Université de Montpellier
L2 informatique

Arcus

Un outil pour la synchronisation multi-services de stockage dans les nuages

Rapport de projet T.E.R.
Projet informatique — HLIN405

Étudiants :
M. Rémi Cérès
M. Mattéo Delabre

Encadrante :
Mme Hinde Bouziane

Année : 2016 – 2017
Soutenance le : 07/06/2017

Réseau Figure
CURSUS MASTER EN INGÉNIERIE

Nous tenons à remercier notre encadrante Mme Hinde Bouziane pour son accompagnement, sa bienveillance et ses conseils précieux tout au long de ce projet qui nous ont permis de le porter à son état actuel. Nous la remercions pour sa relecture attentive de ce rapport.

Table des matières

Introduction

Dans le cadre du module de projet informatique du second semestre de L2, nous avons développé une application permettant la synchronisation et la répartition des données d'un utilisateur entre plusieurs services de stockage dans les nuages.

Notre groupe est composé de deux personnes, Rémi Cérès et Mattéo Delabre, et nous sommes encadrés par Mme Hinde Bouziane. La réalisation du projet s'est déroulée sur une période de 15 semaines, de janvier à mi-mai 2017.

Motivations du projet

Le stockage de données dans les nuages consiste pour un utilisateur à confier ses données à un service qui s'occupe de les stocker pour lui, et de les lui restituer dès qu'il en a besoin. Un tel service est généralement opéré par une entreprise qui est responsable de l'intégrité des informations déposées.

Ce mode de stockage offre de nombreux avantages, permettant par exemple à l'utilisateur d'accéder à ses données depuis plusieurs appareils, de conserver une copie de sauvegarde de celles-ci, voire d'en partager certaines avec d'autres personnes. Ce modèle s'est considérablement développé ces dernières années, et diverses entreprises telles que Amazon, Google ou Microsoft proposent de stocker gratuitement les données de leurs utilisateurs.

Malgré les nombreux avantages du stockage dans les nuages, il cause une centralisation des données de l'utilisateur sur un seul service et une dépendance sur celui-ci. L'utilisateur place en effet ses données entre les mains d'une unique entreprise, et par ailleurs sa confiance dans la bonne gestion de ses informations par cette entreprise.

Cette centralisation le fait dépendre du service et de la société qui le gère, qui peut à tout moment décider de modifier son offre ou d'y mettre fin. Les services gratuits n'offrent par ailleurs souvent aucune garantie quant à la confidentialité des données déposées.

Objectifs du projet et cahier des charges

Nous avons baptisé notre projet Arcus, en référence au nuage bas du même nom, qui prend la forme d'un tube et est souvent associé aux *cumulus* et *cumulonimbus*. L'objectif est de permettre à un utilisateur de synchroniser les répertoires de son choix entre sa machine et différents services de stockage dans les nuages. En agissant comme un tube qui ferait le lien entre ces différents services, il se veut une réponse aux problématiques énoncées précédemment.

Cet objectif est donc similaire à celui d'autres applications de synchronisation comme celle de Dropbox, Nextcloud ou Apple, mais se distingue notamment par le fait que les données soient réparties sur différents services au contraire des moteurs de synchronisation classiques qui ne les synchronisent qu'avec un service particulier.

Nous avons décidé du cahier des charges suivant pour notre application.

Interface commune aux services de stockage dans les nuages. Notre application sera indépendante des services utilisés. Habituellement, chaque fournisseur de service propose sa propre application de synchronisation. Dans le cas où l'utilisateur se sert de plusieurs services, il doit installer plusieurs applications ayant le même but. Un objectif pour notre outil est de pouvoir fonctionner avec le plus de services différents possibles.

Répartition des données de l'utilisateur. Chaque fichier déposé par l'utilisateur dans un des répertoires choisis sera découpé en plusieurs blocs, qui seront envoyés sur différents services de stockage existants. Dans le cas où un de ces services serait ajouté ou supprimé, les données seraient alors redistribuées afin que toutes les informations restent accessibles. Cette répartition se fera de telle sorte que le service avec le plus d'espace libre soit priorisé, afin d'équilibrer au mieux l'espace occupé sur chaque service.

Réplication des données de l'utilisateur. Les blocs de données seront stockés de manière redondante sur au moins deux services différents quand cela sera possible. Cela permettra de se prémunir d'une panne d'un des services dépositaires ou de tout événement entraînant une perte de l'accès aux données.

Chiffrement des données de l'utilisateur. Avant de transmettre les blocs de données aux serveurs, ceux-ci seront chiffrés. Ce chiffrement garantira la confidentialité des données de l'utilisateur puisqu'il empêchera les services dépositaires et toute autre personne tierce d'accéder au contenu des blocs.

Transparence de fonctionnement de l'application. Les opérations de synchronisation, de répartition et de chiffrement des données s'exécuteront en fond. Les interactions directes avec l'utilisateur se limiteront à la gestion des fichiers par l'intermédiaire de l'explorateur de fichiers, à la résolution de conflits éventuels et à la configuration de l'application.

Notre application permettra ainsi de réduire le risque de perte des données de l'utilisateur en les répartissant et en les répliquant à travers plusieurs fournisseurs de services de stockage. En combinant ce mécanisme de répartition au chiffrement, elle accroîtra la confidentialité de ces données. Enfin, elle permettra de fédérer l'accès à plusieurs services en une seule application et de bénéficier des ressources de stockage de plusieurs fournisseurs combinés.

Quelques définitions

Cloud : ressources de calcul et de stockage matérielles et logicielles mises à disposition sous forme de services à travers un réseau.

Stockage distribué de fichiers : système de fichiers partagés de telle sorte qu'ils soient composés de plusieurs blocs répartis sur plusieurs ordinateurs connectés en réseau.

Moteur de synchronisation : application permettant de mettre en correspondance un système de fichiers local avec un distant.

Dans un premier temps, nous présenterons les méthodes et outils de travail que nous avons adoptés pour organiser ce projet. Ensuite, nous détaillerons la modélisation que nous avons choisie pour l'application, puis les technologies utilisées pour son implémentation. Enfin, nous ferons le point sur l'avancement du projet, les difficultés que nous avons rencontrées jusqu'ici et les perspectives qui s'offrent à nous pour la poursuite du développement d'Arcus.

Chapitre 1
Organisation du projet

1.1 Méthode et organisation du travail

Lors du développement d'Arcus, nous avons décidé de travailler un maximum de temps ensemble et de manière très régulière. Pour que nous puissions profiter tous deux des nombreuses connaissances que nous apporte le développement de ce projet, nous avons favorisé la méthode de programmation en binôme (ou *pair programming*). Cette méthode consiste à donner la responsabilité de l'écriture du code à l'un des deux développeurs et à faire relire le code produit au fur et à mesure par le second développeur. Une étude menée par l'IEEE Computer Society (McDowell, Werner, Bullock et Fernald, 2003) a en effet montré que les étudiants utilisant cette méthode produisaient du code de meilleure qualité.

Afin d'être le plus efficace et d'avancer le plus rapidement possible nous nous sommes réunis quotidiennement. Durant les jours de la semaine, nous nous sommes vus entre 16 h 30 et 19 h 30, afin de faire le point sur l'avancement du projet, de définir les objectifs du jour et de les réaliser. Enfin, chaque week-end, nous avons réalisé les tâches en attente que nous n'avions pas pu faire durant la semaine.

Toutes les deux semaines, nous nous sommes réunis avec notre encadrante Mme Hinde Bouziane afin de faire le point sur l'état d'avancement de l'application. Cette réunion bihebdomadaire nous a également permis de bénéficier de ses conseils et de son aide sur les difficultés que nous avons rencontrées lors du développement.

1.2 Répartition du travail dans le temps

Nous avons découpé le développement du projet Arcus en deux périodes de temps. La première s'inscrit dans le cadre du T.E.R., et s'est déroulée de janvier à mi-mai 2017. La seconde aura lieu pendant un stage au Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (Lirmm) encadré par Mme Hinde Bouziane de fin-mai à fin-juin 2017. Par ailleurs, nous avons déjà effectué une semaine de stage au laboratoire au cours de la première semaine des vacances de printemps.

Le but de la première période est d'obtenir une première version fonctionnelle de l'application qui répond à une partie du cahier des charges : la mise en place d'une interface commune aux différents services de stockage, la répartition des données de l'utilisateur sur ces services, leur réplication et le fonctionnement en tâche de fond.

Ce rapport rend compte de notre avancement au cours de la première partie, c'est-à-dire pendant le projet T.E.R. Nous avons découpé cette période de travail en plusieurs phases.

1. **Préparation du projet.** Nous avons écrit le cahier des charges de l'application, choisi les outils de travail et les principales technologies utilisées. Nous avons fait une première version du diagramme de répartition des tâches dans le temps, et une première modélisation de l'architecture de l'application.
2. **Développement du projet.** Nous avons implanté les fonctionnalités de l'application en raffinant la modélisation au fur et à mesure. Pour chaque module implanté, nous nous sommes efforcés d'écrire des tests afin de s'assurer de leur bon fonctionnement.
3. **Finalisation du projet.** Cette phase a consisté en la correction de bogues afin d'obtenir une version suffisamment stable pour pouvoir être présentée en vue de la soutenance et du rendu du projet T.E.R.

L'annexe A détaille cette répartition sous forme d'un diagramme de Gantt.

1.3 Outils de travail collaboratif

Nous avons choisi d'utiliser Git [1] au travers du serveur GitLab [2] hébergé par le Service Informatique de la Faculté (Sif). Le logiciel Git, libre, permet la gestion des versions du projet et facilite la collaboration entre nous, notamment lorsque nous travaillions en même temps sur deux machines différentes. Les serveurs GitLab sont quant à eux basés sur un logiciel libre également et le service est fourni gratuitement par le Sif.

Pour communiquer entre nous à distance, nous avons utilisé Telegram [3] pour les messages écrits et appear.in [4] pour la communication orale et le partage d'écrans.

Pour prendre des notes pendant les réunions, nous avons utilisé le langage Markdown [5] qui permet un formattage rapide. Pour rédiger les différents documents, y compris ce rapport, nous avons utilisé LaTeX [6] pour sa capacité à produire des documents de bonne qualité.

Enfin, pour éditer le code du projet et les documents, nous nous sommes servis de l'éditeur de texte Atom [7], logiciel libre développé par GitHub. Il s'agit en effet de l'éditeur que nous utilisons habituellement.

