Département de génie informatique et génie logiciel

INF3995

Projet de conception d'un système informatique

Proposition répondant à l'appel d'offres

no. A2016-INF3995 de Vintage Inc.

Conception d'un jeu d'échecs électronique

Equipe 02

Barouche Sabrina

Carphin Philippe

Gamache Clement

Ouellet Francis

Lanteigne Clement

Vachon Bureau Guillaume

16 Octobre 2016

Table des matières

1. Vue d'ensemble du projet

1.1 But du projet, portée et objectifs (Q2.1et Q4.1)

Le but du projet est de concevoir un jeu d'échec interactif destiné au divertissement de personnes âgées. Ce contrat nous a été confié par la compagnie *Vintage Inc*. L'objectif est de permettre à la clientèle de pouvoir jouer à ce jeu, l'un contre l'autre, et de retransmettre la partie sur un écran dans la pièce commune d'un centre de personnes âgées pour permettre aux passants d'assister à la partie en cours.

La livraison du projet s'effectuera en deux temps. Seul le code source du projet sera remis. Le premier livrable permettra à deux joueurs d'utiliser une ou deux tablettes pour participer à une partie d'échec minimale et fonctionnelle. Le support pour l'option "en passant", pour la promotion des pions et pour les conditions de fin de jeu, ainsi que les compteurs et les préférences seront plutôt implémentés pour le deuxième livrable. Cette organisation permettra d'obtenir rapidement une rétroaction du client sur la partie fonctionnelle du projet.

1.2 Hypothèse et contraintes (Q3.1)

Hypothèses :

- Utilisation de *Android Studio*
- Utilisation d'une tablette *Nexus 9*
- L'affichage *HDMI* s'adapte à la résolution des écrans du local L-3712
- On prend un code en *C* existant pour la gestion de partie du jeu d'échec
- Le jeu d'échec va être joué par des êtres humains
- Le serveur va être développé sur la carte *FPGA*

Contraintes :

- Le jeu d'échec va se dérouler sou forme d'une communication client-serveur
- Deux êtres humains doivent s'affronter lors d'une partie
- Le client doit être implémenté avec une application *Android* sur *Nexus 9*
- Le serveur doit être implémenté sur une carte *FPGA Zedboard*
- Nous pouvons avoir soit les deux joueurs sur la même tablette, soit un joueur par tablette
- Un code secret doit être partagé entre les deux joueurs
- Le code secret va être encodé en *base64*
- Le serveur enregistre et valide les coups

- Le serveur fournit l'interface *REST*
- Le serveur s'occupe de l'affichage sur *HDMI* du jeu
- Lorsqu'on appuie sur le bouton-poussoir PB1, une nouvelle partie commence
- La *DEL* LD9 s'allume pour signifier qu'une partie est en cours et si elle est éteinte cela signifie qu'aucune partie n'est en cours.
- L'application Android a une fonctionnalité qui permet de changer la forme les couleurs ou autres éléments des pièces
- Utilisation d'un processeur *ARM Cortex-A9*
- Utilisation de mémoire vive externe de 512 MiB
- Utilisation d'un pilote pour contrôler l'interface *HDMI* (vidéo)
- Utilisation d'un un pilote pour contrôler un *UART*
- Utilisation de compteurs et le gestionnaire d'interruptions du *PS* du *Zynq*

1.3 Biens livrables du projet (Q2.3)

Nous allons produire deux livrables, soit le Livrable 1 et le Livrable 2.

- Livrable 1 : Remise 15 novembre 2016
    - Fichiers Matériels *Vivado*
        - Les fichiers servant à la création du projet *Vivado*
        - Les fichiers de blocs de design pour la configuration matérielle
    - Fichiers Logiciels *SDK*
        - Le *board support package*
        - Tous les fichiers source.h
        - Tous les fichiers source.c
        - Les librairies additionnelles au *bsp*
    - Fichiers *Android* Studio
        - Arborescence de projet *Android*
    - Fonctionnalités
        - Début d'une partie par le joueur 1
        - L'interface principale de l'échiquier (Tablette et *FPGA*)
        - Affichage de la partie par *RS232* par le serveur
        - Interface de base de l'application tablette
        - Déplacement des pièces sur tablette et confirmation par le serveur
        - Capturer les pièces de son adversaire
- Livrable 2 : Remise 1er décembre 2016

- Fichiers Matériels *Vivado*
  - Les fichiers servant à la création du projet *Vivado*
  - Les fichiers de blocs de design pour la configuration matérielle
- Fichiers Logiciels *SDK*
  - Le *board support package*
  - Tous les fichiers source.h
  - Tous les fichiers source.c
  - Les librairies additionnelles au bsp
- Fichiers *Android Studio*
  - Arborescence de projet *Android*
- Fonctionnalités
  - Toute les fonctionnalités du livrable 1
  - Détection de situations d'échec, d'échec et mat ou de pat
  - Ajout du roque et de la prise en passant
  - Promotion des pions
  - Ajout des compteurs sur la tablette et sur le serveur
  - Système de préférences usager sur la tablette

2. Organisation du projet

2.1 Structure d'organisation

L'équipe chargée du projet est constituée de six membres, chacun pouvant assurer un rôle de développeur-analyste ou de coordonnateur de projet, au besoin. Les prises de décision, tant au niveau design qu'au niveau technique, sont effectuées par l'équipe.

