Applications client-serveur

1 Programmation client-serveur

Compétences attendues à l'issue de cette séance :
- identifier les protocoles TCP et UDP et la notion de port ;
- décrire une architecture client/serveur ;
- écrire des programmes Python communicant via UDP ou TCP ;
- programmer une application client/serveur simple en Python.

Vous rédigerez un compte rendu, sur lequel vous indiquerez la réponse à chaque question, vos explications et commentaires (interprétation du résultat, commandes ou outils utilisés, référence au code source produit qui sera fournit par ailleurs).

Vous devez créer un répertoire `TP1`.

Pour certains TP vous aurez besoin de deux machines. Vous vous connecterez en SSH à une autre machine par exemple celle d'un de vos collégues.

Exercice 1.1 : Programmation client-serveur

1 Rappels de réseau

Dans ce TP, vous allez devoir implémenter des serveurs et des clients utilisant deux protocoles réseaux : TCP et UDP. Ces deux protocoles sont situés au niveau de la couche 4 du modèle OSI : la couche *transport*. Nous utiliserons IP pour la couche 3 (couche réseau) et, la plupart du temps, Ethernet pour la couche 2 (liaison de données).

7.	Application
6.	Présentation
5.	Session
4.	Transport
3.	Réseau
2.	Liaison de données
1.	Physique

FIGURE 1 – Rappel : les 7 couches du modèle OSI.

1.1 Le protocole UDP

Le protocole UDP est un protocole *simple* et *non-fiable*. Il fonctionne en mode non-connecté et les messages envoyés ne sont pas acquittés (ce qui signifie que le récepteur n'envoie pas automatiquement de message pour indiquer qu'il a bien reçu). Ce protocole ne garantie pas à l'émetteur d'un message que le destinataire l'a bien reçu, ni que celui-ci n'a pas été altéré.

L'absence de phase de connexion permet à l'émetteur de commencer l'envoi de ses données dès le premier paquet (en TCP, nous verrons que l'établissement de la connexion suppose l'échange de plusieurs messages, ce qui demande du temps).

Comme le montre la figure 2, les paquets sont simplement envoyés entre les deux processus sans aucun paquet supplémentaire ajouté par le protocole. Les paquets du message sont envoyés directement dès le début de la communication. Lorsque la communication est terminée, on arrête simplement d'envoyer des paquets.

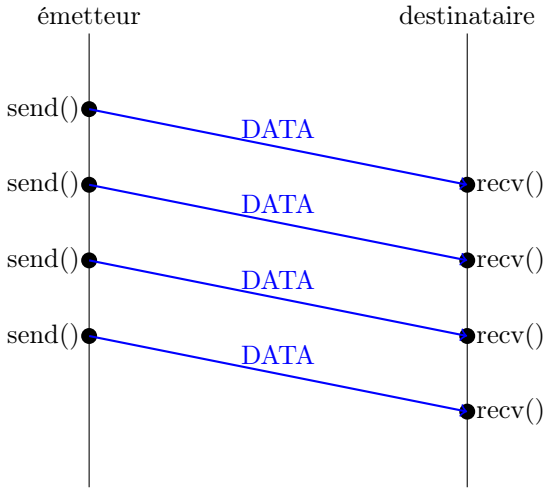


FIGURE 2 – Envoi de données sur UDP entre deux processus.

Le protocole UDP est donc léger et efficace, et se limite essentiellement à introduire la notion de *port* à IP : le numéro de port permettra de distinguer le processus (ou service) destinataire du paquet. UDP est en général utilisé par les applications qui privilégient la performance à la fiabilité : par exemple, des applications multimédia qui peuvent se permettre de perdre des paquets (qu'il ne vaut pas la peine de retransmettre puisque de toutes façons ils arriveraient trop tard).

UDP utilise un système d'adressage sur la machine permettant à une machine donnée d'être impliquée dans plusieurs communications sans les mélanger : ce sont les *ports* réseau. Ainsi, une communication UDP est caractérisée, sur une machine donnée, par le port UDP auquel elle est assignée. On peut faire une analogie avec le téléphone classique en assimilant chaque port à une ligne.

Le numéro de port est un nombre entier non signé codé sur deux octets : les numéros possibles sont donc compris entre 0 et 65 535 ($= 2^{16} - 1$). Les ports inférieurs à 1024 sont réservés au super-utilisateur : une application exécutée par un utilisateur normal ne pourra pas ouvrir ces ports, qui sont pour la plupart réservés à des services connus [1].

1. L'*Internet Assigned Numbers Authority* (IANA) est l'organisme chargé de l'attribution des numéros de ports UDP et TCP. Voir aussi la RFC 6335 et `http://www.iana.org/assignments/port-numbers`.

On dit qu'un programme *écoute sur un port* : c'est-à-dire qu'il est prêt à recevoir des données sur ce port. Un port peut être utilisé par une seule communication à la fois : si un programme essaye d'utiliser un port qui est déjà utilisé, le système renverra une erreur.

Les datagrammes UDP contiennent donc les deux ports (source et destination) entre lesquels la communication est effectuée. La structure d'un datagramme UDP est représentée par la figure 3. On parle ici de *datagramme* puisqu'il s'agit d'un protocole en mode non-connecté. Les longueurs des champs sont données en bits. Outre les ports, l'en-tête du datagramme contient également la longueur totale du datagramme. Ainsi, les données peuvent être de longueur variable. Enfin, l'en-tête contient une somme de contrôle d'erreur qui permet de vérifier l'intégrité des données transmises.

Port source (16)	Port Destination (16)
Longueur (16)	CRC (16)
Données	

FIGURE 3 – Structure d'un datagramme UDP

1.2 Le protocole TCP

Le protocole TCP fonctionne en *mode connecté* : une connexion doit être établie entre deux processus pour que des messages puissent être envoyés entre eux. On a alors la notion de *client* et de *serveur* : le serveur *attend les connexions des clients* et le client *se connecte à un serveur*. Une fois la connexion établie, les messages peuvent être envoyés du client vers le serveur ou du serveur vers le client, sans distinction (liaison bidirectionnelle, dite *full duplex*). À la fin de la séquence de communication, il est impératif de *fermer* la connexion.

L'établissement de la connexion se fait en utilisant le protocole dit de la *triple poignée de main* (triple handshake) :

— Le client envoie une requête de connexion : paquet `SYN`
— Le serveur accepte : paquet `SYN/ACK`
— Le client acquitte la réception de l'acceptation : paquet `ACK`

Ce mécanisme est illustré dans la figure 4, on l'on voit les paquets échangés lorsque la machine A se connecte en TCP à la machine B.

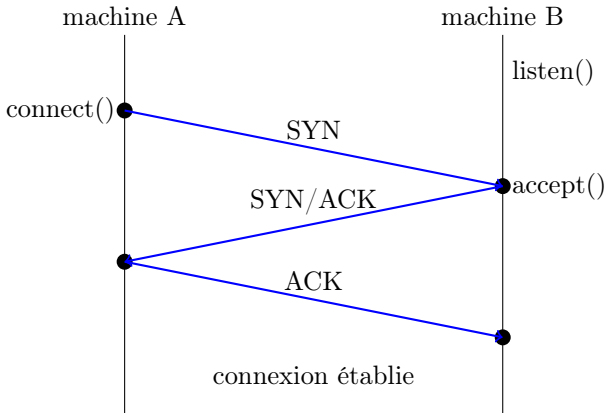


FIGURE 4 – Établissement d'une connexion en TCP : la triple poignée de main.

Les envois de paquets sont acquittés par des paquets `ACK` (acquittement). Le protocole TCP essaie de réduire le nombre de messages d'acquittement. En effet, la solution naïve serait que le récepteur réponde par un acquittement à chaque paquet reçu. Cette méthode simple doublerait le nombre de paquets envoyés sur le réseau ($2N$ paquets protocolaires pour N paquets envoyés par les processus) et augmenterait la charge du réseau, diminuant le débit effectif. De plus, si l'émetteur attend de recevoir un acquittement avant d'envoyer le paquet suivant, on ralenti considérablement la vitesse des communications : le canal de transmission n'est pas utilisé à plein temps.

Pour contourner ces problèmes, TCP utilise la technique de la *fenêtre glissante*. On ne bloque pas les envois de paquets suivants, mais on anticipe la réception des acquittements : l'émetteur continue d'envoyer des paquets sans bloquer sur l'attente de l'acquittement. Il envoie un certain nombre de paquets avant de s'assurer qu'un paquet a été acquitté. Par exemple, si on utilise une fenêtre de taille 5, l'émetteur enverra au plus 5 paquets en attendant que le premier soit acquitté. Si après ces 5 paquets l'acquittement n'a toujours pas été reçu, il bloquera ses envois jusqu'à ce qu'il reçoive l'acquittement, Les acquittements utilisent le numéro de séquence du paquet qu'ils acquittent pour que l'émetteur puisse déterminer quel paquet a été acquitté.

