Master Informatique M1

Programmation client-serveur *sockets* - RPC

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/people/krakowia

Plan de la suite

- **Problème**
  - 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), dans un langage particulier ; exemple : C

  → Cours 6 (aujourd'hui) TP 4
  - Appel de méthode à distance, dans les langages à objets ; exemple : Java RMI

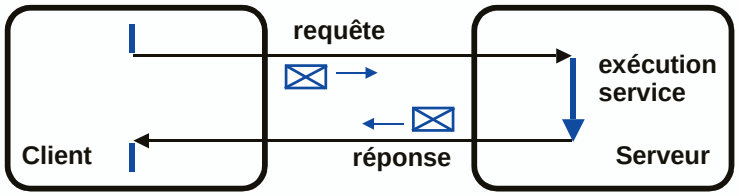
  → Cours 7 TP 5-6
  - *Middleware* intégré : CORBA, EJB, .Net, Web Services

  → En Master 2
- **Exemples**
  - Exemples de services usuels : DNS, Web

  → Cours 8

Mise en œuvre du schéma client-serveur

- **Rappel du schéma client-serveur**
  - Appel synchrone requête-réponse

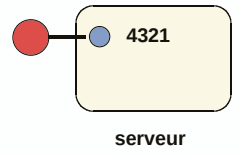


- **Mise en œuvre**
  - Bas niveau : utilisation directe du transport : *sockets* (construit sur TCP ou UDP)
    - Exemple : utilisation des *sockets* en Java
  - Haut niveau : intégration dans un langage de programmation : RPC (construit sur *sockets*)
    - Exemple : RPC en C

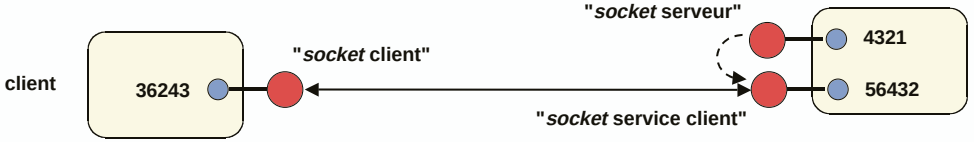
Introduction aux *sockets*

socket **: mécanisme de communication permettant d'utiliser l'interface de transport (TCP-UDP). Introduit dans Unix dans les années 80 ; standard aujourd'hui**

Principe de fonctionnement : 3 phases - illustration ci-dessous avec TCP
1) le serveur crée une "*socket* serveur" (associée à un port) et se met en attente



2) le client se connecte à la *socket* serveur ; deux sockets sont alors crées : une "*socket* client", côté client, et une "*socket* service client" côté serveur. Ces *sockets* sont connectées entre elles



3) Le client et le serveur communiquent par les *sockets*. L'interface est celle des fichiers (*read*, *write*). La *socket* serveur peut accepter de nouvelles connexions

Deux réalisations possibles du client-serveur avec *sockets*

- **Mode connecté (protocole TCP)**
  - Ouverture d'une liaison, suite d'échanges, fermeture de la liaison
  - Le serveur préserve son état entre deux requêtes
  - Garanties de TCP : ordre, contrôle de flux, fiabilité
  - Adapté aux échanges ayant une certaine durée (plusieurs messages)
- **Mode non connecté (protocole UDP)**
  - Les requêtes successives sont indépendantes
  - Pas de préservation de l'état entre les requêtes
  - Le client doit indiquer son adresse à chaque requête (pas de liaison permanente)
  - Pas de garanties particulières (UDP)
  - Adapté aux échanges brefs (réponse en 1 message)
- **Points communs**
  - Le client a l'initiative de la communication ; le serveur doit être à l'écoute
  - Le client doit connaître la référence du serveur [adresse IP, n° de port] (il peut la trouver dans un annuaire si le serveur l'y a enregistrée au préalable, ou la connaître par convention : n°s de port préaffectés)
  - Le serveur peut servir plusieurs clients (1 *thread* unique ou 1 *thread* par client)

Utilisation du mode connecté

- **Caractéristiques**
  - établissement préalable d'une connexion (circuit virtuel) : le client demande au serveur s'il accepte la connexion
  - fiabilité assurée par le protocole (TCP)
  - mode d'échange par flot d'octets : le récepteur n'a pas connaissance du découpage des données effectué par l'émetteur
  - possibilité d'émettre et de recevoir des caractères urgents (OOB : *Out OF Band*)
  - après initialisation, le serveur est "passif" : il est activé lors de l'arrivée d'une demande de connexion du client
  - un serveur peut répondre aux demandes de service de plusieurs clients : les requêtes arrivées et non traitées sont stockées dans une file d'attente
- **Contraintes**
  - le client doit avoir accès à l'adresse du serveur (adresse IP, n° de port)
- **Modes de gestion des requêtes**
  - itératif : le processus traite les requêtes les unes après les autres
  - concurrent : par création de processus fils pour les échanges de chaque requête (ouverture de connexion)