1. Git : `https://git-scm.com/`
2. GitLab : `https://gitlab.info-ufr.univ-montp2.fr/matteodelabre/arcus/`
3. Telegram : `https://telegram.org/`
4. appear.in : `https://appear.in/`
5. Markdown : `https://daringfireball.net/projects/markdown/`
6. LaTeX : `https://www.latex-project.org/`
7. Atom : `https://atom.io/`

Chapitre 2

Conception d'Arcus

Nous avons commencé par décider d'une architecture globale pour l'application. Cette architecture est résumée dans la vue d'ensemble en figure 2.1.

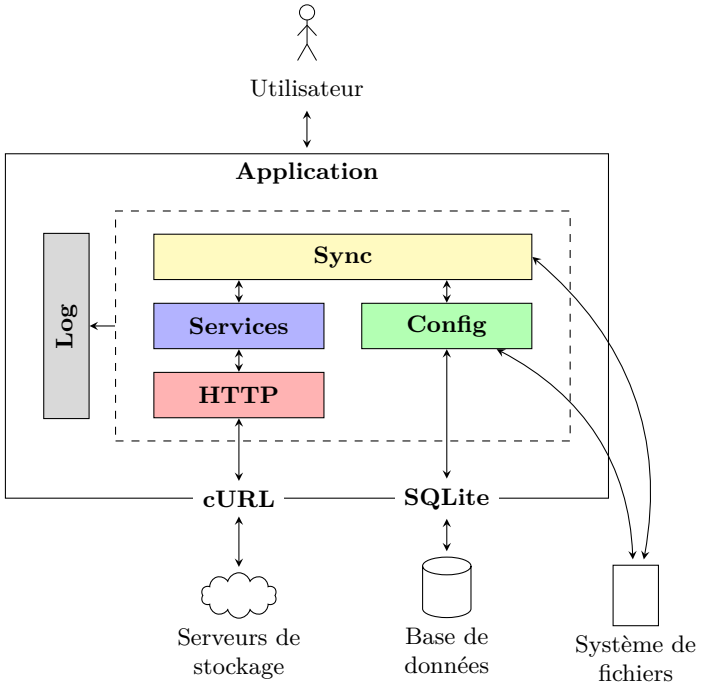


FIGURE 2.1 – **Vue d'ensemble de l'architecture de l'application.** Au centre, dans le cadre « Application », sont présentés les différents modules participant à l'architecture de l'application. Le niveau d'abstraction des modules va en croissant vers le haut du schéma. Les flèches représentent les interactions internes à l'application et celles entre les acteurs et ressources externes et l'application.

Notre application a besoin de communiquer avec les interfaces de programmation (ou A.P.I.) des différents services de stockage de fichiers. Ces services fournissent en général une interface de communication utilisant *HyperText Transfer Protocol* (HTTP). C'est un « protocole applicatif pour les systèmes d'information distribués, collaboratifs utilisant des documents riches » (Fielding et al., 1999), donc particulièrement adapté au transfert de documents. Afin de communiquer avec ces services, notre application doit contenir un module de transmission de données sur HTTP, symbolisé par **HTTP** dans la vue d'ensemble.

Bien que les fournisseurs de services de stockage connus tels que Nextcloud, OneDrive ou Dropbox utilisent HTTP, ils ont adopté des architectures hétérogènes qui font qu'il est difficile d'écrire un code unique permettant de communiquer avec toutes leurs interfaces. Dans notre application, nous devons donc construire une couche d'abstraction de ces hétérogénéités. Ce module, symbolisé par **Services** dans la vue d'ensemble, s'appuie sur les fonctionnalités du module HTTP.

Comme chaque utilisateur de la machine cliente peut choisir les répertoires qu'il souhaite synchroniser via l'application, la liste de ces répertoires doit être sauvegardée à un endroit de la machine. Ces informations sont regroupées dans une configuration dite locale spécifique à chaque utilisateur. Il existe par ailleurs des informations spécifiques à chaque répertoire synchronisé, comme par exemple la liste des fichiers présents ou la liste des services connectés. Ces informations sont quant à elles regroupées dans une configuration synchronisée sur les différentes machines d'un même utilisateur. Nous avons décidé d'isoler cette fonctionnalité dans un module de configuration, symbolisé par **Config** dans la vue d'ensemble.

En s'appuyant sur ces différentes abstractions, nous pouvons mettre en place le moteur de synchronisation, cœur de l'application. Ce moteur doit permettre l'envoi des données contenues dans les répertoires sélectionnés par l'utilisateur vers les services de stockage connectés. Réciproquement, il doit permettre le rapatriement de celles-ci depuis les services distants vers la machine de l'utilisateur. Ce module est symbolisé par **Sync** dans la vue d'ensemble.

Enfin, nous avons décidé qu'il était nécessaire de créer un module de journalisation. Grâce à ce module, les différents composants de l'application seront en mesure d'écrire des informations sur un journal global dans un format unifié. Ce journal global, qui pourra être redirigé vers la sortie standard ou un fichier, aidera au débogage en cas de défaillance de l'application. Dans la vue d'ensemble de l'architecture, il est symbolisé par **Log**.

2.1 Transmission des données aux serveurs

Le module HTTP permet l'échange de données entre un *client* et un *serveur* suivant le protocole HTTP. Ce protocole permet au client d'envoyer des demandes de données au serveur et de recevoir des réponses, mais pas l'inverse : c'est toujours le client qui demande des données au serveur. Pour ce faire, le client formule une requête, composée notamment d'un verbe, d'une *Uniform Resource Locator* (URL) et d'une liste d'en-têtes. Le verbe spécifie l'action qui doit être entreprise sur la ressource désignée par l'URL. Les en-têtes servent à donner plus de détails sur la requête. La norme HTTP/1.1 (Fielding et al., 1999) définit les verbes principaux suivants :

`GET :` récupérer les données contenues dans la ressource désignée par l'URL ;

`HEAD :` récupérer les caractéristiques associées à la ressource désignée par l'URL ;

`POST :` envoyer des données au serveur ;

`PUT :` créer une ressource à l'URL donnée ;

`DELETE :` supprimer la ressource désignée par l'URL.

Une fois la requête formulée par le client dans le module Services, celle-ci est transmise au serveur choisi. Ce serveur doit à son tour formuler une réponse qui satisfait la requête du client et la lui retourner. Ce mode de fonctionnement nous a amené à choisir la modélisation présentée dans la figure 2.2 pour le module HTTP.

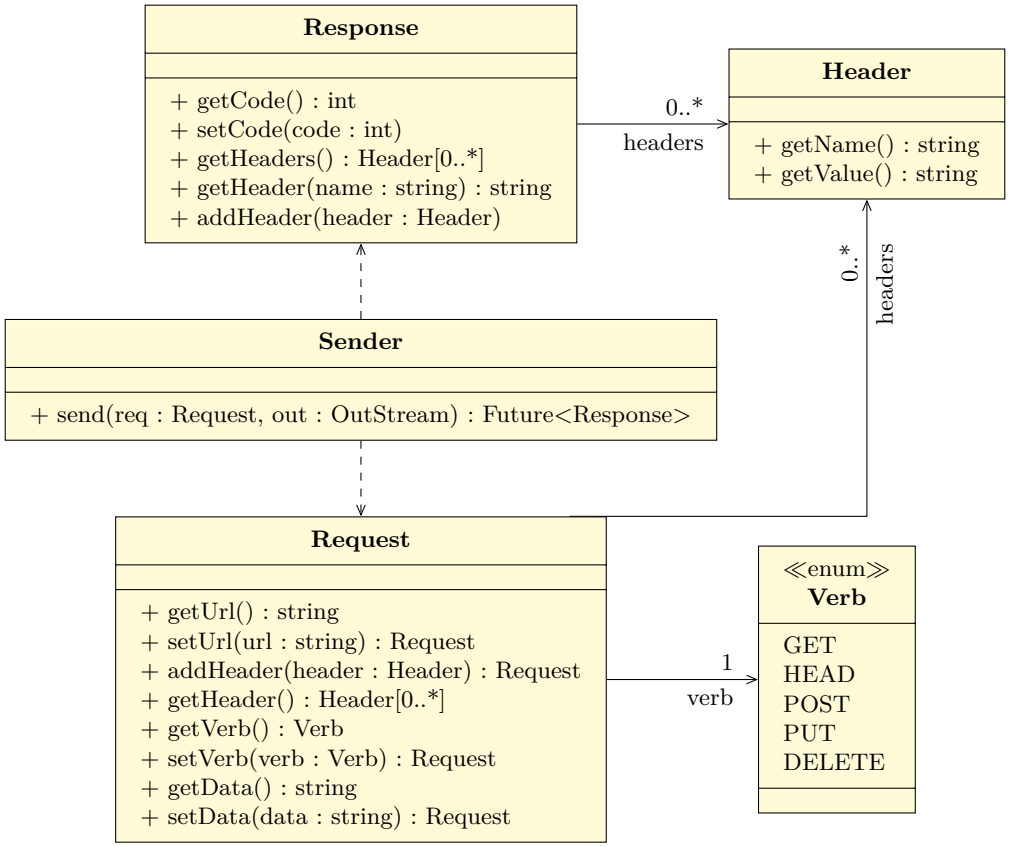


Figure 2.2 – **Modélisation UML du module HTTP.** La classe `Sender` est au cœur du module, elle permet d'envoyer des requêtes et d'obtenir de façon asynchrone une réponse de la part du serveur. Cela signifie que le reste de l'application peut continuer à s'exécuter pendant que l'on attend la réponse du serveur à une demande. Ces requêtes doivent préalablement être construites selon le modèle donné par la classe `Request`. Les réponses fournies par le serveur suivent le modèle de la classe `Response`.

2.2 Définition d'une interface multi-services

Le module Services a pour but d'abstraire le dialogue avec les interfaces de communication des différents services de stockage. Pour créer ce module, nous avons étudié les interfaces de deux fournisseurs de services majeurs : Dropbox et OneDrive. Nous avons constaté que chaque fournisseur avait une interface qui divergeait de celle de ses concurrents, faute de standard établi.

