TP N°4 de Réseaux - Etude des protocoles de la couche transport d'Internet UDP et TCP

Léo Tran Dorian Mounier Eloi Charra

07/04/2022

Q2

Le numéro de port n'est pas obligatoirement le même sur les deux machines, cela est indépendant puisqu'il est relatif à la machine. Il est cependant impossible d'utiliser deux fois le même numéro de port car un port est égal à une communication. On ne peut pas avoir deux communications au même endroit en même temps.

Q3

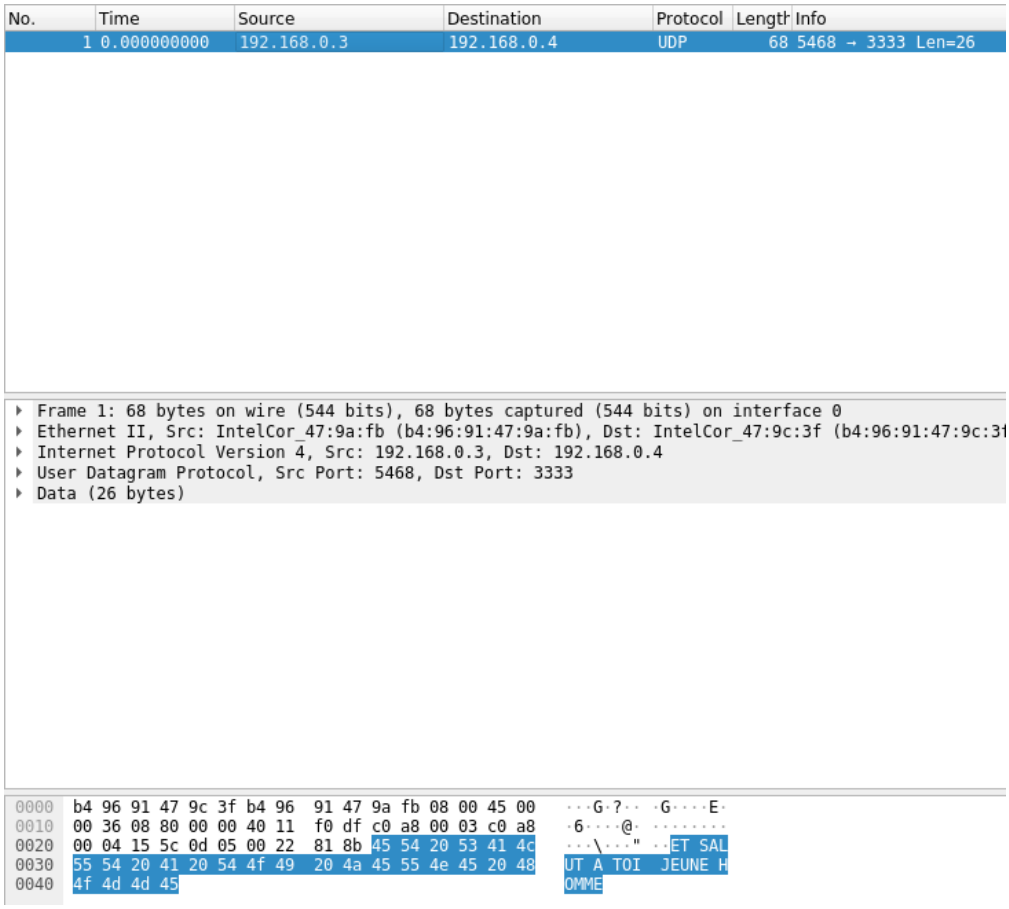


Figure 1: Paquet engendré lors de l'émission d'un message

Ce paquet est constitué du port UDP source, du port UDP destination de la taille en octet du message et d'une détection d'erreur optionnelle en IPV4 mais obligatoire en IPV6. Puis après cette courte entête nous avons : - Numéros de port (sur 2 octets) - Longueur en nombre d'octets (sur 2 octets) - Détection d'erreur optionnelle (par "checksum")

Il y a le message (surligné en bleu sur la capture wireshark ci-dessus). Il ne faut pas oublier toutefois que ce protocole est encapsulé par d'autres couches notamment la couche 3 *Réseau*.

Q4

L'identificateur de socket correspond à un port relié à une addresse. On doit connaître l'id pour identifier le paquet qui sera émis. On ne connaît pas l'identifiant de la socket distante car elle est liée au poste distant.

