Master mention Informatique M1

Construction d'Applications Réparties

Romain ROUVOY
Romain.Rouvoy@univ-lille1.fr
2016-17

Objectifs du cours

Appréhender la conception d'applications réparties
- motivations et concepts
- architectures et exemples
- problèmes et solutions

Comprendre les solutions
- Internet et sockets Java
- Web Services, REST, SOAP
- Objets répartis en Java (RMI)
- Composants Java EE (JSP, servlet, EJB)

Maîtriser par la pratique (importance des TP)
- Java, REST, RMI, Java EE

Sommaire

1. Introduction aux applications réparties
2. Applications réparties en mode message
3. Web Services
4. Objets et répartition
5. Java EE

Sommaire

1. Introduction aux applications réparties
2. Applications réparties en mode message
   1. Protocoles TCP, UDP, Multicast-IP
   2. Programmation concurrente multi-thread
3. Web Services
   1. SOAP, WDSL
   2. REST
4. Objets et répartition
   1. Java RMI
   2. Akka
5. Java EE
   1. JSP/servlet
   2. JDBC
   3. EJB

Organisation

- début cours : 10 janvier
- début TD : semaine du 16/1
- début TP : semaine du 23/1

Enseignants TD/TP
- C. Ballabriga, M. Colmant, L. Duchien, G. Lipari, L. Seinturier

http://www.fil.univ-lille1.fr/portail
- M1S2 > CAR

http://moodle.univ-lille1.fr/course/view.php?id=346
- clé d'inscription : car

Organisation du contrôle

Un examen note : 60%

4 sujets de TPs note : 40%
- chaque sujet de TP dure 3 (ou 2) séances
- utilise le langage Java
- démonstration de chaque TP en séance
- les TPs sont relevés et notés
  - relève par l'application PROF uniquement (mail refusé)
  - dates de remise fermes
  - binôme ∈ même groupe (pas de changement de binôme)

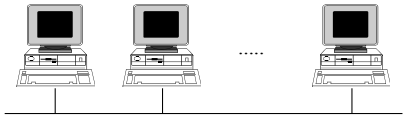
I. Introduction aux applications réparties

Romain Rouvoy

Romain.Rouvoy@univ-lille1.fr

Introduction

Problématique



Permettre à un programme de s'exécuter sur **plusieurs machines**
(PC, *mainframe*, *laptop*, PDA, …) reliées par un réseau
- à large échelle (Internet)
- local (intranet)

∩ de plusieurs domaines de l'informatique

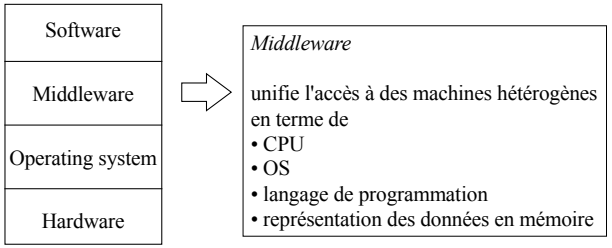
- système d'exploitation - réseau - langage de programmation	- système d'exploitation répartis - librairies de programmation réseau - langages de programmation étendus

Middleware

Introduction

Vocabulaire

• application répartie
• exécutées par des plates-formes dite *middleware* (en français intergiciel)



Middleware

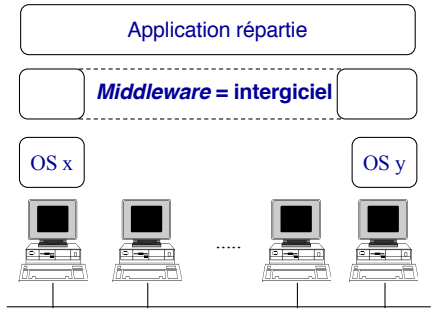
unifie l'accès à des machines hétérogènes en terme de
• CPU
• OS
• langage de programmation
• représentation des données en mémoire

Introduction

Middleware

• masque hétérogénéité des machines et des systèmes
• masque répartition des traitements et données
• fournit une interface aux applications (modèle de programmation + API)

« *Middleware is everywhere* »
© IBM

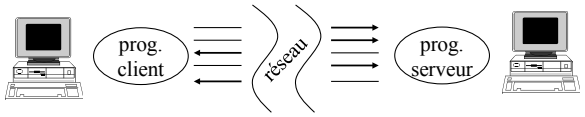


Introduction

Modes de communications

2 principaux : message, requête/réponse

Message

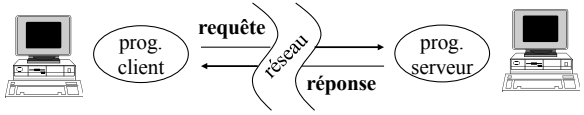


- simple
- le plus proche du fonctionnement du réseau
- primitives send/receive
- l' "assembleur" de la programmation répartie
- communication dite asynchrone

