Environnement Client/Serveur

Cours 3
TCP
Rappels sur le multi-thread en Java

kn@lri.fr
http://www.lri.fr/~kn

Plan

Rappel sur les couches réseau

La couche de routage du modèle Internet (couche IP) s'assure de l'acheminement point à point de paquets de données. La source et la destination sont données par leur adresse IP. Entre ces deux points, peuvent exister plusieurs routes possibles, plusieurs supports physiques (Ethernet, Satellite, Wifi, GSM, ...) ce qui peut donner lieu à :

- Des pertes de paquets
- Des duplications de paquets
- Une arrivée dans le désordre des paquets

La couche de transport (UDP, TCP, ...) de fournir une abstraction de la connexion réseau à la couche application et s'occupe des détails de fragmentation des données et de transmision. Les deux protocoles principaux sont :

- UDP : protocole peu fiable mais presque sans surcoût, orienté performance et simplicité.
- TCP : protocole complexe qui garanti le bon acheminement des données ou signale une impossibilité.

Transmission Control Protocol

Le protocole TCP est un protocole complexe ayant les caractéristiques suivantes :

- Orienté flux (*stream*) plutot que message (à l'inverse d'UDP)
- Bi-directionnel (*Full-Duplex*) : une fois la connexion établie, le client et le serveur peuvent communiquer dans les deux sens (comme UDP)
- Garantie que les données envoyées arrivent dans l'ordre
- Garantie qu'en cas d'erreur de transmission irrécupérable, l'envoyeur est notifié

Fonctionnement de TCP

Pour garantir l'intégrité des données transmises via des paquets IP faillibles, TCP utilise un système d'accusé de réception (ACK pour *acknowledgement*). Il y a trois phases principales lors d'une connexion TCP :

1. La mise en place de la connexion
2. L'échange de données proprement dit
3. La fermeture de la connexion

Mise en place de la connexion

C'est une phase critique car elle permet de garantir par la suite l'intégrité des données (ou la détection d'erreurs irrécupérables). Elle fonctionne sur le principe d'une négociation en 3 étapes:

1. Le client (qui est toujours celui qui initie la connexion, par définition) envoie au serveur un message `SYN` (pour *synchronized*) ainsi qu'un numéro de séquence initial (ISN, *initial sequence number*) X choisi aléatoirement.
2. Le serveur, envoie en réponse un message `SYN-ACK` (*synchronized-acknowledgement*), son numéro de séquence initial Y choisi aléatoirement et le numéro d'acquitement $X+1$
3. Le client renvoie au serveur message `ACK` avec le numéro d'acquitement $Y+1$

Si le protocole n'arrive pas jusqu'à l'étape 3, le client déclare qu'il ne peut pas se connecter au serveur. Si c'est le cas, alors le client et le serveur ont échangé suffisamment d'information pour pouvoir communiquer en détectant les problèmes.

Pourquoi cela fonctionne ?

◆ Sur reception du **SYN-ACK**, le client a la certitude que le serveur a reçu son numéro de séquence X, puis-ce que ce dernier a renvoyé $X+1$.
◆ Sur reception du **ACK** le serveur a la certitude que le client a reçu son numéro de séquence Y car il lui a renvoyé $Y+1$.

Pourquoi choisir X et Y aléatoirement et pas intialement à 0 ? En cas de connexion/déconnexion rapide entre un client et un serveur sur un réseau congestionné ou lent, des paquets avec le même numéro de séquence (par exemple 0) mais appartenant à deux sessions différentes pourraient se mélanger.

Transmission des données

La transmission des données repose de manière cruciale sur les numéros de séquences et les acquitements. Supposons que X=1000 et Y=12000 pour la suite.