Q5

Les champs de l'entête UDP :

- Source port : port de la machine émettrice
- Dest port : port de la machine distante
- Length : longeur du message
- Checksum : permet d'assurer l'intégrité du paquet reçu. Il est calculé à partir des informations de l'entête UDP et des données.
- Data : les données transmises

L'ensemble du protocole UDP est transmis à IP.

Q6

- Demander la réception avant émission : Le message est affiché dès lors qu'il est émis et donc arrivé à l'host. Il n'y a pas de confirmation de connexion entre les deux machines, l'une écoute sur un port et il peut, ou pas, y avoir de message. En demandant la réception avant, la machine écoute sur le port en espérant entendre quelque chose. Dès que le message arrive, la machine le lis.
- Plusieurs paquets avant réception : Nous avons besoin de lancer la commande `rcvfrom` une fois pour chaque message, le premier `rcvfrom` récupère que le premier paquet envoyé.
- Aucune collisions quand envoie et receptionne en même temps. Demander à recevoir plus d'octets revient à recevoir tout le message sans superflux et demander à recevoir moins d'octets c'est recevoir que une partie du message
  - exemple envoi "bonjour" (7 longueur) : reception de 1 octet = "b"
  - exemple envoie de bonjour (7 longueur) : reception de 999 octets = "bonjour"
- Envoie vers une machine débranchée : aucun envoie / aucune trace (rien sur wireshark)

Q7

Le troisième paquet de 4000 octets s'est fait rejeté car il n'y avait pas la place (12k > taille du buffer de récéption de 10k). Si on envoie un petit message, celui-ci arrive à destination car il y a 2k octets de libre.

Q8

Lors d'un envoi d'un paquet UDP vers un port inexistant, on remarque un paquet ICMP envoyé vers l'émetteur `Destination unreachable (port unreachable)`. Le paquet arrive jusqu'à UDP (couche transport) (car IP est connue donc ça passe mais quand ça arrive à UDP, UDP dis qu'il ne connait pas de socket avec ce port) et UDP déclanche un ICMP.

Q9

UDP (User Datagramme Protocol) est un protocole permettant la transmissions de données entre deux entités. Ces entités possèdent un numéro de port et une adresse IP. L'objectif principal de ce protocole est d'être très léger et rapide, néglige totalement les autres points, telle que la correction d'erreur ou même la bonne réception des paquets. Notamment, on ne vérifie pas si la deuxième machine existe ou si son buffer est plein. La correction d'erreurs ou autre doit donc ce faire ou niveau applicatif si le programmeur le souhaite. Ensuite, il n'y a pas de ré-émission de paquet il n'y a donc aucune fiabilité du réseau. Néanmoins ce protocole est simple et rapide.

Q10

La première socket créée est *passive* car elle attend une connexion (c'est le serveur). L'autre est *active* car elle établit la connnection entre les deux machines.

Q11

Une socket est égale à une connexion donc il faut donc la dupliquer pour permettre à d'autres machines de se connecter au serveur.

Q12

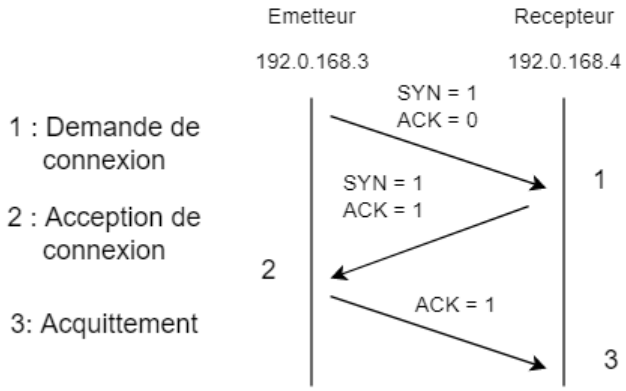


Figure 2: Paquets générés lors de l'établissement de la connexion

Q13

Le flag **SYN** est utilisé pour initialiser et établir une connexion. Il permet également de synchroniser les numéros de séquence. Les numéros de séquence et d'acquittement permettent au client et au serveur de se mettre d'accord sur les paquets reçus et émis. Le serveur incrémente le numéro de séquence lorsqu'il reçoit un paquet en guise de validité de réception. Le client reçoit le numéro de séquence incrémenté et comprend que le paquet est bien arrivé à destination. Les options lors de l'ouverture de connexion par TCP :

Les options TCP sont composées de : - la taille maximale du segment - l'échelle de la fenêtre - TCP Sack Permitted Option - Timestamp : TSval et TSecr

Q14

- accept avant connect : attente du connect
- accept apres connect : connection instantanée

Lorsqu'on ouvre plusieurs connexions d'une machine vers un même port destinataire, on peut voir que dans la liste des sockets, les modifications ont pu être faite.