Par exemple, avec un compte OneDrive, pour obtenir des informations sur le quota alloué et son taux d'utilisation, il faut formuler une requête HTTP avec le verbe `GET` sur l'URL `https://graph.microsoft.com/v1.0/me/drive` (Microsoft, 2015). À l'inverse, avec un compte Dropbox, pour obtenir les mêmes informations, la requête doit utiliser le verbe `POST` et porter sur l'URL `https://api.dropboxapi.com/2/users/get_space_usage` (Dropbox, 2017).

Nous voyons sur ces exemples que les verbes HTTP ou les URL à utiliser pour accéder aux ressources sur les services sont variables. Toutefois, parmi les services de stockage de fichiers les plus connus, une large majorité a adopté une architecture d'interface de communication dite en *Representational State Transfer* (Rest). Ce type d'architecture, théorisée par Fielding en 2000, repose sur les principes suivants.

Utilisation sémantique de HTTP. L'interface utilise le protocole HTTP décrit dans le module précédent. Elle associe sémantiquement les verbes du protocole à chaque action réalisable ; par exemple, elle utilise le verbe `GET` lorsqu'il s'agit de récupérer des informations ou `DELETE` pour en supprimer. Elle utilise les URL pour désigner des ressources et non pas, par exemple, pour désigner une page de connexion.

Échange de requêtes et réponses descriptives. Toutes les informations nécessaires au traitement d'une requête du client sont contenues uniquement dans celle-ci. Le serveur n'utilise pas d'informations contextuelles qui seraient liées aux requêtes précédemment effectuées. Il renvoie des réponses descriptives pouvant être traitées en tant que telles sans avoir besoin d'effectuer une requête supplémentaire.

Exclusivité d'accès aux données via le serveur. Le serveur est le seul ayant accès à la base de données ou aux informations quelles qu'elles soient. Le client n'y accède que par l'intermédiaire du serveur. Cela simplifie notamment la gestion de la cohérence des données puisque seul le serveur en est responsable.

Les interfaces de OneDrive et Dropbox que nous avons étudiées s'efforcent de suivre ces principes, même s'il subsiste des incompatibilités. Nous avons décidé de profiter du fait que le protocole HTTP soit commun à une très grande majorité des interfaces de communication des services de stockage pour construire notre abstraction. Les incompatibilités nous ont amené à conclure qu'il serait très difficile d'écrire un code unique pour accéder aux différents services.

Nous avons donc créé une interface commune pour accéder à celles des services de stockage, puis réalisé différentes implémentations de cette interface, une par service supporté. Cette modélisation nous permettra d'intégrer plus facilement de nouveaux services par la suite sans devoir modifier le code des autres modules. La figure 2.3 présente l'architecture du module Services.

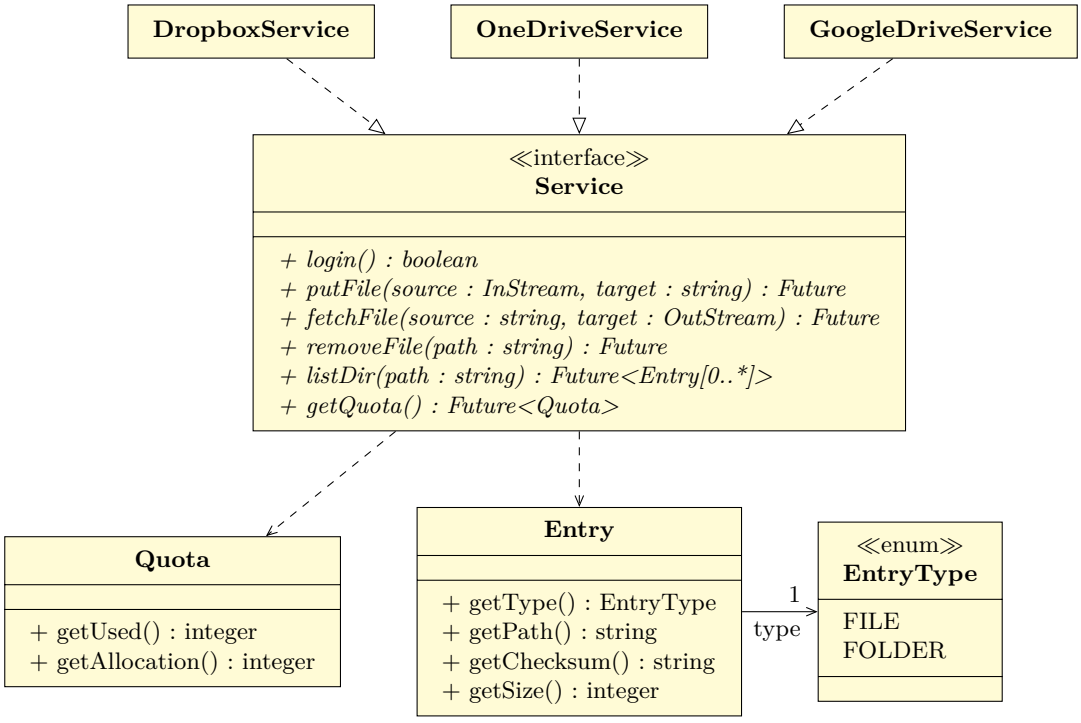


Figure 2.3 – **Modélisation UML du module Services.** L'interface `Service` est l'élément principal du module. Elle définit les actions communes que l'application doit être en mesure de réaliser sur chaque service : authentification, ajout d'un fichier, récupération d'un fichier, suppression d'un fichier, listage d'un répertoire, récupération du quota et de son utilisation. Toutes ces actions sont asynchrones.

2.3 Configuration locale et configurations synchronisées

2.3.1 Notion de zone de synchronisation

Nous avons décidé lors de la rédaction du cahier des charges de permettre à l'utilisateur de synchroniser les données d'un ensemble de répertoires (et de leurs sous-répertoires) de son choix. Nous désignerons dans la suite de ce rapport les données de chaque répertoire par l'expression « zone de synchronisation ».

Ces zones sont associées à un unique répertoire sur la machine de l'utilisateur, situé n'importe où dans son répertoire d'accueil. Elles ne peuvent pas se chevaucher : aucun fichier n'appartient à deux zones de synchronisation.

Pour chaque zone, l'application doit être en mesure de reconstituer les fichiers dispersés en blocs sur les différents services. Pour ce faire, elle a besoin de conserver des informations sur la répartition

de ces blocs. Elle a également besoin de pouvoir comparer l'état des zones de la machine actuelle avec celles du serveur, afin de réaliser la synchronisation.

En particulier, nous avons besoin pour chaque bloc de savoir de quel fichier il fait partie, l'ordre dans lequel il apparaît, les services de stockage sur lesquels il est stocké et peut être récupéré. Enfin, l'application a besoin de conserver les informations d'authentification pour chaque service.

2.3.2 Conception d'un index d'état

Pour répondre à ces besoins, nous avons choisi de mettre en place un index d'état pour chaque zone. Cet index est synchronisé avec chaque service de stockage utilisé, afin de pouvoir être récupéré par d'autres machines et servir de base à la reconstitution du répertoire associé. Il contient les informations sur les blocs, les fichiers et les services.

Chaque bloc fait partie d'un unique fichier et possède un identifiant unique indépendant de ce fichier. Il est stocké sur au moins un service. L'index stocke également le *hash* du bloc, correspondant à une somme de contrôle des données qu'il contient, et permettant de savoir si ces données ont changé. Enfin, chaque bloc possède un numéro d'ordre qui indique la position qu'il occupe dans le fichier original relativement aux autres.

Chaque fichier et chaque répertoire possède un identifiant unique indépendant de son chemin. Nous avons décidé d'utiliser une structure récursive pour stocker l'arborescence de la zone de synchronisation, dans laquelle chaque fichier ou répertoire est lui-même fils d'un autre répertoire. Chaque fichier possède un nom et contient au moins un bloc.

Enfin, les comptes de services de stockage que l'utilisateur a connectés à la zone sont stockés dans l'index. Ces comptes contiennent un ensemble de blocs de fichiers synchronisés. L'application conserve le nom du fournisseur du service en question ainsi que les informations d'authentification. Dans cette première modélisation de l'application, nous conservons les informations d'authentification en clair dans l'index, nous ne tenons donc pas compte de leur sécurité. Nous réglerons ce problème dans le futur en chiffrant l'index.

En raison de la complexité de la structure de l'index, nous avons choisi de le stocker sous forme d'une base de données relationnelle, dont le modèle entité-association est présenté en figure 2.4.

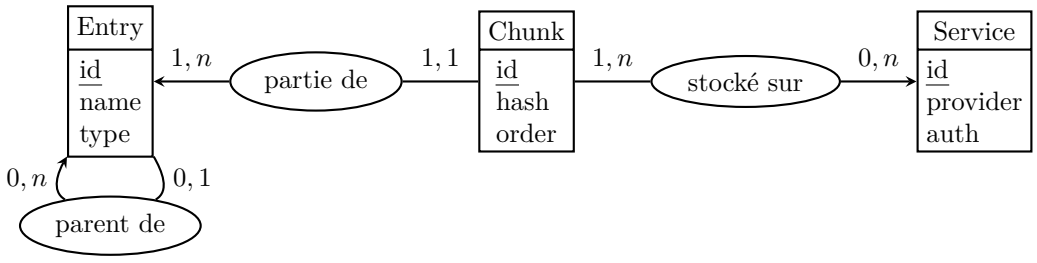


Figure 2.4 – **Modèle entité-association de la base de données pour l'index.**

De ce fait, nous avons choisi de considérer chaque zone de synchronisation comme une entité totalement indépendante, possédant sa propre configuration synchronisée avec différents services, sous forme d'une base de données.

Sur cet aspect, notre conception s'apparente à celle du logiciel Git qui définit des *dépôts*, semblables à nos zones de synchronisation, qui contiennent leur propre index (Rosenberg, 2010) et sont synchronisés avec un serveur distant. Nous divergeons cependant sur le fait que Git se concentre sur l'aspect de gestion de versions des données, ce que notre application ne prend pas en compte car cela ne correspond pas aux objectifs que nous nous sommes fixés.

Ce choix permet une plus grande flexibilité pour l'utilisateur. En effet, si nous avions choisi de centraliser le stockage des informations d'authentification des services de toutes les zones de synchronisation dans un seul fichier, il n'aurait par exemple pas été possible de synchroniser une zone avec deux services particuliers et une autre avec deux différents.

