Informatique Repartie

Chapitre 1 : Bases théoriques

Cecilia Zanni-Merk

cecilia.zanni-merk@insa-rouen.fr

Bureau BO B R1 04

Basé sur le cours de M Alexandre Pauchet, INSA Rouen Normandie, 2016

Références

- Architectures réparties en Java de Annick Fron
  ISBN 9782100738700. Ed Dunod
- Le cours « Développement d'applications reparties » de Amen Ben Hadj Ali, ISI-L3SIL, 2011-2012
  http://isila3sil.weebly.com/uploads/1/5/0/3/15031016/dar-ch1.pdf

Plan

- Introduction
- Architecture client/serveur
- Appel de procédures à distance

Introduction

Système Réparti : définitions

- Un ensemble d'ordinateurs indépendants qui apparaît à un utilisateur comme un système unique et cohérent
  - Les machines sont autonomes
  - Les utilisateurs ont l'impression d'utiliser un seul système (notion de transparence)
- Un **système reparti** (distribué) est un **ensemble d'entités autonomes de calcul** (ordinateurs, processeurs, processus, etc.) interconnectées et qui peuvent **communiquer par un réseau de communication**, par moyen d'envoi de messages ou d'autres moyens

Pourquoi les Systèmes Repartis ?

- Aspects économiques (rapport prix/performance)
- Adaptation de la structure d'un système à celle des applications (géographique ou fonctionnelle)
- Besoin d'intégration (applications existantes)
- Besoin de communication et de partage d'information
- Réalisation de systèmes à haute disponibilité
- Partage de ressources (programmes, données, services)
- Réalisation de systèmes à grande capacité d'évolution

Application repartie

- Une application informatique est un programme exécutable sur une machine ou plusieurs machines qui représente la logique de traitement des données manipulées
  - Elle s'exécute en mémoire vive au dessus d'un système d'exploitation
  - Avant exécution, elle est stockée sur un support persistant
  - L'application émet un résultat sou forme de données soit affichées, soit enregistrées sur un disque
- Un grand nombre d'applications ne s'exécutent pas intégralement sur un seul nœud de calcul. Il s'agit **d'applications reparties** (distribuées)

Application repartie = traitements coopérants sur des données reparties

Système vs application

- **Système :** gestion des ressources communes et de l'infrastructure, lié de manière étroite au matériel sous-jacent

- **Application :** réponse à un problème spécifique, fourniture de services aux utilisateurs (qui peuvent être d'autres applications).

- Une application utilise les services généraux fournis par le système

Services et Interfaces

- Un service est un comportement **défini par contrat**, qui peut être implémenté et fourni par un composant pour être utilisé par un autre composant, sur la base exclusive du contrat

- Un service est accessible via une ou plusieurs interfaces
- Un interface décrit l'interaction entre le demandeur et le fournisseur du service

Interfaces

- La fourniture d’un service met en jeu deux interfaces
  - Interface requise (côté client)
  - Interface fournie (côté serveur)
- Le contrat spécifie la conformité entre ces interfaces
  - Notion d’encapsulation
- Concernant l’opérationnalisation
  - Interface Definition Language (IDL)
- Concernant le contrat
  - Sur la forme
  - Sur le comportement
  - Sur les interactions entre méthodes
  - Sur les aspects non fonctionnels (performance, par exemple)

Services répartis

- Services de désignation
- Services de communication
- Services de fichiers répartis
- Services d’exécution répartie
- Services de transaction réparties
- Services de temps réparti
- Services de mémoire répartie

Exemple : Agence de voyage

- Un produit « voyage » est la combinaison de plusieurs produits
  - Gestion de réservation de billets de transport
  - Gestion de réservation des hôtels
  - Gestion de réservation de voitures de location
  - …
- Le résultat des informations récupérées auprès de différents fournisseurs
  - Compagnies aériennes
  - Chaînes hôtelières
  - Agences de location de voitures
  - …

Agence de Voyage
(plateforme technique 1)

Application Web

Compagnies aériennes
(plateforme technique 2)

Réservations

Tarifs

Promotions

Compagnies d’assurance
(plateforme technique 5)

Compagnies de location de voitures
(plateforme technique 3)

Chaîne hôtelière
(plateforme technique 4)

Domaines d’application

- Coordination d’activités
  - Systèmes de workflow
  - Systèmes à agents
- Communication et partage d’information
  - Collecticiels : pour le travail coopératif, les bibliothèques, musées et magasins virtuels sur Internet, la presse et le commerce électronique, etc.
    - Édition coopérative.
    - Téléconférence.
    - Ingénierie concourante.
- Applications temps réel
  - Contrôle de procédés industriels
  - Localisation de mobiles

Toutes les applications qui nécessitent des utilisateurs ou des données réparties

Programmation “classique” *vs.* Informatique Répartie. Quelques constats

- Applications Client-Serveur
  - La plupart des applications réseaux sont du type Client-Serveur
  - Dans ce cadre, le client appelle un “service” auprès d’un serveur
  - La dénomination architectures orientées services existe par ailleurs ; elle désigne principalement les Services Web
- Exemples
  - Un navigateur web demande une page html à un serveur
  - Un client FTP demande la liste des fichiers et des répertoires contenus dans un répertoire
  - etc.

Programmation “classique” *vs.* Informatique Répartie. Objectifs

- Programmation “classique”
  - En programmation classique lorsque qu’un programme a besoin d’un service, il appelle une fonction d’une librairie, une méthode d’un objet, *etc.*

- Informatique Répartie
  - Proposer des méthodes et outils pour simplifier le développement d’applications réseau Client-Serveur, en essayant de s’abstraire de l’aspect “distant” : proposer une programmation “naturelle”
  - Pour les applications “lourdes” :
    - Décomposer les applications en ensembles de services
    - Rationaliser la répartition des services pour limiter les échanges d’informations

Programmation “classique” *vs.* Informatique Répartie

- Programmation “classique”
  - Une seule machine
  - Même OS
  - Même espace mémoire
  - Pas de problème de transport
  - Disponibilité du service assuré (tant que l’on a accès à la librairie)
- Informatique Répartie
  - Deux machines (sans compter celles “traversées”)
  - OS différents
  - Représentations différentes des types de bases
  - Espace mémoire : “passer un pointeur/référence comme argument” ? Problème de transport : firewall, réseau HS, *etc.*
  - Retrouver le service ? Où se trouve-t-il ? Qui le propose ?

Programmation “classique” *vs.* Informatique Répartie. Interopérabilité des langages

- Un même langage
  - Même paradigme de programmation
  - Même représentation des types de base
  - Même représentation de l’information composite
- Deux langages
  - Représentation des types de base et de l’information composite pouvant être différente
  - Association des paramètres effectifs aux paramètres formels ? comment gérer les différents types de passage de paramètre ? Paradigmes de programmation différents : qu’est-ce que c’est un objet pour un langage procédural ? Comment gérer les erreurs ?