Serveur itératif en mode connecté

- **Le client ouvre une connexion avec le serveur avant de pouvoir lui adresser des appels, puis ferme la connexion à la fin de la suite d'opérations**
  - délimitation temporelle des échanges
  - maintien de l'état de connexion pour la gestion des paramètres de qualité de service
    - traitement des pannes, propriété d'ordre
- **Orienté vers**
  - le traitement ordonné d'une suite d'appels
    - ordre local (requêtes d'un client traitées dans leur ordre d'émission), global ou causal
  - la gestion de données persistantes ou de protocoles avec état

Programmation avec *sockets*

- **Principes**
  - Les différentes opérations de gestion de *sockets* (*socket*, *bind*, etc.) sont fournies comme primitives (appel systèmes) dans Unix (et d'autres systèmes d'exploitation)
  - Si intéressés, lirer les pages *man* de ces opérations
  - On peut utiliser directement ces opérations, mais les langages usuels fournissent des outils facilitant leur usage et couvrant les cas courants, côté client et côté serveur
    - bibliothèques spécialisées en C
    - classes en Java
- **La suite…**
  - Nous allons regarder de plus près (cours et TP) la programmation des sockets en Java
  - Si intéressés par la programmation de *sockets* en C, voir références ci-dessous (contiennent des exemples)

Livres
J. M. Rifflet, J.-B. Yunès. *Unix - Programmation et Communicatio*n, Dunod (2003), chap. 19
R. Stevens. *Unix Network Programming*. Prentice-Hall.

Web
http://www.eng.auburn.edu/department/cse/classes/cse605/examples/index.html
http://www.scit.wlv.ac.uk/~jphb/comms/sockets.html

Programmation des *sockets* en Java (mode connecté)

- **Deux classes de base**
  - **ServerSocket : *socket* côté serveur (attend connexions et crée *socket* service client)**
  - **Socket : *sockets* ordinaires, pour les échanges. Fournissent des classes InputStream et OutputStream (flots d'octets) pour échanger de l'information. Exemple :**

```
PrintWriter out = new PrintWriter(mySocket.getOutputStream(),
true);
BufferedReader in = new BufferedReader(new InputStreamReader(
            mySocket.getInputStream()));
```

Utilisables ainsi :

```
String s1, s2;
out.println(s1);
s2 = in.readLine();
```

Analogie avec fichiers (les *sockets* s'utilisent comme des fichiers) :

```
BufferedReader in  = new BufferedReader(new FileReader("myFile"));
PrintWriter out = new PrintWriter(new FileWriter("myFile"), true);
```

Voir http://java.sun.com/docs/books/tutorial/networking/sockets/

Client-serveur avec *sockets* TCP en Java (très simplifié)

Sur une machine client — **doit être connu du client** — **Sur la machine serveur (ex : "goedel.imag.fr")**

```
mySocket = new Socket("goedel.imag.fr", 7654);
```

```
serverSocket = new ServerSocket(7654);
clientServiceSocket = serverSocket.accept();
```

Maintenant le client et le serveur sont connectés

```
PrintWriter out = new PrintWriter(
        mySocket.getOutputStream(), true);
BufferedReader in = new BufferedReader(
        new InputStreamReader(
            mySocket.getInputStream()));
```

```
PrintWriter out = new PrintWriter(
     clientServicetSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader(
     new InputStreamReader(
          clientServiceSocket.getInputStream()));
```

Maintenant le client et le serveur peuvent communiquer via les canaux

```
BufferedReader stdIn = new BufferedReader(
      new InputStreamReader(System.in));
String request; String reply;

while (true) {
   request = stdIn.readLine(); // l'utilisateur entre la requête
   out.println(request);      // envoyer la requête au serveur
   reply = in.readLine()    // attendre la réponse
  System.out.println(reply); // imprimer la réponse
  }
```

```
String request;
String reply;

while (true) {
   request = in.readLine()
   // executer le service
   // traiter  request pour fournir reply
   out.println(reply);
   }
```


Compléments divers

Importer les packages utiles

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

Prévoir les cas d'erreur

```
Socket mySocket = null;
try {
   mySocket = new Socket("goedel.imag.fr", 7654);
} catch (IOException e) {
   System.out.println("connexion impossible");
   System.exit(-1);
}
```

Prévoir terminaison

(dépend de l'application)

