Intergiciels et applications communicantes

Philippe Quéinnec
http://queinnec.perso.enseeiht.fr/Ens/intergiciels.html

ENSEEIHT
Informatique et Réseaux 2e année

27 janvier 2020

Inspiré de cours de G. Padiou, Ph. Mauran et S. Krakowiak. Certains dessins en sont issus.

Première partie

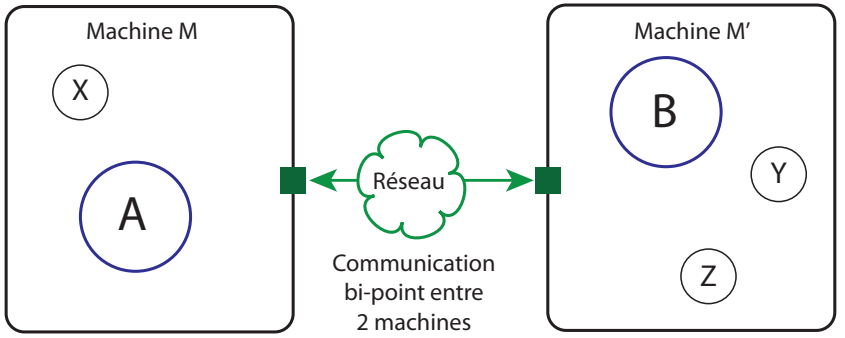
Introduction

Plan

La communication à distance

Communication au niveau réseau

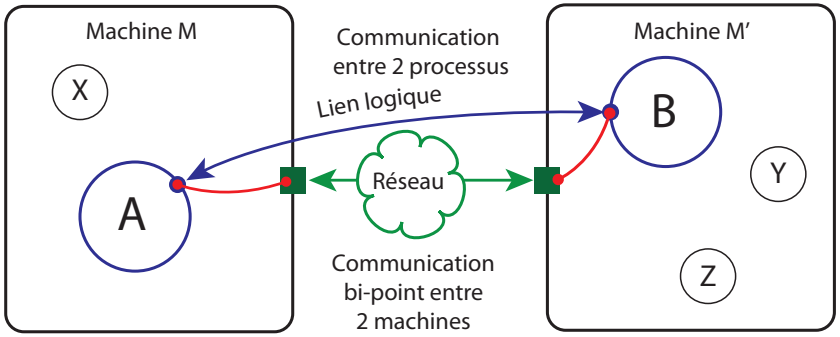
Les ingénieurs réseaux font communiquer des machines



La communication à distance

Communication au niveau intergiciel

Les ingénieurs informaticiens font communiquer des applications



La communication à distance : objectifs

Faire communiquer des processus par échange d'information

Pas de mémoire partagée, communication « à distance »

- Exploiter les réseaux de communication pour faire communiquer des ordinateurs
- Avantage : le partage de ressources et l'échange d'informations
- Comment ? définition de protocoles de communication entre processus distants
- Difficulté : hétérogénéité du matériel, des systèmes d'exploitation, des programmes applicatifs écrits dans différents langages : C, C++, Java...

Plan du cours

Distinction service / serveur

Service

Un service est une description / interface / spécification.

Serveur

Un serveur est une réalisation / implantation / concrétisation d'un service

Besoins

- Trouver les services existants
- Pour un service donné, trouver les serveurs qui le réalisent
- Dialoguer avec un serveur donné (= protocole)

Service/serveur – exemple

Je veux manger

- Quels sont les services offerts dans un village ⇒ boulangerie, bistrot...
- Où trouver une boulangerie ? ⇒ place Dupuy
- Comment acheter une baguette ⇒ protocole d'interaction

Service de nommage internet (DNS)

- Service de conversion de noms symboliques en adresses IP numériques
- À l'N7, serveurs sur les machines 147.127.80.123, 147.127.176.22, 147.127.16.11

Plan

Les protocoles de communication

Grande variété de mise en œuvre

Beaucoup de paramètres

- Point à point ou diffusion
- Synchronisation émetteur-récepteur :
  - Envoi asynchrone : l'émetteur ne se bloque pas
  - Envoi synchrone : l'émetteur attend un acquittement
  - Gestion de tampons en émission et/ou en réception
- Délais de transmission (non) bornés
- Fiabilité : pertes possibles, duplications, erreurs...
- Désignation des entités distantes (processus)

Les protocoles de communication (suite)

Mais aussi selon le niveau d'abstraction

- Simple échange d'un message
- Transaction de messages, par exemple :
  question, réponse [, acquittement]
- Flots de messages sur une liaison pré-établie
- Appel procédural à distance, appel de méthode à distance
- Publication et abonnement
- etc

Plan

Les intergiciels et leur rôle

Définition

- Couche logicielle entre système d'exploitation et applications
- Service implantant un modèle d'interaction entre processus
- API normalisée associée au service de communication implanté

Quelques points communs

- Besoin d'une interface avec la couche « transport »
- Besoin de certains services de base : service de nommage
- Nécessité de traiter l'hétérogénéïté entre les processus

Les sources d'hétérogénéïté

Deux processus communicants peuvent

- exécuter un programme écrit dans des langages différents
- être exécutés par des systèmes d'exploitation différents
- s'exécuter sur des architectures de machine différentes

Deux approches de solution

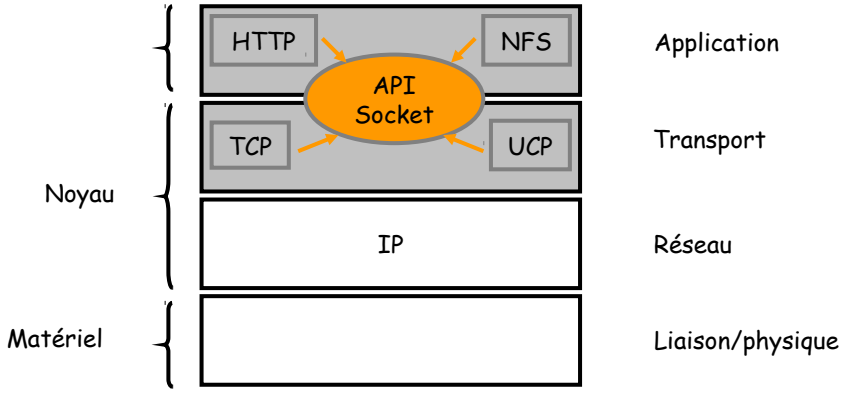
- Définition d'un langage commun (IDL/Interface Description Language) et instanciation dans différents contextes d'exécution → normes RPC, CORBA
- Définition d'un environnement de développement et d'exécution portable et exécution dans n'importe quel contexte → environnement Java

Deuxième partie

Communication par flots
Interface socket

Plan

L'API socket dans la pile IP



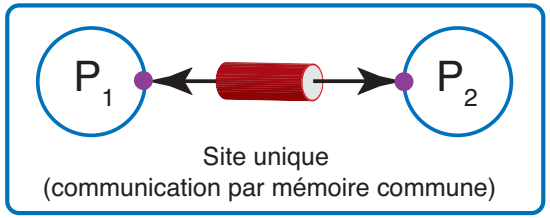
Objectifs de l'API socket

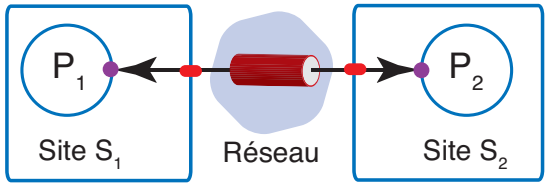
- Fournir une interface homogène aux applications
  - service TCP
  - service UDP
- Conforme au système de gestion de fichiers
  - flot d'octets
  - lecture/écriture
- Modèle client/serveur

La notion de flot centralisé → réparti

Idée

- Pipe « réparti »
- Désignation ?
- Protocole ?
- Interface ?
- Fiabilité ?
- Hétérogénéité ?



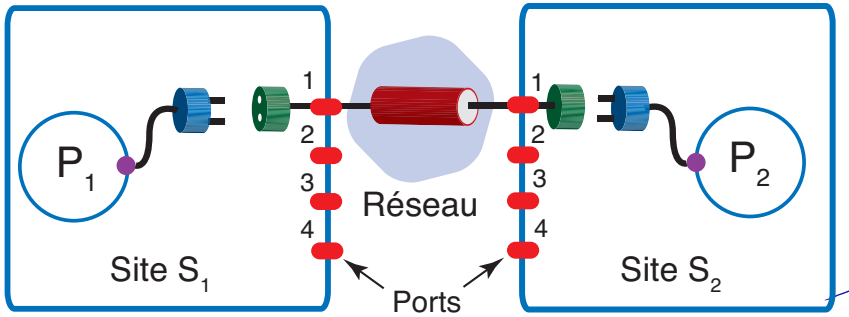


Désignation globale via la notion de port

Comment désigner un processus à distance ?

- Un processus a un nom local : numéro d'ordre par exemple ;
- Un site a un nom global : adresse IP par exemple ;
- Un port est un « point d'accès » à un site ;

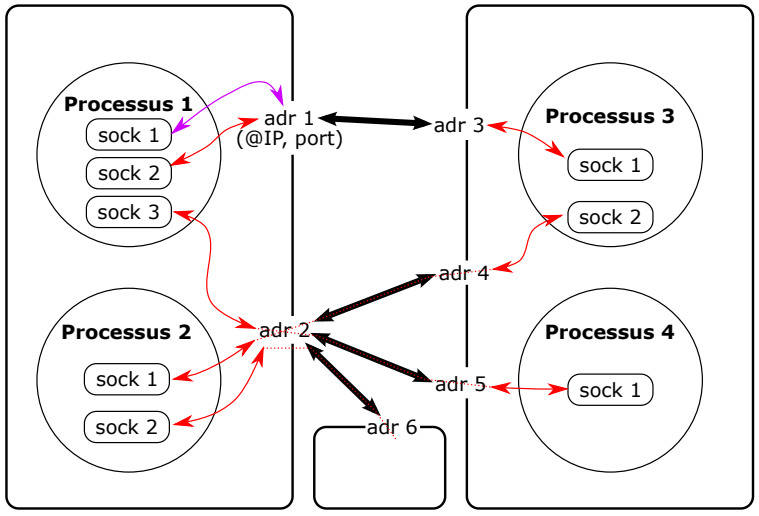
Nom global d'un processus : (adresse site, numéro de port)



Éléments de base

- Socket → descripteur de fichier
- Protocole / domaine d'adresses → famille d'adresses
- Adresse → (adresse IP, numéro de port)
- Liaison → attribution d'une adresse à un socket
- Association → domaine + couple d'adresses (client/serveur ou émetteur/récepteur)

Schéma général



Numéro de ports standards (notorious)

`/etc/services`

```
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
http            80/tcp          www
#
# UNIX          specific services
#
exec            512/tcp
login           513/tcp
printer         515/tcp         spooler
who             513/udp         whod
talk            517/udp
```

Services offerts

Orienté connexion (TCP)

- établissement/acceptation de connexion
- flot d'octets fiable et ordonné
- terminaison de connexion

Orienté datagramme (UDP)

- pas de connexion
- un message applicatif = une opération
- à la fois flot et messages

Plan

Structure générale

Client

- initie la communication
- doit connaître le serveur

Serveur

- informe le système de sa disponibilité
- répond aux différents clients
- clients pas connus a priori

Client/serveur non connecté

Client

```
créer un socket
répéter
    émettre une requête
          vers une adresse
    attendre la réponse
jusqu'à réponse positive
        ou abandon
```

Serveur

```
créer un socket
attribuer une adresse
répéter
    attendre une requête
    traiter la requête
    émettre la réponse
jusqu'à fin du service
```

Client/serveur connecté

Client

```
créer un socket
se connecter au serveur
dialoguer avec le serveur
    par le socket connecté
terminer la connexion
```

Serveur

```
créer un socket
attribuer une adresse
informer le système
répéter
    attendre une demande
          de connexion
    dialoguer avec le client
      par le socket ainsi créé
jusqu'à fin du service
```

- un socket d'écoute pour accepter les connexions,
- un socket connecté pour chaque connexion

