SYSTÈMES DISTRIBUÉS

Pascal Mérindol (CM)
Gabriel Frey (CM + TP)
Antoine Gallais (TD)

merindol@unistra.fr
http://www-r2.u-strasbg.fr/~merindol/

Ce cours est construit sur la base de plusieurs supports pédagogiques, en particulier les cours de Pierre Gançarski et Guillaume Latu. L'usage de ce support ne peut être qu'académique.

Plan et organisation

- La pratique : RPC, RMI, CORBA, SOAP => en TP (+ Projet avec G. Frey)
- Introduction aux Systèmes Distribués
- La "théorie" (en CM/TD) :
  - Horloges, Cohérence & Diffusion/Partage
  - (Exclusion Mutuelle &) Inter-Blocages
  - Ordonnancement
- Divers : tolérance aux pannes, intro. sécurité, notion de consensus, etc.

- Pascal Mérindol CM, Antoine Gallais TD, Gabriel Frey(CM/)TP :
  - 50% CC2, 25% CC1, 25% Projet.
  - 9*2h TP, 6*2h TD, 10*2h CM.

Systèmes distribués

- Aujourd'hui, tout est distribué :
  - Réseaux
  - Bases de données
  - Systèmes
  - Programmes, Objets

- Les systèmes distribués, c'est les systèmes concurrents :
  - Sans mémoire partagée (exclusion, verrou, …)
  - Sans homogénéité des représentations
  - Pas de garanti sur les communications

Définition

- « Système distribué » en opposition à « système centralisé »
- Système centralisé : tout est localisé sur la même machine
  - Logiciels s'exécutant sur une seule machine
  - Accès local aux ressources nécessaires (données, code, périphériques, mémoire ...)
- Système distribué (Andrew Tannenbaum)
  - Ensemble d'ordinateurs indépendants connectés en réseau et communiquant via ce réseau
  - Cet ensemble apparaît du point de vue de l'utilisateur comme une unique entité
  - « Un système distribué est un système qui m'empêche de travailler quand une machine dont je n'ai jamais entendu parler tombe en panne » *Leslie Lamport (prix Turing 2013)*

Historique

- « The Great Distributed Operating System In The Sky » (1970)
- Communication entre processus
- Objets distribués
- Internet of things (IoT)

Avantage des systèmes distribués

- Partage de ressources distantes
  - Systèmes de fichiers : utiliser ses fichiers à partir de n'importe quelle machine
  - Partage de matériels (imprimantes, capteurs, …)
- Performance : optimisation des ressources disponibles, le calcul étant distribué sur un ensemble de machines
- Fiabilité
  - Redondance : duplication des serveurs de fichiers par exemple
  - Plusieurs éléments identiques pour résister à la montée en charge
- Les systèmes distribués sont naturellement concurrent et parallèle

Inconvénients des systèmes distribués

- Possibilité de problème réseau
- Goulet d'étranglement (cas des systèmes centralisés)
  - Complexité des algorithmes totalement décentralisés
- Les systèmes distribués sont naturellement concurrent et parallèle
  - Mais besoin de synchronisation pour la coordination, l'accès aux ressources (exclusion mutuelle, verrou, …)
  - Sans mémoire partagée

Inconvénients des systèmes distribués

Hétérogénéité des :

- Machine utilisées (puissance, architecture, …)
- Des systèmes d'exploitations
- Du codage des données
  - Pour les entiers
    - Petit boutiste (little endian, par exemple SPARC) : octets de poids fort en premier
    - Grand boutiste (big endian, par exemple x86) : : octets de poids faible en premier
  - Pour les chaînes de caractères
    - ASCII ou EPCIDIC (IBM)
    - Symbole de fin de chaîne
  - Nécessité de codage/décodage

Modèles d'interactions

- Les éléments distribués interagissent, communiquent entre eux selon plusieurs modèles possibles
  - client/serveur
  - diffusion de message
  - mémoire partagée
  - Pair à pair

- Abstraction de communication basique
  - Envoi de message d'un élément vers un autre
  - Protocole correspondant à un modèle d'interaction

Modèles d'interactions

- Rôle des messages
  - Données échangées entre les éléments
    - Demande de requête
    - Résultat d'une requête
    - Donnée de toute nature
    - ...
- Gestion, contrôle des protocoles
  - Acquittement : message bien reçu
  - Synchronisation, coordination ...

Modèle client/serveur

- Deux rôles distincts
  - Client : demande que des requêtes ou des services lui soient rendus
  - Serveur : répond aux requêtes des clients
- Interaction
  - Message du client vers le serveur pour faire une requête
  - Exécution d'un traitement par le serveur pour répondre à la requête
  - Message du serveur vers le client avec le résultat de la requête
- Exemple : serveur Web
  - Client : navigateur Web de l'utilisateur
  - Requêtes : récupérer le contenu d'une page HTML générée par le serveur
- Le plus répandu

Diffusion de messages

- Deux rôles distincts
  - Émetteur : envoie des messages (ou événements) à destination de plusieurs récepteurs
    - Diffusion (broadcast) : à tous ceux qui sont présents
    - A un sous-ensemble de récepteurs : multicast
  - Récepteurs : reçoivent les messages envoyés
  - Peut être à la fois émetteur et récepteur
- Interaction
  - Émetteur envoie un message
  - Le middleware s'occupe de transmettre ce message à chaque récepteur

