/projects/server*. Pour utiliser les trois mineurs en même temps il faut les isoler, sinon ils utiliseront les mêmes ressources (base de données, fichiers de chaîne de blocs et logs). La manière d'isoler les mineurs dépend des spécifications de l'utilisation du système. Pour effectuer un test, avoir des mineurs sur des répertoires séparés permet l'initialisation de ressources différentes (il est possible d'utiliser le script

`backend/cmake-build/projects/miner/isolateMiners.sh` et ensuite de lancer les mineurs sur chaque répertoire créé). Il est aussi possible d'utiliser des machines virtuelles ou des conteneurs pour mieux isoler les composantes. La séparation des mineurs sur des noeuds distants (avec Internet) cause quelques problèmes de synchronisation lors du minage et n'est pas prévue dans l'utilisation du système. Elle implique la modification de code pour s'adapter à différentes adresses IP. Le fonctionnement n'est cependant pas compromis si un, deux ou trois mineurs sont disponibles au démarrage. Une fois que les mineurs sont opérationnels et isolés, il faut lancer l'exécution du serveur. Pour ce qui est du serveur, l'utilisation de HTTPS passe par la mise en place d'un tunnel (localhost.com ou ngrok) au port 8080 sur la machine hôte du serveur. La connection à distance se ferait donc sur cette adresse pour la tablette et l'application PC.

Pour le *frontend,* l'application PC est un script python 3, son exécution nécessite l'installation de la libraire tkinter et il faut spécifier l'adresse du serveur. Pour ce qui est de l'application android, l'exécution dépend de si l'APK est disponible ou s'il faut *build* l'application mobile. Si l'APK est disponible dans le Git, il suffit de connecter son appareil android à son ordinateur et transférer l'APK dans l'emplacement souhaité. Il suffira alors d'y accéder à partir de son appareil mobile et de l'installer; il peut être nécessaire de changer les paramètres de sécurité de son appareil afin de permettre l'installation à partir de sources inconnues. Si l'APK n'est pas disponible ou aucun appareil roulant Android n'est disponible, il sera alors nécessaire de cloner le code de l'application mobile, de l'ouvrir dans Android Studio puis de le compiler sur la plateforme. Il est alors possible de le rouler sur un appareil mobile connecté à l'ordinateur ou sur un émulateur directement dans Android Studio.

3. Résultats des tests de fonctionnement du système complet (Q2.4)

Globalement, toutes les fonctionnalités demandées dans le cadre du projet ont été complétées et sont fonctionnelles. Cependant, certaines fonctionnalités détiennent des limites de performances pouvant mener à des dysfonctionnements.

Nous avons notamment eu un problème de performance au niveau de la gestion des PDF dans notre base de données SQLite. En effet, les fichiers PDF étaient encodés en base64 ce qui peut représenter un nombre important de caractères à stocker dans la base de données. Or, notre base de données SQLite ne pouvait stocker que 150 000 caractères dans une colonne ce qui représentait un fichier PDF d'environ 125kB. Ainsi, il était impossible pour nous de téléverser, de stocker et de retrouver un fichier PDF de plus de 125kB sans causer un dysfonctionnement.

Il a également été noté que le message apparaissant lorsque l'utilisateur essaye de faire des recherches ou envoyer des transactions après que son compte ait été supprimé n'était pas celui approprié. En effet, il n'y a présentement pas de distinction entre les différents codes d'erreur retournés pour certaines des requêtes HTTP.

Lors de la conception du système, il a été convenu que les mineurs seraient situés dans le même réseau local. Le développement de la communication entre mineurs a donc été effectué sur cette base et l'environnement de tests a engendré une

latence très faible pour l'envoi de messages entre mineurs. Après avoir testé le minage avec des mineurs à distance, nous avons conclu que l'augmentation de la latence pouvait causer des problèmes si deux mineurs terminent leurs minage avec un intervalle trop similaire.