- à la fin de la séquence **SYN/SYN-ACK/ACK**, X=1001 et Y=12001
- si le client veut envoyer 62 octets au serveur, il crée un paquet TCP comprennant (entre autre) X=1001 puis les 62 octets de données.
- sur réception le serveur renvoie un **ACK** contenant 1063 qui est le numéro du premier octet qu'il s'attends à recevoir pour la suite.
- le client calcule 1063 - 1001 = 62 et a la certitude que le serveur a bien reçu son paquet

Détection de problème

On suppose que le client envoie trois paquets de 62 octets, avec les numéros de séquence :1001, 1063 et 1125.

◆ Si le serveur reçoit 1125, 1001 et 1063 (dans cet ordre), le numéro de séquence lui permet de réordonner.
◆ Si le serveur reçoit 1001, 1001, 1063 et 1125, le numéro de séquence lui permet de détecter le doublon
◆ Si le serveur reçoit 1125, il envoie le ACK:1126 au client et attend un certain temps. Si les paquets n'arrivent pas, le serveur renvoie de nouveau un ACK:1126. Le client comprend que le dernier paquet arrivé au serveur est 1125 donc qu'il lui manque tous les octets entre 1001 et 1125. Le client renvoie les paquets manquants.
◆ Si le client ne recoit pas un **ACK** pour chaque paquet envoyé avant un temps fixé, le paquet est envoyé de-nouveau.

Terminaison de connexion

La terminaison de connexion s'effectue de la manière suivante (on suppose que le client a décidé de terminer la connexion)

1. le client envoie **`FIN`** au serveur et attend un certain temps T_1.
2. le serveur renvoie **`ACK`** au client
3. le client attend un certain temps T_2 (pour laisser une chance au serveur de terminer lui aussi).
4. le serveur envoie **`FIN`** au client
5. le client envoie **`ACK`** au serveur. La connexion est terminée.

Après la fin de la connexion, le port utilisé reste indisponnible pendant un certain intervale de temps, pour éviter que des paquets en retard ne viennent perturber une nouvelle connexion.

Violation du modèle OSI

Modèle OSI : chaque couche fonctionne de manière transparente et indépendante de la couche inférieure. Le programme ci-dessous (pseudo-code) utilise l'établissemnt de connexion TCP comme mécanisme d'établissement de connexion au niveau application. Ce faisant, il n'échange pas suffisamment de donnée pour que TCP puisse garantir leur bon arrivage :

```
s = connect(serveur);   //création d'une socket s
send(s, donnee);        //envoie de données
close(s);               //fin de connexion
```

Ici, comme plus aucune donnée n'est échangée entre l'envoie et la fermeture TCP ne peut pas garantir que les données ont bien étés reçues. Un code correct est :

```
s = connect(serveur);   //création d'une socket s
send(s, "HELLO");        //envoie d'un message
d = receive(s);          //reception d'un ack "application"
send(s, donnee);
d = receive(s);          //reception d'un ack "application"
close(s);                //fin de connexion
```

Avantages et inconvénients de TCP

En plus des garanties présentées précédemment TCP permet :

◆ au receveur de donner à l'envoyeur une indication sur la quantité de données qu'il peut traiter (évite les problèmes de saturation des buffers)
◆ contrôle de congestion : si un participant TCP détecte trop de paquet perdu/retransmis, il adapte sa vitesse d'emission (simplifié)

TCP possède aussi des inconvénients

◆ Obtenir les garanties cause de la latence (jusqu'à plusieurs secondes pour attendre les paquets renvoyés), incompatible avec les applications temps-réel
◆ Trop complexe pour être implémenté sur du matériel embarqué ou a mémoire réduite (par exemple dans un Bios d'ordinateur)
◆ Garantit contre les erreurs mais pas les malveillances (par exemple DoS SYN-Flood)

Plan

> Concurrency is about *dealing* with lots of things at once. Parallelism is about *doing* lots of things at once.
>
> Rob Pike

- *Concurrence* : niveau logique. Plusieurs tâches (qui n'ont peut être rien à voir entre elles) *ont conscience les unes des autres* (et interfèrent entre elles).