Voici les rôles assumés au sein de l'équipe:

- Sabrina Barouche: Développeur Android, concepteur d'interface
- Philippe Carphin: Développeur C, Expert Git
- Clément Gamache: Développeur Android, concepteur d'interface
- Clément Lanteigne: Spécialiste matériel
- Francis Ouellet: Développeur C
- Guillaume Vachon-Bureau: Développeur C et Java, Gestionnaire

Tel que spécifié dans le document *Demande de proposition no. A2016-INF3995 : Conception d'un jeu d'échecs électronique*[2], la supervision du projet est assurée

par *Vintage Inc*. Ce dernier se réserve le droit d'évaluer les capacités techniques de l'équipe de réalisation du projet.

2.2 Entente contractuelle

Nous proposons le contrat *livraison clé en main*, avec possibilité de support dans le futur. Le prix est donc fixe, ce qui permettra de garantir un produit final fonctionnel et constitue un bon premier contact avec le client.

Les détails relatifs aux coûts sont explicités dans la section *4.1*.

3. Solution proposée

3.1 Architecture matérielle (Q4.2)

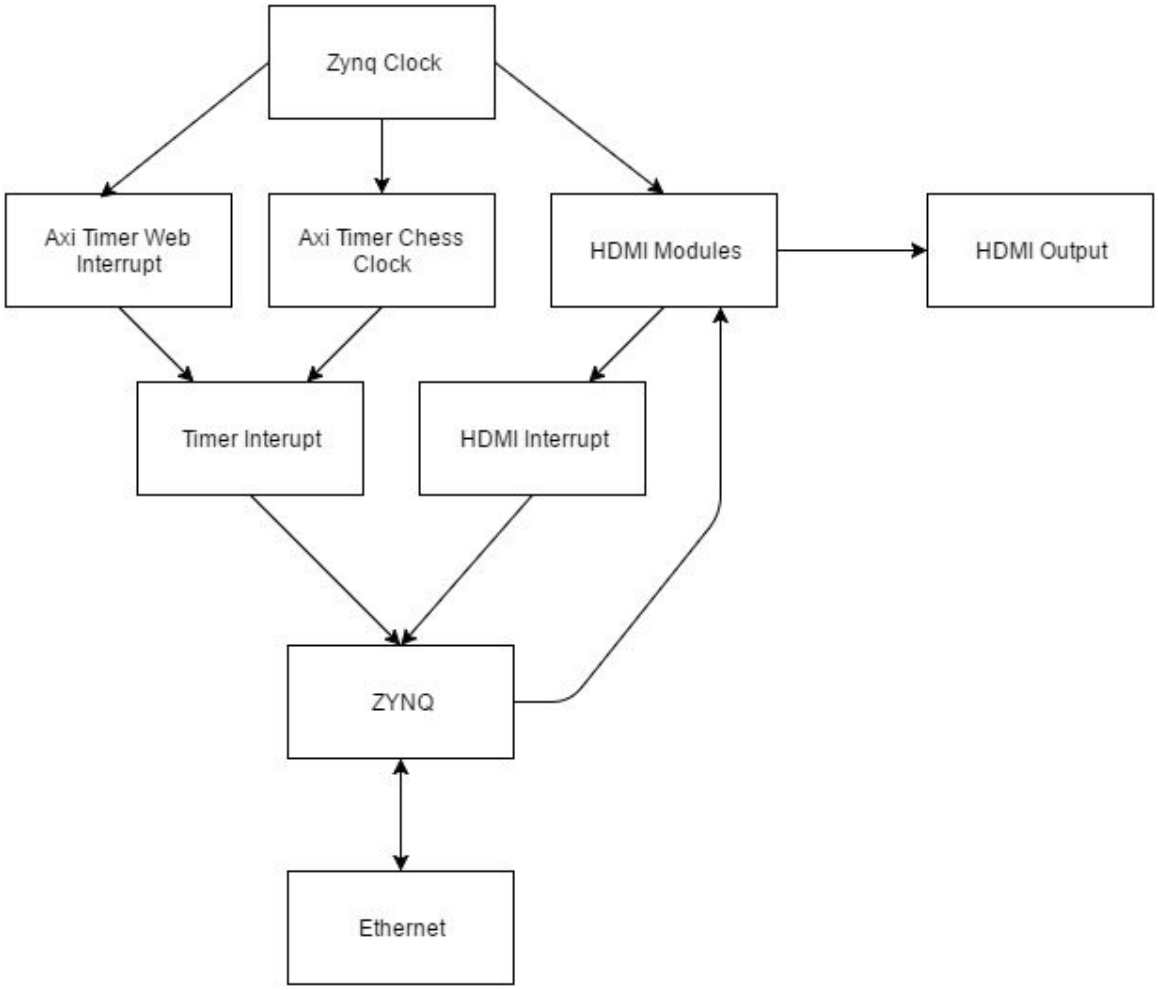

Image 1: Architecture matérielle

Nous utilisons le *Zynq* du *Zedboard* et nous instancions sur le *FPGA* deux compteurs ainsi qu'un groupe de modules pour l'affichage *HDMI*.