Introduction

Modes de communications

Requête/réponse



- 1 requête + 1 réponse
- demande d'exécution d'un traitement à distance et réponse
- ≈ appel procédural étendu au cas où appelant et appelé ne sont pas situés sur la même machine
- communication dite synchrone

Introduction

Modes de communication

Troisième paradigme

⇨ interaction dite **par messagerie** ou par bus de messages
⇨ en anglais MOM (*Message-Oriented Middleware*)

- découplage du client et du serveur
- meilleur passage à l'échelle
- meilleure tolérance aux pannes
- souvent associé à des mécanismes d'abonnement

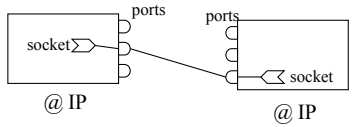
Introduction

Evolution du middleware

Envoi de messages par socket

- primitives send & receive
- conception des progs client et serveur en fonction messages attendus et à envoyer

- socket : API C au-dessus des protocoles TCP & UDP
- fiabilité, ordre, ctrl de flux, connexion
- send/receive bloquant/non bloquant



Introduction

Remote Procedure Call (RPC)

- appel d'une procédure sur une machine distante
- groupement de 2 messages : appel & retour
- adressage : @IP + nom fonction
- définition des signatures des procédures
- compilateur de souches client et serveur

Exemple : RPC Sun

```
struct bichaine { char s1[80]; char s2[80]; };

program CALCUL {
        version V1 {
                int multpar2(int) = 1;
                string concat(struct bichaine) = 2;
                void plus_un() = 3;
        } = 1;
} = 0x21234567;
```


Introduction

RPC Objet

- mise en commun concepts RPC et prog objet
- appel d'une méthode sur un objet distant
- éventuellement +sieurs objets par machine
- adressage serveur de noms + nom logique

Exemple : Java RMI

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface CompteInterf extends Remote {
    String getTitulaire() throws RemoteException;
    float solde() throws RemoteException;
    void deposer( float montant ) throws RemoteException;
    void retirer( float montant ) throws RemoteException;
    List historique() throws RemoteException;
}
```

1. Modèle de programmation

Modèle de programmation

1.1 Côté serveur

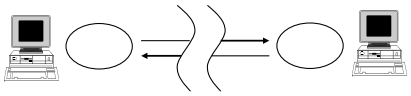
1.2 Communications client/serveur

- Modes
- Concepts
- Notion de connexion
- Gestion d'états
- Représentation des données
- Passage de paramètres
- Traitement des pannes

1. Modèle de programmation

Modèle de programmation

2 programmes



- 1 programme client
- 1 programme serveur

⇨ **processus distincts**
⇨ **mémoires distinctes**
⇨ machines distinctes (sauf si répartition logique)

Selon le contexte 1 programme peut être client et serveur

- 1 programme client
- 1 **programme serveur** qui pour rendre le service **est client** d'un 3è programme
- 1 programme serveur

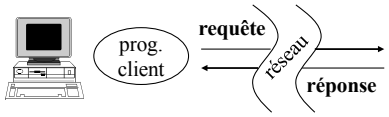
⇨ être client, être serveur, n'est pas immuable
mais **dépend de l'interaction considérée**

1. Modèle de programmation

Modèle de programmation

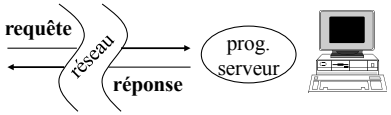
Point de vue du client

1. envoie une requête
2. attend une réponse



Point de vue du serveur

1. attend une requête
2. **effectue un traitement et produit une réponse**
3. envoie la réponse au client



mais le serveur doit aussi pouvoir traiter
les requêtes de **plusieurs clients**

1.1 Côté serveur

Plusieurs clients simultanément

1 bis. sélection de la requête à traiter
(FIFO ou avec priorité)

Plusieurs mises en oeuvre possibles pour le traitement de la requête

- 1 activité unique
- 1 activité par requête
- 1 *pool* d'activités

rq : activité = processus ou *thread*

1.1 Côté serveur

Plusieurs clients simultanément - 1 activité unique

```
while (true) { 1 2 3 }
```

Plusieurs clients envoient des requêtes simultanément
mais le serveur n'en traite qu'une à la fois

→ simple
→ pas de risque de conflit de concurrence
→ suffisant dans certains cas
(ex. 1 requête toutes les 10s qui demande 2s de traitement)

1.1 Côté serveur

Plusieurs clients simultanément - 1 activité par requête

Chaque arrivée de requête
déclenche la création d'une activité