Diffusion de messages

- Deux modes de réception
  - Le récepteur va vérifier lui-même qu'il a reçu un message (pull)
    - Boîte aux lettres
  - Le récepteur est prévenu que le message est disponible et il lui est transmis (push)
    - Le facteur sonne à la porte pour remettre en main propre le courrier
- Particularités du modèle
  - Dépendance plus faible entre les participants
  - Pas besoin pour l'émetteur d'être directement connecté aux récepteurs ni même de savoir combien ils sont
  - Interaction de type « 1 vers N »

Mémoire partagée

- Les éléments communiquent via une mémoire partagée à l'aide d'une interface d'accès à la mémoire
  - Ajout d'une donnée à la mémoire
  - Lecture d'une donnée dans la mémoire
  - Retrait d'une donnée de la mémoire
- Le middleware gère l'accès à la mémoire pour chacun des participants
- Particularite du modèle
  - Aucun lien, aucune interaction directe entre les participants

Mémoire partagée

- Complexité du modèle : dans la gestion de la mémoire
  - On est dans un système distribué
  - Comment gérer une mémoire dans ce contexte ?
  - Plusieurs solutions
    - Déployer toute la mémoire sur un seul site
      - Accès simple mais goulot potentiel d'étranglement et fiabilité faible
  - Éclater la mémoire sur plusieurs sites
    - Avec ou sans duplication des données
    - Il faut mettre en place des algorithmes +/- complexes de gestion de mémoire distribuée

Modèle pair à pair

- Un seul rôle : pas de distinction entre les participants
  - Chaque participant est connecté avec tous les participants d'un groupe et tout le monde effectue les mêmes types d'actions
  - Pour partager des données, effectuer un calcul commun ...
- Exemples
  - Modèles d'échanges de fichiers (bittorrent...)
  - Avec parfois un mode hybride client/serveur – P2P
  - Serveur sert à connaître qui possède un fichier ou faire des recherches
  - Le mode P2P est utilisé ensuite pour les transferts
    - Chacun envoie une partie du fichier à d'autres participants
- Algorithme de consensus : choix d'une valeur parmi plusieurs
  - Chacun mesure une valeur (la même en théorie) puis l'envoie aux autres
  - Une fois reçues les valeurs de chacun, localement, chacun exécute le même algorithme sur ces valeurs pour élire la bonne valeur

Programmation distribuée

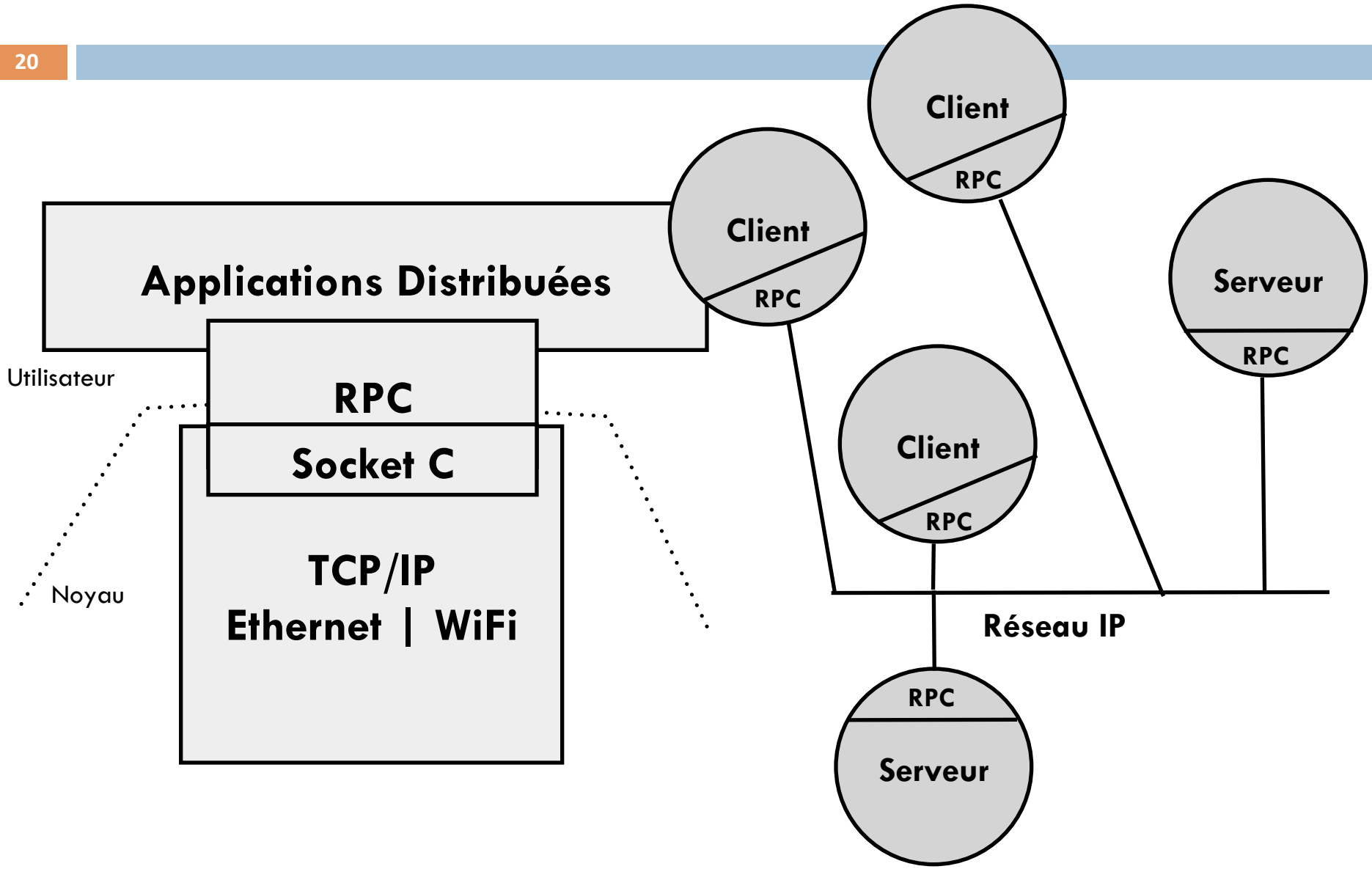
- Programmation dans un systèmes réparti
- Utilisation de librairie spécifiques (RPC, RMI, …)
- Utilisation de protocoles de communications spécifiques
- Utilisation d'intergiciel (middleware type annuaire par exemple, …)
- Hors du domaine du cours : grille, cloud

