***TIC – Filière Informatique***

**Jigé Pont**

**Nicolas Fuchs**

**Programmation avancée Java**

**TP01 - Sockets**



# Configuration

## P0

Serveur : Windows 10 Education  
 Java Jre1.8.0\_71

Client : Windows 10 Famille  
 JavaSE-1.8

# Etablissement et fermeture de la connexion

## P1

Client : A l'instanciation de la classe Socket avec comme arguments l'adresse Ipv4 du serveur et le port d'écoute du serveur. Une trame est envoyée du client au serveur avec un flag SYN (demande de synchronisation).

Serveur : Le serveur a appelé la fonction blocante accept() et à la réception de la trame (SYN) du client, cette fonction envoie une trame SYN/ACK au client.

Client : A la réception du SYN/ACK (synchronisation et acknowledge), le client renvoie un ACK et termine la construction du socket.

Serveur : A la réception du ACK du client, la fonction accept() retourne une instance de la classe Socket.

Fichier Wireshark : TP01\_P1.pcapng (3-way handshake)

## P2

Non, le système cherche un port libre. Cependant, on peut paramétrer le port local à utiliser sur la partie client en utilisant un constructeur avec quatre arguments. On peut également utiliser le constructeur sans argument suivi d'un appel à la méthode bind().

## P3

Le type d'exception lancée lorsque la connexion TCP est interrompue est java.io.EOFException. A l'appel de la méthode close() de l'objet socket, une trame FIN/ACK est envoyé du client au serveur et celui-ci répond par un ACK.

Fichier Wireshark : TP01\_P3\_Client\_close.pcapng

A noter que si la fermeture de la connexion ne se fait pas proprement (fermeture de socket et de objectOutputStream), la trame envoyée contient les flags RST (reset) et ACK.

Fichier Wireshark : TP01\_P3\_Client\_reset.pcapng

## P4

L'effet est semblable à celui observé lorsque le client provoque la fermeture de la connexion. Le seul changement intervient dans le sens des messages (FIN/ACK depuis le serveur puis ACK depuis le client).

Fichier Wireshark : TP01\_P4\_Server\_close.pcapng

# Envoi des messages

## P5

Les trames 10, 11 et 12 établissent la connexion TCP entre le client et le serveur (SYN, SYN/ACK et ACK). Les trames 13 et 14 sont deux trames PSH/ACK et ACK créées par l'instanciation de l'objet objectOutputStream.

Les trames 47 à 50 sont le résultat de l'envoi et de la réception du message en deux trames TCP (PSH/ACK et ACK).

Les trames 234 à 237 sont le résultat de la fermeture de la connexion TCP en deux fois deux trames (fermeture du client vers le serveur puis du serveur vers le client).

