Commentaires sur le TP noté 2025

1 Généralités

La plupart des copies sont bonnes, et même très bonnes mais il y a quelques erreurs que l'on retrouve dans beaucoup de copies. Voici quelques commentaires à leur sujet pour mieux comprendre les commentaires individuels faits dans votre fichier de retour.

2 Usage de la classe `FullDuplexMsgWorker`

Nous avons écrit la classe `FullDuplexMsgWorker` ensemble (pour les méthodes principales), et nous l'avons ensuite utilisée pour tous le TPs. Son usage est donc considéré comme acquis. Pourtant certains d'entre-vous ne savent pas l'utiliser, surtout pour la partie qui type les objets échangés.

La méthode `FullDuplexMsgWorker::sendMsg()` reçoit en premier paramètre un entier qui indique le type de l'objet sérialisé. Cet entier fonctionne avec l'énumération `MessType` qui liste tous les types qui peuvent être échangés. Donc ce premier paramètre doit être l'`ordinal` de l'énumérateur correspondant.

L'énumérateur pour échanger un `Token` est `TOKENTYPE`, donc pour envoyer un `Token` avec cette classe il faut utiliser `worker.sendMsg(MessType.TOKENTYPE.ordinal(),token)`.

Sur certaines copies on trouve des valeurs entières plus ou moins arbitraires :
- parfois l'entier est bien la valeur correspondant à l'`ordinal` de l'énumérateur, du moins tant qu'on insère pas un nouveau type dans l'énumération. Ce qui rend donc le programme fragile face aux éventuels ajouts de nouveaux types de messages ;
- parfois l'entier est remplacé par un numéro de port. Ce qui ressemble beaucoup à l'ajout d'un IDE ou d'une IA qui essaie de compléter avec les variables qu'ils ont sous la main, mais qui n'a aucune logique pour ce cas ;
- et d'autre fois l'entier semble être choisi au hasard.

3 La connexion après le passage de mode asynchrone

Sur les transparents du cours, et dans les corrections de TPs, il n'y a jamais aucun usage de l'appel `SocketChannel::connect()`. Avec `JavaNIO`, il n'y a généralement pas besoin de l'utiliser.

Certes, dans le cadre de la programmation réseau en `C`, la logique d'une connexion client consiste à faire un appel à `socket()` suivi d'un appel à `connect()`. Et effectivement ce `connect()` est nécessaire, puisque l'appel à `socket()` ne caractérise pas un client `TCP`. Il faut attendre soit un `connect()` pour obtenir un client, soit un `listen()` pour obtenir un serveur.

Mais avec `JavaNIO` il y a deux `open()` différents suivant que l'on veuille construire un client `TCP` (on utilise alors `SocketChannel::open()`), ou un serveur `TCP` (on utilise alors `ServerSocketChannel::open()`). Et tant qu'on ne cherche pas à utiliser une option spécifique, on peut faire toute la connexion du client uniquement avec `SocketChannel::open()`), et toute la mise en écoute du serveur avec `ServerSocketChannel::open()`).

Cependant, pour le serveur, on souhaite généralement changé une option, et on procède finalement en plusieurs étapes :

```
listenChannel = ServerSocketChannel.open(); // sans adresse, car on veut utiliser une option
listenChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true); // option
listenChannel.bind(new InetSocketAddress(port));  // bind fait après le changement de l'option
```

Par contre, pour le client, les options par défaut conviennent la plupart du temps, donc l'appel à `SocketChannel::open()`) suffit :

```
// appel d'open() avec l'adresse de connexion, donc le connect est réalisé dès l'open
SocketChannel rwChan = SocketChannel.open(addr);
```

Vous pouvez compliquer le code en utilisant deux instructions là une seule suffit :

```
SocketChannel rwChan = SocketChannel.open(); // open sans connect
rwChan.connect(addr) // puis le connect
```

Le code ci dessus a été accepté sur vos copies. Par contre, il ne faut surtout pas passer le `SocketChannel` en asynchrone avant le `connect()`. En effet, une connexion `TCP` n'est pas instantanée, et la méthode `SocketChannel::connect()` est bloquante. Si on passe le canal en asynchrone, l'appel ne bloque pas, et le programme continue alors que la connexion n'est pas encore réalisée. Si on utilise ce canal en lecture ou en écriture avant que cette connexion soit établie, le programme lève une exception.

