Remote Procedure Call

L. Deruelle
Hinde Bouziane

Modèle et fonctionnement des RPC

Approches de conception d'applications C/S

- 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

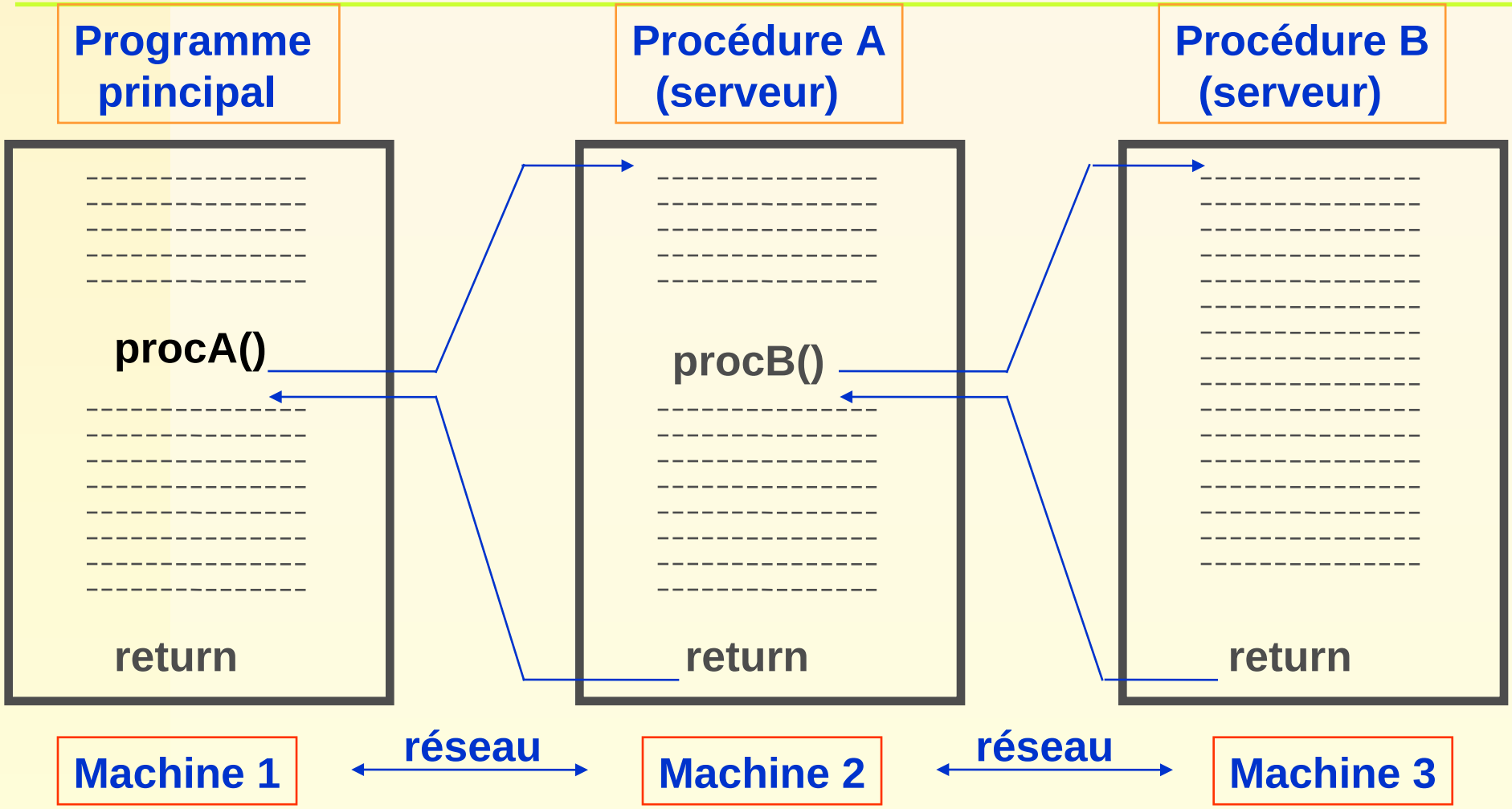
Conception orientée communication

- Problèmes (principaux)
  - Gestion des formats de messages et données par l'utilisateur
  - Gestion de l'hétérogénéité des machines
  - Le modèle n'est pas simple ni intuitif pour la plupart des programmeurs
    - Communication explicite et non transparente

