Plan

- Information générale
- Introduction
- Application aux problèmes clients-serveurs: R.P.C.
- Communication par appel de méthode: R.M.I.
- Communication par appel de méthode: CORBA.
- Web Services, SOAP, ….

Information générale

- Planning
- Cours, Tds, Tps …
- Evaluation
- ….

Introduction

"Un système réparti est un ensemble de machines autonomes connectées par un réseau, et équipées d'un logiciel dédié à la coordination des activités du système ainsi qu'au partage de ses ressources." Coulouris et al. [COU 94].

"Un système distribué ou réparti est un système dont les composants sont répartis sur différents nœuds d'un réseau d'ordinateurs. Ces composants communiquent et coordonnent leurs actions uniquement par l'échange de messages." Coulouris et al. [COU 2001].

"Un système distribué est collection d'ordinateurs indépendants qui apparaissent à l'utilisateur comme un seul système cohérent." A.Tanenbaum et M.Van Steen [TAN 2002].

Introduction

Un système réparti (ou distribué) est:

- ✓ composé de plusieurs systèmes calculatoires autonomes (sinon, non réparti)
- ✓ sans mémoire physique commune (sinon c'est un système parallèle, cas dégénéré)
- ✓ qui communiquent par l'intermédiaire d'un réseau

(quelconque)

Exemple :

- ✓ WWW, FTP, Mail
- ✓ Téléphones portables (et bornes)
- ✓ Guichet de banque, agence de voyage(B to B)
- ✓ Téléphones portables (et bornes)
- ✓ ….

Introduction

Les systèmes répartis sont populaires pour plusieurs raisons:

- Accès distant: un même service peut être utilisé par plusieurs acteurs, situés à des endroits différents,
- Redondance: des systèmes redondants permettent de pallier une faute matérielle, ou de choisir le service équivalent avec le temps de réponse le plus court,
- Performance: la mise en commun de plusieurs unités de calcul permet d'effectuer des calculs parallélisables en des temps plus courts,
- Confidentialité: les données brutes ne sont pas disponibles partout au même moment, seules certaines vues sont exportées,

Introduction

Un protocole est un « langage » de communication utilisé par deux ordinateurs pour communiquer entre eux.

Des protocoles de bas niveau sont utilisés sur les réseaux:

- Réseaux physiques (liaison spécialisée)
- Réseaux virtuels de bout en bout (X25, ATM, modem/téléphone)
- Systèmes basés sur les paquets (IP/Ethernet, RFC 1149, téléphone cellulaire)

Types :

- Unicast
- Multicast
- Anycast

La couche session est chargée d'établir et de réguler des connections multiples vers le même processus à partir d'autres processus.

Avant cette couche, le modèle s'efforce de transmettre les données au bon processus.

Quand deux processus demandent des connections simultanées a un processus unique, chacun doit être identifié individuellement.

L'un des points fort de cette couche est une sécurité relative. Si la couche transport est chargé d'établir et de maintenir la connexion entre des processus, la couche session assure la régulation des dialogues ou des conversations.

Les protocoles émergents pour cette couche sont des RPC (Remote Procedure Call). De nombreuses versions différentes sont produites par des fournisseurs tel que Sun et HP.

Comme pour la couche présentation, cette couche n'est pas identifiable sous la forme d'une couche indépendante au niveau de la hiérarchie TCP/IP.

Cette couche gère les connexions entre les applications.

Avec TCP/IP, cette fonction se réalise bien souvent dans la couche de transport et le terme de session n'est pas utilisé.

Pour TCP/IP, quelques termes sont utilisés pour décrire la voie à suivre pour que les applications puissent communiquer entres elles.

Génération de code

La mise en œuvre d'une personnalité applicative peut nécessiter la production automatique de certains composants intermédiaires entre les objets applicatifs et l'intergiciel :

- côté client : pour créer des entités représentant les objets distants (ces entités déclenchant des actions de répartition lorsqu'elles sont utilisées par les objets applicatifs locaux) ;
- côté serveur : pour créer les servants encapsulant les objets applicatifs et permettant à la couche neutre d'y faire appel, dans le cadre de la fonction d'exécution.

Dans un intergiciel traditionnel, ces composants réalisent directement des opérations d'emballage de paramètres ou de communication.

Génération de code

Le générateur de code d'une personnalité applicative se base sur une description des services offerts par un ensemble donné d'objets applicatifs, et génère le code correspondant pour les clients et pour les serveurs.

La génération de code peut être

- une étape distincte de la compilation du code applicatif (c'est le cas pour une personnalité basée sur une bibliothèque de répartition, telle que CORBA);
- intégrée de façon transparente dans le traitement du code de l'application (dans le cas d'un langage réparti).

Objectif du cours

Réaliser un service réparti en utilisant l'interface de transport (TCP, UDP)

- **Solutions**
  - Sockets : mécanisme universel de bas niveau, utilisable depuis tout langage (exemple : C, Java)
  - Appel de procédure à distance (RPC)
  - Appel de méthode à distance, dans les langages à objets ;
  - exemple : Java RMI
  - Middleware intégré : CORBA, Web service …

Un intergiciel est un programme qui s'insère entre deux applications et qui va permettre de faire communiquer deux machines entre elles, indépendamment de la nature du processeur, du système d'exploitation, voire du langage,…

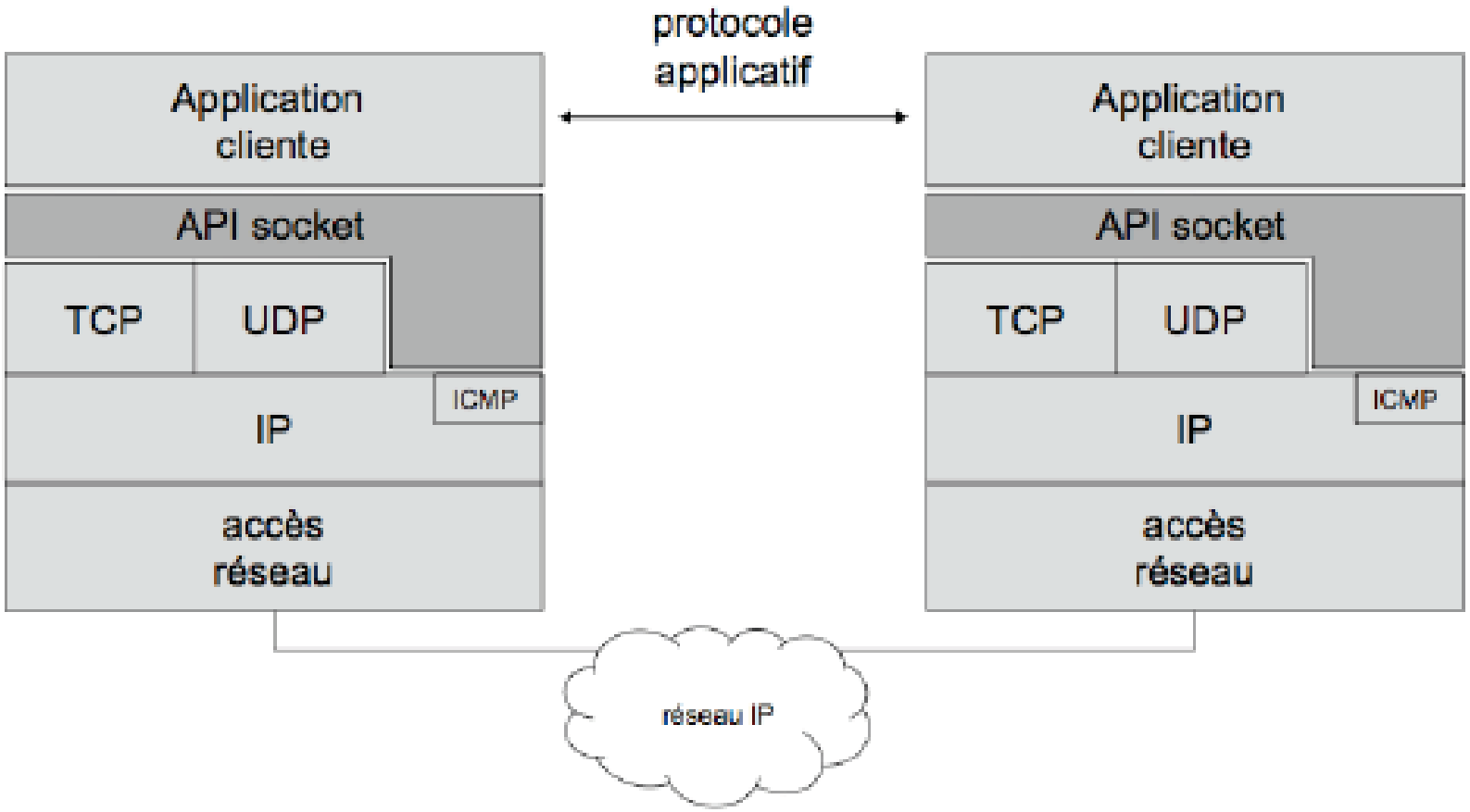
Rappel Sockets (Java)