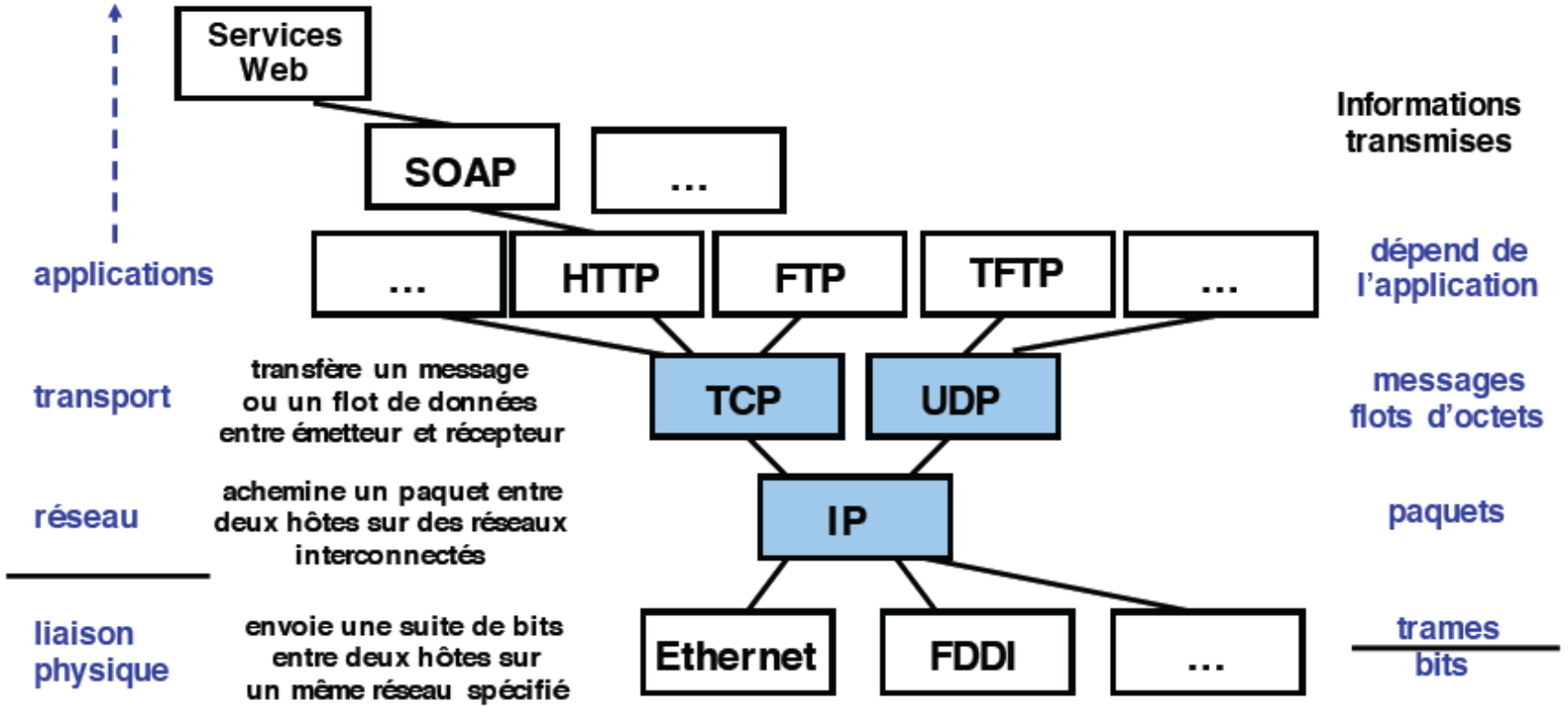
Programmation “classique” *vs.* Informatique Répartie. Solutions

- Séparer la spécification/conception de l’implantation
  - Utilisation d’un langage propre à la spécification/conception
  - Utilisation de “traducteurs” vers le langage cible en distinguant le client du serveur
- Utiliser un langage de représentation de l’information
  - Langage de représentation indépendant du langage de programmation
  - Pour chaque langage de programmation, définir un ensemble d’opérations pour “sérialiser” ces types (prédéfinis ou utilisateur)

Programmation “classique” *vs.* Informatique Répartie. Solutions

- Utiliser un protocole de transport
  - Comment spécifier le service demandé
  - Comment associer les paramètres effectifs aux paramètres formels
  - Comment transmettre les erreurs
- Définir la gestion du service
  - Utiliser un mécanisme permettant d’identifier la librairie (au sens large) qui fournit le service
  - Utiliser un mécanisme qui permet d’activer le service si besoin

Positionnement



HTTP : *HyperText Transfer Protocol* : protocole du Web
TFTP, FTP : (*Trivial*) *File Transfer Protocol*) : transfert de fichiers
TCP : *Transmission Control Protocol* : transport en mode connecté
UDP : *User Datagram Protocol* : transport en mode non connecté
IP : *Internet Protocol* : Interconnexion de réseaux, routage

Terminologie

- **Interlogiciel ou middleware** : un logiciel qui s'insère entre deux applications pour permettre la communication des machines entre elles, indépendamment de la nature du processus, du système d'exploitation, du langage.
- **Répartition vs parallélisme :**
  - Le parallélisme implique des tâches nombreuses mais répétitives. Le problème est de paralléliser les tâches ou le données. *Les architectures parallèles exploitent des régularités de traitement ou de données* (ceci limite le champ d'application)
  - Une architecture répartie implique des données et des tâches différentes sur les machines. Les applications réparties ne partagent pas de mémoire. Le problème est la transmission des données entre tâches et la synchronisation entre celles-ci

Terminologie

- **Grain des applications :** permet d'évaluer la taille des éléments mis en présence. Un ERP → « gros grain », un objet Java communiquant à travers le réseau avec d'autres objets indiviuels → « grain fin »
  - Les services web , plutôt adaptés aux applications « gros grain »
  - Architectures RPC, plutôt adaptées aux applications « grain fin »
- **Couplage des applications:**
  - Lâche: les 2 applis échangent peu de données, ou peu souvent, ou peu de types de messages … → composants gros grain
  - Etroit : besoin de plus de synchronisation et d'harmonisation des données → composants grain fin
  - L'analyse du couplage est la première étape de la conception d'une architecture répartie (diagrammes UML de séquence ou d'interaction)

Terminologie

- **Communication entre applications** : Deux grande familles
  - **Les technologies d'appel à procédure à distance** (RPC, remote procedure call), regroupant des standards tels que CORBA, RMI, SOAP …
    Le principe réside sur l'invocation d'un service (une procédure ou une méthode d'un objet) situé sur une machine distance indépendamment de a localisation et de son implémentation.
    Le concept est le même que l'appel de sous-procédure, avec un déroutement du contexte d'appel et une attente bloquant des résultats
  - **Les technologies d'échange de messages** : échange véhiculé par l'infrastructure MOM (message oriented middleware) Exemple, JMS

Terminologie

- Communication et qualité de service
  - Lors que l'on appelle une fonction sur un réseau, il peut arriver que la communication effectue l'appel zéro, une fois ou plusieurs fois (si on a implémenté un mécanisme de renvoi automatique sur levée d'exception)
  - Assurer que l'opération a été effectué une fois implique une qualité de service qui repose sur nos choix de conception → compromis entre « qualité de service » et « performance »
    - Transfert bancaire vs affichage d'une page web

Impact sur le génie logiciel

- La programmation répartie ajoute des degrés de complexité au génie logiciel
  - Réflexion nécessaire sur la topologie de l'application et sur les interfaces entre systèmes
  - Pour des applications ouvertes (type SOAP ou REST], la prise en compte de connecteurs externes à identifier
  - La prise en compte d'environnements non homogènes avec des langages différents
  - La qualité de service de la communication
  - Des problématiques non fonctionnelles supplémentaires : sécurité et identification, répartition de charge, reprise de panne, persistance des données, reprise sur erreur …

Impact sur le génie logiciel

- **Modification de la conception**
  - Au-delà d'identifier les classes et leurs opérations, il faut identifier la **synchronisation** de tous ces objets sur le réseau
  - Diagramme de **déploiement** !!

Impact sur le génie logiciel

- **Modification du processus de développement**
  - Plusieurs nouvelles bibliothèques et API à utiliser
  - La compilation n’est plus la seule étape
  - Le déploiement = installation, configuration et paramétrage d’une application
  - Debugging
    - Test de temps de réponse du middleware
    - Tests de charge
    - Tests de synchronisation
    - Tests d’accès concurrent

Architectures client/serveur

Architecture Client/Serveur

- L'environnement client/serveur désigne un mode de communication à travers un réseau entre plusieurs programmes ou logiciels : l'un, qualifié de **client**, envoie des requêtes ; l'autre ou les autres, qualifiés de **serveurs**, attendent les requêtes des clients et y répondent.
- Par extension, le client désigne également l'ordinateur sur lequel est exécuté le logiciel client, et le serveur, l'ordinateur sur lequel est exécuté le logiciel serveur.

