CSC4509 — Utilisation des Threads

Éric Lallet
Eric.Lallet@telecom-sudparis.eu

Télécom SudParis

30 avril 2025

Introduction
Solution avec plusieurs threads
Threads JAVA
Accès concurrentiel

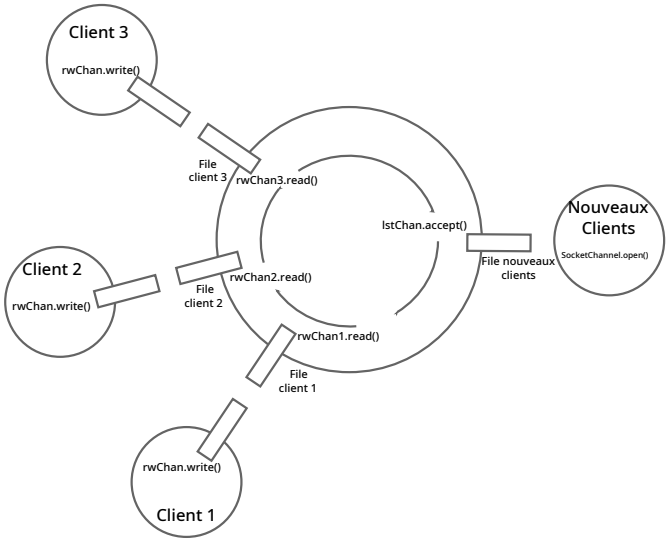
Lors du cours précédent nous avons vu que pour réussir à gérer plusieurs flux TCP sans être géné par les appels bloquants, il y avait deux solutions :

- Rendre les canaux non-bloquants et gérer les évènements avec la classe `Selector` de JAVA NIO ;
- Utiliser plusieurs `threads`.

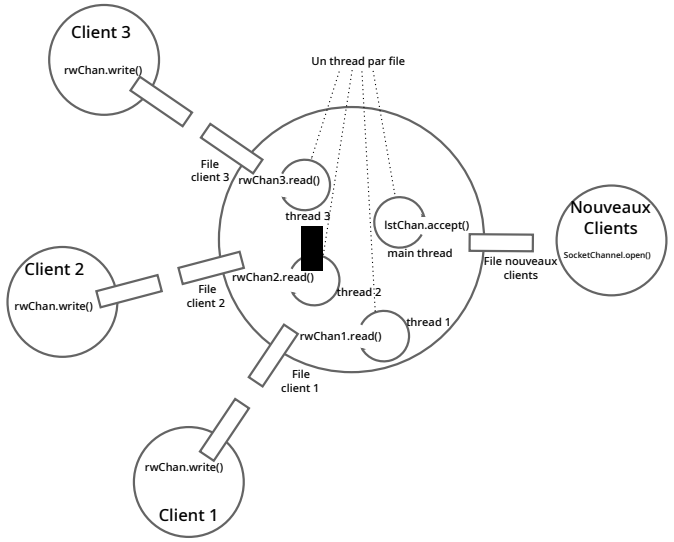
La solution avec les `Selectors` a été vu lors du cours précédent. Ce cours présente la solution avec plusieurs `threads` (solution adaptée pour des serveurs comme ceux de FTP, HTTP...)