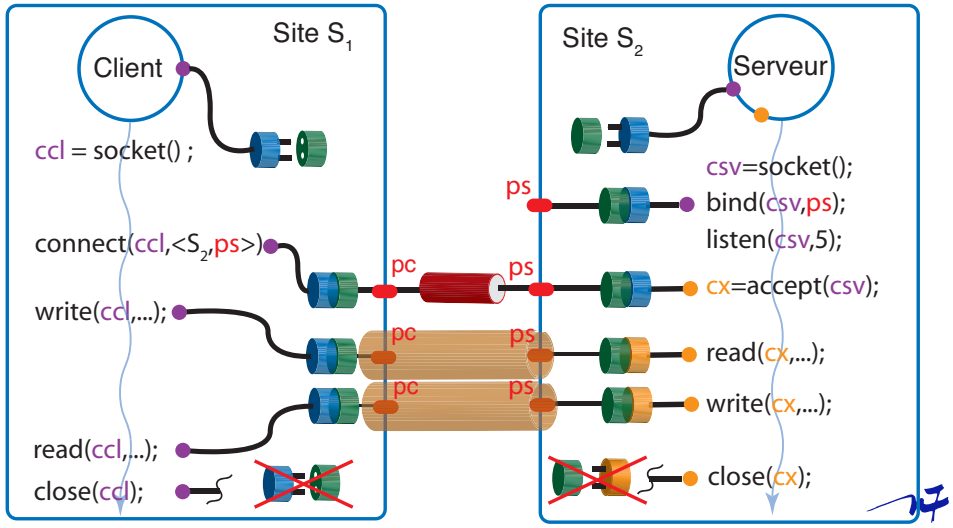
Plan

Primitives principales

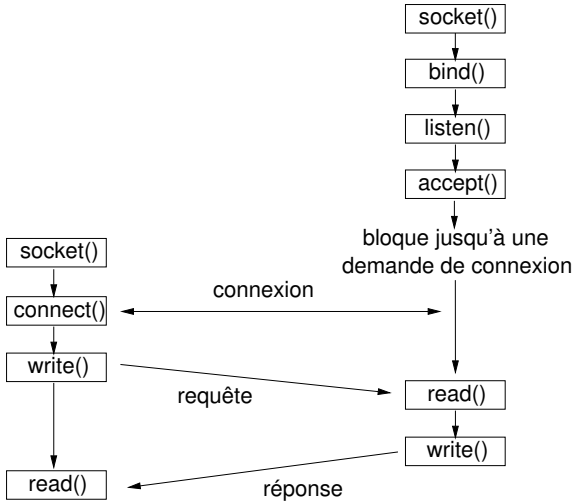
`socket`	création
`bind`	nommage : liaison d'un socket à une adresse
`connect`	connexion : établissement d'une association
`listen`	prêt à attendre des connexions
`accept`	attente de connexion : acceptation d'association
`close`	fermeture
`shutdown`	fermeture (obsolète)

`read/write`
`recv/send`
`recvfrom/sendto`

Exemple – mode connecté



Exemple – mode connecté



Exemple – mode connecté

Le client

```
struct sockaddr_in adrserv;
char reponse[6];

int scl = socket (AF_INET, SOCK_STREAM, 0);
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
inet_aton("147.127.133.111", &adrserv.sin_addr);
adrserv.sin_port = htons (4522);
connect (scl, (struct sockaddr *)&adrserv, sizeof(adrserv));
write (scl, "hello", 6);
read (scl, reponse, 6);
close (scl);
```

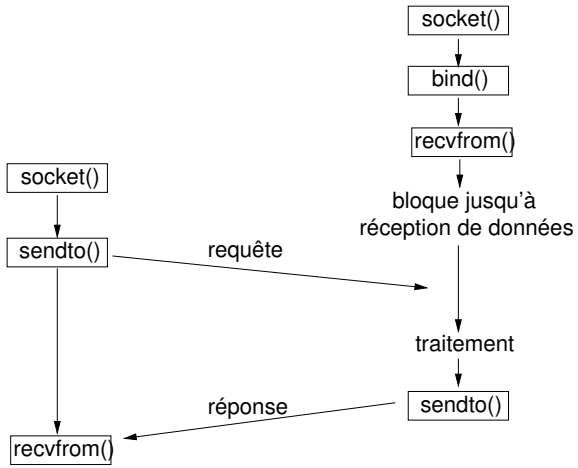
ATTENTION : il manque le contrôle d'erreur, INDISPENSABLE.

Exemple – mode connecté

Le serveur

```
struct sockaddr_in adrserv;
int sserv = socket (AF_INET, SOCK_STREAM, 0);
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
adrserv.sin_addr.s_addr = htonl (INADDR_ANY);
adrserv.sin_port = htons (4522);
bind (sserv, (struct sockaddr *)&adrserv, sizeof(adrserv));
listen (sserv,5);
while (1) {
    char requete[6];
    int scl = accept (sserv, NULL, NULL);
    read(scl, requete, 6);
    if (strcmp(requete, "hello") == 0) write(scl, "world", 6);
    else write(scl, "bouh", 5);
    close(scl);
}
```

Exemple – mode non connecté



Exemple – mode non connecté

Le client

```
struct sockaddr_in adrserv;
struct hostent *sp;
char *requete = "hello";
char reponse[10];

int scli = socket (AF_INET, SOCK_DGRAM, 0);
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
sp = gethostbyname("turing.enseeiht.fr"); // assume OK
memcpy (&sins.sin_addr, sp->h_addr_list[0], sp->h_length);
adrserv.sin_port = htons (4522);

sendto (scli, requete, strlen(requete), 0,
                    (struct sockaddr *)&adrserv, sizeof(adrserv));
recvfrom (scli, reponse, sizeof (reponse), 0, NULL, NULL);
close (scli);
```


Exemple – mode non connecté

Le serveur

```
struct sockaddr_in adrserv, adrcli; char requete[10];

int sserv = socket (AF_INET, SOCK_DGRAM, 0);
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
adrserv.sin_addr.s_addr = htonl (INADDR_ANY);
adrserv.sin_port = htons (4522);
bind (sserv, (struct sockaddr*)&adrserv, sizeof(adrserv));
while (1) {
    bzero (&adrcli, sizeof (adrcli));
    int adrclilen = sizeof (adrcli);
    recvfrom (sserv, requete, sizeof(requete), 0,
                  (struct sockaddr *)&adrcli, &adrclilen);
    sendto (sserv, "world", 6, 0,
                  (struct sockaddr *)&adrcli, adrclilen);
}
```


Création d'un socket

`socket` crée un socket en spécifiant le famille de protocole utilisée.

```
int socket(int domain, int type, int protocol)
```

où

- domain = `AF_INET`, `AF_INET6`, `AF_UNIX`, `AF_X25`...
- type = `SOCK_STREAM`, `SOCK_DGRAM`
- protocol = 0

Retour : un « descripteur » de fichier ou -1

Adresse

```
struct sockaddr_in {
       short sin_family;
       u_short sin_port;
       struct in_addr sin_addr;
       char sin_zero[8];
};
```

- `sin_family` doit être `AF_INET`
- `sin_port` numéro de port sur 16 bits, dans une représentation standard : "network byte ordered" (big endian)
- `sin_addr.s_addr` = adresse IP sur 32 bits, correctement ordonnée (big endian)

Représentation des entiers

La pile IP est « big endian »

Conversion de données (numéro de port, adresse IP)

`htonl`	*host-to-network, long int*
`htons`	*host-to-network, short int*
`ntohl`	*network-to-host, long int*
`ntohl`	*network-to-host, short int*

Conversion ascii ↔ `in_addr`

```
int inet_aton(const char *cp, struct in_addr *inp);
char  *inet_ntoa (struct in_addr in);
```

Service de nommage (DNS)

```
struct hostent {
  char   *h_name;       /* nom canonique */
  char   **h_aliases;   /* liste d'alias */
  int    h_addrtype;    /* type des adresses */
  int    h_length;      /* longueur d'une adresse */
  char   **h_addr_list; /* liste d'adresses */
}; */
struct hostent *gethostbyname(char *name);
struct hostent *gethostbyaddr(void *addr, int len, int type);
```

- `gethostbyname` avec un nom de machine `turing.enseeiht.fr` permet d'obtenir ses autres noms et son (ses) adresse(s). Actuellement `h_addrtype == AF_INET` ou `AF_INET6`, et utiliser `h_addr_list[0]`.
- Pour une adresse format `sin_addr` de `type == AF_INET`, `gethostbyaddr` permet d'obtenir le(s) nom(s) en clair.

Liaison socket/adresse

`bind` nomme localement le socket (machine, port).
Obligatoire pour accepter des connexions ou recevoir des messages.

```
int bind(int sd, struct sockaddr *addr, int addrlen);
```

où

- sd : descripteur du socket
- addr : adresse attribuée à ce socket
- addrlen : taille de l'adresse (`sizeof(struct sockaddr_in)`)

Retour : 0 si ok, -1 si échec avec `errno` :

- EACCESS = permission refusée (adresse réservée)
- EADDRINUSE = adresse déjà utilisée pour un nommage
- ...

Connexion (côté client)

`connect` identifie l'extrémité distante d'une association.

```
int connect(int sd, struct sockaddr *addr, int addrlen);
```

où

- sd : descripteur du socket (du client)
- addr : adresse du socket du serveur
- addrlen : taille de l'adresse

Retour : 0 si ok, -1 si échec.

- EISCONN = socket déjà connecté
- ECONNREFUSED = connexion refusée (pas d'écouteur)
- ENETUNREACH = réseau inaccessible
- ETIMEDOUT = délai de garde expiré avant l'établissement de la connexion
- ...

Déclaration du serveur

`listen` établit une file d'attente pour les demandes de connexions.

```
int listen(int sd, int backlog);
```

où

- sd : descripteur du socket (du client)
- backlog : nombre max de clients en attente

Retour : 0 si ok, -1 si échec.

- EADDRINUSE = un autre socket déjà à l'écoute sur le même port.

Acceptation d'une connexion

`accept` prend la première demande de connexion et crée un nouveau socket ayant les mêmes caractéristiques que `sd` mais connecté à l'appelant $\Rightarrow$ établissement d'une association.

```
int accept(int sd, struct sockaddr *peer, int *addrlen);
```

où

- sd : socket existant de type STREAM
- peer : adresse du socket du client (valeur en retour)
- addrlen = taille de l'adresse fournie et taille de l'adresse retournée (utiliser `sizeof(struct sockaddr_in)`)

Retour : un nouveau descripteur de socket si ok, ou -1 en cas d'erreur

- EOPNOTSUPP = sd n'est pas de type STREAM

Communication de données

- Appels système classiques : `read`/`write`.

```
int write(int sd, const void *buf, int len);
int read(int sd, void *buf, int len);
```

- Flot d'octets : les frontières entre les messages ne sont pas préservées
- $\Rightarrow$ protocole applicatif nécessaire

Communication, mode non connecté

```
int sendto(int sd, void *buf, int len, int flags,
           struct sockaddr *dest, int addrlen);
int recvfrom(int sd, void *buf, int len, int flags,
             struct sockaddr *src, int *addrlen);
```

où

- sd : socket
- buf, len : message à envoyer
- dest : adresse du socket destinataire (sendto, entrée)
- src : adresse du socket émetteur (recvfrom, sortie)
- addrlen : longueur de l'adresse (entrée et sortie pour recvfrom)

Retour : ≥ 0 si nombre d'octets émis/reçus, -1 si erreur

Liaisons implicites/explicites

bind implicite

- Lors d'un `connect` ou d'un `sendto`, le socket doit avoir une adresse locale ⇒ attribution d'un numéro de port non utilisé si nécessaire.
- Il est possible de nommer le socket (`bind`) avant `connect` ou `sendto`, mais guère d'utilité.

connect explicite

Il est possible de « connecter » un socket en mode datagramme (utilisation de `connect` sur un socket `SOCK_DGRAM`) :

- plus nécessaire de spécifier le destinataire de *chaque* message
- sans `connect`, le même socket peut être utilisé vers différents destinataires

Fermeture

- `close` termine l'association et libère le socket après avoir délivré les données en attente d'envoi.

```
int close(int sock);
```

Note : comportement inattendu s'il reste des octets à lire (écritures en attente perdues).

- `shutdown` permet une fermeture unilatérale :

```
int shutdown(int sock, int how);
```

où `how` = `SHUT_RD` (fin de réception), `SHUT_WR` (fin d'émission), `SHUT_RDWR` (fin de réception et d'émission)
Rq : il faudra quand même appeler `close` pour libérer les ressources du système.

Configuration (en tant que fichier)

```
int fcntl(int fd, int cmd, ...)
```

Par exemple : `fcntl(sd, F_SETFL, O_NONBLOCK)`
pour mettre en non bloquant.

