1 Client/Serveur TCP

1.1 Serveur

Question 1.1.1. Que doit faire le serveur pour mettre en place une socket publique ?

Il faut déterminer un numéro de port à allouer au serveur. Pour ce faire, il existe plusieurs possibilités :

1. prendre un numéro de port au hasard. Dans ce cas, il existe une possibilité pour que le port soit déjà pris par un autre programme s'exécutant sur la machine. Ce qui mènera au traditionnel `Address already in use` lors de l'appel à `bind()`.
2. demander à l'administrateur réseau d'allouer un numéro de port spécifique pour l'application. Le client pourra alors récupérer le numéro de port via un appel à `getservbyname()`.

Question 1.1.2. Doit-on prévoir dans le programme sur quel hôte on va faire tourner ce serveur ?

Non, c'est là tout l'intérêt de la constante `INADDR_ANY`. Cette constante permet de récupérer l'adresse IP de l'hôte sur lequel est lancé le programme [1]. Dans l'éventualité où l'hôte bénéficie de plusieurs adresse IP, `INADDR_ANY` en *choisit* une arbitrairement. C'est pourquoi il est quand même possible de spécifier une adresse particulière.

Remarquons dans tous les cas, que le client, afin d'être capable de contacter le serveur, doit connaître l'hôte sur lequel ce dernier est lancé.

Question 1.1.3. Pourra-t-on lancer plusieurs serveurs sur un même hôte ?

Cela est possible lorsque les serveurs ont tous une socket écoutant sur un port différent. Sans quoi nous revenons encore et toujours au fameux `Address already in use`.

Question 1.1.4. Pourra-t-on lancer plusieurs clients sur des hôtes différents ?

Dans la mesure où une socket est identifiée par un numéro de port, une adresse IP et un protocole, deux serveurs lancés sur deux hôtes différents auront deux sockets différentes. Il est donc tout à fait possible de lancer plusieurs serveurs sur des hôtes différents.

Question 1.1.5. Si on lance le serveur, jusqu'à quelle étape doit-il se dérouler sans être bloqué ?

Afin de pouvoir être contacté, un serveur TCP doit effectuer un `accept()`. L'appel a `accept()` étant bloquant, le serveur doit donc se dérouler sans encombre jusqu'au `accept()`.

1.2 Client

Question 1.2.1. Quelles informations sont nécessaires au client pour joindre le serveur ?

Pour pouvoir contacter le serveur, le client doit connaître les informations de sa socket, *i.e.* le numéro de port sur lequel écoute le serveur, l'adresse IP de l'hôte sur lequel le serveur fonctionne et le protocole de communication utilisé par ce dernier.

Question 1.2.2. Si l'utilisateur se trompe dans le nom de l'hôte où le serveur est prévu, que se passe-t-il ?

Le client effectue une demande de connexion, à l'aide de l'appel à `connect()`, vers le mauvais hôte qui (très probablement) ne dispose pas de serveur TCP écoutant sur le port fourni. L'appel à `connect()` a donc toutes les chances d'échouer et donc de retourner une erreur.

Question 1.2.3. En supposant que le serveur a été lancé, jusqu'à quelle étape est-ce que le client de déroule sans être bloqué ?

Voilà un exemple de question très claire... Si le serveur est lancé, on a tendance à penser que le client ne sera pas bloqué, puisqu'il n'y a aucune raison pour que le schéma algorithmique foire. La véritable question sous-jacente est de savoir à quels moments le client peut se retrouver bloqué et surtout pourquoi.

Pour qu'un processus voie son exécution bloquée, il doit, et ce de manière intuitive, réaliser un appel à une fonction bloquante. Or le schéma algorithmique du client en contient trois, à savoir `connect()`, `send()`, `recv()` . Reste maintenant à expliquer dans quels cas un blocage peut se produire. Concernant le `connect()` on peut imaginer une file d'attente saturée du côté serveur [2]. Si la file d'attente n'est pas saturée, le client peut bloquer sur le `send()` si le serveur n'a pas encore réalisé l'appel à `accept()` [3] Ce cas peut arriver facilement si le serveur croule sous les sollicitations des clients.

Concernant le blocage sur `recv()`, il n'y a pas d'explications réellement plausibles autre que la panne du serveur... Mais bon, il m'est quand même demandé de vous le citer. Alors voilà, dans le cas où le serveur plante, où si la connexion physique est rompue entre le `send()` et le `recv()`, ce dernier peut bloquer le client.

