USTL – Master mention Informatique – M1

Construction d'applications réparties

IV. Client/serveur et objets

Lionel.Seinturier@lifl.fr

Plan

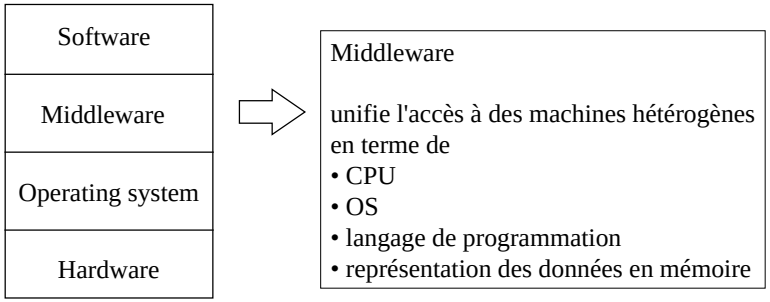
Introduction

1. Modèle de programmation client/serveur
   1.1 Côté serveur
   1.2 Communications
2. Client/serveur orienté objet
3. Bibliographie

Introduction

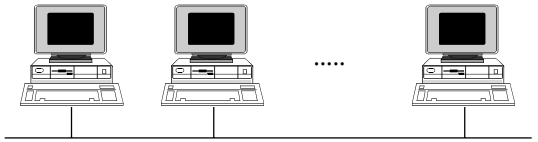
Middleware

désigne dans le cadre de l'informatique répartie, toutes les couches logicielles qui permettent à des applications de **communiquer à distance**



Introduction

Problématique du middleware



Permettre à un programme de s'exécuter sur **plusieurs machines** reliées par un réseau
- à large échelle (Internet)
- local (intranet)

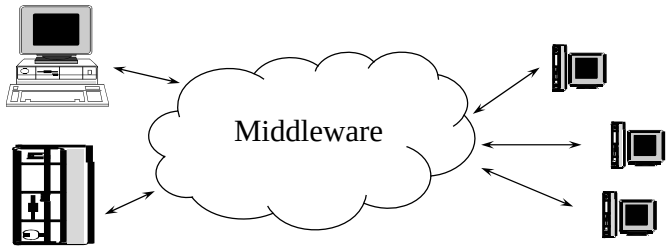
Middleware : ∩ de plusieurs domaines de l'informatique

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

Introduction

Problématique du middleware

Les environnements middleware permettent
à ≠ types de matériels (PC, mainframes, laptop, PDA, ...)
de communiquer à distance

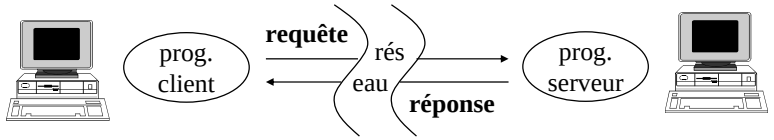


Introduction

Problématique du middleware

Nombreux paradigmes de communication associés

⇨ le principal : interaction **requête/réponse** ou **client/serveur**



Interaction client/serveur

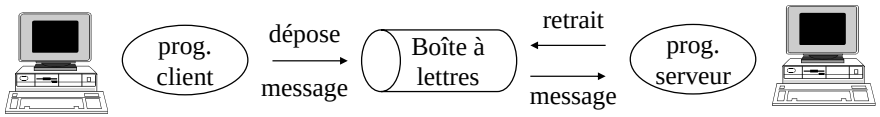
= 1 requête + 1 réponse
= demande d'exécution d'un traitement à distance et réponse
≈ appel procédural étendu au cas où appelant et appelé
ne sont pas situés sur la même machine

Introduction

Problématique du middleware

Deuxième paradigme

⇨ interaction **par messagerie** (MOM : *Message-Oriented Middleware*)



Attention ≠ envoi message sur socket
Protocoles de niveau applicatif (pas transport) + propriétés (ex transactionnelles)

MOM : comm. **asynchrone** (fonctionnement client et serveur découplés)
Interaction client/serveur comm. **synchrone**

Introduction

Problématique du middleware

En général, le middleware

- n'est pas visible par l'utilisateur final
- est un outil pour le développeur d'applications
- se retrouve enfoui dans les applications

Middleware permet de mettre en oeuvre des serveurs

- à finalité fixe : serveur Web, serveur de fichiers, serveur de BD, ...
- effectuant des traitements quelconque : CORBA, EJB, .Net, Web Services, ...