Conception orientée traitement

- Objectif : Garder la démarche de conception des applications centralisées
- Appel de procédure à distance ou *Remote Procedure Call* (*RPC*)
  - Introduit par Birrell & Nelson (1984)
  - Garder la sémantique de l'appel de procédure local ou *Local Procedure Call* (*LPC*)
  - Fonctionnement synchrone
  - Communication transparente entre le client et le serveur

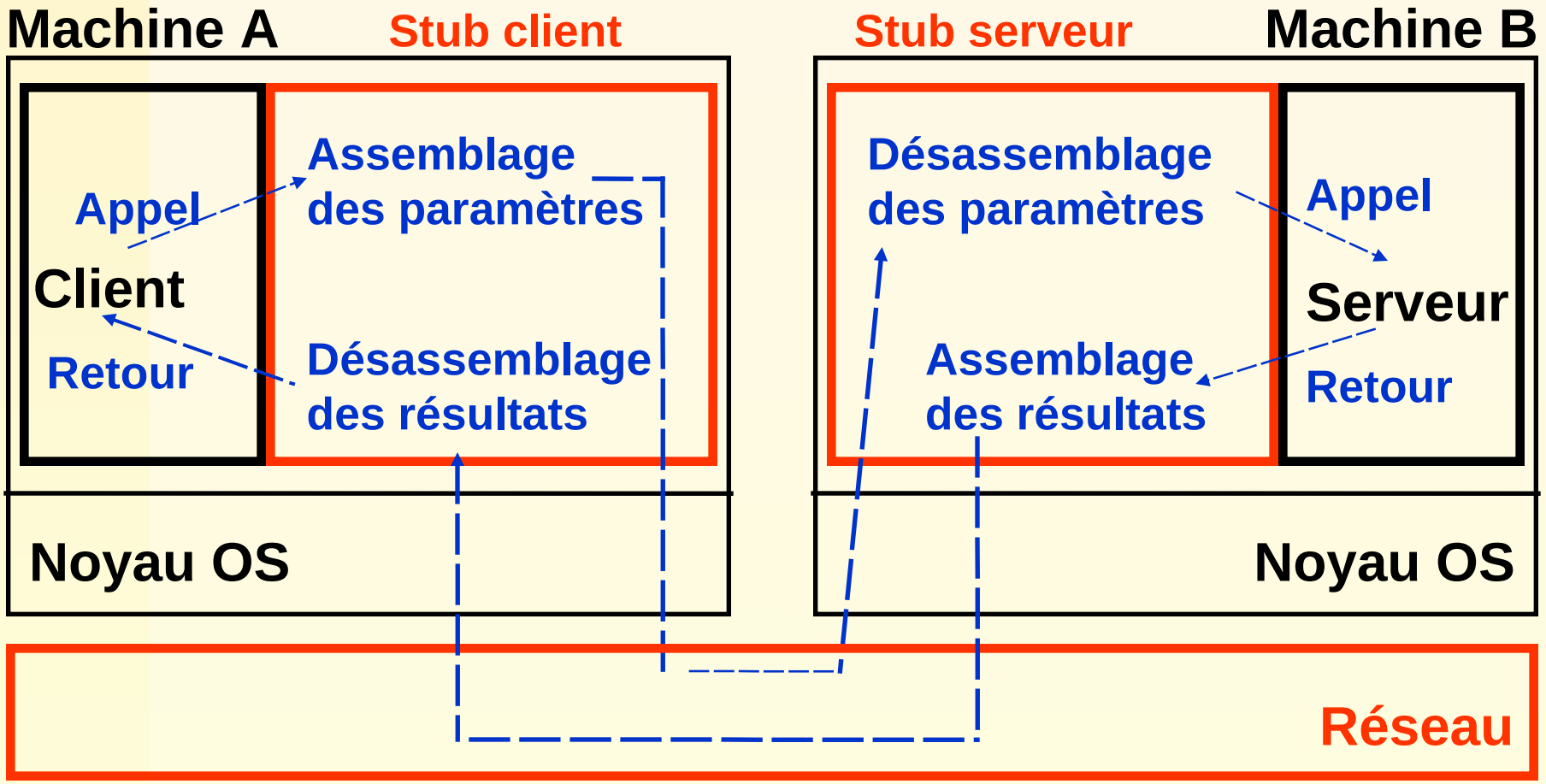
RPC : principe



Le modèle RPC

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

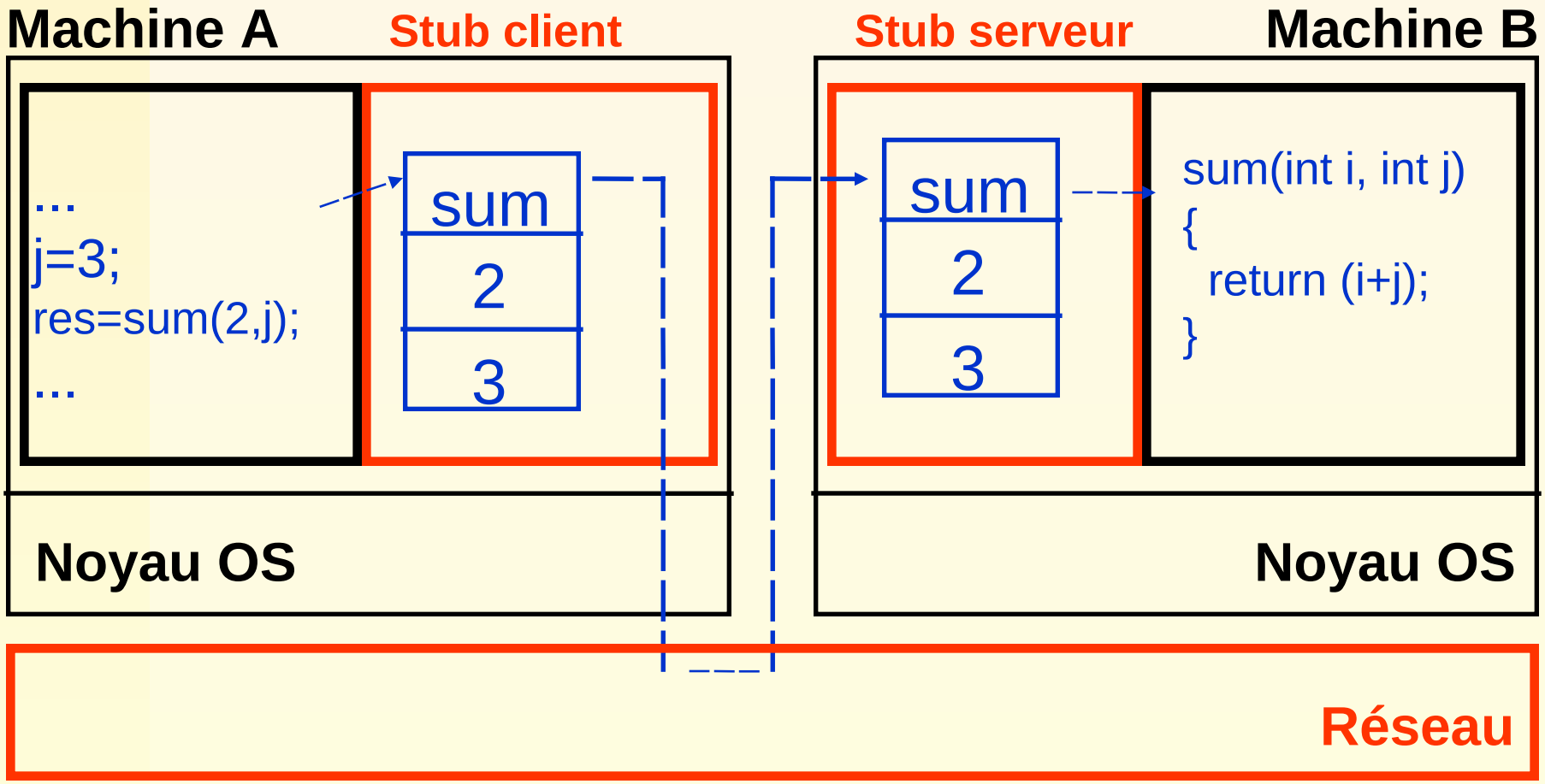
Fonctionnement



Fonctionnement des RPCs

- **Passage de paramètres**
- Identification
  - Localisation (adresse) du serveur
  - Procédure au sein d'un serveur
- Gestion des erreurs dans le RPC