Terminer proprement

```
out.close();
in.close();
mySocket.close();
```

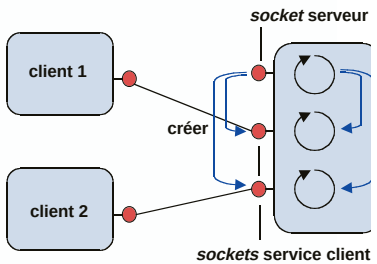
```
import java.io.*;
import java.net.*;

try {
   serverSocket = new ServerSocket(7654);
} catch (IOException e) {
   System.out.println("port 7654 non utilisable");
   System.exit(-1);
}
Socket clientServiceSocket = null;
try {
serverSocket.accept();
} catch (IOException e) {
   System.out.println(accept impossible sur port 7654");
   System.exit(-1);
}
```

```
out.close();
in.close();
clientServiceSocket.close();
serverSocket.close();
```


Gestion du parallélisme sur le serveur

- **Avec le schéma précédent, un seul processus serveur**
  - **S'il y a des clients multiples, ils sont traités en séquence (les requêtes sont conservées dans une file d'attente associée à la *socket* serveur)**
    - **N.B. La longueur max de cette file d'attente peut être spécifiée lors de la création de la *socket* serveur. Valeur par défaut : 50**
- **On peut aussi utiliser un schéma veilleur-exécutants**
  - **Le *thread* veilleur doit créer explicitement un nouveau *thread* exécutant à chaque nouvelle connexion d'un client (donc à l'exécution de *accept*())**
  - **Une *socket* service client différente est créée pour chaque client**
  - **Le *thread* veilleur se remet ensuite en attente**

```
while (true) {
   accepter la connexion d'un nouveau client
            et créer une socket service client;
   créer un thread pour interagir avec ce client sur la
            nouvelle socket service client;
}
```



Un serveur multi-*thread*

programme des exécutants →

programme du veilleur ↓

```
serverSocket = new ServerSocket(7654);

while (true) {
    Socket clientServiceSocket =
            serverSocket.accept() ;

    Service  myService =
            new Service(clientServiceSocket) ;
               // crée un nouveau thread
                // pour le nouveau client
            myService.start () ;
              // lance l'exécution du thread
    }
```

Le programme du client est inchangé

```
public class Service extends Thread {
      protected final Socket socket ;
      String request, reply;

      public myService(Socket socket) {
           this.socket - socket ;
      {

      public void run () {
          PrintWriter out = new PrintWriter(
             clientServicetSocket.getOutputStream(), true);
          BufferedReader in = new BufferedReader(
              new InputStreamReader(
                  clientServiceSocket.getInputStream()));
          try {
               request = in.readLine() ;
               // exécuter le service
               // traiter  request pour fournir reply
               out.println(reply);
          }  finally  {
               socket.close () ;
          }
       }
}
```


Utilisation du mode non connecté (datagrammes, UDP)

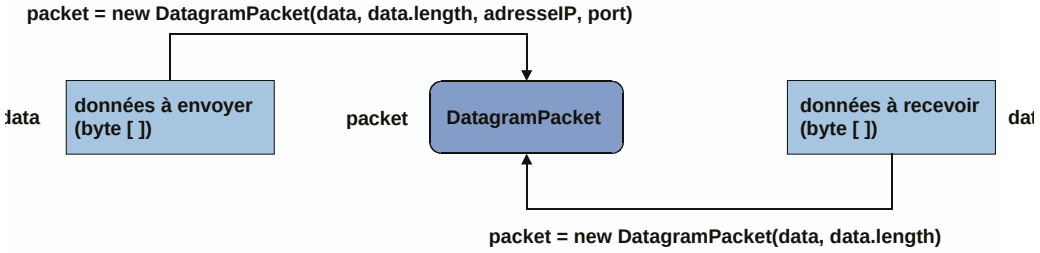
- **Caractéristiques**
  - pas d'établissement préalable d'une connexion
  - Pas de garantie de fiabilité
  - adapté aux applications pour lesquelles les réponses aux requêtes des clients sont courtes (1 message)
  - le récepteur reçoit les données selon le découpage effectué par l'émetteur
- **Contraintes**
  - le client doit avoir accès à l'adresse du serveur (adr. IP, port)
  - le serveur doit récupérer l'adresse de chaque client pour lui répondre (primitives *sendto*, *recvfrom*)
- **Mode de gestion des requêtes**
  - itératif (requêtes traitées l'une après l'autre)
  - concurrent (1 processus ou *thread* par client)

Serveur itératif en mode non connecté

- **Le client peut envoyer un appel au serveur à tout moment**
  - mode léger permettant
    - le traitement non ordonné des appels
    - l'absence de mémorisation entre appels successits (serveur sans données rémanentes et sans état)
  - exemples
    - calcul de fonction numérique
    - DNS (service de noms)
    - NFS (service de fichiers répartis)

Programmation des *sockets* en Java (mode non connecté)

- **Deux classes : *DatagramSocket* et *DatagramPacket***
  - ***DatagramSocket*** **: un seul type de socket**
  - ***DatagramPacket*** **: format de message pour UDP**
    - Conversion entre données et paquet (dans les 2 sens)



