CSC4509 — communication sous TCP en mode asynchrone

Éric Lallet
Eric.Lallet@telecom-sudparis.eu

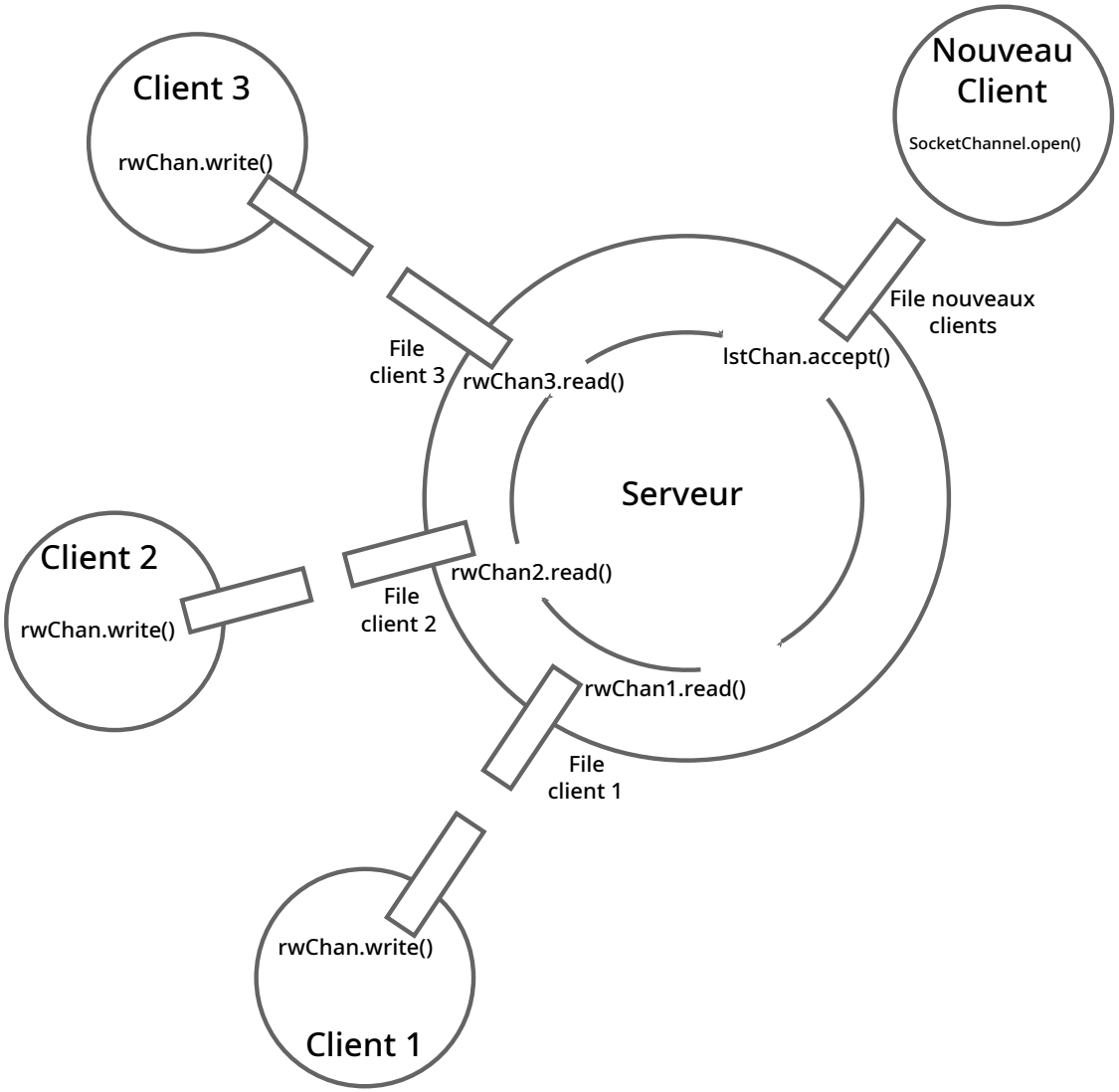
Télécom SudParis

30 avril 2025

Cette Présentation contient trois sections :

1. L'exposition du problème que pose un serveur multiclient
2. Un premier pas vers la solution avec le mode non bloquant
3. La solution complète avec la classe `Selector`

Serveur multiclient



Serveur multiclient

Problèmes de cette architecture :

- Le serveur peut bloquer en lecture sur la file d'attente d'un client
- Le serveur peut bloquer en `accept` sur la file d'attente des nouveaux clients

Les solutions :

- Créer un nouveau `Thread` pour chaque nouveau client. Chaque `Thread` bloque sur sa lecture, sans bloquer les autres. Cette solution sera présentée au prochain cours
- Rendre les *read()* et les *accept()* non-bloquants