Introduction

Client/serveur

Notion d'application client/serveur 2 tiers ou 3 tiers

Découpage d'une application

- présentation
- traitements
- données

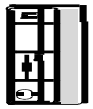
Problématique

Par rapport à 1 client et 1 (ou +sieurs) serveurs
qui assure ces fonctionnalités ?

Introduction

Client/Serveur 2 tiers

client
présentation



serveur
données
+ traitement

Caractéristiques
- 1 gros serveur (mainframes)
- n terminaux légers connectés au serveur

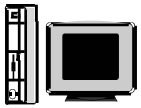
Avantages
- pas de duplication de données (état global observable)
- gestion simple de la cohérence et de l'intégrité des données
- maîtrise globale des traitements

Inconvénients
- modèle trop rigide qui n'assure pas l'évolutivité
- souvent solutions propriétaires fermés
- économiquement trop coûteux

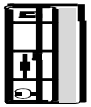
Introduction

Client/Serveur 3 tiers

client
présentation



serveur
traitement



serveur
données

Caractéristiques
- 1 serveur de données et 1 serveur de traitement
- n PC avec IHM "évoluées"

Avantages
- meilleure répartition de charge
- économiquement moins cher
- + évolutif

Inconvénients
- administration + compliquée
- mise en oeuvre + compliquée

Introduction

Client/Serveur 3 tiers

Evolution historique du terme client/serveur 3 tiers

• tiers 1 (PC)	tiers 2 (serveurs départementaux)	tiers 3 (serveur central)
• tiers 1 (client)	tiers 2 (BD locale)	tiers 3 (BD globale)

Actuellement

• tiers 1 (client)	tiers 2 (serveur de traitement)	tiers 3 (serveur de données)

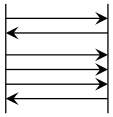
Introduction

Evolution du middleware

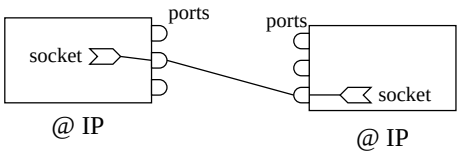
socket C, RPC, RPC objet, bus logiciel (CORBA)

Envoi de messages par socket

- primitives send & receive
- conception des progs cl et serv 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



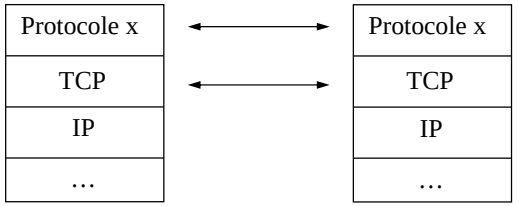
Introduction

Envoi de messages par socket

TCP, UDP : socles pour la construction de protocoles de + haut niveau

Définition de protocole

- message + paramètres
- format des messages
- enchaînement des messages
- cas d'erreur : message, format, paramètres, enchaînement



!! distinction entre niveaux !!
⇨ utilisation des services du protocole sous-jacent
ex. : ouverture de connexion TCP

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

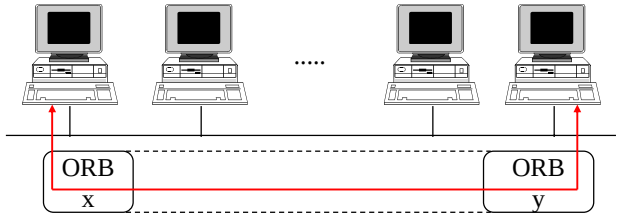
Exemple : Java RMI

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

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

Introduction

Bus logiciel CORBA



- multi-OS, multi-langage
- métaphore du bus logiciel
- langage IDL

```
module tp7 {
        interface CalculatriceItf {
                double add( in double val1, in double val2 );
                double sub( in double val1, in double val2 );
                double mult( in double val1, in double val2 );
                double div( in double val1, in double val2 );
        };
};
```

- services : nommage, courtage, transaction, …

Introduction

Environnements middleware

- Sun Java = support d'exécution universel
- OMG CORBA = middleware universel
- Microsoft .NET + Web Services = l'intéropérabilité universelle

La semaine type d'un développeur

- lundi : C & les sockets
- mardi : C & RPC
- mercredi : Java & servlet
- jeudi : Java, C++ & CORBA
- vendredi : Java & CORBA CCM
- samedi : C# & Web Services
- dimanche : Prozac