Éric LalletEric.Lallet@telecom-sudparis.eu CSC4509 — Utilisation des Threads

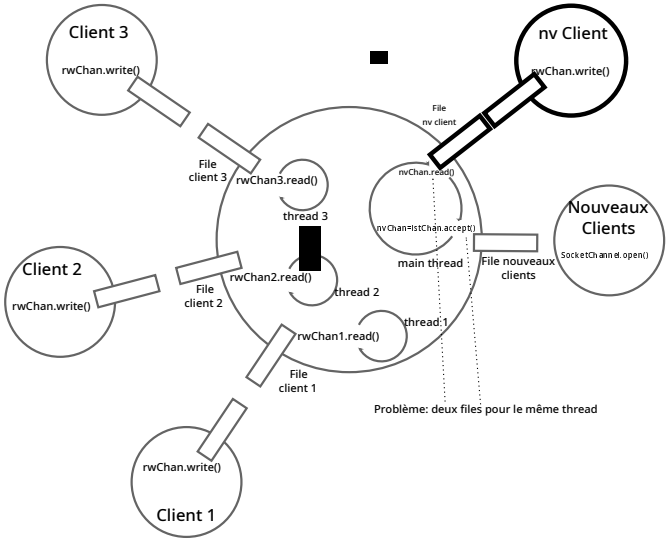
Serveur multiclient : le problème



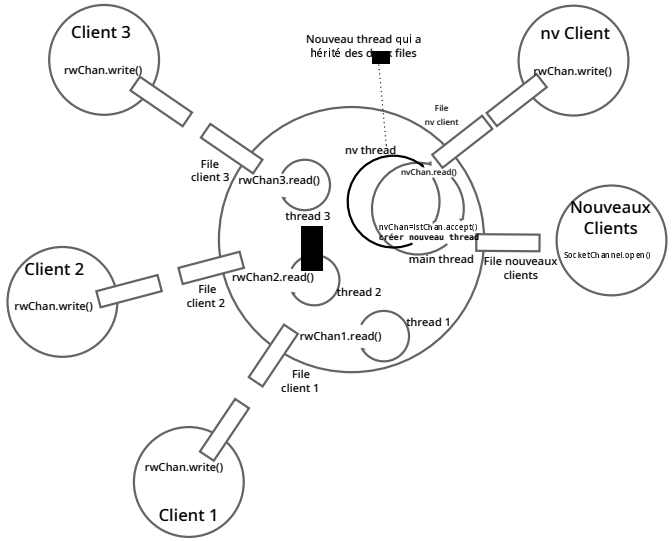
Solution multithread : un thread par file



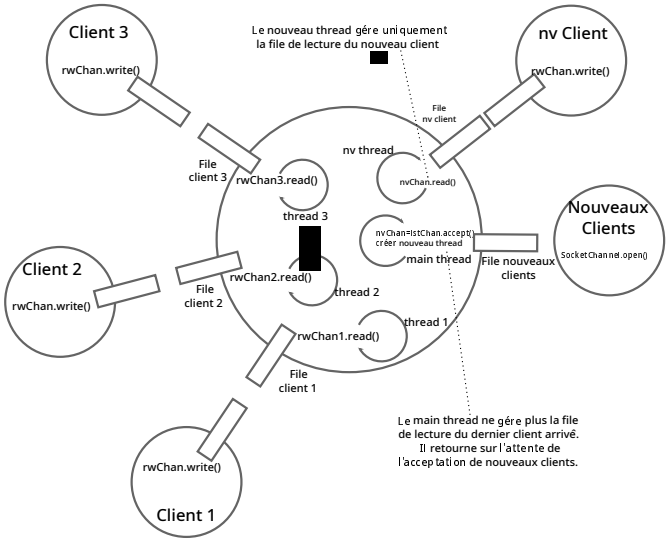
Solution multiThread : gestion d'un nouveau client (1/4)



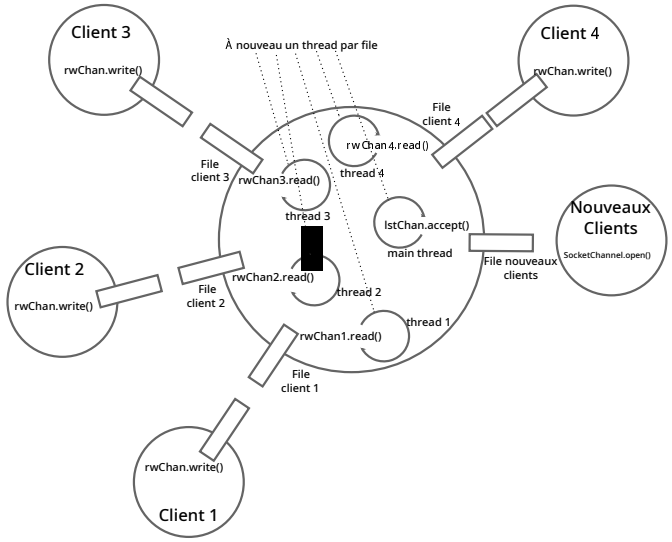
Solution multiThread : gestion d'un nouveau client (2/4)