Plan du cours de programmation distribuée

- RPC
- RMI
- CORBA
- SOAP
- Extensions

RPC

Architecture RPC : vue d'ensemble



Introduction

- Primitive de base d'interaction entre éléments logiciels
  - Appel d'une procédure/fonction, exemple :

```
int resultat;
resultat = calculPuissance(2, 3);
printf('' 2 à la puissance 3 = %d\n'',
resultat);
```

- Ici `calculPuissance` est une fonction qui est appelée localement
  - Son code est intégré dans l'exécutable compilé ou chargé dynamiquement au lancement (librairie dynamique)
- Sockets TCP & UDP
  - Communication par envoi et réception de bloc de données
  - Bien plus bas niveau qu'appel de fonction

RPC : Remote Procedure Call

- Offrir outils familiers pour la conception dApplication distribuées
- Pouvoir appeler « presque » aussi facilement une fonction sur un élément distant que localement
- La fonctions appelante et appelées doivent échanger des données
- Pas d'espace mémoire accessible au deux

  ➔ Ce partage est possible : modèle déchange de message
- RPC est un cas particulier du modèle de passage de messages
- Peut être utilisée comme aussi simplement qu'un appel de procédure local, pour la communication entre processus sur des machines différentes ou même sur la même machine.
- Les appels RPC peuvent être synchrone ou asynchrone (libérer le client)
- Le serveur peut utiliser les Threads pour permettre au serveur de continuer à recevoir requêtes

Idée générale

- On différencie le coté appelant (client) du coté appelé (serveur)
- Serveur offre la possibilité à des éléments distants d'appeler une ou plusieurs fonctions à son niveau
- Client appelle localement la fonction sur un élément spécial qui relayera la demande d'appel de fonction coté serveur
- Coté serveur, un élément spécial appellera la fonction et renverra le résultat coté client
- Eléments spéciaux : **talons** (ou ***stubs***)

Conception d'applications client / serveur

- Conception orientée communication
  - Définition du protocole de communication (format et syntaxe des messages échangés par le client et le serveur)
  - Conception du serveur et du client en spécifiant comment ils réagissent aux messages échangés

- Conception orientée traitement
  - Construction d'une application conventionnelle dans un environnement mono-machine
  - Subdivision de l'application en plusieurs modules pouvant s'exécuter sur différentes machines

Conception orientée communication

- Problèmes
  - Gestion des formats de messages et données par l'utilisateur (hétérogénéité)
  - Empaquettage / désempaquettage des messages (marshalling)
  - Le modèle est souvent asynchrone, ce qui rend la gestion des erreurs plus complexe
  - Le modèle n'est pas naturel pour la plupart des programmeurs

