HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Prof. Yann Thoma

Jonas Chapuis

Laboratoire de Programmation Concurrente
semestre printemps 2019

Serveur de fichiers

Temps à disposition :

- 4 périodes (Multi-threading, première phase)
- 4 périodes (Threadpool, deuxième phase)
- 4 périodes (Cache, troisième phase)

⚠ Pour ces 3 laboratoires, le rendu ne se fera plus avec 'client.py', mais avec une achive *.zip de votre travail. Cela signifie :

- Ne pas oublier de faire un clean de votre projet avant la création de l'archive.
- le rendu se fera avec une archive *.zip de votre dossier à remettre sous cyberlearn.
- Vous pouvez créer les fichiers que vous jugez nécessaire.
- Vous pouvez utiliser git (commit/push) si vous le désirez.

1 Objectifs pédagogiques

Réaliser un programme serveur supportant le traitement de requêtes simultanées grâce à de la concurrence et le paradigme producteur-consommateur. Dans une deuxième phase, introduire un thread pool pour contrôler le flux de requêtes. Finalement, dans la troisième phase ajouter une cache.

2 Cahier des charges

L'objectif est de réaliser un serveur de fichiers qui puisse supporter la montée en charge du nombre de clients et de requêtes grâce à de la concurrence. Le client ainsi qu'un squelette de serveur sont fournis.

Le client est une application web simple qui permet à l'utilisateur de rentrer un chemin du système de fichiers pointant sur un fichier texte et qui affiche le contenu correspondant sur la page. Le client permet aussi de changer l'adresse du serveur et se connecte/reconnecte automatiquement. Il permet par ailleurs de simuler l'envoi simultané d'un grand nombre de requêtes [1].

Le canal de communication est une WebSocket. WebSocket est un protocole réseau applicatif qui offre une communication full-duplex par dessus une connection TCP, et qui est bien supporté par les browsers actuels ainsi que par Qt. C'est une technologie typiquement utilisée pour les applications web modernes en page unique avec un haut degré d'interactivité, car il permet un échange entièrement bi-directionnel avec une faible latence entre un client et un serveur.

Le serveur suit l'exemple de la documentation Qt d'un serveur WebSocket. La classe QWebSocketServer implémente un serveur WebSocket, et envoie un signal `newConnection()` à chaque fois qu'un

1. Pour les curieux, le client web est un exemple de *programmation réactive* qui est une approche fonctionnelle à la concurrence. Les sources (écrites en TypeScript) sont incluses dans le projet et permettent de se faire une idée à quoi cela ressemble - mais cela sort du cadre de ce cours. Vous pouvez aussi trouver plus d'informations sur son principe de fonctionnement ici et ici

client se connecte. L'appel à `nextPendingConnection()` dans le handler permet d'obtenir une instance de la classe QWebSocket, qui permet à son tour d'interagir avec le socket. L'envoi des messages se fait à l'aide des méthodes `sendTextMessage(const QByteArray&)` et `sendBinaryMessage(const QString&)`, et la réception à l'aide des signaux `binaryMessageReceived(const QByteArray&)` et `textMessageReceived(const QString&)`. Dans le cadre de ce laboratoire, nous nous limiterons à l'échange de messages (et fichiers) texte.

Qt est un framework basé événements et toutes les interactions entrées-sorties sont asynchrones. QWebSocketServer est mono-threadé et gouverné par le thread principal de l'application, qui fonctionne avec un principe de boucle d'événements. Par exemple, l'arrivée d'un nouveau message se matérialise par l'insertion d'un événement dans la file. Le traitement de cet événement par le thread principal déclenche l'exécution du signal correspondant et de ses méthodes attachées. Tout travail important dans la méthode de traitement du message a donc pour effet de ralentir l'exécution entière de l'application. Cette implémentation ne permet pas de tirer parti des différents cœurs du processeur. En Qt, l'utilisation d'un socket est même restreint au thread qui l'a créé (le thread principal dans notre cas).