Nous utilisons les horloges du *Zynq* pour alimenter nos deux compteurs *Axi Timer*. Le premier compteur *Axi Timer Web Interrupt est* utilisé pour générer une interruption dans le module logiciel *TCP/IP*. Le deuxième compteur *Axi Timer Chess clock* sert d'horloge comptant le temps de la partie d'échec.

L'horloge du *Zynq* alimente également les modules de *HDMI Modules* pour la synchronisation de ceux-ci.

Finalement, le *Zynq* est connecté à l'internet pour envoyer ses paquets vers la tablette *Android* et au *HDMI Modules* pour envoyer les images *bitmap* vers l'écran.

3.2 Architecture logicielle (Q4.3)

3.2.1 *Zedboard*

Nous avons divisé l'architecture en sept modules principaux.

1. *DrawHDMI*
2. *Bitmap*
3. *BoardDisplay*
4. *ChessBoard*
5. *REST*
6. *HTTP*
7. *TCP*

Les modules *DrawHDMI*, *bitmap* et *BoardDisplay* sont responsables de l'affichage. Le module *bitmap* est utilisé pour lire les fichiers *bitmap* en mémoire et extraire les informations de l'entête. Le module *DrawHDMI* est responsable de gérer un tampon qui correspond à l'écran. Il offre une fonction pour dessiner une image à l'écran, et une fonction pour dessiner du texte à l'écran.

Le module *BoardDisplay*, pour sa part, utilise le module *Bitmap* pour charger des images dans une zone mémoire allouée à la compilation. Il fournit des fonctions pour recevoir une position d'échecs en plus d'autres informations et fait des appels à *DrawHDMI* pour dessiner la position de l'échiquier à l'écran.

Le module *ChessBoard* gère tout ce qui s'apparente au concept réel du jeu d'échecs, c'est-à-dire les positions des pièces sur l'échiquier (appelé «la position»), les règles du jeu, l'horloge (dans un sous-module *ChessClock*), et les

informations de la partie (noms des joueurs, événements, rondes etc…). Ce module fournira, dans son interface, des fonctions pour jouer des coups, régler les informations de la partie et autres. Il passera la position à *BoardDisplay* pour être affichée lorsque celle-ci est modifiée.

Le module *REST* est responsable de la communication haut-niveau et de la gestion des requêtes. Lorsqu'une requête *HTTP* arrive, son contenu est envoyé au module *REST* qui la traduira en appels de fonctions du module *ChessBoard*. Le module offrira aussi des fonctions qui seront appelées par *ChessBoard*; lorsque *ChessBoard* veut informer les tablettes d'un événement relié au jeu, il appellera une de ces fonctions et le module le traduira en une requête en format *REST* qui sera envoyée par *HTTP*.

Le module *HTTP* servira à décoder une requête reçue (enlever l'entête et passer le contenu au module *REST*) et à envoyer des requêtes reçues du module *REST* (ajouter une entête *HTTP* et passer la requête au module *TCP*).

Finalement, le module *TCP* fournit une gestion des connexions réseau et une couche d'abstraction pour permettre au module *HTTP* d'envoyer des transmissions *HTTP* sans se préoccuper de la librairie.

L'image de la page suivante résume le flot de données détaillé ci-haut :