- *Parallélisme* : plusieurs opérations ont lieu *simultanément*

Exemples

- Les différents processus d'un OS multi-tâche s'exécutent *de manière concurrente* :
  - Les programmes sont a priori indépendants
  - Il doivent se *synchroniser* pour l'accès aux ressources partagées (fichiers, périphériques, ...)
  - Existe sur des processeurs scalaires (simple) sans parallélisme
- Soit le programme C suivant :

```
void f(int x, int y, int* p) {
          int z = *p;
          x = x + z;
          y = y + z;
          *p = x + y;
          }
```

lors de l'évaluation de `f` plusieurs (sous-)opérations peuvent être effectuées en *parallèle* sur un processeur super-scalaire moderne (lesquelles) ?

- Les différents processus d'un OS multi-tâche s'exécutent en *parallèle* sur un processeur multi-cœur

Plan

Définitions

Un *thread* (tâche ou fil d'exécution en français) est la plus petite séquence d'instructions manipulable par *l'ordonnanceur* d'un système d'exploitation. Chaque *thread* possède sa propre pile d'exécution. Le tas est peut être partiellement partagé.

Un *processus* est l'exécution d'un programme. Il est usuellement composé de plusieurs *threads*. Deux processus ne partagent pas d'espace mémoire (et donc pas de variables).

Threads en Java

En Java, on peut programmer un *thread* (sous-processus) en créant une classe qui hérite de la classe **`Thread`**. Une telle classe doit redéfinir la méthode **`public void run()`** pour y placer le code à éxécuter en parallèle :

```
class MonThread extends Thread {
   MonThread() { // constructeur }

  @Override
  public void run()  {
  //Code à exécuter en parallèle
  }
}
```

Attention, la méthode **`run()`** n'a pas vocation à être appelée directement par le programmeur. Elle est appelée automatiquement par la JVM lorsque le *thread* démarre.

Création de *threads*

```
public class MonProgramme {
…

  public static void main() {
    MonThread t = new MonThread();
    t.start();
    …
  }
}
```

La méthode **`start()`** rend la main immédiatement. La méthode **`run()`** du *thread* démarré est éxécutée. Si on appelle **`start()`** plus d'une fois, l'exception **`IllegalThreadStateException`** est levée.

Gestion des *threads*

Les méthodes suivantes permettent d'influencer le comportement des *threads* :

`.join()/.join(int millis)` :
: appelée sur un objet **`Thread`** particulier attend **`millis`** millisecondes qu'il se termine (sans argument, équivalent à **`t.join(0)`** attendre jusqu'à ce que le thread se termine). Peut lever l'exception **`InterruptedException`** si un autre thread à interrompu le thread courant pendant un appel à **`.join()`**.

`Thread.sleep(int millis)` :
: (méthode statique) Le *thread* courant interrompt son exécution pendant au moins **`millis`**ms. Après ce délais, le thread courant est de nouveau exécutable (mais la JVM peut choisir de terminer d'autres opérations avant de l'exécuter de nouveau). Peut lever l'exception **`InterruptedException`** si un autre *thread* à interrompu le *thread* courant pendant un appel à **`.join()`**.

`Thread.yield()` :
: (méthode statique) permet au *thread* courant de signaler à la JVM qu'il peut être interrompu.

Accès concurrents

Pour pouvoir coopérer les *threads* doivent partager des variables et les modifier. Pour éviter les problèmes de *race conditions* on peut utiliser des méthodes synchronisées :

```
private int x;
public synchronized void incr() { x++; }
public synchronized void decr() { x--; }
```

Si deux threads appellent en même temps la méthode **`incr()`** ou **`decr()`**, les exécutions sont protégées par un verrou.