3 Mise à jour de la VM

La commmande suivante permet d'installer le support pour des websockets dans la VM.

```
sudo apt install libqt5websockets5-dev
```

4 Première phase - Multi-threading

4.1 Récupération des sources

La récupération du code source s'effectue à l'aide de la commande git suivante :

```
$ git clone https://gitlab.com/<username>/pco19.git pco19_lab05 --branch lab05
```

4.2 Travail demandé

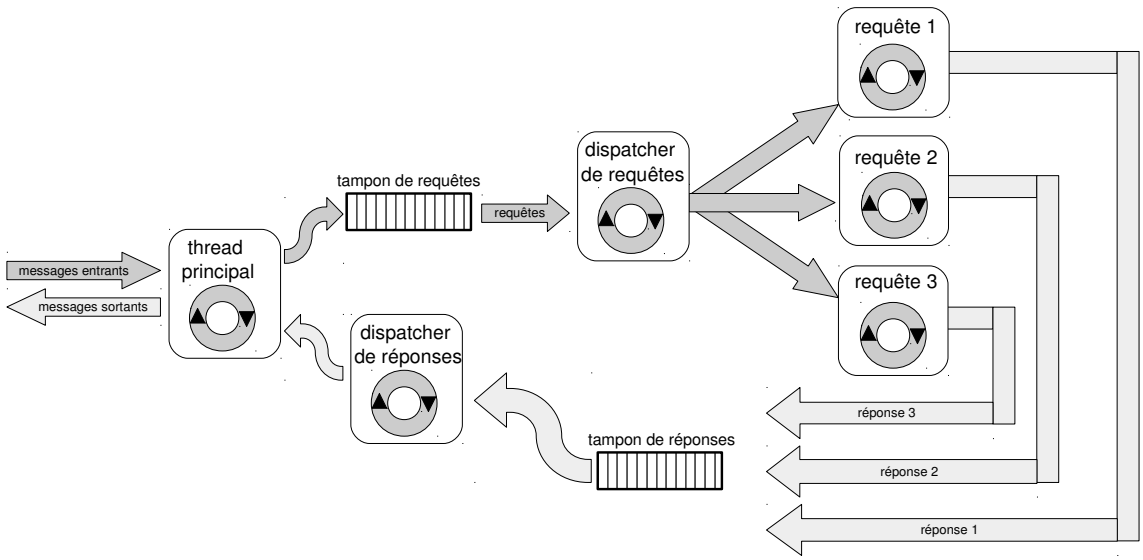
Le premier exercice est d'expérimenter ces limitations avec le code fourni. Le projet QtCreator dans le répertoire `fileserver` compile en l'état, et peut être exécuté directement dans QtCreator ou en ligne de commande. Pour le lancer avec les log activés, ajoutez le flag `-d` sur la ligne de commande (ou dans la configuration de QtCreator). Il est aussi possible de changer le port avec le flag `-p` (1234 par défaut). Pour accéder à l'application client, ouvrez simplement le fichier `webclient/dist/index.html` dans un browser. L'application permet de configurer l'adresse et le port du serveur (`http://localhost:1234` par défaut). Expérimentez la vitesse de réaction en chargeant le ficher `shakespeare.txt` qui contient 5MB des œuvres de Shakespeare. Pour ce faire, insérez le chemin complet du fichier dans la boîte de texte. En augmentant artificiellement le nombre de requêtes, vous devriez apercevoir que le serveur a de la peine à tenir la charge et que néanmoins seulement un cœur est utilisé sur votre machine.

On peut constater ce phénomène sur la copie d'écran suivante (prise sous Windows) : on aperçoit en effet les trois threads de l'application `fileserver.exe`. Le premier de la liste est le thread principal de l'application, qui comme on le voit est occupé à servir les fichiers. Les deux threads suivants sont inactifs et consistent en des opérations systèmes internes à Qt (probablement bloquantes).