Architecture Client/Serveur

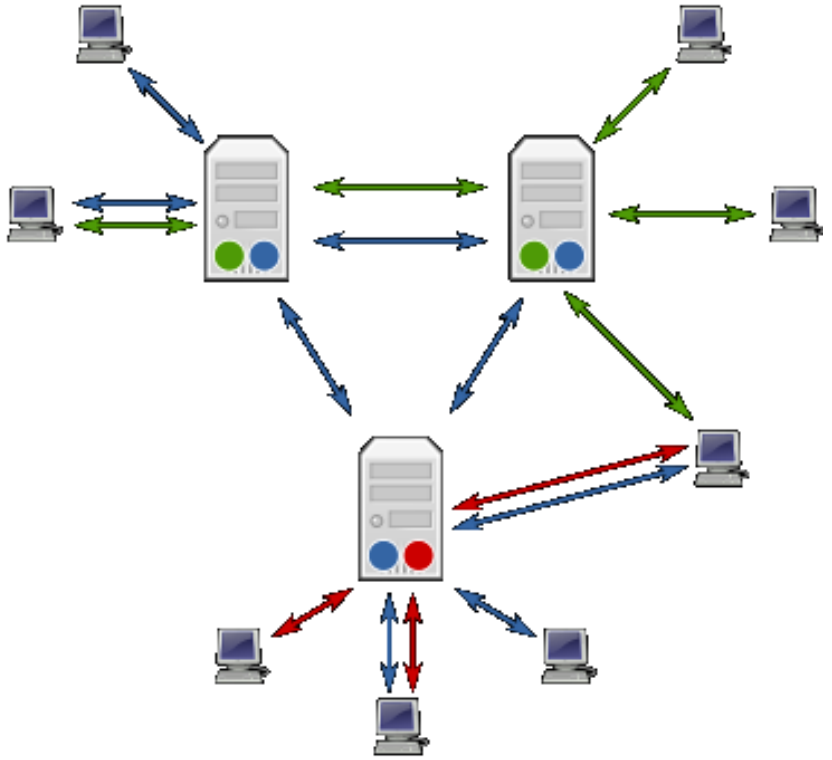
- Il existe une grande variété de logiciels serveurs et de logiciels clients en fonction des besoins à servir : un **serveur web** publie des pages web demandées par des **navigateurs web** ; un **serveur de messagerie électronique** envoie des mails à des **clients de messagerie** ; un **serveur de fichiers** permet de stocker et consulter des fichiers sur le réseau, un **serveur de données** à communiquer des données stockées dans une base de données, etc.

Caractéristiques d'un processus serveur

- Il attend une connexion entrante sur un ou plusieurs ports réseaux ;
- A la connexion d'un client sur le port en écoute, il ouvre un socket local au système d'exploitation;
- Suite à la connexion, le processus serveur communique avec le client suivant le protocole prévu par la couche application

Caractéristiques d'un processus client

- il établit la connexion au serveur à destination d'un ou plusieurs ports réseaux;
- lorsque la connexion est acceptée par le serveur, il communique comme le prévoit la couche application.

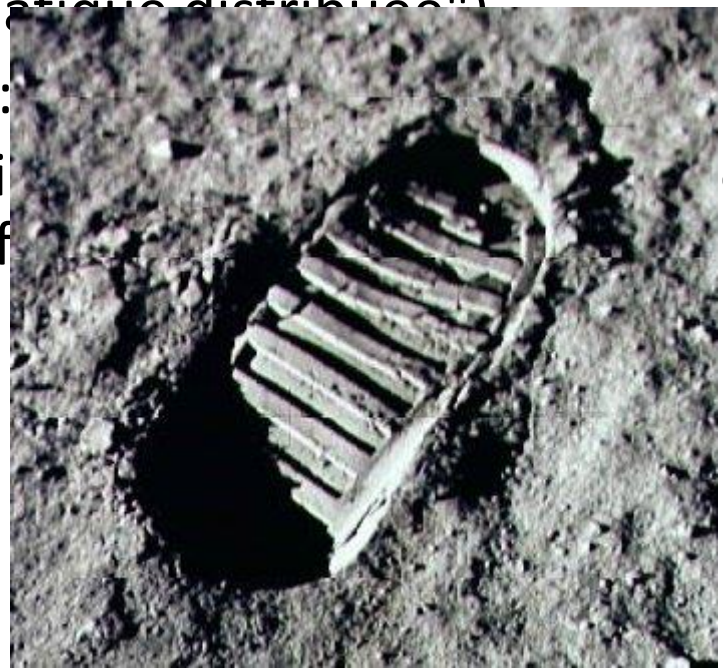


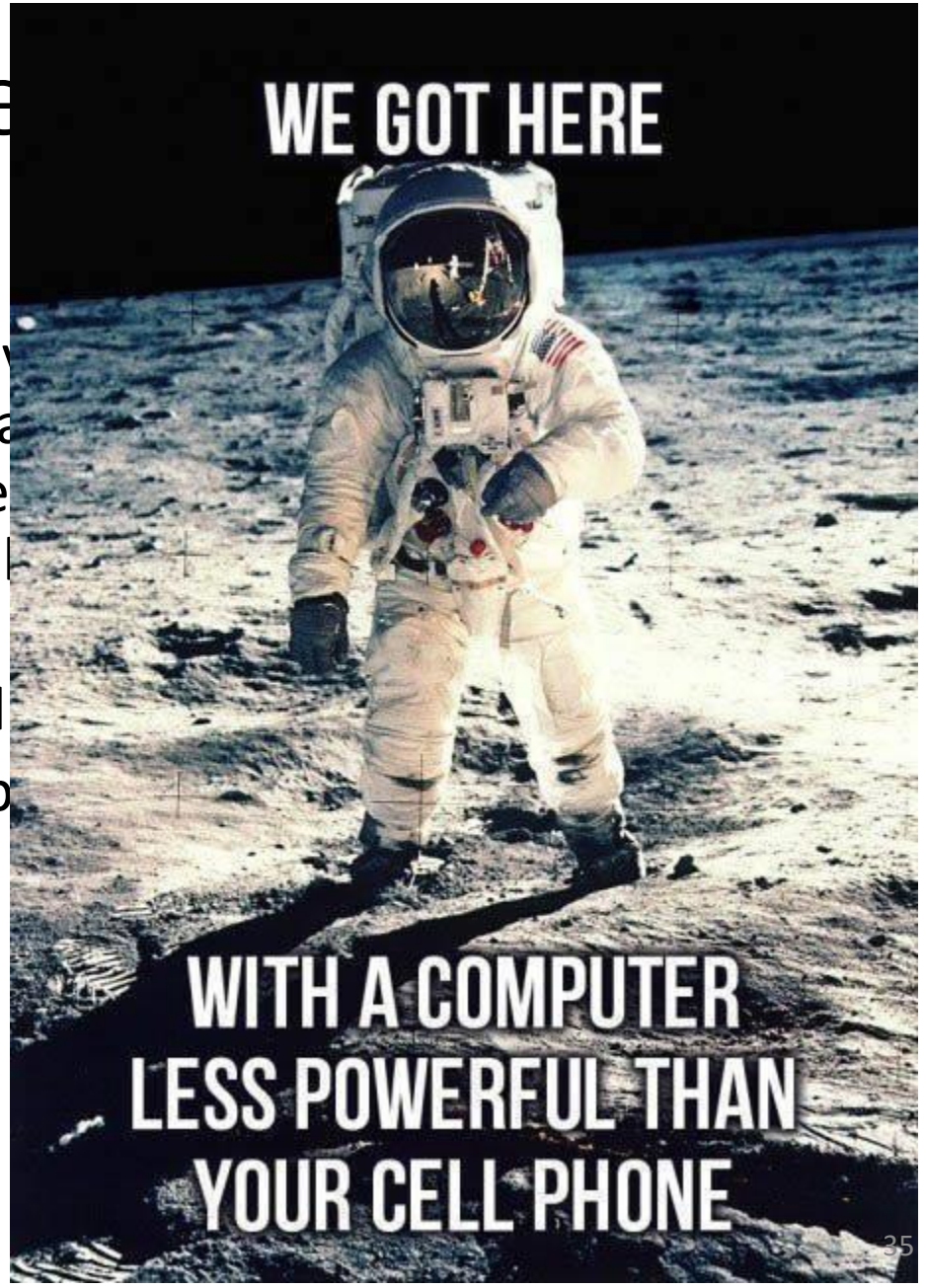
Caractéristiques générales

- Le client et le serveur doivent bien sûr utiliser le même protocole de communication au niveau de la couche transport.
- Un serveur est généralement capable de servir plusieurs clients simultanément. On parle souvent d'**un service** pour désigner la fonctionnalité offerte par un processus serveur.
- On définit aussi comme serveur, un ordinateur spécialisé ou une machine virtuelle n'ayant que pour unique tâche l'exécution d'un ou plusieurs processus serveur.