Configuration (en tant que socket)

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

Par exemple :

```
int ra = 1;
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &ra, sizeof(ra));
```

level	option	description
SOL_SOCKET	SO_REUSEADDR	réutilisation immédiate d'adresse locale
	SO_KEEPALIVE	maintient en vie une connexion
	SO_BROADCAST	autorise les diffusions
IPPROTO_TCP	TCP_MAXSEG	taille max d'un segment TCP
	TCP_NODELAY	disable Nagle's algorithm
IPPROTO_IP	IP_OPTIONS	options des entêtes IP

Adresse d'un socket

- obtenir l'adresse de l'extrémité locale d'une association

```
int getsockname(int sd,
                struct sockaddr *addr, int *addrlen);
```

- obtenir l'adresse de l'extrémité distante d'une association (mode connecté)

```
int getpeername(int sd,
                struct sockaddr *peer, int *addrlen);
```

Plan

Les classes

- `java.net.InetAddress` pour manipuler des adresses IP
- Mode connecté : `java.net.Socket` et `java.net.SocketServer`
- Mode datagramme : `java.net.DatagramSocket` et `java.net.DatagramPacket`

Note : les interfaces présentées sont incomplètes (exceptions supprimées).

La classe java.net.InetAddress

- Deux sous-classes `Inet4Address`, `Inet6Address`
- Obtention :
  - `static InetAddress getLocalHost();`
    renvoie l'adresse IP du site local d'appel.
  - `static InetAddress getByName(String host);`
    Résolution de nom (sous forme symbolique `turing.enseeiht.fr` ou numérique 147.127.18.03)
  - `static InetAddress[] getAllByName(String host);`
    Résolution de nom → toutes les adresses IP d'un site
- Accesseurs

  `String getHostName()`
  : le nom complet correspondant à l'adresse IP

  `String getHostAddress()`
  : l'adresse IP sous forme `d.d.d.d` ou `x:x:x:x:x:x:x:x`

  `byte[] getAddress()`
  : l'adresse IP sous forme d'un tableau d'octets.

Classe java.net.ServerSocket

Représente un socket d'écoute.

- Constructeurs :
  `ServerSocket(int port);`
  `ServerSocket(int port, int backlog, InetAddress bindAddr);`
  Réalise socket – bind – listen.
- Méthodes :
  - `Socket accept();`
    Renvoie un socket connecté. Bloquant. (= accept en C).
  - `InetAddress getInetAddress();`
    Renvoie l'adresse IP locale.
  - `int getLocalPort();`

La classe java.net.Socket

Représente un socket connecté, côté client comme côté serveur.

- Constructeurs (côté client) :

```
Socket(String host, int port);
Socket(InetAddress address, int port);
Socket(String host, int port,
           InetAddress localAddr, int localPort);
Socket(InetAddress addr, int port,
           InetAddress localAddr, int localPort);
```

  = `socket` – `bind` (éventuellement) – `connect`.
  Constructeur bloquant !
- Accès aux flots de données :

```
InputStream  getInputStream();
OutputStream getOutputStream();
```

La classe java.net.Socket (suite)

- Accesseurs :
  - `InetAddress getLocalAddress();`
    Renvoie l'adresse IP locale. (≈ `getsockname` en C)
  - `int getLocalPort();`
    Renvoie le port local.
  - `InetAddress getInetAddress();`
    Renvoie l'adresse IP distante. (≈ `getpeername` en C)
  - `int getPort();`
    Renvoie le port distant.

Socket connecté en Java : exemple client

```
public class Client {
 public static void main(String[] args) throws Exception {
   Socket socket = new Socket("bach.enseeiht.fr", 8080);
   // Un BufferedReader permet de lire par ligne.
   BufferedReader plec = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
   // Un PrintWriter possède toutes les opérations print classiques.
   // En mode auto-flush, le tampon est vidé (flush) lors de println.
   PrintWriter pred = new PrintWriter(
                  new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())),
                    true);
   String str = "bonjour";
   for (int i = 0; i < 10; i++) {
      pred.println(str + i);      // envoi d'un message
      str = plec.readLine();      // lecture de l'écho
   }
   pred.println("END") ;
   plec.close();
   pred.close();
   socket.close();
 }
}
```

Attention aux exceptions !

Socket connecté en Java : exemple serveur

```
public class Serveur {
 public static void main(String[] args) throws Exception {
   ServerSocket s = new ServerSocket(8080);
   while (true) {
      Socket soc = s.accept();
      BufferedReader plec = new BufferedReader(
                          new InputStreamReader(soc.getInputStream()));
      PrintWriter pred = new PrintWriter(
                      new BufferedWriter(
                       new OutputStreamWriter(soc.getOutputStream())),
                       true);
      while (true) {
         String str = plec.readLine();          // lecture du message
         if (str.equals("END")) break;
         pred.println(str);                     // renvoi d'un écho
      }
      plec.close();
      pred.close();
      soc.close();
   }
 }
}
```

Attention aux exceptions !

Socket en mode datagramme java.net.DatagramSocket

- Constructeurs :

```
DatagramSocket(); // port quelconque disponible
DatagramSocket(int port);
```

- Méthodes :

```
void send(DatagramPacket p);
void receive(DatagramPacket p);
```

- Classe java.net.DatagramPacket :

```
DatagramPacket(byte[] buf, int length);
DatagramPacket(byte[] buf, int length,
          InetAddress addr, int port); // destination
```

+ getters et setters

Socket datagramme en Java : exemple

```
import java.net.*;
import java.util.Arrays;

public class Client {
  public static void main(String[] args) throws Exception {
    DatagramSocket sock = new DatagramSocket();

    byte[] data = "question".getBytes();
    DatagramPacket msg = new DatagramPacket(data, data.length,
                                InetAddress.getByName("147.127.133.111")
                                7896);
    sock.send(msg);

    sock.receive(msg);
    byte[] res = Arrays.copyOfRange(msg.getData(), 0, msg.getLength());
    System.out.println("Réponse = " + new String(res));
  }
}
```

Attention aux exceptions !

Socket datagramme en Java : exemple suite

```
import java.net.*;

public class Serveur {
  public static void main(String[] args) throws Exception {
    DatagramSocket sock = new DatagramSocket(7896);

    DatagramPacket req = new DatagramPacket(new byte[256], 256);
    sock.receive(req);

    byte[] data = "42".getBytes();
    DatagramPacket ans = new DatagramPacket(data, data.length,
                                            req.getAddress(),
                                            req.getPort());
    sock.send(ans);
  }
}
```

Attention aux exceptions !

Conclusion

Principes de base

- Extension d'une notion issue du monde « centralisé »
- Connexion point à point entre processus
- Bases du modèle (protocole) client-serveur
- Communication en mode datagramme ou connecté
- Pas de transparence de la communication

Pour aller plus loin

- Trouver une abstraction du contrôle plus simple
  $\Rightarrow$ réutiliser la notion de procédure
- Principe de conception : idée de transparence

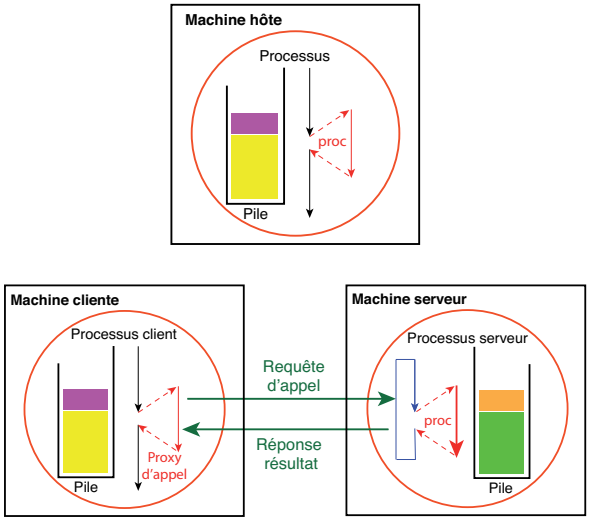
Troisième partie

Appel de procédure et de méthode à distance

Plan

Communication par appel procédural à distance

alias Remote Procedure Call (RPC)



Communication par appel procédural à distance

Extension répartie du mécanisme d'appel procédural

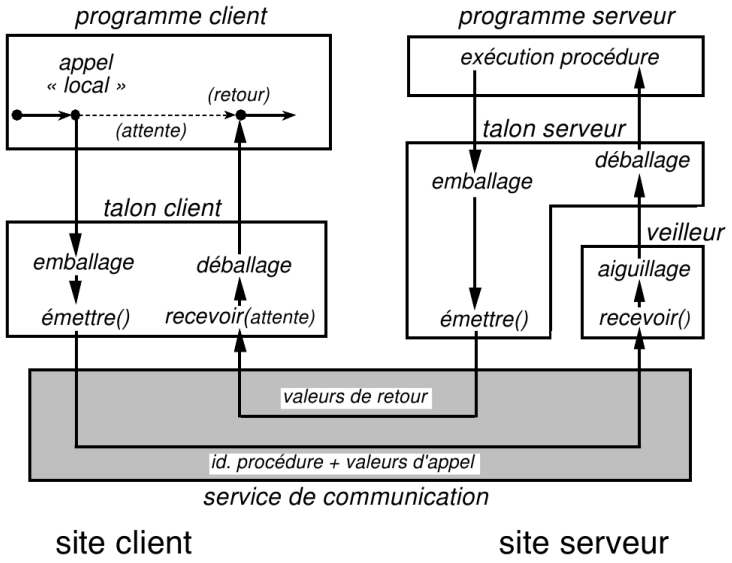
- Procédure appelée exécutée dans un espace différent de celui de l'appelant
- Synchronisation appelant-appelé
- Transaction de messages
  (question - réponse - [acquittement])
- Fiabilité bien moindre qu'en centralisé
- Comment transmettre les paramètres ?
- Problème de l'hétérogénéité
  - du matériel
  - du système d'exploitation
  - de la représentation des données (paramètres)
  - des langages de programmation

Transparence

But : rendre l'utilisation de l'appel à distance aussi conforme (*transparent*) que l'appel local de procédure

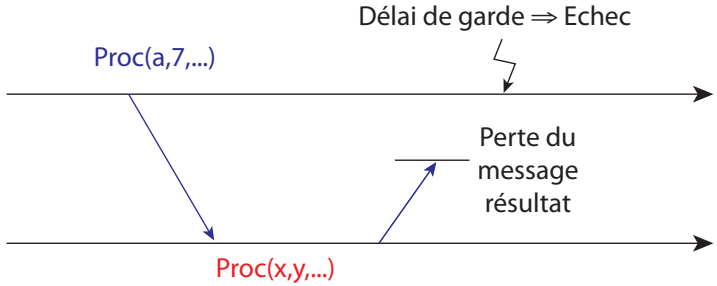
- passage des paramètres
- liaison (nommage)
- protocole de transport
- exceptions
- sémantique de l'appel
- représentation des données
- performance
- sécurité

Principe général



Sémantique de l'appel procédural à distance

Quelques problèmes...

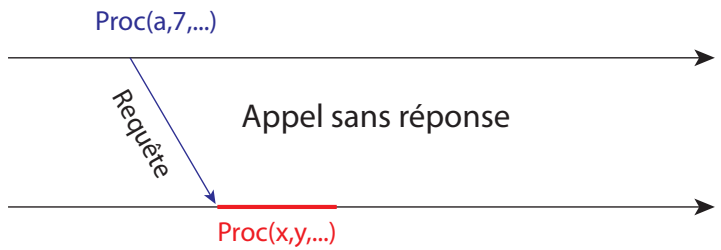


Plusieurs sémantiques possibles !

- « Sans garantie » (Maybe / Best effort)
- « Au moins une fois » (At-least-once)
- « Au plus une fois » (At-most-once)
- « Exactement une fois » (Exactly-once).