TID	CPU	Cycles Delta	Start Address
8992	12.52	4,233,108...	fileserver.exe+0x14e0
17420		56,214	mswsock.dll!sethostname+0xe80
7808			ntdll.dll!TpTimerOutstandingCallbackCount+0x670

Notez cependant que les opérations de lecture disque étant très rapides et que ce labo ne simule l'accès que par un seul client, la lecture de fichiers a été ralentie artificiellement par une boucle d'attente active, afin de pouvoir observer ces phénomènes de performance. On peut imaginer que dans un cas réel cette attente active représente une opération lourde a réaliser sur le fichier, par exemple une traduction, un encodage/décodage, etc.

Cette expérience nous amène à conclure que pour pouvoir effectuer des opérations lourdes en réponse à une requête, il faut trouver le moyen de déléguer ce travail à d'autres threads. Une deuxième version du serveur vous est fournie dans le répertoire `fileserver-threadperreq`. Cette version fonctionne avec une stratégie d'un thread par requête, tout en respectant la contrainte Qt que seul le thread principal peut interagir avec le socket. Le thread principal délègue la réception d'une requête à un thread annexe, qui attend en boucle infinie sur une file de requêtes. Dès qu'il est réveillé, ce thread de dispatch lance un thread dédiée au traitement de la requête. Une fois la réponse prête, ce thread dédié l'empile dans une file de réponses. Un deuxième thread annexe attend en boucle sur cette file de réponses, et est responsable de signaler au thread principal qu'une réponse est prête à être renvoyée.



Cette version multi-threadée s'articule donc autour de la classe `AbstractBuffer<T>` qui définit une file bloquante suivant le paradigme producteur/consommateur.

Attention, le code fourni ne compile pas en l'état car il suggère l'usage de certaines classes que vous

devez implémenter. Ceci est totalement normal et voulu.

Les étapes de la première phase sont donc les suivantes :

— Ajoutez une implémentation de l'interface `AbstractBuffer<T>` qui permettra aux threads de s'échanger requêtes et réponses.

— Implémentez le thread de dispatching des requêtes (celle pour les réponses est fournie étant donné qu'elle fait appel à des mécanismes de signal/slot).

— Implémentez le thread de traitement de requête.

— Comparez la performance de cette version concurrente avec celle de la version de base. Constatez-vous une amélioration ?

— Que se passe-t-il lorsqu'on lance un nombre de requêtes très important (par ex. 10'000), et comment l'expliquez-vous ?

— Comment pourrait-on protéger le serveur de cet effet néfaste ?

5 Deuxième phase - Threadpool

5.1 Récupération des sources

La récupération du code source s'effectue à l'aide de la commande git suivante :

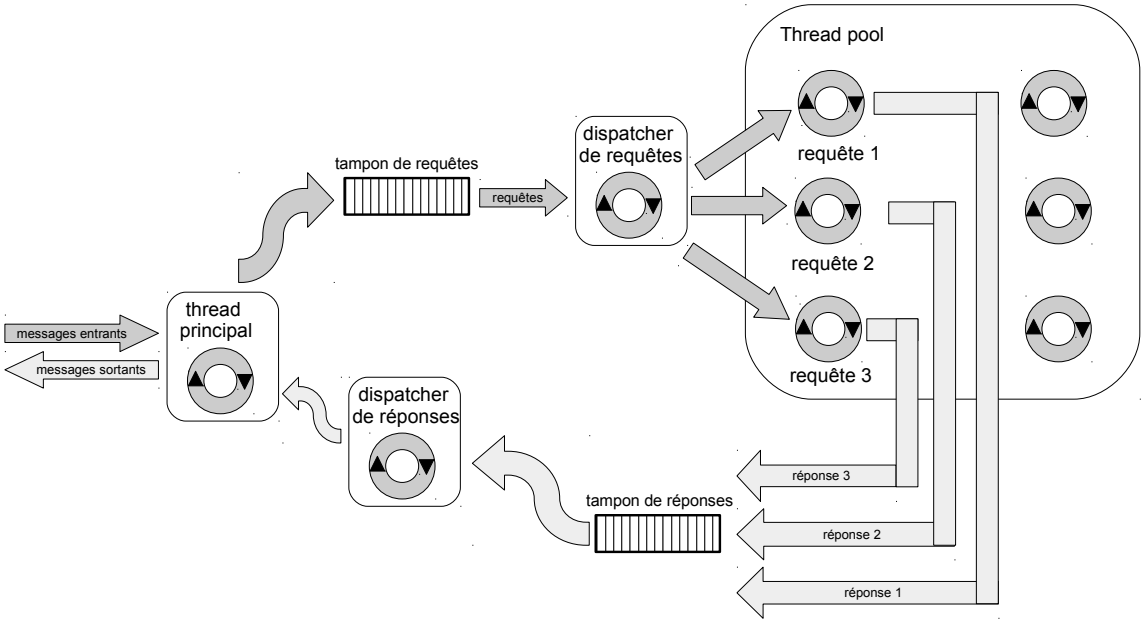
```
$ git clone https://gitlab.com/<username>/pco19.git pco19_lab06 --branch↘
→ lab06
```