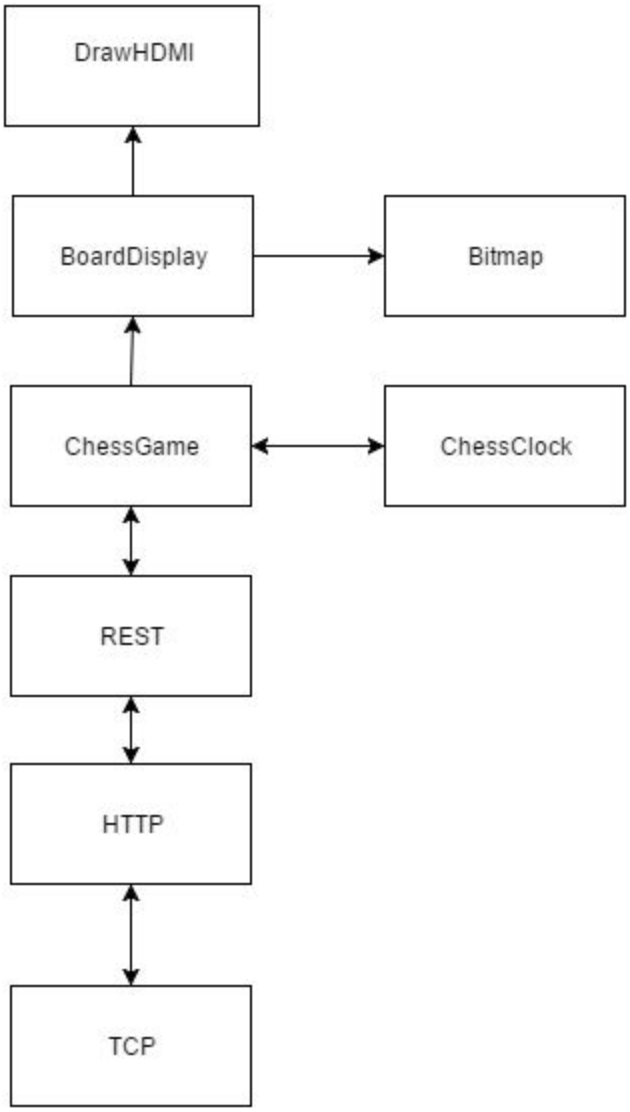

Image 2: Relation entre les modules de l'architecture logicielle du serveur

3.2.2 Partie tablette

L'architecture de la partie *Android* est écrite en *Java*, ce qui permet l'implémentation d'une structure orientée objet. Nous allons également profiter de l'architecture proposée par *Android* pour contrôler les affichages, c'est-à-dire les dérivations de la classe *Activity*. Nous allons donc créer une dérivation de la classe *Activity* pour chaque fenêtre d'affichage. Voici la liste de ces classes :

- *ActivityMenu* : Cette classe illustre la fenêtre d'affichage qui apparaîtra au démarrage de l'application. Cette fenêtre affichera les contrôles dirigeant vers d'autres fenêtres :
    - *buttonToGame* : bouton redirigeant vers l'écran précédant l'ouverture d'une partie. Si aucune partie n'est en cours, *ActivityCreate* est lancé. Si une partie a été créée et personne n'y participe, *ActivityJoinGame* est lancé. Sinon, un message d'erreur est affiché à l'écran.
    - *textBoxIP* : champ de texte dans lequel sera inscrite l'adresse *IP* du serveur à utiliser.

- *ActivityCreateGame* : Cette classe implémente la fenêtre d'affichage de la sélection d'options de création de jeu. Ces options seront montrées sous un format différent en fonction de la variable à établir :
    - *textBoxCode* : champ de texte précisant le code secret de la partie à créer.
    - *radioMode* : boutons radio permettant de définir si la partie se jouera sur une ou deux tablettes.
    - *textBoxLimitedTime* : champ de texte définissant le temps alloué à chaque joueur.
    - *textBoxOverTime* : champ de texte définissant le temps supplémentaire alloué.
    - *textBoxNbCoupsLimite* : champ de texte définissant le nombre de coups permis par le roi avant que la partie soit déclarée nulle.
    - textBoxLocation : champ de texte pour définir l'endroit où la partie se déroule.
    - *buttonPlay* : bouton qui redirige vers *ActivityGame*.

- *ActivityJoinGame* : Cette classe affiche la fenêtre du joueur qui veut joindre la partie créée par une autre tablette. Ses champs sont bien-entendu moins nombreux que ceux de *ActivityCreateGame* :
    - *textBoxCode* : champ de texte dans lequel il faut entrer le code secret entré par le créateur de la partie.
    - *buttonPlay* : bouton qui redirige vers *ActivityGame*.
- *ActivityGame* : Cette classe accueille l'interface de la partie d'échec.
    - *radioAppearance* : boutons radio permettant le choix d'apparence de l'échiquier
    - *labelRemainingTime* : affichage de texte précisant le temps restant à la partie.

    - *labelRemainingTurns* : affichage du nombre de coups restants avant la partie nulle.
    - buttonAbandon : bouton permettant de quitter en abandonnant la partie. Ce bouton se charge de s'assurer que l'utilisateur veut vraiment quitter en affichant un message de confirmation.

La suivante classe doit être créée pour gérer l'affichage de l'échiquier de la partie sur la fenêtre *ActivityGame* :

- *Game* : classe de gestion d'affichage de la partie. À chaque mouvement de l'utilisateur, cette classe reçoit la commande envoyée et l'envoie au serveur. Si le coup est valide, la pièce déplacée est placée au bon endroit et le joueur est en attente du coup de son opposant. La classe possédera donc deux fonctions principales :
    - *manageMove()* : cette fonction sera appelée par *ActivityGame* et prendra en paramètres les coordonnées de déplacement d'une pièce. Une requête *POST* sera envoyée au serveur pour demander si le coup est légal. S'il ne l'est pas, un message d'erreur est envoyé au joueur. S'il l'est, une confirmation est envoyée et le jeu attendra une interruption du serveur disant que l'opposant aura joué.
    - *manageOpponentMove()* : cette fonction consistera à gérer un envoi du serveur du coup joué par l'opposant.

