Mise au propre des TDs d'application réseaux

Yann Miguel

4 mars 2020

Table des matières

1 TD1

1.1 Exercice 1

1.1.1 Question 1

L'intérêt du modèle en couche est la facilité d'abstractin et l'isolation des couches.

1.1.2 Question 2

Le modèle de couche TCP/IP contient 4 couches:

1. Application
2. Transport
3. Internet
4. NetworkAccess

1.1.3 Question 3

Les deux modèles sont des modèles à couches, mais le TCP/IP en contient moins.
Couches du modèle OSI:

1. Application
2. Présentation
3. Session
4. Transport
5. Network
6. DataLink
7. Physical

1.1.4 Question 4

La couche permettant de faire abstraction de la manièrte dont les différents réseaux locaux sont interconnectés est la couche Network pour le modèle OSI et la couche Internet pour le modèle TCP/IP.

1.2 Exercice 2

1.2.1 Question 1

Protocoles de la couche DataLink:
- WiFi
- Ethernet
- Bluetooth

1.2.2 Question 2

Le rôle de la couche DataLink est d'assurer la communication entre plusieurs machines connectées sur un même réseau local.

1.3 Exercice 3

1.3.1 Question 1

Le protocle Ip appartient à la couche Internet.

1.3.2 Question 2

Afin de connaître la destination d'un pacquet, on regarde l'adresse dans la pacquet.

1.3.3 Question 3

La principale amélioration de L'IPv6 sur l'IPv4 est la quantité d'adresses disponibles.

1.3.4 Question 4

On a choisi des adresses de 128 bits pour deux choses:

1. La quantité d'adresses disponibles.
2. La généralisation des adresses à partir de l'adresse Mac.

1.4 Exercice 4

$$\frac{vc}{d} < t \;\Rightarrow\; \frac{36}{d} < 1,2 \;\Rightarrow\; d < \frac{5 \times 16000}{2,4} \;\Rightarrow\; d < \frac{400000}{12} \;\Rightarrow\; d < 33333,333\ m$$

Cette solution prends en compte le retour du disque vers le point de départ.

Si on compte un allé simple, on fait $33,333 \times 2 = 66,67 km,$ ou bien 66666,67 m.

2 TD2

2.1 Exercice 1

2.1.1 Question 1

Afin de représenter l'adresse de ce serveur, on utilise la classe InetAddress de Java.

2.1.2 Question 2

```
1  public static InetAddress getInetAddressByName(String hostname){
2    return InetAddress.getByName(hostname);
3  }
```

Listing 1 – code pour récupérer une adresse IP depuis un String

2.2 Exercice 2

2.2.1 Question 1

Les classes importantes pour UDP sont DatagramSocket et DatagramPacket.

2.2.2 Question 2

Afin de créer un DatagramPacket, un client à besoin de:
- Le buffer.
- La taille du buffer.
- L'adresse du serveur.
- Le port d'écriture.

2.2.3 Question 3

```
1  public static void receiveTextFromUdp(){
2    DatagramSocket s = new DatagramSocket(ssss);
3    byte msg[]= new byte[10];
4    DatagramPacket n = new DatagramPacket(msg, 40);
5    s.receive(in);
6    System.out.println(new String(msg));
7    s.close();
8  }
```

Listing 2 – code pour recevoir du texte via protocole UDP

2.3 Exercice 3

2.3.1 Question 1

La différence entre les transmission UDP et TCP est que UDP envoie un flux unidirectionnel et plus rapides, mais a des pertes, alors que le TCP est plus lent mais n'a aucune perte.

2.3.2 Question 2

Les données UDP se transmettent plus vite.