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

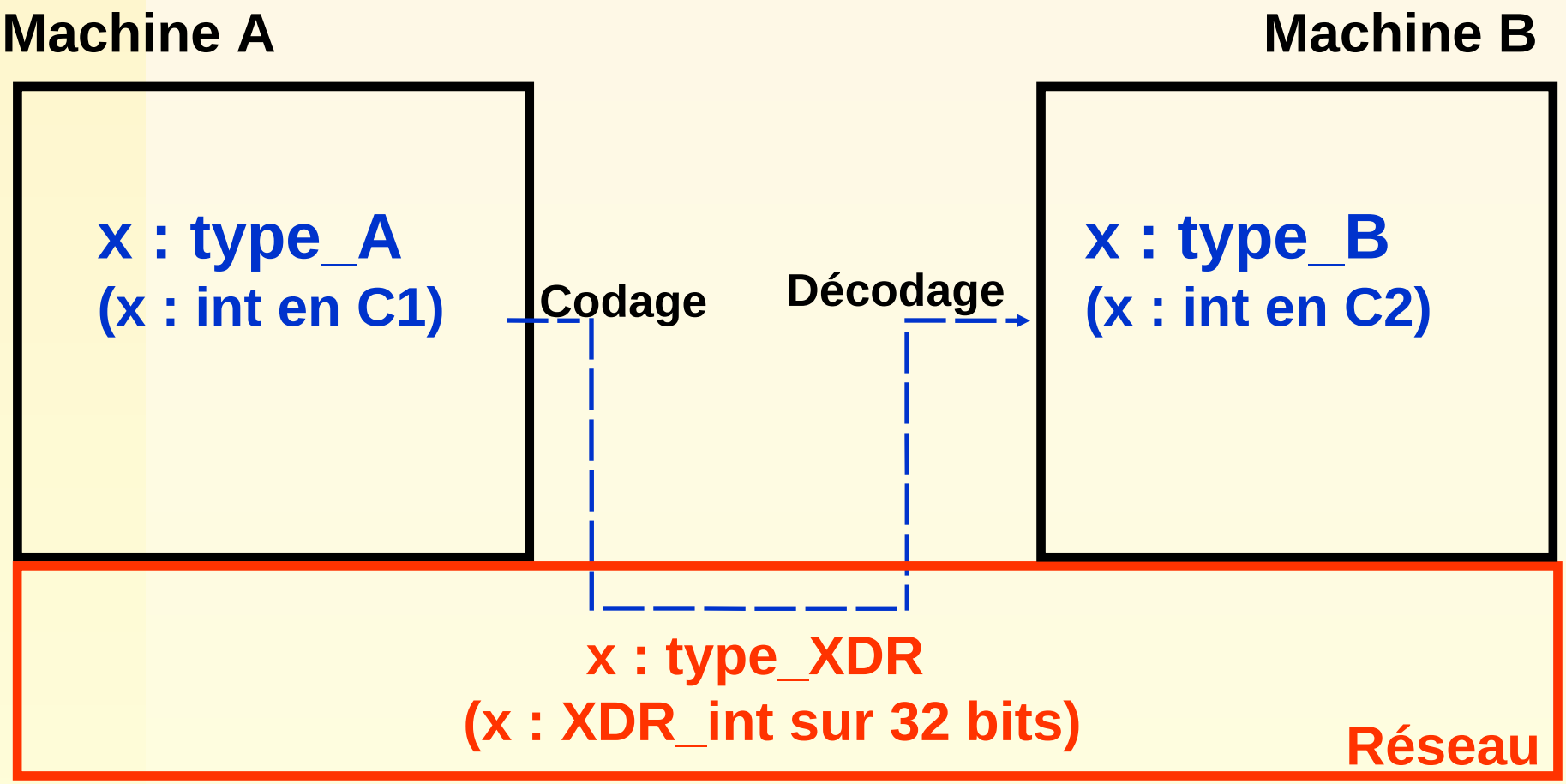
Passage de paramètres

- Passage de paramètres par référence impossible
  - Passage par valeur
  - Passage par copie
- Passage de structures dynamiques (tableaux de taille variable, listes, arbres, graphes, etc.)
- Hétérogénéité des machines
  - *Byte-ordering* : ordre de stockage des octets différent (*little endian (Intel)*, *big endian (Sun-SPARC)*)
  - Représentation des arguments : codes des caractères, C1 et C2, 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 *eXternal 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 (1)