    Communication explicite et non transparente

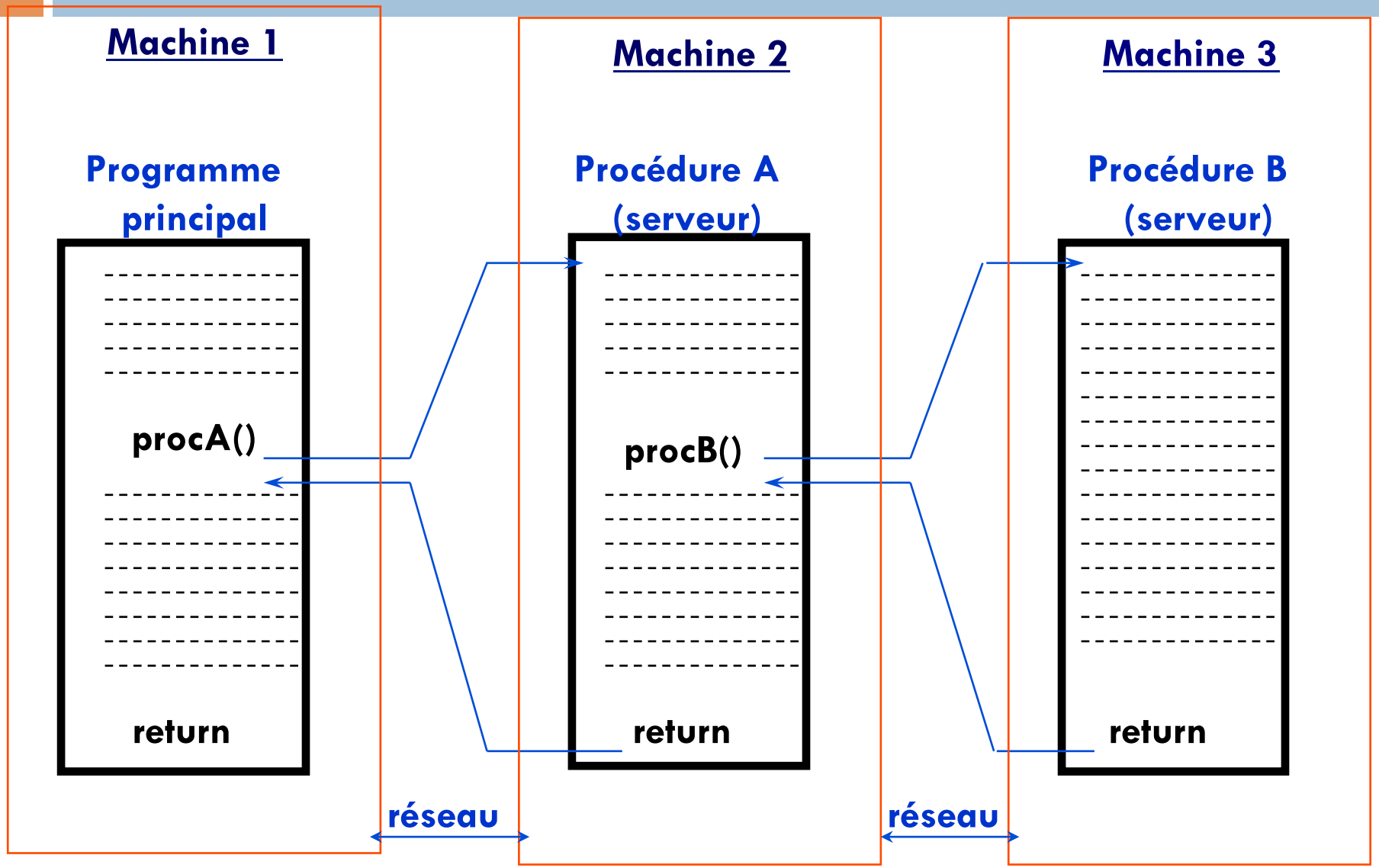
Conception orientée application

- Objectif :

  Démarche de conception des applications centralisées

- *Remote Procedure Call* (*RPC*)
  - Introduit par Birrell & Nelson (1984)
  - Garder la sémantique de LPC (*Local Procedure Call*)
  - Fonctionnement synchrone
  - Communication transparente entre le client et le serveur