Q15

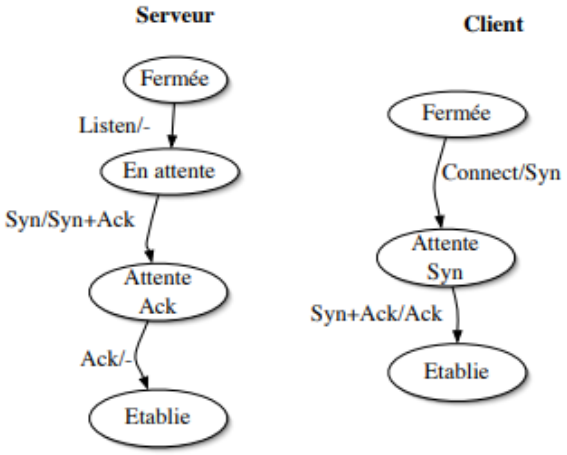


Figure 3: Etapes lors de l'ouverture d'une socket

Q16

Les deux numéros de port ainsi que les deux numéros de séquence permettent d'identifier une connexion.

Q18

Lorsqu'on essaye de se connecter à un port inexistant, la machine distante renvoie une trame contenant le flag RST. Le flag Reset permet de signaler un état inconnu/incertain de la connexion. Cela permet de faire remarquer à la machine émettrice une erreur et qu'il faut qu'elle réinitialise sa connexion.

Q19

1	0.000000000	192.168.0.3	192.168.0.4	TCP	1514 6633 → 8888 [ACK] Seq=1 Ack=1 Win=1026 Len=1448 TSval=2980266674 TSecr=2173444438
2	0.000007711	192.168.0.3	192.168.0.4	TCP	1514 6633 → 8888 [ACK] Seq=1449 Ack=1 Win=1026 Len=1448 TSval=2980266674 TSecr=2173444438
3	0.000014183	192.168.0.4	192.168.0.3	TCP	66 8888 → 6633 [ACK] Seq=1 Ack=2897 Win=981 Len=0 TSval=2173604736 TSecr=2980266674
4	0.000015207	192.168.0.3	192.168.0.4	TCP	1514 6633 → 8888 [ACK] Seq=2897 Ack=1 Win=1026 Len=1448 TSval=2980266674 TSecr=2173444438
5	0.000316739	192.168.0.3	192.168.0.4	TCP	1514 6633 → 8888 [ACK] Seq=4345 Ack=1 Win=1026 Len=1448 TSval=2980266674 TSecr=2173604736
6	0.000323854	192.168.0.4	192.168.0.3	TCP	66 8888 → 6633 [ACK] Seq=1 Ack=5793 Win=936 Len=0 TSval=2173604736 TSecr=2980266674
7	0.000325905	192.168.0.3	192.168.0.4	TCP	1514 6633 → 8888 [ACK] Seq=5793 Ack=1 Win=1026 Len=1448 TSval=2980266674 TSecr=2173604736
8	0.000329705	192.168.0.3	192.168.0.4	TCP	1514 6633 → 8888 [ACK] Seq=7241 Ack=1 Win=1026 Len=1448 TSval=2980266674 TSecr=2173604736
9	0.000334624	192.168.0.4	192.168.0.3	TCP	66 8888 → 6633 [ACK] Seq=1 Ack=8689 Win=890 Len=0 TSval=2173604736 TSecr=2980266674
10	0.000729140	192.168.0.3	192.168.0.4	TCP	1378 6633 → 8888 [PSH, ACK] Seq=8689 Ack=1 Win=1026 Len=1312 TSval=2980266675 TSecr=2173604736
11	0.101795249	192.168.0.4	192.168.0.3	TCP	66 8888 → 6633 [ACK] Seq=1 Ack=10001 Win=870 Len=0 TSval=2173604837 TSecr=2980266675

Figure 4: Paquet engendré par le transport des données

Les champs SEQUENCE NUMBER et ACK NUMBER permettent de compter le nombre d'octets qui sont envoyés d'un côté et réceptionnés de l'autre. Ce systeme permet de vérifier l'intégrité des données envoyées. Le SEQUENCE NUMBER est incrémenté quand un paquet TCP est envoyé tandis que ACK NUMBER est incrémenté quand la machine host a reçu des paquets TCP.