Chacune de ces classes d'affichage fera appel au serveur pour envoyer ou recevoir des informations au serveur. Ces informations seront traitées par les classes suivantes :

- *HttpRequest* : classe statique générale permettant de gérer des requêtes *HTTP*. Elle est dérivée par les deux classes suivantes :
- HttpGetRequest : classe dérivée de *HttpRequest* permettant de gérer tous les envois de requête *GET* et la réception de leur réponse.
- HttpPostRequest : classe dérivée de *HttpRequest* permettant de gérer tous les envois de requête *POST* et la réception de leur réponse. Contrairement à *HttpGetRequest*, la réponse à ce type de requête précisera si une erreur est survenue ou pas.

Les différentes classes qui gèrent l'interface utilisateur génèrent une requête de type *REST* avec les différents paramètres. Cette requête est créée lors de l'appel aux fonctions *onClick*()*. En effet, une fois que le bouton de chaque interface est cliqué les informations sont envoyées au serveur. La requête a une entête spécifique et un corps qui est sous forme d'un fichier *JSON,*

qui contient les différentes informations. Les différentes requêtes sont envoyées au serveur grâce au module *HTTP*. Le serveur également envoie des informations au client en utilisant le module *HTTP*.

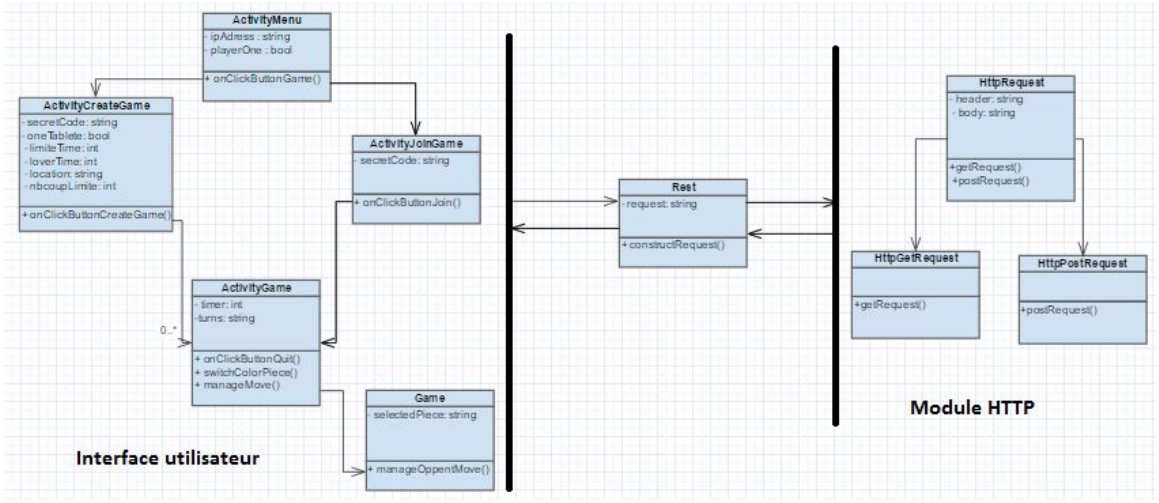

Image 3: Communication de l'interface utilisateur avec le module *HTTP*

4. Processus de gestion

4.1 Estimations des coûts du projet

Voici les estimations des coûts selon le budget et la charge de travail imposés dans *Demande de proposition no. A2016-INF3995 : Conception d'un jeu d'échecs électronique*[2]:

- Salaire d'un développeur-analyste : 130$/h
- Salaire d'un coordonnateur de projet : 145$/h
- Charge de travail: 420 heures-personne
- Nombre de personnes dans l'équipe: 6
- Ratio développement/gestion : 1/14 (30 heures-personne)

On peut donc estimer les coûts totaux du projet à 55 050$, selon la formule suivante:

*Coûts totaux = (13/14 * 130$/h + 1/14 * 145$/h) * 420 h-personnes*

= 55 050$

4.2 Planification des tâches (Q2.2)

Nous avons divisé le travail en plusieurs tâches, selon les modèles illustrés en *3.2*. Deux membres de l'équipe se concentreront sur l'implémentation du client, tandis que le reste de l'équipe travaillera sur le serveur.

Le client peut être divisé en quatre principales tâches: l'interface graphique, le contrôle du jeu d'échec, la gestion *HTTP* et la gestion *REST*.