2.3.3 Question 3

```
Socket s = new Socket(adresse, port);
ObjectOutputStream output = new ObjectOutputStream(s.
  getOutputStream);
ObjectInputStream input = new ObjectInputStream(s.getInputStream)
  ;
input.read();
output.write();
s.close();
```

Listing 3 – code pour récupérer des données d'un socket

2.3.4 Question 4

```
public static void receiveMessageTCP(){
  Socket s = new Socket(myServer, ssss);
  InputStream in = s.getInputStream();
  BufferedReader b = new BufferedReader(new InputStreamReader(in)
  );
  System.out.println(b.readline());
  s.close();
}
```

Listing 4 – code pour recevoir du texte via protocole TCP

3 TD3

3.0.1 Question 1

L'unité consomée n'est pas nécessaire pour le SYN, onn le fait plus par habitude, mais elle l'est pour le FIN.

3.0.2 Question 2

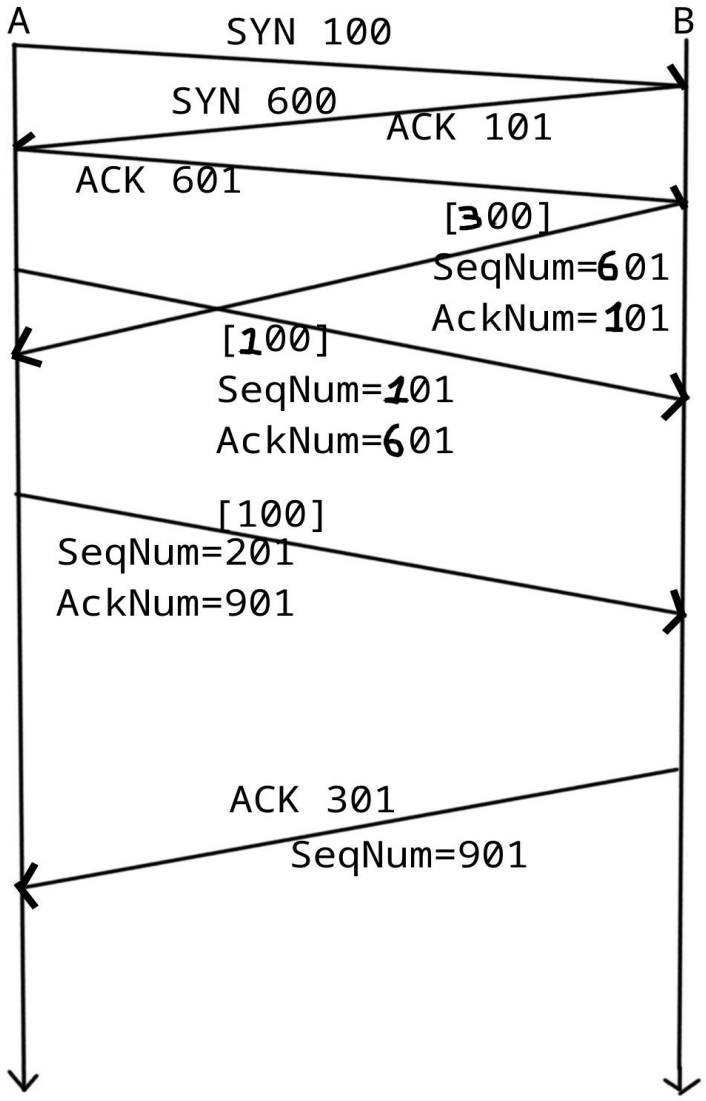

FIGURE 1 – Chronogramme d'échange entre A et B

Si le premier segment de données de A se perds, alors l'AckNum de B contiendra 100 de moins que le SeqNum de A, ce qui pousse B à envoyer un ACK pour indiquer qu'il manque le packet à partir de 101.

3.0.3 Question 3

Résumé du chronogramme:
- Envoie d'abord de 'a', attente d'ACK/tampon plein.
- Envoie "bcde", attende d'ACK/tampon plein.
- Envoie "fghi".

Si la connection est de type "TelNet full duplex", les observations sont les mêmes.

3.0.4 Question 5

Elle va grandir exponentiellement jusqu'à la moitié de la taille de lafenêtre avant le crash, 9 Ko dans ce cas.
Elle sera donc de la taille du seuil au bout de 4 transmission acquitées, si tout se passe bien.
Après le seuil, au lieu de faire une croissance exponentielle, on incrémente seulement la quantité d'octets qui passent.