- 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

Primitives XDR : Exemple

```
XDR  *xdrs;          /* Pointeur vers un buffer XDR */
char  buf[BUFSIZE];    /* Buffer pour recevoir les données encodées */
xdr_mem_create  (xdrs, buf, BUFSIZE, XDR_ENCODE);

/* Un buffer stream est créé pour encoder les données
 * chaque appel à une fonction de codage va placer le résultat
 * à la fin du buffer stream; le pointeur sera mis à jour */

int i;
...
i=100;
xdr_int(xdrs, &i);   /* code l'entier i et le pace en fin de buffer stream */
```

Rq : Le programme récepteur décodera les données : xdr_mem_create (... , XDR_DECODE)

Autres primitives XDR

Fonction	arguments	type de donnée converti
xdr_bool	xdrs, ptrbool	booléen
xdr_bytes	xdrs, ptrstr, strsize, maxsize	chaîne de caractères
xdr_char	xdrs, ptrchar	caractère
xdr_double	xdrs, ptrdouble	virgule flot., double précision
xdr_enum	xdrs, ptrint	type énuméré
xdr_float	xdrs, ptrfloat	virgule flot. simple précision
xdr_int	xdrs, ip	entier 32 bits
xdr_long	xdrs, ptrlong	entier 64 bits
xdr_opaque	xdrs, ptrchar, count,	données non converties
xdr_pointer	xdrs, ptrobj	pointeur
xdr_short	xdrs, ptrshort	entier 16 bits
xdr_string	xdrs, ptrstr, maxsize	chaîne de caractères
xdr_u_char	xdrs, ptruchar	entier 8 bits non signé
xdr_u_int	xdrs, ptrint	entier 32 bits non signé