Mode non-bloquant

JAVA NIO permet de rendre non bloquantes toutes les actions normalement bloquantes (*read()*, *write()*, *accept()*...)

```
import java.net.StandardSocketOptions;
import java.nio.channels.SocketChannel;
import java.nio.channels.ServerSocketChannel;
(...)
ServerSocketChannel lstChan;
SocketChannel rwChan;

lstChan = ServerSocketChannel.open();
lstChan.setOption(StandardSocketOptions.SO_REUSEADDR, true);
lstChan.bind(new InetSocketAddress(5000);
lstChn.configureBlocking(false); // les accept() ne bloquent plus

rwChan = lstChan.accept();
if (rwChan != null) { // null si aucun client pour l'accept()
      rwChan.configureBlocking(false);
            // les read() ou write() ne bloquent plus
      (...)
}
```

Mode non-bloquant

Problème de cette première étape :

- Même sans aucune activité des clients (ou même sans le moindre client connecté) le serveur tourne en boucle pour vérifier l'arrivée de nouveaux messages ou de nouveaux clients

Solution :

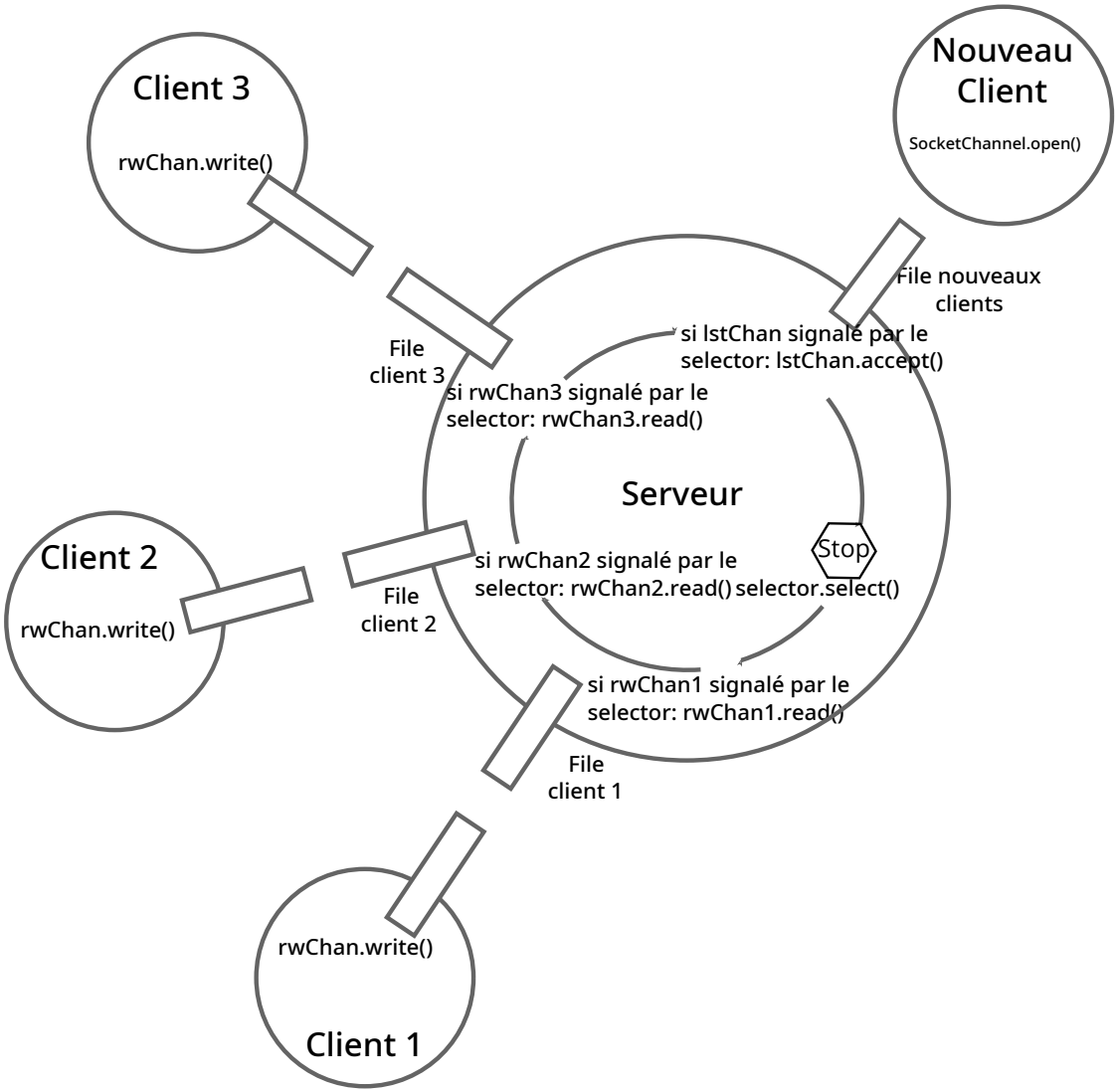
- Utiliser la classe `Selector` de JAVA NIO