⇗ exponentielle de la technologie

Avant 1980 : **procédural**
- procédures, fonctions

1980-1995 : **OO**
- objects, classes, méthodes, héritage, polymorphisme

Depuis 1995 : **composants**
- composants, packages, conteneurs, *frameworks*, *patterns*, …

⇗ exponentielle des concepts

1. Modèle c/s

Modèle de programmation client/serveur

1.1 Problématique côté serveur

Processus
Gestion d'états
Persistance des données

1.2 Communications client/serveur

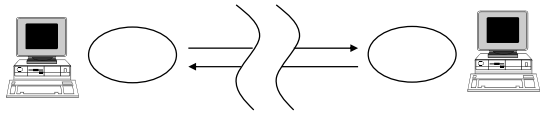
Modes
Concepts
Notion de connexion
Représentation des données
Passage de paramètres
Traitement des pannes

1. Modèle c/s

Modèle de programmation client/serveur

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 (ex. Napster, ...)

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

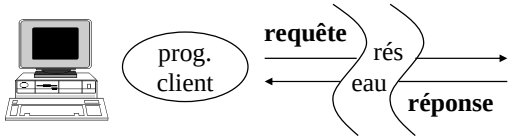
⇨ être client, être serveur, n'est pas immuable
mais **dépend de l'interaction considérée**

1. Modèle c/s

Modèle de programmation client/serveur

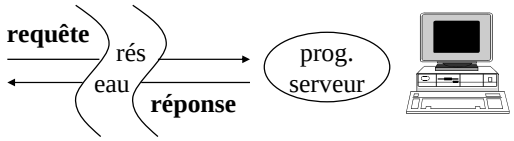
Point de vue du client

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 **simultanément** 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 processus unique
- 1 processus par requête
- 1 pool de processus

rq : processus → *thread*

1.1 Côté serveur

Plusieurs clients simultanément - 1 processus unique

```
while (true) { 1 1bis 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 processus par requête

Chaque arrivée de requête
déclenche la création d'un processus


```
while (true) { 1 1bis 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 de processus traitants

1.1 Côté serveur

Plusieurs clients simultanément - *pool* de processus

• *pool* fixe
• *pool* dynamique

Pool fixe

• 1 nombre constant de processus
• 1 processus qui reçoit les requêtes et les dispatche aux processus du *pool*
• si aucun processus n'est libre, les requêtes sont mises en attente

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

Inconvénients
- un *pool* de processus inactifs 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 processus prêt sans surcharger inutilement la machine

→ le nombre de processus varie
→ mais le nombre de processus prêts est toujours compris entre 2 bornes
→ mélange des politiques 1 proc/req et *pool* fixe

• nb max de proc (ex. 150)
• nb max de proc inactifs (ex. 20) : au delà on détruit les proc
• nb min de proc inactifs (ex. 5) : en deça on crée de nouveaux proc
• nb proc créés au départ (ex. 15)

rq : solution retenue par Apache

1.1 Côté serveur

Gestion d'états

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**

1.1 Côté serveur

Gestion d'états

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

1.1 Côté serveur

Persistance des données

Problématique

le serveur gère-t-il des données globales partageables par tous les clients ?
rq : problématique ≠ de l'état

Mode sans persistance (le + simple)

- le service s'exécute uniquement en fonction des paramètres

ex : calcul d'une fonction mathématique

Situation favorable pour de nbreux pbs de système réparti (tolérance aux pannes, équilibrage de charge, reprise après panne)

Mode avec persistance

ex : serveur de fichiers,
serveur de BD

Pb en cas d'accès simultanés par ≠ clients
→ contrôle de concurrence

1.2 Communications

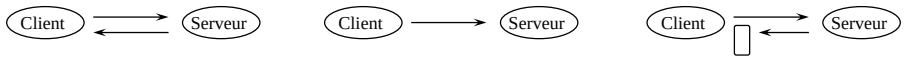
Modèle de programmation client/serveur

1.1 Problématique côté serveur
- Processus
- Gestion d'états
- Persistance des données

1.2 Communications client/serveur
- Modes
- Concepts
- Notion de connexion
- Représentation des données
- Passage de paramètres
- Traitement des pannes

