Introduction aux Réseaux TP4 : Etude des protocoles de la couche transport d'Internet UDP et TCP

Line POUVARET, Mickaël TURNEL

2015-2016

2.1 Le protocole UDP

Configuration des deux machines :

M1 : Adresse IP → 192.168.1.1
M2 : Adresse IP → 192.168.1.2

- Numéro de port socket M1 : 51841 (Id = 3)
- Numéro de port socket M2 : 28316 (Id = 3)
- Donc sur M1 on exécute : sendto 3 M2 28316 (avec un message "coucou M2")
- Sur M2 : recvfrom 3 9 → 9 octets car la chaîne possède 9 caractères.
- recvfrom nous affiche bien le message "coucou M2"

1. L'identificateur socket permet de définir sur quelle socket on veut envoyer ou recevoir un message.

   Lors de la création de la socket, l'ID et le port nous sont indiqués.

2. - Source Port : 51841 (51841) → correspond au port de la machine émettrice du paquet.
   - Destination Port : 28316 (28316) → correspond au port de la machine recevant le paquet.
   - Length : 17 → correspond à la taille de l'entête UDP + la taille des données (Champ Data)
   - Checksum : 0xa8c4 [validation disabled]

   UDP donne l'information de son protocole à IP.

- Si on demande la réception avant l'envoi du paquet, la socket qui va recevoir le paquet se met en attente de réception et dès qu'elle reçoit un paquet, recvfrom se débloque.
- On envoit 3 paquets avant la réception (avec comme messages respectifs "paquet 1" "paquet 2" "paquet 3").

  A la réception, lors du premier appel de recvfrom, on reçoit le premier paquet puis on doit exécuter à nouveau recvfrom pour recevoir le deuxième paquet et une dernière fois pour recevoir le troisième paquet.
- On envoit simultanément 1 paquet à partir de M1 et M2 sur l'autre machine respective.

  On reçoit simultanément sur les 2 machines. On ne constate aucun problème au niveau des croisements des paquets.

Si on demande à recevoir moins d'octets que la taille du paquet envoyé, on ne récupère pas le message en entier (il est tronqué selon la taille que l'on a demandé).

Si on demande à recevoir plus d'octets que la taille du paquet envoyé, on récupère bien le message en entier.

- Lorsque l'on déconnecte M2 du réseau et qu'on souhaite envoyer un paquet UDP de M1 vers M2, la commande sendto nous informe qu'il n'y a aucune route disponible vers M2. ("sendto() : No route to host")

  Aucun paquet n'est donc envoyé. (Observé sur Wireshark)

- La taille du buffer d'envoi est de 9216 ce qui signifie que lorsque l'on veut envoyer un message de taille supérieure à 9216 via un paquet UDP, on ne pourra pas car on recevra une erreur "Message too long".

  La taille du buffer en réception est 42080.

  Lorsque l'on envoit 5 paquets de taille 9000 (donc 45000 de taille totale), on constate que la machine réceptrice des paquets sature son buffer en réception et tout ce qui est reçu après les 42080 octets est perdu.

- Lors d'un envoi d'un paquet UDP vers un port inexistant, on observe le transit du paquet UDP de M1 vers M2 et un paquet ICMP de M2 vers M1 informant cette dernière que la destination n'est pas atteignable (le port est inatteignable).

3. **Fonctionnement du protocole UDP** :

   Lorsqu'on reçoit un paquet UDP, on ignore le message si la taille du message est plus grande que la taille du buffer en réception.

   Si ce n'est pas le cas, on reçoit correctement le message.

   Si la taille de lecture du message est inférieure à la taille du message, on lit uniquement le nombre d'octets de la taille demandée et ce qui reste du message est ignoré.

   Si ce n'est pas le cas, on lit le message en entier.

   Si le port de destination est inatteignable, la machine qui devait recevoir le paquet renvoit un paquet ICMP vers la machine émettrice lui indiquant que le port n'est pas atteignable.