Les architectures réseaux e

- Architecture « Mainframe »
  - Avant que n'apparaisse le mode client-serv
    anciens sont configurés autour d'un ordina
    appelé "Mainframe" qui gère toutes les se
    l'ensemble des terminaux-utilisateurs qui
    "d'informatique distribuée")
  - Avantage: tel d
  - Inconvéni e rep
    du "mainf





Les architectures réseaux existantes

- Architecture « Peer to Peer »
  - Le réseau est dit « pair à pair » (peer-to-peer en anglais, ou P2P), lorsque chaque ordinateur connecté au réseau est susceptible de jouer tour à tour le rôle de client et celui de serveur.
- Architecture à 2 niveaux
  - Ce type d'architecture (2-tier en anglais) caractérise les systèmes client-serveur où le poste client demande une ressource au serveur qui la fournit à partir de ses propres ressources .

Les architectures réseaux existantes

- Architecture à 3 niveaux
  - Dans cette architecture (3-tier en anglais), existe un niveau supplémentaire:
    - Un client (l'ordinateur demandeur de ressources) équipé d'une interface utilisateur (généralement un navigateur web) chargé de la présentation.
    - Un serveur d'application (appelé middleware) qui fournit la ressource , mais en faisant appel à un autre serveur.
    - Un serveur de données qui fournit au serveur d'application les données requises pour répondre au client.
- Architecture à N niveaux

Les types de clients

- Client « léger »
  - Le poste client accède à une application située sur un ordinateur dit "serveur" via une interface et un navigateur Web. L'application fonctionne entièrement sur le serveur, le poste client reçoit la réponse "toute faite" à sa demande qu'il a formulée (appelée : "requête") .
- Client « lourd »
  - Un client lourd est un logiciel qui propose des fonctionnalités complexes avec un traitement autonome. Et contrairement au client léger, le client lourd ne dépend du serveur que pour l'échange des données dont il prend généralement en charge l'intégralité du traitement.

Les types de clients

- Client « riche »
  - *Une interface graphique plus évolué permet de mettre en œuvre des fonctionnalités comparables* à celles d'un client "lourd". Les traitements sont effectués majoritairement sur le serveur , la réponse "semi-finie" étant envoyée au poste client, où le client "riche" est capable de la finaliser et de la présenter.

Avantages

- Toutes les données sont centralisées sur un seul serveur, ce qui simplifie les contrôles de sécurité et la mise à jour des données et des logiciels.
- Une administration au niveau serveur, les clients ayant peu d'importance dans ce modèle, ils ont moins besoin d'être administrés
- Toute la complexité/puissance peut être déportée sur le serveur(s), les utilisateurs utilisant simplement un client léger sur un ordinateur terminal qui peut être simplifié au maximum.
- Recherche d'information : Les serveurs étant centralisés, cette architecture est particulièrement adaptée et véloce pour retrouver et comparer de vaste quantité d'information (Moteur de Recherche sur le Web)

Inconvénients

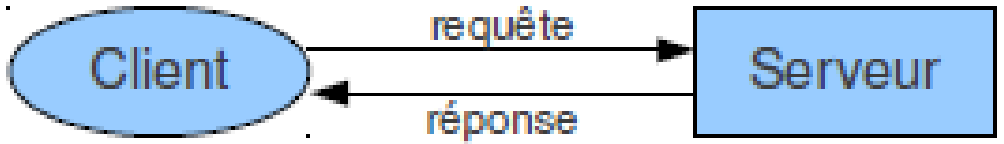
- Si trop de clients veulent communiquer avec le serveur au même moment, ce dernier risque de ne pas supporter la charge (alors que les réseaux pair à pair fonctionnent mieux en ajoutant de nouveaux participants).
- Si le serveur n'est plus disponible, plus aucun des clients ne fonctionne (le réseau pair à pair continue à fonctionner, même si plusieurs participants quittent le réseau).
- Les coûts de mise en place et de maintenance sont élevés.
- En aucun cas les clients ne peuvent communiquer entre eux, entrainant une asymétrie de l'information au profit des serveurs.

Définitions

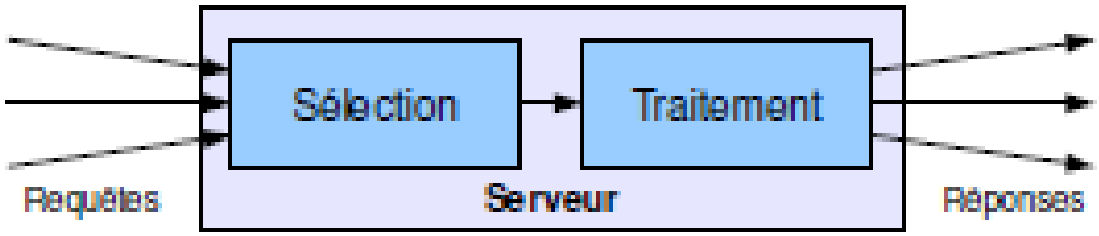
- Application Client-Serveur
  - Application faisant appel à des services distants au travers d'un échange de messages (les requêtes et les réponses) plutôt que par un partage de données (mémoire ou fichiers)
- Serveur
  - Programme offrant un service sur un réseau (par extension, machine offrant un service)
- Client
  - Programme qui émet des requêtes (ou demandes de service). Il est toujours l'initiateur du dialogue

Vues du client et du serveur

- Vue du client



- Vue du serveur



- **Remarques**
  - 2 messages échangés au minimum (requête+réponse)
  - Toute application répartie peut se décomposer en ensemble de requêtes de type Client-Serveur

Mise en œuvre

- Différents types de client-serveur
  - de données (ou procédural/fonctionnel)
  - à objets
  - à composants
- Niveau de description
  - Bas niveau : socket ; orienté objets : RMI ; orienté services : Web Services ; orienté ressources : REST ; orienté composants : J2EE
  - Langages de description d'interface : RPCL (RPC Sun), Corba XDR, Java RMI
  - Intégration dans un langage de programmation

Protocoles de communication

- Un **protocole de communication** formalise les messages (types, contenus et ordre) échangés par les entités d'un système réparti.
  - Ils sont souvent décrits en UML par des diagrammes de séquence.
  - Ils sont une abstraction des protocoles de transport.

- Exemple : calculatrice sur entiers positifs
  - Requête : C, request(opération, entier, opération) → S
  - Réponse : S, response(entier) → C
  - Réponse : S, error(description) → C

Formalisation des messages : de la conception à l'implémentation

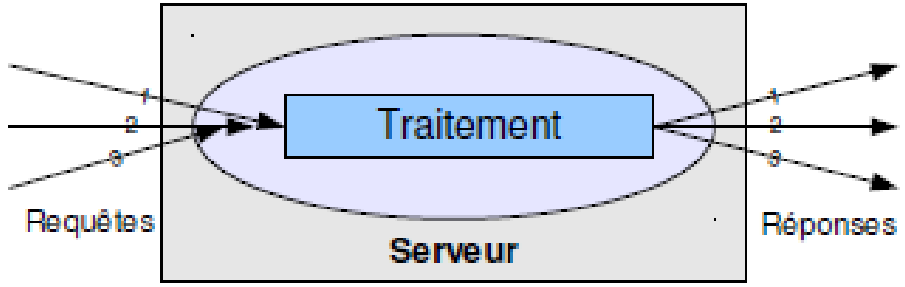
- Selon l'hétérogénéité entre le client et le serveur et le type d'architecture, certains éléments de conception peuvent être modifiés :
  - Contenus des messages,
  - Typage des messages
  - Exceptions

- Exemple
  - Un objet peut être sérialisé par l'ensemble de ses attributs
  - Une valeur en retour peut correspondre à une exception