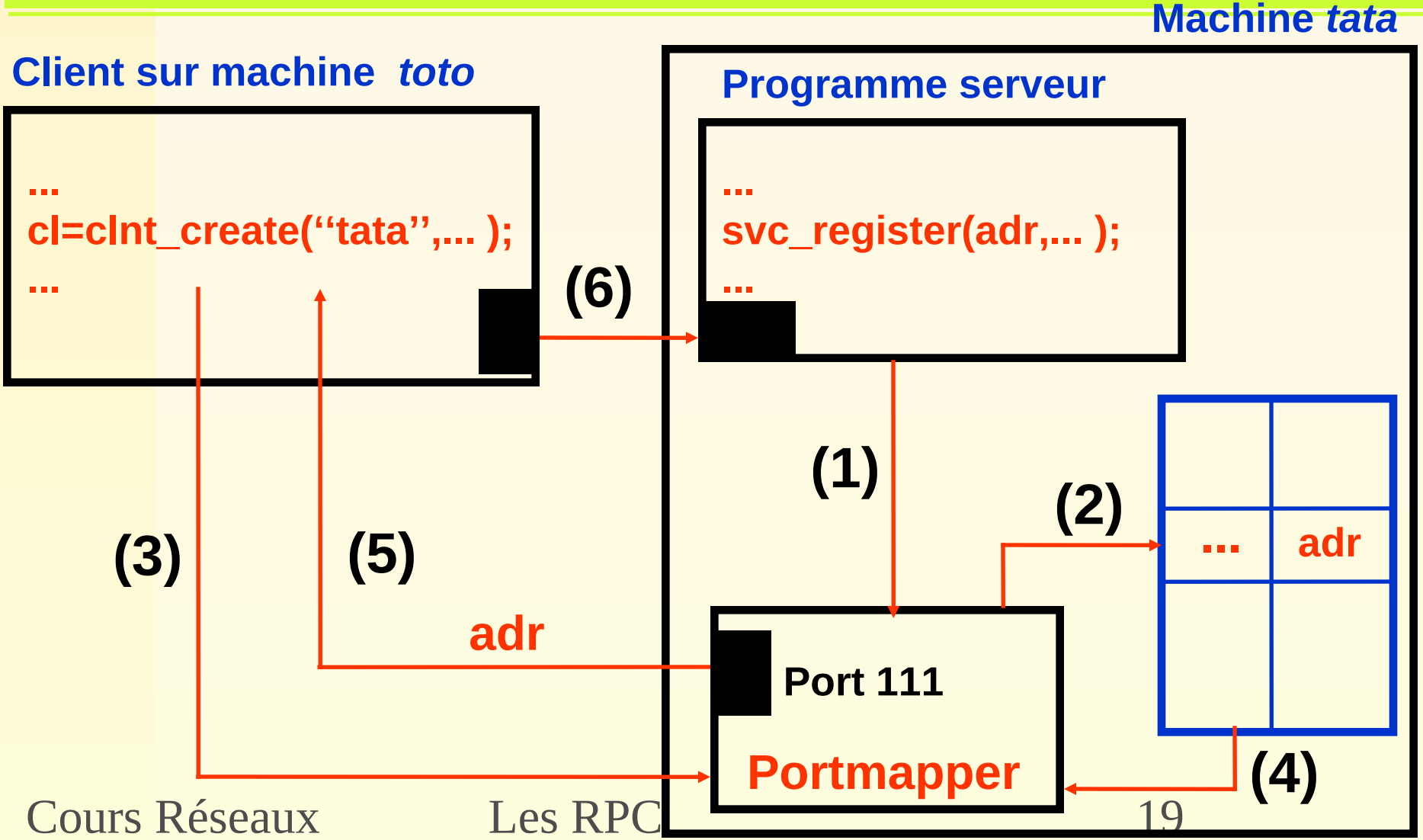
Fonctionnement des RPCs

- Passage de paramètres
- **Identification**
  - **Localisation (adresse) du serveur**
  - **Procédure au sein d'un serveur**
- Gestion des erreurs dans le RPC

Nommage ou *binding*

- Comment un client fait-il pour trouver le serveur ?
  - Solution statique : écrit son adresse dans son code
  - Solution rigide (et si le serveur change 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 (1)



Nommage dynamique (2)

- 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)
  - etc.
- 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

Adresses/numéro de quelques serveurs

Nom	identifieur	description
portmap	100000	port mapper
rstat	100001	rstat, rup, perfmeter
ruserd	100002	remote users
nfs	100003	Network File System
ypserv	100004	Yellow pages (NIS)
mountd	100005	mount, showmount
dbxd	100006	debugger
ypbind	100007	NIS binder
etherstatd	100010	Ethernet sniffer
pcnfs	150001	NFS for PC

0x20000000 - 0x3FFFFFFF : numéros libres
0x00000000 - 0x1FFFFFFF : serveurs officiels

Procédure au sein d'un serveur

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

Fonctionnement des RPCs

- Passage de paramètres
- Identification ou nommage
  - Localisation (adresse) du serveur
  - Procédure au sein d'un serveur
- **Gestion des erreurs dans le RPC**

Différentes classes d'échecs (1)

- 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 (2)

- 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 (4)

- 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 ?

Méthodologie de développement à base de RPC

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

Stub client + Primitives XDR

- 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 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 implémente 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*

Le langage RPCL

- Constantes
  - *Const* nom=valeur;
- Enumérations et structures
  - Idem C
- Unions
  - Idem structures avec variantes en Pascal
- Tableaux
  - <type_de_base> <*nom_tab*> <TAILLE> (*ex* : int toto<MAXSIZE>)
- Définition de types
  - typedef

Forme générale du contrat

```
/* Définitions de types utilisateur */
...
program «nomprog» {
  version «nomversion1» {
     «typeres1» PROC1(«param1») = 1;
      ...
     «typeresn» PROCn(«paramn») = n;
  } = 1;
   ...
   version «nomversionm» {
     ...
   } = m;
} = «numéro_du_programme»;
```