4. Déroulement du projet (Q2.5)

Durant le déroulement du projet, nous avons su tirer avantage des forces de chacun des membres de l'équipe en lui assignant les tâches sur lesquelles son savoir-faire prévalait. Ainsi, grâce à la description des ressources humaines effectuée dans l'appel d'offre, et suite aux différentes rencontres de cohésion et de compte rendu du projet, nous avons pu nous familiariser avec les atouts de chacun. Cependant, l'interchangeabilité des développeurs sur les différentes composantes du projet, comme décrite dans l'appel d'offre, n'a finalement pas été prise en compte. On s'est rendu compte que cela demandait beaucoup de ressources et que l'efficacité de l'équipe en aurait été compromise.

Pour ce qui est du contrôle de la qualité, nous avons réussi à implémenter notre processus de revue de code tel que décrit dans l'appel d'offre. Les demandes de fusion de code de tout membre de l'équipe ont dû être approuvées par au moins trois membres de l'équipe, et ce, afin de s'assurer d'avoir du code de qualité.

De plus, l'idée des tests unitaires n'a pas été retenue compte tenu de la complexité des différentes composantes et leur interdépendance. Cependant, plusieurs tests d'intégration, à différentes échelles ont été réalisées, comme par exemple, le test d'un minage d'un bloc dans la chaîne pour les mineurs.

Pour ce qui est du serveur, nous avons pu tester l'API REST à travers le logiciel Postman pour vérifier que toutes les routes HTTP sont correctes, que le format des json renvoyés au serveur soit correcte, ou encore que le serveur retourne le bon code d'erreur selon telle ou telle situation.

Pour ce qui est de la gestion de configuration du projet, l'implémentation d'une branche master et d'une branche dev, servant de *bleeding-edge* à la branche *master*, a été pertinente. En effet, au cours du projet, il est arrivé que des membres de l'équipe aient introduit des bogues mineurs dans le code, nuisant à la fonctionnalité du système. Ainsi, ces bogues n'ont pas pu être introduits dans le master et ont été détectés sur la branche dev. Des branches hotFix ont ainsi été spécialement créées pour corriger ces bogues.

5. Travaux futurs et recommandations (Q3.5)

Pour ce qui est de l'application mobile, plusieurs améliorations peuvent être apportées afin d'améliorer l'expérience de l'utilisateur. Premièrement, il serait intéressant de valider les entrées séparément. En effet, en ce moment, lorsque l'utilisateur entre plusieurs données sur la même page celles-ci sont validées d'un coup lorsqu'il essaye de soumettre la requête, et un message générique indiquant le format requis pour toutes les requêtes s'affiche. En d'autres mots, si l'utilisateur entre un sigle de cours et un trimestre afin de rechercher un cours, et fait une faute dans le sigle mais pas dans le trimestre, le message lui indiquera de vérifier le format des deux.

Idéalement, nous indiquerons la case spécifique dans laquelle il y a une erreur dans le format. De plus, il serait pertinent d'avoir un message d'erreur descriptif lorsque l'utilisateur essaye de téléverser un PDF excédant les capacités de la base de données. En ce moment, la requête de transaction passe quand même, et fait en sorte qu'une page blanche s'affiche lorsque l'utilisateur essaye de visualiser le PDF. Finalement, il serait pertinent d'avoir une indication côté mobile lorsque le compte en utilisation est supprimé. Ceci pourrait être fait soit avec des vérifications régulières avec le serveur afin de valider les informations du compte, ou avec le serveur qui envoie un message à la tablette lors de la suppression d'un compte actif.

Concernant le serveur, il aurait été intéressant de rajouter un temps d'expiration au JSON Web Token. Par exemple, si un utilisateur sur la tablette reste authentifié et met la tablette en veille pendant, disons 2 jours, celui-ci sera toujours connecté lorsqu'il ouvrira la tablette. Les JSON Web Token permettent justement d'ajouter un temps d'expiration au jeton, présentement défini à un an. De plus, si l'on voulait redimensionner le système pour un plus grand nombre d'utilisateurs, il aurait été pertinent d'avoir une base de données en accès rapide comme SQLite pour emmagasiner les utilisateurs présentement connectés, et non d'avoir un simple objet de type map. De plus, si le serveur avait tourné sur une carte SD sous Arch Linux comme prévu, il aurait été intéressant de garder une trace des ressources utilisées de l'OS et de fixer le seuil maximal d'utilisateurs connectés simultanément en fonction de la limite des ressources disponibles, et ce, afin de ne pas surcharger et faire défaillir le serveur. De plus, il aurait été pertinent d'améliorer la protection des données de l'utilisateur en implémentant des méthodes contre les injections SQL par exemple. Concernant les autres attaques potentielles, s'assurer du maintien du serveur et se protéger contre les attaques qui pourrait l'éteindre ou le saturer pourrait être un ajout important. Pour reprendre sur la communication à distance, la possibilité d'avoir un serveur et trois mineurs dans des sous réseaux différents devrait être implémentée afin d'avoir une plus grande polyvalence dans le déploiement de notre programme.