Ce mécanisme est illustré par la figure 5. La machine A envoie une succession de paquets à la machine B. Chaque paquet a un numéro de séquence. Le premier paquet est acquitté par A en utilisant le numéro de séquence de ce paquet. En attendant la réception de l'acquittement, A continue d'envoyer des paquets à B.

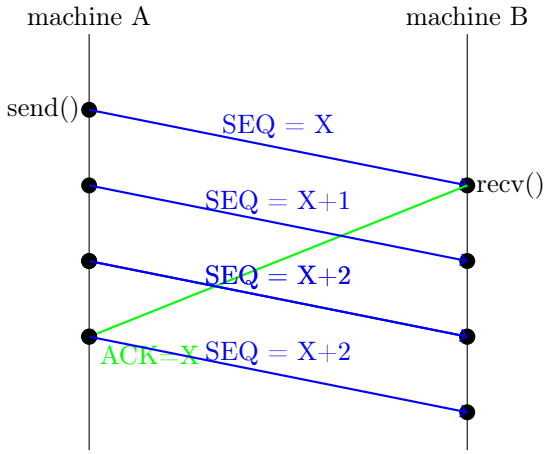


FIGURE 5 – Acquittement des paquets envoyés par TCP.

La fermeture de la connexion doit se faire par les deux processus : chacun doit fermer la connexion de son côté. On utilise alors le protocole dit de *poignée de main bidirectionnelle* (two-way handshake).

Chacun des deux processus ferme la connexion. Il envoie un paquet `FIN`, et l'autre processus lui répond par un paquet `FIN/ACK`. Une fois que les deux échanges ont eu lieu, la connexion est fermée.

Ce mécanisme est illustré par la figure 6. La machine A ferme la connexion : elle envoie un paquet `FIN`, qui est acquitté par la machine B qui lui répond par un paquet `FIN/ACK`. La connexion est alors considérée comme fermée par A mais pas encore par B. La machine B ferme ensuite la connexion de son côté : elle envoie un paquet `FIN`, qui est acquitté par la machine A qui lui répond par un paquet `FIN/ACK`. La connexion est maintenant fermée.

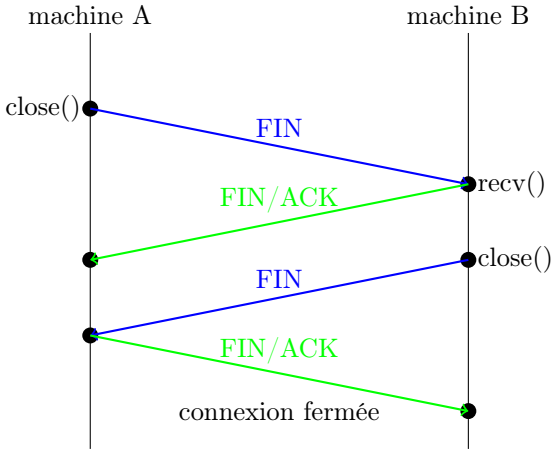


FIGURE 6 – Fermeture d'une connexion TCP : la poignée de main bidirectionnelle.

De même que le protocole UDP, le protocole TCP utilise les *ports* du système. Sur les systèmes pouvant communiquer par TCP et par UDP, il existe une série de ports distincts pour chaque protocole. Par exemple, sur vos machines vous pourrez voir qu'il existe des ports UDP et des ports TCP.

Nous avons vu que TCP fonctionne en mode *connecté*. Un programme qui attend qu'on vienne se connecter à lui écoute sur un port donné. Les autres programmes se connectent à lui sur ce port. Nous avons vu qu'un port ne peut pas être utilisé par plus d'une communication à la fois. Afin de libérer le port d'écoute pour que d'autres connexions puissent être établies, le système établit la connexion *sur un autre port* : le port d'écoute reste donc disponible pour accepter ultérieurement d'autres communications.

La structure d'un paquet TCP est représentée par la figure 7. On parle ici de *paquet* puisqu'il s'agit d'un protocole en mode connecté. Les longueurs des champs sont données en bits.

<table>
<tr><td colspan="3">port source (16)</td><td>port destination (16)</td></tr>
<tr><td colspan="4">numéro de séquence (32)</td></tr>
<tr><td colspan="4">numéro d'acquittement (32)</td></tr>
<tr><td>Lg. entête (4)</td><td>réservé (4)</td><td>drapeaux (8)</td><td>taille fenêtre (16)</td></tr>
<tr><td colspan="3">checksum (16)</td><td>pointeur urgent (16)</td></tr>
<tr><td colspan="4">Options éventuelles</td></tr>
<tr><td colspan="4">Données</td></tr>
</table>

FIGURE 7 – Structure d'un paquet TCP

1.3 Le protocole ARP

Sur un réseau local, les machines peuvent communiquer en utilisant le protocole Ethernet. Il s'agit d'un protocole de niveau 2 dans le modèle OSI. Le protocole Ethernet utilise les adresses MAC des cartes réseaux pour identifier chaque carte réseau sur le réseau local.

Lorsqu'un processus doit communiquer avec un autre processus, par exemple en IP, il détermine en utilisant sa table de routage par où il va faire transiter le réseau, c'est-à-dire si il va l'envoyer directement au destinataire ou si il va l'envoyer à un routeur qui va se charger de le transférer. Dans les deux cas, le processus va envoyer le paquet IP à un autre processus exécuté sur une

machine située sur le même réseau local.

La carte réseau de la machine émettrice doit donc déterminer l'adresse MAC de la carte réseau de la machine destinataire, en connaissant son adresse IP. Pour cela, les machines utilisent le protocole *ARP* (Address Resolution Protocol).

La machine A (émettrice) diffuse sur le réseau local une *requête ARP*. Il s'agit d'une diffusion sur Ethernet : l'adresse destination est `ff:ff:ff:ff:ff:ff`. Les paquets ARP contiennent deux couples (adresse MAC, adresse IP). La requête ARP est du type : `ARP.request(`$\texttt{MAC}_A$`,`$\texttt{IP}_A$`,0,`$\texttt{IP}_B$`)`.

La machine B écoute sur le réseau et voit passer la diffusion d'une requête ARP contenant son adresse IP. Elle y répond donc en envoyant une *réponse ARP* à la machine A : un paquet `ARP.reply(`$\texttt{MAC}_B$`,`$\texttt{IP}_B$`,`$\texttt{MAC}_A$`,`$\texttt{IP}_A$`)`.

La machine A reçoit la réponse et enregistre l'association entre l'adresse IP et l'adresse MAC de A dans son *cache ARP*. Vous pouvez consulter le cache ARP de votre machine en utilisant la commande Unix `arp`.

2 Programmation réseau

2.1 Les sockets

Une socket est un *point terminal sur le réseau*. C'est la structure de donnée qui permet de communiquer sur le réseau. Lorsque l'on écrit un programme communiquant sur le réseau, on manipule des sockets pour établir ou fermer les connexion dans les cas des protocoles connectés, envoyer et recevoir des données sur le réseau.

2.2 Programmation réseau sur les sockets

En Python, les fonctions associées aux sockets sont fournies dans le module `socket`. Vous devez donc, en premier lieu, importer ce module.

3 Programmation client-serveur

Un *serveur* est un programme qui attend des requêtes et y répond. Il tourne en permanence, et est dans l'attente de demandes de service venant d'autres programmes. Le serveur écoute sur un port connu et traite les requêtes qui arrivent sur ce port.

Un *client* est un programme qui émet ces requêtes auprès du serveur. L'établissement d'une connexion (en mode connecté) et d'un échange de messages (en mode non-connecté) se fait *toujours* à l'initiative du client.

3.1 Client et serveur UDP

Le but de cet exercice est d'écrire un client et un serveur UDP qui communiqueront ensemble, et d'observer les échanges effectués entre les deux.

3.1.1 Serveur UDP

La première chose à faire consiste à créer une socket. On doit lui passer des attributs, c'est-à-dire :

— La famille d'adresses utilisées : préciser qu'il s'agit d'une socket IP
— Le type de socket : préciser qu'il s'agit de communications par datagrammes

Parmi les familles possibles, on trouve `AF_INET` pour IPv4 (valeur par défaut), `AF_INET6` pour IPv6 ou `AF_UNIX` pour les socket Unix. Parmi les types on trouve `SOCK_STREAM` pour les protocoles par paquets comme TCP (valeur par défaut) ou `SOCK_DGRAM` pour les protocoles par datagrammes comme UDP.