2.2 Le protocole TCP

Etablissement d'une connexion

- Numéro de port de la socket "passive" créée sur M1 : 30350
- Numéro de port de la socket "active" créée sur M2 : 18376 (avec connect M1 30350)
- Sur M1 : on exécute **accept** qui nous donne l'ID de la socket (3), un appel de M2 (18376) a été intercepté, la connexion est établie sous l'identificateur 4.

4. La socket dite "passive" correspond à celle qui attend une connexion TCP.

   La socket dite "active" correspond à celle qui se connecte à une autre socket.

   La socket "passive" joue le rôle de serveur.

   La socket "active" joue le rôle de client.

5. Sur Wireshark :
   - M2 envoie un paquet TCP à M1 avec le flag SYN pour demander un établissement de connexion. (Sequence number = 201DA294)
   - M1 envoie un paquet TCP à M2 avec le flag SYN et le flag ACK pour confirmer l'établissement de la connexion. (Sequence number = A1B6D2E9, ACK = 201DA295)
   - M2 envoie un paquet TCP à M1 avec le flag ACK pour informer M1 qu'il a bien reçu la confirmation de l'établissement de la connexion. (Sequence number = 201DA295, ACK = A1B6D2EA)
6. Le flag SYN est utilisé lorsque l'on veut demander l'établissement d'une connexion.

   Le numéro de séquence correspond au numéro du paquet TCP envoyé.

   On constate que le numéro d'acquittement du deuxième paquet correspond au numéro de séquence du premier paquet + 1. (201DA294 + 1 = 201DA295)

   On constate que le numéro d'acquittement du troisième paquet correspond au numéro de séquence du deuxième paquet + 1. (A1B6D2E9 + 1 = A1B6D2EA)

   Dans le troisième paquet, le numéro de séquence correspond au numéro d'acquittement du deuxième paquet.

   Le numéro d'acquittement permet d'envoyer un accusé de réception à la machine émettrice.

   Options utilisées :
   - Maximum segment size : permet de définir la taille maximale des fragments
   - No-Operation : A COMPLETER
   - Window scale : permet de définir l'échelle de la fenêtre
   - TCP Sack Permitted Option : booléen qui permet d'autoriser TCP Sack
   - Timestamps : A COMPLETER
7. Au moment de la primitive des sockets "accept" on reste en attente d'une connexion TCP pour permettre à deux machines de communiquer.

   Le fait d'effectuer accept avant ou après connect nous permet quoiqu'il arrive d'établir la connexion entre les deux machines et elles peuvent toujours communiquer entre elles.
8. Après avoir établi plusieurs connexions d'une machine vers un même port destinataire, on se rend compte que ce qui identifie les messages reçus est l'ID du paquet et donc le port de la machine émettrice. (Puisque qu'à chaque connect, on nous attribue un ID différent et un port différent)

   Ce qui signifie que chaque connexion est différenciée. (une socket par connexion pour la machine émettrice)
9. On constate que les états de connexion peuvent etre
   - ESTABLISHED (connexion établie)
   - CLOSE WAIT (en attente de fermeture de la connexion du côté de la machine réceptrice)
   - FIN_WAIT_2 (en attente de fermeture du côté de la machine émettrice)
   - TIME_WAIT (quand on a fermé la connexion d'abord du côté de la machine réceptrice et après du côté de la machine émettrice) : la connexion sera fermée après un certain temps. (timer précis)
   - LISTEN (en écoute)

10. Une demande de connexion vers un port inexistant entraine un "connexion refused".

    Sur Wireshark on constate que le deuxième paquet de M1 vers M2 comporte le flag RST (Reset) et ACK.

Etude du séquencement et la récupération d'erreur

11. Lors d'un envoi de paquet TCP avec une très grande taille de données, on constate sur Wireshark, que le paquet TCP est en fait fragmenté en plusieurs paquets chaînés par les numéros de séquence (la taille des fragments peut être modifiée via l'option Maximum segment size).

    Le champ Sequence number permet donc d'identifier le fragment du paquet et le champ ACK Number permet d'envoyer un accusé de réception concernant le fragment précédent. (ACK Number = Sequence number + la taille du segment)

12. On n'a pas un acquittement par paquet de données car les fragments ne sont pas nécessairement reçus dans l'ordre.

    Si un 2ème fragment arrive avant le 1er alors on aura un seul acquittement.

