cDrive : The Cloud Drive

Laboratoire Réseau 2019

L. SWINNEN – Février 2019

Introduction

Le sujet du laboratoire est l'élaboration d'un *système de stockage décentralisé, fiable et confidentiel*.

L'objectif du laboratoire réseau est de vous faire *découvrir la mise en place d'éléments de sécurisation* (chiffrement symétrique et asymétrique, hachage, …), de vous *permettre d'implémenter un protocole* (avec les différentes étapes comme l'analyse syntaxique des messages, la compréhension et la réponse à ceux-ci), de vous *permettre d'aborder la programmation concurrente* (notamment en gérant des connexions simultanées) et *de vous faire programmer différents types de communication* (unicast avec TCP et multicast avec UDP).

Le travail sera réalisé par **groupe de 3 étudiants.** Exceptionnellement, et avec l'accord de votre responsable de laboratoire, des groupes de 4 étudiants (avec des objectifs plus grands) seront possibles si le nombre d'étudiants n'est pas un multiple de 3.

Le laboratoire comporte **28 h dédiés à la réalisation de cet exercice.** Vous avez normalement le temps, durant ces séances, de développer et terminer l'application demandée. L'ensemble de l'application comporte, environ 2000 lignes (tout compris, sans extension : déclaration, interfaces, code, …).

Pour ce laboratoire, **vous devez utiliser le GitLab HELMo**. Ainsi, on vous demande de déterminer rapidement quel étudiant hébergera le dépôt pour ce travail.

Un planning est proposé à la suite de la présentation de l'énoncé. Veillez à le respecter scrupuleusement.

1. L'architecture

Dans notre système, nous distinguerons (voir figure 1) :

- Les **StorageProvider** qui sont des petits serveurs mettant à disposition de l'espace de stockage. Ils sont responsables de la persistance des données qui leur sont envoyées. Les *storage providers* s'annoncent en multicast au *request handler*. Le système fonctionnera uniquement s'il y a, au moins, 1 instance de *storage provider* disponible.

- Le **RequestHandler** qui gère les requêtes venant des clients. Ainsi, il permet l'*enregistrement d'un utilisateur* , la *connexion d'un utilisateur enregistré*, le *stockage d'un fichier*, l'*obtention d'un fichier enregistré*, *la suppression d'un fichier stocké*. Afin de garantir une certaine confidentialité, les fichiers sont chiffrés avant d'être transmis au(x) *Storage Provider(s)*.

- Le **Client** est le programme applicatif permettant d'interagir avec le système. Il permet de se connecter (ou de s'enregistrer), de charger un nouveau fichier ou récupérer un fichier déjà sauvegardé. Enfin, l'outil permettra aussi de supprimer les fichiers stockés. Tous les échanges avec le *RequestHandler* seront protégés par SSL/TLS.