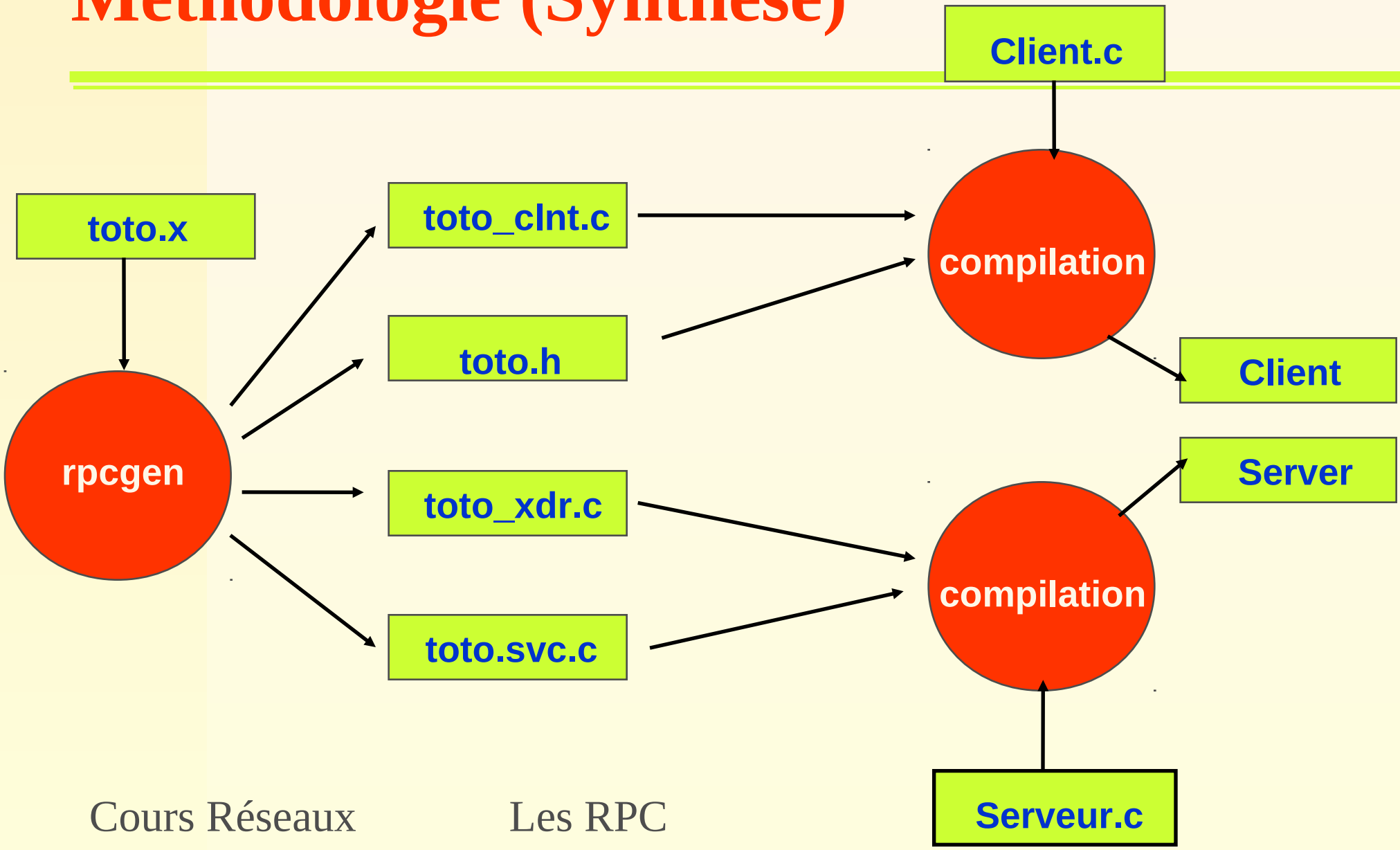
Méthodologie de développement

- Ecrire le contrat *toto.x* dans le langage RPCL
- Compiler le contrat avec RPCGEN
  - toto.h : déclarations des constantes et types utilisés dans le code généré pour le client et le serveur
  - toto_xdr.c : procédures XDR utilisés par le client et le serveur pour encoder/décoder les arguments,
  - toto_clnt.c : procédure *stub* côté client
  - toto_svc.c : procédure *stub* côté serveur
- Ecrire le client (client.c) et le serveur (serveur.c)
  - Serveur.c : implémentation de l'ensemble des procédures