Sémantique de l'appel procédural à distance

Sémantique minimaliste : « Sans garantie »

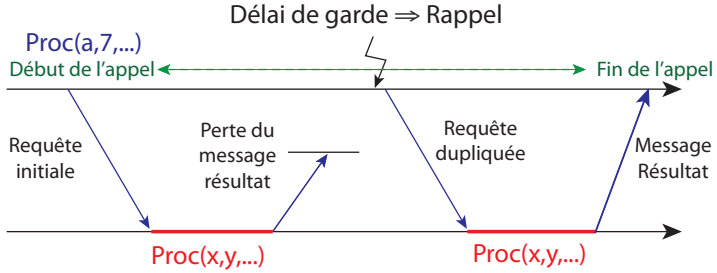


Avantages et inconvénients

- Simple à implanter : envoyer un message
- Pas de réponse donc pas de garantie d'exécution
- Utile dans certains cas (logging)

Sémantique de l'appel procédural à distance

Sémantique « Au moins une fois »

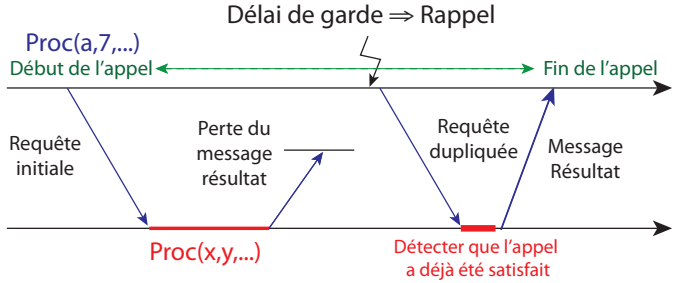


Avantages et inconvénients

- Robuste face aux pertes et lenteurs
- Si terminaison correcte → garantie d'une exécution au moins
- Si terminaison incorrecte (après plusieurs rappels), pas de garantie sur ce qui s'est passé à distance
- Risque de plusieurs exécutions pour un seul appel logique

Sémantique de l'appel procédural à distance

Sémantique « Au plus une fois »

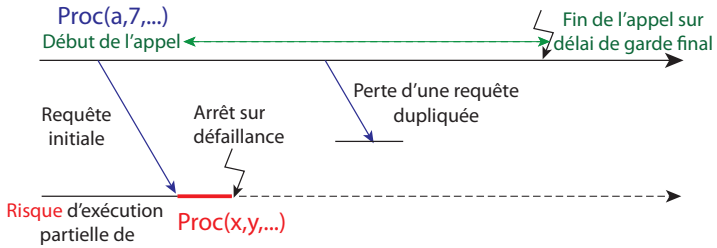


Avantages et inconvénients

- Plus proche de l'appel procédural centralisé
- Terminaison correcte ⇒ garantie d'une seule exécution à distance
- Terminaison incorrecte : pas de garantie sur ce qui s'est passé à distance ⇒ entre autre, risque d'exécution partielle à distance

Sémantique de l'appel procédural à distance

Sémantique « Exactement une fois »



Avantages et inconvénients

- Si terminaison correcte ⇒ équivalent à « au + une fois »
- Si terminaison incorrecte ⇒ garantie d'atomicité à distance

Espace d'adressage séparés

+ Isolation des données du serveur

- Sécurité
- Conception modulaire

– Passage des paramètres au moyen de messages

- pas de variables globales implicites
- passage par valeur (copie)
- Références ?
  - Interdire le passage de références ⇒ expressivité faible
  - Sérialisation : transfert et reconstruction du *graphe* (structure de données)
  - Références globales opaques (objets répartis)

Représentation des données

Problèmes

- hétérogénéité du matériel
- taille des types élémentaires (booléen, entier...)
- ordre des octets pour les entiers
- nombres réels
- codage des caractères
- données composites : structures, tableaux

Approches

1. définir une représentation standard
   - typage implicite : seules les valeurs sont transmises
   - typage explicite : information de type + valeur
2. préfixer par le type local, et laisser le destinataire convertir si nécessaire

Désignation et liaison

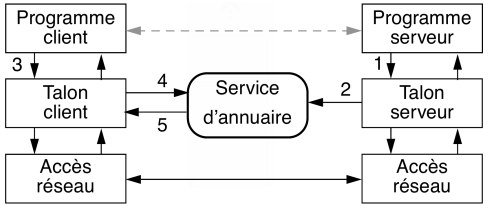
Correspondance : nom symbolique (externe) → nom interne (adresse réseau, identifiant local)
Cas des RPC :

- nom de service → port et adresse serveur
- nom d'opération → procédure sur le serveur correspondant

Instant(s) d'évaluation de la liaison

- Liaison statique (précoce) : localisation du serveur fixée au moment de la compilation du programme client
- Liaison dynamique (tardive) : localisation à l'exécution
  - désignation symbolique des services
  - choix retardé de l'implémentation
  - localisation du service au premier appel seulement, ou à chaque appel
  - adaptation à une reconfiguration du système : régulation, pannes, évolutions

Liaison dynamique : serveur de noms

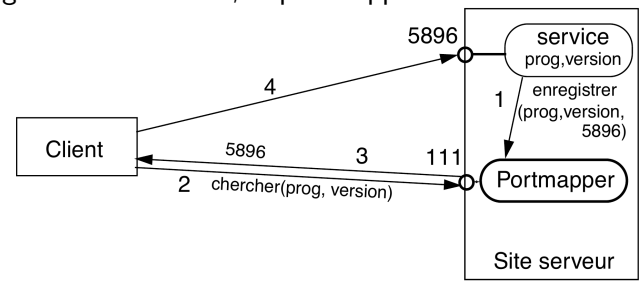


- 1,2 : Enregistrement du service dans l'annuaire sous : ⟨nom⟩ → ⟨adr. serv., n° port⟩
- 3,4,5 : Consultation de l'annuaire pour trouver ⟨adr. serv., n° port⟩ à partir de ⟨nom⟩
- L'appel peut alors avoir lieu

- Tolérance aux pannes (service critique) : mémoire stable, duplication des tables, des serveurs
- Localisation du serveur de noms :
  - diffusion de la requête par le client
  - ou variable de configuration du système
  - ou utilisation d'une adresse conventionnelle

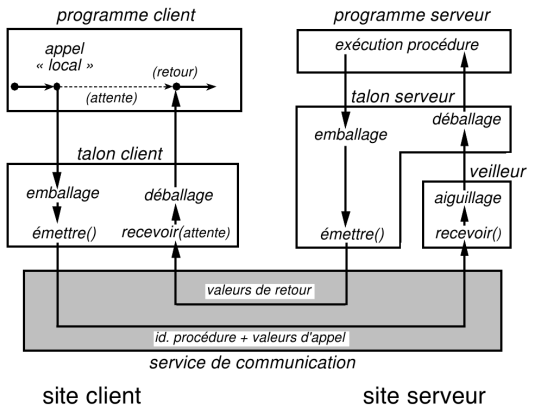
Serveurs de noms local à un site : portmapper

Cas où le site hébergeant le service est connu, mais où le port correspondant au service n'est pas connu ⇒ utiliser un service de nommage local au serveur, le portmapper.



- Le portmapper a un numéro de port fixé par convention (111)
- Un service enregistre le numéro de port de son veilleur auprès du portmapper
- Le veilleur se met en attente sur ce port

Mise en œuvre

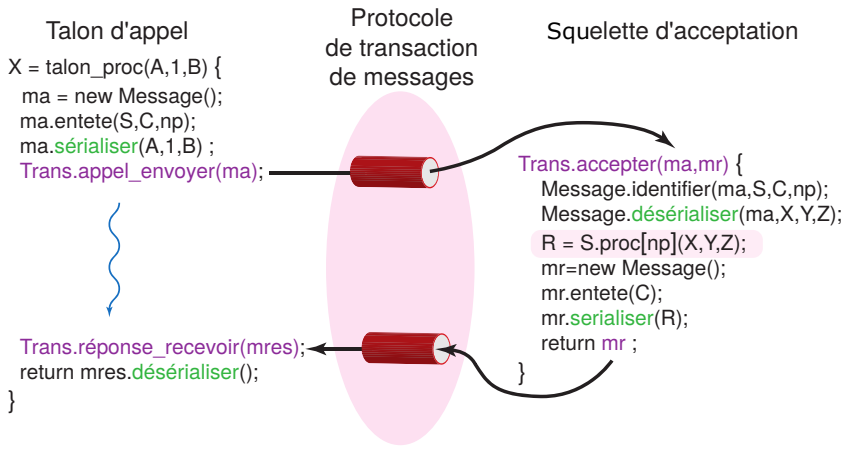


- Définition d'un protocole de transaction de messages
- Sérialisation/Désérialisation des paramètres
- Génération de talon d'appel (stub) et d'acceptation (skeleton)
- Assistance par génération automatique de code

Langage de description d'interface (IDL)

Mise en œuvre

Niveaux de protocole



Description d'interface en IDL

Exemple minimaliste (SUN)

Les langages IDL (Interface Definition Language)

- Langage commun de description d'interface
- Purement déclaratif : types de données, interfaces
- Base pour la génération des talons et squelettes

```
struct arg { int a1; int a2; };
program MONSERVICE {
    version MAVERSION {
        int PROC1 (arg) = 1;
        int PROC2 (int, int) = 2;
    } = 1;
} = 0x30000050;
```


eXternal Data Representation

- Description et codage des données indépendantes du matériel
- Structures de données arbitraires (structures, tableaux, séquences)
- RPC convertit les données machine en XDR avant de les envoyer sur le réseau *(sérialisation)*
- RPC convertit les données XDR en données machine après lecture sur le réseau *(désérialisation)*
- Génération automatique des routines de sérialisation et de désérialisation à partir d'une description proche du langage C
- ou codage manuel de ces routines à partir d'une librairie pour les types élémentaires

Programmation par appel explicite

```
int callrpc (const char *host,
   u_long prognum, u_long versnum, u_long procnum,
   xdrproc_t inproc, char *in,
   xdrproc_t outproc, char *out);
```

Appel d'une procédure ⟨prognum, versnum, procnum⟩ sur la machine host. La procédure reçoit en paramètre in codé avec inproc et retourne out décodé avec outproc.

```
int registerrpc (u_long prognum, u_long versnum,
      u_long procnum,
      char * (*procname) (char *arg),
      xdrproc_t inproc, xdrproc_t outproc);
```

Enregistrement de la procédure procname sous le nom ⟨prognum, versnum, procnum⟩. Le paramètre est décodé avec inproc et le résultat est encodé avec outproc.

Appel explicite : client

```
#include <stdio.h>
#include <rpc/rpc.h>
int main()
{
    int stat;
    int result;
    char *argin = "546";
    stat = callrpc (/*host*/ "cotiella",
                    /*prognum*/ 87654321, /*versnum*/ 1, /*procnum*/ 12,
                    /*inproc*/ xdr_string, /*in*/ (char*) &argin,
                    /*outproc*/ xdr_int, /*out*/ (char*) &result);
    if (stat != 0) { fprintf (stderr, "Call failed: "); /*...*/ }
    printf ("Call succeeded: \"%s\" = %d\n", argin, result);
}
```

Appel explicite : serveur

```
#include <rpc/rpc.h>
#include <stdlib.h>
int resultat;
char *maproc (char *argin)
{
    char *argument = *(char**)argin;
    resultat = atoi(argument);
    return (char*) &resultat;
}
int main()
{
    int stat;
    stat = registerrpc (/*prognum*/ 87654321, /*versnum*/ 1, /*procnum*/ 12,
                        /*procname*/ maproc,
                        /*inproc*/ xdr_string, /*outproc*/ xdr_int);
    if (stat != 0) { fprintf (stderr, "Registering failed\n"); /* ... */ }
    svc_run(); /* veilleur/aiguilleur */
}
```

Diffusion & protocole

```
typedef bool_t (*resultproc_t) (char *out,
                                struct sockaddr_in *addr);
int clnt_broadcast(u_long prognum, u_long versnum,
                   u_long procnum,
                   xdrproc_t inproc, char *in,
                   xdrproc_t outproc, char *out,
                   resultproc_t eachresult);
```

Envoi diffusé sur le réseau, la fonction eachresult est appelée pour chaque réponse obtenue.

Protocole de transport

Par défaut pour callrpc et clnt_broadcast, le protocole de communication est UDP. Pour callrpc, possibilité d'utiliser TCP via client_create (à suivre).