4 TD 4

4.1 Exercice 1

4.1.1 Question 1

Afin d'utiliser les threads en Java, il faut créer une classe que étend la classe Thread. La méthode principale pour un thread est start.

4.1.2 Question 2

```java
public class SocketHandler implements Runnable{
  private Socket socket;
  private List<String> messages;

  public SocketHandler(Socket socket){
    this.socket = socket;
    messages = new ArrayList<String>();
  }

  // TODO: Ajouter la gestion d'exceptions
  @Override
  void run(){
    BufferedReader input = new BufferedReader(new
  InputStreamReader(this.socket.getInputStream());
    String line;
    while((line = input.readLine()) != null) this.messages.add(
  line);

    socket.close();
  }

  public Socket getSocket(){
    return this.socket;
  }

  public List<String> getMessages(){
    return this.messages;
  }

  public void setSocket(Socket socket){
    this.socket = socket;
  }
}
```

Listing 5 – code de la classe SocketHandler

4.2 Exercice 2

4.2.1 Question 1

On peux lui faire créer un thread par client TCP lu.

```java
ServerSocket server = new ServerSocket(1234);
while(true)
  new Thread(new SocketHandler(server.accept())).start();
```

Listing 6 – Serveur TCP multi-thread en trois lignes

4.2.2 Question 2

Comportement eronné(duplication potentielles) et imprévisible.

4.2.3 Question 3

La structure ne sera modifiée que par un seul à la fois via un système de sémaphores et de section critiques, si la structure est sérialisée.

```
class Data{
  public int data=0;
}

class Job implements Runnable{
  Data d;

  Job(Data d){
    this.data = data;
  }

  void run(){
    int newData = d.data+1;
    d.data = newData;
  }
}

public static void main(String[] args){
  Data d = new Data();

  Thread A = new Thread(new Job(d).start());
  Thread B = new Thread(new Job(d).start());

  System.out.println(d.data);
}
```

Listing 7 – Exemple d'écriture sur la même structure de données par deux threads

La donnée affichée sera soit 1 soit 2.

4.2.4 Question 4

Il n'y a aucun moyen.

4.3 Exercice 3

4.3.1 Question 1

Pour la communication entre un socket server et un socket client, il faut utiliser les classes ServerSocketChannel et SocketChannel de java.nio.channels.

4.3.2 Question 2

Utiliser Java NIO pour la communication entre sockets a comme avantages:
- non-bloquant
- pas de threads
- plus optimisé

Il est globalement très efficace.

4.3.3 Question 3

On utilise un Selector pour les communications entre sockets car il trouve ce qu'il faut faire dans l'instant. C'est-à-dire, il dit quel channel est prêt à travailler, et comment.

```
selector.select();
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> iterator = selectedKeys.iterator();
while(iterator.hasNext()){
  SelectionKey key = iterator.next();
}
```

Listing 8 – Exemple selector

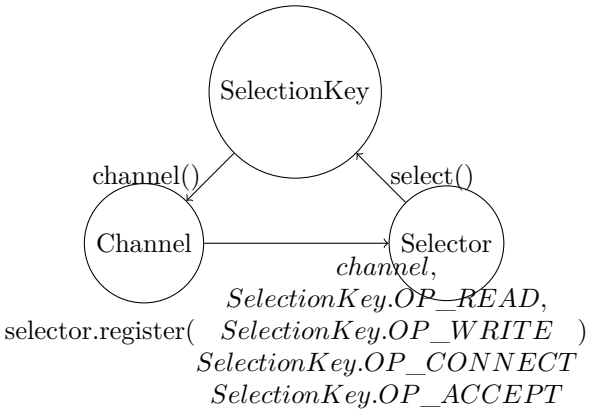

FIGURE 2 – Schéma classes

Les clés en majuscules sont à choisir en fonction de ce dont on a besoin.