Pour les mineurs, plusieurs éléments d'amélioration sont intéressant à explorer. Il faudrait tout d'abord implémenter une file d'attente pour les requêtes de minage d'un bloc d'informations. Ainsi, si une requête est effectuée alors que les mineurs n'ont pas terminé la requête de minage précédente alors cette dernière sera mise dans la file d'attente. Cette fonctionnalité permettrait d'augmenter la fiabilité du système, notamment si plusieurs utilisateurs déclenchent une demande de minage simultanément. En ce moment, le mineur rejette des demandes de minage simultanées et l'utilisateur doit relancer sa demande plus tard, ce qui nuit à l'expérience utilisateur. Ensuite, il faudrait arrêter le minage de manière asynchrone sur le *thread* des mineurs tout en fermant le socket zmq dans le thread. Cela permettra d'économiser du CPU pour les mineurs plutôt que d'obliger un mineur à terminer un minage au complet et vérifier ensuite s'il est le premier à avoir terminé. Puis, il faudrait implémenter un gestionnaire pour la récupération d'un mineur et coordination du contenu de la chaîne de blocs entre les mineurs pour détecter la corruption de données dans le chaîne de blocs. Nous pourrions l'implémenter dans le service de synchronisation des mineurs et du serveur. L'utilisation du niveau de validation du bloc devient alors pertinente. Pour

l'instant, notre système ne gère pas le cas d'un mineur non-fonctionnel. Notre programme peut s'exécuter avec moins de trois mineurs mais ne permet pas de reconnecter un mineur avec les autres composants et ne met pas à jour les informations locales d'un mineur qui a manqué des requêtes de minage ou validation. Enfin, il faudra aussi enlever les informations corrompues dans les bases de données dès qu'il y a une détection d'informations non valides, et les remplacer par l'information mise dans la chaîne de blocs d'un mineur. Il faudra également s'assurer périodiquement que chaque bloc dans une chaîne de blocs d'un mineur ait été validé par les autres mineurs pour mettre à jour leur niveau de vérification. Puisque nous ne nous soucions pas des pertes de mineurs, cette dernière fonctionnalité n'a pas été implémentée.

Enfin, pour ce qui est de l'application PC, la première chose à faire serait de diviser l'application PC en plusieurs modules. Pour cela, il faudrait résoudre le problème de boucle de *import* causé par le JSON Web Token. Cela permettrait d'avoir un code plus compréhensible, plus maintenable et plus évolutif. De plus, il serait pertinent d'améliorer l'expérience utilisateur et l'interface utilisateur de la consultation des logs, de la création de comptes et de la consultation de la chaîne de blocs afin d'avoir une application plus facile d'utilisation et plus intuitive pour l'administrateur. Pour terminer, nous pourrions modifier les différents messages d'erreurs montrés à l'administrateur afin que ces derniers soient davantage révélateurs de la nature des erreurs.

6. Apprentissage continu (Q12)

Omar Talbi:

Durant le projet, je me suis principalement concentré sur le développement du serveur. N'ayant jamais touché à du backend en C++, j'ai dû m'adapter rapidement à de nouvelles librairies et à de nouveaux paradigmes de programmation. Je n'avais pas assez de savoir-faire au niveau de la programmation d'une API REST dans un langage tel que le C++, et aussi au niveau de la sécurité d'un serveur, par exemple comment s'assurer d'avoir un bon système d'authentification qui respecte l'architecture REST. Ma capacité d'adaptation a donc été mise à l'épreuve, et j'ai dû me documenter massivement en un minimum de temps, et essayer de collaborer le plus possible, tout en essayant de développer mes capacités interpersonnelles. Au début, les bogues dans l'application serveur étaient nombreux, mais plus le projet avançait, plus ma compréhension des technologies du serveur s'améliore au fil du processus de débogage. Cet aspect aurait pu être amélioré en identifiant les lacunes de compréhension, de savoir faire des membres de l'équipe qui travaillaient aussi sur le serveur, et essayer de les renforcer, à travers l'échange d'idées et de connaissances.