RPC : principe



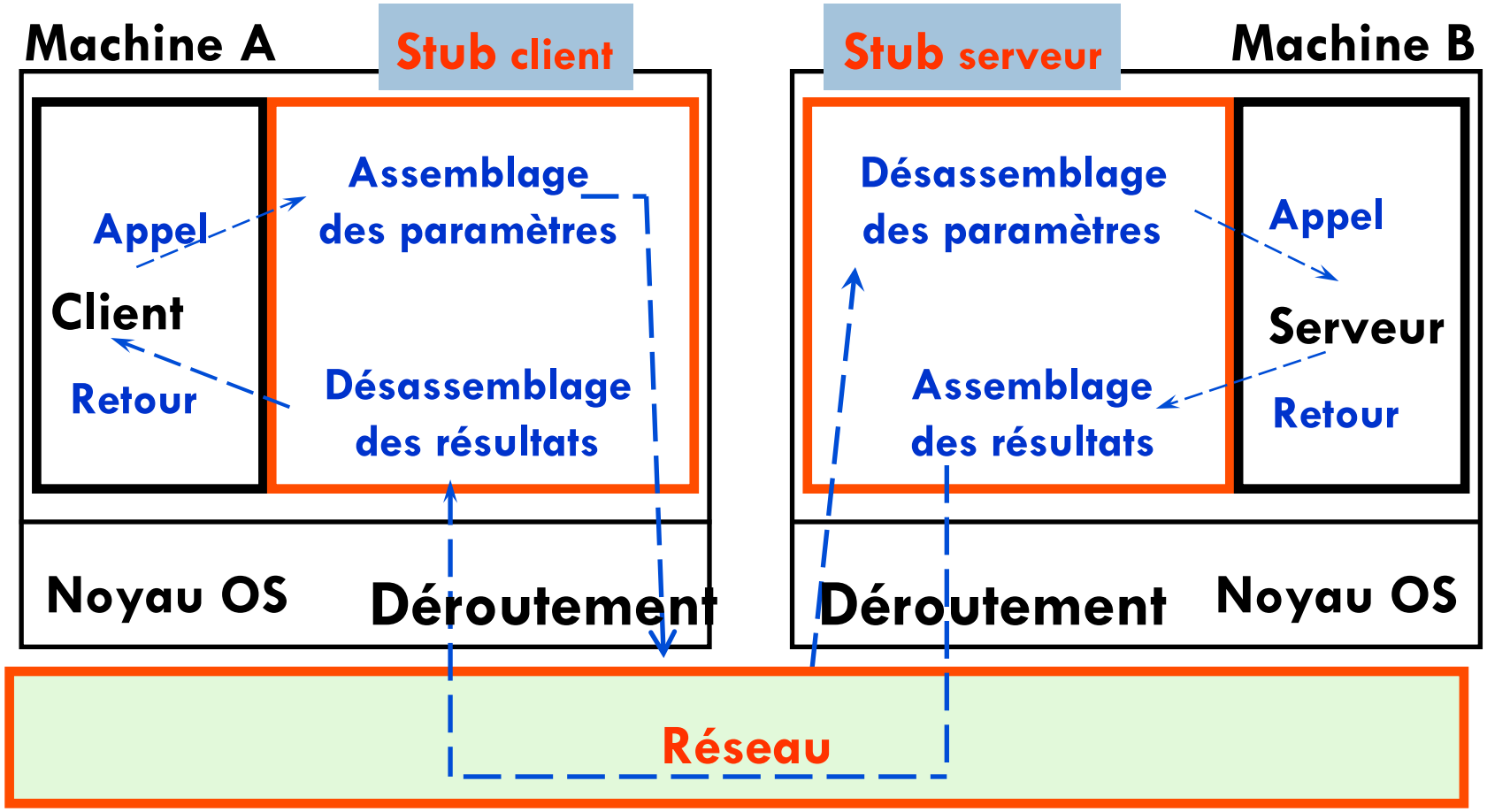
Modèle LPC

- Notion de contexte et de pile d'exécution
- Déroulement
  - Empilement
    - paramètres (valeurs et références)
    - adresse de retour
    - variables locales
  - Exécution du code de la procédure
- Passage de paramètres
  - par valeur, par référence, par copie/restauration

Le modèle RPC

- Même sémantique que le modèle LPC
- Position par rapport à OSI
  - Couche *session*
- Communication synchrone et transparente
  - Utilisation transparente de sockets en mode *connecté*
- Différentes implémentations
  - *DCE-RPC* de l'*Open Software Foundation* (*OSF*)
  - *ONC-RPC* de *Sun* (*NFS*, *NIS*, etc.)