- **Utilisation**
  - Échanges simples (question-réponse)
  - Messages brefs
  - ***"Streaming"*** **temps réel (performances)**

Client-serveur en mode non connecté

Sur une machine client

```
DatagramSocket socket = new DatagramSocket();

byte [ ] sendBuffer = new byte [1024] ;
byte [ ] receiveBuffer = new byte [1024] ;
String request, reply;

request = …  // dépend de l'application
sendBuffer = request.getBytes() ;
DatagramPacket outpacket =
   new DatagramPacket (sendBuffer,
                sendBuffer.length,

                InetAddress.getbyname("goedel.imag.fr"),
                7654) ;
socket.send(outpacket) ;

DatagramPacket inpacket =
   new DatagramPacket (receiveBuffer,
                              receiveBuffer.length);
socket.receive(inpacket );
reply = new String(inpacket.getData());
```

doit être connu du client

Envoi requête

Réception réponse

Sur la machine serveur (ex : "goedel.imag.fr")

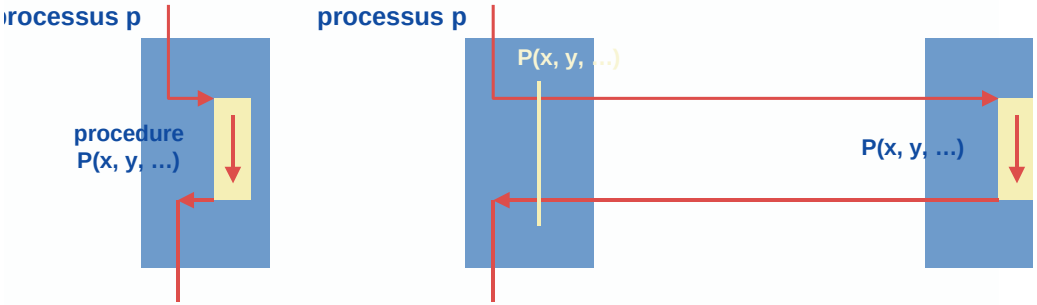
```
DatagramSocket socket =
        new DatagramSocket(7654);

byte [ ] receiveBuffer = new byte [1024] ;
byte [ ] sendBuffer = new byte [1024] ;
String request, reply;

while (true) {
   DatagramPacket inpacket =
      new DatagramPacket (receiveBuffer,
                                 receiveBuffer.length);
   socket.receive(inpacket ) ;
   request =
        new String(receiveBuffer.getData);
   // déterminer adresse et port du client
   InetAddress clentAddress = inpacket.getAddress(
   int clienPort = inpacket.
   // executer service : traiter request pour fournir re
   sendBuffer = reply.getBytes() ;
   DatagramPacket outpacket =
      new DatagramPacket (sendBuffer,
        sendBuffer.length,  clientAddress, clienPort);
   socket.send(outpacket);
   }
```

Réception requête

Envoi réponse

Appel de procédure à distance : définition

- **Appel de procédure à distance (*Remote Procedure Call*, ou RPC) : un outil pour construire des applications client-serveur dans un langage de haut niveau**
- **L'appel et le retour ont lieu sur un site ; l'exécution se déroule sur un site distinct**



L'effet de l'appel doit être identique dans les deux situations. Mais cet objectif ne peut être atteint en toute rigueur en présence de défaillances (cf plus loin)

Appel de procédure à distance : avantages et problèmes

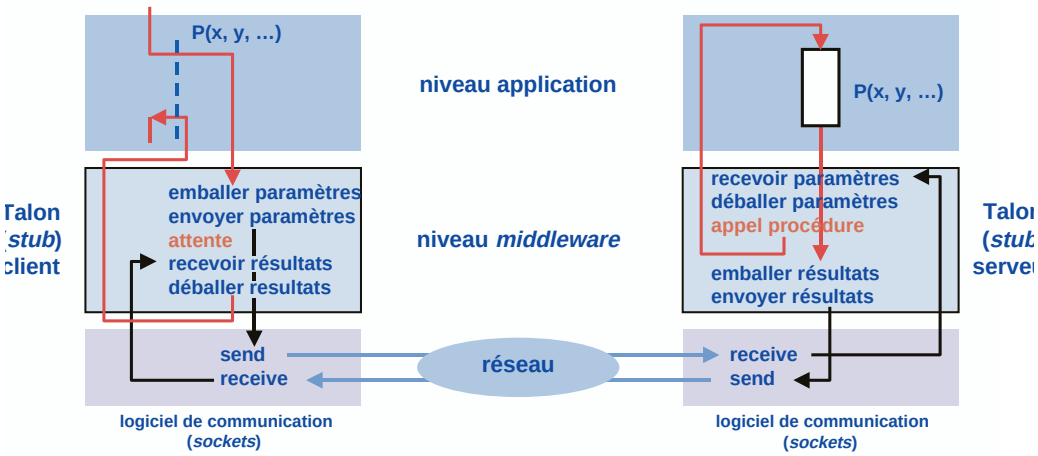
Avantages attendus

- **Facilité de programmation**
  - **La complexité des protocoles de communication est cachée**
- **Facilité de mise au point : une application peut être mise au point sur un site unique, puis déployée sur plusieurs sites**
- **Portabilité : résulte de l'usage d'un langage de haut niveau**
  - **Indépendance par rapport au système de communication**

