Grilles de Calcul et Cloud
GCC

Travaux pratiques

Institut Galilée — Département Informatique — M2 PLS

Camille Coti
camille.coti@lipn.univ-paris13.fr

Première partie

Systèmes distribués

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. Par conséquent, il ne fournit pas à l'émetteur d'un message de garantie que le destinataire a bien reçu le message, ni que celui-ci n'a pas été altéré.

Comme montré sur 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.

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 en voyant chaque port comme une ligne.

Le numéro de port est un entier non signé codé sur deux octets : les numéros possibles sont donc compris entre 0 et 65.535. 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`.

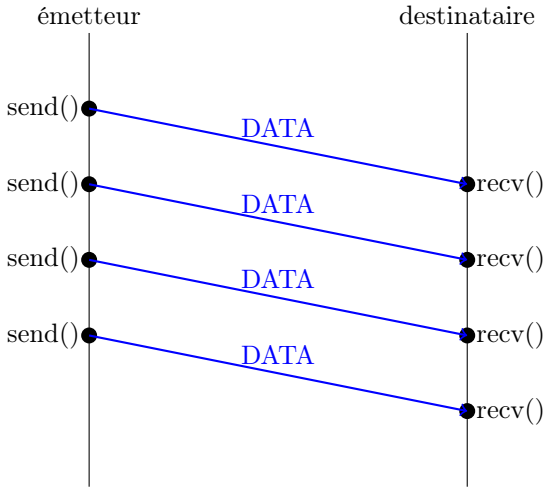

FIGURE 2 – Envoi de données sur UDP entre deux processus.

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.

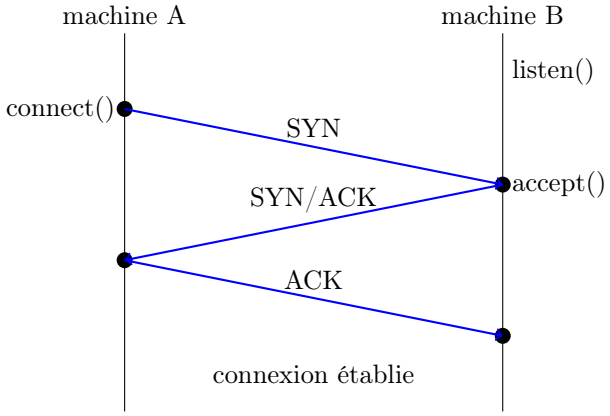

FIGURE 4 – Établissement d'une connexion en TCP : la triple poignée de main.

Les envois de paquets sont acquittés par des paquets `ACK`. Cela pose des problèmes immédiats. L'acquittement de tous les paquets génère un grand nombre de paquets supplémentaires : on double alors le nombre de paquets envoyés sur le réseau (N paquets protocolaires pour N paquets envoyés par les processus). On augmente alors la charge du réseau, au risque de l'engorger. De plus, si l'on bloque l'émission du paquet suivant tant que l'acquittement d'un paquet n'a pas été reçu, on ralenti considérablement la vitesse des communications.

Pour contourner ces problèmes, on 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.

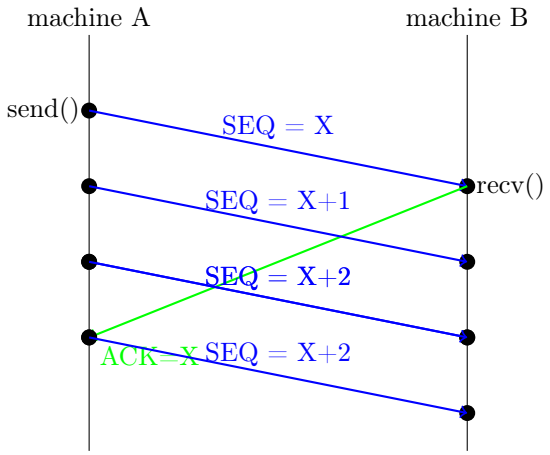

FIGURE 5 – Acquittement des paquets envoyés par TCP.

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.

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.

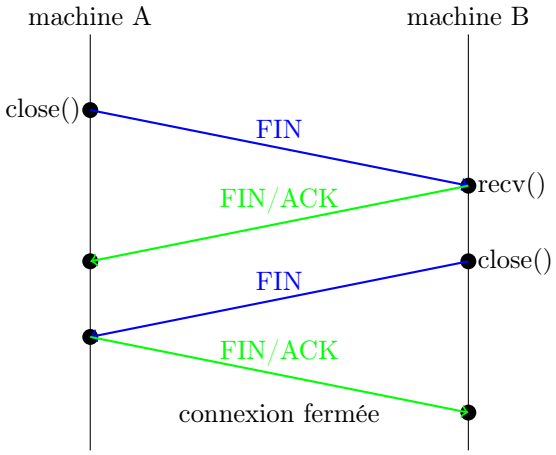

FIGURE 6 – Fermeture d'une connexion TCP : la poignée de main bidirectionnelle.

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.

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.

<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

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 ?

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`.

10. Écrivez un programme serveur qui ouvre une socket TCP et l'associe au port de votre choix.
11. 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.
12. 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()`.
13. 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.
14. 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.
15. 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.

16. 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.

17. 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.
18. 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.
19. 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.