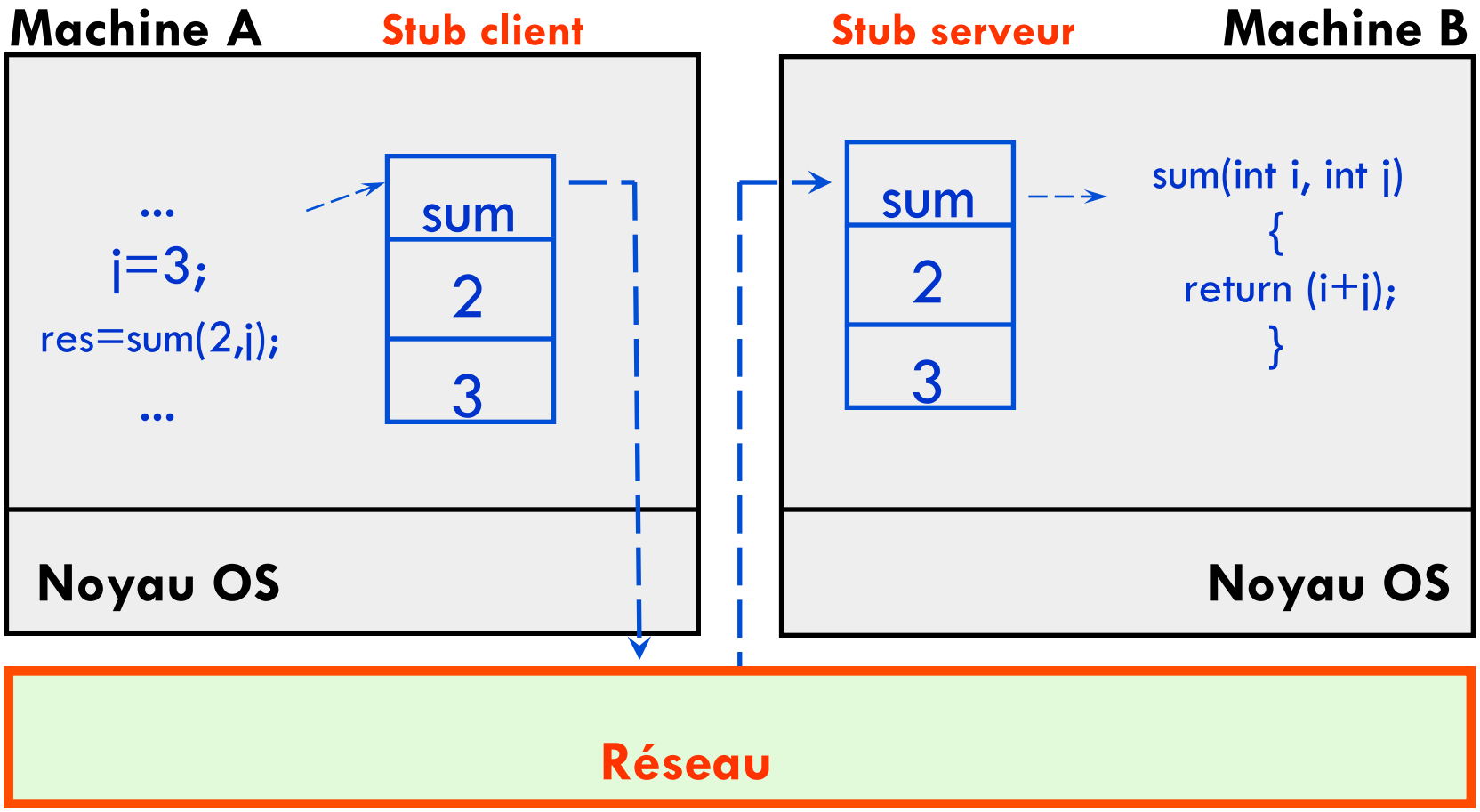
Fonctionnement



Problèmes

- Passage de paramètres
- Identification ou nommage
  - Localisation (adresse) du serveur
  - Procédure au sein d'un serveur
- Sémantique des RPC en présence d'échecs

Exemple


Machine A
Stub client
Stub serveur
Machine B
...
j=3;
res=sum(2,j);
...
sum
2
3
sum
2
3
sum(int i, int j)
{
return (i+j);
}
Noyau OS
Noyau OS
Réseau

Passage de paramètres

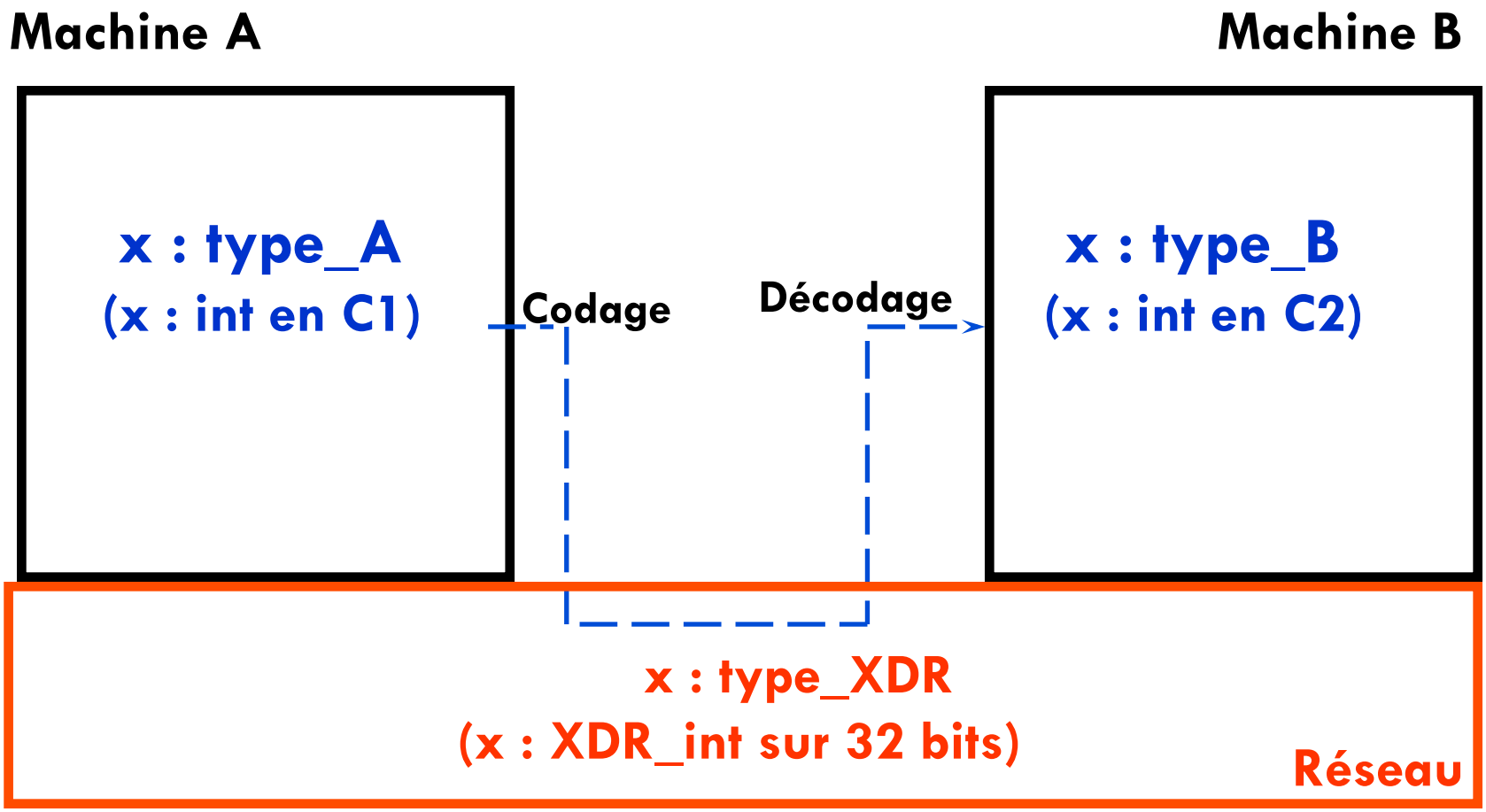
- Passage de paramètres par référence impossible
  - Passage par valeur
  - Passage par copie/restauration
- Passage de structures dynamiques (tableaux de taille variable, listes, arbres, graphes, ...)
- Hétérogénéité des machines
  - *Byte-ordering* : ordre de stockage des octets différent
  - Représentation des arguments : codage des caractères, virgule flottante, etc.