1.2 Communications

Modes de communication client/serveur

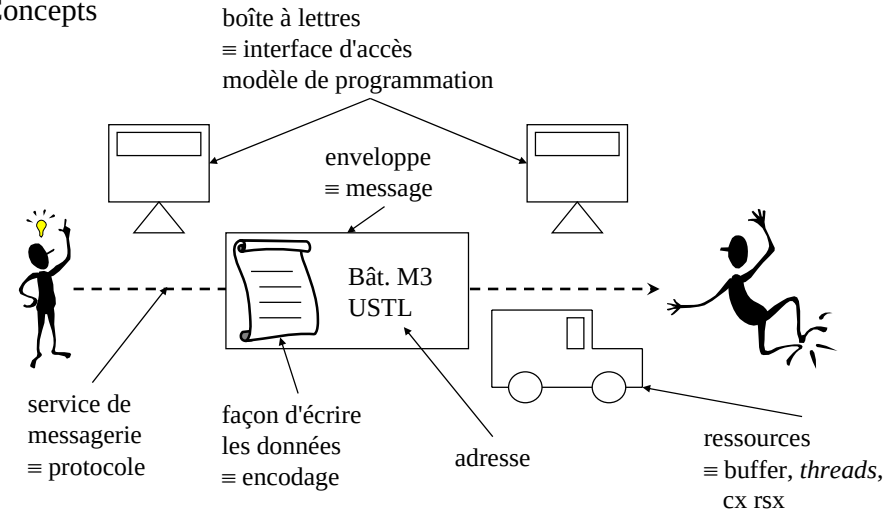


Synchrone Asynchrone Semi-synchrone

- **Synchrone** le client attend la réponse pour continuer son exécution
- **Asynchrone** le client n'attend pas de réponse et continue tout de suite
- **Semi-synchrone**
  le client continue son exécution après l'envoi et récupère le résultat ultérieurement
  - à futur explicite : le résultat est stocké dans une boîte à lettres (BAL) ; le client consulte cette BAL pour récupérer le res.
  - à futur implicite : le résultat est fourni automatiquement au client par ex. via une variable du prog. client

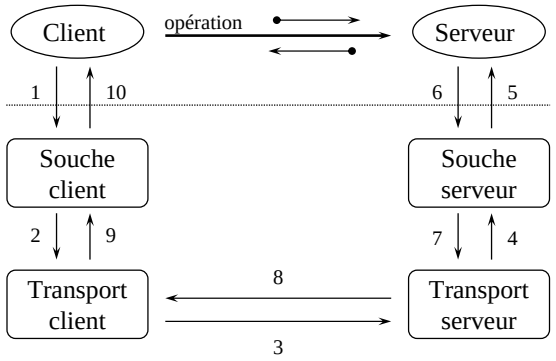
1.2 Communications

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

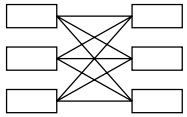
Représentation des données

Problématique

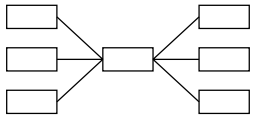
Comm. entre machines avec des formats de représentation de données ≠
→ pas le même codage (*big endian* vs *little endian*)
→ pas la même façon de stocker les types (entiers 32 bits vs 64 bits, ...)

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)



Nbreux formats pivots : ASN.1, Sun XDR, sérialisation Java, CORBA CDR, ...

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

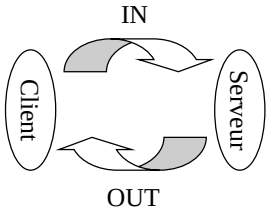
Problème en cas de mises à jour concurrentes de la valeur référencée

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

Passage de paramètres

pb : IN/OUT et OUT pas tjrs faciles à gérer dans les lang. de prog.

ex : Java

- types simples (`int`, `float`, ...) transmis par valeur
- objets transmis par référence

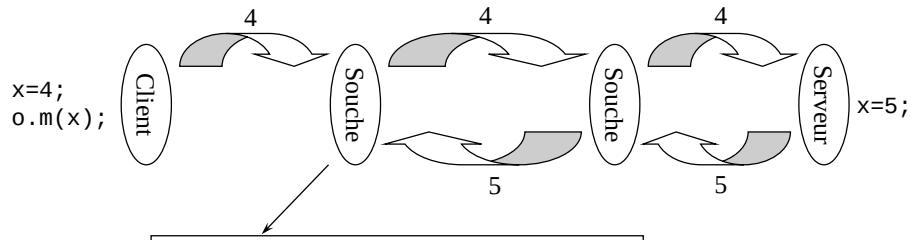
⇨ comment transmettre en IN/OUT (ou OUT) un int ?
⇨ ≡ comment implanter la copie/restauration non prévue par la VM ?
- sans modifier le code du client, ni celui du serveur
- seuls codes "maitrisables" : souches client et serveur

1.2 Communications

Passage de paramètres

pb : IN/OUT et OUT pas tjrs faciles à gérer dans les lang. de prog.