Conception du serveur

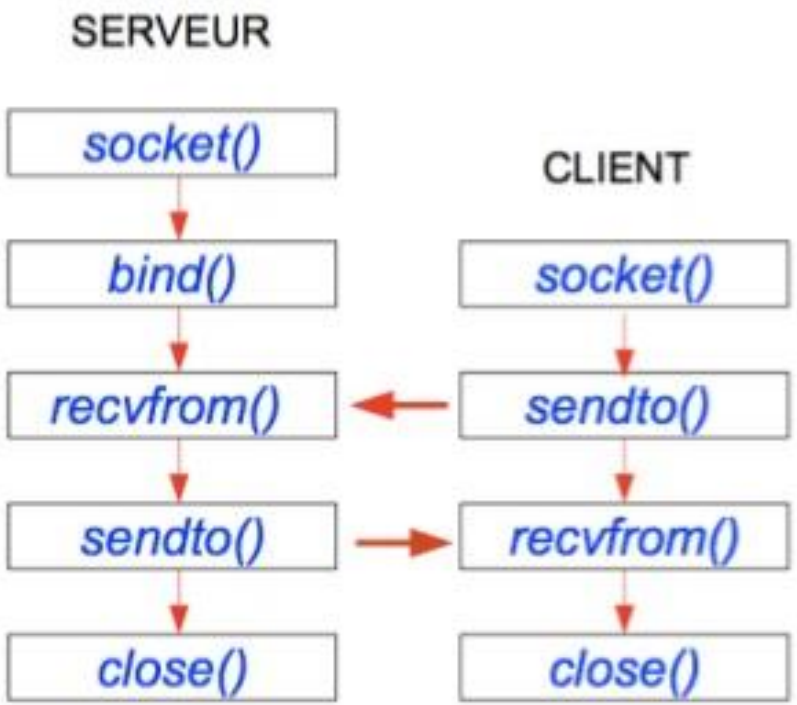
- Éléments à prendre en compte lors de la conception du serveur :
  - Gestion du(des) processus
    - Gestion des requêtes (priorités)
    - Exécution du service (séquentiel/parallèle)
  - Gestion de la mémoire et du stockage des informations
    - Taille des données manipulées
    - Lien entre appels successifs
  - Gestion des pannes
  - Vérification des échanges et détection des pannes,
  - Mémorisation de l'interaction et de l'état du client,
  - Processus de reprise

Processus unique



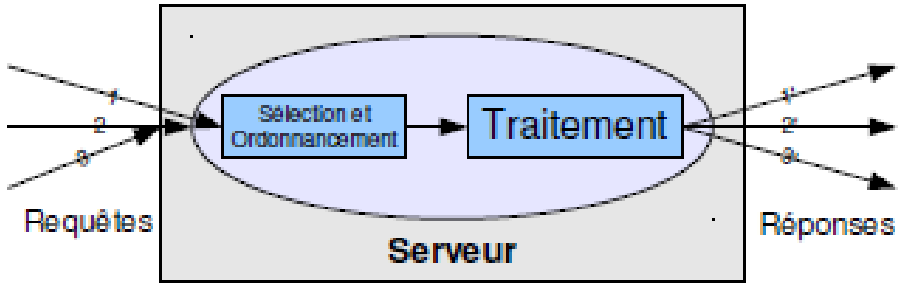

```
tantque Processus actif
     message ← receptionMessage( . . . )
     traitementMessage(message , . . . )
     traitementService( . . . )
     envoyerMessage ( . . . )
fintantque
```

Exemple de processus unique : sockets en mode non connecté « à la » C



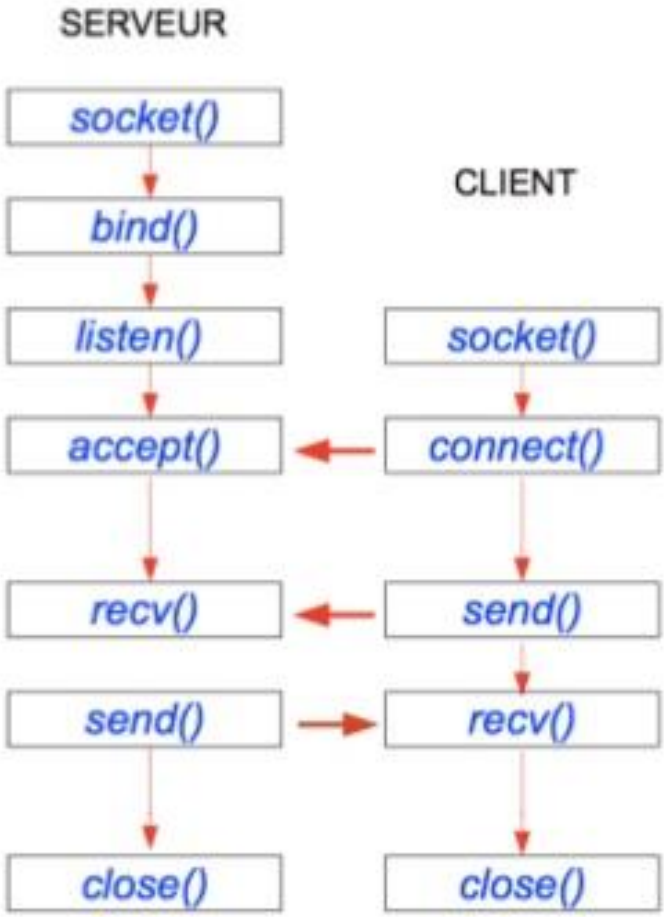
- *socket() : création d'une socket*
- *bind() : attache la socket à une adresse externe (@ et #Port)*
- *sendto() : envoi d'un message*
- *recvfrom() : réception du message*
- *close() : fermeture de la socket*

Processus unique avec gestion de file d'attente



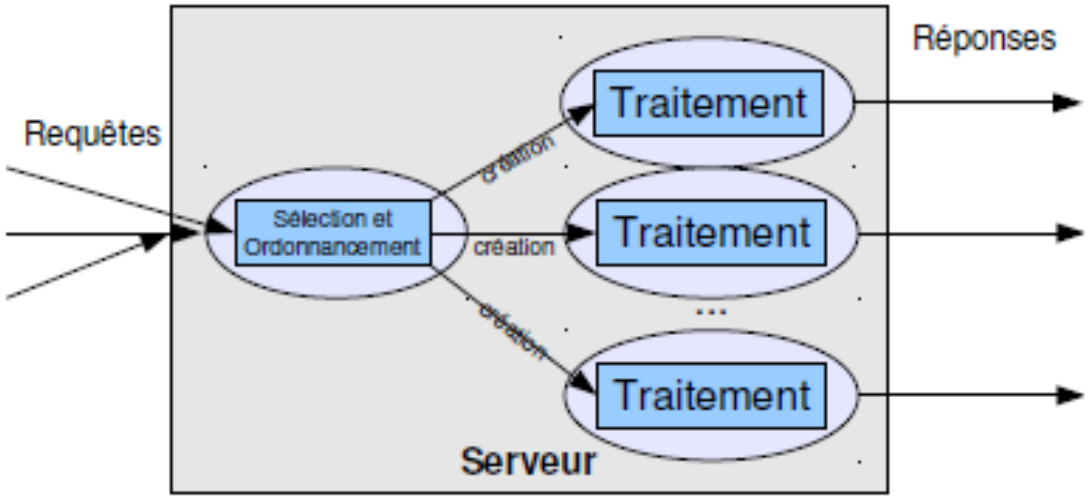

```
tantque Processus actif
     message ← defilerMessage( . . . )
     traitementMessage(message , . . . )
     traitementService( . . . )
     envoyerMessage ( . . . )
fintantque
```

Exemple de processus unique avec file : sockets en mode connecté « à la » C



- *socket() : création d' une socket*
- *bind() : attache la socket à une adresse externe (@ et #Port)*
- *listen() : écoute de connexions*
- *connect() : demande une connexion*
- *accept() : accepte la connexion*
- *send() : envoi de données*
- *recv() : réception des*