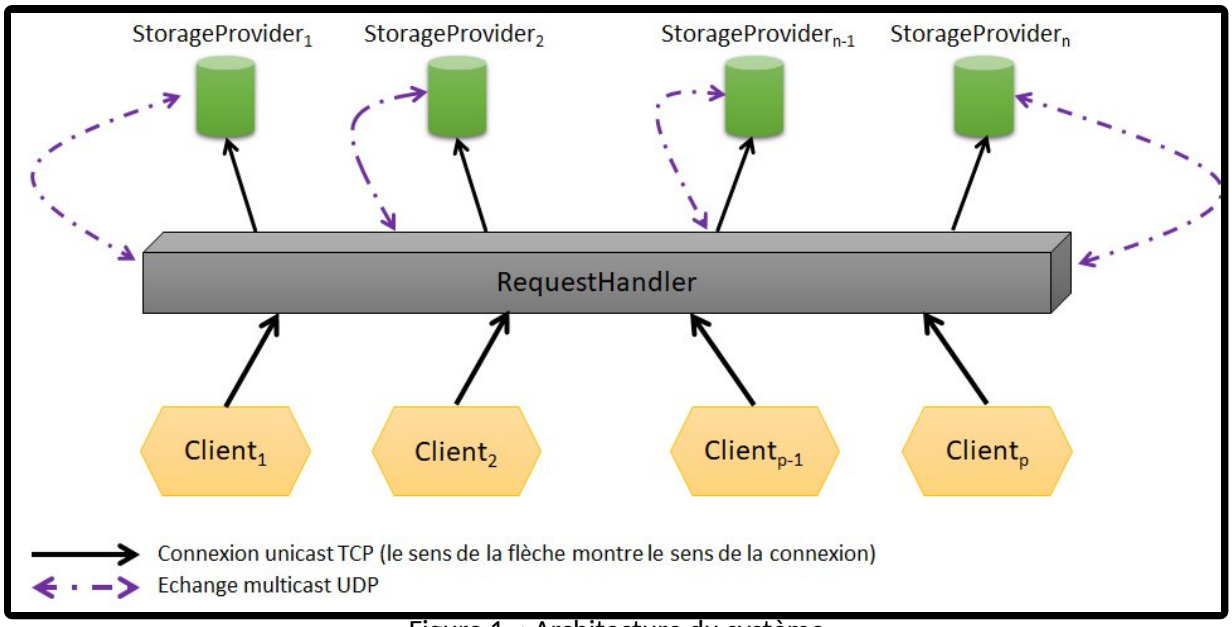

Figure 1 : Architecture du système

Comme montré sur la *figure 1*, l'architecture proposée est variée puisqu'on y trouve des protocoles de transports variés (TCP et UDP), des modes de communication diversifiés (unicast et multicast) et des protocoles applicatifs différents.

Les éléments qui devront être implémentés sont : le *request handler* et le *storage provider*. Un *client* par défaut vous est fourni.

1.1 Fonctionnement

1.1.1 Le Client

Le client est un petit programme qui permet d'interagir avec le *RequestHandler*. Ainsi, il permet d'enregistrer un nouvel utilisateur, de se connecter, d'échanger des fichiers. La connexion avec le *Request Handler* s'effectue en TCP, en mode unicast. En outre, cette connexion est protégée par TLS et tous les échanges sont chiffrés.

L'utilisateur, une fois enregistré, peut donc envoyer ou récupérer un fichier préalablement stocké. Il propose une interface GUI permettant de réaliser ces opérations.

1.1.2 Le RequestHandler

Le *RequestHandler* est l'élément central de notre système. Ainsi, le *RequestHandler* traite les requêtes des différents clients connectés.

Au démarrage, le *RequestHandler* va écouter les annonces des différents *StorageProviders*. Ceux-ci s'annoncent en multicast. Pour chaque *StorageProvider* annoncé, le *RequestHandler* établit une connexion TCP unicast vers celui -ci. Pour le bon fonctionnement du système, il est nécessaire qu'au moins 1 *StorageProvider* soit actif.

Lorsqu'un utilisateur s'enregistre, le *RequestHandler* génère automatiquement une clé secrète AES (d'une taille de 128 bits au moins) pour chiffrer tous les fichiers soumis par celui-ci. La clé, ainsi que les informations de l'utilisateur seront enregistrées dans un fichier JSON chargé au lancement du serveur et mis à jour à chaque modification. Les mots de passe seront hachés en utilisant SHA-384.

Lors de la connexion d'un utilisateur déjà enregistré, le *RequestHandler* vérifie les informations d'identification et transmet au client la liste de ses fichiers (qui en fait la demande).

Lorsqu'un fichier est soumis par le client au *RequestHandler*, celui-ci est chiffré avec la clé AES générée lors de l'enregistrement de l'utilisateur concerné. Ensuite, le fichier chiffré est enregistré sur, *au moins*, un *storage provider*. Pour des raisons de confidentialité, le nom du fichier est haché (avec SHA-384). Ainsi, le *storage provider* ne peut, ni déterminer le nom du fichier (car celui-ci est haché), ni lire son contenu (car celui-ci est chiffré). Le *RequestHandler* mémorise le nom du fichier et le(s) *storage provider(s)* sur lesquels chaque fichier a été enregistré.

Lorsqu'un fichier est demandé par le client au *RequestHandler*, celui-ci est rapatrié depuis l'un des *storage providers* qui le contient. Il est ensuite déchiffré avec la clé AES de l'utilisateur et est envoyé au client. Pour retrouver le fichier, il faut hacher son nom (puisqu'il est connu sous ce nom par les *storage providers*). Bien sûr, seuls les fichiers enregistrés par cet utilisateur peuvent être téléchargés.

Lorsque le client demande la suppression d'un fichier au *RequestHandler*, celui-ci transmet l'ordre aux *storage providers* (après avoir haché le nom du fichier). Bien sûr, seuls les fichiers enregistrés par cet utilisateur peuvent être supprimés.