- L'utilisation de sockets est la manière la plus ancienne de communiquer entre application réparties .

- **rôle des sockets :**
  - Connexion à une machine distante
  - Envoi/Réception de données

But : apprendre comment créer des applications client/serveur qui communiquent avec les socket

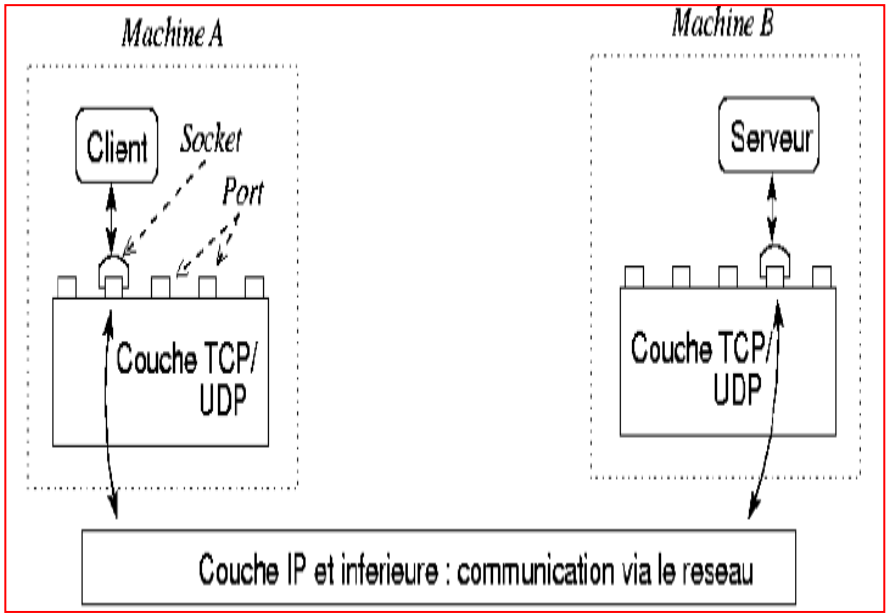
Rappel Sockets (Java)



➔ Les sockets se situent juste au-dessus de la couche transport

Rappel Sockets (Java)

- Une socket est un port de communication ouvert au niveau de couche réseau TCP/UDP ou tout simplement un point d'accès à cette couche liée localement à un port
  - -Adressage de l'application sur le réseau : son couple @IP:port



Elle permet la communication avec un port distant sur une machine distante : c'est-à-dire avec une application distante

Rappel Sockets (Java)

- On distingue deux modes de communication par socket :
  1. Communication avec TCP,
  2. Communication avec UDP.
- Principes de base :

  1- Chaque machine crée une socket. Chaque socket sera associée à un port

  2- Les deux sockets seront explicitement connectées si on utilise un protocole en mode connecté …,

  3- Le client et le serveur communiquent par socket

  Chaque machine lit et/ou écrit dans sa socket,

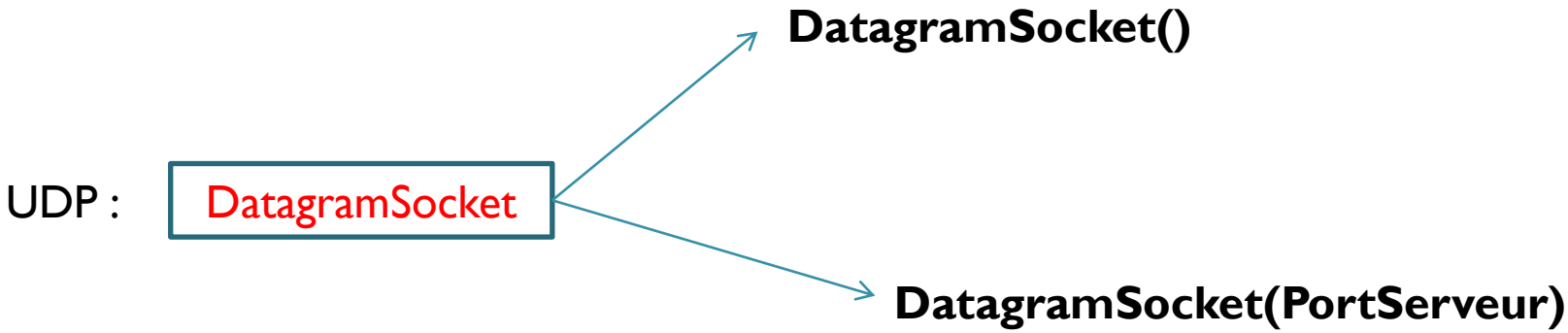
  Les données vont d'une socket à une autre à travers le réseau

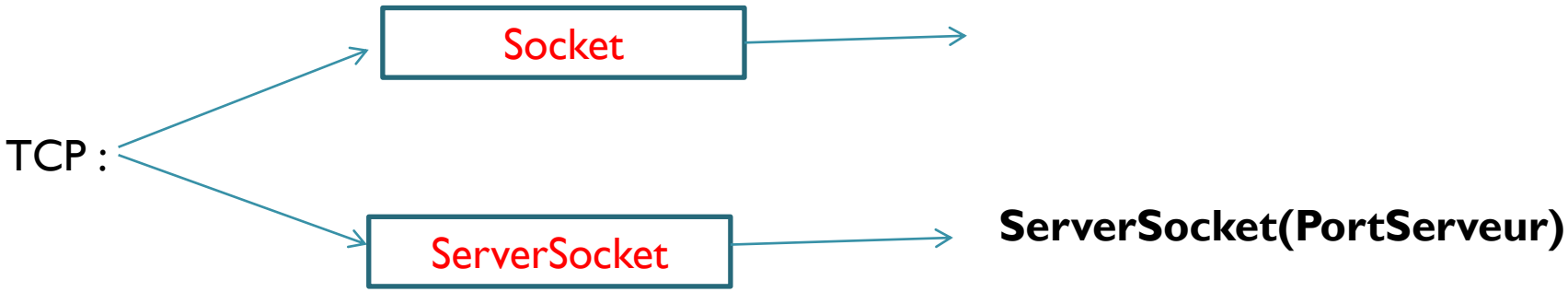
  4- Une fois terminé chaque machine ferme sa socket.