Alexandre Vu:

Durant ce projet, je me suis occupé de l'implémentation de la chaîne de blocs et du fonctionnement des mineurs. Au cours du développement du projet, j'ai réalisé que j'avais très peu de connaissances dans le réseautage de notre système, notamment sur les communications TCP. De plus, j'avais aussi un problème de minutie dans mon implémentation de code et dans la vérification des merges requests sur GitLab des

autres membres de l'équipe. Pour remédier à ces lacunes, j'ai travaillé en binôme avec d'autres membres de l'équipe qui étaient plus confortables à programmer le réseautage entre les mineurs, le serveur, l'application PC et l'application Android. Pour mon problème de minutie, je me suis basé sur les règles de travail que l'équipe avait discuté pour implémenter une convention de code commune (camelCase, noms de variables clairs en anglais, etc.). Ces différents aspects problématiques aurait pu être amélioré avant le projet si je m'étais plus familiarisé avec la documentation et les exemples en ligne pour le réseautage et si je portais plus attention aux détails et que je m'attardais plus régulièrement sur mon code pour trouver des erreurs de codage et des erreurs de convention.

Aurèle Chanal:

Durant ce projet, je me suis principalement occupé de l'implémentation du serveur et de l'organisation de l'équipe. Au départ, je n'avais pas énormément de connaissance dans le développement d'un serveur mais les librairies que nous avons utilisées étaient suffisamment détaillées pour ne pas avoir trop de problèmes. Durant le début du projet, j'était le seul de l'équipe à ne pas avoir de laptop à ma disposition ce qui me poussait à devoir programmer directement sur les cartes ou en utilisant un IDE sur les machines de Polytechnique. De ce fait, j'ai rapidement eu des lacunes sur l'utilisation des cartes qui utilisaient un OS auquel je n'avais pas l'habitude. J'ai pu par la suite acquérir un laptop qui régla une grande partie de mes problèmes d'implémentation. Ensuite, le projet contenant de nombreux aspects, nous avons dû nous séparer les tâches. Cependant, certaines demandaient plus de travail que d'autres et de ce fait, quand j'ai eu terminé ma partie, je souhaitai aider mes collègues dans les parties plus complexes. J'ai eu quelques lacunes dans l'apprentissage des autres parties du projets mais grâce à un échange efficace des connaissances au sein de l'équipe, j'ai pu régler ces problèmes. J'aurais pu régler ces problèmes en amont si je m'étais intéressé plus à l'ensemble des parties du projet et pas seulement ma partie. Finalement, le fait d'avoir un laptop disponible avant le début du projet aurait pu me permettre de gagner du temps quant à l'implémentation de mes tâches.

Edouard Bochin:

Pour ce projet, je me suis principalement occupé de l'implémentation de l'application PC. Lors de cette implémentation, je n'ai pas eu de problèmes pour développer les fonctionnalités correspondantes au bon fonctionnement de l'application mais je me suis découvert des lacunes lors de la conceptualisation et l'implémentation de l'interface utilisateur et de l'expérience utilisateur. En effet, c'était pour moi la première fois que je devais effectuer ce travail de recherche et de design relié à l'interface utilisateur et l'expérience utilisateur. Ainsi, comme mes premiers essais n'étaient guère concluants, j'ai décidé d'exposer l'application PC aux autres membres de l'équipe et aux chargés afin de recevoir des critiques et des conseils pour pouvoir élargir mes connaissances d'une bonne interface graphique. J'aurais pu pousser mes connaissances encore plus loin si j'avais exposé notre application hors de Polytechnique pour avoir des conseils de personnes dont c'est le métier. Enfin, j'aurais

pu me renseigner davantage sur les bonnes pratiques et règles à suivres pour implémenter une bonne interface utilisateur et une bonne expérience utilisateur.