Le serveur, quant à lui, requiert beaucoup plus d'effort. En effet, chaque module de la section *3.2* constitue sa propre tâche, ainsi que les ajouts pour le livrable 2, spécifiés dans la section *1.3*. Voici le diagramme de gantt illustrant la relation entre les tâches:

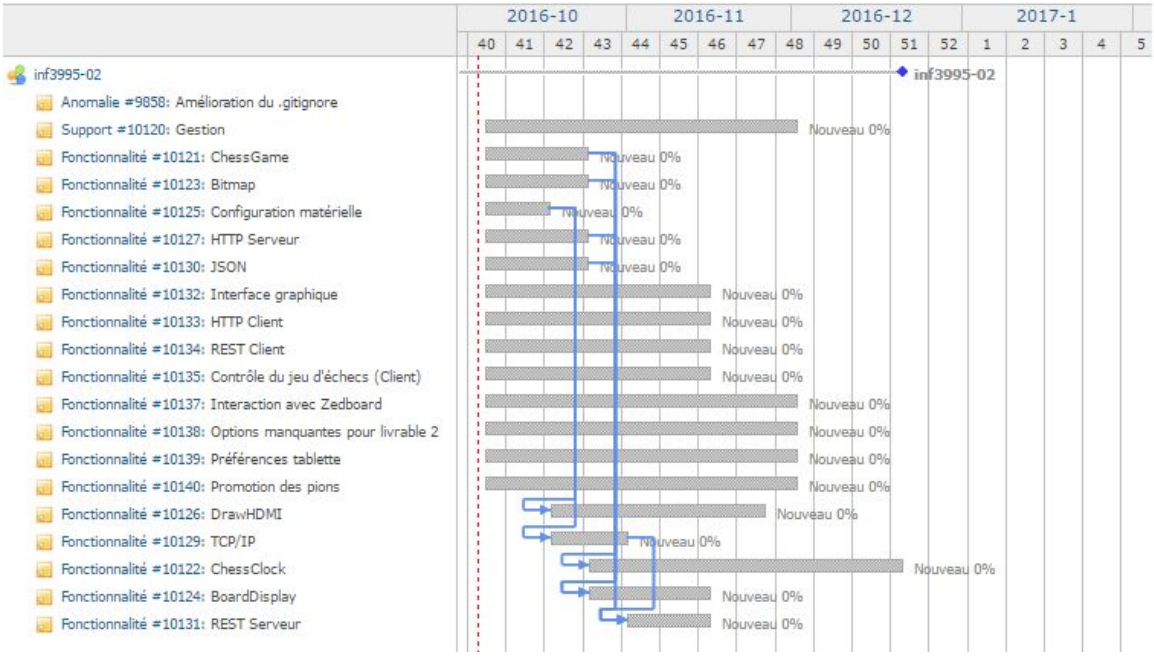

Image 4: Diagramme de gantt du projet

Nous pouvons également voir la progression grâce au *roadmap* :

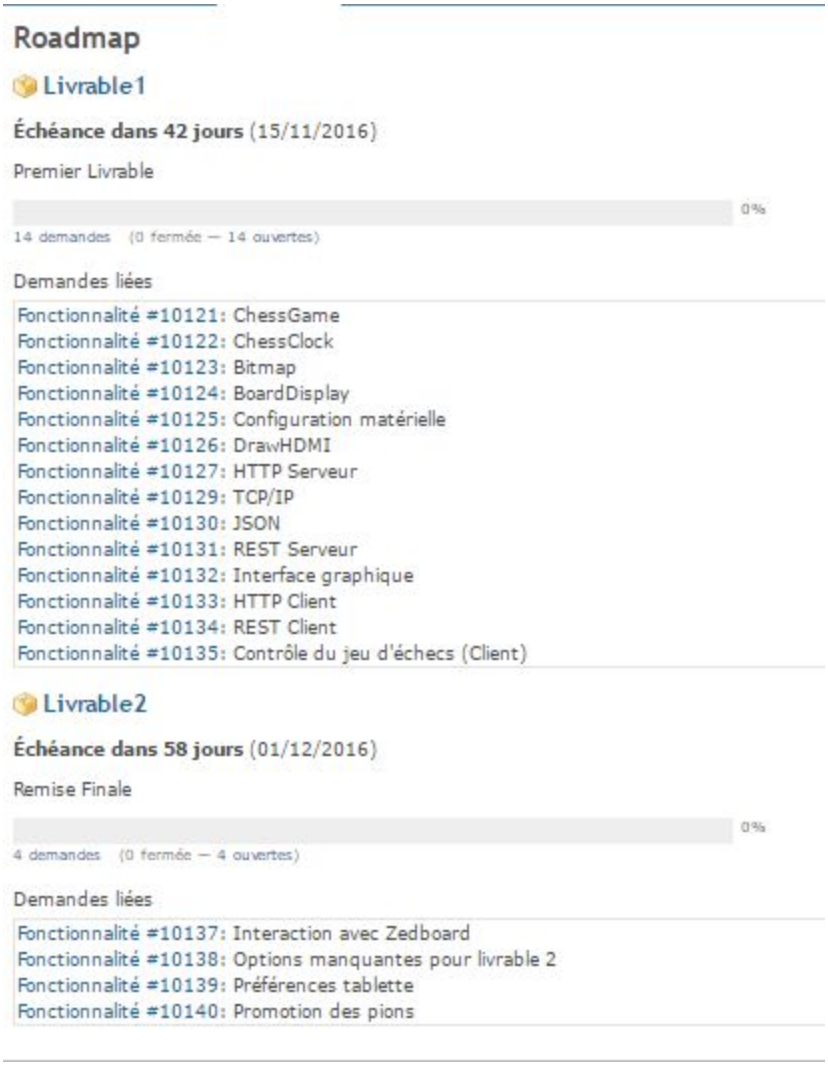

Image 5: *Roadmap* du projet

4.3 Calendrier de projet (Q3.3)

En se basant sur les dates de terminaison des livrables 1 et 2, et en prenant en compte le diagramme de *Gantt* en *4.2*, nous sommes arrivés au présent calendrier :