2 Interrogation d'un service distant : fingerd

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 Le protocole Finger

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 Finger

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

```
1  coti@maximum:~$ finger coti
2  Login: coti                             Name: Camille Coti
3  Directory: /users/coti                  Shell: /bin/bash
4  On since Mon Nov 26 11:55 (CET) on tty8 from :0
5      18 days 16 hours idle
6       (messages off)
7  On since Mon Nov 26 11:56 (CET) on pts/0 from :0.0
8     17 hours 30 minutes idle
9  On since Wed Nov 28 11:59 (CET) on pts/1 from :0.0
10    13 days 18 hours idle
11 On since Tue Dec 11 11:23 (CET) on pts/2 from :0.0
12    40 seconds idle
13 On since Wed Dec 12 10:08 (CET) on pts/3 from :0.0
14 Last login Sun Dec  9 23:25 (CET) on pts/5 from lipn-ssh
15 No mail.
16 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.

2 Implémentation du protocole Finger

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

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

— 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.

2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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 Messagerie et ordre causal

La messagerie électronique est un système distribué : les utilisateurs font appel à plusieurs serveurs pour les envois et le réceptions des messages. La circulation des messages entre les clients de messagerie est donc concernée par les problématiques des systèmes distribués, notamment concernant l'ordre des messages.

1 Architecture et mini protocole

Notre service de messagerie est composé d'un ensemble de *serveurs* qui communiquent entre eux. Chaque client fait appel à un serveur pour émettre et recevoir des messages. Plusieurs clients peuvent avoir le même serveur. Un client ne se connecte qu'à son serveur ; les serveurs communiquent entre eux pour acheminer le message du serveur de l'émetteur au serveur du destinataire. Les serveurs assurent donc l'envoi et la réception des messages.

Lorsqu'un client Afida veut envoyer un message à un client Brandon :
- Afida se connecte à son serveur, et transmet son message ;
- Le serveur d'Afida examine le destinataire du message et détermine à quel serveur il faut l'envoyer ;
- Le serveur d'Afida transmet le message au serveur approprié ;
- Brandon se connecte à son serveur et récupère le message.

On suppose que les canaux de communications et les serveurs ont la propriété *FIFO* : un message ne peut pas en "doubler" un autre si ils prennent le même chemin. Ainsi, si Afida envoie deux messages à Brandon, il les recevra dans l'ordre dans lequel Afida les a envoyés. Par ailleurs, les canaux de communications sont *asynchrones*.

Lorsqu'on a trois serveurs impliqués dans la discussion, la propriété FIFO sur les canaux et les serveurs ne suffit plus à assurer que les messages sont reçus dans l'ordre. Prenons par exemple la situation suivante, avec trois protagonistes et le scénario représenté figure 8. Afida envoie un message à Brandon et Nabilla, et Brandon répond à Afida et Nabilla. Les messages de Afida et Brandon n'ont aucune raison d'arriver à Nabilla dans un ordre en particulier : si la communication entre *Sa* et *Sn* est lente sur l'envoi du premier message, Nabilla peut recevoir le message d'Afida après celui de Brandon. Les messages ne sont donc pas ordonnés causalement.

(a) Premier message (b) Réponse

FIGURE 8 – Scénario d'envoi de messages à trois protagonistes.

Implémentez sur TCP le système de messagerie décrit à la section précédente. Pour simplifier, les serveurs utiliseront une seule socket d'écoute pour recevoir des connexions des clients et des autres serveurs. L'IP et le port de cette socket d'écoute sera connue à l'avance, et ce pour tous les serveurs. Ainsi, chaque client connaît l'IP et le port de son serveur, et chaque serveur connaît l'IP et le port de tous les autres serveurs.

Toujours pour simplifier, vous pourrez considérer que chaque serveur de messagerie est en charge d'un *domaine*. Les adresses utilisées seront de la forme `destinataire@domaine`. Ainsi, quand un serveur reçoit un message à transmettre, il regarde le domaine de destination et décide à quel serveur il doit le transmettre.

Au lancement, les serveurs obtiennent la liste des IP et des ports des serveurs. Vous pouvez les passer dans un fichier dont le chemin est passé en paramètre de la ligne de commande et lu au démarrage du serveur. À chaque serveur est également associé le domaine dont il est en charge. Par exemple, vous pourrez utiliser un fichier utilisant le format suivant :

```
localhost:5556:tf1.fr
localhost:5557:canalplus.com
```

Par exemple, votre serveur peut être lancé avec les paramètres suivants :

```
./serveur.py <port> <domaine> <fichier de serveurs>
```

Le client fonctionnera en mode interactif pour les opérations d'envoi de mail et de relevé de boite. Il pourra être appelé de la façon suivante :

```
./client.py <adresse du serveur> <port du serveur>
```

Si vous choisissez de travailler en Python, vous pourrez par exemple utiliser la fonction suivante pour le lire et séparer les informations qu'il contient :

```
def lireServeurs( fichier ):
    serveurs = []
    fd = open( fichier, 'r' )
    for ln in fd:
        m = ln.strip()
        if m != '':
        host,port,domaine = m.split( ':' )
        serveurs.append( (host, port, domaine) )
        fd.close()
    return serveurs
```

Il vous sera certainement plus simple de stocker les serveurs dans un dictionnaire indexé par le nom de domaine traité par chaque serveur.

```
def dicoServeurs( servlist ):
    serveurs = {}
    for s in servlist:
        serveurs[s[2]] = (s[0], int(s[1]))
    return serveurs
```

Vous pouvez stocker les mails des boîtes locales dans un dictionnaire indexé par l'adresse du destinataire, et dont les valeurs sont les listes de messages.

```
def stockerMail( mess, dst ):
    global mailbox
    if dst in mailbox.keys():
        mailbox[dst].append( mess )
    else:
        mailbox[dst] = [mess]
    print mailbox
    return
```

La consultation, côté serveur, consistera donc à envoyer la liste des mails d'une adresse, avec un petit protocole pour séparer les messages entre eux et pour signifier la fin de la liste.

```
def consulterMail( dst, sock ):
    global mailbox
    if dst in mailbox.keys():
        for m in mailbox[dst]:
            envoiMessage( sock, m+":" )
        envoiMessage( sock, ":" )
    else:
        envoiMessage( sock, "::" )
    return
```

Et, côté client, à recevoir cette liste.

```
def relever( serveur, adresse ):
    releverMail( serveur, adresse )
    mess = recvMessage( serveur )
    while not mess[-2:-1].startswith( "::" ):
        print mess
        mess = commun.recvMessage( serveur )
    return
```

Vous aurez besoin d'un mini-protocole pour les messages envoyés dans votre infrastructure. Par exemple, vous pourrez envoyer l'adresse de destination puis le message en les séparant par un caractère qui n'est pas compris dans les adresses : par exemple, `:`.

Attention, vos clients auront deux sortes d'interactions avec les serveurs : pour envoyer un mail, et pour consulter une boite mail. Pour les différencier, vous pourrez ajouter un caractère protocolaire entre l'adresse et le message, en séparant par `:`.

```
def envoiMessage( s, message ):
    s.send( message )
    return

def construireMail( message, destinataire ):
    return destinataire + ":S:" + message

def construireConsultation( destinataire ):
    return destinataire + ":R:"

def envoyerMail( serveur, message, destinataire ):
    m = construireMail( message, destinataire )
    envoiMessage( serveur, m )
    return

def releverMail( serveur, destinataire ):
    m = construireConsultation( destinataire )
    envoiMessage( serveur, m )
    return
```

En réception, le serveur doit alors séparer l'adresse de destination du message.