```
while (true) { 1 fork() }
p1     p2     p3     ...
2 3    2 3    2 3    ...
```

→ les clients sont servis + rapidement
→ conflits éventuels en cas d'accès simultanés à une ressource partagée (ex. fichier)

Problème : une concurrence "débridée" peut écrouler la machine
→ restreindre le nombre d'activités

1.1 Côté serveur

Plusieurs clients simultanément - *pool* d'activités

• *pool* fixe
• *pool* dynamique

Pool fixe

• 1 nombre constant d'activités
• 1 activité qui reçoit les requêtes et les dispatche aux activités du *pool*
• si aucune activité n'est libre, les requêtes sont mises en attente

Avantage
- pas de risque d'écroulement
(pour peu que le nombre d'activités soit correctement dimensionné)

Inconvénients
- un *pool* d'activités inactives consomme des ressources inutilement
- les pointes de traffic sont mal gérées

1.1 Côté serveur

Plusieurs clients simultanément - *pool* de processus

Pool dynamique

Toujours avoir des activités prêtes sans surcharger
inutilement la machine

→ le nombre d'activités varie
→ mais le nombre d'activités prêtes est toujours compris entre 2 bornes
→ mélange des politiques 1 proc/req et *pool* fixe

• nb max d'activité (ex. 150)
• nb max d'activités inactives (ex. 20) : au delà on détruit les activités
• nb min d'activités inactives (ex. 5) : en deça on crée de nouvelles activités
• nb d'activités créées au départ (ex. 15)

rq : solution retenue par Apache

1.2 Communications

Modèle de programmation

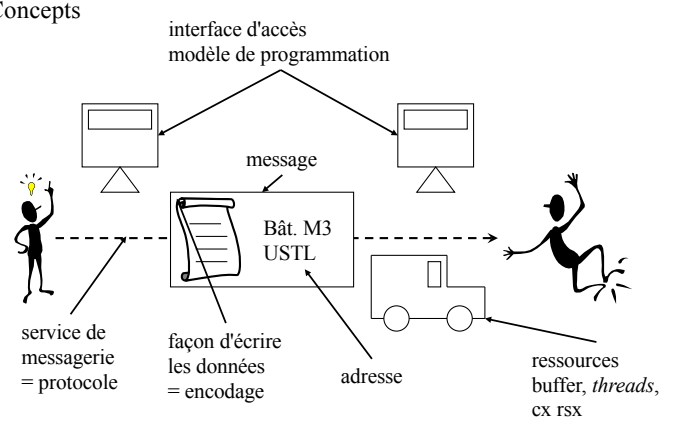
1.1 Côté serveur

1.2 Communications client/serveur

- Concepts
- Notion de connexion
- Gestion d'états
- Représentation des données
- Passage de paramètres
- Traitement des pannes

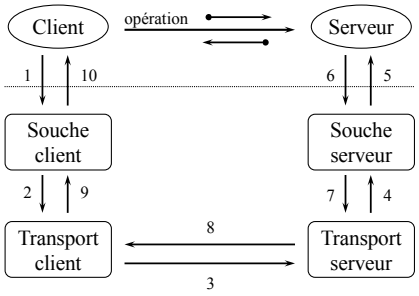
1.2 Communications

Concepts



1.2 Communications

Mise en oeuvre des concepts



1. Appel local
2. Préparation du message d'appel
2. Envoi
4. «Upcall» vers la souche serveur
5. Décodage du message
6..10 : Chemin inverse

Vocabulaire français souche ou talon
Vocabulaire anglais souche cl. = *stub* ou *proxy*, souche serv. = *skeleton*

1.2 Communications

Connexion

Problématique

Délimitation des communications entre un client et un serveur

Mode non connecté (le + simple)
- les messages sont envoyés "librement"
- example : NFS

Mode connecté
- les messages sont
  - précédés d'une ouverture de connexion
  - suivis d'une fermeture de connexion
- facilite la gestion d'état
- permet un meilleur contrôle des clients
- ex : FTP, Telnet, SMTP, POP, JDBC, HTTP

Rq : la cx est le + souvent liée au transport (TCP) plutôt qu'au protocole applicatif lui-même

1.2 Communications

Gestion d'états du protocole de communication

Problématique

2 requêtes successives d'un même client sont-elles indépendantes ?
→ faut-il sauvegarder des infos entre 2 requêtes successives d'un même client ?

Mode sans état (le + simple)

- pas d'info sauvegardée
- les requêtes successives d'un même client sont indépendantes
- ex : NFS, HTTP

Types de requêtes envisageables
- demande d'écriture du **k-ième** bloc de données d'un fichier

Types de requêtes non envisageables
- demande d'écriture du bloc **suivant**

1.2 Communications

Gestion d'états du protocole de communication

Mode avec état

- les requêtes successives s'exécutent
  en fonction de l'état laissé par les appels précédents
  → sauvegarde de l'état

Notion proche : session cliente dans un serveur Web
Suivi de l'activité d'un client entre le moment où il arrive et celui où il quitte le site

Pb : y a-t-il un mécanisme explicite qui indique au serveur que le client part ?
- si oui (ex. déconnexion notifiée au serveur) alors ok
- si non
  - pb : aucun sens de conserver *ad vitam* les données de la session
  - heuristique : la session expire au bout d'un délai fixé
  - inconv. : un client très lent peut revenir après expiration
  - → (re)commencement d'une nouvelle session

1.2 Communications

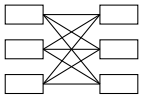
Représentation des données

Problématique

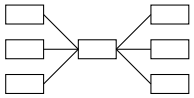
Comm. entre machines avec des formats de représentation de données ≠
→ pas le même codage (*big endian* vs *little endian*)
→ pas la même façon de stocker les types (entiers 32 bits vs 64 bits, ...)

2 solutions

On prévoit tous les cas de conversions possibles
(n^2 convertisseurs)



On prévoit un format pivot et on
effectue 2 conversions (2n convertisseurs)



1.2 Communications

Passage de paramètres

Problématique

Client et serveur ont des espaces mémoire ≠
→ passage par valeur ok
→ passage par référence
  pas possible directement
  une ref. du client n'a aucun sens pour le serveur (et vice-versa)

Solution : mécanisme copie/restauration

1. copie de la valeur référencée dans la requête
2. le serveur travaille sur la copie
3. le serveur renvoie la nouvelle valeur avec la réponse
4. le client met à jour la réf. avec la nouvelle valeur

1.2 Communications

Passage de paramètres

Mais copie/restauration pas parfait

Problème du double incrément