Rappel Sockets (Java)

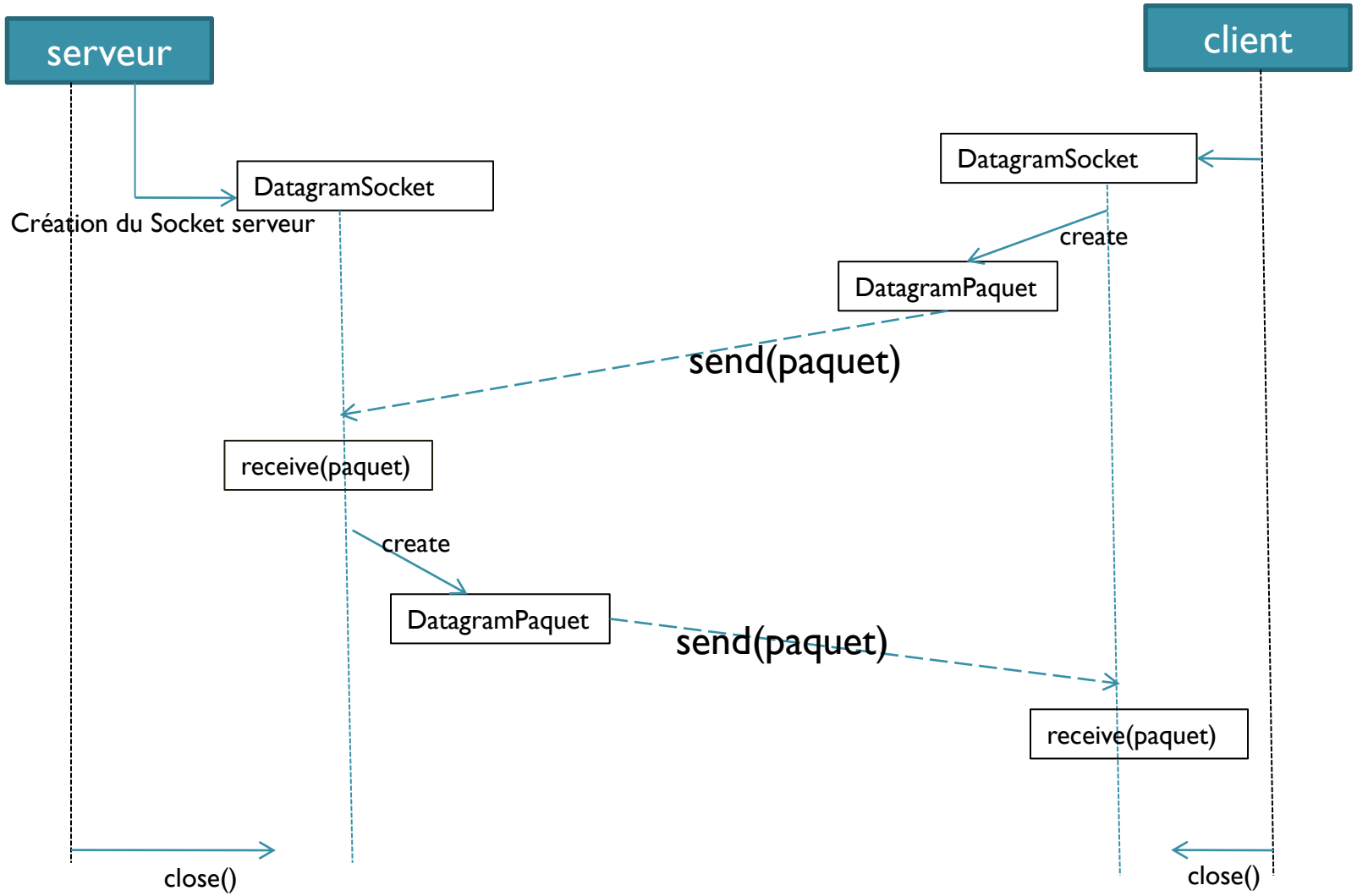
- Java intègre nativement les fonctionnalités de communication réseau au dessus de TCP-UDP/IP : Package java.net

- **Classes de sockets :**





Sockets avec UDP : les étapes



Sockets avec UDP : Les classes

- Classes utilisées pour communication via UDP
  - InetAddress : codage des adresses IP
  - DatagramSocket : socket mode non connecté (UDP)
  - DatagramPacket : paquet de données envoyé via une socket sans connexion (UDP)

- Classe InetAddress : représente les adresses IP et un ensemble de méthodes pour les manipuler. Elle encapsule aussi l'accès au serveur de noms DNS.
  - Constructeurs

    Pas de constructeurs, on passe par des méthodes statiques pour créer un objet
  - Méthodes :

    - public static InetAddress **getByName**(String host) throws UnknownHostException : retourne un objet qui contient l'adresse IP de la machine dont le nom est passé en paramètre
      - L'exception est levée si le service de nom (DNS...) du système ne trouve pas de machine du nom passé en paramètre sur le réseau
      - Si précise une adresse IP sous forme de chaîne ("192.12.23.24") au lieu de son nom, le service de nom n'est pas utilisé

Sockets avec UDP : Les classes

- public static InetAddress **getLocalHost**() throws UnknownHostException : elle retourne un objet qui contient l'adresse IP de la machine locale.
- getHostName() : retourne le nom de la machine dont l'adresse est stockée dans l'objet.
- getAddress() : elle retourne l'adresse IP stockée dans l'objet sous forme d'un tableau de 4 octets.
- toString() : elle retourne un String qui correspond au nom de la machine et son adresse.

Exemple d'utilisation d'InetAddres

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Adressage {
   public static void main(String[] zero) {
                InetAddress LocaleAdresse ;
                InetAddress ServeurAdresse;

                try {

                        LocaleAdresse = InetAddress.getLocalHost();
                        String hote=LocaleAdresse.getHostName() ;
                        System.out.println("L'adresse locale est : "+LocaleAdresse );
                        System.out.println("nom hote : "+hote );
                        ServeurAdresse= InetAddress.getByName("www.google.fr");
                        System.out.println("L'adresse du serveur google:"+ServeurAdresse);

                   } catch (UnknownHostException e) {

                                e.printStackTrace();
                   }
    }
}
```

Rappel Sockets (Java)

- Classe DatagramPacket
  - **Constructeur :**

1- DatagramPacket(byte[] buf, int length, InetAddress address, int port) :

création d'un paquet à destination d'une machine
et d'un port spécifiés

- buf : contient les données à envoyer
- length : longueur des données à envoyer
- address : adresse IP de la machine destinataire des données
- port : numéro de port distant (sur la machine destinataire) où envoyer les données

2- DatagramPacket(byte[] buf, int length) :

Création d'un paquet pour recevoir des données (sous forme d'un tableau d'octets)

Rappel Sockets (Java)

- Classe DatagramPacket (suite)
  - Méthodes « get »

    - InetAddress getAddress()
      Si paquet à envoyer : adresse de la machine destinataire
      Si paquet reçu : adresse de la machine qui a envoyé le paquet
    - int getPort()
      Si paquet à envoyer : port destinataire sur la machine distante
      Si paquet reçu : port utilisé par le programme distant pour envoyer le paquet
    - byte[] getData : Données contenues dans le paquet
    - int getLength()
      Si paquet à envoyer : longueur des données à envoyer
      Si paquet reçu : longueur des données reçues

Rappel Sockets (Java)

- **class DatagramSocket : (coté client et coté serveur)**
  - Constructeur :

    DatagramSocket(port) : Crée une nouvelle socket en la liant au port local précisé par le paramètre port

    DatagramSocket( ) : Crée une nouvelle socket en la liant au un libre

    ➔ throws SocketException : exception levé e n cas de problème

  - Méthodes :

    close() : ferme la socket.

    receive(DatagramPacket p) : reçoit un « DatagramPacket » de cette socket.

    send(DatagramPacket p): envoi un « DatagramPacket » passé en paramètre.

    Le destinataire est identifié par le couple @IP/port précisé dans le paquet

>> *En pratique*

Pour envoyer des datagrammes en Java nous devons suivre les étapes suivantes :

- Obtenir l'adresse du destinataire et la mettre dans une instance de la classe InetAddress.

- Mettre les données et l'adresse dans une instance de la classe DatagramPacket

- Créer une instance de la classe DatagramSocket et lui confier l'envoi du datagramme

Pour recevoir des datagrammes en Java nous devons suivre les étapes suivantes :

- Créer une instance de la classe de la classe DatagramSocket qui devra attendre l'arrivée de données à travers le réseau.

- Créer une instance de la classe DatagramPacket qui recevra données qui lui seront passées par l'instance de DatagramSocket.

- **programme client**

création d'une socket, sans la lier à un port particulier

contient l'@IP du serveur. Traduire le nom du poste en adresse IP utilisant DNS

données à envoyer

Créer le datagramme avec donnée à envoyer, sa longueur, @IP et port

envoi du paquet via la socket

Mettre le nom du serveur