En choisissant de stocker le plus d'informations possible dans une configuration spécifique à chaque zone, l'application peut non seulement fonctionner dans le cas où ces zones ne partagent pas les mêmes services, mais également dans le cas où elles les partagent.

2.3.3 Conception d'une configuration locale

Comme expliqué avant, les zones de synchronisation sont associées, sur une machine, à un répertoire racine. Si l'on synchronise cette zone avec une autre machine, il y a de fortes chances pour que l'on souhaite l'associer à un répertoire ayant un chemin différent. Par exemple, le nom de l'utilisateur peut avoir changé entre les deux machines, ou elles peuvent ne pas fonctionner sur le même système d'exploitation. [1]

Pour gérer ces cas, nous avons décidé de créer un fichier de configuration local sur chaque machine et pour chaque utilisateur. Ce fichier contient les paramètres qui n'ont pas lieu d'être partagés par plusieurs machines.

Dans cette configuration, pour le moment seule la liste des associations entre chaque zone de synchronisation et son répertoire racine pour la machine actuelle est stockée. Nous pourrons y inclure plus de paramètres si le besoin se présente dans le futur. La figure 2.5 montre un exemple de mise en place de ces différentes configurations sur un système GNU/Linux.

2.3.4 Gestion des configurations dans l'application

Notre application doit ainsi gérer deux configurations, une configuration synchronisée spécifique à chaque zone, et une configuration locale pour chaque utilisateur, qui est dépendante de la machine utilisée. Pour ce faire, nous avons conçu un module de configuration dont la modélisation est présentée en figure 2.6.

1. Le système Windows place par exemple le répertoire d'accueil de l'utilisateur dans `C:\Users\utilisateur` alors que les dérivés de GNU/Linux utilisent généralement `/home/utilisateur`.

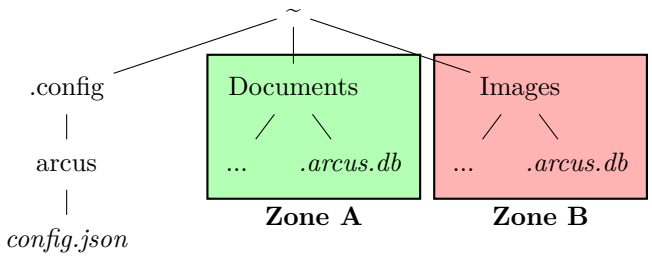


Figure 2.5 – **Exemple de répartition des différentes configurations de l'application.** Les fichiers dont le nom est en italique contiennent les configurations de l'application telles que décrites dans les parties précédentes. La configuration locale, unique pour l'utilisateur, est stockée dans `~/.config/arcus/config.json`. Dans cet exemple, l'utilisateur a choisi de synchroniser deux zones, A associée aux répertoire des documents et B associée aux répertoire images. Dans chacune de ces zones, la base de données est stockée dans le fichier `.arcus.db`.

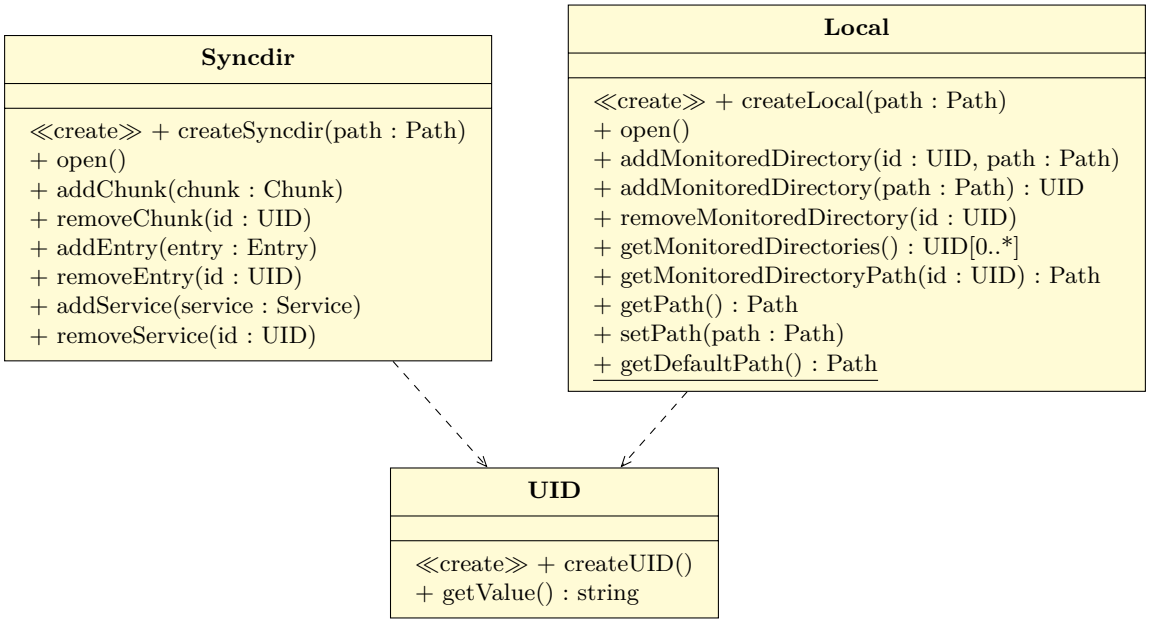


Figure 2.6 – **Modélisation UML du module Config.** La classe `Local` permet la lecture et l'écriture de la configuration locale et la gestion des associations entre les zones de synchronisation et leur répertoire. La classe `Syncdir` permet de gérer la base de données associée à une zone particulière. Enfin, la classe `UID` permet la génération d'identifiants uniques, utilisés pour identifier les fichiers, les zones de synchronisation, les blocs et les services.

2.4 Moteur de synchronisation des fichiers

Le moteur de synchronisation constitue la partie principale de l'application. Il s'appuie sur les fonctionnalités et les abstractions construites dans les modules précédents. Son rôle est de s'assurer

que les zones de synchronisation locales et sur le serveur restent synchronisées.

Pour ce faire, il observe les changements effectués par l'utilisateur dans les zones de synchronisation locales à sa machine. À chaque changement local, il téléverse les données vers les serveurs distants en les répartissant. Réciproquement, à chaque changement distant, il télécharge les données et reconstitue les fichiers en local.

Lors de cette procédure de synchronisation, nous distinguons trois cas qui peuvent se présenter.

Modification locale sans modification distante. Ce cas est le plus courant. Il se produit lorsque des changements locaux sont effectués sur une machine et qu'aucune modification n'est effectuée entre temps sur les serveurs.

Modification distante sans modification locale. Ce cas ne peut se produire que lorsqu'au moins deux machines sont synchronisées avec les mêmes serveurs. Par exemple, si un utilisateur effectue des changements sur une machine B et les envoie sur les serveurs, et qu'aucune modification n'est effectuée sur une machine A. Alors, dans le référentiel de la machine A, il y a eu des changements distants sans modification locale.

Modification locale et distante. Ce cas est le plus problématique, et ne peut se produire que lorsqu'au moins deux machines sont synchronisées avec les mêmes serveurs. Par exemple, si un utilisateur effectue des changements sur une machine A et une machine B, que les changements de la machine B sont envoyés en premier et que la machine A cherche à se synchroniser. Alors, dans ce cas, dans le référentiel de la machine A, il y a eu des changements distants et des changements locaux, il y a donc conflit.

Chacun de ces trois cas doit être traité par une procédure différente. Le moteur doit donc être en mesure de distinguer les trois cas afin d'enclencher la bonne procédure.

Pour discriminer ces cas, nous avons choisi d'utiliser un numéro de version, stocké dans la base de données de chaque zone. Ce numéro est fixé à 0 initialement. À chaque modification ce numéro est incrémenté, avant d'envoyer les changements vers le serveur. Un algorithme permettant de discriminer les trois cas en utilisant le numéro de version est présenté dans la figure 2.7.

Cependant, il persiste un cas ambigu : si le numéro de version distant est incrémenté entre le moment où nous le téléchargeons depuis le serveur et le moment où nous le comparons dans l'algorithme. Dans la version initiale d'Arcus, nous avons décidé de ne pas traiter ce cas pour simplifier le traitement, mais nous le traiterons toutefois dans le futur.

2.5 Journal des actions de l'application

Comme nous l'avons détaillé précédemment, l'application est responsable de la synchronisation de plusieurs répertoires, d'échanges de données avec des serveurs, de la gestion de configurations... Ces fonctionnalités sont souvent complexes, et nous avons anticipé le fait que lorsque des bogues se produiront à l'exécution, ils seront éventuellement difficiles à résoudre.

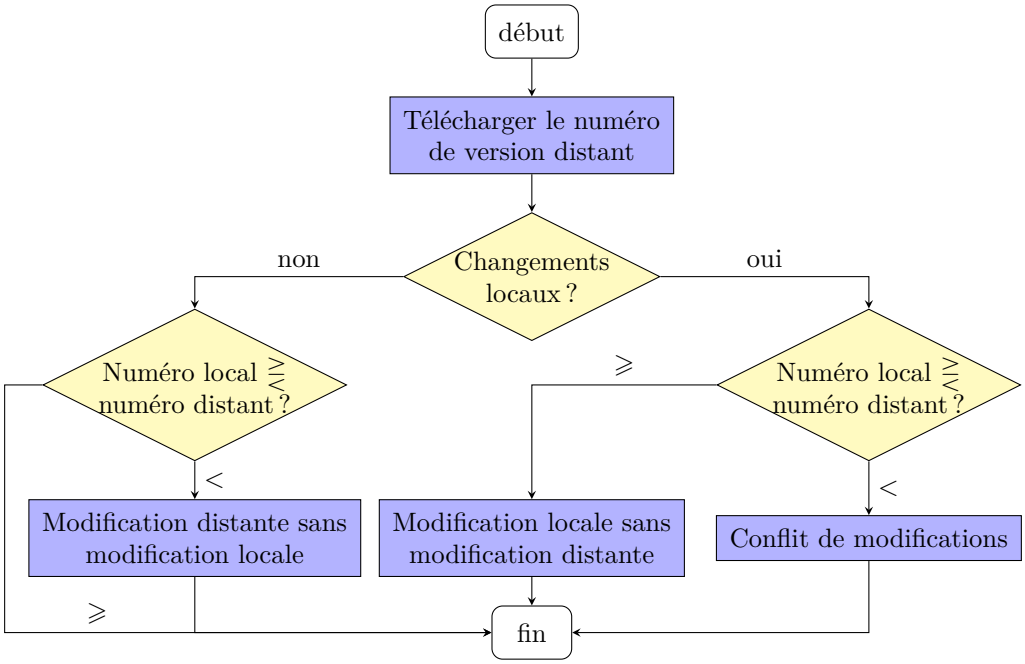


Figure 2.7 – **Algorithme de discrimination des différents cas de synchronisation.** Dans le cas où il y a des changements locaux, alors, si le numéro local est strictement inférieur au numéro distant, on se trouve dans le cas où il y a des modifications locales sans modification distante. Sinon, il y a un conflit de modifications. À l'inverse, s'il n'y a pas de modificaiton locale, alors il y a des modifications distantes seulement si le numéro local est strictement inférieur au numéro distant. Sinon, il n'y a rien à faire.