1.1.3 Le StorageProvider

Le *Storage Provider* est l'élément final de notre système : il permet le stockage de l'information. Les services proposés sont : ajouter un fichier, supprimer un fichier, télécharger un fichier. Le contenu du fichier est chiffré (opération à charge du RequestHandler) et sa dénomination est le résultat d'un hach de son nom initial (en utilisant SHA384, cette opération est également réalisée par le *RequestHandler*).

Lors de l'ajout d'un fichier, le fichier est envoyé par le *RequestHandler* en fournissant son nom haché, sa taille et l'empreinte du contenu (hach du contenu du fichier, utilisé pour vérifier si aucune erreur de transmission n'est survenue). Ensuite, le fichier est envoyé et est stocké tel quel par le *Storage Provider*.

Pour récupérer un fichier, il faut mentionner le nom *haché*. Le fichier est envoyé avec sa taille et l'empreinte du contenu (afin de détecter toute erreur de transmission).

La **destruction d'un fichier** s'opère en mentionnant le nom haché du fichier à détruire. En aucun cas le *StorageProvider* ne peut déterminer qui est le propriétaire du fichier, son contenu ou son nom.

Par facilité, **chaque *Storage Provider* disposera d'un identifiant unique**. Cet identifiant est annoncé, en multicast, toutes les 30 secondes au *RequestHandler*. Cette annonce mentionne également le port unicast TCP sur lequel le *RequestHandler* peut se connecter pour échanger des informations. Il n'y a pas de méthodes d'authentification prévues, aussi par souci de sécurité, il n'est pas possible de « lister » les fichiers enregistrés sur le *Storage Provider*.

2. Protocoles

Dans ce paragraphe, nous allons détailler les différents protocoles applicatifs. Ainsi, il y a le protocole utilisé *entre le RequestHandler* et le(s) *storage provider(s)*. Il y a les annonces multicast des *storage providers* permettant de s'annoncer. Et enfin, il y a le protocole utilisé entre *un client* et le *request handler*.

Tous les protocoles sont décrits sous la forme d'une *grammaire ABNF*[1]. Ainsi, la syntaxe est strictement spécifiée et les implémentations doivent s'y conformer précisément.

2.1 Définitions standards

```
lettre = %d65-90 / %d97-122      ; [A-Z][a-z]
chiffre = %d48-57                ; [0-9]
lettre_chiffre = lettre / chiffre ; une lettre ou un chiffre
crlf = %d13 %d10                 ; \r suivi de \n
port = 1*5chiffre                ; 1 à 5 chiffre (1 à 65535)
caractere = %d32-255             ; tous les caractères imprimables et l'espace
caractere_pass = %d34-255        ; tous les caractères imprimables sauf !
binaire = %d0-255                ; un caractère quelconque
sp = %d32                        ; espace
id = 5*10lettre_chiffre          ; 5 à 10 lettres/chiffres
hash = 50*200lettre_chiffre      ; 50 à 200 lettres/chiffres
hashnom = hash
hashcontenu = hash
taille = 1*10chiffre             ; 1 à 10 chiffres
nomfichier = 1*20(lettre_chiffre/".")   ; 1 à 20 lettres/chiffres et le point
mot_passe = 3*50caractere_pass   ; 3 à 50 caractères imprimables
infofichier = hashnom sp taille sp hashcontenu
```

2.2 Echanges multicast entre le *StorageProvider* et le *RequestHandler*

Les *StorageProvider* envoient leurs messages *en multicast* vers le *RequestHandler*. Il n'y a pas de connexion du *StorageProvider* vers le *RequestHandler* (mais bien l'inverse).

Les échanges multicast ont lieu dès le démarrage du *StorageProvider*, et toutes les 30 secondes. Le message prend la forme suivante :

```
annonce_provider = "BIP" sp id sp port crlf
```

Le message *annonce_provider* est envoyé par le *StorageProvider* en multicast sur l'adresse `224.224.200.1` et *le port propre à votre groupe*[2], toutes les `30` secondes en mentionnant son identifiant et le port TCP unicast sur lequel le *RequestHandler* peut le contacter. Il faut noter que l'adresse IP du *StorageProvider* est obtenue en analysant la source de l'annonce.