Pour créer une socket on utilise la fonction `socket()` du module `socket` en lui passant ces deux arguments. Cette fonction retourne un objet socket, que l'on va manipuler pour effectuer nos communications sur le réseau.

Une fois la socket créée, on lui associe un nom avec `bind()`. On passe un tuple à cette fonction constitué d'un nom de machine (*hostname*) et d'un port. Le port est celui sur lequel le serveur va écouter. Le nom de machine est, dans le cas d'un serveur qui accepte des connexions en provenance de toutes les machines, la chaîne de caractères vides.

1. Écrivez un serveur UDP qui ouvre une socket et l'associe au port 23.
2. Lancez votre serveur sur une autre machine de la salle TP. Que se passe-t-il ?
3. Modifiez votre serveur pour associer votre socket au port 9875. Que se passe-t-il maintenant quand vous lancez votre serveur ?
4. Les fonctions associées aux sockets sont susceptibles de lever des exceptions de type `socket.error`. Attention à bien les gérer. Modifiez votre programme afin d'afficher un message d'erreur et quitter proprement le programme dans le cas où une exception est levée.
5. Nous l'avons vu, le protocole UDP est un protocole *non-connecté* : le serveur reçoit directement les messages des clients. Pour cela, les sockets disposent d'une fonction `recvfrom()` qui prend en paramètre la taille du tampon à utiliser et retourne deux éléments : les données reçues, et l'adresse du client par qui ces données ont été envoyées. Si les données à recevoir sont trop grosses pour tenir dans le tampon, l'excédent est perdu. L'adresse du client est un tuple contenant l'adresse IP et le port source. Modifiez votre serveur afin qu'il puisse recevoir un message. Si une exception est soulevée, n'oubliez pas de fermer la socket avec la fonction `close()` avant de quitter le programme.
6. Un serveur écoute en permanence ce qui vient. Modifiez votre programme pour mettre le `recvfrom()` dans une boucle infinie. Lorsqu'un message est reçu, affichez le message reçu et l'adresse source.
7. Lancez votre serveur. Normalement, votre programme ne devrait pas vous rendre la main : il est dans le `recvfrom()`. Vous pouvez voir les ports utilisés sur votre système avec l'utilitaire `netstat`. Ce programme prend quelques options en paramètres, parmi lesquels les protocoles concernés, les informations visualisées... Regardez l'aide en ligne de `netstat` et affichez les ports UDP utilisés sur votre machine. Normalement vous devriez voir votre serveur écouter sur le port que vous avez choisi.

3.1.2 Client UDP

Le client UDP est très simple : nous avons vu qu'il n'est pas nécessaire d'établir de connexion, et qu'il doit simplement envoyer des données au serveur (figure 2).

On dispose pour cela de la fonction socket `sendto()`, qui prend deux paramètres : le message à envoyer, et un tuple contenant l'adresse du serveur et le port sur lequel celui-ci écoute. Cette fonction retourne le nombre d'octets ayant été effectivement envoyés sur le réseau.

8. Écrivez un programme client qui crée une socket UDP et envoie un message sous forme d'une chaîne de caractères à votre serveur. Côté serveur, le programme doit afficher le message reçu.
9. Si le port n'est pas correct, que se passe-t-il ?
10. Avec l'outil Wireshark vous pouvez capturer le trafic passant sur une interface réseau. Lancez Wireshark sur votre machine et lancez une capture, exécutez votre client et arrêtez la capture. Quels sont les messages effectivement échangés sur le réseau ?

3.2 Client et serveur TCP

Nous avons vu que le protocole TCP fonctionne, à l'inverse du protocole UDP, en mode *connecté* : il est nécessaire de commencer par établir une connexion entre le client et le serveur, et de terminer l'échange en fermant la connexion.

3.2.1 Serveur TCP

La socket à ouvrir pour une communication TCP est de type `SOCK_STREAM`.

11. Écrivez un programme serveur qui ouvre une socket TCP et l'associe au port de votre choix.
12. On passe la socket en mode écoute avec la fonction `listen()`. Cette fonction prend comme paramètre la taille du *backlog*, qui peut être 0 si on souhaite utiliser la valeur du système. Le backlog est un tableau dans lequel le système conserve les connexions en cours d'établissement, c'est-à-dire dont on a reçu le paquet SYN mais pas encore le paquet ACK. Modifiez votre programme serveur pour passer votre socket en mode écoute.
13. Le serveur accepte les connexions entrantes avec la fonction socket `accept()`. Cette fonction retourne une socket vers le client et son adresse. En effet, nous avons vu que pour libérer le port d'écoute, le serveur établit les connexions avec les clients sur un autre port que le port d'écoute. Modifiez votre programme serveur pour accepter les connexions entrantes et afficher l'adresse du client qui vient de se connecter. Petit rappel : un serveur tourne indéfiniment. Une fois qu'une connexion a été acceptée, il est donc nécessaire de retourner dans la fonction `accept()`.
14. Pour recevoir des données, on dispose de la fonction `recv()`. De même qu'avec UDP, on lui passe en paramètre la taille du tampon à utiliser. Ici La fonction retourne les données reçues. À l'inverse d'UDP, si les données ne rentrent pas dans le tampon de réception, elles ne sont pas perdues : on peut continuer à recevoir en revenant dans la fonction `recv()`. Modifiez votre serveur pour recevoir des données du client.
15. Une fois que le client a envoyé son message, on veut que le serveur lui renvoie le message `OK`. Un message est envoyé sur une socket TCP connectée avec la fonction `send()`, qui prend en paramètre le message à envoyer, par exemple une chaîne de caractères. Modifiez votre serveur pour qu'il envoie ce message au client.
16. On veut que la fermeture de la connexion soit faite à l'initiative du client. On a vu que celle-ci doit se faire des deux côtés de la connexion. Il faut donc que le serveur détecte que le client a fermé la connexion, puis que lui-même ferme sa socket. On détecte qu'une socket a été fermée dans la fonction `recv()` : celle-ci retourne comme si un message avait été reçu, mais elle retourne un message de longueur nulle. Modifiez votre programme pour que le serveur détecte la fermeture de connexion côté client et ferme sa socket de communication avec ce client.
17. Lancez votre serveur et regardez l'état de son port avec `netstat`.

3.2.2 Client TCP

Le client TCP, dans tous les cas, doit commencer par établir une connexion avec le serveur et terminer par la fermeture propre de sa connexion. Il doit d'abord créer une socket, sur laquelle seront effectuées les opérations de communication avec le serveur.

18. Un client se connecte à un serveur en utilisant sur une socket de type `SOCK_STREAM` la fonction `connect()`. Cette fonction prend comme paramètres un tuple contenant le nom de la machine sur laquelle il doit se connecter et le port sur lequel le serveur écoute. Écrivez un programme client qui se connecte à votre serveur TCP.

19. Ajoutez une instruction après le `connect()` (par exemple, un appel à `sleep()`)pour bloquer votre client après l'établissement de la connexion avec le serveur. Lancez maintenant `netstat` sur la machine sur laquelle s'exécute le serveur et constatez l'état des connexions ouvertes avec le serveur.
20. Le client doit ensuite envoyer un message au serveur. Vous pouvez par exemple envoyer le nom de votre machine, que vous obtiendrez avec la fonction `gethostname()` du module socket. Le serveur envoie ensuite un court message d'acquittement au client : le client doit donc s'attendre à recevoir un message. Modifiez votre programme client pour envoyer et recevoir un message, puis fermer la connexion avec le serveur.
21. Lancez une capture avec Wireshark pour capturer l'intégralité des échanges entre votre client et votre serveur. Quels messages sont échangés effectivement sur le réseau ?

2 Finger

Vous connaissez peut-être l'utilitaire `finger` qui, sous Unix, permet d'obtenir des informations sur un utilisateur. Pour plus d'informations, lisez l'aide en ligne (`man finger` et essayez de l'utiliser sur vos machines.

Il existe un service, décrit par la RFC 1196, permettant d'obtenir à distance ces informations sur un utilisateur d'une machine. Le but de ce TP est de réaliser une implémentation minimale (sans implémenter toutes les options) de ce service.

1 Les services sous Unix

1.1 Ports des services réseaux

Nous avons vu lors du TP précédent que les ports de numéro inférieur à 1024 n'étaient utilisables que par des programmes exécutés par le super-utilisateur. Certains ports sont "connus" (les "well-known ports" assignés par l'IANA) et l'association entre les principaux services et les ports qui leurs sont associés est définie dans le fichier `/etc/services`.