```
def recvMessage( s ) :
    BUFFSIZE = 1024
    msg = s.recv( BUFFSIZE )
```

```
4       return msg
5
6   def recevoirMail( sock  ):
7       mess = recvMessage( sock )
8       # Separer l'adresse du destinataire du message
9       dst,t,m = mess.split( ":", 2 )
10      return dst, t, m
```

Une fois le serveur destinataire déterminé en utilisant la liste de serveurs, le serveur qui a le message décide soit de le garder (car le domaine de destination est celui dont il a la charge), soit il le transfère à un autre serveur.

```
1   def localOrNot( dst ):
2       loc,dom = dst.split( "@" )
3       moi = sys.argv[2]
4       return dom == loc:
5
6   def transfererMail( message, dest, serveurs ):
7       # separer le domaine de l'adresse
8       loc,dom = dest.split( "@" )
9       # comment contacter le serveur correspondant
10      ip, port = serveurs[dom]
11      dst = ouvreSocketConnect( ip, port )
12      # transferer le message
13      envoyerMail( dst, message, dest )
14      return
```

2 Utilisation d'horloge logique

On peut assurer l'ordre causal des messages en réception en utilisant une horloge logique. L'horloge est insérée dans les messages et utilisée pour les mettre en ordre en réception.

Pourquoi une horloge scalaire (horloge de Lamport) ne suffirait-elle pas à assurer l'ordre causal dans le système ? Trouvez un contre-exemple sur un scénario schématisé par un dessin. Une horloge scalaire assure-t-elle l'ordre FIFO entre deux éléments du système (client-serveur ou serveur-serveur) ? Mêmes questions avec une horloge vectorielle.

Modifiez le mini-protocole de circulation des messages pour y insérer une horloge matricielle. L'horloge sera insérée par les éléments du système de messagerie et lue par le client pour mettre en ordre les messages reçus.

Mettez en place les opérations sur ces horloges dans le client et dans le serveur : incrément de l'horloge locale, mise à jour de l'horloge matricielle, comparaison de deux horloges.

Pour tester si votre système fonctionne, vous pourrez ajouter des temps d'attente dans le traitement de certains messages.

4 Exclusion mutuelle I

Le but de cet exercice est de mettre en place un système d'exclusion mutuelle centralisé de type clients-serveur.

1 Infrastructure

Mettez en place une infrastructure avec un serveur et plusieurs clients. Les clients doivent rester connectés indéfiniment au serveur (jusqu'à ce que vous les tuiez).

2 Exclusion mutuelle

Vous souhaitez que vos processus écrivent dans un fichier unique. Vous avez la chance de disposer pour cela d'un répertoire partagé sur votre compte Unix, grâce au système de fichiers en réseau de type NFS. Cependant, il ne faut pas que tous les processus écrivent dedans en même temps.

Vous avez donc besoin d'assurer une exclusion mutuelle entre vos processus. L'ordre est arbitraire. Implémentez l'algorithme d'exclusion mutuelle centralisé pour assurer l'exclusion mutuelle entre vos processus. Le serveur décidera d'attribuer l'entrée en section critique selon une politique de "premier arrivé, premier servi".

5 Exclusion mutuelle II

Le but de cet exercice est de mettre en place une infrastructure de processus reliés en anneau et anonymes, et d'assurer une exclusion mutuelle entre ces processus.

1 Lancement des processus

Pour lancer un processus sur une machine distante, le processus lanceur effectue une connexion SSH sur la machine distante pour lancer le processus distant. Celui-ci se connecte à un autre processus, pour *rejoindre l'infrastructure*. On doit donc lui passer une adresse IP et un port en paramètres.

SSH permet de lancer une commande à distance. Vous pourrez pour cela utiliser la commande `Popen` du module `subprocess` si vous choisissez de travailler en Python. Le processus distant que l'on lancera ici sera un *démon*. Pour obtenir un démon, le processus doit faire un `fork()` et le processus père doit mourir. Si vous avez choisi de faire le TP en Python, vous pourrez utiliser, par exemple :

```
1 def main():
2     pid = os.fork()
3     if pid > 0:
4         sys.exit( 0 )
5     else:
6         daemon()
7     return
```

Écrivez un programme distribué qui :
- Ouvre une socket TCP en écoute, et découvre sur quel port elle écoute ;
- Lance un autre processus, sur une machine distante, via SSH ;
- Le processus distant est un démon qui se connecte au processus lanceur et lui envoie un petit message ;
- Le processus lanceur affiche ce message.

Vous pourrez utiliser par exemple la fonction suivante pour lancer un processus distant sur une machine `machine` avec les arguments `arguments`.

```
1  def lanceProcessus( machine, arguments ):
2      directory = os.getcwd()
3      command = 'ssh  ' + machine + ' "cd ' + directory
4      command = command + '; ./3-daemon.py ' + arguments + '"'
5      try:
6          remote = subprocess.Popen( command, shell=True )
7      except OSError as e:
8          print "OS Error", e.strerror,  "with filename", e.filename
9          sys.exit( -1 )
10     except:
11         print "Error", sys.exc_value, sys.exc_type
12         sys.exit( -1 )
13     remote.communicate()
14     return
```

2 Mise en place de l'infrastructure de communications

Vous voulez maintenant que vos processus établissent une topologie d'anneau. Par définition, un anneau est une topologie dans laquelle il n'existe qu'une seule composante connexe (tous les processus sont dedans) et où chaque processus a exactement deux voisins.

Si chaque processus (sauf le dernier) lance un autre processus, vous allez obtenir une chaîne : vous avez bien une seule composante connexe, chaque processus à part le premier et le dernier a exactement deux voisins, et les deux processus extrêmes ont un seul voisin.

Il faut alors que le premier processus soit connecté au dernier. Pour cela, il y a deux possibilités :

1. Quand le dernier processus est lancé, il ouvre une socket d'écoute et il envoie ses informations de connexion à son voisin (le processus qui vient de le lancer). Chaque processus qui reçoit ce message le transfère à son autre voisin, jusqu'à ce que toute la chaîne soit remontée. Quand le premier processus le reçoit, il se connecte au dernier processus.
2. Quand un processus est lancé, il reçoit en paramètre de sa ligne de commande non seulement les informations de connexion du processus qui vient de le lancer, mais aussi celles du premier processus. Quand le dernier processus est lancé, il se connecte non seulement au processus qui vient de le lancer, mais également au premier processus de la chaîne.

Dans le premier cas, chaque processus se connecte à exactement un processus et il accepte exactement une connexion entrante. Cependant, cela nécessite de transférer un message qui doit remonter toute la chaîne, soit $N-1$ communications dans un système à N processus. Dans le deuxième cas, les deux processus extrêmes sont différents des autres dans le sens où l'un accepte deux connexions entrantes et de se connecte à aucun autre processus, et l'autre se connecte à deux autres processus mais n'accepte aucune connexion entrante ; cependant cette solution ne nécessite aucune transmission de message.

La liste des machines sur lesquelles les processus doivent être lancés peut être envoyée sur la socket reliant les processus, ou passée en paramètre de la ligne de commande. Vous pourrez la lire sur le programme de lancement depuis un fichier en utilisant la fonction suivante :