1. J'en profite au passage pour rappeler qu'à ce titre cette constante ne doit pas être utilisée dans le code du client pour initialiser la structure de données `struct sockaddr_in` relative au serveur.
2. Sous Linux et Unix, lorsqu'une file d'attente est saturée, le client réémet périodiquement une demande de connexion, ce jusqu'à ce que la file d'attente se vide un peu ou jusqu'à expiration d'un timeout.
3. Un appel à `connect()` est un appel réussi à partir du moment où la demande de connexion du client est placée en file d'attente, et non comme on pourrait le penser à partir du moment où le serveur effectue l'appel à `accept()`.

Question 1.2.4. Peut-on lancer plusieurs clients sur un même hôte ?

Tout dépend de l'implémentation du client. Si vous avez développé votre client en lui affectant un numéro de port fixe, alors il est impossible de lancer plusieurs client sur un même hôte, si, au contraire, le port de la socket client a été laissé à la charge du système [4], alors on peut lancer autant de clients que de ports disponibles sur l'hôte.

Dans la mesure où le serveur est un serveur mono processus, chaque client attendra donc son tour bien au chaud dans la file d'attente.

Question 1.2.5. Peut-on lancer plusieurs clients sur plusieurs hôtes ?

Oui.

1.3 La communication

Question 1.3.1. Que se passe-t-il du côté du client, si le serveur ferme le circuit virtuel sans envoyer de message ?

Le client et le serveur utilisent un protocole de communication dit connecté. Dans ce cas là, le client est averti de la déconnexion du serveur. Si l'appel à `send()` n'a pas encore été effectué, ce dernier retournera une erreur spécifié à l'aide de la valeur de retour -1, sinon c'est l'appel à `recv()` qui informera le client de la déconnexion du serveur à l'aide d'une valeur de retour 0.

Question 1.3.2. Même question du côté serveur si le client ferme le circuit sans envoyer de message au serveur ?

Même réponse sans la digression sur `send()`.

Question 1.3.3. Décrire comment le client peut dépasser l'expédition de la requête et rester bloqué ensuite (par la volonté du serveur) ?

Il faut entendre par là, *Comment peut-on modifier le schéma algorithmique du serveur pour faire en sorte que e client reste bloqué après l'expédition de sa requête ?*. Dans ce cas, il suffit de ne jamais répondre au client sans pour autant fermer la connexion.

Question 1.3.4. Est-ce que le serveur peut annoncer de quel hôte et quelle boîte est venue la requête du client **avant** d'accepter la demande de connexion ?

Non, c'est impossible dans la mesure où la connexion, en TCP, est la base de tout échange d'informations. Sans la réalisation d'un `accept()`, il est donc impossible d'analyser les informations relatives à un client. Notez cependant qu'avec certaines options, il est possible de lire une demande de connexion avant de l'extraire de la file d'attente, mais ici aussi, il faut *lire* avant d'analyser, et donc accepter la connexion.

Question 1.3.5. En fin de compte, quels sont les moyens qui sont à la disposition d'un processus pour refuser des clients ?

Rien n'empêche le serveur de mettre fin à un connexion juste après un `accept()`.

2 Notion de masque

2.1 Exercice

Question 2.1.1. Dans un réseau de classe B, on veut créer des sous-réseaux permettant de voir le réseau global comme un ensemble de réseaux de classe C. Combien de sous-réseaux peut-on déterminer ? Quel est le nombre maximal d'hôtes possible dans chaque sous-réseau ? Comment sont représentées les adresses "réseaux" et "tous" ? Quel masque faut-il utiliser dans chaque sous-réseau ?

Prenons comme exemple le second réseau de l'exercice précédent, à savoir `130.160.0.0/16`.

La question est un peu ambiguë dans la mesure où un réseau de classe C est caractérisé par une adresse IP commençant par la série de bit `110` et donc ayant une adresse commençant par un nombre compris dans l'intervalle $[192, 223]$. Il est donc impossible d'obtenir un réseau de classe C à partir d'un réseau de classe B. Mais ce que l'on cherche réellement à faire, c'est de partitionner le réseau de classe B (avec un masque de 16 bits) en des sous-réseaux (eux aussi de classe B) avec un masque de 24 bits.

Maintenant que l'ambiguïté est levée, il est possible de réaliser, à partir d'un réseau B/16 [5], 256 sous-réseaux B/24 [6].