Génération automatique : rpcgen

À partir d'une description des types et d'une déclaration des procédures, RPCGEN engendre :

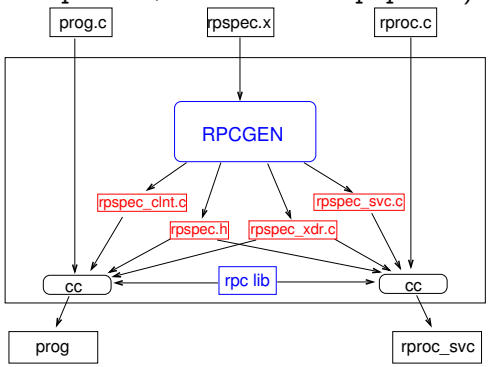
- les routines XDR pour convertir les types de données
- le talon client masquant l'appel à distance
- le talon serveur gérant l'appel
- la procédure principale (`main`) et la procédure de sélection (`dispatch`) pour le serveur

Le programmeur doit :

- décrire les types de données échangées
- écrire la programme principal client
- écrire le code des procédures coté serveur

RPCGEN : exemple

Un programme (fichier `prog.c`) appelle deux procédures distantes (implantées dans `rproc.c`, décrites dans `rpspec.x`) d'un serveur




```
$ rpcgen rpspec.x
$ cc -o server rproc.c rpspec_svc.c rpspec_xdr.c -lnsl
$ cc -o client prog.c rpspec_clnt.c rpspec_xdr.c -lnsl
```

RPCGEN : Fichier d'interface

`rpspec.x` contient :

- les noms, et les numéros de programme, de version et de procédures, des procédures distantes
- les types de données manipulés

```
/* rpspec.x */
struct arga {
        int arga1;
        int arga2;
};

program MONPROG {
   version MAVERS {
        long RPROCA (string, int) = 1; /* proc #1 */
        arga RPROCB (void) = 2;        /* proc #2 */
   } = 1;                              /* version #1 */
} = 0x1234567;                         /* prog num */
```

RPCGEN : un exemple de client

```
#include <rpc/rpc.h>
#include "rpspec.h"
main() {
   CLIENT *cl;
   long *res1;
   struct arga *res2;

   /* Mise en place du lien. cotiella est le nom de la machine distante. */
   cl = clnt_create ("cotiella", MONPROG, MAVERS, "tcp");
   if (cl == NULL) { clnt_pcreateerror ("cotiella"); exit(1); }

   res1 = rproca_1 ("45", 42, cl); /* appel de rproca version 1 */
   if (res1 == NULL) { clnt_perror (cl, "failed"); exit (1); }
   printf("Result: %ld\n", *res1);

   res2 = rprocb_1 (cl);  /* appel de rprocb version 1 */

   clnt_destroy (cl);     /* fin du lien */
}
```

RPCGEN : procédures du serveur

```
#include "rpspec.h"

/* Implantation de RPROCA version 1 */
/* Passage par pointeur du retour */
/* req contient des informations de contexte */
long *rproca_1_svc (char *arg1, int arg2, struct svc_req *req) {
   static long res;
   res = atoi(arg1) + arg2;
   return &res;
}

/* Implantation de RPROCB version 1 */
struct arga *rprocb_1_svc (struct svc_req *req) {
    /* ... */
}
```

RPC : bilan

Apports

- Transparence partielle (désignation, localisation)
- Modèle de programmation classique (appel procédural ↔ interaction C/S)
- Conception modulaire
- Outils de génération automatique des talons

Limitations

- Développement traditionnel monolithique
- Construction et déploiement statique
- Structure d'exécution asymétrique, centralisée sur le serveur
- Peu de services extra-fonctionnels : supervision, équilibrage de charge, tolérance aux pannes, données rémanentes...

Et variantes

XML-RPC

Un protocole RPC qui utilise XML pour encoder les données et HTTP comme mécanisme de transport.

SOAP (Simple Object Access Protocol)

Architecture client-serveur via des échanges de messages décrits en XML. Neutralité du transport (HTTP, SMTP, TCP, JMS).

REST (Representational state transfer)

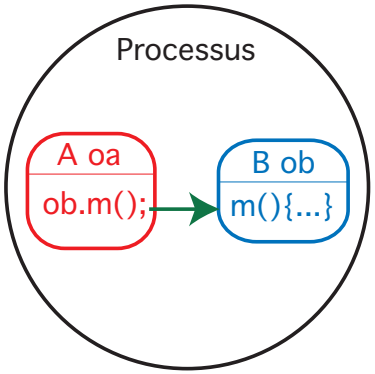
Règles architecturales de construction d'applications client-serveur, s'appuyant sur HTTP et axé sur les ressources (RPC est axé sur les actions). Une règle essentielle est que la relation client-serveur est sans état (pas de session).

Plan

L'appel de méthode centralisé

Principe et propriétés

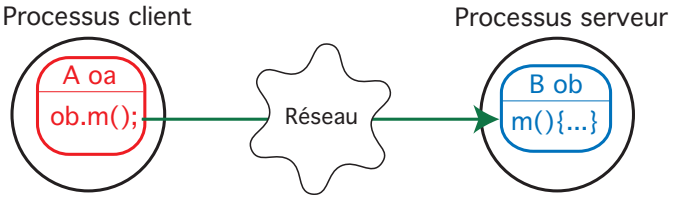
- Un seul espace d'exécution
- Point de contrôle unique
- Fort couplage
- Fiabilité
- Sécurité



L'appel de méthode à distance

Principe et propriétés

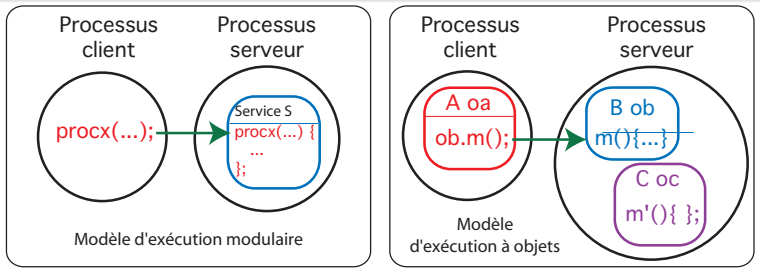
- Deux espaces d'exécution
- Deux points de contrôle
- Couplage plus faible
- Protocole de communication entre processus.



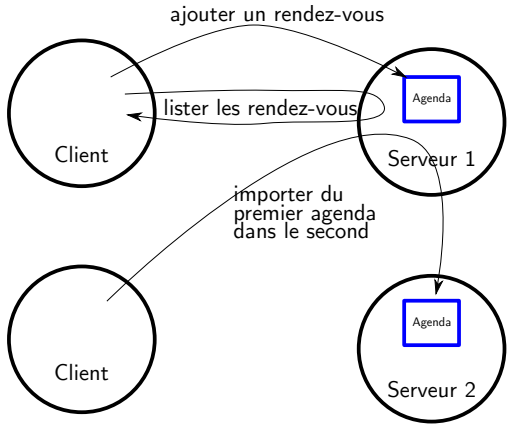
L'appel de méthode à distance

Différences avec l'appel de procédure

- Contexte d'exécution différent : l'un module, l'autre objet
- Appel d'une méthode sur un objet
- Aspect dynamique : création de services (par création d'objets) + transmission de services à distance

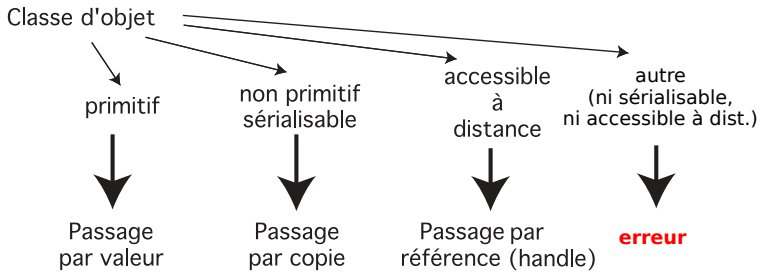


Appel de méthodes à distance



- Passer/recevoir des valeurs (des objets par copie)
- Passer/recevoir des références à des objets

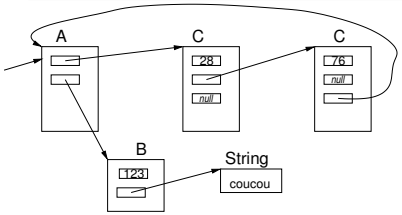
Le passage de paramètres



Sérialisation

Définition

La sérialisation d'un graphe d'objets consiste à obtenir une représentation linéaire inversible de ces objets et de leurs relations. La désérialisation reconstruit une forme interne et structurée du graphe d'objet.



- but : exportation vers un fichier ou un autre processus
- Difficulté : la présence de cycles

Sérialisation

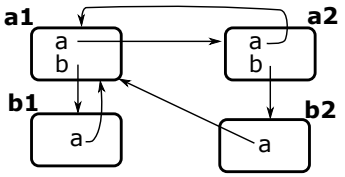
Un objet est sérialisable, s'il appartient à une classe :

- qui implante l'interface `java.io.Serializable`
  ⇒ pas de code à fournir, mécanisme par défaut :
  Sont récursivement sérialisés les attributs non statiques ni *transient*s contenant :
  - des types primitifs (`int`, `bool`...)
  - ou des objets qui doivent être sérialisables.
- ou qui implante l'interface `java.io.Serializable` et fournit les méthodes (code utilisateur arbitraire) :

```
private void writeObject(java.io.ObjectOutputStream out)
                throws IOException;
private void readObject(java.io.ObjectInputStream in)
                throws IOException, ClassNotFoundException;
```


Exemple sérialisation

```
class A implements java.io.Serializable {
  public B b;
  public A a;
}
class B implements java.io.Serializable {
  public A a;
}
  A a1 = new A();    A a2 = new A();
  B b1 = new B();    B b2 = new B();
  a1.a = a2;         a1.b = b1;
  a2.a = a1;         a2.b = b2;
  b1.a = a1;         b2.a = a1;
  ObjectOutputStream oos = new ObjectOutputStream(
                             new FileOutputStream("/tmp/toto"));
  oos.writeObject(a1);
```



Compatibilité de versions

Comment s'assurer, à l'endroit et au moment de la désérialisation, que l'implantation de la classe est la même qu'à la sérialisation (mêmes attributs en particulier) ?
⇒ gestionnaire de version des classes

Solution élémentaire : un attribut statique `serialVersionUID` (type long) dans chaque classe :

```
private static final long serialVersionUID = 76428734L;
```

Par défaut si absent, le compilateur calcule un tel champ (à partir des attributs notamment), mais le calcul est sensible à son humeur ⇒ à gérer soi-même.

Le mécanisme RMI (Remote Method Invocation)

Proxy

Objet local « remplaçant » l'objet distant = objet ayant la même interface que l'objet distant, et sachant appeler l'objet distant.

Servant

Objet interne sachant discuter à distance avec des proxys et localement avec l'objet applicatif.

Service de nommage

Désignation globale par serveurs de noms (Registry)

Talons : proxy/servant

Proxy = talon client = stub = a la même interface que l'objet applicatif distant.

Servant = talon serveur = squelette = reçoit les requêtes, appelle la méthode correspondant de l'objet applicatif, et gère les erreurs. Le servant peut être un objet distinct (association), ou être commun à l'objet applicatif (héritage).

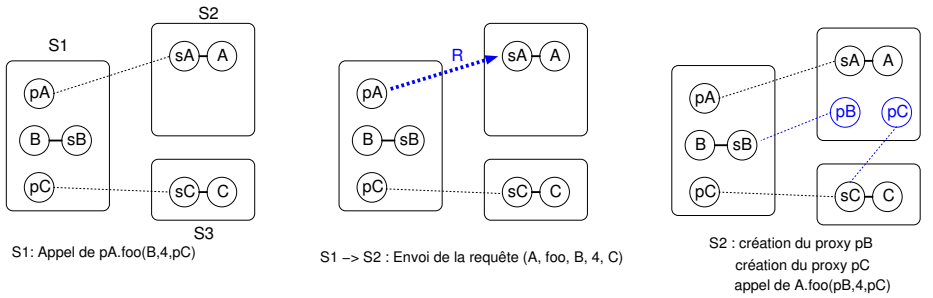
Obtenir un proxy ?

Comment obtenir un proxy sur un objet distant :

- Utiliser un service de nommage, qui conserve des associations entre objet accessible à distance et *nom externe* (une chaîne de caractères ou un URL) : le client demande au service de nommage de lui fournir un proxy correspondant à un nom externe donné.
- Avoir appelé une méthode (à distance) qui transmet/renvoie un (autre) objet accessible à distance : création implicite des proxys.

Le mécanisme RMI

La gestion des proxys et squelettes



Passage des paramètres

`oad.foo(param)`, où l'appel a lieu sur un site $S1$ et `oad` est situé sur un site $S2 \Rightarrow$ l'exécution effective de `foo` a lieu sur $S2$.
Que se passe-t-il pour `param` ?

- Valeur d'un type primitif (int, bool) $\rightarrow$ copie
- Objet sérialisable $\rightarrow$ copie
- Objet accessible à distance situé sur $S1 \rightarrow$ un proxy vers *param* est créé sur $S2$ (ou réutilisé si déjà existant)
- Proxy vers un objet oad' situé sur $S3$ ($\neq S2$) $\rightarrow$ un proxy vers oad' est créé sur $S2$ (ou réutilisé si déjà existant)
- Proxy vers un objet oad' situé sur $S2 \rightarrow$ l'objet natif oad' de $S2$ est utilisé

Et pour une valeur en retour ? Même mécanisme.