Illustration



- la souche connaît simplement la valeur 4
- elle n'a pas la ref. de x
⇨ elle ne peut pas la mettre à jour

1.2 Communications

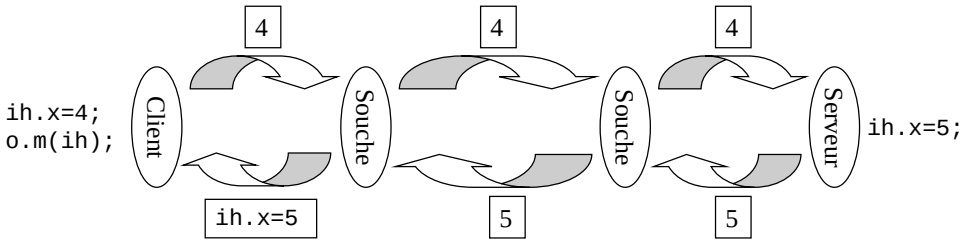
Passage de paramètres

Solution

- pas possible de ne pas modifier les codes client et serveur
- le client transmet un conteneur pour la valeur

```
class IntHolder { int x; }
```

⇨ la souche cliente peut mettre à jour la valeur contenue



1.2 Communications

Traitement des pannes

Dans la majorité des cas

- symptôme : absence de réponse
- cause **inconnue**

3 types

- client
- serveur
- réseau
  - panne de rsx local en général détectable (ex. brin Ethernet)
  - panne rsx large échelle (Internet) non signalée généralement

1.2 Communications

Traitement des pannes

Problématique

Comportement de l'interaction c/s satisfaisant en présence de ré émissions
(≈ appel de méthode local)
sans que cela soit trop lourd à gérer pour le middleware

⇨ Notion de **sémantique d'invocation**

- au moins 1 fois garantie traitement demandé exécuté au moins 1 fois [1..n]
- au plus 1 fois [0..1]
- exactement 1 fois
  - le plus satisfaisant
  - mais le plus lourd

1.2 Communications

Traitement des pannes

Au moins 1 fois

Il faut que le traitement demandé soit idempotent

i.e. plusieurs exécutions du même traitements ne doivent pas poser pb

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

Algorithme

client envoie requête
tant que résultat non reçu
    attendre délai
    renvoyer requête

1.2 Communications

Traitement des pannes

Au plus 1 fois

Algorithme

client envoie requête
si au bout d'un délai d'attente pas de résultat
alors signaler la panne au client

Exactement 1 fois (le + dur à assurer)

On ne sait pas si l'absence de réponse est due
- à une perte de message sur le réseau
- à une panne de serveur
- à un serveur très lent

⇨ ré émissions
⇨ détection des ré émissions

1.2 Communications

Traitement des pannes

Exactement 1 fois

ex : client renvoie sa requête à l'expiration du délai

- serveur planté définitivement
  en général, pas de réessai ad vitam → abandon au bout de 3 envois
- serveur planté mais redémarre
  est-ce que la req. avait quand même été reçue ?
  → si oui continuer l'exécution
  est-ce que la réponse avait été envoyée mais s'est perdue ?
  → si oui renvoyer la réponse, si non exécuter
- serveur très lent
  donc requête bien reçue et traitement commencé
  → continuer l'exécution
- réseau coupé définitivement (idem serveur planté définitivement)
- réseau coupé mais reprend (idem serveur planté mais redémarre)

1.2 Communications

Détection des pannes

Mécanisme complémentaire pour détecter des pannes
en cours d'exécution d'un traitement

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

2. C/S OO

Client/serveur orienté objet

2.1 Nommage
2.2 Sécurité d'accès
2.3 Durée de vie
2.4 Objets concurrents
2.5 Synchronisation
2.6 Migration
2.7 Réplication
2.8 Ramasse-miettes