```
1 def lireMachines( fichier ):
2     machines = []
3     fd = open( fichier, 'r' )
4     for ln in fd:
5         m = ln.strip()
6         if m != '':
7             machines.append( m )
8     fd.close()
9     return machines
```

Vous implémenterez une infrastructure de processus mettant en place une topologie d'anneau en utilisant les solutions de votre choix. Vous pourrez utiliser deux programmes, un pour le premier processus (que vous lancez à la main) et un démon pour les autres processus.

Une fois cette topologie mise en place, faites circuler un message : le dernier processus envoie un petit message (par exemple, son pid, ou un nombre aléatoire), et chaque processus qui reçoit ce message concatène un petit message à lui et l'envoie à son autre voisin. Le processus à l'origine de l'envoi affiche le message total.

3 Exclusion mutuelle

Vous souhaitez que vos processus écrivent dans un fichier unique. Vous avez la chance de disposer pour cela d'un répertoire partagé sur votre compte Unix, grâce au système de fichiers en

réseau de type NFS. Cependant, il ne faut pas que tous les processus écrivent dedans en même temps.

Vous avez donc besoin d'assurer une exclusion mutuelle entre vos processus. L'ordre est arbitraire. Implémentez l'algorithme d'exclusion mutuelle par jeton unique pour assurer l'exclusion mutuelle entre vos processus.

6 Élection de leader

Le but de cet exercice est de mettre en place une infrastructure de processus reliés en anneau et de les numéroter en utilisant un système de nommage avec des entiers consécutifs commençant à 0 et attribués de façon unique. Si on a N processus, ils sont donc numérotés entre 0 et $N-1$.

1 Lancement des processus

NB : cette partie est identique à la première partie du TP précédent. Vous pourrez donc réutiliser votre travail précédent.

Pour lancer un processus sur une machine distante, le processus lanceur effectue une connexion SSH sur la machine distante pour lancer le processus distant. Celui-ci se connecte à un autre processus, pour *rejoindre l'infrastructure*. On doit donc lui passer une adresse IP et un port en paramètres.

SSH permet de lancer une commande à distance. Vous pourrez pour cela utiliser la commande `Popen` du module `subprocess` si vous choisissez de travailler en Python. Le processus distant que l'on lancera ici sera un *démon*. Pour obtenir un démon, le processus doit faire un `fork()` et le processus père doit mourir. Si vous avez choisi de faire le TP en Python, vous pourrez utiliser, par exemple :

```
1  def main():
2      pid = os.fork()
3      if pid > 0:
4          sys.exit( 0 )
5      else:
6          daemon()
7      return
```

Écrivez un programme distribué qui :
- Ouvre une socket TCP en écoute, et découvre sur quel port elle écoute ;
- Lance un autre processus, sur une machine distante, via SSH ;
- Le processus distant est un démon qui se connecte au processus lanceur et lui envoie un petit message ;
- Le processus lanceur affiche ce message.

Vous pourrez utiliser par exemple la fonction suivante pour lancer un processus distant sur une machine `machine` avec les arguments `arguments`.

```
1  def lanceProcessus( machine, arguments ):
2      directory = os.getcwd()
3      command = 'ssh  ' + machine + ' "cd ' + directory
4      command = command + '; ./3-daemon.py ' + arguments + '"'
5      try:
6          remote = subprocess.Popen( command, shell=True )
7      except OSError as e:
8          print "OS Error", e.strerror,  "with filename", e.filename
9          sys.exit( -1 )
10     except:
11         print "Error", sys.exc_value, sys.exc_type
12         sys.exit( -1 )
13     remote.communicate()
14     return
```

2 Mise en place de l'infrastructure de communications

Vous voulez maintenant que vos processus établissent une topologie d'anneau. Par définition, un anneau est une topologie dans laquelle il n'existe qu'une seule composante connexe (tous les processus sont dedans) et où chaque processus a exactement deux voisins.

Si chaque processus (sauf le dernier) lance un autre processus, vous allez obtenir une chaîne : vous avez bien une seule composante connexe, chaque processus à part le premier et le dernier a exactement deux voisins, et les deux processus extrêmes ont un seul voisin.

Il faut alors que le premier processus soit connecté au dernier. Pour cela, il y a deux possibilités :

1. Quand le dernier processus est lancé, il ouvre une socket d'écoute et il envoie ses informations de connexion à son voisin (le processus qui vient de le lancer). Chaque processus qui reçoit ce message le transfère à son autre voisin, jusqu'à ce que toute la chaîne soit remontée. Quand le premier processus le reçoit, il se connecte au dernier processus.
2. Quand un processus est lancé, il reçoit en paramètre de sa ligne de commande non seulement les informations de connexion du processus qui vient de le lancer, mais aussi celles du premier processus. Quand le dernier processus est lancé, il se connecte non seulement au processus qui vient de le lancer, mais également au premier processus de la chaîne.

Dans le premier cas, chaque processus se connecte à exactement un processus et il accepte exactement une connexion entrante. Cependant, cela nécessite de transférer un message qui doit remonter toute la chaîne, soit $N-1$ communications dans un système à N processus. Dans le deuxième cas, les deux processus extrêmes sont différents des autres dans le sens où l'un accepte deux connexions entrantes et de se connecte à aucun autre processus, et l'autre se connecte à deux autres processus mais n'accepte aucune connexion entrante ; cependant cette solution ne nécessite aucune transmission de message.

La liste des machines sur lesquelles les processus doivent être lancés peut être envoyée sur la socket reliant les processus, ou passée en paramètre de la ligne de commande. Vous pourrez la lire sur le programme de lancement depuis un fichier en utilisant la fonction suivante :