```
import java.io.*;
import java.net.*;
class ClientUDP {
        public static void main(String args[]) throws Exception
         {
            DatagramSocket  socketClient = new DatagramSocket();

            InetAddress AdresseIP = InetAddress.getByName("localhost");

            byte[] donneeEnvoyee = new byte[1024];
            String phrase ="Salut!, mode datagramme en illustration« ;
            donneeEnvoyee = phrase.getBytes();

            DatagramPacket sendPacket =
            new DatagramPacket(donneeEnvoyee, donneeEnvoyee.length, AdresseIP, 9877);

            socketClient.send(sendPacket);

            socketClient.close();

         }
}
```

Rappel Sockets (Java)

La première chose que nous faisons est de résoudre le nom de la machine destinatrice en adresse réseau. Cette adresse sera placée dans un objet de type InetAddress.

Ensuite nous transformons notre chaîne de caractères en une suite d'octets afin de les transmettre sur le réseau. N'oublions pas que le réseau ne comprend pas les caractères Unicode (ni toute autre forme d'objet ou de variable).
En effet tout ce qu'un réseau comprend est une suite d'octets qui transitent. Cette transformation se fait par la méthode getBytes() de la classe String.

Il nous faut ensuite construire un datagramme en indiquant l'emplacement des données à transmettre, leur longueur et enfin l'adresse et le port de destination.

Enfin nous ouvrons une DatagramSocket et utilisons sa méthode send() pour envoyer notre datagramme.

Côté réception (voir le code serveur ci-après) :
Nous préparons une zone mémoire tampon pour recevoir les données. Ensuite nous créons un DatagramSocket pour écouter sur le port de destination en attente de données. L'ordre d'attente se fait par la méthode receive().

Cette méthode se charge de placer les données dans un DatagramPacket gérant lui même notre tampon (socketServeur.receive(paquetRecu);).

Enfin les données sont extraites du tampon sous forme de chaîne de caractères et imprimées à l'écran.

Sockets avec UDP : Exemple

- **programme serveur**

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
class ServeurUDP1 {
        public static void main(String args[]) throws Exception
        {
                DatagramSocket socketServeur = new DatagramSocket(9877);
                byte[] donneeRecue = new byte[1024];

                while(true)
                {

                        DatagramPacket paquetRecu =
                        new DatagramPacket(donneeRecue, donneeRecue.length);

                        socketServeur.receive(paquetRecu);

                        String phrase = new String(paquetRecu.getData());
                        System.out.println(" Message : "+phrase);


                        InetAddress adresseIP = paquetRecu.getAddress();
                        int port = paquetRecu.getPort();
                        System.out.println(" ca vient de :"+ adresseIP +":"+ port);


                }
        }
}
```

création d'une socket, sans la lier à un port particulier

Tableau d'octets qui contiendra les données reçues

création d'un paquet en utilisant le tableau d'octets

Recevoir le datagramme

Récupération des données

Extraire l'adresse IP, numéro de port de l'expéditeur

Sockets avec UDP

- Avantages
  - Simple à programmer (et à appréhender)
- Inconvénients
  - Pas fiable
  - Ne permet d'envoyer que des tableaux de byte

Structure des données échangées

- Format des données à transmettre

  Très limité a priori : tableaux de byte

  Et attention particulière à la taille réservée : si le récepteur réserve un tableau trop petit par rapport à celui envoyé, une partie des données est perdue

➔ necessité de conversion :

  - à l'envoi : un objet quelconque en byte[] pour l'envoyer
  - à la réception : un byte[] en un objet d'un certain type

- solutions :
  - créer les méthodes qui font cela : lourd et dommage de faire des tâches de si « bas-niveau » avec un langage évolué comme Java
  - utiliser les flux Java pour conversion automatique

Sockets avec UDP

Conversion Object <-> byte[]

➔ Pour émettre et recevoir n'importe quel objet via des sockets UDP

En écriture : conversion de Object en byte[]

```
ByteArrayOutputStream byteStream =new ByteArrayOutputStream();
ObjectOutputStream objectStream =new ObjectOutputStream(byteStream);
objectStream.writeObject(object); // objet=objet à envoyé
byte[] donneeEnvoyee = byteStream.toByteArray();
```

En lecture : conversion de byte[] en Object

```
ByteArrayInputStream byteStream =new
ByteArrayInputStream(donneeRecue);
ObjectInputStream objectStream =new ObjectInputStream(byteStream);
Objetct object = objectStream.readObject();
```

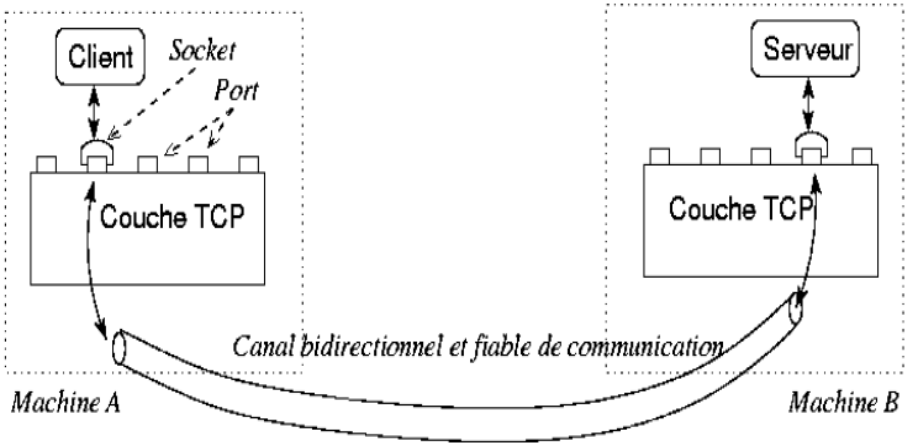
Puis manipuler le contenu de l'objet :

Exemple : si l'objet devait contenir un tableau de string

```
String[] phrase=null;
    if (object  instanceof String[]) phrase = (String[]) object;
```

Sockets TCP : principe

- Fonctionnement en mode connecté
  - Données envoyées dans un « tuyau » et non pas par paquet
  - Fiable : la couche TCP assure que
    - *) Les données envoyées sont toutes reçues par la machine destinataire
    - *) Les données sont reçues dans l'ordre où elles ont été envoyées



- La communication se fait en mode client/serveur. Le serveur écoute les requêtes des clients sur un port défini par l'application

Sockets TCP : principe

- Principe de communication

1- Le serveur crée une socket serveur(socket d'écoute) et la lie une sur un certain port bien précis

2- Le serveur reste en écoute et attend une demande de connexion (appelle un service d'attente de connexion de la part d'un client)

3- Le client appelle un service pour ouvrir une connexion avec le serveur : Il récupère une socket (associée à un port)