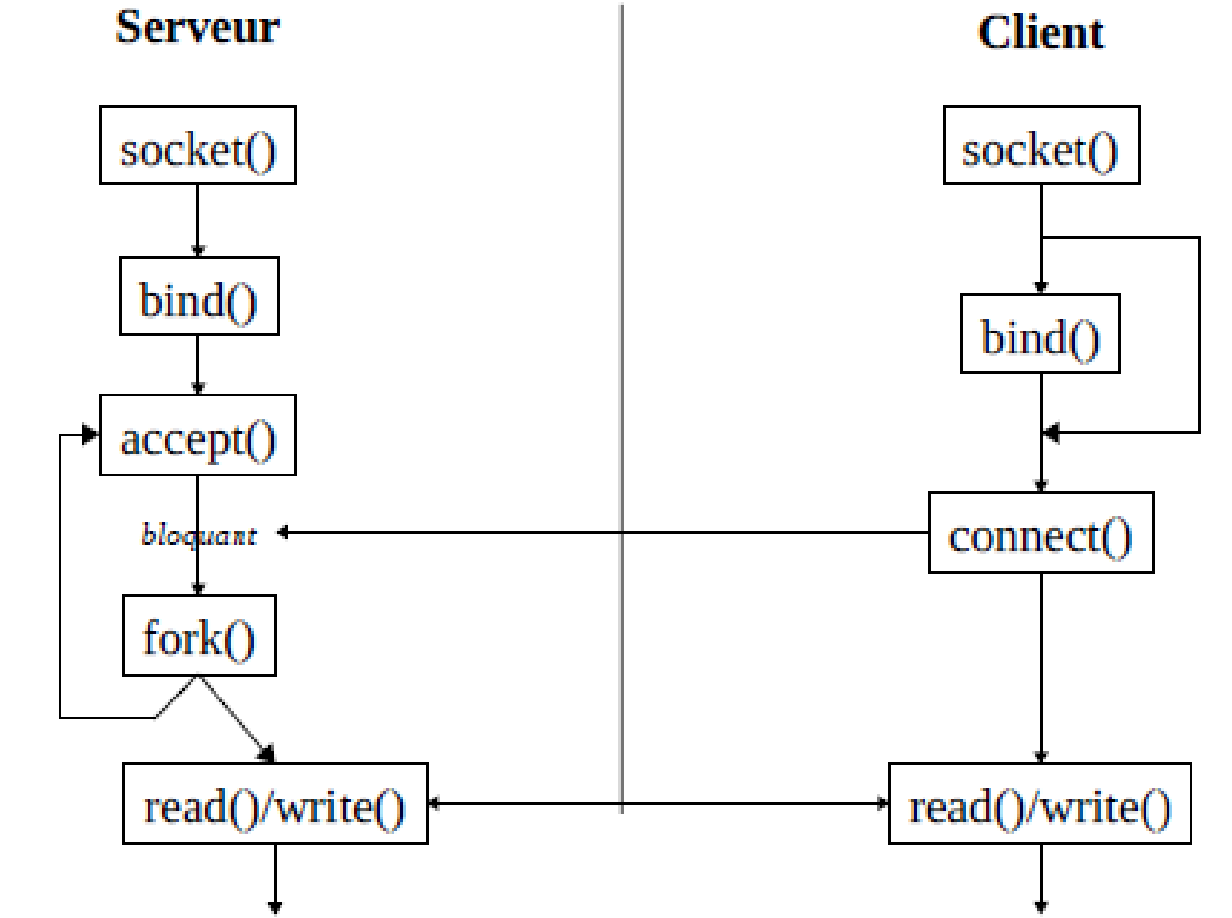
Création d'exécutants




```
tantque processus actif
      message ← receptionMessage(...)
      traitementMessage(message ,...)
      p ← creationProcessus
      processusTraitement (p, ...)
fintantque
```

```
procedure processusTraitement(p, ...)
debut
      traitementService( . . . )
      envoyerMessage ( . . . )
fin
```

Exemple de création d'exécutants : sockets connectés avec fork()



Types de service / données manipulées

- Sans données persistantes
  - Service fonction des paramètres d'entrée uniquement
  - Solution très favorable
    - tolérance aux pannes
    - contrôle de la concurrence
  - Exemple : calcul de fonction
- Avec données persistantes
  - Exécutions successives manipulent les données
    - modification du contexte d'exécution
    - problèmes de contrôle de la concurrence
    - difficultés en cas de panne en cours d'exécution
  - Exemple : serveur de fichiers répartis

Types de service / mode

- Appels de procédures non liés
  - Modification de données globales possible mais l'opération s'effectue sans lien avec les appels précédents
  - Exemple : serveur d'enregistrement avec accès aléatoire
- Appels de procédures liés
  - Appels successifs s'exécutent selon l'état laissé par les appels antérieurs
    - ordonnancement des requêtes
  - Exemples : serveur d'enregistrement avec accès séquentiel, utilisation de variables statiques, calculatrice avec mémoire

Appel de procédures à distance (RPC : Remote Procedure Calls)

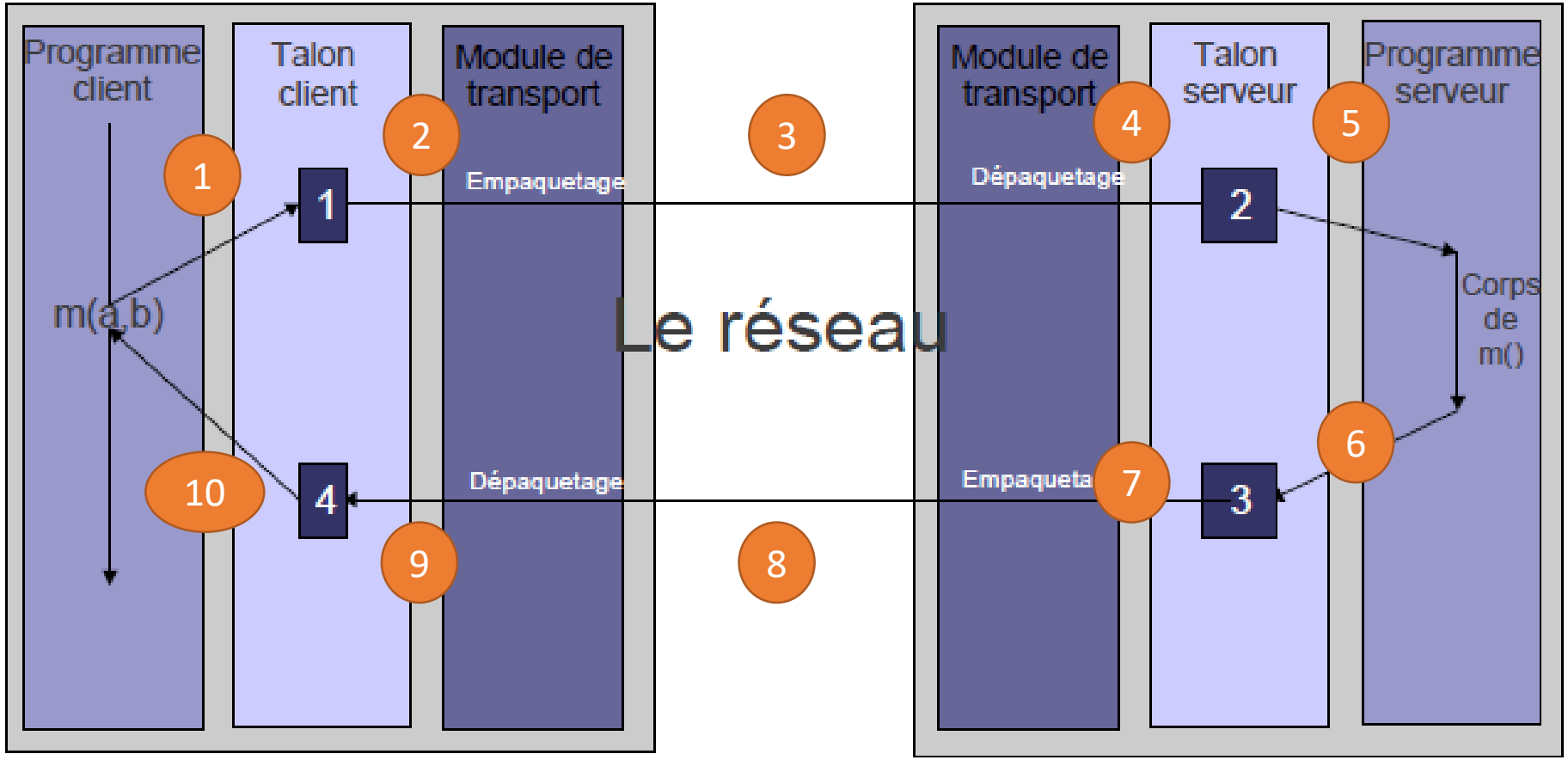
Description

- Infrastructure minimale pour mettre en place un Client-Serveur
- Service : procédure ou fonction que le client peut faire exécuter à distance par le serveur
- But : forme et effet identiques à ceux d'un appel local
  - sans se préoccuper de la localisation de la procédure
  - sans se préoccuper du traitement des pannes
- Mise en œuvre classique : **sockets**, RPC Sun
- Mise en oeuvre objet : Corba, **RMI**
- Problèmes courants :
  - Pannes indépendantes client/serveur
  - Problèmes réseau
  - Temps de réponse