Problèmes de réalisation

- **Transmission des paramètres (conversion entre la forme interne, propre à un langage, et une forme adaptée à la transmission)**
- **Gestion des processus**
- **Réaction aux défaillances**
  - **Trois modes de défaillance indépendants : client, serveur, réseau**

Appel de procédure à distance : principes de mise en œuvre



- **Les talons client et serveur sont créés à partir d'une description d'interface**

Utilisation de l'appel de procédure à distance (1)

Exemple (en langage C) : un service d'annuaire

Les étapes :

1) Préparer une description de l'interface (cf plus loin) : fichier annuaire.x

décrit les types de données et les opérations (fonctions de l'annuaire : init, ajouter, supprimer, consulter)

2) Compiler ce fichier à l'aide du générateur de talons :

`rpcgen annuaire.x` résultat : les fichiers suivants :

annuaire.h	include
annuaire_clnt.c	talon client
annuaire_svc.c	talon serveur
annuaire_xdr.c	proc. conversion données

fournit aussi des fichiers auxiliaires (modèles de prog. client, serveur, Makefile)

3) Sur le site client, construire l'exécutable client (programme : annuaire_client.c)

```
gcc -o client annuaire_client.c annuaire_clnt.c
```

4) Sur le site serveur, construire l'exécutable serveur (programme : annuaire_serveur.c)

```
gcc -o server annuaire_server.c annuaire_svc.c
```

5) Sur le site serveur, lancer l'exécutable serveur

```
server &
```

6) Sur le site client, lancer l'exécutable client

```
client
```


Utilisation de l'appel de procédure à distance (2)

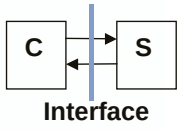
Description d'interface (fichier "annuaire.x")

```
/* constantes et types */

const max_nom = 20 ;
const max_adr = 50 ;
const max_numero = 16 ;

typedef string typenom<max_nom> ;
typedef string typeadr<max_adr> ;
typedef string
        typenumero<max_numero> ;

struct personne {
        typenumero numero ;
        typenom nom ;
        typeadr adresse ;
        } ;

typedef struct personne personne ;
```

```
/* description de l 'interface */

program ANNUAIRE {
   version V1 {
        void INIT(void) = 1;
        int AJOUTER(personne p) = 2 ;
        int SUPPRIMER (personne p) = 3 ;
        personne CONSULTER (typenom nom) = 4 ;
   } = 1 ;
} = 0x23456789 ;
```


Description d'interface

- **Interface = "contrat" entre client et serveur**
  - **Définition commune abstraite**
    - Indépendante d'un langage particulier (adaptée à des langages multiples)
    - Indépendante de la représentation des types
    - Indépendante de la machine (hétérogénéité)
  - **Contenu minimal**
    - Identification des procédures (nom, version)
    - Définition des types des paramètres, résultats, exceptions
    - Définition du mode de passage (IN, OUT, IN-OUT)
  - **Extensions possibles**
    - Procédures de conversion pour types complexes
    - Propriétés non-fonctionnelles (qualité de service), non standard)



RPC : structure du programme serveur

N.B. : ce programme ne contient que les fonctions à appeler. La procédure main est contenue dans le talon. Ce programme est à écrire par le développeur, alors que le talon est engendré automatiquement

```
#include "annuaire.h"

void *
init_1_svc(void *argp, struct svc_req *rqstp)

        static char * result;
        /* le programme de la fonction init */
        return (void *) &result;


int *
ajouter_1_svc(personne *argp, struct svc_req *rqstp)

        static int  result;
        /* le programme de la fonction ajouter */
        return &result;
```

```
int *
supprimer_1_svc(personne *argp, struct svc_req *rqstp)
{
        static int  result;
        /* le programme de la fonction supprimer */
        return &result;
}

personne *
consulter_1_svc(typenom *argp, struct svc_req *rqstp)
{
        static personne  result;
        /* le programme de la fonction consulter */
        return &result;
}
```