2. C/S OO

Client/serveur orienté objet

But : coupler la notion d'objet avec les concepts client/serveur

Objectifs

- meilleure modularisation des applications
- entités logiciels plus réutilisables
- applications mieux *packagées*, portables et maintenables plus facilement

Résultats

- unité de distribution = objet
- un objet = un ensemble de traitement fournis par ses méthodes
- interaction client/serveur = invocation de méthode

⇨ Nombreux aspects techniques à intégrer pour y arriver

2.1 Nommage

Identifier les objets dans un environnement réparti

- deux objets ≠ sur le même site ou sur des sites ≠
  ne doivent pas avoir la même identité (on parle de **référence**)
- la référence sert à «retrouver» l'objet pour pouvoir invoquer ses méthodes
- généralisation de la notion de pointeur à un environnement réparti

Deux techniques principales

- un ID sans rapport avec la localisation généré par une fonction mathématique
  + une table de translation entre l'ID et la localisation de l'objet
- un ID en deux parties : son site de création + un numéro local unique

Recherche d'un objet à partir de sa référence

- annuaires de localisation centralisés ou répartis, ou diffusion
- interrogation du site contenu dans la référence + liens de poursuite

2.2 Sécurité d'accès

Gérer le partage des objets dans un environnement réparti

Pour des raisons de sécurité, l'accès à certains objets peut être restreint

- en fonction de l'identité de l'objet appelant
  ex : seuls les accès en provenance de l'intranet sont autorisés
- à partir d'une liste de contrôle d'accès
  ex : mot de passe, mécanismes de clés de session, ...

La restriction peut

- interdire complètement l'accès à l'objet
- fournir une vue «dégradée»
  ex : autoriser les méthodes qui fournissent un service de consultation
  mais interdire celles qui modifient l'état de l'objet

⇨ Nombreuses informations à ajouter aux objets

2.3 Durée de vie

Gérer la durée de vie des objets

- Objets **temporaires** : leur durée de vie correspond à celle de leur créateur

- Objets **persistants**
- l'objet reste présent en mémoire tant qu'il n'est pas détruit
- la persistance peut être limitée par la durée de vie du système
  ou s'étendre au delà des redémarrages

Plusieurs possibilités

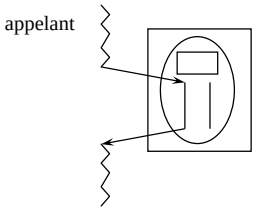
- tous les objets sont persistants ou non
- un objet est crée persistant ou non
- un objet temporaire peut devenir persistant et vice-versa
- un objet référencé par un objet persistant est persistant ou non

2.4 Objets concurrents

Deux types d'objets concurrents

Objet passif

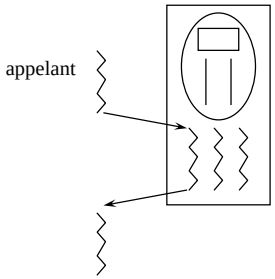
L'activité
- manipulée de façon explicite
- orthogonal à l'objet
- se «plaque» sur les méthodes



Ex :
thread Java

Objet actif

Une (+sieurs) activité dédiée
à l'objet exécutent les méthodes



2.5 Synchronisation

Restreindre la concurrence d'exécution des méthodes

En présence d'invocations de méthodes concurrentes
toutes les invocations ne sont pas forcément traitables simultanément

- soit pour des raisons strictement nécessaires (cohérence de l'application)
- soit pour assurer une qualité de service (ex. ne pas écrouler la machine)

⇨ on en met en attente ou on en rejette certaines

3 types de synchronisations

- pour 1 méthode d'un objet
  - synchronisation intra-objet
  - synchronisation comportementale
- pour plusieurs méthodes situées sur des objets ≠
  - synchronisation inter-objets

2.5 Synchronisation

Synchronisation comportementale

Fonction de l'état **des données** de l'objet

ex : Pile avec *empiler* et *depiler*
vide : *empiler*
1/2 : *empiler* et *depiler*
plein : *depiler*

Synchronisation intra-objet

Fonction de l'état **d'exécution** de l'objet

ex : Fichier avec *lire* et *écrire*
soit plusieurs *lire* simultanément
soit 1 seul *ecrire*