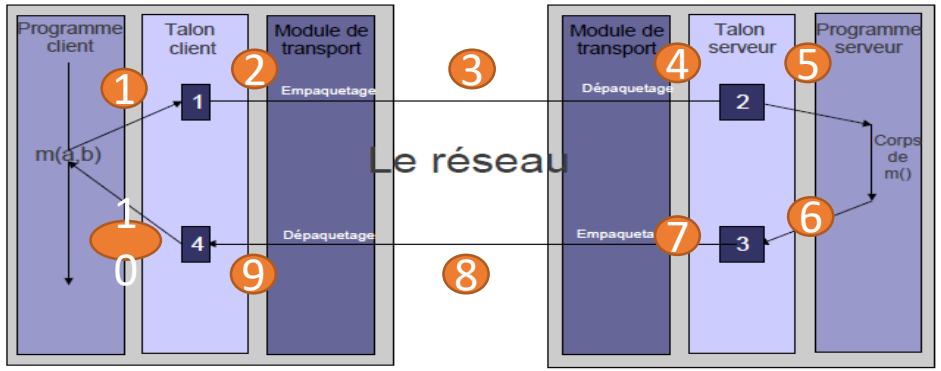
Description

- Le compilateur doit faire quelque chose de différent à un appel à procédures locales pour appeler une procédure distante
- Le RPC est une construction au niveau du langage (et pas au niveau du système d'exploitation)
- Il va être nécessaire de simuler l'appel à une procédure distante avec les appels à procédures locales avec des outils pour gérer la communication réseau (des sockets, par exemple)
- L'astuce est de créer des « fonctions talon » (*stub functions*) qui font croire au client qu'il est en train de faire un appel local
  - Mais le code concerne juste l'envoi et réception de messages sur le réseau

Principe de fonctionnement

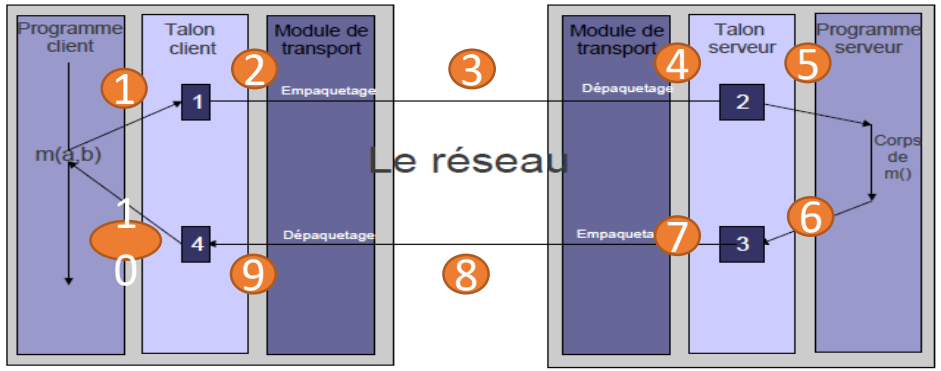


Principe de fonctionnement



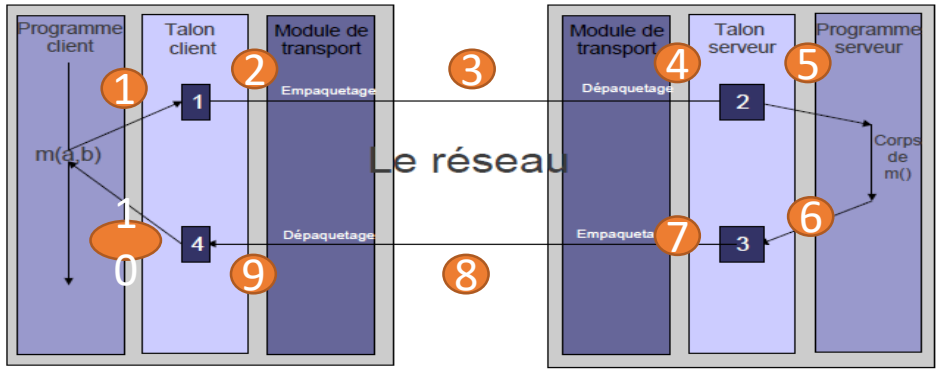
1. Le client appelle une procédure locale, le **talon client** (*client stub*). Ce talon emballe les paramètres de la procédure distante and construit un ou plusieurs messages réseau. Cet emballage s'appelle **marshalling** et précise la sérialisation de toutes les données dans un format plat (un tableau d'octets)
2. Les messages sont envoyés par le talon client vers le système distant (avec un appel système local)
3. Les messages sont transférés par le noyau du SE vers le système distant par le biais d'un certain protocole (connecté ou pas)

Principe de fonctionnement



4. Un **talon serveur** (*skeleton*) reçoit le message sur le serveur. Il désemballe les arguments du message et, si nécessaire, les convertit du format réseau standard vers un format spécifique à la machine
5. Le talon serveur appelle la fonction dans le serveur (qui, pour le client, est la procédure distante) et lui passe les arguments reçus du client
6. Quand la fonction serveur finit son exécution, les valeurs de retour sont retournées au talon serveur
7. Le talon serveur converti les valeurs de retour, si nécessaire, et les emballe dans un ou plusieurs messages réseau

Principe de fonctionnement



8. Les messages sont envoyés via le réseau
9. Le talon client lit les messages du noyau du SE
10. Le talon client retourne les résultats à la fonction cliente, en les convertissant, au préalable et si besoin, de la représentation réseau vers la représentation locale

- Le client continue son exécution

Avantages

- Les avantages principaux de l'appel à procédures distantes est double
  - Nous pouvons utiliser une sémantique classique d'appel à procédures pour invoquer des fonctions distantes et obtenir des réponses
  - L'écriture d'applications distribuées est facilitée, parce que le RPC cache tout le code réseau dans les fonctions talon. Les applications ne doivent pas se préoccuper des détails tels que les numéros de port, ou la conversion ou le « parsing » des données
- Par rapport au modèle OSI, les RPC se placent au niveau de las couches 5 et 6 (session et présentation)

Passage de paramètres

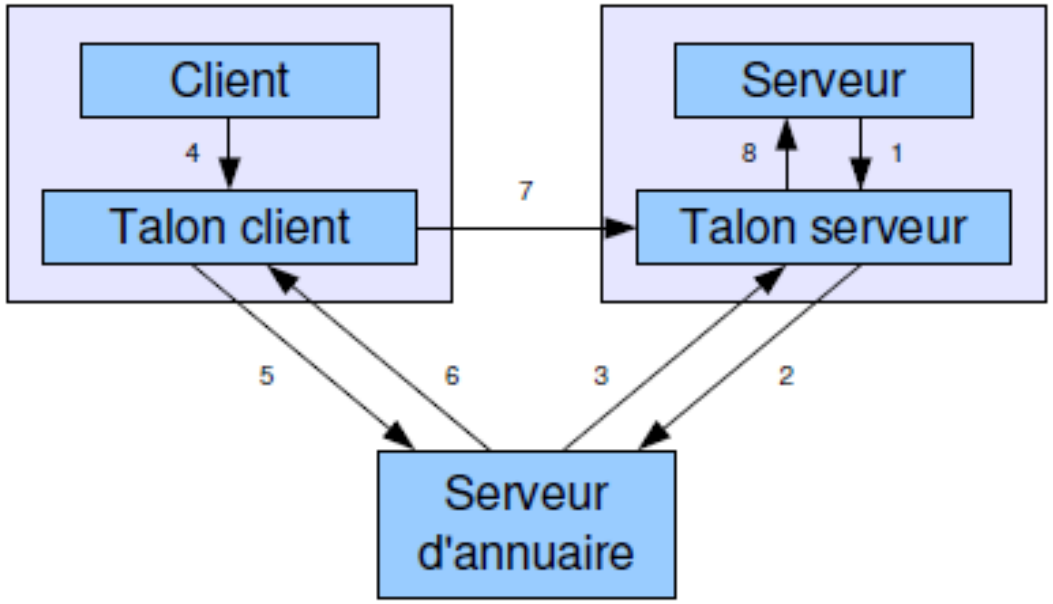
- Par valeur : facile !
- Par référence : adresse mémoire chez le client (resp. serveur)
  ⇒ aucun sens pour le serveur (resp. client)
  - Le pointeur (adresse) n'est connu que dans l'espace d'adressage du processus qui le crée. Cette adresse n'est valide que dans la machine où s'exécute le processus.
- Dans la plupart des RPC, interdiction totale des pointeurs
  - Introduit une différence dans le développement de procédures locales et celles destinées à un usage distant.
- Si absolument nécessaire
  - le talon client qui récupère un pointeur, copie le contenu de la zone adressée dans le message. Au retour elle place le résultat dans la zone → solution dangereuse
  - Mise en place d'une mémoire virtuelle partagée repartie → solution coûteuse