2.3 Protocole entre le *RequestHandler* et le *StorageProvider*

Le *RequestHandler* contacte le *StorageProvider* suite à la réception de son annonce multicast. Cette annonce comprend l'identifiant du *StorageProvider* ainsi que le port à utiliser pour la connexion (remarquons que l'adresse IP du *StorageProvider* est extraite de l'annonce multicast).

Les messages du protocole sont (en rouge, les requêtes du *RequestHandler -> StorageProvider*) :

```
envoi_fichier = "UPFILE" sp infofichier crlf 0*(binaire)
reponse_envoi = ("FILEOK" / "FILEERR") crlf
supprime_fichier = "DELFILE" sp hashnom crlf
reponse_supprime = ("DELOK" / "DELERR") crlf
recoit_fichier = "DOWNFILE" sp hashnom crlf
reponse_recoit = ("DOWNOK" sp infofichier crlf 0*(binaire)/"DOWNERR" crlf)
```

Dans le cas de `envoi_fichier` et `reponse_recoit`, la fin du message correspond à l'envoi des octets[3] du fichier (son contenu). Il faut noter que la connexion entre le *RequestHandler* et le *StorageProvider* est persistente. Afin d'éviter de surcharger le *StorageProvider*, il convient d'implémenter une file de tâche unique au niveau du *RequestHandler*.

---

[1] Augmented Backus-Naur Form (RFC 5234 – https://tools.ietf.org/html/rfc5234)
[2] Chaque groupe d'étudiants doit utiliser un port réseau qui lui est propre
[3] Nous considérerons que les machines de chaque coté sont de mêmes types (même architecture)

2.4 Protocole entre le Client et le RequestHandler

Le *Client* se connecte au *RequestHandler*. La 1ère étape est l'authentification : soit l'utilisateur fournit ses identifiants, soit il faut qu'il s'enregistre (rouge = requêtes *Client -> RequestHandler*).

```
connexion_client = "CONNECT" sp id sp mot_passe crlf
enregistrement_client = "REGISTER" sp id sp mot_passe crlf
reponse_authentification = ("OK"/"ERR") crlf
```

Il faut bien noter que lors de *l'enregistrement d'un utilisateur*, le *RequestHandler* génère la clé AES de chiffrement qui sera utilisée pour tous les fichiers de ce client. Cette clé, ainsi que les informations d'identification (*id* et *mot de passe haché*) seront enregistrées dans un fichier de configuration JSON sauvegardé sur le *RequestHandler*. Il faut remarquer que la connexion peut échouer si les informations de connexion ne sont pas correctes. Il en va de même lors de l'enregistrement si un utilisateur avec cet identifiant est déjà enregistré. En cas d'erreur, la connexion est coupée.

Une fois connecté, les messages suivants sont possibles (rouge = requêtes *Client -> RequestHandler*):

```
obtient_liste = "REQLISTFILE" crlf
reponse_liste = "LISTFILES 0*50(sp nomfichier!taille) crlf
store_fichier = "STORE" sp nomfichier sp taille crlf 0*(binaire)
reponse_store = ("STOREOK"/"STOREERR") crlf
recupere_fichier = "RETRIEVE" sp nomfichier crlf
reponse_recupere = ("RETROK" sp nomfichier sp taille crlf 0*(binaire) / "RETRERR" crlf)
detruit_fichier = "WIPE" sp nomfichier crlf
reponse_detruit = ("WIPEOK"/"WIPEERR") crlf
quitte = "BYE" crlf
```

Comme nous pouvons le voir, les messages permettent d'obtenir la liste des fichiers de l'utilisateur (`obtient_liste` et réponse du *RequestHandler* `reponse_liste`), d'enregistrer un fichier sur le système (`store_fichier` et réponse du *RequestHandler* par `reponse_store`), de récupérer un fichier existant (`recupere_fichier` et réponse du *RequestHandler* par `reponse_recupere`) et de détruire un fichier (`detruit_fichier` et réponse du *RequestHandler* par `reponse_detruit`). L'enregistrement peut échouer s'il n'y a pas, au moins, 1 *StorageProvider* disponible ou pour des raisons techniques (plus d'espace libre, …). La récupération d'un fichier peut échouer si le(s) *StorageProvider* hébergeant ce fichier ne sont pas disponibles au moment de la demande. La destruction peut échouer si tous les *StorageProvider* ne sont pas disponibles au moment de la demande ou si le fichier n'existe pas. Enfin, le client peut également se déconnecter (message `quitte`).