Donc, soit vous faites la connexion dès l'`open()`, et vous pouvez passer le canal en asynchrone après celui-ci, soit vous faites la connexion en deux méthodes (`open()` + `connect()`), et vous devez passer le canal en asynchrone après le `connect()`.

4 Gestion de la section critique

4.1 Introduction

Cette question n'a pas été traitée par tout le monde, mais on retrouve deux types d'erreurs sur les certaines copies qui l'ont abordée :

— erreur dans le choix de la ressource à synchroniser ;
— erreur sur le début et la fin des sections critiques.

4.2 Choix de la ressource à synchroniser

Pour gérer les exclusions mutuelles des sections critiques, le sujet demande d'utiliser les blocs synchronisés. L'instruction `synchronized` demande d'indiquer une référence sur la ressource commune dont il faut protéger l'accès dans la section critique. Le sujet vous annonce clairement que la ressource problématique est l'attribut de classe `workerSuivant`. Il faut donc protéger les blocs synchronisés par les instructions :

```
synchronized (workerSuivant) {
      // section critique
}
```

Sur certaines copies, les blocs sont synchronisés sur la référence `this` (donc la référence qui désigne le `Node` traité). Cela va effectivement protéger les sections critiques, mais de manière trop générale. Si, par exemple, deux `threads` partageaient aussi la collection `connexions` (autre attribut de la classe), et que l'on protégeait cette ressource de même manière, une section critique utilisant une ressource (`workerSuivant` ou `connexions`), bloquerait les sections critiques de l'autre ressource sans aucune nécessité.

Même si cela ne concerne pas ce programme, je profite de cette section pour rappeler un point important abordé dans le module d'Introduction aux systèmes d'exploitation (CSC3102), lors du cours présentant les sections critiques (transparents 30 à 36,sur l'inter-blocage) : si, dans une section critique, vous avez besoin de plusieurs ressources partagées, vous pouvez imbriquer les bloc synchronisés, mais il est important de toujours les imbriquer dans le même ordre pour éviter des inter-blocages. Par conséquence, on doit parfois bloquer une ressource bien en avance, si son ordre de réservation l'oblige.

Voici la syntaxe pour imbriquer les blocs synchronisés :

```
synchronized (ressource1) {
      // usages de la ressource1 protégés
      synchronized (ressource2) {
            // usages des ressource1 et ressource2 protégés
      }
      // usages de la ressource1 protégés
}
```

4.3 Début et fin des sections critiques

Le choix des instructions à placer dans les sections critiques est important. Si on ne couvre pas toute la section critique, les données peuvent devenir incohérentes, si on couvre des instructions qui n'ont pas besoin d'y être, le programme peut perdre en performance en étant bloqué quand il ne devrait pas l'être.

Certaines erreurs trouvées sur les copies illustrent tous ces cas :

— Les `threads` peuvent appeler deux méthodes qui utilisent la ressource partagées (`workerSuivant`) : `traiterJeton()` et `changerSuivant`. Il faut donc identifier une section critique dans chacune de ces méthodes. C'est explicitement dit dans le sujet. Pourtant certains n'ont placé une section critique que dans l'une et pas dans l'autre, ou encore l'ont placée dans d'autres méthodes (comme `changerPrecedent`, qui n'utilise pourtant pas cette ressource) ;

— Certains ont été trop larges dans les instructions couvertes par les sections critiques. Ils ont englobé la simulation de tâche présente dans `traiterJeton()`. On perd alors tout l'intérêt d'avoir placé cette méthode dans un nouveau `thread`. Le but est de pouvoir chaîner un nouveau `Node` pendant que la tâche se réalise. Si on place cette tâche dans la section critique, le chaînage est bloqué, et on se retrouve dans le cas du programme sans `thread`.

— Certains ont été trop courts dans les instructions couvertes par les sections critiques. Dans la méthode `changerSuivant()`, la section critique commence du premier usage (`workerSuivant.sendMsg()`) jusqu'au dernier usage (`this.workerSuivant = nouveauSuivant;`). Certaines copies ne placent pas cette dernière instruction dans leur section critique. Si la méthode `traiterJeton()` reprend la main entre le `close()` (placé juste avant cette instruction), et cette instruction, elle va utiliser le `workerSuivant` fermé pour envoyer le jeton, et cela va provoquer une exception et effondrer l'anneau.