5.2 Travail demandé

Afin d'éviter les phénomènes néfastes liés au lancement d'un thread par requête du point de vue des temps de réponse et de l'utilisation des ressources, nous allons introduire un mécanisme de *ThreadPool*.

Avec un thread pool, l'idée est d'allouer dynamiquement des threads au fur et à mesure que cela devient nécessaire dans une "piscine" de threads. La piscine a un taille maximale : une fois sa capacité atteinte, les threads sont "recyclés" pour traiter les requêtes suivantes.

Le schéma du serveur devient le suivant :



Le système de thread pool limite donc le "débit" maximal du serveur. Si le serveur est temporairement débordé, la latence augmente malgré tout pour les clients, mais avec la garantie qu'il ne va pas s'écrouler sous la demande.

Il faut aussi ajouter au thread pool un mécanisme de gestion de surplus de requêtes : en effet, sans limitations, la file d'attente des requêtes peut devenir suffisamment grande pour arriver à la limite de la mémoire du serveur. Pour cette raison, il s'agit aussi de limiter la file à une taille maximale et rejetter les requêtes entrantes si elle est pleine. Cette mesure produit une expérience temporairement dégradée pour les clients, mais garantit que le serveur peut se récupérer une fois la "tempête" passée.

Jusqu'à présent, nous avions "représenté" les tâches par des sous-classes de `QThread`. Etant donné que le thread pool gère ses propres threads, cela n'est plus possible. Nous avons donc besoin d'une autre abstraction.

La définition ci-dessous est proposée :

```
class Runnable {
public:
    virtual ~Runnable() {}
    virtual void run() = 0;
    virtual QString id() = 0;
};
```

Et voici le squelette de déclaration pour la classe `ThreadPool` qui utilise cette définition :

```
class ThreadPool
{
public:
        ThreadPool(int maxThreadCount) {...}
```

```
    /* Start a runnable. If a thread in the pool is available, handle ↘
    →the runnable with it. If no thread is available but the pool can ↘
    →grow, create a new pool thread and handle the runnable with it. If↘
    → no thread is available and the pool is at max capacity, block the↘
    → caller until a thread becomes available again. */
       void start(Runnable* runnable) {...}
}
```

Les étapes de la deuxième phase sont donc les suivantes :

- Implémentez la classe `ThreadPool` qui réalise ce concept d'allocation dynamique et de recyclage de threads. Utilisez la classe abstraite `Runnable` pour représenter les requêtes. Notez que le code est censé gérer correctement la destruction du thread pool, nécessitant potentiellement l'utilisation de mécanismes d'interruption de threads.
- Ajoutez une gestion de taille maximale au niveau de la file de traitement des requêtes `AbstractBuffer<T>`, ainsi qu'une méthode supplémentaire `virtual bool tryPut(T item)= 0;`
- Adaptez le code de traitement des requêtes entrantes au niveau du thread principal dans fileserver.cpp pour utiliser cette méthode au lieu de `put` et rejeter les requêtes lorsque la file est remplie avec un message d'erreur, par exemple :

```
pClient->sendTextMessage(Response(req, "server overloaded, try later").toJson());
```