Représentation des données

- Conversion est nécessaire si le client et le serveur
  - n'utilisent pas le même codage
  - utilisent des formats internes différents
- *Solution normalisée* : syntaxe abstraite de transfert et représentation externe commune avec *typage implicite* ou *explicite*
  - Typage implicite : seule la valeur est transmise (pas le nom ou le type de la variable) ; ex : XDR (eXternal Data Representation) Sun
  - Typage explicite : Le type de chaque champ est transmis avec sa valeur ; ex : JSON, Google Protocol Buffers, ou des représentations basées sur XML

Localisation du serveur

- Pour localiser le hôte distant et le processus correspondant , deux solutions
  - Maintenir une BD centralisée pour identifier le hôte qui peut fournir un certain service. Un serveur envoie
    indiquant le type de RPC qu'il acc
    centrale pour identifier le service
  - Les clients connaissent d'avance l
    serveur de noms sur ce serveur m
    fournis localement (moins élégant



Sémantique des RPC

- Une procédure distante peut être exécutée
  - 0 fois, si le serveur s'arrête ou si le processus meurt avant d'exécuter le code du serveur
  - 1 fois, si tout va bien
  - 1 fois ou plus, si le serveur s'arrête après avoir retourné des résultats au talon serveur mais avant d'envoyer la réponse.
    - En absence de réponse, le client peut décider de réessayer, et alors, la fonction s'exécute plus d'une fois.
    - Si le client ne réessaye pas, la fonction est exécutée une seule fois
  - Plus d'une fois, le délai d'attente du client expire et il retransmet la requête.
    - Il est possible que la requête initiale soit en retard. Les deux requêtes peuvent être exécutées (ou pas)

Sémantique des RPC

- En général, les systèmes RPC offrent une sémantique « au moins une fois » ou « au plus une fois » ou un choix entre ces deux possibilités
  - Il est nécessaire de comprendre la nature de l'application et de la fonction pour déterminer si c'est sûr d'appeler une fonction plus d'une fois
- Une fonction qui peut être exécuté plusieurs fois sans danger est appelée **idempotente**
  - heure de la journée, fonctions mathématiques, lecture de données statiques
- Sinon, il s'agit d'une fonction **non idempotente**
  - Modification d'un fichier, par exemple

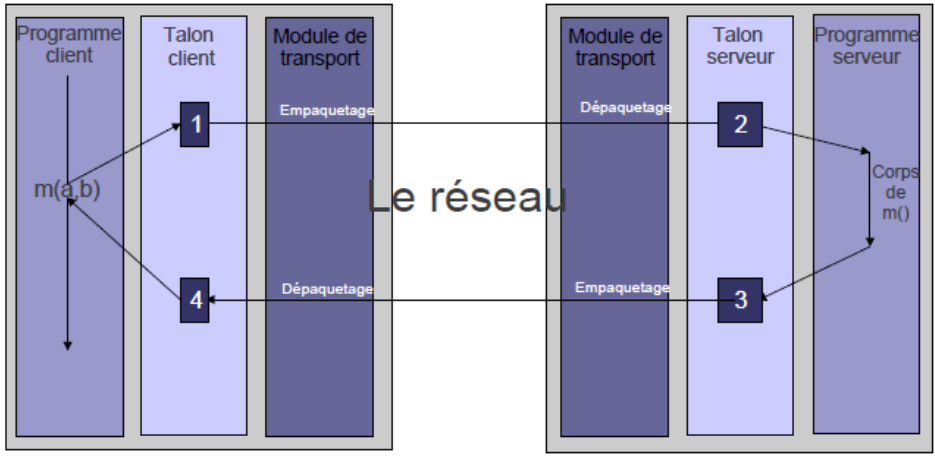
Problématiques courantes

- Défaillances
  - Congestion du réseau ou du serveur
  - Panne du client
  - Panne du serveur
  - Erreur de transport ou de communication
  - …
- Problèmes de sécurité
  - Authentification du client
  - Authentification du serveur
  - Confidentialité des échanges
  - …
- Performance
- …

Types de panne

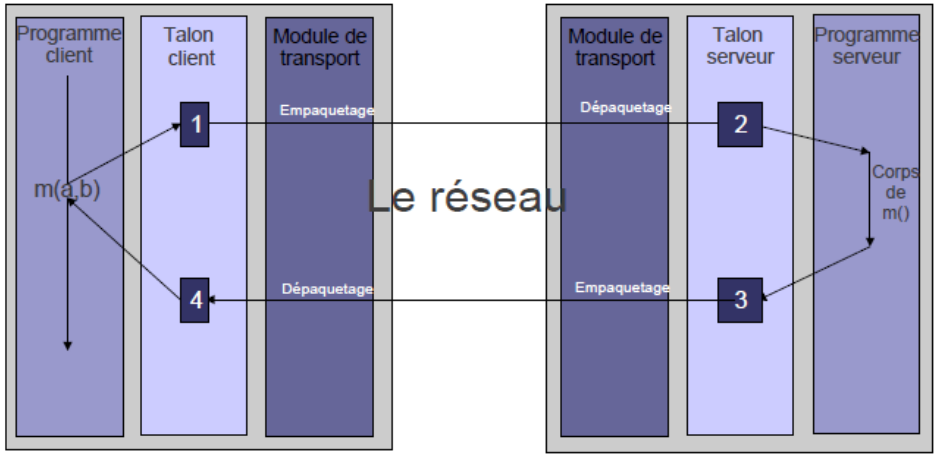
- Panne du serveur ⇒ attente du client
  - Client décide de la stratégie de reprise
  - Serveur applique la stratégie de reprise
  - Risque d'exécuter plusieurs fois la même procédure
- Panne du client ⇒ serveur orphelin
  - Réalisation de travaux inutiles
  - Risque de confusion du client
  - États inconsistants
- En cas d'erreur
  - Détection à l'aide d'horloges de garde
  - Mécanisme de reprise : nombre de relances en cas de dépassement de délai (infini, au moins une fois, au moins X fois, ...)

Traitement d'une panne client



- Panne du client après émission de la requête
  ⇒ requête est correctement traitée
  - Changement d'état du serveur
  - L'appel de procédure est déclaré orphelin
- Détection : expiration du délai de garde 3
- Recouvrement :
  - Client réémet la requête : sémantique "Au moins UN"
  - Serveur ne peut pas détecter la répétition (id différente)
    - Service idempotent : pas d'incidence
    - Service non idempotent : service transactionnel (annulation par le client des effets de l'appel orphelin)

Traitement d'une panne serveu



- Panne du serveur après émission de la requêt
  ⇒ requête peut-être partiellement traitée
- Détection : expiration du délai de garde 1
- Recouvrement :
  - Client réémet la requête : sémantique "Au moins UN"
  - Client ne connaît pas l'endroit de la panne
    - Si avant 2 : pas d'incidence
    - Si entre 2 et 3 : changement d'état du serveur
  - Service transactionnel pour mémoriser id et l'état avant exécution
    ⇒ gestion serveur

Sécurité

- Plusieurs aspects à considérer
  - S’assurer que le processus distant n’est pas un imposteur
  - S’assurer que la machine distante n’est pas un imposteur
  - S’assurer que le serveur accepte des requêtes de clients légitimes.
  - S’assurer que le message ne soit pas lu lors de la transmission sur le réseau
  - S’assurer que le message ne peut pas être intercepté et changé lors de la transmission sur le réseau
  - S’assurer que le protocole ne peut pas subir des attaques de substitution de messages par un hôte malicieux
  - S’assurer que le message ne soit pas corrompu ou tronqué lors de la transmission
  - …

Performance

- Typiquement, un RPC est des milliers des fois plus lent qu'un appel local
- L'utilisation de mémoires cache au niveau du processus, dans le noyau ou dans un serveur spécialisé peut améliorer la performance globale
  - Ex DNS