

Architecture et Systèmes

Stefan Schwoon

Cours L3, 2018/19, 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 accéléré sur la programmation socket (TCP/IPv4)

⇒ Cours de Réseau en M1

Modèle OSI

OSI = Open Systems Interconnection, modèle standardisé par l'ISO
(Organisation internationale de normalisation)

Décrit une suite de protocoles définissant les différentes tâches (côté matériel et logiciel) requises pour la communication sur réseau

Objectifs :

établir des notions pour bien séparer les différents problèmes

permettre une définition modulaire des protocoles

Les couches ISO

L'OSI définit sept couches de communication :

physique / liaison / réseau / transport / session / présentation / application

Les couches “basses” fournissent des fonctionnalités simples.

Les couches “hautes” s'en servent pour accomplir des tâches plus complexes.

Une couche sur un ordinateur communique avec une couche du même niveau sur un autre ordinateur.

En principe, les données entre deux utilisateurs/applications sont censées traverser les instances de toutes les couches des deux côtés.

Couches physique et liaison

Les deux premières couches s'occupent de la communication entre deux ou plusieurs machines physiquement liées (réseau local).

Physique : Standards pour transmettre des bits (voltage, vitesse, etc)

Liaison : Grouper des bits/octets en paquets, détection d'erreurs

Exemples: Ethernet, ISDN, DSL, ...

Matériel: repeater, hub, bridge, switch

Adresse MAC : Identifiant (6 octets) d'une interface physique (p.ex., carte réseau).

Couche réseau

La couche **Réseau** gère la communication entre deux machines à travers d'un réseau, c'est à dire plusieurs machines/réseaux locaux interconnectés.

Chaque machine dans le réseau est identifié par une *adresse réseau*.

Le protocole décide par quels liens les données seront acheminées.

Il envoie des données par *paquets* (de taille limitée), et chaque paquet est acheminé individuellement.

Les protocoles réseau sont en général **non-fiables**, c'est à dire ils ne garantissent pas l'arrivée des données à la destination (dû à des problèmes de matériel, congestion, ...).

Exemple : IP (Internet protocol)

Machines identifiés par une adresse IP (32 bits pour IPv4, 128 pour IPv6)

Adresse hiérarchiques : partie haute identifie un sous-réseau, partie basse une machine dans un réseau

IPv4 : ancienne version, espace d'adresses plus ou moins épuisé

IPv6 : nouvelle version, mais pas encore disponible partout

Couche transport

La couche **Transport** établit une “connection” entre deux processus sur deux machines.

Découper les données en paquets individuels (qui pourront prendre de chemins différents).

Protocoles importants :

UDP : protocole simple et efficace, mais sans traitement d’erreurs en cas de perte de données

TCP : assure que tous les paquets seront bien reçus et dans le bon ordre

Couches “logiciel”

Les couches **session** / **représentation** / **application** sont des notions pour la construction des logiciels et se confondent souvent.

Session : Preserver une ‘connection logique’ à travers de plusieurs connections physiques (services d’authentification, cookies, ...)

Représentation : comment représenter des données (caractères, valeurs numériques, ...)

Applications : des protocoles comme SSH, HTTP, FTP, ...

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, \dots 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 présenté) 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 étiqueter chaque paquet avec un identifiant unique qui augmente pour chaque paquet, Bob élimine les messages qui arrivent hors ordre.

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 IPv4, 128 bits pour IPv6).

Adresse affectée automatiquement lorsqu'on utilise le protocole DHCP.

Pas très pratique pour l'usage humain, on préfère, 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 adresse consiste donc d'une *adresse machine* et d'un *port*. Le nombre de ports est de 2^{16} .

Les ports 0..1023 sont typiquement réservés 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.

Servir plusieurs clients en même temps

Comment un serveur peut-il communiquer avec plusieurs clients :

Problème: Le serveur ne sait pas quel client va envoyer le prochain bloc de données.

`read` (sur un descripteur donné) attendra patiemment les prochaines données ce qui bloquerait le serveur.

Solutions:

Ou créer un processus (ou thread) fils qui servira exclusivement ce client.

Utiliser l'appel `select` pour trouver un descripteur qui a des données disponibles.

Terminer une connection sur socket

Première possibilité : `close`

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

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

Deuxième possibilité : `shutdown`

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

Affecte tous les processus partageant le descripteur.