On peut alors définir tous les sous-réseaux ainsi : pour $i = 0, 1, \cdots, 255$,

Désignation `130.160.i.0`

Masque `255.255.255.0`

4. Ceci est réalisé en initialisant le champ `sin_port` de la structure `struct sockaddr_in` du client à 0 avant l'appel à `bind()`
5. Parce que je suis fainéant, je définis la notation suivante : B/16 -> réseau de classe B possédant un masque de 16 bits. Attention, il ne s'agit pas d'une notation standard, elle n'est donc pas utilisable dans un examen.
6. C'est quand même super pratique !

Plage d'adresses `130.160.i.1 - 130.160.i.254`

Adresse du réseau `130.160.i.0`

Adresse de broadcast `130.160.i.255`

Pour rappel, en possédant l'adresse d'une des machines du réseau et le masque du réseau, on peut calculer :

— l'adresse du réseau comme suit : $(\text{IP machine})_2 \wedge (\text{Masque})_2$, $(\text{IP})_2$ désignant la représentation binaire de IP et $\wedge$ désignant le ET bit à bit. Dans notre exemple, considérons la machine H_3 ayant pour adresse IP `130.160.21.22` et pour masque `255.255.255.0`, l'adresse réseau est définie par

	10000010	10100000	00010101	00010110
$\wedge$	11111111	11111111	11111111	00000000
	10000010	10100000	00010101	00000000

soit `130.160.21.0`

— l'adresse de broadcast comme suit : $(\text{IP machine})_2 \vee \overline{(\text{Masque})_2}$, $\overline{(\text{Masque})_2}$ représentant le complémentaire de la représentation binaire du masque et $\vee$ le OU bit à bit. Ce qui nous donne, pour notre exemple :

	10000010	10100000	00010101	00010110
$\vee$	00000000	00000000	00000000	11111111
	10000010	10100000	00010101	11111111

soit `130.160.21.255`

Question 2.1.2. On veut diviser un réseau de classe C en huit sous-réseaux. Proposer une solution ; donner un exemple en précisant la capacité d'adressage de chaque sous-réseau, les masques ainsi que les adresses réservées.

Notons tout d'abord que $8 = 2^3$. Il faut donc allonger le masque de trois bits. Prenons pour exemple le réseau contenant H_1 à savoir `194.195.196.0/24`. Pour le diviser en huit sous réseaux on allonge le masque de trois bits. On peut alors définir les huit sous-réseaux suivants :

Désignation	Masque	Plage adresses	Adr. Réseau	Broadcast
194.195.196.0	255.255.255.224	194.195.196.{1-30}	194.195.196.0	194.195.196.31
194.195.196.32	255.255.255.224	194.195.196.{33-62}	194.195.196.32	194.195.196.63
194.195.196.64	255.255.255.224	194.195.196.{65-94}	194.195.196.64	194.195.196.95
194.195.196.96	255.255.255.224	194.195.196.{97-126}	194.195.196.96	194.195.196.127
194.195.196.128	255.255.255.224	194.195.196.{129-158}	194.195.196.128	194.195.196.159
194.195.196.160	255.255.255.224	194.195.196.{161-190}	194.195.196.160	194.195.196.191
194.195.196.192	255.255.255.224	194.195.196.{193-222}	194.195.196.192	194.195.196.223
194.195.196.224	255.255.255.224	194.195.196.{225-254}	194.195.196.224	194.195.196.255

Le nombre d'adresses de chaque réseau est donné par $2^l - 2$, où l représente le nombre de bits laissés libres par le masque, dans notre exemple, $l = 32 - 27 = 5$. Le nombre d'adresse est donc $2^5 - 2 = 30$.

Question 2.1.3. Peut-on faire une division en sis sous-réseaux, quatre de 30 hôtes chacun et deux autres de 62 chacun ?

Oui, il suffit de reprendre le découpage précédent et de fusionner deux par deux les quatre derniers. Attention cependant à bien mettre à jour le masque de réseau pour chacun d'entre eux.

Désignation	Masque	Plage adresses	Adr. Réseau	Broadcast
194.195.196.0	255.255.255.224	194.195.196.{1-30}	194.195.196.0	194.195.196.31
194.195.196.32	255.255.255.224	194.195.196.{33-62}	194.195.196.32	194.195.196.63
194.195.196.64	255.255.255.224	194.195.196.{65-94}	194.195.196.64	194.195.196.95
194.195.196.96	255.255.255.224	194.195.196.{97-126}	194.195.196.96	194.195.196.127
194.195.196.128	255.255.255.192	194.195.196.{129-190}	194.195.196.128	194.195.196.191
194.195.196.192	255.255.255.192	194.195.196.{193-254}	194.195.196.192	194.195.196.255

Question 2.1.4. On veut maintenant avoir une division en trois sous-réseaux. Quelles solutions peut-on proposer ?