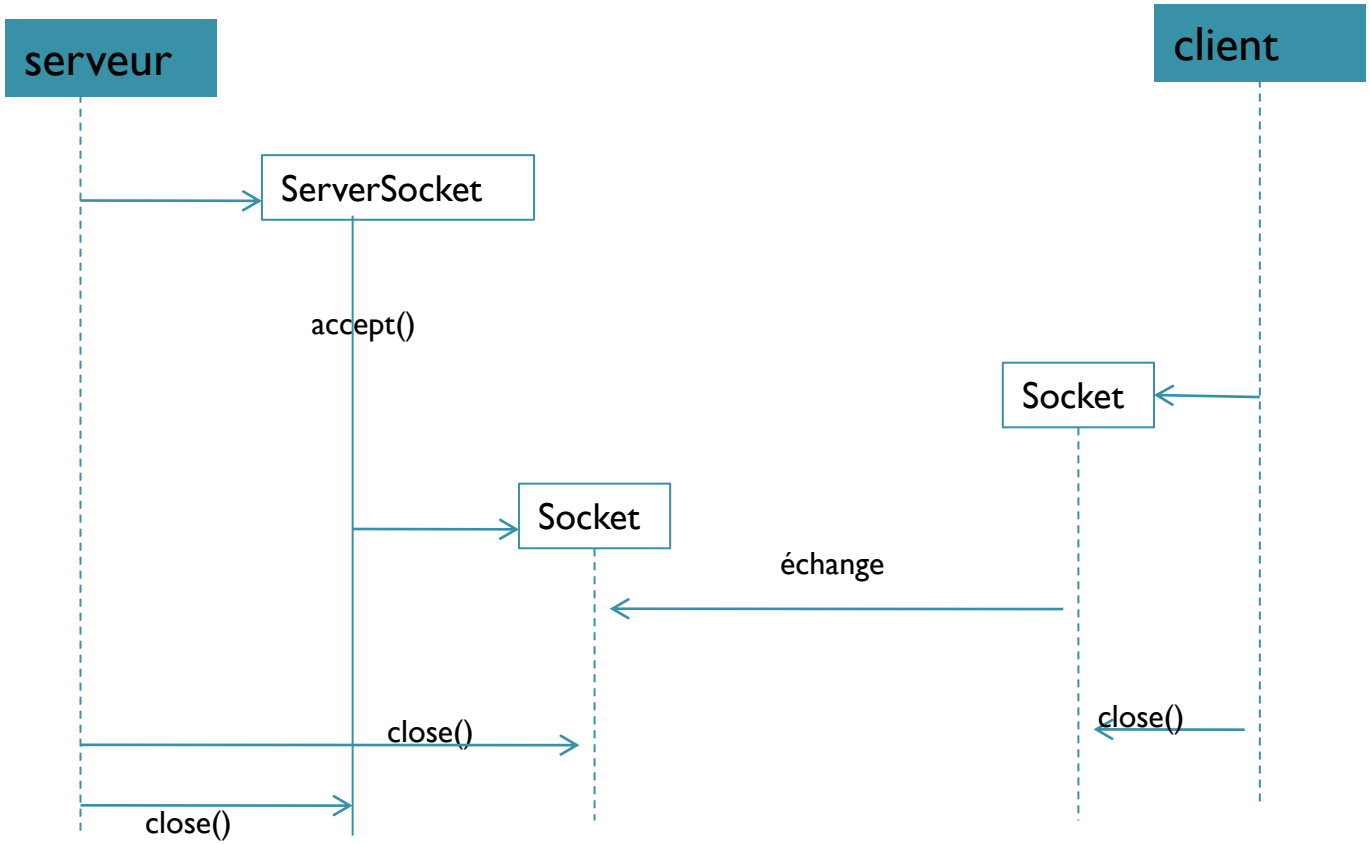
4-Le serveur accepte la connexion et le service d'attente de connexion retourne une socket de service (associée à un port quelconque) : C'est la socket qui permet de dialoguer avec ce client

5- Le client et le serveur dialoguent en envoyant et recevant des données via leur socket

6- A la fin des échange on ferme (close) les sockets client coté client, socket de service et socket serveur coté client.

Sockets TCP : principe

Schéma



Sockets TCP : principe

- Classes du package java.net utilisées pour communication via TCP

- InetAddress : codage des adresses IP
  Même classe que celle décrite dans la partie UDP et usage identique
- Socket : socket mode connecté
- ServerSocket : socket d'attente de connexion du coté serveur

Sockets TCP : principe

- Classe SeverSocket (coté serveur)

Constructeurs

public ServerSocket(int port) throws IOException :

Crée une socket d'écoute (d'attente de connexion de la part de clients) liée au port dont le numéro est passé en paramètre

L'exception est levée notamment si ce port est déjà lié à une socket

Méthodes

- Socket accept() throws IOException :

Ecoute et attend une requête de connexion d'un client distant

Quand connexion est faite, retourne une socket sur laquelle écouter le nouveau client (et lui seul) permettant ainsi de communiquer avec ce dernier : socket de service

- void setSoTimeout(int timeout) throws SocketException :

Positionne le temps maximum d'attente de connexion sur un accept

Si temps écoulé, l'accept lève l'exception SocketTimeoutException

Par défaut, attente infinie sur l'accept

Sockets TCP : principe

- Classe Socket (coté client et coté serveur)

  Constructeurs

  1- Socket(InetAddress address, int port) throws IOException :
  Crée une socket locale et la connecte à un port distant d'une machine distante identifié par le couple address/port

  2- public Socket(String address, int port) throws IOException,UnknownHostException
  Idem mais avec nom de la machine au lieu de son adresse IP codée
  - Lève l'exception UnknownHostException si le service de nom ne parvient pas à identifier la machine

Sockets TCP : principe

- Classe Socket (suite)

Méthodes :

-Méthodes d'émission/réception de données :

les sockets TCP n'offre pas directement de services pour émettre/recevoir des données (contrairement aux sockets UDP) : on récupère les flux d'entrée/sorties associés à la socket avec :

OutputStream getOutputStream() : revoie un flux de sortie pour cette socket.

IntputStream getIntputStream() : revoie un flux d'entrée pour cette socket.

- close() : ferme la socket
- int getPort() : renvoie le port distant avec lequel est connecté la socket
- InetAddress getAddress() : Renvoie l'adresse IP de la machine distante
- int getLocalPort() : Renvoie le port local sur lequel est liée la socket
- public void setSoTimeout(int timeout) : Positionne l'attente maximale en réception de données sur le flux d'entrée de la socket

Si temps dépassé lors d'une lecture : exception SocketTimeoutException est levée

Par défaut : temps infini en lecture sur le flux

Sockets TCP : principe

- Avantages sockets TCP

  - Mode connecté avec phase de connexion explicite
  - Flux d'entrée/sortie
  - Fiable

- Inconvénients
  - Plus difficile de gérer plusieurs clients en même temps
  - Nécessite du parallélisme avec des threads (voir suite cours)
  - Mais oblige une bonne structuration coté serveur

Sockets UDP ou TCP

- Comme on vient de voir UDP est à priori simple
- TCP est fiable et mieux structuré
- intérêt tout de même pour UDP dans certains cas
  - Si la fiabilité n'est pas essentielle
  - Si la connexion entre les 2 applications n'est pas utile

Exemple

un thermomètre envoie toutes les 5 secondes la température de l'air ambiant à un afficheur distant => on peut se permettre de perdre une mesure de temps en temps

=> pas grave d'envoyer les mesures même si l'afficheur est absent

- Serveur

Exemple

```
import java.io.*;
import java.net.*;

        class ServeurTCP{
            public static void main(String[] args) throws Exception
            {

        String recept;
        int port= 7000;
        ServerSocket socketEcoute = new ServerSocket(port);
        System.out.println("serveur prêt :");
        Socket   socketService= socketEcoute.accept();
        // affiche les coordonnées du client qui vient de se connecter
         System.out.println(" Ce client vient de se connecter:" +socketService.getInetAddress()+
        "/port:"+socketService.getPort());

        // connexion acceptée : récupère les flux pour communiquer avec le client
        BufferedReader entreeDepuisClient=new BufferedReader(
                                        new InputStreamReader(socketService.getInputStream()) );


         recept= entreeDepuisClient.readLine();
        System.out.println("Réçu du client: " + recept);

        socketService.close();

            }
        }
```