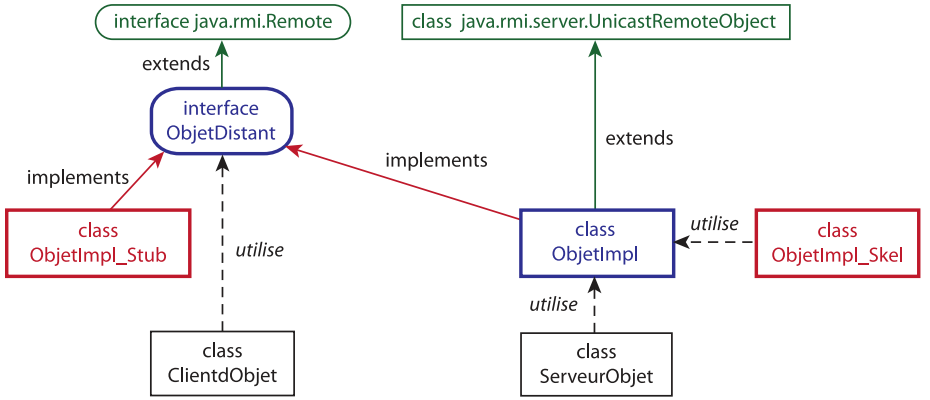
Réalisation

L'environnement Java fournit :

- la génération dynamique des talons, en s'appuyant sur l'API d'introspection
  - le proxy est généré à l'exécution, lors de la création d'une référence à un objet accessible à distance, à partir de l'environnement d'exécution du serveur ;
  - les fonctions du servant sont fournies et intégrées à l'objet accessible à distance, par héritage ;
  - historiquement, il existait un générateur statique de talons (`rmic`), analogue à `rpcgen`.
- un service de nommage (package `java.rmi.registry`), pour nommer symboliquement des objets accessibles à distance.
- un mécanisme de chargement de code dynamique, qui permet aux clients de charger le code des objets fournis en paramètre lorsqu'il n'est pas disponible localement (en particulier le code des proxys).

Mise en œuvre du protocole

Composants d'une classe d'objet accessible à distance



La description d'une classe d'objet accessible à distance

Définition de l'interface d'appel

- Hérite de l'interface `Remote`
- Chaque méthode lève l'exception `RemoteException`
- Un objet local non primitif doit être sérialisable :
  `class ObjParam implements Serializable`
- Un objet accessible à distance peut être passé en paramètre

Exemple

```
interface Agenda extends Remote {
   void ajouter(RendezVous r) throws RemoteException;
   RendezVous[] lister(String nom) throws RemoteException;
   void importer(String nom, Agenda orig)
                            throws RemoteException;
   ...
}
```

La description d'une classe d'objet accessible à distance

Définition de la classe d'implantation

- Hérite de la classe `UnicastRemoteObject`
- Implante l'interface du proxy correspondant

Exemple

```
class AgendaImpl extends UnicastRemoteObject
                 implements Agenda {
    void ajouter(RendezVous rdv) {
        ...
    }
    RendezVous[] lister(String nom) {
        ...
        return tab;
    }
    ...
}
```

Exemple : un agenda

Rendez-vous transmis par copie → réalise l'interface `Serializable`

```
public class RendezVous implements java.io.Serializable {
  private String qui;
  private java.util.Date date;       // est sérialisable
  private java.time.Duration duree;  // est sérialisable
  private String salle;

  public String toString(){... }

  public RendezVous (String qui, Date date, Duration duree,
                     String salle) {
    this.qui = qui;
    this.date = date;
    this.duree = duree;
    this.salle = salle;
  }
}
```

Interface de l'agenda accessible à distance

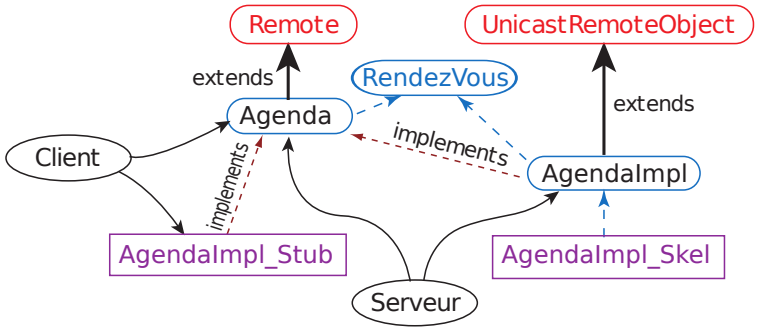
```
import java.rmi.*;
import java.util.Date;
import java.time.Duration;
interface Agenda extends Remote {
  public void ajouter(RendezVous rdv)
                                  throws RemoteException;
  public boolean deplacer(RendezVous rdv, Date date)
                                  throws RemoteException;
  public boolean estLibre(Date date, Duration duree)
                                  throws RemoteException;
  public void effacer(String nom, Date date)
                                  throws RemoteException;
  public RendezVous[] lister(String nom)
                                  throws RemoteException;
  public void importer(String nom, Agenda orig)
                                   throws RemoteException;
}
```

Agenda : une implantation

```
import java.rmi.*;
import java.rmi.server.*;

public class AgendaImpl
         extends UnicastRemoteObject implements Agenda {
   private Set<RendezVous> table = new HashSet<>();

   AgendaImpl() throws RemoteException { }

   public void ajouter(RendezVous rdv) { table.add(rdv); }
   ⋮
   public void importer(String nom, Agenda orig)
                              throws RemoteException {
       RendezVous[] rr = orig.lister(nom);
       for (RendezVous r : rr)  this.table.add(r);
   }
}
```

Structure des classes

- Génération statique ou dynamique des stubs et des skeletons.



⇒ Cas statique : utilisation du générateur `rmic` (obsolète)

Agenda : un programme serveur

Cas où le serveur de noms est créé en tant que thread interne

```
import java.rmi.registry.*;

public class Serveur {
  public static void main(String args[]) throws Exception {
    Agenda aa = new AgendaImpl();
    Registry dns = LocateRegistry.createRegistry(1099);
    dns.bind("Travail",aa);
  }
}
```

Agenda : un programme serveur

Cas où le serveur de noms existe ailleurs

```
import java.rmi.registry.*;

public class Serveur {
  public static void main(String args[]) throws Exception {
    Agenda aa = new AgendaImpl();
    Registry dns =
           LocateRegistry.getRegistry("toto.enseeiht.fr",1099);
    dns.bind("Travail",aa);
  }
}
```

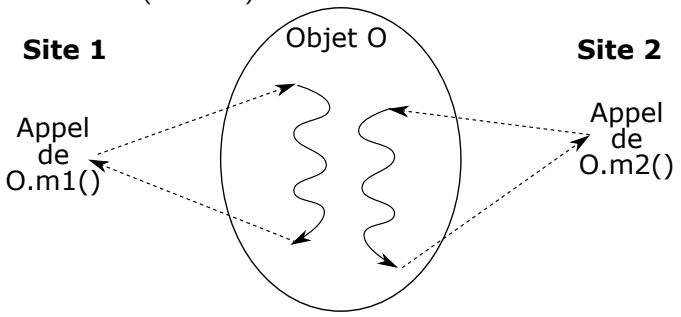
Agenda : un programme client

```
import java.rmi.registry.*;

public class Client  {
 // args[0] contient le site support du service de nommage
 public static void main(String args[]) throws Exception {
  Registry dns = LocateRegistry.getRegistry(args[0],1099);
  Agenda proxy = (Agenda)dns.lookup("Travail");
  RendezVous rdv = new RendezVous("chef", new Date(...),
                                  Duration.ofMinutes(30), "F303")
  proxy.ajouter(rdv);
  RendezVous[] tous = proxy.lister("chef");
  for (RendezVous r : tous)  System.out.println(r.date);
 }
}
```

Exemple d'appel : `java Client toto.enseeiht.fr`

Concurrence

Chaque invocation d'une méthode par un appel à distance se fait dans une activité (thread) distincte.



⇒ nécessité de gérer la protection des attributs et autres données partagées (p.e. `synchronized` pour toutes les méthodes).

Interface du service de nommage

`java.rmi.registry.Registry` : interface d'accès au nommage.
Les noms d'objets doivent obéir à la syntaxe générale des URL : `rmi://host:port/name`, ou `name` si le contexte est non ambigu.

```
public class Registry {
   public Remote lookup(String name)
            throws NotBoundException, MalformedURLException,
                   UnknownHostException, RemoteException;
  public void bind(String name, Remote obj)
            throws AlreadyBoundException, MalformedURLException,
                   UnknownHostException, RemoteException;
  public void unbind(String name)
             throws RemoteException, NotBoundException,
                    MalformedURLException, UnknownHostException;
  public String[] list(String name)
             throws RemoteException, MalformedURLException,
                    UnknownHostException;
}
```

Localisation et accès au service de nommage

La classe `java.rmi.registry.LocateRegistry` offre un ensemble de méthodes pour créer ou obtenir l'accès à un serveur de noms local ou distant :

```
public final class LocateRegistry {
    public static Registry getRegistry(String host, int port)
                throws RemoteException, UnknownHostException;
    ...
    public static Registry createRegistry()
                throws RemoteException;
}
```

Remarques sur le service de nommage

- Le service de nommage peut aussi être une application autonome (`programme rmiregistry`), et peut résider sur un site différent.
- Un objet accessible à distance n'a pas nécessairement à être enregistré dans le service de nommage : seuls les objets racines le doivent.

Implantation du service de nommage

Le service de nommage n'est lui-même qu'un objet accessible à distance. C'est un objet « notoire », avec une identité fixée : ⟨adr machine, n^o port⟩ suffit à le trouver.

```
class RegistryImpl extends java.rmi.server.RemoteServer {
  private Map<String, Remote> bindings
              = new HashMap<String, Remote>();

  public Remote lookup(String name)
           throws RemoteException, NotBoundException
  {
      synchronized (bindings) {
          Remote obj = bindings.get(name);
          if (obj == null) throw new NotBoundException(name);
          return obj;
      }
  }
  ...
```

Implantation des RMI sur sockets – côté client

Appel `x = proxy.foo(y)` ayant lieu sur une machine $S1$ avec `proxy` désignant un objet situé sur $S2$

1. Identifiant global d'un objet contenu dans le proxy = ⟨@ du site de l'objet, port, réf. locale⟩
2. Création d'un socket connecté à ⟨$S2$, port⟩
3. Envoi de l'id de l'objet et du nom de la méthode (`"foo"`)
4. Envoi des paramètres (cf 46) : sérialisable → écriture de la sérialisation de `y` / accessible à distance → informations de proxy pour `y`
5. Lecture du résultat : sérialisable → construit la valeur / accessible à distance → construit un proxy avec les informations fournies (⟨ @, port, id ⟩)
6. Fermeture de la connexion

(très simplifié !)

Implantation des RMI sur sockets – côté serveur

1. Au moins un objet accessible à distance → écoute sur un port arbitraire
   Identifiant global d'un objet = ⟨@, port, id local⟩
2. Acceptation d'une connexion sur ce port
3. Création d'une nouvelle activité (`Thread`) pour la suite
4. Lecture de l'id de l'objet → référence locale
5. Lecture du nom de la méthode et obtention (réification)
6. Lecture des paramètres (désérialisation / construction des proxys)
7. Appel de la méthode
8. Renvoi du résultat
9. Fermeture de la connexion

(très simplifié : connexion maintenue, plusieurs ports. . .)

Schéma de callback (rappel)

But : permettre au serveur d'appeler un client l'ayant contacté auparavant

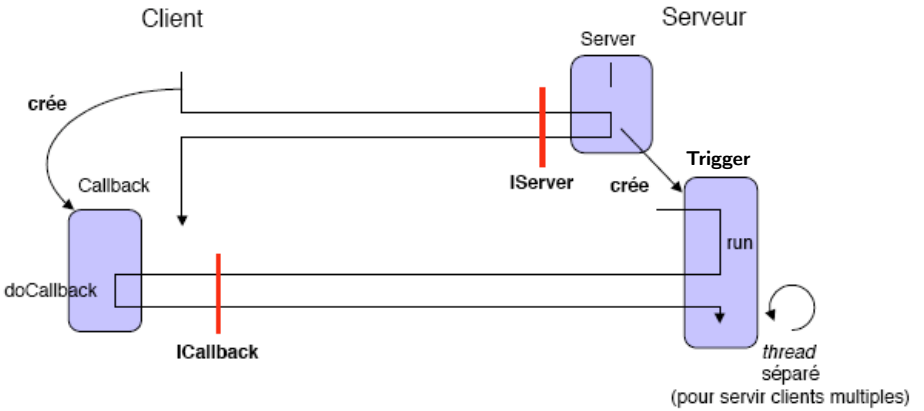
- Augmenter l'asynchronisme : schéma publier/s'abonner :
  1. appel client → serveur avec retour immédiat (s'abonner)
  2. rappel serveur → client quand le service est exécuté (publier)
- Augmenter les interactions : le serveur peut demander au client des données complémentaires
- Programmation événementielle

Principe

- Le client passe en paramètre au serveur l'objet à rappeler
- Le serveur exécute un appel sur cet objet

La relation client/serveur est conceptuelle, pas une relation d'usage !