RPC : structure du talon serveur

```
es parties en gras ne sont pas détaillées
 includes ... */
atic void
nuaire_1(struct svc_req *rqstp, register SVCXPRT *transp){
   union {
      personne ajouter_1_arg;
      personne supprimer_1_arg;
      typenom consulter_1_arg;
   } argument;
   char *result;
   xdrproc_t _xdr_argument, _xdr_result;
   char *(*local)(char *, struct svc_req *);

    switch (rqstp->rq_proc) {
            case NULLPROC:
              (void) svc_sendreply (transp,
                      (xdrproc_t) xdr_void, (char *)NULL);
              return;
            case INIT:
              _xdr_argument = (xdrproc_t) xdr_void;
              _xdr_result = (xdrproc_t) xdr_void;
              local = (char *(*)(char *,
                      struct svc_req *)) init_1_svc;
              break;
            case AJOUTER:
              _xdr_argument = (xdrproc_t) xdr_personne;
              _xdr_result = (xdrproc_t) xdr_int;
              local = (char *(*)(char *,
                            struct svc_req *)) ajouter_1_svc;
              break;
            case SUPPRIMER:
                      /* idem */
            case CONSULTER:
                      /* idem */
            default:      svcerr_noproc (transp);
                      return;
    }

    /*  préparer arguments dans zone argument */
    result = (*local)((char *)&argument, rqstp);        /* exécution */
       if (result != NULL && !svc_sendreply(transp, /* envoi résultats */
                (xdrproc_t) _xdr_result, result)) {
           svcerr_systemerr (transp);
       }
    /*  libérer arguments */

    return;
    }
    Int main (int argc, char **argv)
    {
      /* créer une socket serveur TCP sur un port p  */
      /* créer une socket serveur UDP sur un port p1 */
      /* enregistrer le service annuaire_1 sous TCP (port p)  */
      /* enregistrer le service annuaire_1 sous UDP (port p1) */
      svc_run () ; /* se mettre en attente d'un appel du client */
    }
```


RPC : structure du programme client

```
annuaire_client.c
/* include */
/* déclarations */

main(argc, argv)
int argc ; char * argv ;
{
        CLIENT *clnt ;
        char * host
…
        if (argc <2) {
                printf("usage: %s server_host\n", argv[0]) ;
                exit(1)
        {
        host = argv[1] ;

        clnt = clnt_create (host, ANNUAIRE, V1, "netpath") ;     /* "poignée" d 'accès au serveur */
        if (clnt == (CLIENT *) NULL {
                {clnt_pcreateerror(host) ;
                exit(1) ;
        {
…                                                                /* saisir paramètres */
        result_2 = ajouter_1(&ajouter_1_arg, clnt)
        if (result_2 == (int *) NULL) {
                clnt_perror(clnt, "call failed") ;
        {
…                                                                /* afficher résultats */
}
```

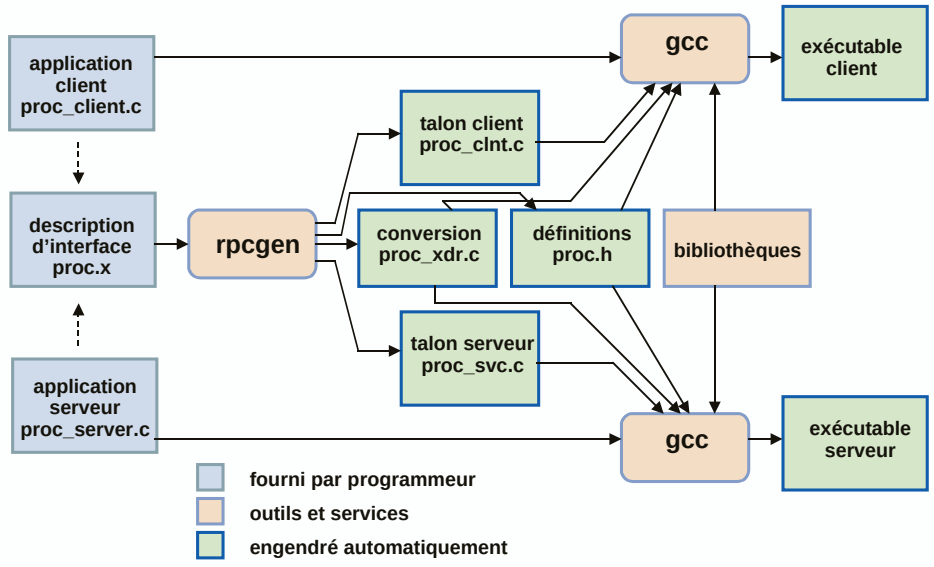

Structure du talon client

```
#include <memory.h> /* for memset */
#include "annuaire.h"
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = {25, 0 };

void * init_1(void *argp, CLIENT *clnt) {
        static char clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, INIT,
                (xdrproc_t) xdr_void, (caddr_t) argp,
                (xdrproc_t) xdr_void, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return ((void *)&clnt_res);
}

int * ajouter_1(personne *argp, CLIENT *clnt) {
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, AJOUTER,
                (xdrproc_t) xdr_personne, (caddr_t) argp,
                (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```

L'appel via la socket est caché dans clnt_call