- Observez le nouveau comportement du serveur, et répondez au questions suivantes :
  - Quelle est la taille optimale du thread pool pour cette application ? Quels facteurs influencent ce choix ?
  - Que se passe-t-il maintenant lorsque que l'on inonde le serveur de requêtes ? Constatez-vous une amélioration au niveau de la stabilité du serveur ? et qu'en est-il au niveau de l'occupation mémoire, par rapport à la version d'un thread par requête ?

Pour la réalisation de cette deuxième phase nous vous fournissons un framework de test. Il s'agit de deux scénarios de test qui permettront de vérifier une partie des fonctionnalités de votre implémentation du thread pool. Le nouveau `repo` contient un répertoire `threadpool` qui contient lui-même deux répertoires : `fileserver` et `tst-threadpool`. Le projet à ouvrir est alors celui présent dans `threadpool` : `threadpool.pro`. Il s'agit d'un projet qui considère les deux sous-répertoires comme deux sous-projets. QtCreator sera capable de détecter les tests automatiquement, et vous pourrez exécuter votre application ou les tests (suivez la démonstration en classe).

⚠ Ce n'est pas parce que nos deux scénarios sont validés que votre code est bon. Vous avez le droit d'ajouter des scénarios si vous le désirez. Pour ce faire, il suffit d'ajouter une méthode en-dessous de `testCase2()` et de vous inspirer de ce qui vous est fourni.

6 Troisième phase - Cache

6.1 Récupération des sources

La récupération du code source s'effectue à l'aide de la commande git suivante :

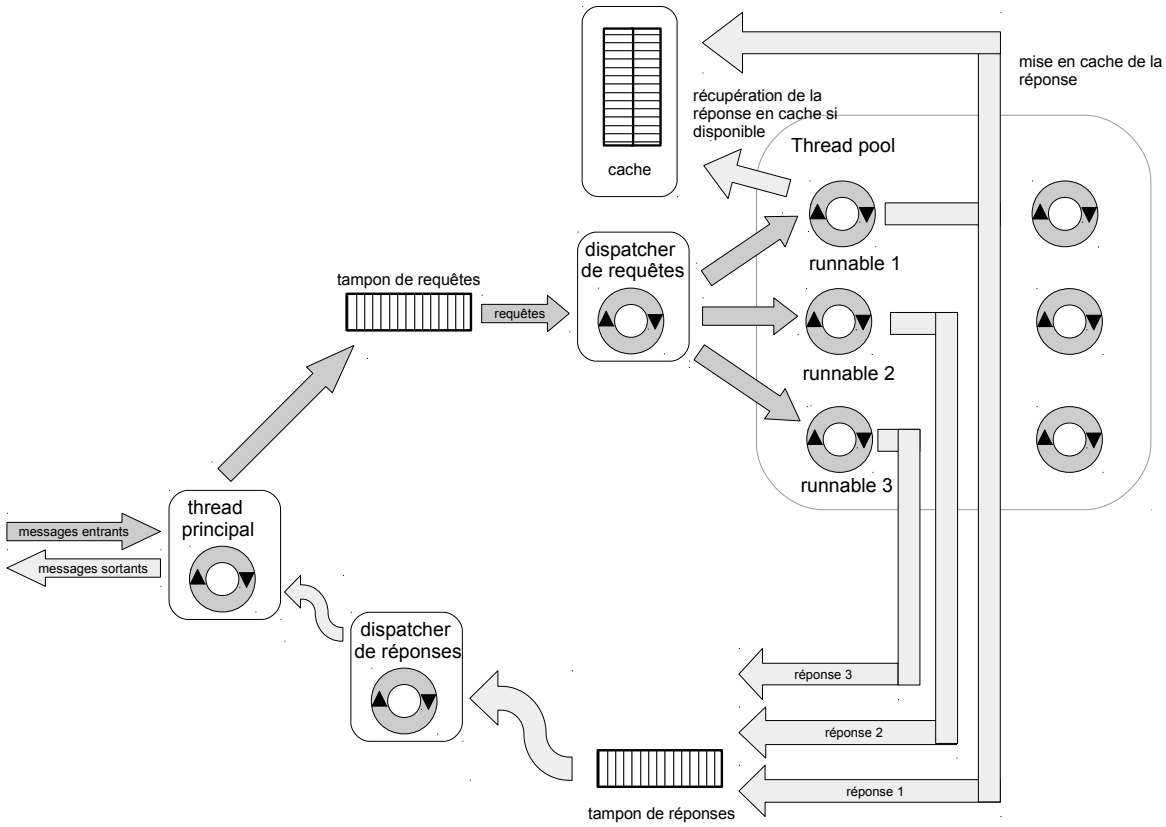
```
$ git clone https://gitlab.com/<username>/pco19.git pco19_lab07 --branch↘
→ lab07
```

