Devoir de programmation Réseaux

Martin Delacourt, Nicolas Ollinger

Le but de ce devoir est d'implémenter un serveur pour le protocole ICB (www.icb.net). C'est un protocole client-serveur de chat centralisé, utilisant des messages principalement textuels. Vous avez à disposition :

- une spécification du protocole qui détaille la structure et le contenu des messages échangés entre client et serveur;
- un lab netkit (**labicb**) contenant un serveur ICB et un client python.

Vous rendrez un seul fichier **icbserv.py** contenant le code python du serveur ICB que vous aurez programmé.

Protocole ICB

Les échanges du protocole ICB se font au travers d'une connexion TCP entre chaque client et le serveur d'accueil. Une fois la connexion établie, les paquets échangés sont décrits dans la spécification. Tous les paquets ont la même structure `LTd` où :

- `L` est la taille en octets du paquet, elle est codée sur un seul octet (i.e. un caractère). La taille ne compte pas l'octet correspondant à `L`, en revanche, elle inclut `T` et le caractère NULL (`\x00`) terminal;
- `T` est le type du paquet codé sur un seul octet;
- `d` est un ensemble de champs textuels séparés par le caractère `\x01`. La liste de champs correspondants à un type de paquet est décrite dans la spécification, certains champs sont susceptibles d'être optionnels;
- le caractère `\x00` marquant la fin du paquet.

Utilisation du client

On lance le client avec la commande `/icb`, ce qui établit la connexion avec le serveur et procède au login. Toute ligne ne commençant pas par `/` est considérée comme un message ouvert. Le serveur connaît l'ensemble des groupes et utilisateurs connectés, auquel le client a accès par la commande `/w`.

Alice

```
alice:~# ./icb
welcome to python icb.
warning: can't read config file, using
defaults.
connected to the daicbd ICB server (icbd)
Logged in.
```

Bob

```
bob:~# ./icb
welcome to python icb.
warning: can't read config file, using defaults.
connected to the daicbd ICB server (icbd)
Logged in.
[=Status=] You are now in group agora
```

```
[=Status=] You are now in group agora
[=Sign-on=] bob (bob@10.0.0.2) entered
group
[=Sign-on=] charlie (charlie@10.0.0.3)
entered group
/w

Group: agora    (pvl) Mod: (None)
Topic: (None)
   charlie          11s  14:32
charlie@10.0.0.3
   bob              14s  14:32
bob@10.0.0.2
   alice              -  14:32
alice@10.0.0.1

Group: 1        (pvl) Mod: (None)
Topic: (None)
Total: 3 users in 2 groups
```

```
[=Sign-on=] charlie (charlie@10.0.0.3) entered group
```

On considère ici uniquement les commandes `/w`, `/m`, `/g`, `/name`, `/topic`, `/pass`, `/q` ainsi que la commande `/?` qui liste toutes les commandes recevables par le serveur. Les exemples ci-dessous illustrent les usages de ces commandes. La commande `/m` permet d'envoyer un message personnel, `/g` permet de changer de groupe ou de créer un groupe s'il n'existe pas encore, auquel cas on en devient modérateur. La commande `/topic` change le thème du groupe, et `/pass` permet à un modérateur de transférer les droits de modération ou de se les attribuer s'il n'y a pas de modérateur. On affiche son surnom ou on en change avec `/name`, enfin `/q` provoque une déconnexion.

Alice

```
Hello
/m bob world!
```

```
‹alice› Hello
‹*alice*› world!
```

```
[=Depart=] bob (bob@10.0.0.2) just left
```

```
/g piscine
[=Status=] You are now in group piscine as moderator
```

```
/w

Group: piscine  (mvl) Mod: bob           Topic:
(None)
 * bob              22s  14:46  bob@10.0.0.2

Group: agora    (pvl) Mod: (None)        Topic:
(None)
   alice              -  14:46  alice@10.0.0.1
   charlie        4m22s  14:46
charlie@10.0.0.3

Group: 1        (pvl) Mod: (None)        Topic:
(None)
Total: 3 users in 3 groups
‹charlie› Bob?
```

```
[=Notify=] server has passed moderation to
charlie
[=Topic=] charlie changed the topic to "Where
is Bob?"
/name
[=Name=] Your nickname is alice
/name zebra
[=Name=] alice changed nickname to zebra
```

```
/w

Group: piscine  (mvl) Mod: bob           Topic: (None)
 * bob                -  14:46  bob@10.0.0.2

Group: agora    (mvl) Mod: charlie       Topic: Where is Bob?
   alice          7m54s  14:46  alice@10.0.0.1
 * charlie        3m19s  14:46  charlie@10.0.0.3

Group: 1        (pvl) Mod: (None)        Topic: (None)
Total: 3 users in 3 groups
```

Bob

```
[=Sign-off=] charlie (charlie@10.0.0.3) just
left
/w

Group: piscine  (mvl) Mod: bob          Topic:
(None)
 * bob             9m7s  14:46  bob@10.0.0.2

Group: agora    (pvl) Mod: (None)       Topic:
Where is Bob?
   zebra              -  14:46  alice@10.0.0.1

Group: 1        (pvl) Mod: (None)       Topic:
(None)
Total: 2 users in 3 groups
```

Paquets échangés

Le détail des paquets est donné dans la spécification du protocole. Une session standard est constituée de la suite de paquets suivante :

- un paquet de type `Protocol` est envoyé par le serveur une fois la connexion TCP établie, on utilisera le niveau de protocole 1 pour le premier champ;
- un paquet de type `Login` envoyé par le client, avec la commande `login` dans le champ 3. On ne se préoccupera pas des champs 4, 5 et 6 dans cette implémentation;
- un paquet réponse de type `Login` envoyé par le serveur suivi d'un paquet de type `Status` contenant `Status` dans son champ 0;
- une suite de paquets de types `Open`, `Personal`, `Status`, `Error`, `Command` et `Command Output`;
- le client ferme la connexion TCP.