```
int * supprimer_1(personne *argp, CLIENT *clnt) {
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, SUPPRIMER,
                (xdrproc_t) xdr_personne,
                        (caddr_t) argp,
                (xdrproc_t) xdr_int,
                        (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}

personne * consulter_1(typenom *argp, CLIENT *clnt) {
        static personne clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, CONSULTER,
                (xdrproc_t) xdr_typenom,
                        (caddr_t) argp,
                (xdrproc_t) xdr_personne,
                        (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```


Chaîne de production pour l'appel de procédure à distance



Appel de procédure à distance : détails de réalisation

La réalisation de l'appel de procédure à distance soulève divers problèmes techniques que nous n'examinons pas en détail

- **Passage de paramètres**
  - Pas de passage par référence (car les pointeurs perdent leur signification)
  - Conversion des données (dépend du type et des conventions de représentation - il existe des standards)
- **Désigantion**
  - Comment le client détermine l'adresse du serveur (et vice-versa)
- **Traitement des défaillances**
  - cf résumé plus loin
- **Serveurs multi-*threads***
- **Outils**
  - Génération de talons
  - Mise au point

Références pour aller plus loin :
Exemples en ligne http://www.cs.cf.ac.uk/Dave/C/node33.html
J.-M. Rifflet et J.-B. Yunès, *UNIX, Programmation et communication*, Dunod (2003), chap. 21
R. Stevens, *Unix Network Programming*, vol. 2, 2nd ed., Prentice Hall (1999)

Traitement des défaillances (résumé)

- **Difficultés du traitement des défaillances**
  - Le client n'a pas reçu de réponse au bout d'un délai de garde fixé. 3 possibilités :
    - a) Le message d'appel s'est perdu sur le réseau
    - b) L'appel a été exécuté, mais le message de réponse s'est perdu
    - c) Le serveur a eu une défaillance
      - c1: Avant d'avoir exécuté la procédure
      - c2 : Après avoir exécuté la procédure mais avant d'avoir envoyé la réponse
  - Si le client envoie de nouveau la requête
    - Pas de problème dans le cas a)
    - L'appel sera exécuté 2 fois dans le cas b)
      - Remède : associer un numéro unique à chaque requête
    - Divers résultats possibles dans le cas c) selon remise en route du serveur
- **Conséquences pratiques sur la conception des applications**
  - Construire des serveurs "sans état" (donc toute requête doit être self-contenue, sans référence à un "état" mémorisé par le serveur)
  - Prévoir des appels idempotents (2 appels successifs ont le même effet qu'un appel unique). Exemple : déplacer un objet
    - move(deplacement_relatif) : non idempotent
    - move_to(position absolue) : idempotent

Conclusions sur l'appel de procédure à distance

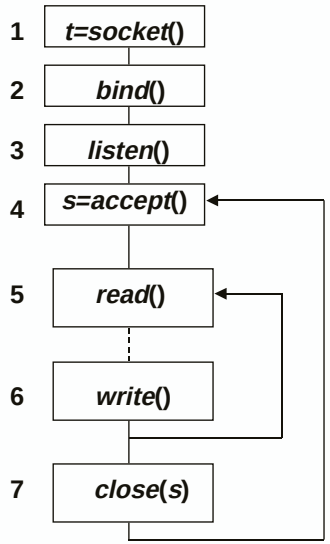
- **Avantages**
  - Abstraction (les détails de la communication sont cachés)
  - Intégration dans un langage : facilite portabilité, mise au point
  - Outils de génération, facilitent la mise en œuvre
- **Limitations**
  - La structure de l'application est statique : pas de création dynamique de serveur, pas de possibilité de redéploiement entre sites
  - Pas de passage des paramètres par référence
  - La communication est réduite à un schéma synchrone
  - La persistance des données n'est pas assurée (il faut la réaliser explicitement par sauvegarde des données dans des fichiers)
  - Des mécanismes plus évolués visent à remédier à ces limitations
    - Objets répartis (ex : Java RMI, à voir en cours)
    - Composants répartis (ex : EJB, Corba CCM, .Net)

Matériau complémentaire

(non présenté en cours)

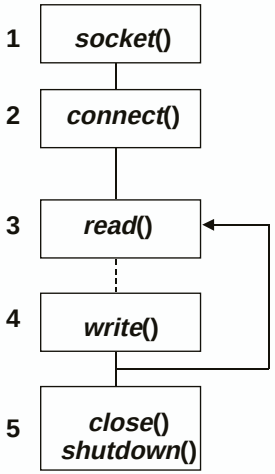
Algorithme d'un serveur en mode connecté

