Architecture et Systèmes

Stefan Schwoon

Cours L3, 2020/2021, ENS Cachan

Réseau

Applications:

- Communication entre utilisateurs (e-mail, web)
- Partager des données entre plusieurs machines (Network file system, NFS)
- Remote procedure call (RPC)
- ...

Aujourd'hui:

- cours accéleré sur la programmation socket (TCP/IPv4)
- comment assurer une connection fiable sur un médium non-fiable

Programmation réseau

On discutera les bases pour écrire des applications simples (genre serveur/client) sur réseau, en utilisant le protocole TCP.

Adresse IP et ports

utilisation des sockets

Adressage IP

Pour participer au protocole IP, une machine doit être équipée d'une adresse IP (32 bits pour IPv6, 128 bits pour IPv6).

Adresse affectée automatiquement lorsqu'on utilise le protocole DHCP.

Pas très pratique pour l'usage humain, on prèfere, p.ex. `www.lsv.fr` à `138.231.81.11`.

Mécanisme de *résolution de nom* pour convertir une représentation sous forme de texte vers une adresse IP.

Fonction C à utiliser: `gethostbyname`

Cas spécial : `localhost` équivaut `127.0.0.1`

Les ports

Pour permettre aux machines de gérer plusieurs communications en même temps, on utilise des *ports*.

Dans les protocoles TCP et UDP une addresse consiste donc d'une *adresse machine* et d'un *port*. Le nombre de ports est de 2^{16}.

Les ports 0..1023 sont typiquement reservés pour certains protocoles, p.ex. 80 pour HTTP. Ils ne peuvent pas être occupés par les processus d'un utilisateur simple.

Structure de données : `sockaddr`, généralise `sockaddr_in` (IPv4) et `sockaddr_in6` (IPv6).

Attention : le port doit être stocké en *format réseau* (`htons/ntohs`)

Les sockets

En POSIX, un socket est une structure que permet de communiquer sur un port IP.

Un descripteur de fichier peut être lié à un socket.

- Création d'un socket : appel `socket`(2).
- Lier un socket à un port : appels `bind`(2) ou `connect`(2).
- Envoi/reception de données : `read`, `write`

Modèle typique :

- Serveur : ouvre un port sur sa machine et attend une connection.
- Client : se connecte au port d'un serveur

Serveur

Typiquement, un serveur fait les étapes suivants (voir l'exemple) :

1. Créer un socket socket (avec `socket`).

2. Connecter le socket à un port IP (avec `bind`).

3. Configurer le socket en mode écoute (avec `listen`).

4. Attendre un client (avec `accept`).

Remarque: accept renvoie un descripteur qui sert uniquement à communiquer avec ce client. Du coup, on peut communiquer avec plusieurs clients sur un même socket.

Client

Un client typique :

1. Créer un socket TCP (comme dans le serveur).

2. Connecter le socket à un port du serveur (avec `connect`).

Après connect, le socket est lié à un port libre du client qui est en communication avec le bon port du serveur. Le socket peut être utilisé comme descripteur pour envoyer/recevoir des données.

Terminer une connection sur socket

Première possibilité : `close`

Interdit tout communication (lecture/écriture) future sur le descripteur donné.

Si descripteur partagé entre plusieurs processus, n'affecte que le processus qui fait `close`.

Deuxième possibilite : `shutdown`

Permet de terminer sélectivement l'écriture ou la lecture sur un descripteur/socket donné.

Affecte tout les processus partageant le descripteur.

Scénario I : Serveur avec clients indépendants

P.ex. serveur web : tout client envoie une requête,
pas de communication entre clients

Le serveur met en place le socket (étapes 1 à 3).

Ensuite, le serveur boucle en attendant de nouveaux clients (`accept`).

Pour chaque connection, on crée un nouveau thread ou processus fils qui gère ce client.

Entretemps, le thread principal peut accepter de nouvelles connections.

Scénario II : Communication entre clients

P.ex. serveur tchat : le serveur accepte des clients, puis les messages d'un client sont envoyés aux autres.

Le serveur met en place le socket (étapes 1 à 3).

Ensuite le serveur surveille en même temps les nouvelles connections et les messages des clients pour y réagir (avec `select`).

Parenthèse : Protocôle HTTP

Normalement, toute requête de page est une connection différente:
le client se connecte, demande une page, et se déconnecte

Pour réaliser une session avec login: *identifiant de session*
généré par le serveur lors du login, le navigateur le garde et l'envoie avec toute requête par la suite ; détruit quand on ferme son navigateur

Cookie: Principe similaire, mais gardé par le navigateur plus longtemps

Streaming: La connection reste ouverte, le serveur envoie un flux de données continu (p.ex. vidéos, cours en ligne, . . .)

Comment réaliser une connection fiable

Sur certains couches (notamment 2 et 4) on doit réaliser une connection fiable sur des canaux non-fiables. Hypothèses :

Le canal de communication transmet des paquets individuels.

Le temps d'un transfert est variable (sans aucune borne).

Le transfert d'un paquet peut échouer sans notification.

Le canal n'est pas complètement bloqué (de temps en temps, un paquet arrivera).

Les paquets arrivent dans l'ordre d'envoi.

On suppose que l'intégrité des paquets est assurée (par des *sommes de contrôle*).

Scénario

On étudie un protocole très simple pour le scénario suivant :

> Alice produit des messages $m_0, m_1, \ldots$ et les envoie à Bob. L'objectif est d'assurer que Bob reçoit tous les messages.

Hypothèse additionnelle :

> Bob ne gère qu'un message à la fois.
>
> Du coup, Alice doit attendre la confirmation de Bob après chaque message.

Solution simple : Alternating-bit protocol

Alice et Bob gèrent tous les deux leur propre bit “séquence” s_A and s_B, initialement 0 et 1, respectivement.

Alice possède un compteur i, initialement 0.

Les messages d’Alice sont de la forme (m_i, s_A).

Les messages de Bob sont de la forme s_B.

Le protocole

Le protocole d'Alice :

1. Alice envoie (m_i, s_A) et répète ce message de temps en temps.

2a. Quand Alice reçoit un bit égal à s_A de la part de Bob, elle augmente i et invertit s_A.

2b. Si Alice reçoit un bit de Bob qui n'est pas égal à la valeur de s_A, elle ignore ce message et continue comme avant.

Le protocole de Bob :

1. Bob envoie s_B et répète ce message de temps en temps.

2a. Quand Bob reçoit $(m, \neg s_B)$, il stocke le message m et invertit s_B.

2b. Si Bob reçoit (m, s_B), il ignore le message.

Remarques

Ce protocole construit un flux de communication fiable sur un canal non-fiable, sous les hypothèses évoquées.

Le protocole (tel qu'il est presenté) est inefficace : Alice doit attendre une confirmation pour tout paquet. On peut le rendre plus efficace en utilisant un ensemble plus grand d'identifiants qu'on adapte en cas d'erreur.

On peut facilement limiter la communication aux données qui arrivent dans l'ordre d'envoi : Alice va étiquetter chaque paquet avec un identifiant unique qui augmente pour chaque paquet, Bob élimine les messages qui arrivent hors ordre.

Détection et correction d'erreurs

Problème : Alice envoit un message (séquence de bits) à Bob.
Le message peut être altéré (aléatoirement) par des erreurs de transmission.
Comment tester si des erreurs sont intervenues ?

Détection d'erreurs : méthode de parité
Alice compte si le nombre de 1 est pair ou impair.
Si impaire, elle ajoute 1 au message sinon 0.
Bob vérifie si le nombre de 1 au total est pair.

Correction d'erreurs : Codes de Hamming