```
1 def lireMachines( fichier ):
2     machines = []
3     fd = open( fichier, 'r' )
4     for ln in fd:
5         m = ln.strip()
6         if m != '':
7             machines.append( m )
8     fd.close()
9     return machines
```

Vous implémenterez une infrastructure de processus mettant en place une topologie d'anneau en utilisant les solutions de votre choix. Vous pourrez utiliser deux programmes, un pour le premier processus (que vous lancez à la main) et un démon pour les autres processus.

Une fois cette topologie mise en place, faites circuler un message : le dernier processus envoie un petit message (par exemple, son pid, ou un nombre aléatoire), et chaque processus qui reçoit ce message concatène un petit message à lui et l'envoie à son autre voisin. Le processus à l'origine de l'envoi affiche le message total.

3 Élection de leader et numérotation

Maintenant que vous avez un ensemble de processus connectés selon une topologie d'anneau, vous disposez des fondations pour implémenter un algorithme d'élection de leader. Implémentez

l'algorithme de Chang-Roberts pour élire un leader parmi vos processus Vous pouvez l'implémenter avec plusieurs initiateurs, qui peuvent être choisis aléatoirement.

Vous aurez pour cela besoin de mettre en place un mini-protocole différenciant les messages d'élection et les messages d'élu. Pour cela, vous pouvez par exemple faire commencer le message par un caractère donnant le type du message.

```
1  def electionMessage( message ):
2      m = "E" + str( message ) + " "
3      return m
4
5  def electedMessage( message ):
6      m = "F" + str( message ) + " "
7      return m
8
9  def depackMessage( message ):
10     return message[1:].strip(), message[0]
```

Attention, il est possible de recevoir plusieurs messages avant de faire le `recv`. Dans ce cas, soit vous avez ajusté la taille du buffer de réception de telle sorte que vous ne lisez qu'un seul message à la fois, soit vous devez être capables de séparer les messages (d'où l'espace inséré à la fin du message dans le code ci-dessus).

Une fois que le leader a été élu, il a le rang 0. Il démarre alors la circulation d'un jeton sur l'anneau. Ce jeton est utilisé pour la numérotation des processus : lorsqu'un processus reçoit le jeton

— il s'attribue sa valeur comme rang ;
— il l'incrémente ;
— il le passe à son voisin.

Vous ajouterez alors un type de messages à votre mini-protocole, correspondant au jeton en circulation dans le système.

Attention : les applications distribuées sont souvent pénibles à débugger parce que les sorties standard ne sont pas forcément transférées. On rate donc beaucoup d'affichages de débuggage. Il est alors très appréciable d'utiliser une fonction qui écrit dans un fichier de log (bien sûr, distinct pour chaque processus).

```
1  def log( m ):
2      fd = open( "/tmp/tp" + str( os.getpid() ), "a+" )
3      fd.write( m + "\n" )
4      fd.close()
5      return
```

La circulation du jeton de numérotation termine lorsque le processus de rang 0 le reçoit. Sa valeur est alors égale au nombre de processus dans le système : c'est donc également un système de comptage.

7 Détection de défaillances

Le but de cet exercice est de mettre en place un détecteur de défaillances dans un système à petite échelle hiérarchique. Les processus surveillés n'ont pas tous des détecteurs de défaillances : on considère deux processus comme les détecteurs, et chaque détecteur surveille un ensemble de processus.

1 Mise en place de l'infrastructure de surveillance

On considère que l'on a deux catégories de processus :
— Les détecteurs de défaillances
— Les processus à surveiller

Chaque détecteur de défaillances est relié à la moitié des processus à surveiller (vous en utiliserez environ 4 par détecteur). Concrètement, le détecteur est un serveur auquel les processus se connectent.

Les deux détecteurs de défaillances sont reliés l'un avec l'autre, par une simple connexion TCP. Vous choisirez arbitrairement lequel est le serveur et lequel est le client.

Pour lancer votre infrastructure, vous pourrez utiliser, comme dans les TP précédents, une liste de machine lue par un processus. Par exemple, vous pourrez réutiliser la fonction suivante, et prendre la première machine comme deuxième détecteur de défaillances :

```
1 def lireMachines( fichier ):
2     machines = []
3     fd = open( fichier, 'r' )
4     for ln in fd:
5         m = ln.strip()
6         if m != '':
7             machines.append( m )
8     fd.close()
9     return machines
```

Vous pourrez déployer votre infrastructure de la façon suivante :
— Un détecteur de défaillances est lancé
— Il ouvre une socket en écoute
— Avec une connexion SSH, il lance l'autre processus en lui passant en paramètre les informations de connexion de la socket qu'il a ouverte
— L'autre détecteur de défaillances ouvre une socket d'écoute
— Chacun des deux détecteur de défaillances lance les processus qu'il devra surveiller en leur passant en paramètre les informations de connexion de la socket qu'il a ouverte
— Le deuxième détecteur de défaillances se connecte au premier, et les processus se connecteur à leur détecteur de défaillances.

Pour toutes ces opérations, vous pourrez réutiliser du code écrit pour les TP précédents. Notamment, vos processus (tous) seront des démon : chaque processus lancé doit faire un `fork()` et le processus père doit mourir :

```
1 def main():
2     pid = os.fork()
3     if pid > 0:
4         sys.exit( 0 )
5     else:
6         daemon()
```

```
7       return
```

Vous pourrez utiliser la fonction `subprocess.Popen` pour lancer un processus distant sur une machine `machine` avec les arguments `arguments`.