Solution multiThread : gestion d'un nouveau client (3/4)



Solution multiThread : gestion d'un nouveau client (4/4)



Threads JAVA

Il existe plusieurs solutions pour qu'une classe JAVA puisse être utilisée pour créer un `thread` :

1. Hériter de la classe `Thread` ;
2. Implémenter l'interface `Runnable` ;
3. Implémenter l'interface `Callable<V>`.

Pour les deux premières solutions la classe doit définir une méthode *public void run()*. Pour la troisième elle doit définir une méthode *public V call()*, et le thread appelant doit utiliser une instance de l'interface `Future<V>` pour la réception du résultat.
Ce cours expose les solutions 2 et 3.

Threads JAVA : `Runnable` versus `Callable<V>`

- La méthode *void run()* de l'interface `Runnable` n'a pas de résultat et ne transmet les exceptions. S'il faut transmettre ces informations à un autre thread, il faut ajouter des références communes qui compliquent la programmation ;
- La méthode *public V call()* fournit un résultat et sait transmettre les exceptions. Un simple appel à la méthode *public V get()* de l'interface `Future<V>` permet à un autre thread de les connaître.

Pour un thread qui doit continuer sa vie sans chercher à partager d'informations avec les autres threads, l'interface `Runnable` sera plus simple à mettre en œuvre, et sinon il faudra plutôt choisir l'interface `Callable<V>`.

Threads JAVA avec l'interface Runnable

Avec l'interface Runnable, pour créer un nouveau thread en JAVA, il faut :

- Écrire une classe qui implémente l'interface Runnable ;
- Créer une instance de cette classe ;
- Créer une instance de la classe Thread en passant la référence de la classe Runnable en paramètre du constructeur ;
- Appeler la méthode *start()* de l'instance de *Thread*.

La méthode *start()* de Thread va démarrer le nouveau thread en démarrant la méthode *run()* de la classe Runnable

Introduction
Solution avec plusieurs threads
Threads JAVA
Accès concurrentiel

Threads JAVA avec l'interface Runnable

Exemple d'usage :

```
MaClasseRunnable monRunnable; // une classe Runnable
monRunnable=new MaClasseRunnable();//une instance de cette classe
Thread thread = new Thread(monRunnable); // une instance de Thread
thread.start(); // la méthode run() s'exécute dans le thread
```

La plupart du temps, les variables qui contiennent les références de l'exemple ci-dessus ne servent plus. Donc, généralement on résume ces quatre étapes en une seule ligne :

```
new Thread(new MaClasseRunnable()).start();
```

Éric LalletEric.Lallet@telecom-sudparis.eu CSC4509 — Utilisation des Threads

Note : L'interface Runnable est une interface fonctionnelle, donc le paramètre du constructeur du Thread peut être une expression lambda. Par exemple, si on souhaite exécuter une méthode *simple()* sans paramètre dans un nouveau thread on peut écrire le code :
new Thread(() -> simple()).start() ;

Threads JAVA avec l'interface Callable<V>

Avec l'interface Callable<V>, pour créer un nouveau thread en JAVA, il faut :

- Écrire une classe qui implémente l'interface Callable<V> ;
- Créer une instance de cette classe ;
- Créer une instance d'un classe qui implémente Future<V> (par exemple la classe FutureTask<V>) en passant la référence de la classe Callable<V> en paramètre du constructeur ;
- Créer une instance de la classe Thread en passant la référence de la classe Future<V> en paramètre du constructeur ;
- Appeler la méthode *start()* de l'instance de *Thread* ;
- Récupérer le résultat de la méthode *V call()* par un appel de la méthode *V get()* de Future<V>.