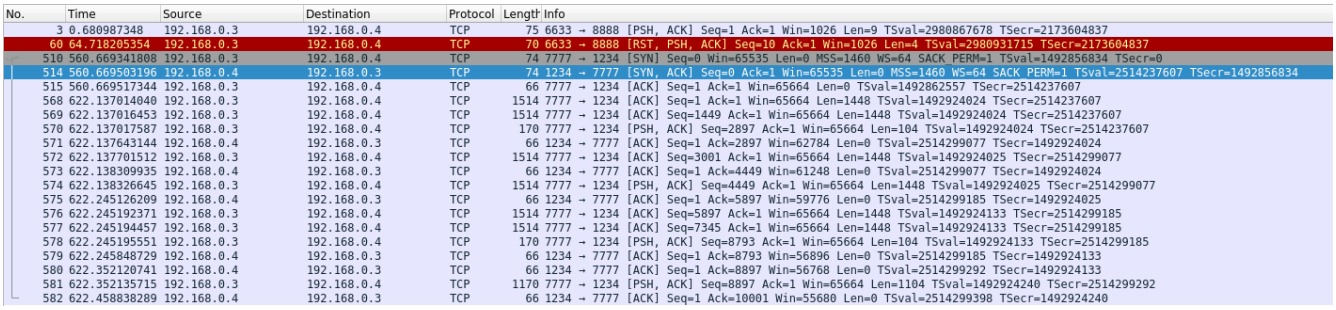
Q20

On remarque qu'il n'y a pas toujours d'acquittement pour chaque paquet. Un seul acquittement peut suffir pour valider la réception de plusieurs paquets.

Q21

Comme dit précédemment, UDP ne renvoie pas de paquet s'ils sont perdus. Contrairement à TCP qui assure de ré-envoyé les paquets perdus.

Q22



No.	Time	Source	Destination	Protocol	Length	Info
3	0.680987348	192.168.0.3	192.168.0.4	TCP	75	6633 → 8888 [PSH, ACK] Seq=1 Ack=1 Win=1026 Len=9 TSval=2980867678 TSecr=2173604837
60	64.718205354	192.168.0.3	192.168.0.4	TCP	70	6633 → 8888 [RST, PSH, ACK] Seq=10 Ack=1 Win=1026 Len=4 TSval=2980931715 TSecr=2173604837
510	560.669341808	192.168.0.3	192.168.0.4	TCP	74	7777 → 1234 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 SACK_PERM=1 TSval=1492856834 TSecr=0
514	560.669503196	192.168.0.4	192.168.0.3	TCP	74	1234 → 7777 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=64 SACK_PERM=1 TSval=2514237607 TSecr=1492856834
515	560.669517344	192.168.0.3	192.168.0.4	TCP	66	7777 → 1234 [ACK] Seq=1 Ack=1 Win=65664 Len=0 TSval=1492862557 TSecr=2514237607
568	622.137014040	192.168.0.3	192.168.0.4	TCP	1514	7777 → 1234 [ACK] Seq=1 Ack=1 Win=65664 Len=1448 TSval=1492924024 TSecr=2514237607
569	622.137016453	192.168.0.3	192.168.0.4	TCP	1514	7777 → 1234 [ACK] Seq=1449 Ack=1 Win=65664 Len=1448 TSval=1492924024 TSecr=2514237607
570	622.137017587	192.168.0.3	192.168.0.4	TCP	170	7777 → 1234 [PSH, ACK] Seq=2897 Ack=1 Win=65664 Len=104 TSval=1492924024 TSecr=2514237607
571	622.137643144	192.168.0.4	192.168.0.3	TCP	66	1234 → 7777 [ACK] Seq=1 Ack=2897 Win=62784 Len=0 TSval=2514299077 TSecr=1492924024
572	622.137701512	192.168.0.3	192.168.0.4	TCP	1514	7777 → 1234 [ACK] Seq=3001 Ack=1 Win=65664 Len=1448 TSval=1492924025 TSecr=2514299077
573	622.138309935	192.168.0.4	192.168.0.3	TCP	66	1234 → 7777 [ACK] Seq=1 Ack=4449 Win=61248 Len=0 TSval=2514299077 TSecr=1492924024
574	622.138326645	192.168.0.3	192.168.0.4	TCP	1514	7777 → 1234 [PSH, ACK] Seq=4449 Ack=1 Win=65664 Len=1448 TSval=1492924025 TSecr=2514299077
575	622.245126209	192.168.0.4	192.168.0.3	TCP	66	1234 → 7777 [ACK] Seq=1 Ack=5897 Win=59776 Len=0 TSval=2514299185 TSecr=1492924025
576	622.245192371	192.168.0.3	192.168.0.4	TCP	1514	7777 → 1234 [ACK] Seq=5897 Ack=1 Win=65664 Len=1448 TSval=1492924133 TSecr=2514299185
577	622.245194457	192.168.0.3	192.168.0.4	TCP	1514	7777 → 1234 [ACK] Seq=7345 Ack=1 Win=65664 Len=1448 TSval=1492924133 TSecr=2514299185
578	622.245195551	192.168.0.3	192.168.0.4	TCP	170	7777 → 1234 [PSH, ACK] Seq=8793 Ack=1 Win=65664 Len=104 TSval=1492924133 TSecr=2514299185
579	622.245848729	192.168.0.4	192.168.0.3	TCP	66	1234 → 7777 [ACK] Seq=1 Ack=8793 Win=56896 Len=0 TSval=2514299185 TSecr=1492924133
580	622.352120741	192.168.0.4	192.168.0.3	TCP	66	1234 → 7777 [ACK] Seq=1 Ack=8897 Win=56768 Len=0 TSval=2514299292 TSecr=1492924133
581	622.352135715	192.168.0.3	192.168.0.4	TCP	1170	7777 → 1234 [PSH, ACK] Seq=8897 Ack=1 Win=65664 Len=1104 TSval=1492924240 TSecr=2514299292
582	622.458838289	192.168.0.4	192.168.0.3	TCP	66	1234 → 7777 [ACK] Seq=1 Ack=10001 Win=55680 Len=0 TSval=2514299398 TSecr=1492924240