Callback



Exemple 2 : callback

Interfaces

```
// les deux interfaces des objets appelés à distance

public interface ICallback extends Remote {
  public void wakeUp(String msg) throws RemoteException;
}

public interface IServer extends Remote {
 public void callMeBack(int time, String param,
                ICallback callback) throws RemoteException;
}
```

Exemple 2 : callback

Trigger

```
public class Trigger extends Thread {
  private int time;
  private String param;
  private ICallback callback;

  // le callback cb sera appelé avec param dans time seconds
  public Trigger(int time, String param, ICallback cb) {
     this.time = time; this.param = param; this.callback = cb;
  }

  public void run() { // exécution comme thread
     try {
         Thread.sleep(1000*time); // attend time secondes
         callback.wakeUp(param);
     } catch(Exception e) { e.printStackTrace(); }
  }
```

Exemple 2 : callback

Serveur

```
import java.rmi.*;
import java.rmi.server.*;

public class Server extends UnicastRemoteObject
                                implements IServer {
  public Server() throws RemoteException { }
  public void callMeBack(int time, String param,ICallback cb)
                                        throws RemoteException {
     Trigger action = new Trigger(time, param,cb);
     action.start();
  }

  public static void main(String[] args) throws Exception {
     Server server = new Server();
     Naming.rebind("ReveilMatin", server);
  }
}
```

Exemple 2 : callback

Callback & client

```
public class Callback extends UnicastRemoteObject
                                        implements ICallback {
  public Callback() throws RemoteException { }
  public void wakeUp(String message) throws RemoteException {
     System.out.println(message);
  }
}

public class Client {
  public static void main(String[] args) throws Exception {
     Callback callback = new Callback();
     IServer serv = (IServer) Naming.lookup("ReveilMatin");
     serv.callMeBack(5, "coucou", callback);
     ...
  }
}
```

RMI : Conclusion

- Génération automatique des stubs et skeletons
- Sérialisation automatique (en général) par simple référence à l'interface `Serializable`
- Serveur multi-thread
- Sémantique au plus une fois (at-most-once)
- Problème : ramassage des objets accessibles à distance
- Attention à la fiabilité : `RemoteException`

Quatrième partie

Intergiciels à message
Message Oriented Middleware

Plan

La communication asynchrone par message

Objectif : exploiter les possibilités d'une communication asynchrone

Avantages

- Évite le blocage de l'appelant inhérent à l'appel procédural
- Découple l'envoi de la réception
- Récepteur(s) anonyme(s)
- Étend le simple protocole point-à-point
- Autorise une communication de type publication/abonnement (publish/subscribe) : protocole $m \rightarrow n$

Difficulté

- Programmation délicate car asynchrone :
  Approche événementielle : Événement → Réaction

Exemple : supervision d'un réseau

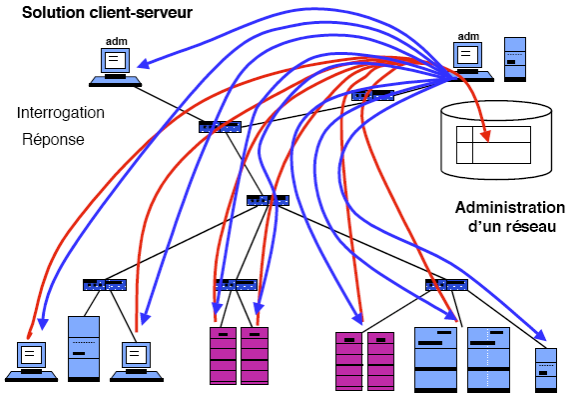
Contexte

- Surveillance de l'état de machines, systèmes et applications dans un environnement distribué
- Flot permanent de données en provenance de sources diverses sur le réseau
- Possibilité permanente d'évolution (ajout, suppression, déplacement des équipements)
- Possibilité d'accès des administrateurs depuis n'importe quel poste de travail

Objectifs

- suivi des changements de configuration dynamiques
- signaler les changements d'état et les mises à jour
- statistiques, journal de fonctionnement

Approche client-serveur



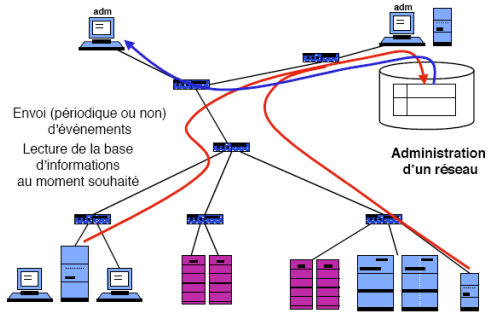
- Interactions synchrones
- Communication essentiellement 1 vers 1 (ou n vers 1)
- Entités (clients, serveurs) désignées explicitement
- Organisation de l'application plutôt statique

Approche client-serveur par inversion de contrôle

Client-serveur avec inversion du contrôle

- Le service d'administration s'abonne auprès des clients sur les événements qui l'intéressent
- Les clients contactent le service d'administration en cas de tel événement
- ⇒ mécanisme de « callback »
- Entités (clients, serveurs) désignées explicitement
- Organisation de l'application plutôt statique
- Découverte de nouveaux équipements ?

Approche MOM



- Les composants administrés émettent des messages :
  - changements d'état et de configuration, d'alertes
  - horloge (relevé périodique, statistiques)
- Des processus cycliques (démons) mettent à jour l'état du système à partir des notifications reçues
- Inversion des rôles des producteurs et des consommateurs de données

Interaction par messages

Modèle élémentaire

Send(message, destination)
Receive(message, source)

Synchrone/asynchrone

- Communication synchrone : rendez-vous entre émission et réception bloquantes
- Communication asynchrone :
  - Émission non bloquante
  - Réception bloquante, non déterministe

Paramètres

Désignation

- Communication point à point entre activités (canaux)
- Communication indirecte : passage par une boite à lettres
  - attachée à un processus ($n \rightarrow 1$: port, porte)
  - partagée par plusieurs processus ($n \rightarrow m$: file de messages)
- Statique (ex : IPv4) ou dynamique

Propriétés du service de communication

- Fiabilité (perte)
- Intégrité
- Qualité de service (débit, latence, gigue)
- Ordonnancement relatif des réceptions par rapport aux émissions (p.e. fifo)

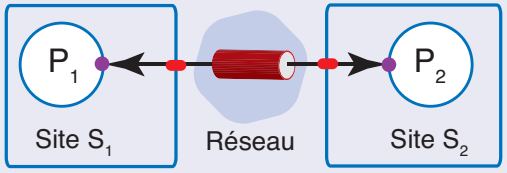
Communication asynchrone par messages

- Envoi asynchrone : l'émetteur ne se bloque pas
- Réception sélective à la demande du récepteur
- Diffusion possible d'un message à plusieurs récepteurs
- Couplage minimal entre émetteur et récepteur :
  - L'émetteur ne connaît pas le(s) récepteur(s) : il publie
  - Un récepteur doit explicitement souscrire pour recevoir
  - Un récepteur contrôle à quel moment il accepte
- Le récepteur peut ne pas être actif (présent) lorsque le message est envoyé

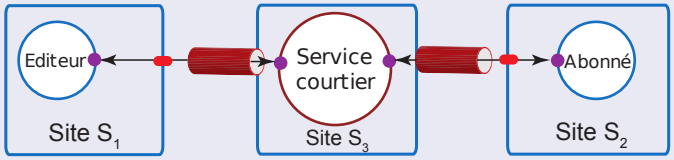
Usage : Architectures logicielles à composants

Communication indirecte

Communication directe



Communication indirecte



Queue vs Sujet (Topic)

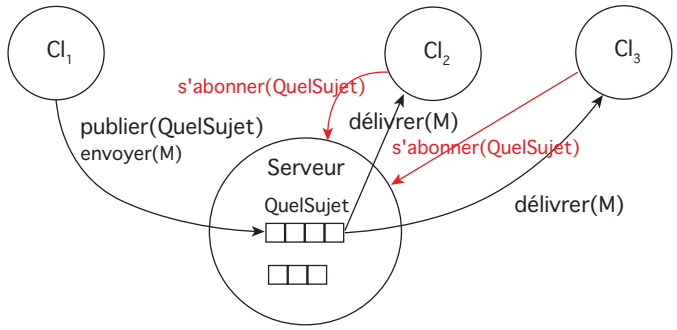
File de messages / Message Queue

- Interface : ajouter dans la file, retirer de la file
- Plusieurs producteurs, plusieurs consommateurs
- Retrait destructif
- Persistance
- Découplage temporel production / consommation

Publication – abonnement / Publish – subscribe

- Interface : envoyer un message, obtenir un message
- Plusieurs producteurs, plusieurs consommateurs
- Retrait non destructif, mais chaque consommateur n'obtient qu'au plus une fois chaque message
- Abonnement pour délivrance asynchrone

L'échange par publication/abonnement (publish/subscribe)



- Précédence temporelle : on ne peut recevoir que ce qui a été publié après s'être abonné (contrairement à une queue)

Modèle requête/réponse

Modèle client-serveur avec des messages :

- Une file de requêtes par serveur
- Une file de réponse par client
- Une requête identifie la file de réponse à utiliser
- Client :
  1. envoyer message de requête sur la file de requête
  2. attendre message de reponse sur sa file de réponse
- Serveur :
  1. attendre message de requête sur la file de requête
  2. traiter la requête
  3. envoyer message de reponse sur la file de réponse identifiée dans la requête

Domaine d'application : systèmes faiblement couplés

- Découplage temporel : interactions asynchrones / systèmes autonomes communicants
  - Communication en mode « push » : découverte des évolutions de l'environnement
  - Fonctionnement en mode déconnecté : site absent ou utilisateur mobile
- Découplage spatial : systèmes à grande échelle
  - Fonctionnement en mode partitionné / déconnecté
  - Communication « anonyme »
  - Communication n–m
- Découplage sémantique : systèmes hétérogènes
  Modèle d'interaction minimal → possibilité d'intégrer des environnements (systèmes, réseaux) / applications hétérogènes

Plan

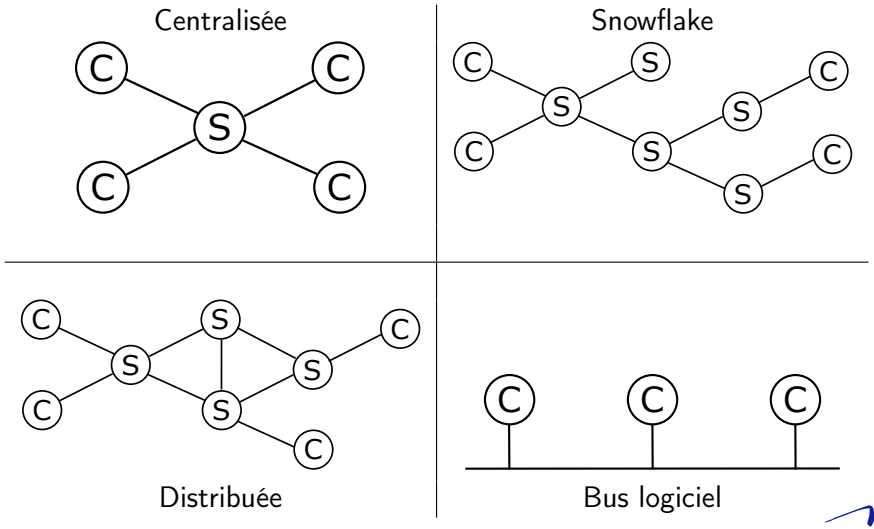
Les intergiciels à messages

MOM : Message-Oriented Middleware

Architecture et fonctionnalités de base

- Les clients (applicatifs) s'adressent à un serveur
- Ils envoient/recoivent leurs messages au(x) serveur(s)
- Un service courtier (broker ou provider) sert d'intermédiaire pour stocker et router les messages vers leurs destinataires
- Le protocole d'échange peut être de type publier/s'abonner (publish/subscribe)
- Critère de réception par le contenu, par le sujet (topic)
- Gestion de la persistance des messages
- Communication point-à-point possible

Architectures du service courtier



Intergiciel à messages (suite)

Les éléments d'un intergiciel à message

- Service de gestion des messages implanté par un ou plusieurs serveurs (providers)
- Une API client
- Les objets messages pour communiquer

La notion de message

- l'en-tête ou header qui contient les informations de gestion :
  - file destinataire, identifiant du message
  - dates de disponibilité, d'échéance, ...
- les propriétés : suite de couples (clé,valeur) précisant le contenu du message
- les données applicatives (charge utile ou payload)

Intergiciel à messages (suite)

Fonctionnalités complémentaires

- Définition de priorités entre messages
- Compression des données utiles du messsage
- Échéance maximale pour recevoir un message
- Publication à date minimale fixée
- Routage des messages d'un serveur à l'autre (forward)
- Lancement d'applications lorsque des messages sont disponibles pour elles
- Possibilité d'alertes sur critères :
  - Présence de messages dans une file donnée
  - Nombre de messages présents