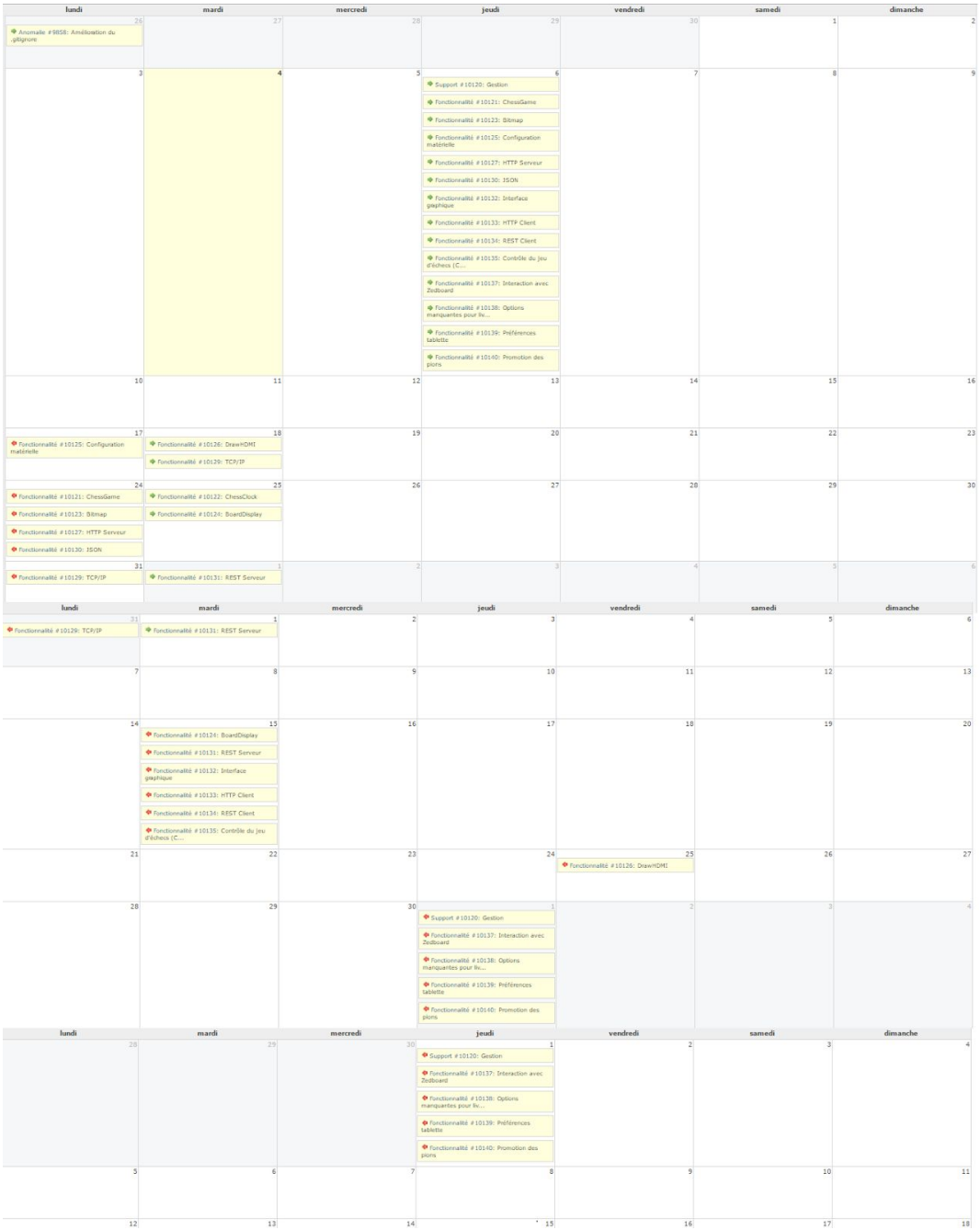

Image 6: Calendrier du projet

4.4 Ressources humaines du projet

Pour ce projet, nous avons besoin d'un coordonnateur de projet qui se connaît en *C* ainsi qu'en *Java Android* pour avoir une vision globale du projet. Nous avons aussi besoin de deux personnes ayant de l'expérience avec le développement *Android* et ayant un flair pour la conception d'interface usager et l'expérience utilisateur.