```
1  def lanceProcessus( machine, arguments ):
2      directory = os.getcwd()
3      command = 'ssh  ' + machine + ' "cd ' + directory
4      command = command + '; ./process.py ' + arguments + '"'
5      try:
6          remote = subprocess.Popen( command, shell=True )
7      except OSError as e:
8          print "OS Error", e.strerror,  "with filename", e.filename
9          sys.exit( -1 )
10     except:
11         print "Error", sys.exc_value, sys.exc_type
12         sys.exit( -1 )
13     remote.communicate()
14     return
```

2 Implémentation d'un détecteur de défaillances

Vous allez implémenter un détecteur de défaillances se basant sur des horloges logiques. À un certain intervalle (par exemple, 15 secondes), chaque détecteur de défaillances envoie un heartbeat à chacun de ses processus.

Chaque détecteur de défaillances garde les informations concernant les processus qu'il surveille dans un tableau. Pour chaque élément du tableau (chaque processus surveillé), on a :

— La valeur du dernier heartbeat envoyé
— La valeur du dernier heartbeat reçu

En Python, vous pouvez facilement l'implémenter avec une liste de listes. La lite principale est le tableau de processus, et chaque élément de la liste est une liste contenant ces élément.

Lorsque le décalage entre les deux valeurs, pour un processus, dépasse un seuil (à fixer en dur), le détecteur de défaillances suspecte le processus d'être mort. Pour cela, il maintient deux listes de processus : les morts et les vivants. Attention, ces listes doivent contenir l'intégralité des identifiants des processus, et pas uniquement ceux des processus surveillés localement.

Quand le détecteur de défaillances considère qu'un processus est mort, il envoie son identifiant à l'autre détecteur de défaillances, qui met alors à jour ses listes de processus.

Afin de tester votre détecteur, vous pourrez suspendre un processus en lui envoyant le signal `SIGTSTP`. N'oubliez pas de le tuer complètement en fin d'exécution.

Quelles sont les limites en robustesse de cette architecture ? Quelles sont les simplifications et les hypothèses faites ici pour rendre la détection (et le consensus) possibles ?

8 Checkpoint distribué

Dans ce TP, vous allez implémenter un protocole de checkpoint distribué coordonné basé sur l'algorithme de Chandy et Lamport.

1 Mise en place de l'infrastructure de communications

La première chose à faire est de mettre en place une infrastructure de communications. Pour simplifier, mettez en place une topologie de graphe complet : chaque processus peut communiquer avec chaque processus.

Le processus lanceur lit la liste des machines à utiliser dans un fichier. Par exemple, vous pourrez utiliser le même code que dans les TP précédents :

```
1 def lireMachines( fichier ):
2     machines = []
3     fd = open( fichier, 'r' )
4     for ln in fd:
5         m = ln.strip()
6         if m != '':
7             machines.append( m )
8     fd.close()
9     return machines
```

De même, vous pourrez réutiliser la fonction qui lance les processus distants. Le processus lanceur est en charge de lancer tous les processus distants :

```
1  def lanceProcessus( machine, arguments ):
2      directory = os.getcwd()
3      command = 'ssh  ' + machine + ' "cd ' + directory
4      command = command + '; ./processus.py ' + arguments + '"'
5      try:
6          remote = subprocess.Popen( command, shell=True )
7      except OSError as e:
8          print "OS Error", e.strerror,  "with filename", e.filename
9          sys.exit( -1 )
10     except:
11         print "Error", sys.exc_value, sys.exc_type
12         sys.exit( -1 )
13     remote.communicate()
14     return
```

Vous pourrez utiliser un port d'écoute fixe : tous les processus ouvrent une socket d'écoute sur le même port. La première chose faite par les processus, après leur lancement, consiste à se connecteur au lanceur. Une fois que le lanceur a reçu une connexion de tous les processus, il acquitte le lancement : il leur envoie à chacun un message spécial. Ce message signifie que tous les processus ont terminé leur initialisation et peuvent se connecter les uns aux autres. En effet, les processus ne peuvent pas essayer de se connecter aux autres processus dès leur lancement : il est possible que le processus cible ne soit pas encore lancé.

Il est donc nécessaire de maintenir un ensemble de connexions avec tous les autres processus. Vous pouvez utiliser `poll()` ou `select()` pour écouter ce qui vient sur toutes ces sockets en même temps.

2 Enregistrement de l'état d'un processus

Pour simplifier, l'état d'un processus sera représenté par la valeur d'une variable. Par exemple, cette variable peut être le nombre de marqueurs reçus au moment où le checkpoint est fait. Vous l'enregistrerez dans un fichier local :

```
1 def ckpt( m ):
2     fd = open( "/tmp/tp" + str( os.getpid() ), "a+" )
3     fd.write( m + "\n" )
4     fd.close()
5     return
```

3 Implémentation d'un algorithme de prise d'état global

Chaque processus :
- Écoute sur ses sockets pendant un certain temps (le timeout du `poll()` ou du `select()`)
- Quand il en sort, il incrémente un compteur local (qui sera enregistré dans le checkpoint)
- Il envoie la valeur de son compteur à un autre processus (qui pourra par exemple être choisi aléatoirement)
- Puis il rentre à nouveau dans l'écoute

Un des processus, par exemple le lanceur, est l'initiateur de la vague de checkpoint. À un moment donné, il initie la vague en envoyant un marqueur à tous les processus. Implémentez un protocole de checkpoint *bloquant*.

Attention, vous autre donc à définir deux types de messages : les communications de l'application et les marqueurs.

Deuxième partie

Calcul intensif

9 Premiers pas avec MPI

Exercice 9.1 : Configuration de votre environnement

Pour préparer votre environnement pour les TP de MPI, vous avez besoin de configurer votre compte de manière à pouvoir utiliser SSH sans mot de passe.

Générez une paire de clés publique/privée avec `ssh-keygen`. Tapez entrer à toutes les questions qu'on vous pose (clés à mettre dans les fichiers par défaut, passphrase vide). Un répertoire `~/.ssh` est alors créé, contenant notamment le fichier contenant votre clé privée et votre clé publique. Mettez votre clé publique dans un fichier `~/.ssh/authorized_keys`.

```
$ ssh-keygen -t dsa
[...]
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Il faut maintenant que vous puissiez vous connecter à d'autres machines sans que SSH vous demande si vous acceptez la clé des machines. Ajoutez la ligne suivante dans un fichier `~/.ssh/config`.

```
StrictHostKeyChecking no
```

Exercice 9.2 : Exécution d'un programme sur plusieurs machines

En utilisant `mpiexec`, exécutez le programme `hostname` en parallèle sur 4 machines différentes.