1. Quels sont le protocole et le numéro de port associés à finger ?

1.2 Scripts de démarrage

Les services sont démarrés par des scripts situés dans le répertoire `/etc/init.d/`. Regardez dans ce répertoire les scripts qui s'y trouvent et lisez-en quelques-uns pour comprendre ce qu'ils font et comment ils s'utilisent.

2. Quels sont les services disponibles sur vos machines ?
3. Comment les utilise-t-on pour lancer, arrêter ou redémarrer un service ?

1.3 Fichiers de traces et pid

Les services lancés écrivent certaines informations dans des fichiers situés sur le disque dur. On trouve notamment le *process id* des services (`pid`) dans un fichier par service situé dans le répertoire `/var/run`.

4. Regardez les fichiers contenus dans ce répertoire. Lesquels contiennent le pid des services ? Comment est enregistré le pid dans ces fichiers ?
5. Les services écrivent des informations sur ce qui se passe durant leur exécution dans des fichiers de *traces*. Ces fichiers se trouvent dans le répertoire `/var/log`. On peut journaliser les erreurs qui surviennent (log d'erreurs) ou tous types d'évènements (logs d'accès par exemple). Regardez le contenu de quelques fichiers de logs pour voir ce qu'ils contiennent.

   La plupart de ces fichiers ne sont accessibles en lecture que par le super-utilisateur. Par exemple, le fichier `/var/log/auth.log` contient les informations d'authentification des utilisateurs sur le système :

```
1 Dec  9 15:38:27 maximum sshd[20283]: Accepted publickey for coti from
      10.10.0.113 port 39763 ssh2
2 Dec  9 15:38:27 maximum sshd[20283]: pam_unix(sshd:session): session opened
      for user coti by (uid=0)
```

2 Le protocole Finger

Lorsqu'on lui passe un nom d'utilisateur, l'utilitaire `finger` affiche des informations sur cet utilisateur :

```
coti@maximum:~$ finger coti
Login: coti                             Name: Camille Coti
Directory: /users/coti                  Shell: /bin/bash
On since Mon Nov 26 11:55 (CET) on tty8 from :0
    18 days 16 hours idle
     (messages off)
On since Mon Nov 26 11:56 (CET) on pts/0 from :0.0
   17 hours 30 minutes idle
On since Wed Nov 28 11:59 (CET) on pts/1 from :0.0
   13 days 18 hours idle
On since Tue Dec 11 11:23 (CET) on pts/2 from :0.0
   40 seconds idle
On since Wed Dec 12 10:08 (CET) on pts/3 from :0.0
Last login Sun Dec  9 23:25 (CET) on pts/5 from lipn-ssh
No mail.
No Plan.
```

Le protocole `finger` permet d'obtenir ces informations *à distance* : on interroge une machine distante pour obtenir des informations sur un utilisateur [2].

Il utilise le port TCP 79. Il n'y a pas d'implémentation sur UDP. Il s'agit d'une *interface* pour l'utilitaire `finger` : la réponse obtenue est la même que si l'on avait appelé `finger` sur la machine.

3 Implémentation du protocole Finger

Nous ferons ici une implémentation minimale de ce protocole qui aura les fonctionnalités suivantes :

- Le client est appelé avec en paramètre le nom de l'utilisateur et le nom de la machine à interroger.
- Côté serveur, on appelle l'utilitaire `finger` et on renvoie sa sortie au client.
- Le client reçoit la réponse du serveur et l'affiche.

Il s'agit donc d'une application client-serveur avec un client qui interroge un serveur et un serveur qui obtient des informations sur sa machine locale pour les renvoyer au client.

6. Écrivez un programme Python `fingerd.py` serveur qui écoute sur le port TCP 7979 (étant donné que vous n'avez pas le droit d'utiliser le port 79) et reçoit de ses clients une chaîne de caractères. Vous pourrez réutiliser un programme écrit lors du TP précédent, en l'adaptant aux besoins du TP présent.
7. Lorsque le serveur reçoit une chaîne de caractères contenant le login à chercher sur le système, il construit la commande qui fera l'appel à `finger` correspondant en concaténant `finger` avec la chaîne de caractères reçus. Il s'exécute et récupère la sortie avec la fonction `getoutput()` du module `commands`. En vous aidant de la documentation Python en ligne, modifiez votre serveur pour exécuter cette commande lors de la réception d'une chaîne de caractères et récupérer sa sortie.
8. Une fois la commande exécutée et sa sortie récupérée, on l'envoie au client. Modifiez votre programme serveur pour envoyer la sortie de la commande `finger` au client.
9. Écrivez une fonction qui écrit le pid du processus en cours dans un fichier. Ce fichier pourra être, par exemple, `/tmp/finger.pid`. Appelez cette fonction à l'initialisation de votre serveur.

2. Description du protocole : `http://www.gsp.com/cgi-bin/man.cgi?section=8&topic=fingerd`

10. Lorsqu'une requête arrive, on souhaite journaliser cet appel dans un fichier de log. Ce fichier pourra être par exemple `/tmp/finger.log`. On souhaite écrire, pour chaque requête, le nom de login demandé et l'adresse du client qui a envoyé la requête. Modifiez votre serveur pour journaliser les requêtes reçues dans le fichier de logs.
11. Écrivez un client qui se connecte sur le serveur `fingerd.py`, lui envoie une chaîne de caractère, reçoit une réponse et affiche le résultat reçu. Attention, la réponse reçue du serveur peut être de n'importe quelle longueur : il conviendra de vérifier, lors d'une réception, que l'on a bien tout reçu et, le cas échéant, de ré-émettre une réception pour recevoir la suite.
12. Les arguments de la ligne de commande d'un programme peuvent être récupérés avec la fonction `getopt()` du module `getopt`. Modifiez votre client de façon à permettre à l'utilisateur de passer le nom d'utilisateur à demander et le nom de la machine serveur en paramètres, comme suit :

```
1 fingercli.py --login <login> --host <hostname>
```

3 RPC

Le but de ce TP est de présenter le modèle d'applications client-serveur RPC, et d'implémenter un serveur fournissant quelques fonctions de calcul et un client qui fait appel à ces fonctions.

1 Le modèle RPC

Les RPC (*Remote Procedure Call*) ont été introduites en 1976 dans la RFC 707, puis implémentées sous Unix par Sun en 1984 (publication de la RFC 1831). L'idée est de généraliser l'appel de fonctions non seulement dans un même programme, mais aussi entre programmes exécutés sur des machines différentes reliées par un réseau.

1.1 Présentation du modèle

Les fonctions sont implémentées sur un *serveur* et appelées par un *client*. Le client appelle la fonction du serveur en envoyant un message sur le réseau, et reste bloqué en attente de la réponse envoyée par le serveur. Des implémentations fournissent la possibilité de ne pas bloquer en attente de la réponse et de poursuivre l'exécution du client, jusqu'à ce que le résultat du calcul soit nécessaire et auquel cas il est attendu. Le client envoie les paramètres d'appels de la fonction, le calcul est effectué côté serveur et le résultat est envoyé au client.

Le modèle RPC a été étendu et adapté sous d'autres noms, mais le paradigme de programmation client-serveur reste le même : on peut citer les RMI pour Java (Remote Method Invocation), XML-RPC, SOAP, CORBA, JAX-RPC... Parmi les applications courantes utilisant des RPC, on peut citer NFS (Sun ayant développé son implémentation des RPC pour NFS), NIS. Beaucoup d'applications sur cartes à puces suivent le modèle RPC, notamment les Java cards (utilisant donc les RMI) : le calcul, consommateur en puissance de calcul et en mémoire, ressources en quantités limitées sur la puce de la carte, est effectué sur le terminal (téléphone...).

Le rôle des RPC est d'abstraire les couches basses d'une telle application : les fonctions de communications sur le réseau sont fournies par la bibliothèque ou le module RPC. Ainsi, le programmeur de l'application RPC n'a pas à écrire lui-même les communications sur le réseau et peut se concentrer sur les fonctionnalités de son application.