6.2 Travail demandé

Dans notre effort d'amélioration de la capacité de montée en charge du serveur, une mesure additionnelle possible est d'introduire un cache. Le cache mémorise les réponses des requêtes récentes et permet de répondre plus vite, diminuant donc la latence. Le cache doit être périodiquement invalidé - ceci en accord avec le domaine. Dans notre cas, nous pouvons considérer qu'une période de délai de 2 minutes entre la modification du fichier et la visibilité de ces changements au niveau des clients est acceptable. Le cache est en effet par essence un compromis entre la "fraîcheur" des données et le temps de réponse, et nécessite de faire des choix. Il est aussi important de remarquer que l'introduction systématique de cache peut nuire à la compréhension du comportement d'un système en production.

Le cache est généralement conçu avec une structure de données clé-valeur : la clé est la signature de la requête (hashcode) et la valeur contient la réponse. L'accès au cache est concurrent entre les threads de traitement de requête : on commence en effet par vérifier si une réponse existe déjà en cache, auquel cas la réponse stockée est renvoyée directement. En cas d'absence, la réponse est préparée comme d'habitude et insérée dans le cache avant de la renvoyer.

L'accès en lecture est en principe plus fréquent que celui en écriture, car la motivation pour l'introduction d'un cache est justement d'accélérer la réponse pour des requêtes courantes. Le schéma du serveur équipé avec le cache est le suivant :



Le squelette de déclaration du cache est donné ci-dessous :

```
class ReaderWriterCache
{
private:
        struct TimestampedResponse {
```

```
            Response response;
            long timestamp;
        };

        class InvalidationTimer: public QThread {
            friend ReaderWriterCache;

        private:
            ReaderWriterCache* cache;

        public:
            InvalidationTimer(ReaderWriterCache* cache): cache(cache↘
            →) {}

        protected:
            void run() {
                    // TODO
            }
        };

        QHash<QString, TimestampedResponse> map;
        int invalidationDelaySec;
        int staleDelaySec;
        InvalidationTimer* timer;
        ReaderWriterLock lock;

public:
        ReaderWriterCache(int invalidationDelaySec, int staleDelaySec);

        Option<Response> tryGetCachedResponse(Request& request);
        void putResponse(Response& response);
};
```

Cette classe exploite la collection `QHash` de Qt, qui implémente un dictionnaire clé-valeur (doc ici). `QHash` génère automatiquement un hashcode de la clé et utilise une hashtable dynamique pour offrir un accès par clé en temps constant.

Les arguments du constructeur permettent de définir les temps pour la gestion du contenu du cache. `invalidationDelaySec` définit l'intervalle entre deux vérifications d'obsolescence de données, et `staleDelaySec` définit le temps après lequel une donnée est considérée obsolète.