Deux solutions sont possibles :

1. diviser en au moins quatre sous-réseaux, en laissant tous les sous-réseaux surnuméraires inoccupés pour une utilisation extérieur,
2. diviser en au moins quatre sous-réseaux et procéder à des fusions de sous-réseaux pour en obtenir trois finaux.

2.2 Exercice

Question 2.2.1. Donner un exemple de d'adresses consécutives permettant de réaliser ce réseau ; préciser comme ci-dessus la capacité, les masques et adresses spécifiques.

Pour réaliser un réseau commun à partir de deux réseaux de classe C, il faut que les adresses choisies aient 23 bits en commun. Si l'on veut de plus que les adresses soient consécutives, il faut prendre une adresse se terminant par un nombre pair et l'adresse terminant par le nombre impair suivant (sans quoi les deux adresses présenteront une différence d'au moins deux bits).

Ainsi `192.193.194.0/24` et `192.193.195.0/24` dont compatibles. Fusionner ces deux réseaux de classe C nous donnera un réseau d'adresse et de masque `192.193.194.0/23`. Le nombre d'adresses disponibles sur ce réseau est donné par $2^{32-23} - 2 = 2^9 - 2 = 510$.

Question 2.2.2. Profiter pour corriger légèrement cet énoncé.

Le nombre maximal de machines est 510 et non 508.

Question 2.2.3. Donner un exemple de deux adresses de classe C consécutives, non compatibles pour former un réseau.

Comme précisé ci dessus, il suffit de prendre comme premier réseau un réseau dont l'adresse finit par un nombre impair. Ainsi `192.193.193.0/24` et `192.193.194.0/24` sont deux réseaux non compatibles.

Question 2.2.4. Est-il nécessaire de limiter les masques à une suite consécutive de bits à 1 et à une suite consécutive de bits à 0 ?

Même si les conventions abondent dans ce sens, et même si une immense majorité des logiciels réseaux utilisent cette convention, la notion même de masque n'est pas soumise à cette restriction. Cependant, utiliser des masques ne respectant pas cette convention ne ferait que détruire un peu plus la lisibilité dans un domaine de l'informatique qui n'est pas reconnu pour sa facilité d'accès.

3 Transformations subies par un message

3.1 Exercice

Question 3.1.1. M_1 et M_2 sont sur le même réseau local et ont une seule connexion au réseau chacune. Leur adresse IP respective est `193.2.4.8` et `193.2.4.16`. Le réseau local est un réseau Ethernet et les adresses physiques sont `8:4:CF:20:36:AB` et `8:20:FE:10:20:48`.

L'objectif ici est de décrire le parcours d'un paquet dans les différentes couches du modèle OSI et de décrire les interactions entre ces couches. On ne décrira que sommairement toutes les modifications liées à l'encapsulation des données.

3.1.1 Interface entre l'application et la couche transport.

L'application fournit à TCP un grand nombre d'informations :

- un pointeur sur l'espace mémoire contenant le message à envoyer,
- un numéro de port et une adresse IP identifiant la socket du destinataire,
- un descripteur de fichier identifiant la socket d'émission, ce descripteur de fichier est lié (par un appel à `bind()` ou `connect()`) à un numéro de port et à une adresse IP (dans le cas d'un appel à `bind()` avec la constante `SOCKADDR_IN`, l'adresse IP fournie est l'une des adresses identifiant l'hôte sur lequel tourne le client).

3.1.2 Rôle du transport

TCP va ensuite découper le message en paquets. La taille maximale de ces paquets dépend de la négociation préalable entre les deux entités TCP, ici M_1 et M_2. Du côté de M_1, TCP récupère ces paquets un par un et y ajoute en entête le port de l'émetteur et celui du destinataire, il ajoute également une numérotation des paquets ainsi qu'un checksum (code de contrôle).

3.1.3 Interface entre le transport et le réseau

TCP envoie ensuite chacun des paquet en couche IP en y adjoignant les adresses IP fournies par la couche applications. On remarque que la couche transport n'a utilisé que les numéro de port.