1.2 Appel d'une fonction distante par RPC

Concrètement, le scénario d'appel d'une fonction RPC, schématisé par la figure 8 se déroule comme suit :

1. Le client appelle la fonction distante
2. Les couches RPC du client sérialisent les paramètres passés à la fonction et les envoient au serveur sur le réseau
3. Le serveur reçoit l'appel du client ; ses couches RPC interprètent l'appel de fonction et désérialisent les paramètres
4. Le serveur appelle la fonction concernée
5. Le calcul est effectué côté serveur
6. La fonction appelée retourne une valeur
7. Les couches RPC du serveur sérialisent la valeur retournée par la fonction et l'envoient au client

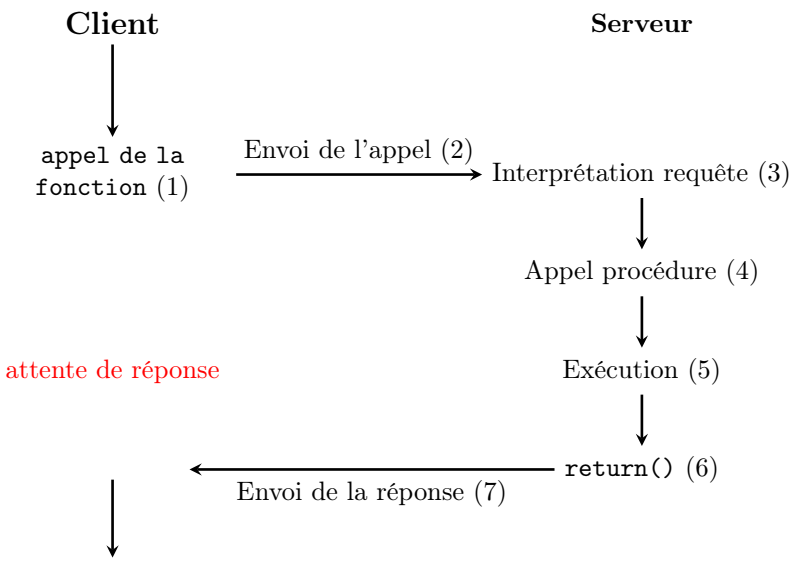


FIGURE 8 – Schéma d'exécution d'un appel RPC.

1.3 Communications en RPC

Pour être envoyées sur le réseau, les données (arguments passés à la fonction et valeur de retour ou erreur) doivent être alignées les unes après les autres afin d'être transmises sur le réseau comme une suite de bits. On parle alors de *sérialisation* ou de *marshalling*.

De plus, les programmes client et serveur peuvent s'exécuter sur des machines d'architectures différentes, utilisant une représentation des données différente : big vs little endian, entiers sur 32 vs 64 bits... Afin de rendre possible la cohabitation de machines hétérogènes, les RPC communiquent les données sur le réseau en les représentant selon un format de données intermédiaire : le *format XDR* (eXternal Data Representation).

Le module ou la bibliothèque RPC s'occupe ensuite de transmettre sur le réseau les données converties en XDR et sérialisées.

2 Le module `rfoo`

Il existe de nombreuses implémentations des RPC. L'implémentation d'origine de Sun, ONG RPC (également appelée Sun RPC), est destinée à être utilisée avec des programmes écrits en C.

En Python, nous pouvons utiliser le module `rfoo`. Il s'agit d'une implémentation très légère et performante, tout en étant simple d'utilisation. Le module `rfoo` prend en charge la sérialisation (*marshalling*) des données et leur conversion vers et depuis le format XDR.

On l'importe comme tous les autres modules :

```
1 import rfoo
```

2.1 Serveur RPC

Les fonctions implémentées par le serveur RPC doivent être implémentées dans une *classe*. Vous avez déjà rencontré le concept de classe avec les exceptions : cette classe regroupe les fonctions

fournies par votre serveur. Cette classe *hérite* de la classe `BaseHandler` fournie par `rfoo`, c'est-à-dire qu'elle dispose de ses caractéristiques et leur ajoute celles qui lui sont propres. En l'occurrence, votre classe ajoute à la classe `BaseHandler` les fonctions que votre serveur va implémenter.

On définit une classe avec le mot-clé `class` suivi du nom de la classe (par convention, souvent en minuscule avec la première lettre en majuscule), entre parenthèses le nom de la classe dont elle hérite, et enfin le signe `':'` :

```
1  class MonServeur( rfoo.BaseHandler ):
```

Tout le code de cette classe doit être dans un bloc d'instructions indenté d'un cran vers la droite. Il peut comprendre une fonction d'initialisation optionnelle, nommée `__init__`, et il comprend les fonctions implémentées par le serveur.

Les fonctions de la classe peuvent prendre les paramètres que vous jugerez nécessaires. Cependant, elles ont une contrainte : le premier paramètres de leur définition doit obligatoirement être `self`. Ce paramètre n'est pas utilisé dans les appels de fonction. Par exemple, la fonction définie comme

```
1  def addition( self, a, b ):
```

ne sera appelée qu'avec deux paramètres correspondant à `a` et `b`. Ce paramètre `self` est nécessaire pour indiquer explicitement que la fonction appartient à la classe dans laquelle elle est définie.

Des données peuvent être conservées par le service : on les conserve sous forme d'*attributs* de la classe. Pour cela, on les désigne avec ce mot-clé `self`. Par exemple, l'attribut `maVar` sera désignée par `self.maVar` et sera accessible de toutes les fonctions de la classe.

Lorsqu'une fonction `__init__()` est utilisée, il est nécessaire d'initialiser explicitement dedans un certain nombre d'attributs de la classe. Le minimum à mettre est comme suit :

```
1      def __init__( self, handler, conn ):
2          self._handler = handler
3          self._conn = conn
4          self._buffer = ''
5          self._counter = 0
6          self._server = rfoo.Server(self._handler)
7          self._methods = {}
```

On ajoute ensuite d'éventuelles initialisations d'attributs.

Lorsque des données sont conservées dans un service d'un appel à un autre, on parle de conservation d'un *état*. Par exemple, on peut avoir un compteur qui est incrémenté à chaque appel d'une fonction du service. La valeur de cette variable compteur est conservée d'un appel de fonction à un autre. Le service est donc *à état*. Un service dont les variables ne sont pas conservées d'un appel à un autre est dit *sans état*.

Le serveur est lancé avec la fonction `start_server()`, qui prend trois paramètres :

- `host` : le nom de l'hôte du client que le serveur va accepter, ou la chaîne de caractères vide si l'on ne veut pas restreindre à un seul client mais accepter des requêtes de tous les clients qui se présenteront.
- `port` : le port sur lequel le serveur va écouter. Le module `rfoo` fournit la constante `DEFAULT_PORT`.
- `handler` : le nom de la classe qui contient les fonctions qui vont être appelées pour répondre aux requêtes.

Ensuite, une fois le serveur lancé, tout se passe dans la classe qui implémente les fonctions du serveur.

2.2 Client RPC

Le client RPC doit se connecter au serveur : pour celà, il dispose de la fonction `connect()` qui prend deux arguments :

— `host` : le nom de l'hôte sur lequel le serveur est en train de tourner.

— `port` : le port sur lequel le serveur écoute.

Si la connexion ne se passe pas bien, une exception peut être soulevée. Attention à bien les gérer, elles peuvent être instructives.

La fonction `connect()` retourne un *handler*, c'est-à-dire une variable d'un type particulier utilisée pour représenter la connexion avec le serveur. Ce handler est de type `rfoo._rfoo.InetConnection`.

```
1  handler = rfoo.connect( host=host, port=port )
```

À la fin de l'échange, le client doit fermer la connexion avec la fonction `close()` sur ce handler.

Les appels de fonction distantes se font sur un objet particulier créé à partir de ce handler, appelé le *proxy*. On le crée avec la fonction `Proxy()` :

```
1  proxy = rfoo.Proxy( handler )
```

On appelle ensuite simplement les fonctions du serveur sur le proxy ainsi créé, en leur passant des paramètres entre parenthèses. Par exemple, si le serveur implémente une fonction `add()` qui prend deux variables en entrée et retourne le résultat de leur addition, on appellera :

```
1  a = proxy.add( 1, 3 )
```

Les appels de fonctions distantes peuvent soulever une exception de type `rfoo.ServerError`.

Exercice 3.1 : Calculatrice RPC

Le but de cet exercice est d'implémenter un serveur de calcul fournissant les fonctions suivantes :

— `add()` : additionne deux variables passées en paramètres et retourne leur somme.

— `mult()` : multiplie deux variables passées en paramètres et retourne leur produit.

— `diff()` : soustrait une variable passée en deuxième paramètre d'une autre variable passée en premier paramètre et retourne leur différence.

— `quotient()` : divise une variable passée en premier paramètre par une autre variable passée en deuxième paramètre et retourne leur quotient.

— `absolue()` : retourne la valeur absolue d'une variable passée en paramètre.