Créer une socket serveur d'écoute sur le port 7000

Attendre une connexion d'un client

Créer le flux d'entrée attaché à la socket

Lire la chaîne depuis la socket

Fermeture des sockets de service et d'écoute

Rappel Sockets (Java)

- Client

Créer
le flux d'entrée

ouverture de
connexion avec le
serveur sur le port
7000

Créer
le flux de sortie
attaché à la socket

Envoyer la
chaîne
au serveur

Fermeture de
la socket

```
import java.io.*;
import java.net.*;
class ClientTCP {
   public static void main(String[] args) throws Exception
   {
        String phrase;
        System.out.println("client >> saisir votre message :");
        BufferedReader entreeDepuisUtilisateur =new BufferedReader(new
        InputStreamReader(System.in));
        phrase = entreeDepuisUtilisateur .readLine();
        // InetAddress adr = InetAddress.getByName(« nomServeur");

        Socket socketClient = new Socket("localhost", 7000);
        //Socket socketClient = new Socket(adr, 7000);


        PrintWriter sortieVersServeur =new PrintWriter(socketClient.getOutputStream());

        sortieVersServeur.println(phrase);
        sortieVersServeur.flush();  //  vider le tompon

        socketClient.close();
   }
}
```

>> Autre exemple : avec l'utilisation de ObjectOutputStream et ObjectInputStrean

- client

```
import java.io.*;
import java.net.*;
public class Client {
        public static void main(String[] args) throws Exception{
                Socket socket =new Socket("localhost",9000);

                ObjectOutputStream output =new
              ObjectOutputStream(socket.getOutputStream());
                String[] tab={"3","+","16"};
                 // écriture d'une chaîne dans le flux de sortie : c'est-à-dire envoi de
                // données au serveur

                output.writeObject(tab);
                socket.close();

        }

}
```

Rappel Sockets (Java)

- serveur

```
import java.io.*;
import java.net.*;
public class Serveur{
public static void main(String[] args) throws Exception{
                  // serveur positionne sa socket d'écoute sur le port local 9000
                  ServerSocket serverSocket = new ServerSocket(9000);
                  System.out.println(" serveur prêt :" );
                  // se met en attente de connexion de la part d'un client distant
                  Socket socket = serverSocket.accept();
                // construction de flux objets à partir des flux de la socket
                  ObjectInputStream input =new ObjectInputStream(socket.getInputStream());
                  // attente les données venant du client
                 // ici réception des données dans un tableau
                  String[] chaine = (String[])input.readObject();
                  for(int i=0; i<chaine.length;i++)
                    System.out.println(" reçu : "+chaine[i]);
                socket.close();
                serverSocket.close();
        }
}
```

- **A la manipulation des flux d'objets :**

  Souvent utile de vérifier le type de l'objet reçu : utiliser instanceof

  Exemple :

```
String chaine; Personne pers;
Object obj = input.readObject();
if (obj instanceof String) chaine = (String)obj;
if (obj instanceof Personne) pers = (Personne)obj;
```

- Quand on manipule des flux d'objets
  - Souvent il est utile de vérifier le type de l'objet reçu
  - Utilise instanceof
  - Exemple :

```
String chaine; Personne pers;
  Object obj = input.readObject();
  if (obj instanceof String) chaine = (String) obj;
  if (obj instanceof Personne) pers = (Personne) obj;
```

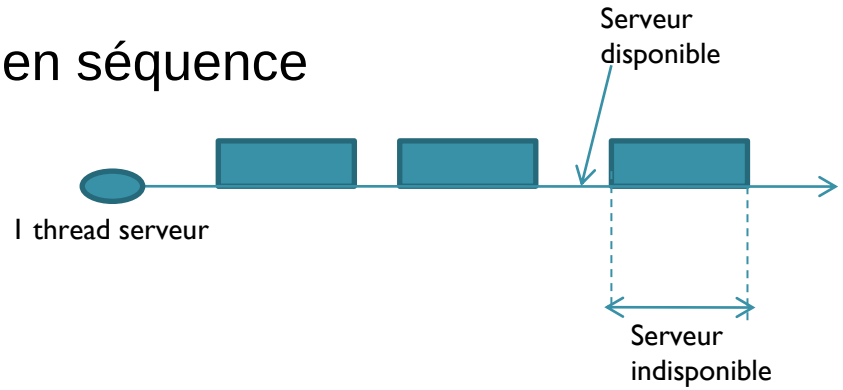
gestion plusieurs clients :Threads

-Le programme serveur précédent sert un seul client et puis s'arrête
➔ pas d'intérêt pratique

- Une première idée est de boucler en séquence sur le accept() afin de permettre des connexion successives de plusieurs client :
=> mettre le bloc entre ces deux ligne en rouge comme suit :

```
while(true){ Socket   socketService= socketEcoute.accept();
                ….
               socketService.close();
            }
        ….
```

➔ Traitement de requêtes en séquence



gestion plusieurs clients : Threads

➔Solution : utilisation des threads

- Motivation

  - Accepter (et servir) plusieurs connexions simultanées de plusieurs clients

- Méthode

  - Le serveur reste à l'écoute (socket serveur), *attend les demandes de* connexions
  - Délègue aux Thread : dès qu'un client souhaite se connecter avec le serveur, un Thread s'occupe de la connexion, et prend en charge le dialogue avec le client
  - La socket serveur retourne en attente d 'une nouvelle demande de connexion

gestion plusieurs clients :Threads

- 1ere méthode : objet héritant de la classe Thread

```
public class nomThread extends Thread {
        public void  run() {
                              //traitement
                             }
}
```

- Programme lanceur :

```
nomThread t= new nomThread();
t.start();
```

!! on n'appelle jamais directement run (start() le fait) !!

- 2eme possibilité : objet implémentant l'interface Runnable

```
class nomClasse implements Runnable{
                …
        public void run() {
                              //traitement
                             }
}
```

- Programme lanceur :

```
Thread t = new Thread(new nomClasse(…));

t.start();
```

gestion plusieurs clients : Threads

- **Exemple appliqué au programme serveur précédent:**

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.*;

public class ServeurThread1 {

        public static void main(String[] args){
        ServerSocket socketServeur=null;
        Socket socket=null;
                try {
                        socketServeur = new ServerSocket(2009);
                        System.out.println("serveur prêt :");
                        int nbrclient=0;
                        while(true){
                                socket= socketServeur.accept();
                                nbrclient++;
                                System.out.println("Le client numéro "+nbrclient+ " est connecté !");
                                Thread t = new Thread(new Accepter_clients(socket, nbrclient));
                                t.start();
                         }

                } catch (IOException e) {
                     e.printStackTrace();
                }
        }
}
```

gestion plusieurs clients :Threads

```
class Accepter_clients implements Runnable {
        private Socket socket;
        private int nbrclient;
        public Accepter_clients(Socket s, int nucl){
            socket = s;
            nbrclient=nucl;
          }

      public void run() {

            try {

                  BufferedReader entreeDepuisClient=new BufferedReader( new
  InputStreamReader(socket.getInputStream()) );

                  String recept= entreeDepuisClient.readLine();
                   System.out.println(" le mess du client "+nbrclient +":"+ recept);

                 socket.close();
                 System.out.println("thread"+nbrclient+"fermé:");

            } catch (IOException e) {
                  e.printStackTrace();
             }
      }