- 1. Création de la *socket* serveur
- 2. Récupération de l'adresse IP et du numéro de port du serveur ; lien de la *socket* à l'adresse du serveur
- 3. Mise en mode passif de la *socket* : elle est prête à accepter les requêtes des clients
- 4. (opération bloquante) : acceptation d'une connexion d'un client et création d'une *socket* service client, dont l'identité est rendue en retour
- 5. et 6. Lecture, traitement et écriture (selon algorithme du service)
- 7. Fermeture et remise en attente

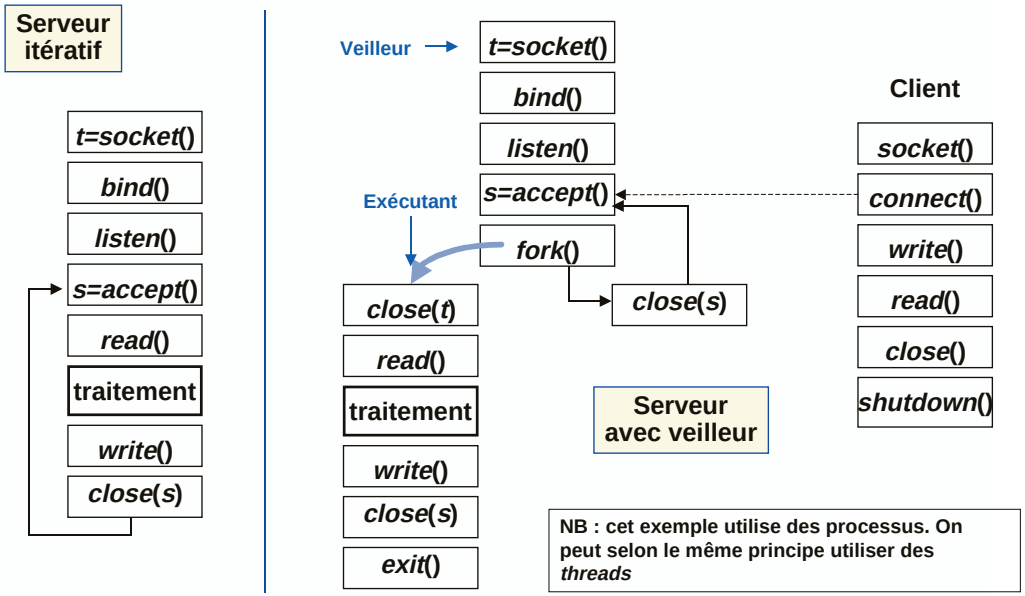


Algorithme d'un client en mode connecté

- 1. Création de la *socket*
- 2. Connexion de la *socket* au serveur
  - choix d'un port libre pour la *socket* par la couche TCP
  - attachement automatique de la *socket* à l'adresse (IP machine locale + n° de port)
  - connexion de la *socket* au serveur en passant en paramètre l'adresse IP et le n° de port du serveur
- 3. et 4. Dialogue avec le serveur (selon algorithme du service)
- 5. Fermeture de la connexion avec le serveur



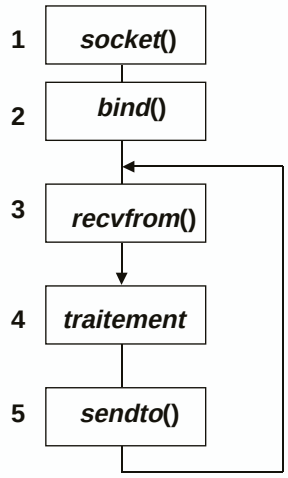
Gestion des processus (mode connecté)



NB : cet exemple utilise des processus. On peut selon le même principe utiliser des *threads*

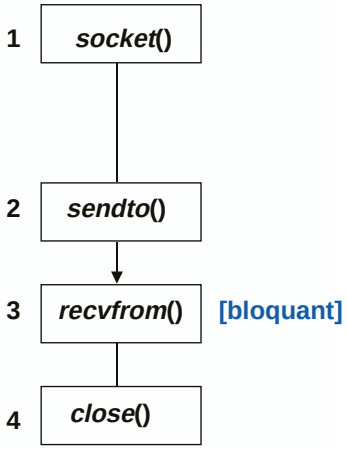
Algorithme d'un serveur en mode non connecté

- 1. Création de la *socket*
- 2. Récupération de l'adresse IP et du numéro de port du serveur ; lien de la *socket* à l'adresse du serveur
- 3. Réception d'une requête de client
- 4. Traitement de la requête ; préparation de la réponse
- 5. Réponse à la requête en utilisant la *socket* et l'adresse du client obtenues par *recvfrom* ; retour pour attente d'une nouvelle requête



Algorithme d'un client en mode non connecté

- 1. Création de la *socket* (l'association à une adresse locale [adresse IP + n° port] est faite automatiquement lors de l'envoi de la requête)
- 2. Envoi d'une requête au serveur en spécifiant son adresse dans l'appel
- 3. Réception de la réponse à la requête
- 4. Fermeture de la *socket*



Gestion des processus (mode non connecté)