Si les paramètres ne sont pas de types sur lesquels on peut effectuer ces opérations mathématiques (Float ou Int), la fonction doit retourner `None`. De plus, si l'on tente de diviser par 0 dans la fonction `quotient()`, celle-ci doit également retourner None.

1. Dans un fichier `serveur.py`, créez une classe `Calcul` dans laquelle vous devez implémenter les fonctions du serveur.
2. Complétez le programme serveur afin de lancer le serveur RPC lorsque le programme est exécuté. Le serveur doit accepter des connexions depuis n'importe quel client, et écouter sur un port de votre choix.
3. Écrivez un client qui se connecte à votre serveur en utilisant le module RPC `rfoo` et appelle à distance les fonctions que celui-ci implémente.

Exercice 3.2 : Annuaire en ligne

Le but de cet exercice est d'implémenter une version en ligne du répertoire que nous avons vu lors du TP 2 du module M02. Il s'agit maintenant d'un service *à état* : le contenu du répertoire

doit être conservé d'un appel à l'autre. Le répertoire est stocké sous forme d'un dictionnaire, de la même façon que nous l'avons vu dans le TP 2.

1. Écrivez un serveur RPC utilisant une classe `Annuaire` qui implémente les fonctions suivantes :
   - `__init__` : initialise la classe service et notamment le répertoire sous forme d'un dictionnaire vide ;
   - `ajouterEntree` : prend en paramètre un nom et un numéro de téléphone et les ajoute au répertoire ;
   - `trouverNumero` : prend en paramètre un nom et retourne le numéro de téléphone correspondant à ce nom, ou la chaîne de caractères vide si ce nom n'est pas présent dans le répertoire ;
   - `nbNumeros` : retourne le nombre d'entrées dans le répertoire ;
   - `getAll` : retourne tout le contenu du répertoire (et donc le répertoire lui-même) ;
   - `supprimerEntree` : prend en paramètre un nom et supprime l'entrée correspondante du répertoire ;
   - `supprimerTout` : vide le répertoire en supprimant toutes les entrées contenues dedans.
2. Écrivez un client qui affiche le menu suivant et propose à l'utilisateur de saisir une action à effectuer, et appelle par RPC la fonction du serveur correspondante pour manipuler le répertoire contenu dans le serveur.

```
Choix de l'action a effectuer :
1: Ajouter une entree dans le repertoire
2: Afficher le numero de telephone d'une personne
3: Afficher le nombre de numeros enregistres dans le repertoire
4: Afficher le contenu de tout le repertoire
5: Supprimer du repertoire une personne et son numero
6: Effacer tout le contenu du repertoire
0: Quitter le programme
-->
```

4 Transfert de fichiers avec FTP

1 Le protocole FTP

Le protocole FTP est un protocole de transferts de fichiers qui fonctionne au-dessus de TCP. Il utilise deux canaux de communication entre le client et le serveur, et donc deux ports :

— Le canal de *contrôle*, sur lequel sont transmises les commandes du protocole, sur le port 21
— Le canal de *données*, sur lequel sont transmises les données elles-mêmes, sur le port 20

Il permet d'échanger des fichiers de façon très simple, selon cette séquence d'actions :

— Ouverture de connexion avec login et mot de passe
— Transfert de fichiers
— Fermeture de connexion

Le but de ce TP est d'implémenter un sous-ensemble du protocole FTP dans un client puis dans un serveur.

1.1 Architecture

Le client et le serveurs sont constitués de deux modules :

— Le *Protocol Interpreter* (PI), qui interpète les commandes reçes du client sur le canal de contrôle
— Le *Data Transfer Process* (DTP), qui effectue les échanges de données eux-mêmes (envois et réceptions) sur le canal de données

Le PI commande le DTP : quand le client envoie une commande d'échange de données sur le canal de contrôle, le PI appelle le DTP pour lui demander de participer à cet échange de données.

Le client a également, en outre, une interface d'interaction avec le client, qui peut être graphique ou en ligne de commandes.

Dans un fonctionnement client-serveur, l'architecture utilisée par le protocole FTP peut-être représenté comme sur la figure 9. Chaque machine, cliente comme serveur, dispose de son propre système de fichier. Les fichiers sont transmis entre le client et le serveur (dans un sens ou dans l'autre) par les DTP sur le canal de données, sous le contrôle du PI.

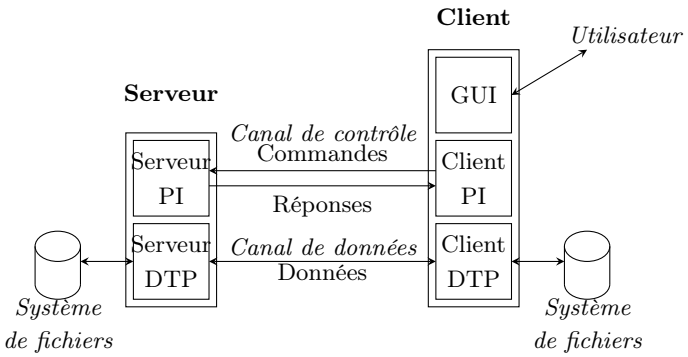


FIGURE 9 – Fonctionnement de FTP dans un échange client-serveur

Les fichiers peuvent également être transmis entre deux serveurs, sur demande du client. Dans ce cas, le PI du client ouvre un canal de contrôle avec chaque serveur, et les serveurs ouvrent un canal de données entre eux deux. Ce mode de fonctionnement est représenté par la figure 10.

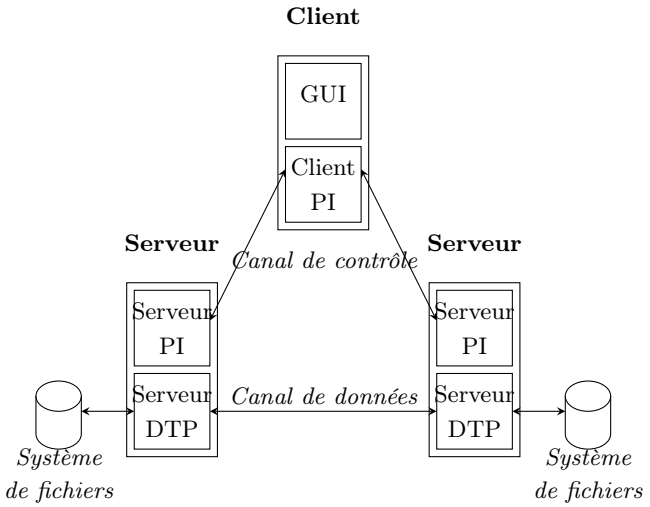


FIGURE 10 – Fonctionnement de FTP dans un échange serveur-serveur

1.2 Protocole

Le protocole TCP fonctionne *en mode texte*. Les commandes et les réponses échangées entre les PI sont envoyées en étant encodées en ASCII 7 bits. Les communications sur le canal de données peuvent se faire en binaire ou en ASCII, selon le paramétrage demandé par le PI.

Les commandes TCP sont consistuées de quelques lettres majuscules. On dispose des commandes de contrôle d'accès suivantes :
— USER : identification de l'utilisateur
— PASS (password) : mot de passe de l'utilisateur (suit immédiatement une commande USER)
— CWD (change working directory) : changement de répertoire dans l'arborescence distante
— CDUP (change directory up) : remonter dans l'arborescence distante
— QUIT : fin de la session
Certains serveurs FTP autorisent les utilisateurs anonymes. Pour cela, un nom d'utilisateur particulier est utilisé : *anonymous*. N'importe quelle chaîne de caractères est ensuite acceptée comme mot de passe.

Parmi les commandes de service, on trouve les suivantes :
— RETR (retrieve) : copie sur la machine locale
— STOR (store) : copie sur la machine distante
— LIST : liste des fichiers distants
— HELP : liste des commandes
Enfin, le PI permet de paramétrer le transfert qui sera effectué entre les DTP. On a les commandes de paramétrage suivantes :
— TYPE : type de format d'échange des données (A=ASCII)
— STRU (structure) : structures échangées (F=FILE)
— MODE : mode de transfert des données (S=STREAM)
— PORT : numéro de port auquel le DTP-serveur doit se connecter

FTP fonctionne selon le mode "requête-réponse" : lorsque le client envoie une commande (requête), le serveur lui répond. Les réponses commencent par un code composé de trois chiffres, constitué comme suit :

— 1er chiffre : status de la requête
— 2eme chiffre : référence de la réponse
— 3eme chiffre : spécifie la réponse

Par exemple, quelques codes de réponse existant :

— 1xx : réponse préliminaire positive
  — 125 : canal de données déjà ouvert ; début du transfert
— 2xx : réponse positive de réalisation
  — 200 : ok
— 3xx : Réponse intermédiaire positive (action supplémentaire nécessaire)
  — 331 : nom d'utilisateur ok ; entrer le mot de passe.
— 4xx : Réponse négative de réalisation (échec temporaire)
  — 425 : erreur d'ouverture du canal de données
— 5xx : Réponse négative permanente
  — 504 Commande non implémentée pour ce paramètre.