Problème d'hétérogénéité

- Deux solutions possibles
  - Codage/décodage de *chaque type* de donnée de *toute architecture à toute autre architecture*
  - Format universel intermédiaire (XDR, CDR, etc.)

- Solution de Sun Microsystems
  - Format e*Xternal Data Representation* ou *XDR*
  - Librairie XDR (types de données XDR + primitives de codage/décodage pour chaque type)
  - rpc/xdr.h

XDR : Principe



XDR : Codage/Décodage

- L'encodage XDR des données contient uniquement les données représentées mais aucune information sur leur type
  - Si une application utilise un entier de 32 bits le résultat de l'encodage occupera exactement 32 bits et rien n'indiquera qu'il s'agit d'un type entier
  - Le client et le serveur doivent alors s'entendre sur le format exact des données qu'ils échangent

Problèmes

- Identification ou nommage
  - Localisation (adresse) du serveur
  - Procédure au sein d'un serveur

Nommage ou *binding*

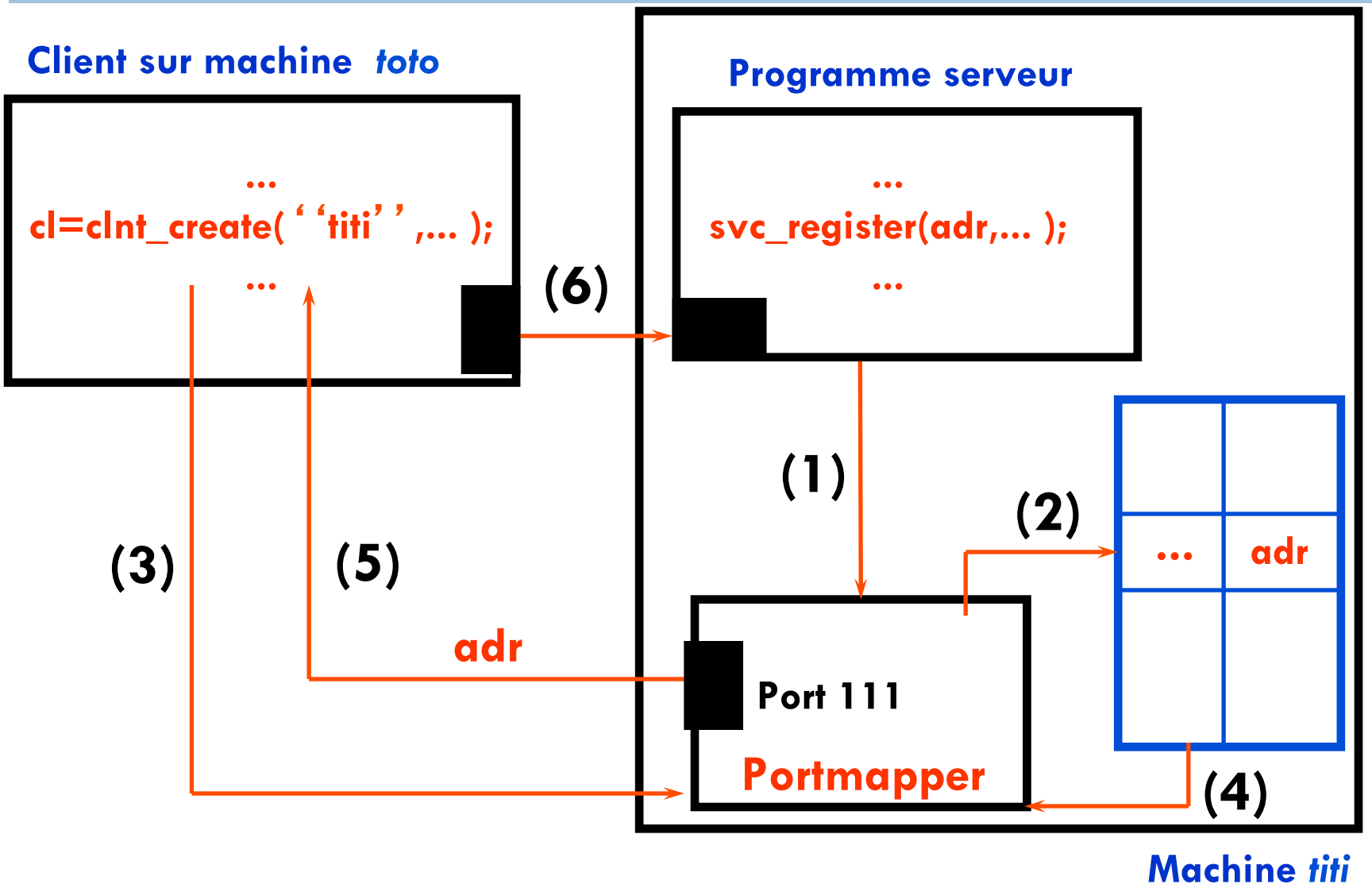
- Comment un client fait-il pour trouver le serveur ?
  - Solution statique : écrit son adresse dans son code
  - Problème : solution rigide (le serveur peut changer d'adresse)