Ioana Bruj:

Au début de ce projet, je me suis vu assigner le développement de l'application mobile. Pour moi, ne pas avoir une vue d'ensemble et toucher à plus d'une technologie durant l'exécution d'un projet représente une importante lacune, tant technique que personnelle. En effet, limiter l'apprentissage à une seule technologie avec laquelle je suis familière ne contribue pas à mon développement en tant qu'ingénieure. Afin de pallier à cette lacune, j'ai pris l'initiative de développer la base de données côté mineurs avec un autre des membres de l'équipe. C'était la première fois que je travaillais sur une base de données, et cela m'a également permis de plus toucher au *backend* du projet. De plus, je me suis assurée de revoir tout le code du projet, incluant celui du *backend* et de l'application PC, afin d'essayer de comprendre les technologies utilisées et être consciente du fonctionnement global du système. Afin d'améliorer encore plus cet aspect de mon apprentissage, j'aurais pu insister plus afin d'aider au développement des mineurs et du serveur, ce qui m'aurait permis de me familiariser avec ces composantes.

Imrane Belhadia:

Mes deux principaux rôles dans ce projet étaient de développer les communications entre les mineurs et le serveur, puis, avec un autre membre de l'équipe, d'intégrer mon travail avec la gestion de la chaîne de blocs et de la base de données pour gérer les requêtes du serveur web. Pour ce qui est de la deuxième partie, nous avons pas eu de problème de conception avec les autres membres. Par contre, je n'avais jamais effectué la conception d'un système de communication aussi complexe entre 4 composantes. Je me suis donc lancé trop rapidement dans le développement, ce qui était une erreur. J'ai cependant su reconnaître les potentiels problèmes de se lancer dans l'implémentation d'une telle composante sans définir tous les points de manière assez détaillée. J'ai ensuite passé plus de temps sur la documentation de la librairie utilisée afin de mettre à profit les fonctionnalités les mieux adaptées à nos requis. On peut prendre pour exemple le type de socket ZeroMQ à utiliser ou l'implémentation d'un service personnalisé pour gérer les messages ZeroMQ plutôt que d'utiliser celui conseillé dans la documentation qui était trop générale et rendait le code moins lisible. Des améliorations restent, bien sûr, nécessaires pour rendre l'utilisation de ZeroMQ plus efficace et plus encore plus flexible pour le système.

7. Conclusion (Q3.6)

À l'échéance d'un projet de grande envergure tel que celui-ci, il est normal de constater certaines déviations par rapport à la planification initiale, et il peut être intéressant et éducatif de faire un retour sur celles-ci et leurs causes. Malgré la complétion adéquate des quatre composants du système, des difficultés ont été rencontrées durant leur développement, nous obligeant alors à ajuster notre façon de travailler ainsi que les exigences du projet.

Premièrement, étant donné les circonstances exceptionnelles de ce semestre, la partie matérielle du projet a été mise de côté. Les différentes composantes du système n'ont pas pu être implémentées sur les cartes FPGA.

De plus, au dépôt de la réponse à l'appel d'offre, nous avions prévu que les développeurs travaillent sur toutes les composantes à tour de rôle pour que tout le monde bénéficie d'un maximum de connaissances techniques. Nous avons cependant découvert que l'implémentation de différents composants avec des technologies très différentes les unes des autres limitait cette flexibilité, et que la spécialisation de chacun des membres sur une technologie particulière fut préférable afin de créer de l'expertise au sein de l'équipe.

Par ailleurs, les composantes étant interdépendantes, il était nécessaire de communiquer et de prendre connaissance des différentes interfaces de communication des composantes. À cause de la spécialisation de chaque membre de notre équipe, où personne n'avait une connaissance globale du fonctionnement des différentes composantes de notre système, cette étape de partage de connaissances fut primordiale dans le bon fonctionnement de notre système.

L'utilisation et le respect des échéanciers que nous nous étions fixés en tant qu'équipe nous furent utiles afin d'assurer la complétion des composantes dans un délai respectable. Nous avons dû, plusieurs fois, ajuster les dates ou contenus des sprints initiaux déterminés dans la réponse à l'appel d'offre dû à des difficultés rencontrées dans certaines fonctionnalités dont en dépendaient d'autres, mais l'établissement des nouveaux échéanciers a permis à l'équipe de mieux comprendre le progrès global du système et de s'ajuster en conséquence.