```
void m(&x, &y) { x++; y++; }
a=0; m(a,a);
```

résultat attendu a=2, résultat obtenu a=1 !!

⇨ détecter les doubles (multiples) référencements
⇨ pas facile dans le cas général

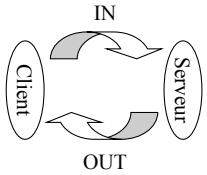
Problème en cas de mises à jour concurrentes de la valeur référencée

1.2 Communications

Passage de paramètres

Définitions couramment adoptée (au lieu de valeur/référence)

- mode IN (entrée) passage par valeur avec la requête
- mode OUT (sortie) passage par valeur avec la réponse
- mode IN/OUT (entrée/sortie) copie/restauration de la valeur



• IN si le serv. modifie la valeur, le cl. ne "voit" pas cette modif.
• OUT si le cl. transmet une valeur, le serv. ne la "voit" pas

1.2 Communications

Traitement des pannes

Dans la majorité des cas

• symptôme : absence de réponse
• cause inconnue : réseau ? client ? serveur ?

Techniques **logicielles** de détection des pannes

• *heart beat* périodiquement le serveur signale son activité au client
• *pinging* périodiquement le client sonde le serveur qui répond

➢ les résultats ne sont jamais sûr à 100%
➢ périodicité délicate à régler
➢ impossibilité de distinguer une “vraie” panne d’un ralentissement dû à surcharge
➢ pas une vraie détection, possibilité de fausse détection
➢ on parle de **suspicion de panne**

1.2 Communications

Traitement des pannes

Comportement possible en présence de pannes

client envoie requête
si panne signalée par détecteur
alors signaler la panne au client

➢ la requête s’exécute (si pas de panne), sinon elle ne s’exécute pas
➢ comportement dit “au plus 1 fois” (0 fois ou 1 fois)

1.2 Communications

Traitement des pannes

2ème comportement possible en présence de pannes : “au moins 1 fois”
(1 fois ou n fois)

client envoie requête
tant que résultat non reçu
  attendre délai // éventuellement attente interrompue par détecteur
  renvoyer requête

- tentatives de réémissions pour compenser les pertes de message
- en cas de fausse détection de panne
  le message est reçu +sieurs fois ⇨ le traitement s’exécute +sieurs fois

ok si idempotent
i.e. plusieurs exécutions du même traitement ne doivent pas poser problème

idempotent	x:=5	lire_fichier(bloc k)	ecrire_fichier(bloc k)
¬idempotent	x++	lire_fichier_bloc_suivant()	

1.2 Communications

Traitement des pannes

3ème comportement possible en présence de pannes

idem “au moins 1 fois”
+ numérotation des messages
en vue de détecter (côté serveur) les réémissions

- le traitement s’exécute exactement 1 fois