Introduction
Solution avec plusieurs threads
Threads JAVA
Accès concurrentiel

Threads JAVA avec l'interface Callable<V>

```
MaClasseCallable<V> maTache; // une classe Callable<V>

// création d'une instance de la la classe MaClasseCallable
maTache = new MaClasseCallable<V>();

// création d'une instance de la classe Future
FutureTask<V> future = new FutureTask<V>(maTache);

// création d'une instance de la classe Thread
Thread thread = new Thread(future);

// démarage du thread
thread.start(); // la méthode V call() s'exécute dans le thread

// récupération du résultat: par défaut get() bloque
// en attendant la fin du thread.
V result = future.get():
```

Éric LalletEric.Lallet@telecom-sudparis.eu CSC4509 — Utilisation des Threads

Note 1 : Il n'est pas possible de créer un tableau de classes paramétrées. Il faut créer un tableau de la classe sans paramétre :
FutureTask<V> [] mesTachesFutures = new FutureTask[SIZE] ; // cause un warning
Une façon plus propre de répondre à ce besoin consiste à utiliser une Collection :
List< FutureTask<V> > mesTachesFutures = new ArrayList< FutureTask<V> >() ;
Note 2 : Il existe une version de get avec un timeout. Par exemple, si vous voulez attendre 20 secondes au plus, il faut écrire :
V result = future.get(20, TimeUnit.SECONDS) :

Si le thread n'est pas fini au bout de 20 secondes, l'exception TimeoutException est levée.

Accès concurrentiel : le problème

L'accès à la mémoire d'un `thread` peut être partagé :

- Un `thread` peut avoir des références sur des objets ou des tableaux utilisés par d'autres `threads` ;
- Un autre `thread` peut modifier une de vos ressources en passant par les méthodes d'accès publiques ;
- ...

L'accès concurrentiel à une ressource commune va de paire avec la gestion de sections critiques et une politique d'exclusion mutuelle

Accès concurrentiel : les solutions

JAVA fournit de nombreuses solutions pour mettre en œuvre une politique d'exclusion mutuelle :

- La classe `Lock` ;
- La classe `Semaphore` ;
- Les blocs synchronisés ;
- ...

Dans ce cours, nous abordons les blocs synchronisés

Blocs synchronisés

Il existe trois procédés pour gérer une section critique avec les blocs synchronisés.

1. Soit on synchronise sur un objet explicite
2. Soit on synchronise une méthode d'instance
3. Soit on synchronise une méthode de classe

Blocs synchronisés : sur un objet explicite

Pour synchroniser un bloc avec un objet explicite, il faut :

- Avoir une référence sur un objet qui sera partagée par les différents `threads` que vous voulez synchroniser
  Par exemple : Ressource maRessource ;
- Utiliser les instructions suivantes :

```
synchronized(maRessource) {
    // ici les instructions protégées
}
```

  Lorsqu'un `thread` entre dans le bloc synchronisé, les autres `threads` qui veulent entrer dans un bloc synchronisé sur le même objet vont être bloqués jusqu'à ce que le premier `thread` quitte le bloc synchronisé
  Dit autrement, un seul thread pourra être en train d'exécuter des instructions d'un quelconque bloc synchronisé sur une même référence.

Bloc synchronisés : sur une ressource

```
public class MonThread implements Runnable {
private Ressource maRessource;

public MonThread(Ressource maRessource) {
  this.maRessource = maRessource;
}

@Override
public void run() {
  int val1, val2;
 while (true) {
   try {

      val1 = maRessource.getVal();
      maRessource.consommer();
      val2 = maRessource.getVal();

    assert val2 == (val1 -1 );
  } [redacted]

  }
 }
}
}
```

```
public class Ressource {
private int val = 0;

public Ressource(final int val) {
  this.val = val;
}

public int getVal() {
  return val;
}

public  void  consommer()
            throws InterruptedException {
  int maVal = val;

  Thread.sleep(5000);
  maVal--;

  val = maVal;
}
}
```


Blocs synchronisés : sur une ressource

```
public class MonThread implements Runnable {
private Ressource maRessource;

public MonThread(Ressource maRessource) {
  this.maRessource = maRessource;
}

@Override
public void run() {
  int val1, val2;
 while (true) {
   try {
    synchronized (maRessource) {
      val1 = maRessource.getVal();
      maRessource.consommer();
      val2 = maRessource.getVal();
    }
    assert val2 == (val1 -1 );
  } [redacted]

  }
 }
}
}
```

```
public class Ressource {
private int val = 0;

public Ressource(final int val) {
  this.val = val;
}

public int getVal() {
  return val;
}

public  void  consommer()
            throws InterruptedException {
  int maVal = val;

  Thread.sleep(5000);
  maVal--;

  val = maVal;
}
}
```