3. Machine à états finis

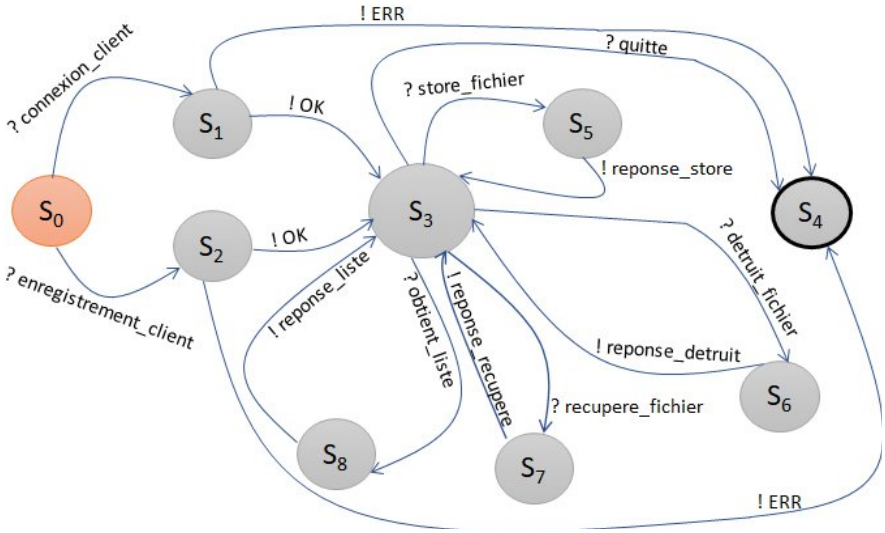

Figure 2 : Machine à états finis

La figure 2 montre la machine à états finis entre un client et le *RequestHandler*. Cette machine permet de montrer la succession des différents messages. Il faut noter que lorsqu'un message inconnu (non conforme par exemple) ou invalide (non attendu à ce moment-là) est reçu, il est ignoré.

L'état S_0 est l'état initial et l'état S_4 est l'état final. Les états S_1 et S_2 représentent la phase d'authentification / enregistrement. Si cette étape n'aboutit pas (une erreur est rencontrée), la connexion est abandonnée et l'on arrive dans l'état final S_4.

Une fois l'état S_3 atteint, l'utilisateur est identifié et peut demander *la liste de ses fichiers* (via la commande `obtient_liste`), il arrive alors dans l'état S_8 puis, lorsque le *RequestHandler* répond (par le message `reponse_liste`), il retourne dans l'état S_3, *l'ajout d'un nouveau fichier* (via la commande `store_fichier`), il arrive alors dans l'état S_5 puis, lorsque le *RequestHandler* répond (par le message `response_store`), retourne dans l'état S_3. Le client peut également *récupérer un fichier* (via la commande `recupere_fichier`), il arrive alors dans l'état S_7. Il retourne dans l'état S_3 lorsque le *RequestHandler* a envoyé sa réponse (via le message `reponse_recupere`). Enfin, le client peut demander *la suppression d'un fichier enregistré* (via le message `detruit_fichier`) ; il arrive alors dans l'état S_6. Une fois la destruction achevée, le *RequestHandler* répond au client (par le message `reponse_detruit`) et retourne dans l'état S_3. Enfin, le client peut quitter le système (par le message `quitte`) et arrive ainsi dans l'état final S_4.

4. Anatomie du RequestHandler

L'élément central du système est le *RequestHandler*, nous allons, dès lors, explorer un peu son architecture.