De plus, l'utilisation d'un logiciel de gestion de configuration, ici Gitlab, comme nous l'avions prévu dans la réponse à l'appel d'offre, nous a été d'une grande utilité. En effet, les fonctionnalités de Redmine étant limitées, le choix d'effectuer un mirroring et de travailler sur GitLab a été justifié. Ce logiciel nous a notamment permis de revoir le code de chacun, de commenter la moindre faute de programmation, apporter des améliorations, et faciliter l'échange de connaissances. Son système de suivi de bugs nous a également permis d'améliorer la qualité logicielle des composantes, grâce aux demandes d'assistances des différents développeurs.

En somme, notre démarche de conception s'est avérée utile dans le cadre du développement de notre système. On peut voir qu'une grande partie de notre plan a été respecté, comme le choix des librairies ou le modèle de données décrit à la section 3 de la réponse à l'appel d'offre. Pour ce qui est des divergences entre le système et la description, celles-ci s'expliquent par deux facteurs. Le premier est le changement dans les requis. Une composante toujours présente dans la réalisation de projet en ingénierie. D'autre part, notre manque de connaissances techniques dans le domaine a forcé la modification de certaine partie de la conception. Il s'agissait cependant d'une activité nécessaire au bon déroulement du projet et au respect des échéances.

8. Références

[1]: Corvusoft, "Corvusoft/restbed," *GitHub*, 15-Oct-2019. [Online]. Available: https://github.com/Corvusoft/restbed. [Accessed: 16-Apr-2020].

[2]: F. Copes, "JSON Web Token (JWT) explained," *Flavio Copes*, 13-Dec-2018. [Online]. Available: https://flaviocopes.com/jwt/. [Accessed: 16-Apr-2020].

[3]: P. Hintjens, "ØMQ - The Guide," *Chapter One - ØMQ - The Guide*. [Online]. Available: http://zguide.zeromq.org/php:chapter1. [Accessed: 16-April-2020].

[4]: Vidyarthi A. R., "ZMQ Poller", 31-May-2017. [Online]. Available: https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/multisocket/zmqpoller.html. [Accessed 17-Apr-2020]

[5]: P. Hintjens, "ØMQ - The Guide," *Getting the Message Out*. [Online]. Available: http://zguide.zeromq.org/page:all#Getting-the-Message-Out. [Accessed: 17-Apr-2020].

[6]: Google Developers, "RecyclerView", *Google Developpers*, 04-Apr-2020. [Online]. Available: https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView . [Accessed 17-Apr-2020].

[7]: Vantassin K., "kittinunf/fuel", *GitHub,* 19-Apr-2020. [Online]. Available: https://github.com/kittinunf/fuel. [Accessed 19-Apr-2020].

[8] Google Developpers, "Looper", *Google Developpers*, 12-Feb-2020. [Online]. Available: https://developer.android.com/reference/android/os/Looper. [Accessed 17-Apr-2020]

[9] Google, "Design", *Google*, 19-Mar-2020. [Online]. Available: https://material.io/design. [Accessed 17-Apr-2020].

[10] konform-kt, "Portable validations for Kotlin", 21-Mar 2020. [Online]. Available: https://www.konform.io/. [Accessed 17-Apr-2020].

[11] Zapate, J., "barteksc/AndroidPdfViewer", *GitHub*, 17-Aug-2019. [Online]. Available: https://github.com/barteksc/AndroidPdfViewer. [Accessed 17-Apr-2020].

[12] Python, "TkInter", *Python*, 06-Dec-2019. [Online]. Available: https://wiki.python.org/moin/TkInter. [Accessed 17-Apr-2020].

[13] Ronquillo, A., "Python's Requests Library (Guide)", *Real Python Tutorials*. [Online]. Available: https://realpython.com/python-requests/. [Accessed 17-Apr-2020].

[14] Lofaro, L., "Working With JSON Data in Python", *Real Python Tutorials.* [Online]. Available: https://realpython.com/python-json/. [Accessed 17-Apr-2020].