Exercice 9.3 : Exécution d'un code C

Copiez le code suivant dans votre éditeur de texte préféré, et sauvegardez-le dans un fichier `.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main( int argc, char** argv ) {
    int size, rank, len;
    char name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Get_processor_name( name, &len );
    fprintf( stdout, "Je suis le processus %d parmi %d sur %s\n",
                                rank, size, name);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

1. Compilez le code.
2. Exécutez le programme obtenu sur 8 processus sur 4 machines différentes.

3. Avec `mpiexec -help`, quelle est la différence entre les options `-bycore`, `-bynode` et `-byslot` ? Vérifiez leur fonctionnement sur votre programme.

Exercice 9.4 : Exécution d'un code Python

Copiez le code suivant dans votre éditeur de texte préféré, et sauvegardez-le dans un fichier `.py`.

```
#!/usr/bin/env python

from mpi4py import MPI

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

print "Je suis le processus", rank, "parmi", size, "sur", name
```

Ou, si vous utilisez Python 3 :

```
#!/usr/bin/env python

from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Je suis le processus %d parmi %d sur %s.\n"
    % (rank, size, name))
```

Exécutez-le sur 8 processus sur au moins 4 machines.

Exercice 9.5 : Un bug !

Examinez le programme suivant.

```
#!/usr/bin/env python

from mpi4py import MPI
import numpy

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
comm = MPI.COMM_WORLD

if rank%2:
    dest = rank - 1
else:
    dest = rank + 1
```

```
14
15 print "Je suis le processus", rank, "parmi", size, "et j'envoie a",
       dest
16
17 data = numpy.arange(1000, dtype='i') # Creation d'un tableau de
       taille 1000
18 comm.send( data, dest=dest, tag=42 )
```

1. Que fait-il ?
2. Quel est le problème ?
3. Quelle serait la modification que vous y feriez pour le faire fonctionner ?

10 Communications point-à-point en MPI

Exercice 10.1 : Ping-pong

1. Créez un fichier contenant le code du ping-pong vu dans le cours
2. Compilez-le avec `mpicc`
3. Exécutez-le sur plusieurs machines de la salle de TP

Exercice 10.2 : Communications non-bloquantes

1. Reprenez le ping-pong de l'exercice précédent et modifiez-le pour utiliser des communications non-bloquantes `MPI_Isend` et `MPI_Irecv`.
2. Où placez-vous les `MPI_Wait` ?
3. Modifiez le code pour effectuer non pas un ping-pong mais un envoi simultané. Auriez-vous pu le faire avec des communications bloquantes ?

Exercice 10.3 : Anneau

Prenons un ensemble de processus MPI communiquant en suivant une topologie d'anneau.

1. Si un processus de cet ensemble a le rang k, quel est le rang du processus à qui k va envoyer des messages ?
2. Quel est le rang du processus de qui k va recevoir des messages ?
3. Comment la circulation du jeton doit-elle être initiée ?
4. Comment la circulation du jeton se termine-t-elle ?
5. Implémentez en MPI une circulation de jeton (un entier) selon un anneau entre des processus. Votre programme doit fonctionner quel que soit le nombre de processus.

11 Communications collectives en MPI

Exercice 11.1 : Diffusion

Écrivez un programme en MPI dans lequel le processus de rang 0 récupère son *process id* et le diffuse à tous les autres processus. Les processus doivent alors tous afficher cette valeur.

Exercice 11.2 : Réduction

Écrivez un programme en MPI qui se comporte comme suit :

1. Chaque processus alloue un tableau de 1014 éléments
2. Chaque processus remplit le tableau avec des valeurs aléatoires comprises entre 0 et 100 (attention à l'initialisation de votre générateur de nombres aléatoires, qui doit être seedée différemment sur chaque processus)
3. Chaque processus calcule la valeur maximum contenue dans ce tableau
4. En utilisant une opération collective, récupérez le maximum des maxima locaux dans le processus de rang 0.

Exercice 11.3 : Scatter-Gather

Question 1 : Écrivez un pogramme en MPI qui se comporte comme suit :

1. Le processus de rang 0 alloue un tableau dont le nombre d'éléments est égal au nombre de processus multiplié par 1024
2. Il remplit alors le tableau de valeurs aléatoires comprises entre -100 et 100
3. En utilisant une opération collective, ce tableau est distribué à tous les processus du système
4. Chaque processus a alors une portion du tableau. Chacun élève au cube toutes les valeurs comprises dans ce tableau.
5. Chaque processus calcule la moyenne contenue dans son tableau.
6. En utilisant une opération collective, récupérez l'ensemble des moyennes dans un tableau contenu dans le processus de rang 0 (attention à l'allocation mémoire).
7. Le processus de rang 0 calcule la moyenne des moyennes récupérées.

Question 2 : Comparez le temps de calcul de l'implémentation parallèle que vous venez d'écrire avec une implémentation séquentielle.

Exercice 11.4 : Implémentation de la diffusion

Question 1 : En utilisant uniquement des opérations d'envoi et de réception, implémentez la diffusion d'un entier en utilisant l'algorithme de l'étoile.

Question 2 : Même question en utilisant un arbre binomial.

Exercice 11.5 : Implémentation de la distribution

Question 1 : En utilisant uniquement des opérations d'envoi et de réception, implémentez la distribution d'un entier en utilisant l'algorithme linéaire.

Question 2 : Même question en utilisant un arbre binomial.

Question 3 : Même question en utilisant un arbre binaire.

Exercice 11.6 : Implémentation du all-to-all

En utilisant uniquement des opérations d'envoi et de réception (vous pourrez utiliser des communications non-bloquantes), implémentez un all-to-all où chaque processus envoie un entier différent à chaque autre processus.

Exercice 11.7 : Normalisation d'une matrice

Le but de cet exercice est de normaliser les données d'une matrice, c'est-à-dire de diviser la valeur de tous ses éléments par la valeur du plus grand élément.

1. Créez un programme parallèle avec MPI dans lequel les processus disposent chacun d'une sous-matrice aléatoire de taille MxN.
2. Sur chaque processus, recherchez la valeur du plus grand élément de la sous-matrice.
3. En utilisant une opération collective, calculez la valeur du maximum global. Tous les processus doivent avoir cette valeur.
4. Sur chaque processus, divisez la valeur des éléments de la sous-matrice par le maximum global.