13. Lorsqu'on déconnecte une machine du réseau et que l'on veut envoyer un paquet vers celle-ci, on n'observe aucun transit de paquet dans Wireshark mais aucune erreur n'est annoncée dans la console.

    On nous informe même que les octets ont été envoyés.

    Lorsque l'on rebranche la machine et que l'on exécute read directement, on attend un court instant et le message que l'on avait envoyé lorsque la machine était déconnectée est bien reçu.

    Contrairement à UDP, TCP conserve les messages qui n'ont pas pu être envoyés et attend que la connexion soit de nouveau établie pour les renvoyer.

    Le protocole UDP perdait tout simplement ces messages.

14. Après avoir réduit la taille du buffer d'émission du côté de la machine émettrice, lors d'un envoi de 10 000 octets, on constate sur Wireshark (lancé sur la machine émettrice) que les données sont fragmentées en 5 paquets de 2000 octets.

    Ainsi la taille du buffer d'émission va influer sur le débit applicatif, plus la taille du buffer est grande et plus le débit applicatif sera grand.

    En effet, plus on fragmente nos données et plus le débit va être réduit.

15. Le buffer d'émission de TCP sert à stocker les données à envoyer en attendant leur acquittement (après les avoir envoyé).

    Il est très important car il nous permet de renvoyer nos données si la connexion a été perdue.

16. Un buffer d'émission trop petit entrainer un débit applicatif plus faible car les données seront beaucoup plus fragmentées avant leur envoi (et leur acquittement). On doit attendre l'acquittement pour plus de fragments.

17. Pour calculer le débit applicatif que l'on obtiendrait avec différentes tailles de buffer et une latence de 10 ms, on se sert de la formule suivante :

    Nombre d'envois total = $\frac{N_{donnees}}{N_{Buffer}}$

    avec $N_{donnees}$ = taille des données

    et N_{Buffer} = taille du buffer d'émission

    Etant donné que l'on doit attendre l'acquittement pour les données, la latence du réseau est appliquée 2 fois sur le nombre d'envois total.

Donc :

Débit applicatif = (latence*2)*Nombre d'envois total

On obtient donc le tableau suivant pour un envoi de 10 000 octets :

Taille du Buffer d'Emission (octets)	Debit Applicatif (ms)
1000	200
2000	100
10000	20

18. **Algorithme de mise à jour de ACK et SEQ → Envoi d'un paquet**

si *Evenement == "Reception_paquet(SEQ S, ACK A, Longueur L)"* **alors**
| envoi_paquet(SEQ A, ACK S+L, 0) ;
fin

Algorithme de mise à jour de ACK et SEQ → Réception d'un paquet

si *Taille_données > Taille_Buffer* **alors**
| fragmenter(données) ;
fin
pour *Chaque donnée à envoyer OU chaque timer_envoi(données) == 0 (timer d'envoi dépassé)* **faire**
| mettre_dans_buffer(données) ; envoi_données(SEQ S, ACK A, Longueur L) ;
| **si** *Evenement == "Reception_paquet(SEQ A, ACK A2, 0)" ET A2 == S + L* **alors**
| | fin ;
| **fin**
fin

2.2.3 Controle de flux

19. La machine qui reçoit le message renvoit des paquets TCP avec le flag ACK et avec Window size value = 0.

    Cela signifie que l'émetteur doit en quelque sorte ralentir le rythme d'envoi de son message.

    L'émetteur se retrouve en quelque sorte "bloqué".

20. En faisant un seul read, on renvoit un paquet TCP avec le flag ACK et avec un Window size value différent de 0.

    Cela débloque donc l'émetteur qui peut renvoyer.

21. En faisant des reads de peu d'octets, on libère progressivement le buffer ce qui fait qu'on permet de débloquer petit à petit l'emetteur qui renvoit au fur et à mesure.

22. Quand le buffer de réception a été complètement libéré, l'émetteur se retrouve "débloqué".

23. Principe du controle de flux de TCP : A COMPLETER.

Libération d'une connexion

A COMPLETER (Q24 - Q30 + 2.3)