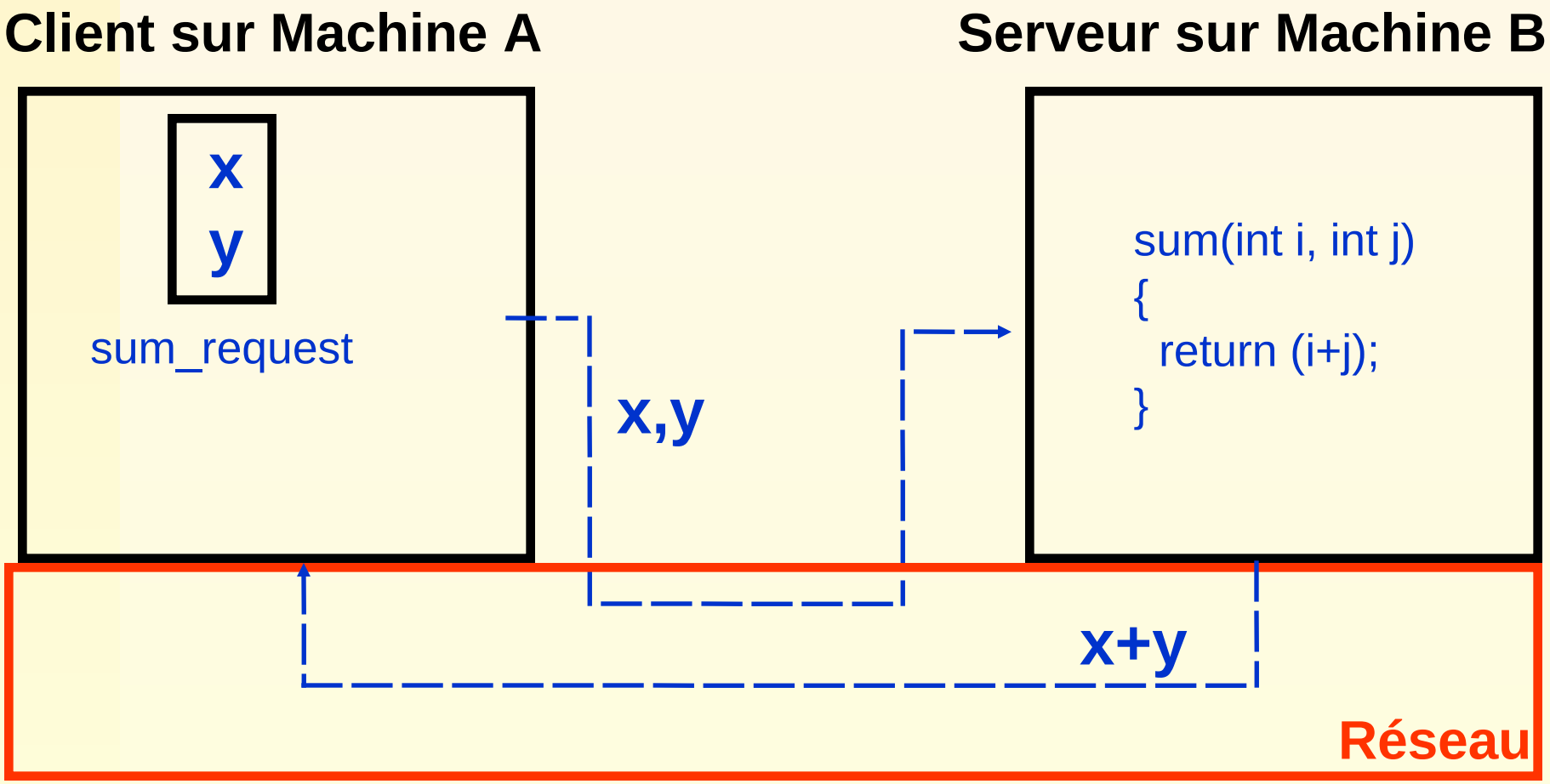
Compilation et exécution

- Compilation
  - (g)cc serveur.c toto_svc.c toto_xdr.c -o Serveur
  - (g)cc client.c toto_clnt.c toto_xdr.c -o Client
- Exécution
  - rsh «host» Serveur ou ssh « host »
  - Client «host» «liste d'arguments»

Méthodologie (Synthèse)



Exemple : sum


Client sur Machine A
Serveur sur Machine B
x
y
sum_request
x,y
sum(int i, int j)
{
return (i+j);
}
x+y
Réseau

Exemple de contrat (sum.x)

```
struct sum_request {
  int x; int y;
};

program SUMPROG {
  version SUMVERS {
     int SUM(sum_request) = 1;
  } = 1; /* numéro de version du programme */
} = 0x2000009a; /* numéro du programme */
```