Figure 5: Comportement de TCP lors de l'envoi de paquets

Q23

Dans le cas d'un buffer trop petit, cela va réduire le débit applicatif. Premièrement, le buffer sera utilisé à de multiple reprise avec des parcours de ce dernier à chaque fois. Le simple parcours est négligeable mais la répétition elle ne l'est pas est augmente la latence. De plus, à chaque fois il faut ajouter une entête donc des trames avec peu de données va réduire le débit. Inversement, si la taille de ce buffer est grande le débit sera plus élevé. La latence dépendra grandement de ce temps d'émission.

Q24

Débit applicatif = taille du buffer / RTT

Pour 2000 octets: Débit applicatif : 1000/20 = 50 Ko/s

Pour 4000 octets: Débit applicatif : 2000/20 = 100 Ko/s

Pour 10000 octets: Débitapplicatif : 10000/20 = 500 Ko/s

Q25

La taille optimale du buffer se calcule avec la formule suivante: RTT * débit minimale du réseau. Le débit minimum que l'on utilise est de 10 mo/s, alors on trouve une taille de 200 000 octets.

Q26

Si l'on utilise un buffer d'émission de taille non adéquate à la latence du réseau, ce dernier peut se remplir plus vite qu'il ne se vide, et peut donc perdre des paquets dans le cas où le buffer reste constamment plein.

Q27

Récepteur:

```
Tant que(Vrai){
  Si SEQ est attendu alors on retourne ACK = SEQ + longueur
}
```

Si le paquet reçu n'est pas celui attendu alors le paquet est mémorisé dans le buffer de réception mais aucun acquittement n'est envoyé

Emetteur:

Envoie le premier paquet. Si pas d'acquittement après la fin du timer alors le timer est réévalué et armé à nouveau. Si on est pendant notre timer, on continue à envoyer les paquets en incrémentant le SEQ comme si on avait reçu le ACK. le dernier ACK va valider les anciens acquittements

Q28

Win dépasse pas 10 000. L'émetteur ne peut pas emettre plus d'octets que la taille du buffer de réception.

Suite à cette échange, le client continuera à rester en relation avec le serveur en lui demandant si celui-ci est toujours présent. Ce à quoi le serveur répondra que, oui ilest toujours là, mais que son buffer de réception est plein. Ce court échange de question réponse se répétera jusqu'à ce que le buffer du serveur ne sois plus plein avec une intervalle de temps deux fois plus important à chaque fois.

Lorsqu'on a fait un read du côté du serveur, celui-ci envoie un paquet TCP en indiquand que l'espace restant dans son buffer a été mis à jour, le client va répondre en envoyant les données restantes

Q29

Le champ Window est codé sur 16 bits et correspond au nombre d'octets du buffer de réception. Il permet de ne pas envoyer des paquets et inonder le réseau alors que le récepteur ne peut pas réceptionner les données.