Toutes les notifications de type

```
[=status_category=] status message
```

sont des paquets de type `Status` contenant `status_category` dans le champ 0.

Une commande `/cmd` est transmise par un paquet de type `Command` avec `cmd` dans le champ 0. Une exception : lors d'une commande `/?`, le champ 0 contient `Help`. Si une réponse est nécessaire, elle sera transmise au moyen d'un paquet de type `Command Output`, le champ 0 (`Output Type`) contiendra `co`, `wl` ou `wg` dans votre implémentation. Un message personnel est traité par le client comme étant la commande `/m` puis transmis par le serveur au moyen d'un paquet de type `Personal`. Le serveur envoie des paquets de type `Error` lorsque nécessaire, par exemple en cas de paquet mal constitué ou d'opération interdite.

Attention, la commande `/q` est interne au client, elle n'est pas transmise au serveur. Elle provoque la fermeture de la socket du côté client. En détectant cette fermeture, le serveur apprend le départ de l'utilisateur concerné.

Travail à accomplir

Vous écrirez un serveur ICB en Python. Le fichier sera nommé **icbserv.py**.

Dans la version fournie de **labibc**, un serveur ICB de test se lance au démarrage du lab, ce qui vous permettra de manipuler le client à volonté. Pour tester votre serveur, vous ouvrirez le fichier **server.startup** du lab, il contient la ligne

```
./icbd -4 -C -d -G agora -L log -n -S daicbd -v 0.0.0.0:7326 &
```

que vous supprimerez ou commenterez (#) pour annuler le lancement automatique du serveur ICB. Ensuite, vous placerez votre fichier **icbserv.py** dans le répertoire **shared/root/** et ajouterez une ligne dans **server.startup** pour lancer votre serveur au démarrage.

Il est conseillé de s'inspirer du serveur de chat vu en TP (fiche TP2, 1.3), ICB utilise des connexions TCP et le serveur écoute sur le port 7326.

Votre serveur doit :

- autoriser une connexion demandée par le client fourni;
- imiter le comportement du serveur exemple lors de la réception des paquets de type `Login`, `Open` et `Command`;
- traiter les commandes `/w, /m, /g, /name, /topic, /pass` et `/?`;
- détecter le départ d'un utilisateur et le supprimer de ses listes d'utilisateurs/groupes.

Éléments de programmation réseau en Python 3

Un socket est une interface de connexion qui permet à des processus de communiquer, notamment à travers un réseau TCP/IP.

Il existe deux modes de communication :

- connecté : le protocole TCP est utilisé et une communication durable est établie, ce qui permet plusieurs échanges.
- non connecté : le protocole UDP est utilisé et la communication ne dure que le temps d'envoyer les données, la communication est ensuite coupée.

Nous utiliserons dans la suite un socket connecté.

Créer un socket

On commence par créer un socket et définir quelques options :

```
from socket import socket, SOL_SOCKET, SO_REUSEADDR

s = socket()
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

La dernière ligne définit l'option `SO_REUSEADDR` qui autorise la réutilisation d'un port immédiatement après la fermeture du socket (sinon, il y a un temps de latence, qui peut parfois retourner une erreur `Address already in use` lors de la mise au point du code).

Établir une connexion

Côté serveur : attendre une connexion sur le socket

Le socket doit être lié à un port sur lequel se fera l'écoute. La fonction `bind` attend en paramètre une paire constituée d'une adresse IP servant à filtrer les clients autorisés à se connecter (`0.0.0.0` si on ne filtre pas) et du numéro de port à réserver. La fonction `listen` attend en paramètre le nombre maximum de connexions en attente dans la queue de réception, avant acceptation par le serveur.

Ensuite, chaque appel à la fonction `accept` attend la connexion d'un client et retourne un nouveau socket connecté au client ainsi que l'adresse du client.

```
# lier l'interface par défaut sur le port 6666
s.bind(('0.0.0.0',6666))
s.listen(1)

sc, addr = s.accept()
```

(*Note:* pour la programmation d'un serveur pouvant gérer plusieurs clients simultanément, on se tournera vers le module `select`.)

Côté client : connecter le socket

Une fois le socket créé sur le client, nous pouvons le connecter à une adresse IP et un port, sur un serveur en attente d'une connexion.

```
# connexion sur la machine 127.0.0.1 (localhost) sur le port 6666
s.connect(('127.0.0.1',6666))
```

remarque. La fonction `gethostbyname` peut être utilisée pour obtenir l'adresse IP d'un nom d'hôte. Par exemple, `gethostbyname("localhost")` retourne `127.0.0.1`.

Utiliser un socket

Le socket est connecté, il n'y a plus qu'à envoyer ou recevoir des données. Celles-ci doivent être codées sous forme de `bytes`. La conversion entre les chaînes de texte `utf-8` et les `bytes` se réalise à l'aide des fonctions `encode` et `decode` : `send(msg.encode('utf-8'))` ou `recv(1024).decode('utf-8')`.

envoyer à un bout

```
msg = "bonjour\n"
buf = msg.encode('utf-8')
c.send(buf)
```

recevoir à l'autre bout

```
data = s.recv(1024)
msg = data.decode('utf-8')
```

remarque. La fonction `recv` est bloquante : tant qu'elle ne reçoit rien, elle attend.

Déconnecter un socket

```
s.close()
```

Quelques liens

- doc sur le module `socket`
- socket programming HowTo
- doc sur le module `select`
- programmation réseau en python sur openclassrooms
- les RFC
- la spécification d'ICB

Dernière modification le 17 Novembre 2017.