Blocs synchronisés : sur la référence this

La référence utilisée pour synchroniser un bloc peut être `this`

```
synchronized(this) {
    // ici les instructions protégées
}
```

Tous les `threads` qui utilisent une méthode de l'objet bloqueront sur les blocs synchronisés par `this` si un autre `thread` est déjà dans un de ces blocs là

Introduction
Solution avec plusieurs threads
Threads JAVA
Accès concurrentiel

Blocs synchronisés : sur la référence this

```
public class MonThread implements Runnable {
private Ressource maRessource;

public MonThread(Ressource maRessource) {
  this.maRessource = maRessource;
}

@Override
public void run() {
  try {
    while (true) {
     maRessource.changer();
     assert maRessource.getVal() >= 0;
   }
 } catch (InterruptedException e) {
    e.printStackTrace();
 }
}
}
```

```
public class Ressource {
private int val = 15;

public Ressource(final int val) {
  this.val = val;
}

public int getVal() {
  return val;
}

public   void  changer()
            throws InterruptedException {

   if (val >= 10) {
     Thread.sleep( 5000);
     val -= 10;
   }
   else {
     Thread.sleep( 5000);
     val += 20;
   }

}
}
```

Blocs synchronisés : sur la référence this

```
public class MonThread implements Runnable {
private Ressource maRessource;

public MonThread(Ressource maRessource) {
  this.maRessource = maRessource;
}

@Override
public void run() {
  try {
    while (true) {
     maRessource.changer();
     assert maRessource.getVal() >= 0;
   }
 } catch (InterruptedException e) {
    e.printStackTrace();
 }
}
}
```

```
public class Ressource {
private int val = 15;

public Ressource(final int val) {
  this.val = val;
}

public int getVal() {
  return val;
}

public   void  changer()
          throws InterruptedException {
 synchronized(this) {
   if (val >= 10) {
     Thread.sleep( 5000);
     val -= 10;
   }
   else {
     Thread.sleep( 5000);
     val += 20;
   }
 }
}
}
```


Blocs synchronisés : sur une méthode d'instance

Sur l'exemple précédent toutes les instructions de la méthode sont dans le bloc synchronisé sur `this`. Il y a alors une autre syntaxe pour signifier la même chose :

```
public synchronized void maMethode()

        // toutes les instructions de la
        // méthode sont protégées

}
```

Sections synchronisées : sur une méthode d'instance

```
public class MonThread implements Runnable {
private Ressource maRessource;

public MonThread(Ressource maRessource) {
  this.maRessource = maRessource;
}

@Override
public void run() {
  try {
    while (true) {
     maRessource.changer();
     assert maRessource.getVal() >= 0;
   }
 } catch (InterruptedException e) {
    e.printStackTrace();
 }
}
}
```

```
public class Ressource {
private int val = 15;

public Ressource(final int val) {
 this.val = val;
}

public int getVal() {
 return val;
}

public synchronized void changer()
            throws InterruptedException {
 if (val >= 10) {
  Thread.sleep( 5000);
  val -= 10;
 }
 else {
  Thread.sleep( 5000);
  val += 20;
  }
}
}
```


Sections synchronisées : sur une méthode de classe

Il est aussi possible de synchroniser une méthode de classe. Dans ce cas, la référence `this` n'existe pas
La synchronisation se fait sur la référence MaRessource.class :

```
class MaRessource {
 (...)
 static public synchronized void maMethode()
      // toutes les instructions de la
      // méthode sont protégées
 }
}
```

Dans ce cas, si cette méthode est exécutée par un `thread`, tous les autres `thread` tenteront d'appeler une méthode de classe synchronisée de cette classe seront bloqués.