Q30

L'émetteur est débloqué lorsque le serveur est en capacité d'accepter une quantité de données correspondant à la moitié de la taille de son buffer d'origine. Ceci a été créé pour éviter d'inonder le réseau avec des lectures de 1 octet par 1 octet du côté récepteur et du coup des envoies de 1 octet par 1 octet du côté émetteur ce qui serait catastrophique pour le réseau. Prenons le cas suivant :

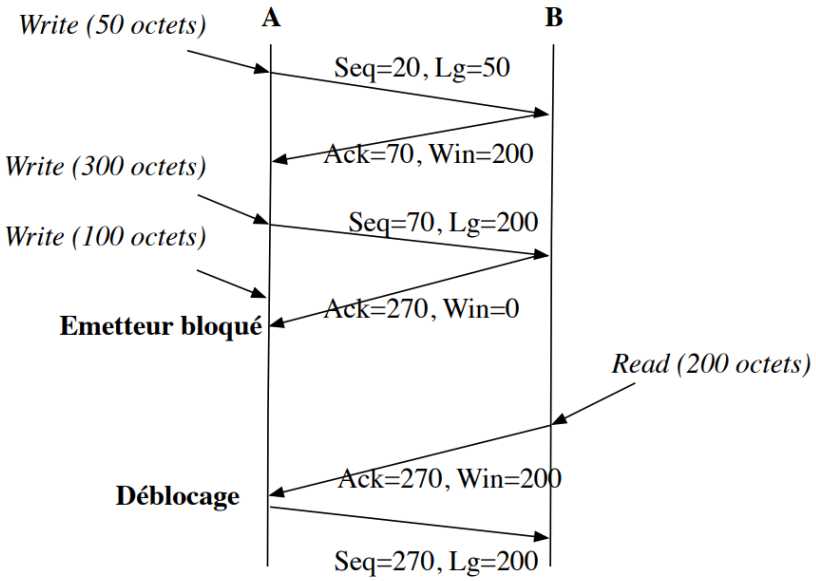


Figure 6: Chronogramme d'échange de données

Q31

Au cours de la synchronisation, le client et le serveur s'échangent une première fois la taille de leur buffer de réception. Si la taille nécessaire est disponible dans le buffer, lorsqu'on envoie des paquets. Tant qu'un paquet READ n'aura pas été envoyé, ce qui permettra de vider en partie le buffer, les paquets vont être renvoyé.

Q32

Il n'est pas pertinent d'avoir un buffer d'émission plus grand que le buffer de réception car une partie de ce buffer ne sera jamais utilisé donc c'est juste une perte de mémoire inutile comme on peut le voir sur ce schéma.

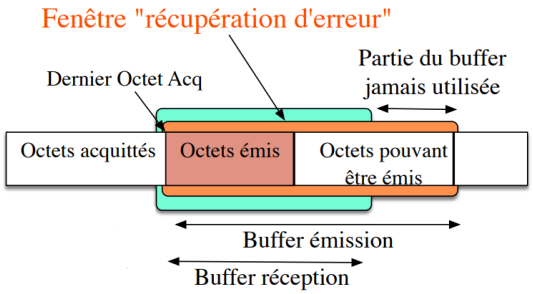


Figure 7: Schéma buffer

Q33

Paquet avec flag "FIN" envoyé

Le flag FIN permet d'informer la fin d'une connexion entre les deux machines. Ce flag est émit des deux machines pour fermer définitivement la connexion et éviter que l'un d'entre deux retransmettent un paquet entre temps.

Q34

Pour la fermeture voir la question 36. Afin de répérer que cette trame TCP est une trame `FIN` une option sur un bit, un flag, est mis en place avec 1 si c'est une trame `FIN` 0 sinon.

Si on ferme la connexion d'un seul côté qu'on essaye de communiquer avec la machine alors un message d'erreur apparaît : Socket is not connected. Si on ferme la connexion entre les machines A et B, on a les mêmes paquets que précédemment. Lorsque que l'on fait un write de B vers A, on envoie un premier paquet TCP de B vers A pour dire que l'on veut émettre un message, puis un paquet TCP de A vers B est envoyé avec le flasg RSH, qui indique que la machine A n'est plus accessible. Ce flag permet de terminer la connexion entre les deux machines.

Q35

Lorsque l'on refais un write un reçoit un message d'erreur `Broken Pipe` puisque le fermeture est un `close` et permet de différencier d'un `shutdown` qui fermerai la communication que d'un côté et permettrai toujours au second partenaire d'envoyer des messages.