Pour pallier ce problème, nous avons décidé de créer un journal des actions de l'application. Celui-ci permet de retranscrire précisément la séquence d'actions effectuée par l'application, d'identifier plus facilement les problèmes éventuels et de remonter à la source de ceux-ci.

Nous avons pour cela décidé d'associer à chaque message du journal la date et l'heure à laquelle il a été produit. Cette date est formattée en ISO 8601 (ISO, 2004) pour éviter toute ambiguïté et faciliter son interprétation par d'autres applications. Nous avons attribué à chaque message un niveau de sévérité (informatif, avertissement ou erreur) et un code couleur qui permettent de relever facilement les informations importantes. Enfin, la ligne de code ayant produit un message lui est clairement associée afin de pouvoir diagnostiquer plus facilement les éventuels problèmes. La figure 2.8 résume le format que nous avons choisi pour les messages du journal.

```
YYYY-MM-DDTHH:MM:SS [info|warn|err!] fichier:ligne - Message
```

Date et heure *Sévérité* *Source*

Figure 2.8 – **Format unifié d'écriture des messages pour le journal de l'application.**

Chapitre 3

Implémentation d'Arcus

Nous avons choisi de développer le projet Arcus en C++. En effet, le C++ est un langage orienté objet, ce qui correspond à la modélisation objet que nous avons réalisé pour notre application, et multi-plateformes, ce qui nous permettra de porter l'application sur plusieurs systèmes relativement facilement.

Par ailleurs, le C++ est l'un des principaux langages enseignés dans notre formation. Nous possédons donc des acquis avec ce langage, et ce projet nous permet d'en améliorer notre connaissances. Enfin, ce langage jouit d'une communauté importante, de nombreuses bibliothèques ainsi que d'une documentation très complète.

Nous avons choisi d'utiliser C++11, la version de C++ spécifiée par le standard publié en 2012 (ISO, 2012). Celle-ci ajoute de nombreuses fonctionnalités au langage et à la bibliothèque standard, entre autres le multi-tâches, les lambdas, les tables de hachage et les expressions régulières.

Nous compilons notre code avec le compilateur Clang, et, pour automatiser la compilation, nous avons utilisé CMake. CMake dispose d'un méta-langage qui permet de produire à la fois des fichiers pour Make, XCode, Visual Studio et d'autres environnements. Il facilite donc le portage.

Ces choix sont résumés dans la table 3.1.

3.1 Mise en place de la communication avec les serveurs

Pour réaliser le module de communication, nous avons décidé d'utiliser la bibliothèque cURL. [1] L'utilisation de cette bibliothèque nous permet de bénéficier de sa robustesse, sa simplicité et sa fiabilité. Il s'agit par ailleurs d'une des bibliothèques les plus utilisées pour la communication sur HTTP, notamment utilisée par Adobe, Apple, Microsoft, Mozilla et Spotify (cURL, 2017). La plupart des problèmes que nous pouvons rencontrer avec cette bibliothèque sont donc déjà documentés.

1. cURL : `https://curl.haxx.se/`

Nous avons opté pour une implémentation permettant de traiter les requêtes en provenance du module Services de manière asynchrone. Le but est que, pendant l'envoi d'une requête et l'attente de sa réponse, le module puisse continuer d'accepter d'autres requêtes de la part de l'application. Ainsi l'application peut continuer à s'exécuter en attendant le téléversement ou le téléchargement de données vers ou depuis un serveur.

Pour ce faire, nous avons réalisé une implémentation multi-tâches avec deux tâches parallèles : une tâche principale qui reçoit les requêtes et une tâche qui s'occupe de les envoyer et de réceptionner les réponses. Cette implémentation exploite les fonctionnalités de multi-tâches introduites en C++11, en complément de la bibliothèque Boost Thread. [2]

Lorsque la tâche principale reçoit une requête en provenance du module Service, elle l'ajoute dans une file d'attente des requêtes à envoyer et continue son exécution. En parallèle, la tâche d'envoi et de réception boucle en permanence en attendant que des requêtes arrivent dans la file d'attente. Elle les envoie alors une par une aux serveurs distants et réceptionne les réponses. Lorsqu'une réponse est reçue par la tâche d'envoi et de réception, celle-ci est transmise à la tâche principale via une promesse. La figure 3.1, donne un exemple de comportement de ces deux tâches.

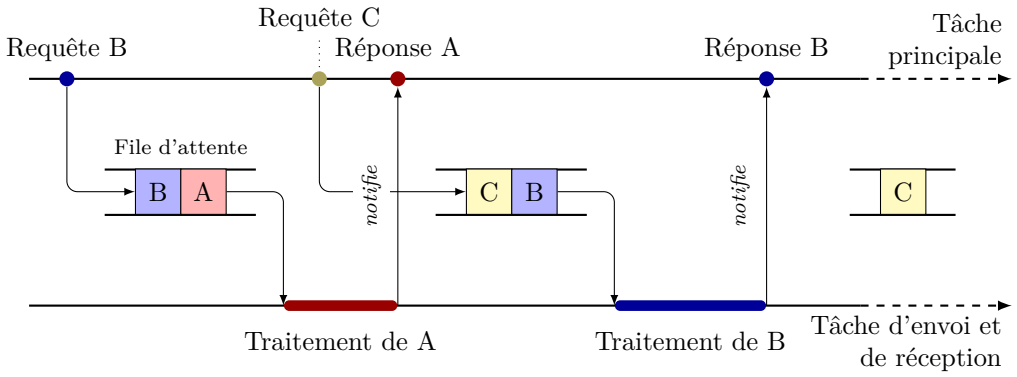


Figure 3.1 – **Exemple de comportement des deux tâches lors d'envois de requêtes.** La tâche principale reçoit une requête supplémentaire C pendant que la tâche d'envoi et de réception est en train de traiter la requête A.

À chaque envoi de requête, la bibliothèque cURL ouvre une nouvelle connexion au serveur demandé. Ce comportement peut se révéler inefficace notamment si plusieurs requêtes vers le même serveur sont envoyées à la suite. Pour limiter ce problème, nous avons créé une classe de piscine *(pool)* permettant de mutualiser les ressources et notamment de garder la même connexion ouverte pour plusieurs requêtes vers le même serveur dans un intervalle de temps court.

L'annexe B.1 présente les principaux extraits du code gérant la synchronisation entre les deux tâches et l'utilisation de la bibliothèque cURL. La figure 3.2 montre un exemple d'utilisation du module comme il pourrait l'être dans le module Services.

2. Boost Thread : `http://www.boost.org/doc/libs/1_64_0/doc/html/thread.html`

```
HTTP::Request req = HTTP::Request()
    .setUrl("https://fr.wikipedia.org/wiki/Wikip%C3%A9dia:Accueil_principal")
    .setVerb(HTTP::Verb::GET);

HTTP::Sender sender;
sender.send(req, &std::cout).get();
```

Figure 3.2 – **Exemple d'utilisation du module HTTP pour envoyer une requête.** Dans cet exemple, le contenu de la page d'accueil de Wikipédia est téléchargé et affiché la sortie standard. Grâce au travail réalisé dans ce module, nous sommes en mesure d'écrire un code concis de haut niveau pour envoyer des requêtes HTTP et récupérer leur réponse.

3.2 Création de l'interface commune aux services

L'objectif du module Services est de fournir une abstraction de la communication avec les interfaces des services de stockage. Pour fournir cette abstraction, nous avons implémenté une classe pour chaque service de stockage. Chacune de ces classes réalise l'interface présentée dans la partie conception.

Dans ces classes, chaque action abstraite requise par l'interface, comme l'ajout d'un fichier ou la récupération du quota, est « traduite » en une requête ou série de requêtes permettant de réaliser cette action sur le service. Une fois la réponse du serveur du service reçue, celle-ci est transformée dans les structures de données utilisées par notre application.

L'implémentation de ce module ne présente pas de complexité technique particulière, si ce n'est celle de manier correctement le concept de promesses. Cette simplicité est due au fait que les détails d'implémentation de la communication sont cachés dans le module HTTP. L'annexe B.2 montre quelques exemples de traduction des actions pour l'A.P.I. de Dropbox.

Vu de l'extérieur de ce module, la communication avec l'un ou l'autre des services supportés est totalement transparente. Cela nous permet, dans les autres modules, de manipuler un service de façon abstraite sans craindre de devoir modifier le code en cas d'ajout d'un nouveau service.

3.3 Formats de stockage des configurations

Nous avons choisi, pour stocker la configuration synchronisée, une base de données SQLite 3. [3] Nous avons donc converti notre modèle entité-association, présenté dans la partie conception, en modèle relationnel, décrit dans la figure 3.3. Nous avons choisi la bibliothèque SQLite 3 car elle dispose d'une A.P.I. simple d'utilisation, et que les bases de données produites sont contenues dans un unique fichier, ce qui simplifie grandement leur synchronisation avec les différents services.