Nous avons besoin d'au moins deux programmeurs ayant de l'expérience avec les *FPGAs*, l'utilisation du logiciel *Vivado* et de très bonnes connaissances en *C* pour le développement du serveur.

Finalement, nous allons avoir de besoin d'un spécialiste réseau ayant de l'expérience en *Java* et en *C* pour s'assurer de la bonne communication et de la synchronisation de la tablette *Android* et du serveur *FPGA*.

5. Suivi de projet et contrôle

5.1 Contrôle de la qualité

Nous avons mis en place plusieurs méthodes de contrôle de la qualité pour nous assurer de pouvoir livrer un produit de qualité. Tout d'abord, nous avons instauré un guide de codage pour la partie *FPGA* codée en *C* ainsi qu'un guide de codage différent pour notre partie *Android*.

Par la suite, pour s'assurer que notre guide de codage soit appliqué et pour réduire le nombre de bogues, nous avons instauré un système de révision par les paires où chaque soumission vers le serveur doit être approuvée par les paires.

Nous avons également établi des règles de gestion de l'entrepôt pour assurer un standard de qualité qui sera discuté dans la section *5.4*.

Nous avons défini une liste de tests en *5.3* que nous allons appliquer à chaque version du code que nous allons déposer dans la branche *master* qui est réservée pour les remises des livrables.

5.2 Gestion de risque (Q2.6)

Principaux risques du projet:

- Dépassement des coûts
- Dépassement du temps de développement

- Changement des spécifications par le client
- Nouveau matériel

Solution :

- Calendrier détaillé des différents livrables et de leur évolution durant le développement
- Utilisation technique *Agile* comme méthode de développement pour être plus flexible en cas de changement des requis
- Utilisation des expertises de chaques personnes pour maximiser le travail tout en minimisant le temps consacré

5.3 Tests (Q4.4)

Nous allons avoir deux types de test pour l'application, soit des tests unitaires sur chacun des modules qui seront testés à l'exécution de l'application de façon courante et des tests plus exhaustif sur les fonctionnalités de chaque livrable.

Pour les tests modulaires sur nos parties en *C*, nous allons utiliser la librairie *cunit*. Les tests vont être définis dans un module de tests et il seront appelés dans la routine principale pour assurer que nos modules fonctionnent correctement.

Pour tester les classes de l'architecture *Android*, nous allons préconiser des tests unitaires en boîte noire également avec *junit*. Les tests seront déclarés dans une classe de test indépendante qui sera exécutée au lancement de l'application *Android* en mode *debug* et durant le développement.

Pour tester l'application et s'assurer de ne pas avoir de régression au niveau des fonctionnalitées, nous allons avoir un plan de tests fonctionnel. Ce plan comporte une liste de toutes les fonctionnalitées implémentées jusqu'à présent. Cette liste sera dérivée de la liste de nos artéfacts fonctionnels du livrable tel que décrit dans la section *1.3* .

5.4 Gestion de configuration

Nous avons choisi d'utiliser *Git* comme logiciel pour le système de contrôle de version. Dans *Git*, nous avons choisi d'utiliser la fonctionnalité des branches pour mieux organiser le code. Nous avons la branche *master* qui contient les versions stables qui ont été rigoureusement testées pour la remise du livrable. Par la suite, nous avons la branche *develop* qui sert à contenir le code de production avec certaines fonctionnalités complètes. Nous avons également les sous-branches de fonctionnalités qui servent à développer de nouvelles

fonctionnalités instables, sans briser notre code stable. Ces branches suivent le pipeline suivant : *master<-develop<-feature*.

Pour les livrables, dans le dossier racine, nous allons avoir un dossier *livrable* qui lui-même contiendra un dossier *livrable 1* et *livrable 2* qui eux-même contiendront deux sous-dossiers : *client* et *serveur*. Chaque sous-dossier aura sa propre arborescence de fichiers avec ses fichiers *.gitignore*. Finalement, nous aurons également un fichier *readme* dans le dossier racine du *Git* pour expliquer le fonctionnement et la structure de l'entrepôt et les procédures pour construire le serveur et le client.

6. Références (Q3.2)

[1] Jérôme Collin, *« INF3995 - Projet de conception d'un système informatique » [notes de cours de Jérôme Collin]*, Montréal : École Polytechnique de Montréal, Automne 2016.

[2] Jérôme Collin, Christian Harper-Cyr, Aymen Djellal ; *« INF3995 - Demande de proposition no. A2016-INF3995 - Conception d'un jeu d'échec électronique », Montréal* : École Polytechnique de Montréal, Automne 2016

[3] Jérôme Collin, Christian Harper-Cyr, Aymen Djellal ; *« INF3995 - Exigences techniques - Conception d'un jeu d'échec électronique »*, Montréal : École Polytechnique de Montréal, Automne 2016

[4] AVNET electronics marketing, *« ZedBoard (Zynq™ Evaluation and Development Hardware) User's Guide »*