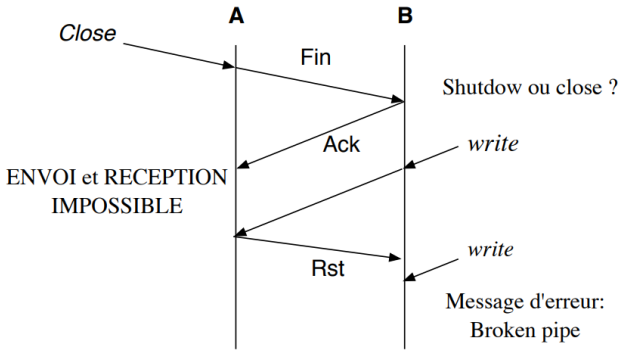


Figure 8: Chronogramme d'échange de données

Q36

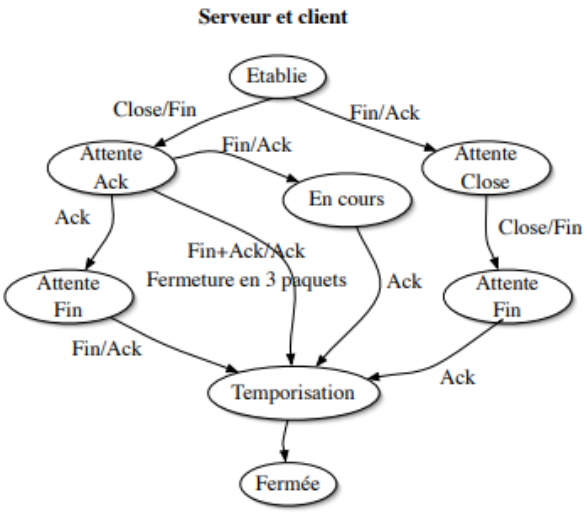


Figure 9: Automate TCP

Pour la fermeture il y a trois cas possible du côté de celui qui décide de fermer la communication. Contre seulement un pour celui qui le reçoit.

Celui qui reçoit le `close` envoie le ack puis ferme à son tour le communication avec le `fin` puis reçoit à son tour le `ack` pour valider.

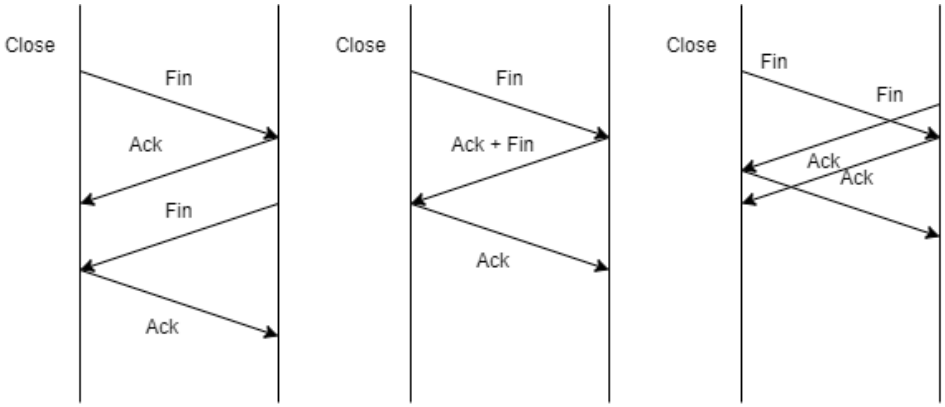


Figure 10: Chonogrammes des trois cas de fermeture

Le premier cas classique est `Fin` envoyé par l'un des deux puis le second qui envoie `Ack` puis de-même pour le second.

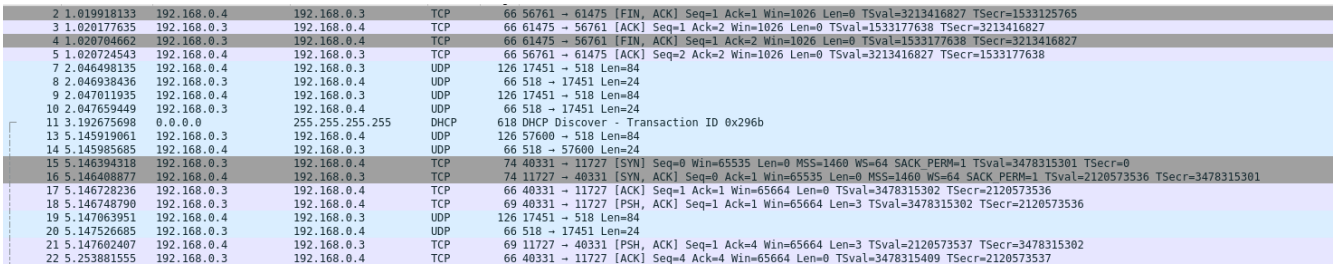
Le deuxième très rare ce produit lorsque le fin du premier est reçu par le second lorsque souhaite fermer équalement la communication il fait donc le `Fin` et le `Ack` dans une même trame ce qui permet de faire le dernier `Ack` et de fermer la communication en seulement 3 trames et non 4.