3. SQLite 3 : `https://www.sqlite.org/`

```
Entry(id, name, type, parent)
Chunk(id, hash, order, File.id)
ChunkService(Chunk.id, Service.id)
Service(id, provider, auth)
```

Figure 3.3 – **Modèle relationnel de la base de données.**

En raison de la simplicité de la structure de la configuration locale de notre application, nous avons choisi un format plus léger qu'une base de données : le *JavaScript Object Notation* (JSON, ECMA, 2013). Ce format a l'avantage d'être simple et compréhensible par un humain. Pour lire et écrire les fichiers JSON, nous utilisons la bibliothèque `nlohmann::json`[4]. La figure 3.4 présente le format de ce fichier.

```
{
    "monitored_directories": {
        "ID-ZONE-1": "/chemin/zone-1/",
        "ID-ZONE-2": "/chemin/zone-2/",
        "ID-ZONE-3": "/chemin/zone-3/"
    }
}
```

Figure 3.4 – **Format du fichier de configuration local.** Sous la clef `monitored_directories` se trouve un dictionnaire associant les identifiants des zones de synchronisation aux chemins des répertoires sur lesquels elles sont attachées.

Pour les identifiants dans les fichiers de configuration, nous utilisons des *Universally Unique IDentifier* (UUID). Pour les générer, nous utilisons la bibliothèque Boost UUID [5] qui est compatible avec la majorité des systèmes d'exploitation.

3.4 Détection des changements dans le système de fichiers

Afin de détecter l'ajout, la suppression ou la mise à jour des fichiers locaux, nous utilisons fswatch [6]. Cette bibliothèque a le grand avantage d'abstraire derrière une A.P.I. C++ de haut niveau les mécanismes très divers des différents systèmes d'exploitation permettant l'observation des changements effectués dans le système de fichiers. Son utilisation permettra d'adapter Arcus sur plusieurs systèmes plus facilement.

4. `nlohmann::json` : `https://github.com/nlohmann/json`
5. Boost UUID : `http://www.boost.org/doc/libs/1_64_0/libs/uuid/uuid.html`
6. `fswatch` : `http://emcrisostomo.github.io/fswatch/`

	Fonction	**Outil**
Code	Langage	C++11
	Compilateur	Clang
	Construction automatisée	CMake
Bibliothèques	Requêtes HTTP	cURL
	Base de données	SQLite 3
	Interprétation JSON	`nlohmann::json`
	Observation arborescence	fswatch
	Chemins et système de fichiers	Boost Filesystem
	Identifiants uniques	Boost UUID
	Threads	Bibliothèque standard et Boost Thread

Table 3.1 – **Résumé des technologies choisies pour l'implémentation.**

Chapitre 4
Bilan et difficultés rencontrées

4.1 Bilan de l'avancement du projet

Nous avons terminé l'implémentation du module de communication via HTTP. Pour tester ce module, nous avons développé un programme de test effectuant des échanges avec le serveur `www.example.com` sur différents types de requêtes. Nous nous sommes ainsi assurés du bon fonctionnement de ce module, y compris lorsque plusieurs requêtes sont envoyées en parallèle. Désormais, le module est utilisable pour l'envoi de requêtes avec n'importe quel verbe sur n'importe quelle URL.

Nous avons également fini la création du module des services, et créé une classe permettant d'utiliser le service de stockage Dropbox. Nous avons écrit un programme de test qui récupère le quota, liste un répertoire, envoie des fichiers et en supprime sur un compte Dropbox. Ce test permet également de valider le bon fonctionnement du module HTTP qui est utilisé par la classe.

À l'heure où nous écrivons ces lignes, Arcus est capable de communiquer avec Dropbox. Nous travaillons actuellement sur l'intégration de OneDrive, le service de Microsoft, que nous comptons terminer pour le rendu du livrable. Ce travail permettra de montrer la facilité avec laquelle un nouveau service peut être intégré dans l'interface multi-services, et ce sans aucune modification dans les autres modules.

Nous avons achevé la réalisation du module de journalisation des actions de l'application, et celui-ci est utilisé à travers les autres modules. Nous avons également développé un programme pour tester la création du journal et la cohérence des messages affichés.

Nous avons aussi terminé la mise en place du module permettant de manipuler les configurations locales et distantes, y compris l'intégration avec la bibliothèque de lecture et d'écriture du JSON et SQLite 3. Nous avons vérifié que le comportement de ce module soit correct avec un programme de tests écrivant et lisant des configurations locales et créant des bases de données.

Enfin, au moment de la rédaction de ce rapport, nous sommes en train de finaliser le moteur de synchronisation afin de faire fonctionner notre application. Nous prévoyons de finaliser ce dernier module dans les semaines à venir.

Sur les modules que nous avons mis en place et testés, nous nous sommes efforcés de régler tous les bogues que nous avons pu rencontrer, et nous n'avons pas pour le moment connaissance des éventuels bogues restants.

4.2 Difficultés rencontrées

La conception de notre application a été une première difficulté. En effet, nous n'avions encore jamais travaillé sur la conception d'une application de cette taille. Nous avons donc passé plusieurs semaines à définir le cahier des charges et les modèles afin d'éviter de nous bloquer par la suite en poursuivant avec un modèle erroné. Nous les avons ensuite raffinés pendant la phase d'implémentation.

Au cours de la création du module HTTP de notre application, nous avons été confrontés pour la première fois à la programmation concurrente. Nous avons donc dû apprendre les concepts liés à ce type de programmation, tels que les tâches, les méthodes de synchronisation de ces tâches, les promesses et valeurs futures. Nous avons rapidement rencontré les problèmes couramment associés aux tâches et à leur synchronisation, tels que les *deadlocks* ou les *race conditions*. Ce nouvel apprentissage a constitué pour nous une difficulté importante, et nous sommes parvenus à la surmonter grâce à l'aide de notre encadrante.

Conclusion et perspectives

Connaissances et apprentissages

Ce projet nous a permis de mettre en pratique et de consolider de nombreuses connaissances aquises tout au long de notre parcours universitaire. Nous donnons dans cette partie des exemples d'enseignements qui nous ont été utiles dans la réalisation de ce projet.

L'enseignement en systèmes d'information et bases de données (HLIN304) nous a appris a concevoir une base de données, à la gérer et à l'interroger avec le langage SQL. Cela nous a permis de concevoir la base de données de notre application.

Nous avons appris à programmer en C++ à l'aide des enseignements de programmation impérative (HLIN202, HLIN302). Nous avons pu utiliser la programmation orientée objet et modéliser nos modules en UML grâce aux cours de modélisation et de programmation par objet (HLIN406).

Les cours de techniques de communication, de conduite de projets (HLIN408) et de projet C.M.I. (HLSE205) nous ont fait découvrir la gestion de projet et permis d'apprendre à utiliser de nombreux outils utiles tels que LaTeX, Git, GitLab, Make et gdb...

Enfin, nous avons découvert et appris à nous servir du format JSON avec l'enseignement « du binaire au web » (HLIN102).

Ce projet nous a également permis de développer de nouvelles compétences. En particulier, tout ce qui concerne le réseau a été une découverte pour nous. Nous avons appris les principes de HTTP, exploré l'A.P.I. Rest fournie par Dropbox et OneDrive et étudié le fonctionnement de la bibliothèque cURL. Nous avons également appris les bases de la programmation concurrente.

Enfin, ce projet a été l'occasion de découvrir le fonctionnement et l'interface de SQLite 3. Il nous a permis de voir une première utilisation concrète des bases de données.

Perspectives

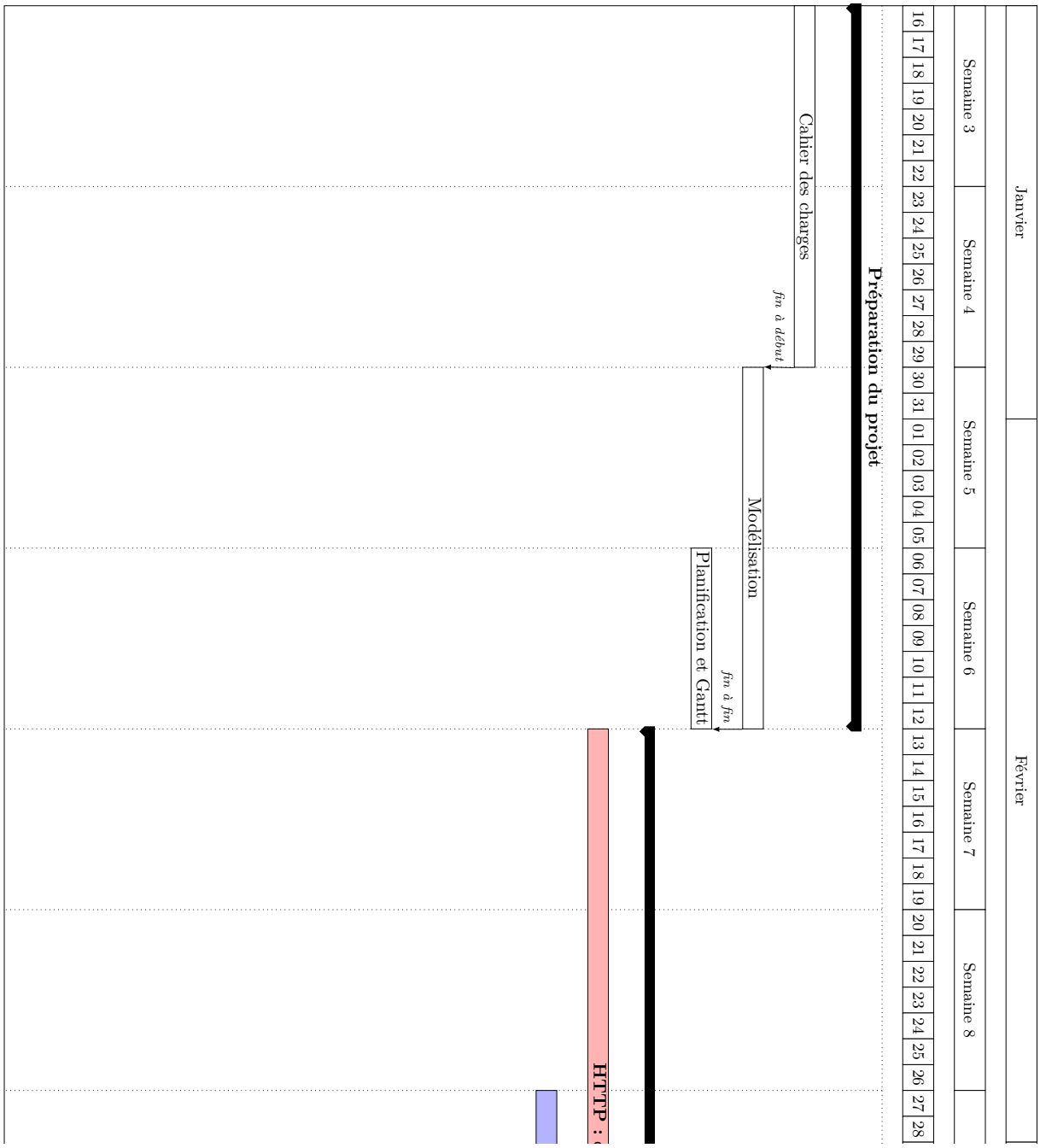
La seconde partie de notre projet, que nous réaliserons en stage au laboratoire d'informatique, de robotique et de microélectronique de Montpellier (Lirmm) entre le 30 mai et le 30 juin, sera l'occasion d'implémenter les fonctionnalités encore manquantes à notre application.