Dans le squelette vous pouvez aussi remarquer la déclaration du lock lecteurs-rédacteurs. Cette classe `ReaderWriterLock` devra également être implémentée, afin d'offrir un protocole lecteur-rédacteur au cache. Une fois réalisé, ce cache peut être exploité en modifiant la méthode `run` de la classe `RequestRunnable` ou équivalente (issue du travail réalisé pour l'introduction du thread pool dans la phase 2). Voici un exemple fonctionnel d'implémentation :

```
void RequestRunnable::run()
{
    Response resp;
    Option<Response> cachedResponse = cache->tryGetCachedResponse(req);
    if (cachedResponse.hasValue()) {
        resp = cachedResponse.value();
    } else {
        resp = RequestHandler(req, hasDebugLog).handle();
        cache->putResponse(resp);
    }
```

```
    responses->put(resp);
}
```

Remarquez au passage l'utilisation d'un type `Option` qui permet d'une façon simple de représenter la présence ou non d'un élément en cache. Son listing complet est donné ci-dessous :

```
#ifndef OPTION_H
#define OPTION_H
template <class T>
class Option
{
    bool _hasValue;

    QList<T> _value;

public:
    bool hasValue() {
        return _hasValue;
    }

    T value() {
        return _value[0];
    }

    static Option<T> some(const T& value) {
        return Option(value);
    }
    static Option<T> none() {
        return Option();
    }

private:
    Option(): _hasValue(false) {}
    Option(const T& value): _hasValue(true), _value(QList<T>() << value)↘
    →{}
};
#endif // OPTION_H
```

Un projet vous est fourni, et contient deux sous-répertoires : `fileserver`, et un projet de tests. Ces derniers sont relativement sommaires mais testent quelques fonctionnalités du cache. N'hésitez pas à en rajouter.

Les étapes de la troisième phase sont donc les suivantes :

- Implémentez la classe `ReaderWriterLock`
- Implémentez la classe `ReaderWriterCache`, en incluant un mécanisme de timer pour périodiquement invalider les réponses trop "âgées" (stale data). Le squelette de ce mécanisme est aussi inclus dans le squelette de déclaration de la classe fournie.
- Adaptez le code existant du dispatcher de requêtes pour exploiter le cache.
- Adaptez le code de `RequestRunnable` ou équivalent pour exploiter le cache.
- Observez le nouveau comportement du serveur avec le cache, et répondez au questions suivantes :
  - Le cache améliore-t-il la performance et dans quels cas ?
  - Y a-t-il des schémas d'utilisation qui pourraient être problématiques avec votre implémentation ? Si oui, commentez lesquels et quelles seraient les mesures à prendre pour y palier.

7 Déroulement

Le projet est décomposé en trois phases. Chacune des phases fera l'objet d'un rendu, et ce avant la session de laboratoire où la phase suivante doit commencer. Les consignes habituelles sont à appliquer pour chaque rendu.

⚠ Pour la synchronisation/coordination de tâches, veuillez utiliser le moniteur de Mesa pour chacune des phases.

8 Travail à rendre

— Comme pour les laboratoires précédents, la description de l'implémentation, ses différentes étapes, la manière dont vous avez vérifié son fonctionnement, la réponse aux questions et toutes autres informations pertinentes doivent figurer dans votre fichier README.md. Les fichiers sources doivent évidemment être correctement documentés et commentés. Aucun rapport n'est demandé.

— Inspirez-vous du barème de correction pour savoir là où il faut mettre votre effort.

— Vous *devez* travailler en équipe de deux personnes.

— Vous devez rendre votre travail par groupe sur cyberlearn. Sur la page du cours, rejoignez un groupe existant dans le formulaire mis en place.

9 Barème de correction

Conception, conformité au cahier des charges et simplicité	45%
Exécution et fonctionnement	10%
Codage	15%
Documentation	20%
Commentaires au niveau du code	10%