3.1.4 Rôle de la couche transport (IP)

À partir de chaque paquet TCP, un paquet IP est créé en ajoutant l'adresse IP source et destination au paquet. Un problème se pose, c'est à la couche réseau de faire parvenir à la couche liaison l'adresse MAC de la prochaine étape. Il faut donc dans un premier temps identifier le destinataire. C'est le rôle de l'algorithme de routage qui va déterminer si la machine cible appartient au même réseau que la machine source et sinon vers quel routeur faire transiter la requête. Dans notre cas, l'algorithme de routage identifie M_2 comme faisant partie du même réseau que M_1, M_2 est donc la prochaine étape, il faut maintenant récupérer son adresse MAC. Là encore deux possibilités :

— l'adresse IP de M_2 est déjà contenue dans la table ARP de M_1, dans ce cas, l'adresse MAC permettant de contacter M_2 est connue et cette adresse mac est transmise à la couche liaison avec le paquet IP créé.

— M_1 ne connaît pas l'adresse MAC de M_2. Dans ce cas, le paquet est mis en attente le temps d'effectuer une recherche ARP. La couche réseau transmet alors à la couche liaison l'adresse IP pour laquelle une adresse MAC est recherchée (celle de M_2). Cette dernière transmet la requête en broadcast sur le réseau. M_2 répond alors en fournissant son adresse MAC qui sera ajoutée à la table ARP.

Le paquet reprend maintenant sa route, après que la couche réseau a calculé et vérifié le checksum.

3.1.5 Interface entre le réseau et la liaison

IP fournit le paquet avec l'adresse mac de M_2.

3.1.6 Rôle de la couche liaison

C'est cette couche qui demande l'accès au média, gère l'émission, détecte les collision, réexpédie au besoin...

Question 3.1.2. On suppose maintenant que M_1 et M_2 sont sur deux réseaux distincts

Dans ce cas deux changements interviennent :

— l'algorithme de routage détectera que M_2 n'appartient pas au même réseau que M_1 et fournira donc l'adresse du routeur permettant de contacter M_2. C'est donc l'adresse MAC du routeur qu'il faudra déterminer via ARP.

— Arrivé sur le routeur R, le paquet remonte jusqu'à la couche réseau. L'algorithme de routage est à nouveau lancé pour déterminer si M_2 appartient à l'un des réseaux de R, ce qui est le cas ici. On reprend ensuite la démarche précédente.

Question 3.1.3. Si plus d'un routeur intervient, expliquer ce qui se passe dans chaque routeur.

Comme précédemment, à ceci près que lors de l'algorithme de routage de pour le routeur R_i, c'est l'adresse de R_{i+1} qui est retournée.

4 Routage dans les réseaux

4.1 Exercice

Question 4.1.1. Donner un exemple de boucle infinie comportant au moins trois routeurs.

Il suffit d'imaginer une table de routage erronée sur l'un des routeurs, par exemple R_3 réexpédie sur R_1 qui expédie sur R_2 qui expédie sur R_3.

Question 4.1.2. Rappeler l'algorithme utilisé pour éviter que ces boucles soient infinies.

Algorithme 1 : Algorithme TTL

```
ttl ← ttl - 1;
if ttl == 0 then
    Envoyer(MessageErreur, MachineSource);
    Supprimer(Paquet);
```

Question 4.1.3. Est-ce que cet algorithme est différent selon que les applications concernées par cet échange communiquent en utilisant un mode de transport connecté ou sans connexion ?

L'algorithme est implémenté en couche réseau et donc est indépendant du transport.

Question 4.1.4. Montrer comment deux paquets avec les mêmes adresses IP *source* et *destination* peuvent l'un arriver, l'autre être supprimé.

Nous avons vu que deux paquets peuvent prendre deux chemins différents. En utilisant cette remarque, il est facile d'imaginer que le second paquet prenne un chemin plus long que le ttl.

Question 4.1.5. Que doit faire R_s en plus de la suppression du paquet ?

Il doit envoyer un message d'erreur (ICMP) à l'hôte émetteur du paquet. C'est d'ailleurs la seule adresse qu'il peut connaître, car c'est la seule figurant dans l'entête du message.

Question 4.1.6. Décrire ce qui se passe à la suite d'une telle suppression lorsque le paquet supprimé fait partie d'une communication en TCP ? Préciser quelle entité doit être avertie de la suppression.

TCP doit se charger de la réémission du paquet, et ce jusqu'à ce que l'émission de ce dernier aboutisse ou jusqu'à atteindre le nombre maximum de réémission. Dans ce cas, l'application doit alors être mise au courant.

Dans les faits, notons que c'est la couche réseau qui prévient la couche transport de la suppression du paquet lorsqu'elle reçoit un ICMP.

Question 4.1.7. Même question avec UDP.

La couche réseau prévient la couche transport de la suppression du paquet (suite à la réception d'un ICMP). UDP n'étant pas prévu pour gérer les erreurs, ignore l'avertissement.