La standardisation et les produits MOM

- API standard pour Java : JMS (Java Message Service)
- Quelques MOM réalisant l'API JMS :
  - Open Message Queue (intégré dans GlassFish, implantation de référence)
  - ActiveMQ de Apache
  - Joram de l'INRIA (intégré dans Jonas)
  - OpenJMS
- Autres standards :
  - HLA (High-Level Architecture) pour interconnecter des simulateurs
  - XMPP (eXtensible Messaging and Presence Protocol)
  - AMQP (Advanced Message Queuing Protocol)
  - Microsoft Message Queuing (MSMQ)

Apache Kafka

A distributed streaming platform

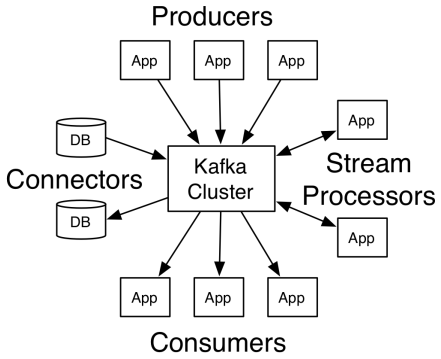
- Distribution, stockage et traitement de flux de données
- Points forts : haut débit, latence faible, tolérant aux fautes → traitement de gros flux de données temps réel

Les données

- Un flux est identifié par un sujet (*topic*)
- Un flux est constitué d'enregistrements (*record*)
- Un enregistrement contient une clef, une valeur et une date (*timestamp*)

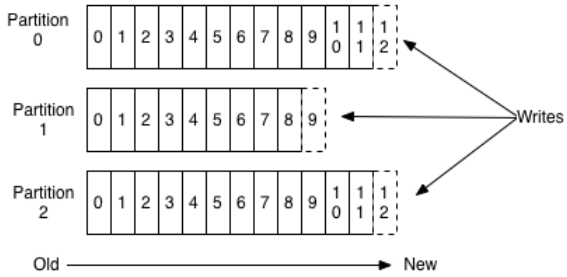
Les APIs

- Producer : publication d'enregistrements
- Consumer : abonnement à des topics et obtention d'enregistrements
- Streams : transformation de flux, un ou des flux en entrée, un ou des flux en sortie
- Connector : vision flux de BD



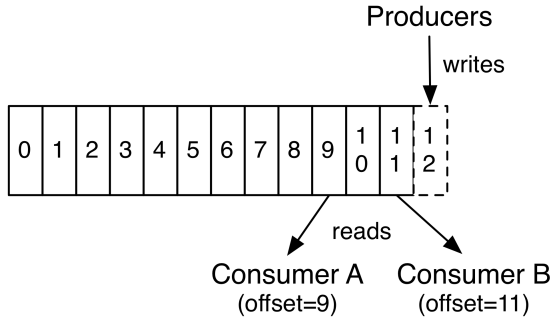
source : https://kafka.apache.org

Topic et partitions



- Un topic est multi-producteurs, multi-consommateurs
- Un topic est décomposé en partitions → parallélisation
- Chaque partition contient une séquence ordonnée et immuable d'enregistrements, strictement croissante
- Immuable : pas d'effacement à la consommation (mais possibilité de délai de rétention avec oubli)

Consommation

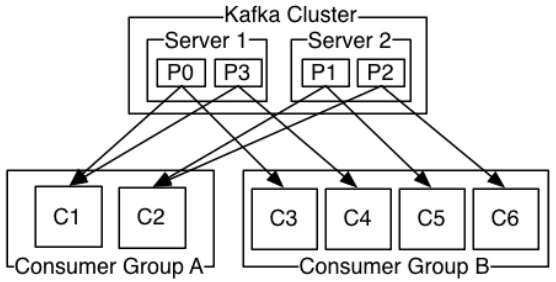


- Les producteurs d'une partition ajoutent en queue
- Chaque consommateur a un offset, qu'il peut changer arbitrairement (pour retourner en arrière ou pour sauter aux plus récents enregistrements)

Distribution & tolérance aux fautes

- Un *Kafka cluster* est un ensemble de serveurs sur un ensemble de machines
- Chaque partition est répliquée sur plusieurs serveurs
- Chaque partition possède un serveur leader et des serveurs de secours
- Le leader gère toutes les lectures et écritures, les secours suivent passivement
- Un algorithme d'élection choisit un nouveau leader en cas de défaillance (cf cours *Systèmes répartis*)
- En pratique, un serveur leader d'une partition est secours pour d'autres partitions du même ou d'autres topics

Architecture globale



- Un producteur publie dans une partition d'un topic
- Les consommateurs forment des groupes.
- Un groupe reçoit depuis toutes les partitions, mais un enregistrement n'est délivré qu'une fois par groupe (à un seul consommateur)
  → partage de charge

Traitement de flux

- Séparation (*branch*) : scission d'un flux en plusieurs flux
- Filtrage : flux limité aux enregistrements vérifiant un prédicat
- *map* : applique une transformation à chaque enregistrement
- *flatmap* : applique une transformation à chaque enregistrement, pour produire 0, 1 ou plusieurs enregistrements
- Agrégation des enregistrements ayant la même clef en un unique enregistrement
- Comptage des enregistrements d'une même clef
- Jointure de plusieurs flux

```
builder.stream("visiteurs-topic")
    .filter((nom, date) -> date >= today - 7)
    .map((nom, date) -> KeyValue.pair(nom.toLowerCase(),date))
    .groupByKey()
    .count()
    .filter((nom, count) -> count > 100)
    .to("clients-fidèles-topic");
```

Plusieurs points de vue

Kafka est ...

- un intergiciel à messages, unifiant queue et abonnement, avec objectif de performance (de Kafka et des applications l'utilisant) → parallélisation
- un système de stockage d'informations incrémentales, fiable (réplication) et performant
- un système de traitement de flux

Plan

Le standard JMS (Java Message Service)

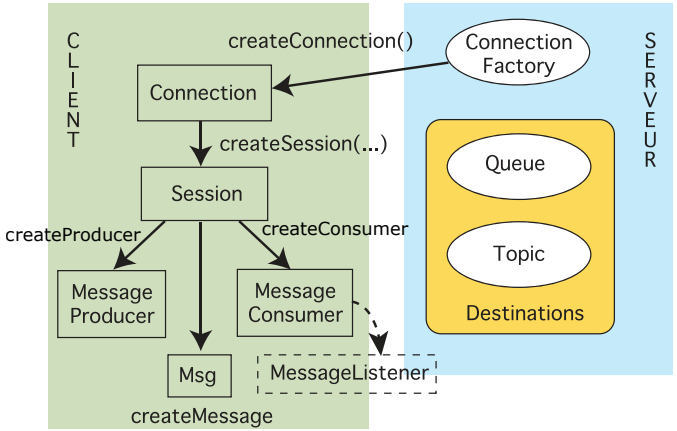
Les objets globaux (administrés) accessibles à distance

- Désignation par JNDI (Java Naming and Directory Interface)
- Les fabriques de connexions (connection factories)
- Les destinations réparties en deux domaines de désignation : files (queues) et sujets (topics)
- Ces objets sont créés dans le(s) serveur(s) courtier(s) implantant JMS

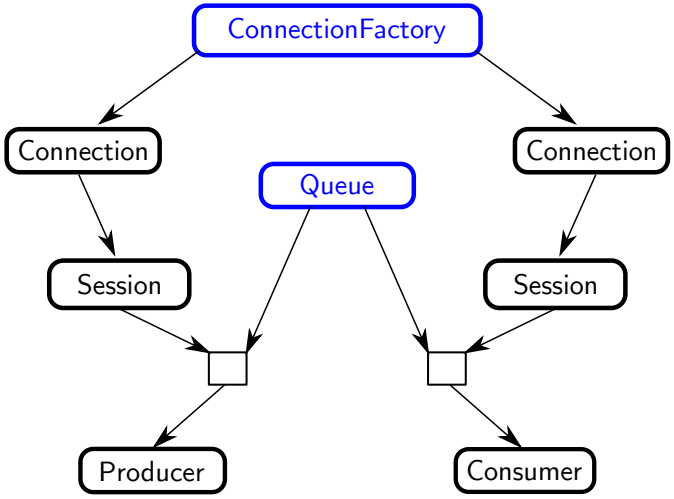
Les objets clients

- Les connexions permettent de se connecter à un serveur JMS
- Les sessions gèrent les échanges via une file ou un sujet
- Les producteurs/consommateurs de messages pour l'envoi/la réception de messages dans le cadre d'une session

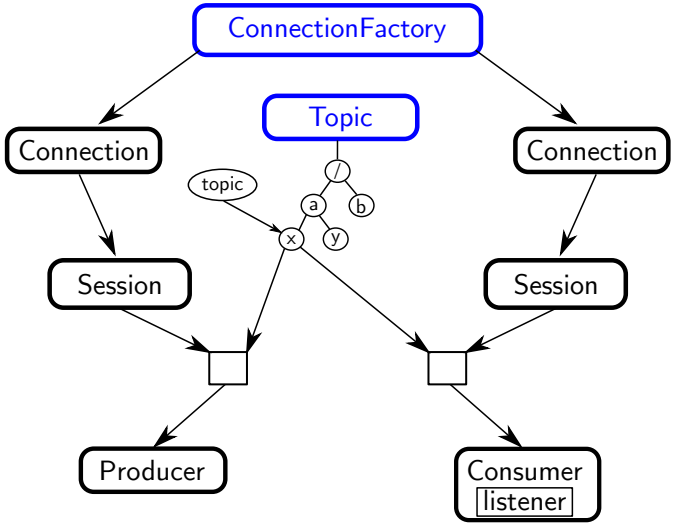
Les objets de communication



Les objets – Queue



Les objets – Topic



Un exemple de publication/abonnement

Le client éditeur : nom de la destination fourni dans args[0]

```
import javax.jms.* ;   import javax.naming.* ;
public class Editeur {
 public static void main(String[] args) {
 try {
  InitialContext jndiCtx = new InitialContext();
  ConnectionFactory scf = (C...) jndiCtx.lookup("MaConnFactory")
  Destination dest = (Destination) jndiCtx.lookup(args[0]);
  Connection conn = scf.createConnection();
  conn.start();
  Session session =
       conn.createSession(false,Session.AUTO_ACKNOWLEDGE);
  MessageProducer editeur = session.createProducer(dest);
  TextMessage m = session.createTextMessage();
  m.setText("publication exemple");
  editeur.send(m); ...
}
```


Un exemple de publication/abonnement

Un client abonné : nom de la source fourni dans args[0]

```
import javax.jms.* ;   import javax.naming.* ;
public class Abonne {
 public static void main(String[] args) {
  try {
  InitialContext jndiCtx = new InitialContext();
  ConnectionFactory scf = (C...) jndiCtx.lookup("MaConnFactory")
  Destination src = (Destination) jndiCtx.lookup(args[0]);
  Connection conn = scf.createConnection();
  Session session =
       conn.createSession(false,Session.AUTO_ACKNOWLEDGE);
  MessageConsumer abonné = session.createConsumer(src);
  abonné.setMessageListener(new MonMsgListener());
  conn.start();
  ...
```

Un exemple de publication/abonnement

Le traitant activé sur occurrence d'un message

```
import javax.jms.* ;
public class MonMsgListener implements MessageListener {
  public void onMessage(Message m) {
   try {
    if (m instanceof TextMessage) {
     TextMessage msg = (TextMessage) m ;
     System.out.println(msg.getText()) ;
     }
   } catch (JMSException je) {...}
  }
}
```

Un exemple de publication/abonnement

La création des objets globaux

```
import org.objectweb.joram.client.jms.tcp.TcpConnectionFactory;
public class CreateDestination {
  public static void main(String args[]) throws Exception {
    // Creating the JMS administered objects:
    javax.jms.ConnectionFactory connFactory
             = TcpConnectionFactory.create("localhost", 16010);

    Destination destination = Topic.create(0);
    // Destination destination = Queue.create(0);

    // Binding objects in JNDI
    javax.naming.Context jndiCtx = new InitialContext();
    jndiCtx.bind("MaConnFactory", connFactory);
    jndiCtx.bind("MonTopic", destination);
    }
}
```

Conclusion

- Mode de communication adaptée à la circulation des informations dans le système informatique des entreprises
- Intégration dans des intergiciels à base de composants
- Mécanisme de base dans les bus de services

Conclusion

Communication à distance

- interconnexion d'applications
- Abstraire les protocoles de communication

⇒ Intergiciel : service implantant un modèle d'interaction entre processus / logiciel

Modèle / mécanisme

1. Intergiciel de base : communication par socket
2. Intergiciel client-serveur : appel de procédure et appel de méthode à distance
3. Intergiciel asynchrone : communication par messages