Le deuxième chiffre donne des informations sur la nature de la réponse :

— x0z : Syntaxe (erreur de syntaxe ou syntaxe correcte mais ne se référant à aucune fonction connue ou implémentée)
— x1z : Information (réponse à des demandes d'information)
— x2z : Connexions (réussite ou problème de connexion)
— x3z : Identification et authentification
— x4z : Non encore spécifiée.
— x5z : Système de fichiers

Le protocole FTP est décrit intégralement par la RFC 959 pour les commandes d'échanges, puis des extensions ont été faites au protocole (RFC 3659), notamment en matière de sécurité (RFC 3659), de prise en charge de IPv6 (RFC 2428) et d'autres types d'encodage (RFC 3659).

1.3 Exemple de discussion avec un serveur en Telnet

On peut donc utiliser Telnet pour effectuer une discussion basique avec un serveur.

```
1  coti@thorim:~$ telnet ftp.free.fr 21
2  Trying 212.27.60.27...
3  Connected to ftp.proxad.net.
4  Escape character is '^]'.
5  220 Welcome to ProXad FTP server
6  USER anonymous
7  331 Please specify the password.
8  PASS touchmytralala
9  230 Login successful.
10 QUIT
11 221 Goodbye.
12 Connection closed by foreign host.
```

On voit qu'un client telnet s'est connecté au serveur `ftp.free.fr` sur le port 21 (correspondant au canal de commandes sur FTP). Après les affichages d'usage de Telnet, le serveur a répondu positivement ligne 5 : le code de réponse 220 indique que tout va bien. On doit alors s'*authentifier* : on envoie la commande `USER` avec un nom d'utilisateur, ici *anonymous*. On a alors le code de réponse 331 : jusque là tout va bien (l'utilisateur existe sur le serveur) mais une action est demandée relative l'authentification : il faut fournir un mot de passe. On envvoie alors un mot de passe avec la commande `PASS` : n'importe quelle chaîne de caractères est acceptée. L'authentification est correcte : on reçoit le code de réponse 230. On se déconnecte du serveur avec la commande `QUIT`.

2 Écriture d'un client FTP

Écrivez un client FTP qui se connecte à un serveur [3], s'authentifie de manière anonyme, affiche la liste des fichiers, se déplace dans un répertoire et télécharge un fichier. Le DTP et le PI peuvent être implémentés dans le même script Python.

Si votre client respecte correctement le protocole TCP, il sera capable de discuter avec n'importe quel serveur respectant ce protocole. C'est l'intérêt des protocoles réseaux : des entités différentes respectant toutes un même protocole donné sont capables de se comprendre.

Attention : les chaînes de caractères que vous envoyez doivent se terminer par `'\r\n'`.

3 Écriture d'un serveur TCP

Écrivez un serveur TCP qui comprend un sous-ensemble des commandes disponibles dans le protocole TCP. Votre serveur devra être capable d'authentifier un utilisateur anonyme, de changer de répertoire et de transférer des fichiers entre le client et le serveur (dans les deux sens). Le DTP et le PI peuvent être implémentés dans le même script Python.

Pour accéder au système de fichiers local, vous pourrez notamment utiliser les fonctions du module `os`.

Pour transférer un fichier, il est nécessaire de l'ouvrir puis d'envoyer ce que l'on a lu dans la socket. Pour recevoir un fichier, on fait la même chose de l'autre côté dans l'ordre inverse : on reçoit les données, puis ont les écrit dans un fichier.

Une fois que cela fonctionne en lisant tout le fichier avant de l'envoyer, vous pourrez améliorer votre implémentation en pipelinant la lecture et l'envoi du fichier d'une part, la réception et l'écriture d'autre part. Vous pourrez lire une partie du fichier et l'envoyer, puis lire la suite du fichier, et ainsi de suite. Même chose côté réception : vous pourrez recevoir ce qui vient sur la socket, l'écrire dans le fichier, recevoir ce qui vient ensuite, et ainsi de suite. Les fonctions d'entrées-sorties étant asynchrones, cela permet de mettre en place un peu de recouvrement entre la lecture/écriture du fichier et l'émission/réception sur le réseau.

3. Vous pourrez par exemple vous connecter à `ftp.debian.org`

5 Interaction avec un service web : exemple de l'API Twitter

Compétences attendues à l'issue de cette séance :
— notions sur les architectures REST ;
— lire des données au format JSON ;
— utiliser l'API Twitter ;
— programmer un client Twitter simple en Python.

Vous rédigerez un compte rendu, sur lequel vous indiquerez la réponse à chaque question, vos explications et commentaires (interprétation du résultat, commandes ou outils utilisés, référence au code source produit qui sera fournit par ailleurs).

1 Twitter et son API

Twitter (`http://twitter.com`) fournit un service permettant aux utilisateurs d'entrer en relation sur le web en échangeant de courts messages (textes d'au plus 140 caractères). Chaque utilisateur voit un fil d'actualité où apparaissent les derniers messages postés par les utilisateurs qu'il a décidé de suivre.

Pour effectuer ce TP, vous devrez disposer d'un compte sur Twitter. Si vous n'en avez pas déjà un, ou si vous ne souhaitez pas l'utiliser pour des tests, vous devrez en créer un nouveau (c'est rapide et gratuit).

Outre l'interface web classique, Twitter est utilisable via des applications (en général sur téléphone ou tablette). Des applications tierces peuvent aussi utiliser Twitter via des services regroupés en une API (*Application Programming Interface*) facile d'emploi. Cette API est documentée sur `https://dev.twitter.com/docs`

Dans un premier temps, nous allons utiliser directement l'API de Twitter, afin de se familiariser avec les formats de données utilisés.

Ensuite, nous utiliserons un module Python qui permet un usage bien plus facile des services.

1.1 Introduction à l'API REST

REST est une architecture simple très utilisée pour les services sur le Web. L'acronyme REST signifie *Representational State Transfer*. Les principales caractéristiques d'une architecture REST sont issues de l'architecture Web classique (protocole HTTP) et sont en résumé :
— Client/Serveur
— Sans état : chaque requête est indépendante des précédentes. Cela simplifie grandement la gestion côté serveur.
— Cache : les réponses sont souvent associées à une période de validité qui permet aux clients d'éviter de renouevler leurs demandes et surtout autorise la mise en place de caches (sur des proxy et/ou sur les clients).
— Interface uniforme : les ressources sont identifiées de façon unique (par exemple ID numériques).

Les applications Web REST utilisent des requêtes HTTP :
— `GET` permet au client d'obtenir des informations du serveur. L'état du serveur n'est jamais modifié. Exemple : informations sur un utilisateur, ou lecture d'un message.
— `POST` permet au client de modifier l'état du serveur, en général en lui envoyant des données. Exemple : poster un message.

1.2 Formats de données : JSON

Les données échangées entre le client et le serveur sont dites *sérialisées* : il s'agit au bout du compte d'une suite d'octets, qui doit être interprétée de la même façon par les deux entités. Dans la plupart des protocoles modernes, on échange des données au format texte. Les données manipulées par les applications (objets, tableaux, pointeurs, nombres) doivent alors être *sérialisées*, sous la forme d'une suite de caractères.

Les deux principaux formats d'échange utilisés dans les applications Web sont le XML, dont nous ne parlerons pas ici, et le JSON. JSON est l'acronyme de *Javascript Object Notation*. Le format JSON est simplement la notation JavaScript pour déclarer les données.

Exemple : soit un objet avec deux champs "nom" et "age". Il pourrait être représenté par le texte JSON suivant :

```
{
  "nom" : "Dupont",
  "age" : 42
}
```

La notation JSON a l'avantage d'être simple à lire et écrire, facile à interpréter pour les humains, et pas trop verbeuse (elle est en générale plus concise que le XML correspondant). D'autre part, les clients Web écrits en JavaScript peuvent interpréter directement les données JSON qu'ils reçoivent.

1.3 Requêtes simples sans authentification

Un certain nombre d'appels à l'API Twitter permettent d'obtenir des informations sans s'identifier (en effet, contrairement à d'autres réseaux sociaux, la plupart des interactions sur Twitter sont destinées à être publiques).