12 Types de données

Exercice 12.1 : Type dérivé I

1. Créez un type dérivé en MPI contenant un ensemble d'entiers contigus
2. Écrivez un programme où deux processus s'échangent un tableau d'entiers, en utilisant le type que vous venez de définir. Le nombre d'éléments envoyés doit être égal à 1, le type de données est votre type dérivé.

Exercice 12.2 : Type dérivé II

1. Créez un type dérivé en MPI contenant des entiers non-contigus :
   — un entier par bloc
   — le stride (écart entre le début d'un bloc et le début du suivant) est égal à un paramètre N
   — le nombre de blocs est égal à un nombre M
2. Écrivez un programme où deux processus disposent chacun d'une matrice de M lignes et N colonnes stockée linéairement dans un tableau à une dimension. Ces processus s'échangent les premiers éléments de chaque ligne :
   (a) Implémentez-le avec des communications envoyant les éléments un par un. Combien de messages cette méthode implique-t-elle ? Quel est le volume de données (total et par message) envoyées ?
   (b) Implémentez-le en utilisant le type dérivé que vous venez de définir. Comparez le nombre de message et le volume de données transportées avec l'implémentation précédente.

Exercice 12.3 : Transposition de matrice

Le but de cet exercice est de transposer efficacement une matrice distribuée sur des processus parallèles.

1. Écrivez un programme parallèle avec MPI dans lequel deux matrices sont réparties sur les processus en étant stockées linéairement dans des tableaux à une dimension. La première matrice est composée de M lignes et N colonnes, la deuxième est composée de N lignes et M colonnes. La matrice est distribuée par décomposition de domaine selon les lignes : ainsi, chaque processus a une matrice (M/P)xN et une matrice (N/P)xM. Pour des raisons de simplicité, prenez N=M=P. Ainsi, chaque processus dispose d'une ligne de chaque matrice.
2. En utilisant les types dérivés définis dans les exercices précédents, modifiez votre programme afin que chaque processus envoie le premier élément de chaque ligne de sa matrice au processus 0, qui stocke tous ces éléments de façon contiguë dans la ligne de la matrice dont il dispose. Pour cela, vous utiliserez une opération collective en étant particulièrement attentif aux types de données utilisés en envoi et en réception (indication : ils ne sont pas identiques).
3. En utilisant une autre opération collective, modifiez votre programme pour que les k-ièmes éléments de la matrice soient rassemblés sur le processus k. L'opération effectuée sur la matrice est une transposition.

4. Pour utiliser une matrice de taille quelconque, quelle doit être la taille des matrices contenues sur chaque processus ?
5. Quels doivent être les paramètres des types de données dérivés qui doivent alors être utilisés (nombre d'éléments, stride, nombre de blocs) ?
6. Modifiez votre programme pour transposer des matrices de taille quelconque.

Exercice 12.4 : Maître-esclave statique

Implémentez un squelette de maître-esclave statique en utilisant des opérations collectives pour distribuer les données et récupérer les résultats.

Exercice 12.5 : Maître-esclave dynamique

Implémentez un squelette de maître-esclave dynamique en mode pull : les esclaves demandent du travail au maître quand ils envoient un nouveau résultat ou à l'initialisation, et celui-ci leur envoie des données (un tableau d'entiers) tant qu'il en a.

13 Communications unilatérales

Exercice 13.1 : Intermédiaire

Écrivez un programme qui s'exécute avec au moins trois processus. Le processus 0 initialise un entier avec son `pid`. Le processus 1 va le chercher avec un `MPI_Get` et l'écrit dans la mémoire du processus 2 avec un `MPI_Put`.

Exercice 13.2 : Anneau à jeton

Écrivez une variante de l'anneau à jeton utilisant des communications unilatérales. Attention : n'oubliez pas qu'un processus ne sait pas si un autre processus a lu ou écrit dans sa mémoire. Vous devrez donc mettre en place un mécanisme de synchronisation.

Exercice 13.3 : Aléa reproductible

Les générateurs de nombres aléatoire sont le plus souvent en réalité pseudo-aléatoire : les nombres générés dépendent de l'état du générateur. C'est pour cela qu'on doit toujours initialiser le générateur avec une *graine*.

Le but de cet exercice est de générer le même tableau pseudo-aléatoire dans tous les processus. Pour cela, il faut qu'ils initialisent tous leur générateur de nombres aléatoires avec la même graine. Il y a trois façons d'implémenter ceci.

La première consiste à générer un tableau sur un processus et le diffuser aux autres processus. L'inconvénient est que si le tableau est très gros, les communications peuvent être très longues.

La deuxième solution consiste à générer la graine sur un processus et la diffuser à tous les autres processus, qui initialisent leur générateur de nombres aléatoires et génèrent chacun un tableau aléatoire. L'inconvénient de cette approche réside dans son caractère synchrone : il faut que tous les processus soient arrivés à ce point à ce moment de l'exécution du programme.

Vous allez implémenter la troisième solution. Un processus génère la graine et la stocke dans sa mémoire accessible par les autres processus. Les autres processus lisent la graine et génèrent leur tableau.

Quel est l'inconvénient de cette approche par rapport à la deuxième solution ?

Exercice 13.4 : Maximum irrégulier

Le but de cet exercice est de calculer le maximum global d'un tableau distribué irrégulier. Chaque processus dispose d'un tableau aléatoire de taille différente (en pratique, vous tirerez la taille aléatoirement entre 100 et 100 000). Chaque processus calcule le maximum de son tableau et le processus 0 calcule le maximum des maxima.

Une façon simple d'implémenter un maximum global consiste à calculer le maximum local sur chaque processus et effectuer une réduction avec `MPI_MAX` vers un processus racine, qui disposera alors du maximum global. Cependant, dans le cas présent, le calcul est trop irrégulier, et les processus rapides devront attendre les processus les plus lents lors de la communication collective.

Vous pouvez alors utiliser des communications unilatérales : un processus qui a terminé écrit son résultat dans le processus 0 avec `MPI_Accumulate`, et il passe au suivant. Attention, pour que le processus 0 sache que le calcul est terminé, il doit savoir combien de processus ont écrit : pour cela, vous pourrez utiliser un compteur qui est incrémenté également avec `MPI_Accumulate`.