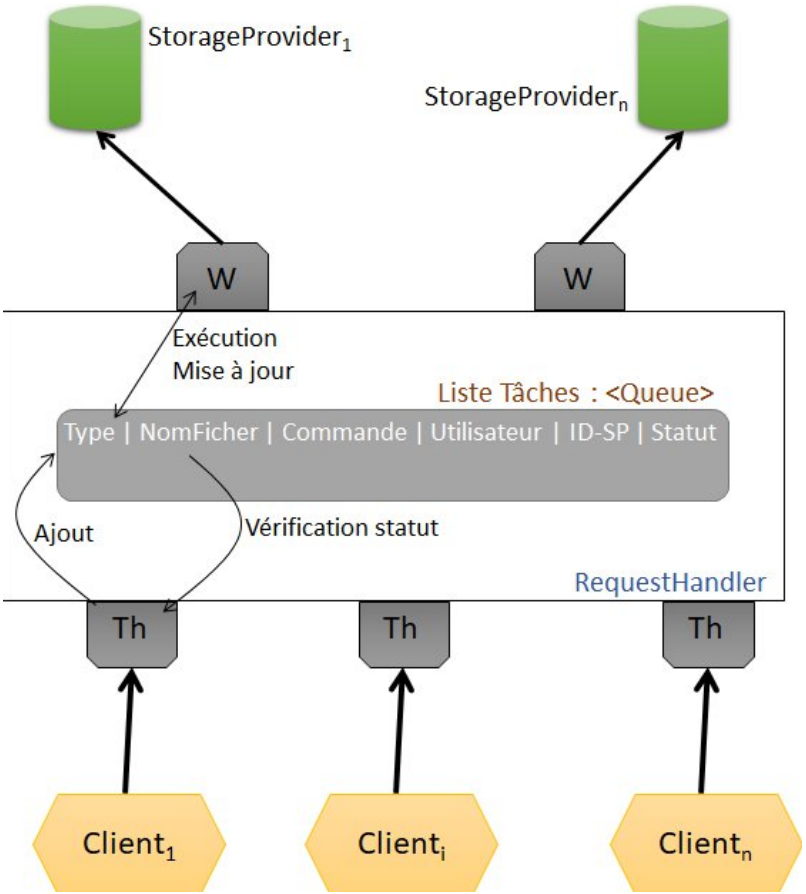

Figure 3 : Architecture du *Request Handler*

Comme nous pouvons le voir sur la figure 3, le *RequestHandler* créée de nombreux threads, au moins un par connexion avec chaque client et un par connexion avec chaque *Service Provider* (que j'appellerai *Worker* pour les différencier). Il faut mettre en place un mécanisme d'interaction entre les *Threads* et les *Workers* : l'envoi d'un fichier par un client doit être finalement transmis au *Storage Provider*.

Pour permettre d'y arriver, une solution pourrait être la mise en place d'une **file de tâches**. Cette file serait alimentée par les *Threads* qui ajouteraient des tâches et par les *Workers* qui exécuteraient celles-ci et mettraient à jour les statuts. Les *Threads* pourraient alors répondre aux clients en fonction de ce statut.

4.1 Ebauche d'une tâche

La tâche pourrait contenir les informations suivantes : ***le type de tâche*** (envoi, réception, suppression d'un fichier), ***le nom du fichier*** (le nom original, non haché), ***la commande*** (c'est-à-dire le message du protocole qui doit être envoyé par le *Worker* comme `envoi_fichier`, `supprime_fichier` ou `recoit_fichier`), ***l'utilisateur*** (qui est à l'origine de cette tâche), ***l'identifiant du service provider concerné*** (qui permet au « bon » *Worker* d'exécuter la tâche) et enfin ***le statut*** de cette tâche (*en cours*, *terminée*, *erreur*, …).

4.2 Dynamique du système

Les *Threads* alimentent la file de tâche et *attendent* qu'une tâche soit terminée. Nous supposerons qu'un *thread* ne crée qu'une tâche à un instant donné (impossible d'envoyer un second fichier tant que le premier envoi n'est pas finalisé).

Chaque *Worker* récupère la 1ère tâche qui le concerne (en fonction de l'identifiant du *Storage Provider*) et exécute celle-ci. Une fois l'exécution réalisée, le statut de la tâche est mis à jour. Ainsi, lorsque le *Thread* consulte la tâche et voit le statut, il peut répondre valablement au client.

4.3 Envoi et réception de fichiers et chiffrement

Afin de gérer facilement l'envoi et la réception de fichier, il peut être utile que le *RequestHandler* dispose d'un *répertoire de travail*. Ainsi, lorsqu'un fichier est envoyé par le client, il est enregistré, temporairement, sur le RequestHandler avant l'envoi vers le StorageProvider.