- Solution robuste : nommage dynamique (*dynamic binding*)
  - Gestionnaire de noms : intermédiaire entre le client et le serveur
  - ***Portmapper*** : processus daemon s'exécutant sur le serveur

Nommage dynamique

- Le serveur s'enregistre auprès du *portmapper*
  - son nom (ou numéro)
  - sa version (car il peut en avoir plusieurs)
  - son adresse (IP, numéro de port) ou *handle* sur 32 bits
  - ...
- Le *portmapper* enregistre ces informations dans sa table de liaisons
- Le client demande l'adresse du serveur au *portmapper* en lui passant le nom et la version du serveur et le protocole de communication à utiliser

Nommage dynamique



Procédure au sein d'un serveur

- Identification
  - Nom (ou numéro) de son programme (PROGNUM)
  - Version du programme (VERSNUM)
  - Nom ou numéro de la procédure (PROCNUM)

Problèmes

Sémantique des RPC en présence d'échecs

Différentes classes d'échecs

- Le client est incapable de localiser le serveur
  - Le serveur est en panne
  - L'interface du serveur a changé
  - Solutions : retourner -1 , exceptions, signaux
- La requête du client est perdue
  - Temporisation et ré-émission de la requête
- La réponse du serveur est perdue
  - Temporisation et ré-émission de la requête par le client

Différentes classes d'échecs

- Problème : risque de ré-exécuter la requête plusieurs fois (opération bancaire !!!!)
- Solution : un bit dans l'en-tête du message indiquant s'il s'agit d'une transmission ou retransmission

- Le serveur tombe en panne après réception d'une requête
  - (i) Après exécution de la requête et envoi de réponse
  - (ii) Après exécution de la requête, avant l'envoi de la réponse
  - (iii) Pendant l'exécution de la requête
  - Comment le client fait-il la différence entre (ii) et (iii) ?

Différentes classes d'échecs

- Trois écoles de pensée (sémantiques)
  - Sémantique *une fois au moins* : le client ré-émet jusqu'à avoir une réponse (RPC exécuté au moins une fois)
  - Sémantique *une fois au plus* : le client abandonne et renvoie un message d'erreur (RPC exécuté au plus une fois)
  - Sémantique *ne rien garantir* : le client n'a aucune aide (RPC exécuté de 0 à plusieurs fois)

- Le client tombe en panne après envoi d'une requête
  - Requête appelée *orphelin*. Que doit-on en faire ?

Différentes classes d'échecs

Solutions de *Nelson*

- Extermination : le client utilise un journal de trace et tue les orphelins : solution coûteuse en espace et complexe (orphelins d' orphelins…)
- Réincarnation : définition de périodes d'activité incrémentale du client. Après une panne, il diffuse un message indiquant une nouvelle période. Ses orphelins sont détruits.
- Réincarnation douce : variante de la précédente. Un orphelin est détruit seulement si son propriétaire est introuvable.
- Expiration : chaque RPC dispose d'un quantum *q* de temps pour s'exécuter. Il est détruit au bout de ce quantum.
  - Pb : valeur de q ?
- Problème de ces solutions : si l'orphelin détruit a verrouillé des ressources ?

Composants d'une application RPC

- Client
  - Localiser le serveur et s'y connecter
  - Faire des appels RPC (requêtes de service)
    - Emballage des paramètres
    - Soumission de la requête
    - Désemballage des résultats
- Serveur
  - S'enregistrer auprès du *portmapper*
  - Attendre les requêtes du client et les traiter
    - Désemballage des paramètres
    - Appel local du service (procédure) demandé(e)
    - Emballage des résultats

Stub client
+
Primitives XDR

Stub serveur
+
Primitives XDR

Besoins

- Le client a besoin de connaître le nom symbolique du serveur (numprog,numver) et ses procédures
  - Publication dans un **contrat** (fichier .x)
  - Langage *RPCL*
- Le serveur a besoin des implémentations des procédures pour pouvoir les appeler
  - Fichier contenant les implémentations des procédures
- Serveur et client ont besoin des stubs de communication
  - Communication transparente ==> génération automatique des stubs et des fonctions XDR de conversion de paramètres
  - *RPCGEN* (compilateur de contrat) + *Runtime RPC*