Accès concurrents (suite)

Sans utilisation de **`synchronized`** la situation suivante peu se produire. On suppose **`x=0`** :

Thread 1

```
→ appel à incr()
  →lecture de x,  x vaut 0
  →calcul x+1, vaut 1
  →écriture x = 1
```

Thread 2

```
→ appel à decr()

  →lecture de x,  x vaut 0

  →calcul x-1, vaut - 1

  →écriture x = -1
```

Après exécution, **`x`** vaut **`-1`**. Avec **`synchronized`** les appels de méthode sont mis en séquence.

synchronized, expliqué (1)

Chaque objet Java (i.e. n'importe quoi qui n'est pas un type primitif comme **`int`**, **`boolean`**, ...) possède un *verrou interne* (*intrinsic lock* ou *monitor lock*), un entier 32 bits stocké dans l'objet.
Lorsque le *thread* courant veut acquerrir le verrou:

- si le verrou vaut 0, incrémenter le verrou
- si le verrou est non nul, mais que le *thread courant* possède déjà le verrou, incrémenter le verrou
- sinon, le verrou ne peut pas être acquis (bloquer ou lever une exception selon les cas)

Pour relacher le verrou il suffit de le décrémenter

synchronized, expliqué (2)

- Lors de l'appel à une méthode *synchronized* sur un objet **o**, le *thread* courant essaye de prendre le verrou de **o** et le relache en fin d'appel (même en cas d'exception)
- Le mot clé synchronized peut aussi être utilisé sur un bloc et un objet Java *arbitraire* :

```
void f(HashMap map) {
            … //traitements longs qui ne dépendent pas de map

            synchronized (map) {
                        // seul le thread courant peut
                        // exécuter ce bloc
                         map.put(…);
                         map.get(…);
                         map.remove(…);
            }
            … //autre traitements longs
            }
```

Appels périodiques

Une utilisation courante des threads est de vouloir effectuer une tâche à *intervalles réguliers* (ex: recharger le contenu d'un fichier toutes les X secondes).
Java fournit les classes ***Timer*** et ***TimerTask*** pour ce cas d'utilisation très courant.

```
class PeriodicEcho extends TimerTask {
      private int counter = 0;
      @Override
      public void run () {
            System.out.println ("Hello numero " + counter);
            counter++;
      }
}
      …
      Timer timer = new Timer();
      timer.schedule(new PeriodicEcho(), 1000, 4000);//passe en tâche de fond
      …
      timer.cancel();
```

TimerTask et Timer

La classe contenant l'action doit étendre **`TimerTask`** et doit redéfinir la méthode ***`public void run()`***.
La classe ***`Timer`*** permet de lancer et interrompre des tâches :

.schedule(task, delay, period)
: exécute la méthode **`run()`** de **`task`** après **`delay`** milliseconde et la répète toutes les **`periode`** milliseconde avec un délais fixe relativement à la dernière tâche.

.scheduleAtFixedRate(task, delay, period)
: exécute la méthode **`run()`** de **`task`** après **`delay`** milliseconde et la répète toutes les **`periode`** milliseconde relativement au début de l'exécution.

.cancel()
: Annule toutes les tâches suivantes du **`timer`**. Si une tâche est en court d'exécution, on a la garantie que c'est la dernière exécutée.

Classes anonymes (digression)

Il est souvent inélégant de créer une classe séparée pour définir une seule méthode. On peut utiliser une classe anonyme :

```
Thread th = new Thread () {  //thread est une classe
                @Override
                public run () { … }
      };
TimerTask tt = new TimerTask () {        //TimerTask est une interface
                 @Override
                 public run () { … }
      };
```

Plan

Classes importantes

La mise en place de connexions TCP se fait de manière similaire à UDP. Cependant les rôles du serveur et du client sont encore plus assymétriques, ce qui est reflété dans la structure des classes :

ServerSocket
: objet crée par le serveur. Cette socket peut être mise en mode écoute (`.bind()`)

Socket
: objet représentant une communication entre le serveur et un client ou entre un client et le serveur

ServerSocket

La socket peut être mise en mode « écoute » au moyen de la méthode **`.bind()`**. On appelle le constructeur sans arguments

`void .bind(InetSocketAddress addr) :`

Place la socket en mode écoute sur l'adresse de socket donnée. Peut créer une adresse d'écoute de la manière suivante :

```
new InetSocketAddress(port);
```

où **`port`** est un numéro de port sur lequel écouté. Une telle adresse est associée par défaut à tous les périphériques réseau de la machine.