Attention cependant, ces requêtes appartiennent à l'API version 1 qui est obsolète (*deprecated*) et ne sera bientôt plus supportée. Nous verrons plus loin comment utiliser la nouvelle API (version 1.1).

Par exemple, l'appel :
`http://api.twitter.com/1/users/show.json?screen_name=ReseauxTelecom`

permet d'obtenir des informations sur le compte utilisateur indiqué (ici "`ReseauxTelecom`"). Les informations sont fournies au format JSON, voici un extrait des données renvoyées dans la réponse :

```
1  {
2    "id" : 987149456,
3    "id_str" : "987149456",
4    "name" : "IUT Reseaux Telecoms",
5    "screen_name" : "ReseauxTelecom",
6    "location" : "Université Paris 13",
7    "url" : "http : \/\/www.iutv.univ-paris13.fr\/reseaux-telecommunications.html",
8    "description" : "Département Réseaux et Télécommunications de l'IUT de
         Villetaneuse",
9    "protected" : false,
10   "followers_count" : 6,
11   "friends_count" : 15,
12   "created_at" : "Mon Dec 03 18:17:24 +0000 2012",
13   "favourites_count" : 0,
14   "statuses_count" : 13,
15   "lang" : "fr",
16
17   "status" : {
18      "created_at" : "Wed Jan 09 20:45:33 +0000 2013",
19      "id" : 289110685168832513,"id_str" : "289110685168832513",
```

```
20      "text" : "Un message d'essai",
21      "source" : "web",
22      "truncated" : false,
23      "retweet_count" : 0,
24      "favorited" : false,
25      "retweeted" : false,
26    },
27
28    "profile_background_color" : "C0DEED",
29    "profile_image_url" : "http:\/\/a0.twimg.com\/profile_images\/2928118435\/
          a532c7bea82d493467aa1923ec49b3b6_normal.jpeg",
30    "profile_text_color" : "333333",
31     (...)
32    }
```

Question 1 : Qu'est-ce que le `screen_name` ? Comment Twitter identifie-t-il les comptes utilisateurs ?

Question 2 : Que contient le champ `status` ?

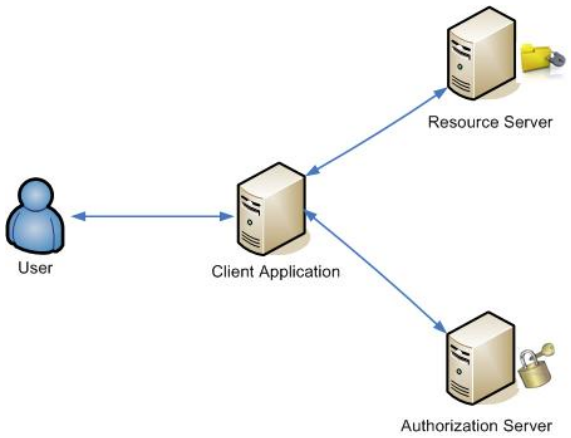
Question 3 : Combien d'amis (`friends`) a notre compte de test ? A quoi correspond un "ami" dans Twitter ?

Question 4 : Combien de suiveurs (`followers`) a-t-il ?

1.4 OAuth

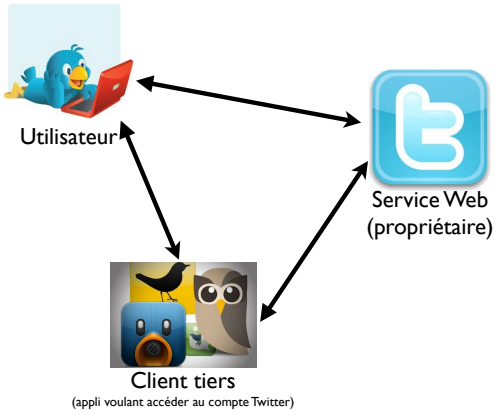
La nouvelle API de Twitter (version 1.1) nécessite une authentification préalable du client.

Plutôt que de requérir l'envoi un *login* et d'un mot de passe, Twitter utilise un mécanisme nommé OAuth pour contrôler l'accès des clients. Ce mécanisme évite d'avoir à fournir les identifiants (nom et mot de passe) du compte que l'on veut utiliser. C'est important, car dans la plupart des cas, le programme qui veut utiliser Twitter s'exécute sur une machine tiers, à laquelle l'utilisateur ne veut pas fournir d'informations sensibles comme son mot de passe.



OAuth fait l'objet de la RFC 5849 (`http://tools.ietf.org/html/rfc5849`) et vous pourrez trouver des explications détaillées ici : `http://hueniverse.com/oauth/guide/`. (Il existe aussi deux versions du protocole OAuth, 1.0a et 2.0, en cours de finalisation [4]).

L'utilisation de OAuth est plus complexe que l'authentification classique par *login*/mot de passe, mais apporte une réelle flexibilité à l'utilisateur final et permet l'interaction d'applications opérées par des acteurs différents.



Les principes d'OAuth sont les suivants :

— L'utilisateur final autorise une application tierce (dite "cliente") à accéder à (une partie de) ses données gérées par un service web (le propriétaire de la ressource, ici Twitter). Cet utilisateur ne communique pas son mot de passe à l'application cliente.

— Au lieu de demander le mot de passe, le client redirige l'utilisateur vers le propriétaire, et l'utilisateur indique au propriétaire qu'il autorise l'accès au client.

— Le client est alors notifié par le propriétaire, qui lui transmet un code d'authorisation pour l'opération à effectuer.

— Le client présente alors le code d'autorisation, accompagné de l'identifiant du client, et reçoit un *access token*.

1.5 Déclaration d'une application sur Twitter

Avant de pouvoir utiliser OAuth avec twitter, il faut déclarer votre application cliente, via un formulaire très simple. Connectez vous à votre compte Twitter, puis allez sur la page `https:`

4. Voir `http://tools.ietf.org/html/draft-ietf-oauth-v2-10`

`//dev.twitter.com/apps/new`. Vous devez simplement donner un nom (unique au monde) à votre application (par exemple "EssaiXXX", où XXX est votre nom).

A la suite de cette déclaration, vous allez récupérer les deux identifieurs OAuth que votre code client devra utiliser :

— `Consumer key`
— `Consumer secret`

1.6 Installation des outils nécessaires

Il faut installer le module `twitter`, disponible sur `http://pypi.python.org/pypi/twitter/`.

Sur la même page, vous avez des exemples d'utilisation.

Question 5 : Installer le module `twitter` sur votre machine. Décrivez sur votre compte rendu les étapes suivies.

Question 6 : Écrire le code pour obtenir en Python les informations sur l'utilisateur `ReseauxTelecom`

Indication : utiliser la méthode `users.show(screen_name=xxx)`

Notez que les méthodes Python renvoient directement des objets Python, le décodage du JSON est automatiquement effectué.

1.7 Envoi d'un tweet

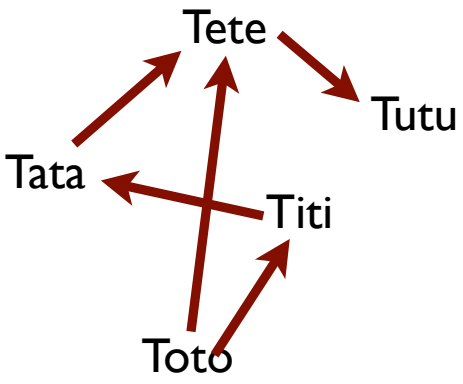
Question 7 : Écrire le code pour envoyer un tweet sur votre compte.

Attention : pour pouvoir envoyer un tweet, votre application doit avoir le droit d'écriture (mode "read/write"). Par défaut, elle a été créée en lecture seule, il faut donc retourner sur `https://dev.twitter.com/apps` pour changer ce réglage.

1.8 Construction du graphe social autour d'un utilisateur

Question 8 : Écrire un programme en python qui construise et enregistre dans un fichier le "graphe social" d'un utilisateur donné (qui sera identifié par son `screen_name`), c'est à dire la liste des relations entre cet utilisateur et ses amis, et de ses amis entre eux.

Par exemple, si on a 5 utilisateurs Tata, Tete, Titi, Toto et Tutu, comme sur la figure :



où les flèche indiquent qui suit qui (“Titi” suit “Toto” et est suivi par “tata”), on veut générer un fichier texte contenant une ligne par flèche, comme ceci :

```
Toto Tete
Tete Tutu
Toto Titi
Titi Tata
Tata Tete
```

(notez que l'ordre des lignes n'a aucune importance).

Question bonus : trouver et mettre en œuvre des outils pour dessiner le graphe stocké dans le fichier obtenu.