}
```

Remote Procedure Call (RPC)

L'appel de fonction est un mécanisme très bien connu des programmeurs.

→ *programmer des applications distribuées en appelant des fonctions qui sont situées sur une machine distante, ce qui explique le nom de "Remote Procedure Call".*

Vous pouvez utiliser les RPC pour toutes sortes d'applications distribuées, par exemple l'utilisation d'un ordinateur très puissant pour des calculs intensifs (décryptage de données, calculs numériques, ...). Cet ordinateur sera donc le serveur. Un autre ordinateur sera le client et appellera la procédure distante pour commander des calculs au serveur et récupérer le résultat.

Remote Procedure Call (RPC)

L'idée de Remote Procedure Call n'est pas nouvelle. De fait, de nombreux systèmes RPC sont disponibles (et incompatibles entre eux).

Nous allons étudier ici le système de Sun Microsystems ou "Sun RPC" qui est devenu un standard car ses spécifications sont dans le domaine public.

Ce système a été développé pour servir de base au système NFS (Network File System) de Sun, largement utilisé sous Linux.

Principe de fonctionnement

Le système RPC s'efforce de maintenir le plus possible la sémantique habituelle des appels de fonction, autrement dit tout doit être le plus transparent possible pour le programmeur.

Pour que cela ressemble à un appel de fonction local, il existe dans le programme client une fonction locale qui a le même nom que la fonction distante et qui, en réalité, appelle d'autres fonctions de la bibliothèque RPC qui prennent en charge les connexions réseaux, le passage des paramètres et le retour des résultats.

De même, côté serveur il suffira (à quelques exceptions près) d'écrire une fonction comme on en écrit tous les jours, un processus se chargeant d'attendre les connexions clientes et d'appeler votre fonction avec les bons paramètres. Il se chargera ensuite de renvoyer les résultats. Les fonctions qui prennent en charge les connexions réseaux sont des "stub". Il faut donc écrire un stub client et un stub serveur en plus du programme client et de la fonction distante.

Le travail nécessaire à la construction des stubs client et serveur sera automatisé grâce au programme **rpcgen** qui produira du code C qu'il suffira alors de compiler. Il ne restera plus qu'à écrire le programme client qui appelle la fonction et en utilise le résultat (par exemple en l'affichant) et la fonction elle-même.

Interface

Pour comprendre le fonctionnement des RPC, nous allons donc écrire, à titre d'exemple, un programme simple mais complet .

Il y a en fait deux programmes, un programme client et un programme serveur.

La fonction distante prendra deux nombres en paramètres et renverra leur somme ainsi qu'un code d'erreur indiquant s'il y a eu un overflow ou non.

Le travail commence donc par la définition de l'interface, celle-ci étant écrite en utilisant l'IDL (Interface Definition Language) du système RPC (proche du C).

Remote Procedure Call (RPC)

```
struct data {
  unsigned int arg1;
  unsigned int arg2;
};
typedef struct data data;
struct reponse {
  unsigned int somme;
  int errno;
};
typedef struct reponse reponse;
program CALCUL{
  version VERSION_UN{
    void CALCUL_NULL(void) = 0;
    reponse CALCUL_ADDITION(data) = 1;
  } = 1;
} = 0x20000001;
```

Cette définition est enregistrée dans le fichier calcul.x .

Ce fichier décrit notre programme et les fonctions qu'il contient. La définition parle d'elle-même. Notre programme s'appelle CALCUL et dans sa version VERSION_UN contient deux procédures: CALCUL_NULL et CALCUL_ADDITION.

Le programme a un nom (ici CALCUL) et un numéro (ici 0x20000001), ce numéro identifie ce programme de manière unique dans le monde. C'est pratique pour des programmes comme le daemon NFS. Dans notre cas, le numéro est choisi dans l'intervalle allant de 0x20000000 à 0x3FFFFFFF, réservé pour les utilisateurs et ne risque pas d'entrer en conflit avec des programmes tournant déjà sur votre machine.

Ensuite, pour un programme donné, on peut avoir plusieurs versions, ceci afin de pouvoir offrir de nouvelles fonctions dans la nouvelle version tout en conservant les versions précédentes (pour les anciens programmes clients). Ici, nous avons donc une version appelée VERSION_UN et qui a pour numéro 1.

Remote Procedure Call (RPC)

Vient ensuite, la liste des procédures que le serveur implémente. Chaque procédure a un nom et un numéro. Une procédure de numéro 0 et qui ne fait rien (ici CALCUL_NULL) est toujours requise. Ceci afin de tester si le système marche (une sorte de ping en quelque sorte). On peut l'appeler afin de vérifier que le réseau fonctionne et que le serveur tourne. La seconde procédure décrite est notre procédure d'addition. Elle prend un argument de type data et renvoie un argument de type reponse. Le système RPC n'autorise qu'un seul argument en paramètre et un seul en retour, pour passer plusieurs arguments (dans notre cas les deux nombres à additionner), on utilise donc une structure. De même pour renvoyer plusieurs valeurs (dans notre cas le résultat de l'addition et un code d'erreur), on utilise une structure .

Ce fichier de définition est ensuite traité par l'utilitaire rpcgen (RPC program generator), il suffit de taper sur la ligne de commande:

rpcgen -a calcul.x

L'option -a permet de produire un squelette pour notre programme client (calcul_client.c) et un squelette pour la fonction distante (calcul_server.c). Avec ou sans cette option, les fichiers suivants sont également produits: calcul.h (entête), calcul_clnt.c (stub client), calcul_svc.c (stub serveur) et calcul_xdr.c (routines XDR).

Le format XDR (eXternal Data Representation) définit les types utilisés pour l'échange de variables entre le client et le serveur. Il est possible que les processus serveur et client ne tournent pas sur la même plateforme, il est donc indispensable de parler une langue commune. Ainsi ce format définit précisément un codage pour les entiers, les flottants... Les structures plus complexes utilisent les types de base. Ainsi, les types que nous avons nous-mêmes définis nécessitent un filtre XDR, c'est le rôle des fonctions définies dans le fichier calcul_xdr.c, à compiler puis à lier avec le client et le serveur. La compilation peut être faite maintenant (c'est déjà ça) et produit un calcul_xdr.o :

```
gcc -c calcul_xdr.c
```

Les stubs client et serveur sont complets et peuvent déjà être compilés. La seule connaissance de l'interface suffit à rpcgen pour les générer. Donc, nous pouvons compiler pour produire les fichiers calcul_clnt.o et calcul_svc.o :

```
gcc -c calcul_clnt.c
gcc -c calcul_svc.c
```

Processus serveur

Ceci étant fait, écrivons maintenant la fonction distante qui effectue réellement le travail. Grâce à l'option -a de rpcgen nous avons le squelette de fonction suivant dans le fichier calcul_server.c:

/*This is sample code generated by rpcgen. These are only templates and you can use them as a guideline for developing your own functions.*/

```
#include "calcul.h"
void * calcul_null_1_svc(void *argp, struct svc_req *rqstp) {
  static char* result;
  /* insert server code here   */
  return((void*) &result); }
 reponse * calcul_addition_1_svc(data *argp, struct svc_req *rqstp) {
  static reponse  result;
  /* insert server code here  */
  return(&result);
}
```

Après examen du fichier produit par rpcgen, quelques explications s'imposent, car on peut remarquer quelques différences par rapport à notre définition. Les fonctions ont deux arguments, le deuxième n'a pas d'utilité pour notre exemple. Au lieu de récupérer une variable du type demandé dans la définition, on doit en fait passer par un pointeur sur cette variable (ceci évite une copie des arguments et donc il y a un gain de temps et de mémoire). Pour le retour, c'est également un pointeur qui est renvoyé. Comme on utilise des pointeurs, la variable dans laquelle on met la valeur de retour est déclarée 'static' car il faut bien évidemment passer l'adresse d'une variable qui existe encore après la fin de la fonction.

Comme vous le voyez, il n'y a pas de fonction main. La fonction main est située dans le stub serveur. Le stub serveur s'occupe de recevoir et de "dispatcher" les appels aux fonctions adéquates. Notre seul travail est d'écrire les fonctions.

Donc, après modifications, le fichier calcul_server.c devient:

Remote Procedure Call (RPC)