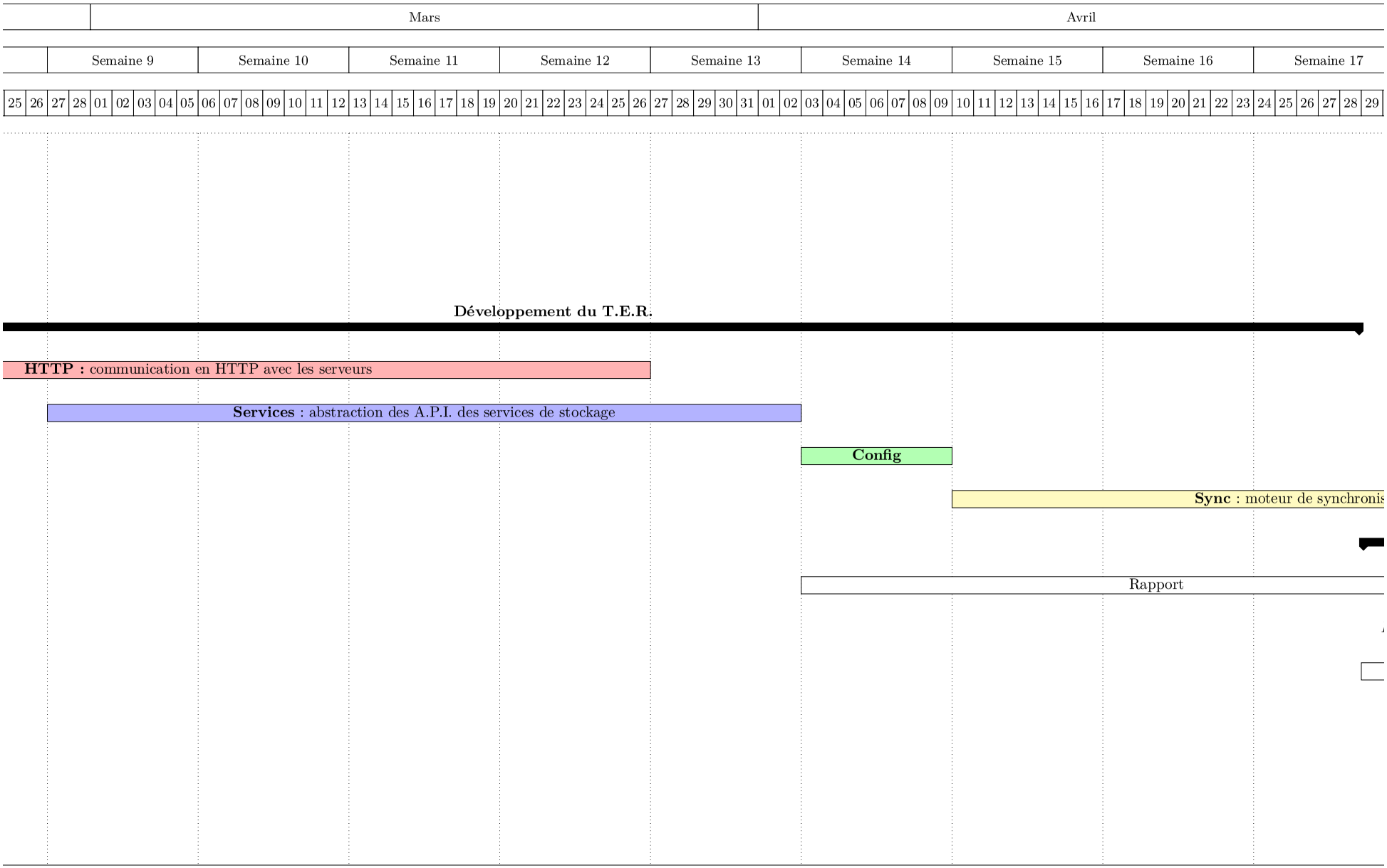
Nous nous occuperons de la sécurité de l'application, notamment du chiffrement des informations d'authentification et des données de l'utilisateur. Nous doterons également notre application d'une interface graphique simple permettant de faciliter sa configuration. Nous prévoyons également de porter notre application sur plusieurs systèmes. Enfin, nous prévoyons d'intégrer notre application avec plus de services.

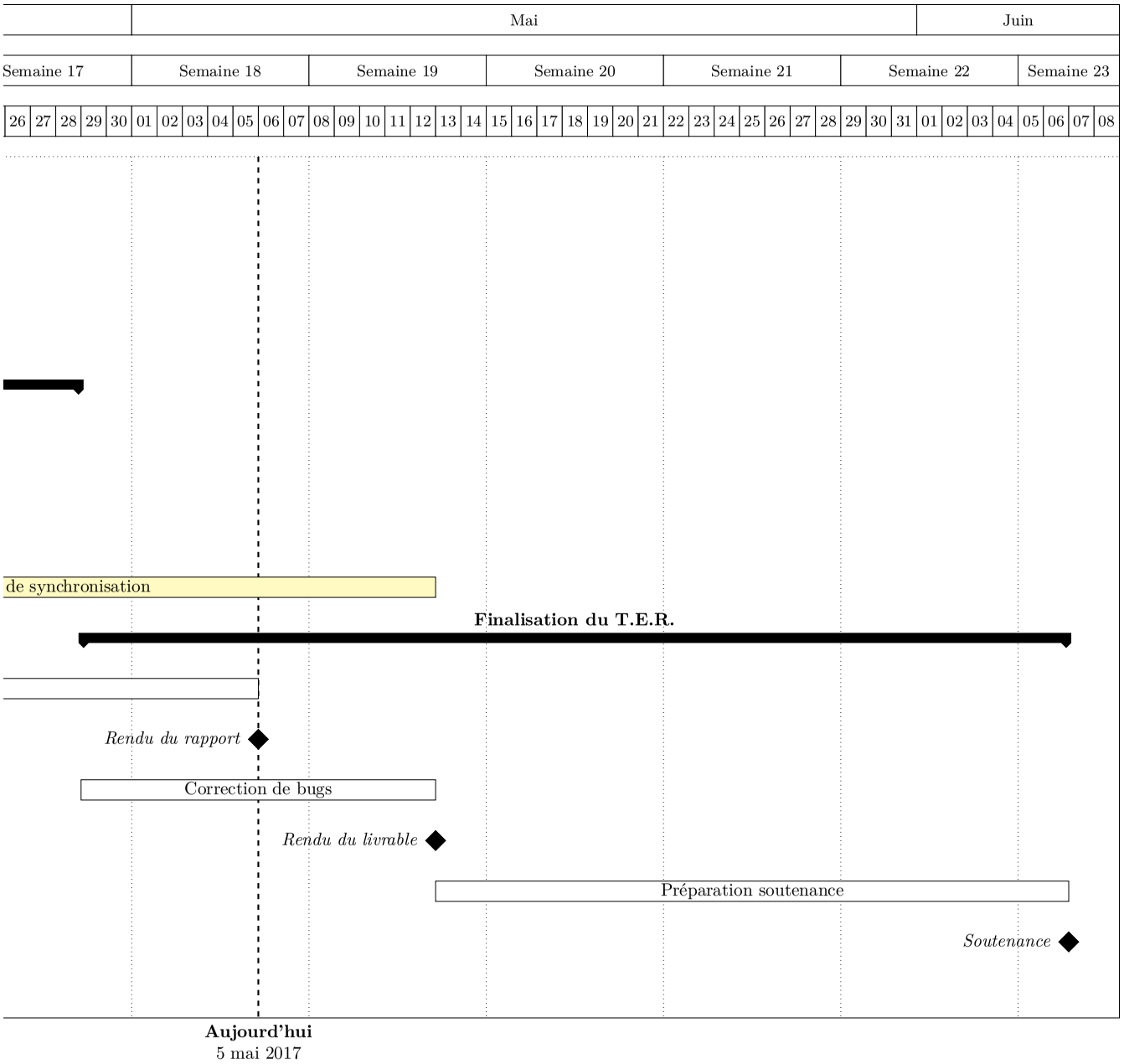
Annexes

Annexe A

Diagramme de Gantt




Mars
Avril
Semaine 9
Semaine 10
Semaine 11
Semaine 12
Semaine 13
Semaine 14
Semaine 15
Semaine 16
Semaine 17
25 26 27 28 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
Développement du T.E.R.
HTTP : communication en HTTP avec les serveurs
Services : abstraction des A.P.I. des services de stockage
Config
Sync : moteur de synchronis
Rapport


Mai
Juin
Semaine 17
Semaine 18
Semaine 19
Semaine 20
Semaine 21
Semaine 22
Semaine 23
26 27 28 29 30 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08
de synchronisation
Finalisation du T.E.R.
Rendu du rapport
Correction de bugs
Rendu du livrable
Préparation soutenance
Soutenance
Aujourd'hui
5 mai 2017