Pour le chiffrement de l'information, il y a 2 possibilités : soit le fichier est chiffré après la réception de celui-ci par le client, soit il est chiffré pendant la réception. Cette seconde méthode est plus facile à gérer et est aussi plus sûre : le fichier est chiffré en arrivant sur le *RequestHandler*. Donc une méthode simple pour gérer ce chiffrement est de :

- Chiffrer le fichier qui est reçu par le *RequestHandler* (il peut alors être transmis « tel quel » au *Storage Provider*).
- Déchiffrer le fichier quand il est envoyé au client

5. Exigences techniques et liens utiles

Votre implémentation devra, impérativement :

- **Utiliser des expressions régulières** pour valider tous les messages (pas de split / substring autorisé). Les langages Java / C# proposent des implémentations complètes.
- Même si vous ne supportez l'enregistrement d'un fichier que sur un seul *storage provider*, **le *request handler* doit permettre la connexion à plusieurs *storage provider*** (donc des threads sont nécessaires).
- Au niveau du *Request Handler*, **une séparation importante doit être opérée entre *le traitement avec le client* et le *traitement avec le Storage Provider***.

Voici quelques liens utiles. Même si ceux-ci sont principalement proposés pour Java, le passage vers C# ne devrait pas poser un problème (fonctionnement similaire) :

- Cryptographie
  - [JAVA] Chiffrement AES-256 : https://stackoverflow.com/questions/992019/java-256-bit-aes-password-based-encryption
  - [JAVA] Génération d'un sel : https://stackoverflow.com/questions/18142745/how-do-i-generate-a-salt-in-java-for-salted-hash
  - [JAVA] Conversion d'une clé secrète en chaîne de caractères : https://stackoverflow.com/questions/5355466/converting-secret-key-into-a-string-and-vice-versa
  - [JAVA] Création d'un certificat SSL autosigné : https://www.sslshopper.com/article-how-to-create-a-self-signed-certificate-using-java-keytool.html
  - [JAVA] Support SSL : https://stackoverflow.com/questions/18787419/ssl-socket-connection
- JSON et Java : La libraire Google Gson - https://github.com/google/gson

6. Extensions

- Techniques
  - Permettre la sauvegarde d'un fichier sur plusieurs *Storage Providers*
  - Récupération depuis plusieurs *Storage Providers* en parallèle (téléchargement des morceaux du fichier plutôt que le fichier entier)
  - Authentification à plusieurs facteurs (Google Authenticator, …)
  - Sauvegarde de la configuration du *RequestHandler* sur le(s) *Storage Providers* directement
- Développement
  - Développer tout/une partie du système dans un langage différent mais compilé comme Kotlin, C/C++/Qt, Google Go, … (donc pas de langage de scripts comme NodeJS, Python, Perl, …)
- Nouvelles fonctionnalités
  - A proposer à votre professeur de laboratoire pour que celui-ci valide

7. Planning

Voici le planning à respecter pour ce laboratoire :

Phase	Description	Heures[4]
1	Implémentation du **Storage Provider**. • Mise en place du multicast • Mise en place du serveur • Envoi / réception de fichiers	**8**
2	Implémentation du **RequestHandler** - Discussion avec le(s) Storage Provider • Ecoute multicast et connexion vers les *providers (! threads)* • Mise en place de la liste de tâches • Analyse des requêtes/réponses vers le storage providers	**8**
3	Implémentation du **RequestHandler** – Discussion avec le(s) clients • Serveur d'écoute et support SSL/TLS • Analyse des requêtes/réponses vers le client	**8**
4	Extensions • Groupe de 3 étudiants : 1 pour aller au-delà de 14/20 • Groupe de 4 étudiants : 1 **obligatoire** pour atteindre le seuil de 14/20 ; et une 2ème pour aller au-delà	**4**
Eval	Présentation du projet durant la session de juin	

Les phases 1 à 3 doivent être terminées pour espérer la réussite de ce laboratoire. Si les phases 1 à 3 **sont terminées et conformes EXACTEMENT à ce qui est demandé**[5], la note obtenue sera de **MAXIMUM** 14/20. Pour espérer obtenir une note supérieure, il faut terminer la 4ème phase.

Une présence **d'au moins à la moitié des laboratoires de tous les membres du groupe** est requise pour être autorisé à défendre le projet.

La version qui sera défendue sera celle **déposée le dimanche 19 mai à 23h55** sur le GitLab HELMo au plus tard.

Une évaluation des pairs vous sera également demandée, individuellement. Celle-ci pourra être utilisée pour ajuster la note de chaque membre du groupe.

[4] Il s'agit des heures de laboratoire : j'ai tenu compte que vous étiez, au moins, 3 étudiants par groupe
[5] C'est-à-dire si les exigences techniques sont rencontrées et que, fonctionnellement, tout est implémenté.