La classe `Selector`

Principe d'utilisation :

- Préparer un *selector* de la classe `Selector`
- Enregistrer dans un ensemble tous les canaux rendus non-bloquants (*lstChan* et les *rwChan*).
- Bloquer sur l'appel *select()* du *selector* jusqu'à ce qu'un évènement arrive sur un des canaux :
  - l'arrivée d'un message à lire ;
  - la déconnexion d'un client ;
  - la connexion d'un nouveau client.
- Parcourir l'ensemble pour connaître la liste des canaux sur lesquels une activité s'est produite

Serveur multiclient avec Selector



La classe `Selector` : les grandes étapes (partie 1/2)

- Créer le `Selector` par une méthode de classe :
  Selector selector = Selector.open() ;
- Enregistrer le canal par une méthode d'instance des canaux.
  Le résultat est une clef qui est associée au canal enregistré :
  SelectionKey clef =
  rwChan.register(selector, SelectionKey.OP_READ) ;
  ou bien *lstChan.register(selector, SelectionKey.OP_ACCEPT) ;*
- Bloquer en attendant un évènement sur un des canaux enregistrés : selector.select() ;
- Récupérer et parcourir l'ensemble des clefs :
  - *Set<SelectionKey> lesClefs = selector.selectedKeys() ;*
  - *for (SelectionKey clef : lesClefs)*

La classe `Selector` : les grandes étapes (partie 2/2)

Dans la boucle :

- Tester si c'est la clef de l'*accept()* :
  - Tester : *if (clef.isAcceptable())*
  - Accepter : *rwChan = lstChan.accept() ;*
- Tester si c'est une clef pour la lecture :
  - Tester : *if (clef.isReadable())*
  - Retrouver le canal associé : *rwChan = (SocketChannel) clef.channel() ;*
  - Lire : rwChan.read(buff).
  - Clore : si le read retourne *-1*, clore le client :
    - Fermer le canal : *rwChan.close() ;*
    - Nettoyer l'ensemble du `Selector` : *clef.cancel() ;*

Après la boucle :

- Nettoyer l'ensemble des clefs actives, pour repartir avec un ensemble vide : *lesClefs.clear()*

La classe Selector : exemple d'usage (lien Selector/Canal)

```
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
(...)
Selector selector;
ServerSocketChannel lstChan;
SocketChannel rwChan;

selector = Selector.open();
lstChan = ServerSocketChannel.open();
lstChan.setOption(StandardSocketOptions.SO_REUSEADDR, true);
lstChan.bind(new InetSocketAddress(5000));
lstChan.configureBlocking(false);//les accept() sont non-bloquants
lstChan.register(selector, SelectionKey.OP_ACCEPT);
                              // canal lstChan dans l'ensemble
```

La classe Selector : exemple d'usage (les *accept()*)

```
while (true) {
  selector.select(); // appel bloquant
  Set<SelectionKey> lesClefs = selector.selectedKeys();
                                    // ensemble des clefs inscrites
  for (SelectionKey clef: lesClefs) { // parcours des clefs
    if (clef.isAcceptable()) { // accept: sur le canal lstChan
      SelectionKey nvlClef;
      SocketChannel rwChan = lstChannel.accept();
      if (rwChan != null) { // nouveau client
        rwChan.configureBlocking(false);//lecture non bloquante
        nvlClef = rwChan.register(selector, SelectionKey.OP_READ);
                                    // canal rwChan dans l'ensemble
      }
    } else {
      // voir la partie 3

    } // fin du else
  } // fin du for
  lesClefs.clear();
} // fin du while
```

La classe `Selector` : exemple d'usage (les *read()*)

```
while (true) {
  selector.select(); // appel bloquant
  Set<SelectionKey> lesClefs = selector.selectedKeys();
                                    // ensemble des clefs inscrites
  for (SelectionKey clef: lesClefs) { // parcours des clefs
    if (clef.isAcceptable()) { // accept: sur le canal lstChan
      // voir la partie 2
    } else {
      SocketChannel rwChan = (SocketChannel) clef.channel();
                          // on retrouve le canal de cette clef
      int nbRcv = rwChan.read(buff); // lecture du message
      if (nbRcv == -1) { // connexion perdue
          rwChan.close();
          clef.cancel();//retrait de l'ensemble du Selector
                          // de la clef et du canal
      }
    } // fin du else
  } // fin du for
  lesClefs.clear();
} // fin du while
```