`Socket .accept() :`

Une fois la **`ServerSocket`** placée en mode écoute, un appel à **`.accept`** bloque jusqu'à ce qu'un client se connecte. Si une connection se produit, la méthode renvoie un objet **`Socket`** qui permet de communiquer avec ce client.

Socket

La classe **`Socket`** représente une connexion entre deux machines (un client et le serveur). Les méthodes à utiliser sont les suivantes :

`OutputStream .getOutputStream() :`
: renvoie un flux de sortie, connecté au flux d'entrée de la machine de l'autre coté de la socket.

`InputStream .getInputStream() :`
: renvoie un flux de entrée, connecté au flux de sortie de la machine de l'autre coté de la socket.

`.close() :`
: Permet de fermer la socket

On peut traîter les flux d'entrées et de sorties comme **`System.in`** et **`System.out`**. Les écritures/lectures y sont blocantes.

Question : comment la différence entre TCP et UDP est elle reflétée au niveau de l'API Java ?

Architecture générale du code

Le serveur possède deux parties bien distinctes :

1. Un *thread* principal qui utilise la **`ServerSocket`** et y attend des connexions. Sur réception d'une connexion, le serveur principal crée un nouveau *thread* qui gère uniquement cette connexion.

2. Un *thread* auxiliaire pour chaque connexion.

Le client est simple :

1. Un programme principal qui utilise la **`Socket`** s'en sert pour se connecter au serveur.

Exemple de code (Serveur)

```
class TraiteConnexion extends Thread {

Socket toClient;
PrintWriter out;
BufferedReader in;

TraiteConnexion(Socket s) {
   toClient = s;
   out = new PrintWriter(new BufferedWriter(s.getOutputStream()));
   in = new BufferedReader(new InputStreamReader(s.getInputStream()));
}

@Override
public void run () {
    //Lire et écrire sur in et out pour parler au client.
  }

}
```

Exemple de code (Serveur, suite)

```
  class MonServeur {
  …
  void serveur() {
  ServerSocket s = new ServerSocket;
  s.bind(new InetSocketAddress(10000));


  while(true) {

   Socket p = s.accept();
   //bloque jusqu'à une connexion du client.

   TraiteConnexion t = new TraiteConnexion(p);
   t.start();   //démare le thread qui va gérer cette connexion
  }
 }
}
```

Exemple de code (client)

Ce code est souvent le symétrique du code de traitement de connexions dans le serveur :

```
class MonClient  {
Socket toServer;
PrintWriter out;
BufferedReader in;
…
void client() {
…
   toServer = new Socket();
   toServer.connect(new InetSocketAddress(InetAddress.getByName(host),
                    port));
   out = new PrintWriter(new BufferedWriter(s.getOutputStream()));
   in = new BufferedReader(new InputStreamReader(s.getInputStream()));
…
    //Lire et écrire sur in et out pour parler au serveur.

}
```

Interrompre le code du serveur

Si le serveur doit périodiquement faire autre chose que d'attendre une connexion (par exemple lire son entrée standard pour voir si une touche a été pressée), on peut configurer la **`ServerSocket`** au moyen de la méthode **`.setSoTimeout(int millis)`**. Une fois fait, tout appel à **`.accept()`** sera interrompu au bout de **`millis`**ms millisecondes si aucune connexion n'arrive, avec l'exception **`SocketTimeoutException`**.