Synchronisation des threads

Il est parfois utile qu'un thread soit bloqué en attente d'un autre :

- attendre qu'un objet soit construit par un autre thread pour l'utiliser ;
- attendre qu'une ressource soit produite par un autre thread pour la consommer ;
- attendre la fin d'un autre thread.

Il existe des méthodes pour faire ces synchronisations : *wait()*, *notify()*, *notifyAll()*, *join()*.

Synchronisation des threads

Les méthodes *wait()*, *notify()* et *notifyAll()* sont héritées de la classe `Object` et sont donc implémentées pour toutes les classes. Elles ne peuvent être appelées que sur un objet d'un bloc synchronisé.

```
public void methode1() {
  synchronized(maRessource) { // obligatoire pour
      (...)                   // l'usage de wait()
      maRessource.wait();
      (...)
  }
}
public void methode2() {
  synchronized(maRessource) { // obligatoire pour
      (...)                   // l'usage de notify()
      maRessource.notify();
      (...)
  }
}
```


Note1 : Version avec un synchronized sur une méthode d'instance :

```
public synchronized void methode1() {
      (...)
      wait();
      (...)
}
public synchronized void methode2() {
      (...)
      notify();
      (...)
}
```

Synchronisation des threads

- *wait()* est bloquant jusqu'à ce qu'un autre thread appelle *notify()* ou *notifyAll()* sur le même objet ;
- l'attente de *wait()* est équivalent à une sortie du bloc synchronisé ;
- le réveil de *wait()* est équivalent à une entrée dans le bloc synchronisé ;
- si plusieurs threads bloquent sur un *wait()* sur le même objet, un appel de *notify()* ne réveillera qu'un seul d'entre-eux ;
- *notifyAll()* reveillent tous les *wait()* associés ;
- *wait(long timeout)* permet de fixer un délais maximal d'attente (en milliseconde).

Introduction
Solution avec plusieurs threads
Threads JAVA
Accès concurrentiel

Synchronisation des threads

```
private Queue<Integer> maFile;

public void produire(int val) {
   synchronized (maFile) {
      maFile.add(val);
      maFile.notify();
   }
}
public Integer consommer() {
   synchronized (maFile) {
      while (maFile.size() == 0) {
         try {
            maFile.wait();
         } catch (InterruptedException e) {}
      }
      Integer res = maFile.poll();
      return res;
   }
}
```

Éric LalletEric.Lallet@telecom-sudparis.eu CSC4509 — Utilisation des Threads

Synchronisation des threads

- *join()* est une méthode de la classe Thread. Elle permet de bloquer en attendant la fin du thread ;
- Il est possible de donner un délais maximal d'attente avec la méthode *join(long timeout)*. Le timeout est exprimé en milliseconde ;
- Notez que pour les threads `Callable`, la méthode *get()* de l'interface `Future` fonctionne sur le même principe.


```
public class ExempleJoin {
   private static final int MAX = 5;

   public ExempleJoin() throws InterruptedException {
     Thread [] taches = new Thread[MAX];
     for (int ind = 0; ind < MAX; ind++) {
       final int delais = ind;
       taches[ind] = new Thread(() -> exemple(delais));
       taches[ind].start();
     }
     for (int ind = 0; ind < MAX; ind++) {
       taches[ind].join();
     }
     System.out.println("Tous les threads sont terminés");
   }
   public void exemple(int delais) {
     System.out.println(delais + " débute.");
     try {
       Thread.sleep(delais*1000);
     } catch (InterruptedException e) { }
     System.out.println(delais + " termine.");
   }
}
```


Note1 : Dans l'expression lambda qui implémente la méthode *run()* du thread, la méthode *exemple()* reçoit un paramètre. Il n'est pas possible de passer la variable de boucle *ind* pour fixer la valeur de ce paramètre. Il est possible d'utiliser une variable du contexte englobant dans une expression lambda **à condition que cette variable soit finale ou effectivement finale**. Ici la variable de boucle *ind* est incrémentée à chaque tour. Elle ne répond pas à la condition. Pour résoudre ce problème, il suffit de recréer à chaque tour une nouvelle variable (ici la variable *delais*) qui recopie cette valeur.