2.5 Synchronisation

Outils pour les synchro. intra-objet & comportementale

sémaphores	objets avec méthodes *P* et *V*
moniteurs	objet avec méthodes *synchronized* *wait* et *notify* (ex Java)
expressions de chemin	opérateurs , * | // pour spécifier les séquences valides de méthodes
gardes	chaque méthode est associée à une condition booléenne
graphes états/transitions	chaque objet est associé à un ensemble d'états chaque état est associé à un sous-ensemble de méthodes pouvant être exécutées

2.5 Synchronisation

Synchronisation inter-objets

Assurer l'exécution cohérente d'une suite d'invocations de méthodes

Exemple : un transfert entre deux objets comptes bancaires

Assurer qu'un ensemble de 2 ou +sieurs invocations de méthodes

```
compte1.depot(50);
compte2.retrait(50);
```

s'effectuent complètement ou pas du tout (+ propriétés ACID /* cf. BD */)

Problématique de la théorie de la sérialisatibilité et des moniteurs transactionnels

⇨ intégration du moniteur dans le système réparti objet avec un
- protocole de **validation** (2PC ou 3PC)
- mécanisme de **verrouillage** des ressources
- mécanisme de **détection** et de **traitement** des conflits

2.6 Migration

Déplacer des objets

• d'un espace d'adressage à un autre
• d'une zone de stockage à une autre (mémoire, disque)
• d'un **site à un autre**

Objectifs

- réduire les coûts de communication entre objets «bavards»
- répartir la charge vers les sites inutilisés
- augmenter la disponibilité d'un service en le rapprochant de ses clients

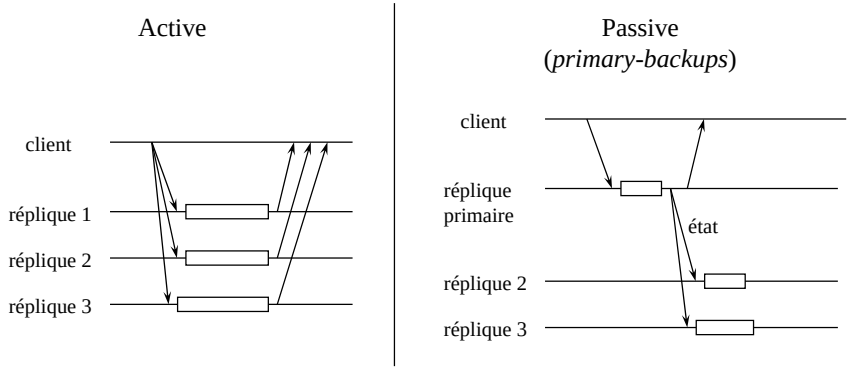
Nombreux problèmes à résoudre

- interrompre ? attendre la fin ? des traitements en cours avant de migrer
- impact de la migration sur la référence ?

2.7 Réplication

Dupliquer des objets sur +sieurs sites

Objectif principal : tolérance aux fautes



2.8 Ramasse-miettes

Détruire les objets qui ne sont référencés par aucun autre

Objectif : récupérer les ressources (mémoire, CPU, disque) inutilisées
Difficulté / aux systèmes centralisés : suivre les références entre sites distants

Deux techniques

• **comptage** de références : on associe un compteur à chaque objet
+1 lorsque la référence est acquise
-1 lorsque elle est libérée
inconvénient : utilisation mémoire + pas de gestion des cycles de réf.
avantage : simple

• **traçage** (*mark and sweep*) : parcours graphe objets à partir d'une racine
objets non marqués détruits
inconvénient : temps CPU pour le traçage
avantage : pas de modification des objets

3. Bibliographie

• R. Orfali, D. Harkey, J. Edwards. *The Essential Client/Server Survival Guide.* Wiley 1996.

• B. Meyer. *Object-Oriented Software Construction*. Prentice Hall 1997.

• A. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall 1995.

• C. Atkinson. *Object-Oriented Reuse, Concurrency and Distribution.*
Addison Wesley 1992.

• R. Balter, J.P. Banâtre, S. Krakowiak.
Constructions des systèmes d'exploitation répartis. 1991.

• *Concurrent Object Oriented Programming*. CACM 36(9). Sept. 1992.

• J.P. Briot, R. Guerraoui. *Objets pour la Programmation Parallèle et Répartie :*
Intérêts, Evolutions et Tendances. TSI 15(6):765-800. Juin 1996.