```
#include "calcul.h”
void * calcul_null_1_svc(void *argp, struct svc_req *rqstp){
  static char* result;
  /* Ne rien faire */
  return((void*) &result); }
reponse * calcul_addition_1_svc(data *argp, struct svc_req *rqstp){
  static reponse  result;
  unsigned int max;
  result.errno = 0; /* Pas d'erreur */
 /* Prend le max */
  max = argp->arg1 > argp->arg2 ? argp->arg1 : argp->arg2;
   /* On additionne */
  result.somme = argp->arg1 + argp->arg2;
   /* Overflow ? */
  if ( result.somme < max ) {     result.errno = 1;   }
  return(&result); }
```

Nous pouvons alors le compiler en faisant:

gcc -c calcul_server.c

Puis pour obtenir le programme serveur complet, il faut lier calcul_svc.o, calcul_server.o et calcul_xdr.o ensemble:

gcc -o server calcul_svc.o calcul_server.o calcul_xdr.o

A ce stade, nous avons terminé la partie serveur de notre application. On peut alors démarrer le serveur, puis utiliser rpcinfo pour vérifier qu'il tourne:

```
> ./server &
[2] 209
> rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  rpcbind
    100000    2   udp    111  rpcbind
    100005    1   udp    673  mountd
    100005    2   udp    673  mountd
    100005    1   tcp    676  mountd
```

```
100005    2   tcp    676  mountd
   100003    2   udp   2049  nfs
   100003    2   tcp   2049  nfs
 536870913    1   udp    897
 536870913    1   tcp    899
```

> rpcinfo -u localhost 536870913

program 536870913 version 1 ready and waiting

L'option -p de rpcinfo permet de connaître la liste des programmes RPC actuellement enregistrés sur la machine. Pour chaque programme, on obtient le numéro de version, le protocole et le port utilisés. Veuillez noter, que les numéros de programmes sont affichés en décimal . Sachant que 536870913 est l'équivalent décimal de 0x20000001, tout va bien notre programme est bien enregistré. L'option -u quant à elle, permet de tester le programme indiqué en appelant sa procédure 0 (rappelez vous qu'il est obligatoire d'avoir au moins une procédure de numéro 0). Pour plus d'information, vous pouvez consulter la page man de rpcinfo.

Processus client

Là encore, pour simplifier, on utilise le squelette que nous a fourni rpcgen en produisant le fichier calcul_client.c :

```
/* This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions. */
#include "calcul.h"
Void  calcul_1( char* host ){
  CLIENT *clnt;
  void  *result_1;
  char*  calcul_null_1_arg;
  reponse  *result_2;
  data  calcul_addition_1_arg;
clnt = clnt_create(host, CALCUL, VERSION_UN, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror(host); exit(1); }
```

Remote Procedure Call (RPC)

```
result_1 = calcul_null_1((void*)&calcul_null_1_arg, clnt);
  if (result_1 == NULL) {  clnt_perror(clnt, "call failed:"); }
  result_2 = calcul_addition_1(&calcul_addition_1_arg, clnt);
  if (result_2 == NULL) { clnt_perror(clnt, "call failed:"); }
  clnt_destroy( clnt );
}
main( int argc, char* argv[] )
{
  char *host;
  if(argc < 2) {
    printf("usage: %s server_host\n", argv[0]);
    exit(1);
  }
  host = argv[1];
  calcul_1( host );
}
```

Des variables sont déclarées pour les arguments et les valeurs de retour. Comme vous pouvez le constater le programme squelette généré comprend un appel à chacune des fonctions définies dans l'interface (ici CALCUL_NULL et CALCUL_ADDITION). On peut remarquer que chaque appel est suivi d'un test qui détecte les erreurs de niveau "RPC" (serveur ne répondant pas, machine inexistante, ...). L'erreur éventuelle est alors explicitée par la fonction clnt_perror(). Quand une erreur de niveau "RPC" se produit, le pointeur renvoyé est à NULL. Dans vos fonctions, vous ne devez donc pas mettre le pointeur à NULL pour spécifier une erreur. Pour un niveau d'erreur autre que RPC, vous pouvez utiliser une valeur particulière de la valeur de retour (comme un nombre négatif par exemple) et renvoyer tout à fait normalement le pointeur sur cette variable. Dans l'exemple, le choix a été fait d'utiliser une deuxième variable (champ errno de la structure reponse) pour spécifier s'il y a eu une erreur ou non (car l'utilisation de valeurs particulières est discutable).

Remote Procedure Call (RPC)

Pour faire un vrai programme, il nous faut donner des valeurs aux paramètres et il faut utiliser effectivement les résultats des appels distants (par exemple en les affichant à l'écran). C'est ce que nous faisons avec notre programme client (dans lequel il n'y a volontairement pas d'appel à la procédure CALCUL_NULL) :

```
#include <limits.h>
#include "calcul.h”
CLIENT *clnt;
void
test_addition (uint param1, uint param2)
{
  reponse  *resultat;
  data  parametre;
 /* 1. Preparer les arguments */
  parametre.arg1 = param1;
  parametre.arg2 = param2;
  printf("Appel de la fonction CALCUL_ADDITION avec les parametres: %u et %u \n",
    parametre.arg1,parametre.arg2);
```

```
/* 2. Appel de la fonction distante */
    resultat = calcul_addition_1 (&parametre, clnt);
  if (resultat == (reponse *) NULL) {
    clnt_perror (clnt, "call failed");
    clnt_destroy (clnt);
    exit(EXIT_FAILURE);
  }
  else if ( resultat->errno == 0 ) {
    printf("Le resultat de l'addition est: %u \n\n",resultat->somme);
  } else {
    printf("La fonction distante ne peut faire l'addition a cause d'un overflow \n\n");
  }
}
```

Remote Procedure Call (RPC)

```
Int main (int argc, char *argv[])
{
  char *host;
    if (argc < 2) {
    printf ("usage: %s server_host\n", argv[0]);
    exit (1);
  }
  host = argv[1];
  clnt = clnt_create (host, CALCUL,VERSION_UN, "udp");
  if (clnt == NULL) { clnt_pcreateerror (host);
                              exit (1); }
  test_addition ( UINT_MAX - 15, 10 );
  test_addition ( UINT_MAX, 10 );
  clnt_destroy (clnt);
  exit(EXIT_SUCCESS);
}
```

Nous pouvons alors compiler puis lier avec calcul_clnt.o et calcul_xdr.o pour produire le client. Enfin, on peut tester le résultat final (en ayant démarré le processus serveur avant).

```
> gcc -c calcul_client.c
> gcc -o client calcul_client.o calcul_clnt.o calcul_xdr.o
> ./client localhost
Appel de la fonction CALCUL_ADDITION avec les parametres: 4294967280 et 10
Le resultat de l'addition est: 4294967290

Appel de la fonction CALCUL_ADDITION avec les parametres: 4294967295 et 10
La fonction distante ne peut faire l'addition a cause d'un overflow
>
```

Le client prend en paramètre le nom de la machine serveur (ici localhost car le processus serveur a été démarré sur la même machine). Le nom peut être court (machine) pour une machine du même domaine que le client ou complet (machine.domaine.com).

- **Conclusion**
- Avec l'aide de la bibliothèque de fonctions RPC (clnt_create, clnt_destroy, ...) et des outils comme rpcgen et rpcinfo, il a été très facile de construire une application simple mais toutefois représentative du mécanisme RPC. Pour aller plus loin, vous pouvez consulter la page man de rpcgen ainsi que le RFC 1057.
- Aujourd'hui des systèmes plus perfectionnés ont fait leur apparition comme par exemple CORBA. Certains diront peut-être que les RPC, sont en quelque sorte l'ancêtre de CORBA dans lequel il ne s'agit plus de fonctions distantes mais d'objets (dans le sens de la programmation orienté objet) distants.