Annexe B

Extraits de code

B.1 Gestion de la tâche d'envoi et de réception des requêtes

Cet extrait du code montre comment est gérée la synchronisation entre la tâche principale et la tâche d'envoi et de réception afin de permettre au module Services d'effectuer des requêtes asynchrones. Le code source correspondant se trouve dans le fichier `src/HTTP/Sender.cpp`.

```
1  // (...)
2  Sender::Sender()
3  : _send_running(false), _multi(curl_multi_init())
4  {}
5
6  Sender::~Sender()
7  {
8      // Si le thread d'envoi a été démarré, on veut désormais le stopper
9      if (_send_worker.joinable())
10     {
11         {
12             // Réveil du thread et demande d'arrêt
13             std::lock_guard<std::mutex> wakeup_lock(_wakeup_mutex);
14             _send_running = false;
15             _send_wakeup.notify_one();
16         }
17
18         // Jonction des deux fils d'exécution
19         _send_worker.join();
20     }
21
22     // Libération de la file d'attente cURL
23     curl_multi_cleanup(_multi);
24 }
25
26 boost::future<Response> Sender::send(Request& req, std::ostream* out)
27 {
28     _start();
29
30     // Mise en attente d'une instance cURL pour la nouvelle requête
31     std::lock_guard<std::mutex> sendings_lock(_sendings_mutex);
32     CURL* simple = _prepare(req, out);
```

```
33
34      // Réveil du thread d'envoi pour la prise en charge de la requête
35      std::lock_guard<std::mutex> wakeup_lock(_wakeup_mutex);
36      _send_wakeup.notify_one();
37
38      return _sendings_infos[simple].promise.get_future();
39  }
40
41  // (...)
42  void Sender::_start()
43  {
44      if (!_send_running)
45      {
46          _send_running = true;
47
48          // Si le thread d'envoi a rencontré une erreur, on l'arrête
49          if (_send_worker.joinable())
50          {
51              _send_worker.join();
52          }
53
54          // Lancement du thread d'envoi
55          _send_worker = std::thread(&Sender::_sendWorker, this);
56      }
57  }
58
59  CURL* Sender::_prepare(Request& base_req, std::ostream* out)
60  {
61      // Récupération d'une instance de cURL depuis la piscine
62      CURL* simple = _pool.pop();
63
64      // Création d'un objet d'informations sur l'envoi. Les informations sur
65      // la requête sont copiées afin de rester accessibles tout au long de
66      // l'envoi.
67      _sendings_infos[simple] = {
68          base_req,
69          Response(),
70          boost::promise<Response>()
71      };
72
73      // (...)
74      // Ajout de la nouvelle instance dans la file d'attente
75      _pending_sendings.push(simple);
76
77      return simple;
78  }
79
80  void Sender::_fulfill(CURL* simple)
81  {
82      // Recherche des informations sur l'envoi à associé à l'instance donnée
83      auto infos_search = _sendings_infos.find(simple);
84
85      // Recherche de l'instance dans la liste des envois en cours
86      auto current_search = std::find(
87          _current_sendings.begin(),
88          _current_sendings.end(),
89          simple
90      );
```

```
91
92      if (
93          infos_search != _sendings_infos.end() &&
94          current_search != _current_sendings.end()
95      )
96      {
97          _SendingInfo& info = infos_search->second;
98
99          // Remplissage du code de réponse HTTP
100         long response_code = 0;
101         curl_easy_getinfo(
102             simple,
103             CURLINFO_RESPONSE_CODE,
104             &response_code
105         );
106
107         info.res.setCode(response_code);
108
109         // Satisfaction de la promesse associée à la requête
110         info.promise.set_value(std::move(info.res));
111
112         // Suppression des informations pour cet envoi
113         _sendings_infos.erase(infos_search);
114
115         // Suppression de la liste des envois en cours
116         _current_sendings.erase(current_search);
117
118         curl_multi_remove_handle(_multi, simple);
119     }
120
121     // Dans tous les cas, libération de la ressource
122     _pool.push(simple);
123 }
124
125 void Sender::_sendWorker()
126 {
127     CURLMcode code;
128     int unused = 0;
129     int file_descriptors = 0;
130
131     while (_send_running)
132     {
133         // Demande de réalisation des envois associés aux requêtes cURL
134         // en attente (non-bloquant)
135         code = curl_multi_perform(_multi, &unused);
136
137         if (code != CURLM_OK)
138         {
139             _send_running = false;
140             break;
141         }
142
143         // Attente de la réalisation des envois ou au plus une seconde
144         code = curl_multi_wait(_multi, NULL, 0, 1000, &file_descriptors);
145
146         if (code != CURLM_OK)
147         {
148             _send_running = false;
```

```
149             break;
150         }
151
152         if (file_descriptors == 0)
153         {
154             // cURL nous signale qu'on ne peut pas savoir quand se terminera
155             // le travail en cours. On attend donc 200 ms.
156             std::this_thread::sleep_for(std::chrono::milliseconds(200));
157         }
158
159         std::unique_lock<std::mutex> sendings_lock(_sendings_mutex);
160
161         // Si certains envois sont terminés, satisfaction des promesses
162         CURLMsg* message = curl_multi_info_read(_multi, &unused);
163
164         while (message != nullptr)
165         {
166             if (message->msg == CURLMSG_DONE)
167             {
168                 _fulfill(message->easy_handle);
169             }
170
171             message = curl_multi_info_read(_multi, &unused);
172         }
173
174         // S'il y a moins d'envois en cours que le nombre maximal autorisé
175         // d'envois parallèles, et s'il y a des envois en attente, on peut
176         // retirer des éléments de la file d'attente et les lancer
177         while (
178             !_pending_sendings.empty() &&
179             _current_sendings.size() < _concurrency
180         )
181         {
182             CURL* simple = _pending_sendings.front();
183             Request& req = _sendings_infos[simple].req;
184
185             _pending_sendings.pop();
186
187             _current_sendings.push_back(simple);
188             curl_multi_add_handle(_multi, simple);
189         }
190
191         // Si plus aucun envoi n'est en attente, on met en sommeil
192         // le thread
193         if (_current_sendings.size() == 0 && _send_running)
194         {
195             std::unique_lock<std::mutex> wakeup_lock(_wakeup_mutex);
196             sendings_lock.unlock();
197             _send_wakeup.wait(wakeup_lock);
198         }
199     }
200 }
```

B.2 Traduction des actions en requêtes sur les services

Cet extrait du code montre l'implémentation des méthodes permettant d'envoyer un fichier vers Dropbox, de récupérer un fichier ou de récupérer le quota. Le code source correspondant se trouve dans le fichier `src/Services/DropboxService.cpp`.

```
boost::future<void> DropboxService::putFile(std::istream& source, std::string target)
{
    std::shared_ptr<std::stringstream> out(new std::stringstream);
    json payload = {
        {"path", target},
        {"mode", "overwrite"},
        {"mute", true},
    };

    return _sender.send(
        HTTP::Request()
            .setVerb(HTTP::Verb::POST)
            .setUrl(CONTENT_BASE_URL + "files/upload")
            .addHeader("Authorization", "Bearer " + _token)
            .addHeader("Dropbox-API-Arg", payload.dump())
            .setFile(&source),
        out.get()
    ).then([out, &source, target, this](boost::future<HTTP::Response> future)
    {
        HTTP::Response res = future.get();

        if (res.getCode() != 200)
        {
            throw std::runtime_error("Impossible d'envoyer le fichier.");
        }
    });
}

boost::future<void> DropboxService::fetchFile(std::string source, std::ostream& target)
{
    json payload = {{"path", source}};

    return _sender.send(
        HTTP::Request()
            .setVerb(HTTP::Verb::POST)
            .setUrl(CONTENT_BASE_URL + "files/download")
            .addHeader("Authorization", "Bearer " + _token)
            .addHeader("Dropbox-API-Arg", payload.dump()),
        &target
    ).then([](boost::future<HTTP::Response> future)
    {
        HTTP::Response res = future.get();

        if (res.getCode() != 200)
        {
            throw std::runtime_error("Impossible de recevoir le fichier.");
        }
    });
}

```

```
51  boost::future<Quota> DropboxService::getQuota()
52  {
53      std::shared_ptr<std::stringstream> out(new std::stringstream);
54
55      return _sender.send(
56          HTTP::Request()
57              .setVerb(HTTP::Verb::POST)
58              .setUrl(API_BASE_URL + "users/get_space_usage")
59              .addHeader("Authorization", "Bearer " + _token),
60          out.get()
61      ).then([out](boost::future<HTTP::Response> future)
62      {
63          HTTP::Response res = future.get();
64
65          if (res.getCode() != 200)
66          {
67              throw std::runtime_error(
68                  "Impossible de récupérer le quota."
69              );
70          }
71
72          json data;
73          data << *out;
74
75          return Quota(data["used"], data["allocation"]["allocated"]);
76      });
77  }
```

Bibliographie

cURL. *Companies Using curl in Commercial Environments*. Disponible sur : `https://curl.haxx.se/docs/companies.html` (consulté le 29/04/2017).

Dropbox. *Dropbox for HTTP Developers*. 2017. Disponible sur : `https://www.dropbox.com/developers/documentation/http/documentation` (consulté le 03/05/2017).

ECMA. *ECMA-404 : The JSON Data Interchange Format*. Première édition. Oct. 2013. Disponible sur : `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf` (consulté le 01/05/2017).

Fielding, Roy T. "Architectural styles and the design of network-based software architectures". Thèse de doct. University of California, Irvine, 2000. Disponible sur : `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf` (consulté le 30/04/2017).

Fielding, Roy T., James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach et Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. IETF, juin 1999. Disponible sur : `http://www.rfc-editor.org/rfc/rfc2616.txt` (consulté le 29/04/2017).

ISO. *ISO/IEC 14882 :2011 Information technology – Programming languages – C++*. International Organization for Standardization, fév. 2012. Disponible sur : `https://www.iso.org/standard/50372.html` (consulté le 30/04/2017).

— *ISO/IEC 8601 :2004 Data elements and interchange formats – Information interchange – Representation of dates and times*. International Organisation for Standardization, déc. 2004. Disponible sur : `https://www.iso.org/standard/40874.html` (consulté le 05/05/2017).

McDowell, C., L. Werner, H. E. Bullock et J. Fernald. "The impact of pair programming on student performance, perception and persistence". In : *25th International Conference on Software Engineering, Proceedings*. International Conference on Software Engineering. IEEE ; IEEE Comp Soc, Tech Council Software Engn ; ACM ; ACM SIGSOFT ; IBM ; NORTHROP GRUMMAN Space Technol ; BMW ; NOKIA ; SUN Microsyst ; DaimlerChrysler ; Microsoft Res. 2003, p. 602–607.

Microsoft. *OneDrive Files API*. 2015. Disponible sur : `https://dev.onedrive.com/README.htm` (consulté le 03/05/2017).

Rosenberg, Robin. *Git index format*. 2010. Disponible sur : `https://github.com/git/git/blob/master/Documentation/technical/index-format.txt` (consulté le 30/04/2017).