4.3.4 Question 4

```
Selector selector = Selector.open();
ServerSocketChannel server = ServerSocketChannel().open;
server.configureBlocking(false);
selector.register(server, SelectionKey.OP_ACCEPT);
// Exemple selector, code ensuite dans la boucle while
if(key.isAcceptable()){
  SocketChannel client = server.accept();
  selector.register(client, (SelectionKey.OP_WRITE|SelectionKey.
  OP_READ));
  client.configureBlocking(false);
}
if(key.isReadable()){
  SocketChannel client = key.channel();
  // lecture
}
```

Listing 9 – Serveur Mono-Thread qui gère jusqu'à 10 clients en simultanés

5 TD 5

5.1 Exercice 1

5.1.1 Question 1

Afin d'assurer la communication non-bloquante pour la lecture ou l'écriture dans un SocketChannel en Java NIO il faut utiliser la méthode channel.configureBlocking(false).

Les objets de type SelectableChannels ont comme propriétés:

- Avoir une configuration bloquante ou non bloquante.
- On peux l'enregistrer dans un Selector.

5.1.2 Question 2

2 octets seront transférés vers socketChannel.

Les nouvelles valeurs seront:

- *Position* = 3
- *Limit* = 3
- *Capacity* = 8

5.1.3 Question 3

Les différences entre Buffer.clear() et Buffer.flip() sont:

- Buffer.clear() remet la position à 0.
- Buffer.flip() remet la position à gauche des données qu'on a écrit, et la limite à droite des données afin de ne pouvoir travailler que sur des données qu'on à déjà écrites.

5.2 Exercice 2

5.2.1 Question 1

La classe Selector est un classe qui donne des informations sur des Channel sur lesquels on veut travailler, via les SelectionKey.

La classe SelectionKey est une interface entre le Selector et les Channel, qui permet de au Selector de récupérer des informations sur le Channel.

La question est mal posée, on demande quel quantité de types plutôt qu'une quantité de clés, et on peux en associer 4 à un Selector.

5.2.2 Question 2

On peux enregistrer un SocketChannel à un Selector pour attendre les événements de type écriture en utilisant la méthode socketChannel.register(selector, SelectionKey.OP_WRITE).

On peux enregistrer un ServerSocketChannel à un Selector pour attendre les demandes de connection de clients avec la méthode serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT).

5.2.3 Question 3

```
// Java NIO cree des objets avec open() au lieu de new.
Selector selector = Selector.open();
// Il ne faut pas oublier de mettre les channels non bloquants.
ssc.configureBlocking(false);
csc.configureBlocking(false);
ssc.register(selector, SelectionKey.OP_ACCEPT);
csc.register(selector, SelectionKey.OP_WRITE);
while(true){
  selector.select();
  Iterator<Set<SelectionKey> > iterator = selector.selectedKey().
  iterator();
  while(iterator.hasNext()){
    SelectionKey key = iterator.next();
    if(key.isAcceptable()){
      scs.accept();
    }
    if(key.isWriteable()){
      // ecrire
    }
    iterator.remove();
  }
}
```

Listing 10 – Selector qui attend des demandes de connection à un serveur et des demandes d'écriture d'un client

5.3 Exercice 3

5.3.1 Question 1

Il est possible d'utiliser un seul Selector partagé par plusieurs threads.

La méthode select() est protégée, alors que la gestion des clés non. Il faut donc toujours vérifier les clés lors du multi-threading.

5.3.2 Question 2

La méthode Selector.wakeup() sert à sort un thread de la sélection lorsqu'il est bloqué.

On a besoin de l'utiliser quand on a besoin de modifier le selector.

5.3.3 Question 3

1. Communication d'un channel vers plusieurs buffers:
   - SocketChannel.write(ByteBuffer[] srcs)
   - ScatteringByteChannel.write(ByteBuffer[] srcs)(interface)
2. Communication de plusieurs buffers vers un channel:
   - SocketChannel.read(ByteBuffer[] dsts)
   - GatheringByteChannel.read(ByteBuffer[] dsts)(interface)

Les deux interfaces implémentent les méthodes dans le SocketChannel.