Fichier Wireshark : TP01\_P5.pcapng

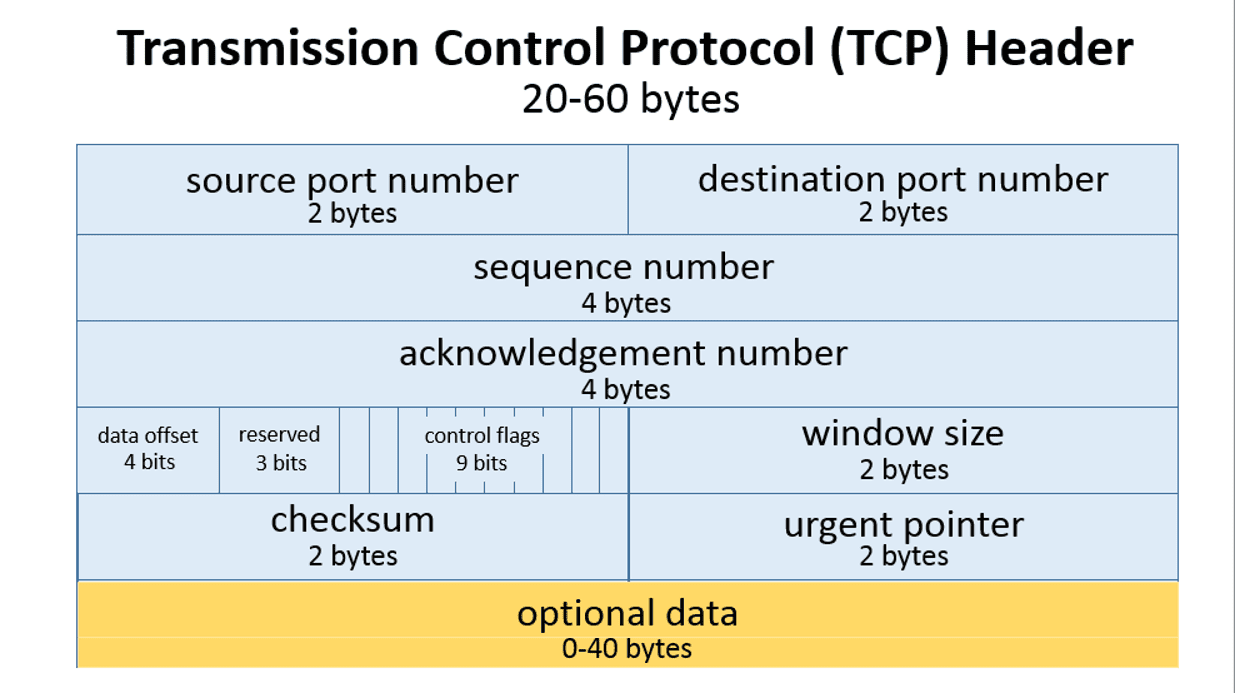
160.98.127.74:11609  
Client

160.98.117.114:7676  
Server

1. SIN
2. SIN/ACK
3. ACK
4. PSH/ACK
5. ACK
6. PSH/ACK
7. ACK
8. PSH/ACK
9. ACK
10. FIN/ACK
11. ACK
12. FIN/ACK
13. ACK

## P6

Une trame TCP est constituée d'un header et de données (payload). Le header TCP contient 20 bytes avec les informations suivantes :



https://fthmb.tqn.com/FHdeC8UhfnoaTE8jvZ4KlobWu1E=/1235x693/filters:fill(auto,1)/tcp-header-56a1adc85f9b58b7d0c1a24f.png

On constate que les adresses IP de destination et de source ne sont pas indiquées. Elles sont présentes uniquement dans la couche 3 (IP).

La charge utile est la taille des données maximale (1500 bytes) moins les deux headers TCP et IP (20 bytes chacun). Résultat = 1460 bytes.

## P7

1. Pour le client, on ne remarque aucun changement. Les trames échangées entre le client et le serveur sont les mêmes. La connexion a été faite, le message a été envoyé depuis le client, mais il a été stocké dans un buffer du côté serveur.
2. Elle permet de spécifier la taille du buffer (taille des données de la trame sans le header) du côté serveur. En la paramétrant à 1, on constate que le nombre de messages nécessaire entre le client et le serveur augmente drastiquement puisqu'il faudra autant de message que la taille des données en bytes.

Fichier Wireshark : TP01\_P7\_B.pcapng

## P8

# Envoi d'un objet non-standard

## P9

On s'aperçoit que l'objet MyMessage est sérialisé. En inspectant les données de la trame envoyée contenant l'objet MyMessage, on retrouve la chaîne de caractères originale sans le 'é' qui est codé sur 4 caractères en UTF-8 (c3a9). La chaîne de caractères originale est bien réceptionnée sous la bonne forme après déserialisation du côté serveur.

Fichier Wireshark : TP01\_P9.pcapng

## P10

Le premier message contient la structure de l'objet (attributs) et le deuxième message contient les valeurs de ces attributs. Pour plus de précision, voir à la question P11.

## P11

Nous avons ajouté un attribut Numero de type int avec la valeur 65535 (FF FF en hexadécimal) de manière à retrouver facilement cette valeur dans la visualisation des datas dans wireshark.

Nous retrouvons effectivement la valeur 00 00 FF FF dans la trame du deuxième message, juste avant la chaîne de caractères 'génialè' qui était la valeur du message envoyé.

Fichier Wireshark : TP01\_P11.pcapng

# Echange de datagrammes

## P12

On constate que contrairement aux messages TCP, UDP n'établit pas de connexion et donc qu'un seul datagramme est envoyé. Aucune réponse n'est envoyée pour confirmer la réception du datagramme. Les ports source et destination sont bien ceux fixés par le code java.

Fichier Wireshark : TP01\_P12.pcapng

## P13

Non, puisqu'il n'y a pas de réponse venant du serveur avec le protocole UDP. Le paquet est envoyé et perdu. Si le client envoie un deuxième message et que le serveur est instancié dans l'intervalle, ce dernier recevra le message comme son premier et unique message reçu. Par contre, le numéro de ce deuxième message sera 1 et non pas 0 grâce à la fonctionnalité d'incrément de msgNum.

Fichier Wireshark : TP01\_P13.pcapng

## P14

Une fois l'objet de la classe DatagramSocket instancié, un buffer est mis en place de la même façon que pour la question P7. Donc dès que l'on appelle la méthode receive(), le message est récupéré grâce au buffer.