Le troisième plus probable avec les deux qui souhaitent fermés quasiment en même temps ce qui permet correspond au même cas que le premier mais avec un temps beaucoup plus réduit.

Q37

Il est difficile de lire le contenu des paquets car ces-derniers sont chiffrés. On constate tout de même un certains "mot" qui se répète à plusieurs reprise peut-être un mot de passe. Cependant on aperçoit que des trames UDP sont utilisé lors de la conversation. Il est bon de noté également que si on met un port connu par exemple 80 Wireshark

va croire que l'on fait ce type d'échange, avec l'exemple HTTP. Le contenu des paquets n'est pas intéressant à cause du chiffrement mais on voit clairement que l'échange de login et de mot de passe.



2	1.019918133	192.168.0.4	192.168.0.3	TCP	66 56761 → 61475 [FIN, ACK] Seq=1 Ack=1 Win=1026 Len=0 TSval=3213416827 TSecr=1533125765
3	1.020177635	192.168.0.3	192.168.0.4	TCP	66 61475 → 56761 [ACK] Seq=1 Ack=2 Win=1026 Len=0 TSval=1533177638 TSecr=3213416827
4	1.020704662	192.168.0.3	192.168.0.4	TCP	66 61475 → 56761 [FIN, ACK] Seq=1 Ack=2 Win=1026 Len=0 TSval=1533177638 TSecr=3213416827
5	1.020724543	192.168.0.4	192.168.0.3	TCP	66 56761 → 61475 [ACK] Seq=2 Ack=2 Win=1026 Len=0 TSval=3213416827 TSecr=1533177638
7	2.046498135	192.168.0.4	192.168.0.3	UDP	126 17451 → 518 Len=84
8	2.046938436	192.168.0.3	192.168.0.4	UDP	66 518 → 17451 Len=24
9	2.047011935	192.168.0.4	192.168.0.3	UDP	126 17451 → 518 Len=84
10	2.047659449	192.168.0.3	192.168.0.4	UDP	66 518 → 17451 Len=24
11	3.192675698	0.0.0.0	255.255.255.255	DHCP	618 DHCP Discover - Transaction ID 0x296b
13	5.145919061	192.168.0.3	192.168.0.4	UDP	126 57600 → 518 Len=84
14	5.145985685	192.168.0.4	192.168.0.3	UDP	66 518 → 57600 Len=24
15	5.146394318	192.168.0.3	192.168.0.4	TCP	74 40331 → 11727 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 SACK_PERM=1 TSval=3478315301 TSecr=0
16	5.146408877	192.168.0.4	192.168.0.3	TCP	74 11727 → 40331 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=64 SACK_PERM=1 TSval=2120573536 TSecr=3478315301
17	5.146728236	192.168.0.3	192.168.0.4	TCP	66 40331 → 11727 [ACK] Seq=1 Ack=1 Win=65664 Len=0 TSval=3478315302 TSecr=2120573536
18	5.146748790	192.168.0.3	192.168.0.4	TCP	69 40331 → 11727 [PSH, ACK] Seq=1 Ack=1 Win=65664 Len=3 TSval=3478315302 TSecr=2120573536
19	5.147063951	192.168.0.4	192.168.0.3	UDP	126 17451 → 518 Len=84
20	5.147526685	192.168.0.3	192.168.0.4	UDP	66 518 → 17451 Len=24
21	5.147602407	192.168.0.4	192.168.0.3	TCP	69 11727 → 40331 [PSH, ACK] Seq=1 Ack=4 Win=65664 Len=3 TSval=2120573537 TSecr=3478315302
22	5.253881555	192.168.0.3	192.168.0.4	TCP	66 40331 → 11727 [ACK] Seq=4 Ack=4 Win=65664 Len=0 TSval=3478315409 TSecr=2120573537

Q38

On effectue une commande "pwd" qui va nous afficher le chemin pour aller à notre répertoire courant. A l'aide de Wireshark, on observe deux sortes de paquets : les paquets UDP et les paquets TCP.

On initialise la connexion avec le protocole TCP, puis on échange les données grâce au protocole UDP. On observe que les paquets UDP se compose de la même façon que les paquets TCP et sont typés comme des protcoles TCP. On peut donc dire que l'échanges de messages est possible grâce au protcole TCP.