RE52

RPC

Remote Processus Call

Appel de procédure distante

Approches de conception 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

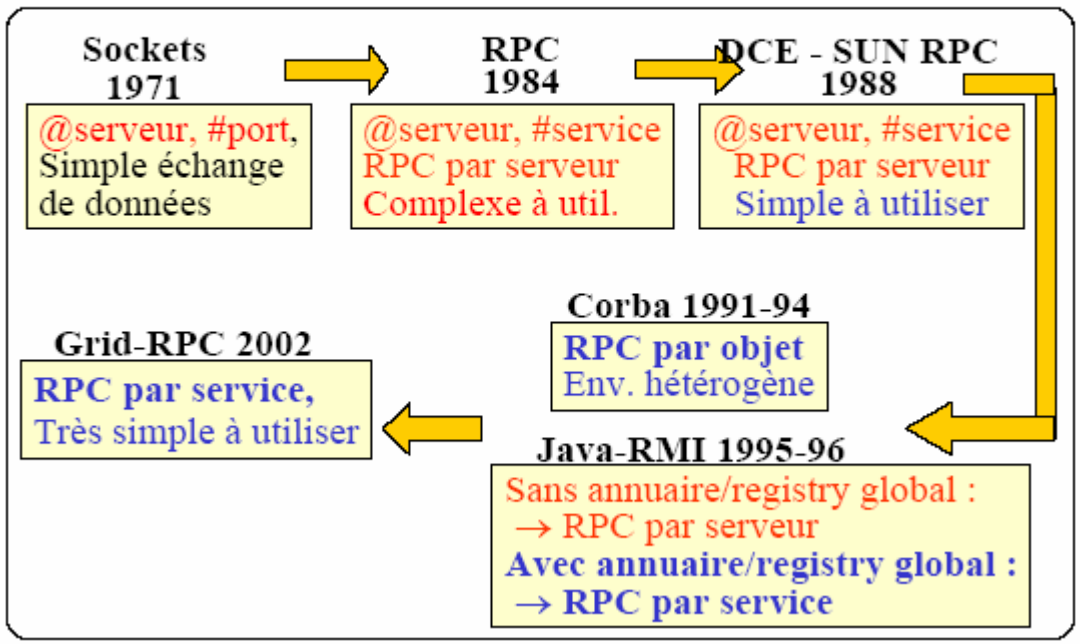
- Problèmes
  - Gestion des formats de messages et données par l'utilisateur (hétérogénéité)
  - Empaquettage/désempaquettage des messages
  - Le modèle est souvent asynchrone, ce qui rend la gestion des erreurs plus complexe
  - Le modèle n'est pas naturel pour la plupart des programmeurs

Conception orientée application

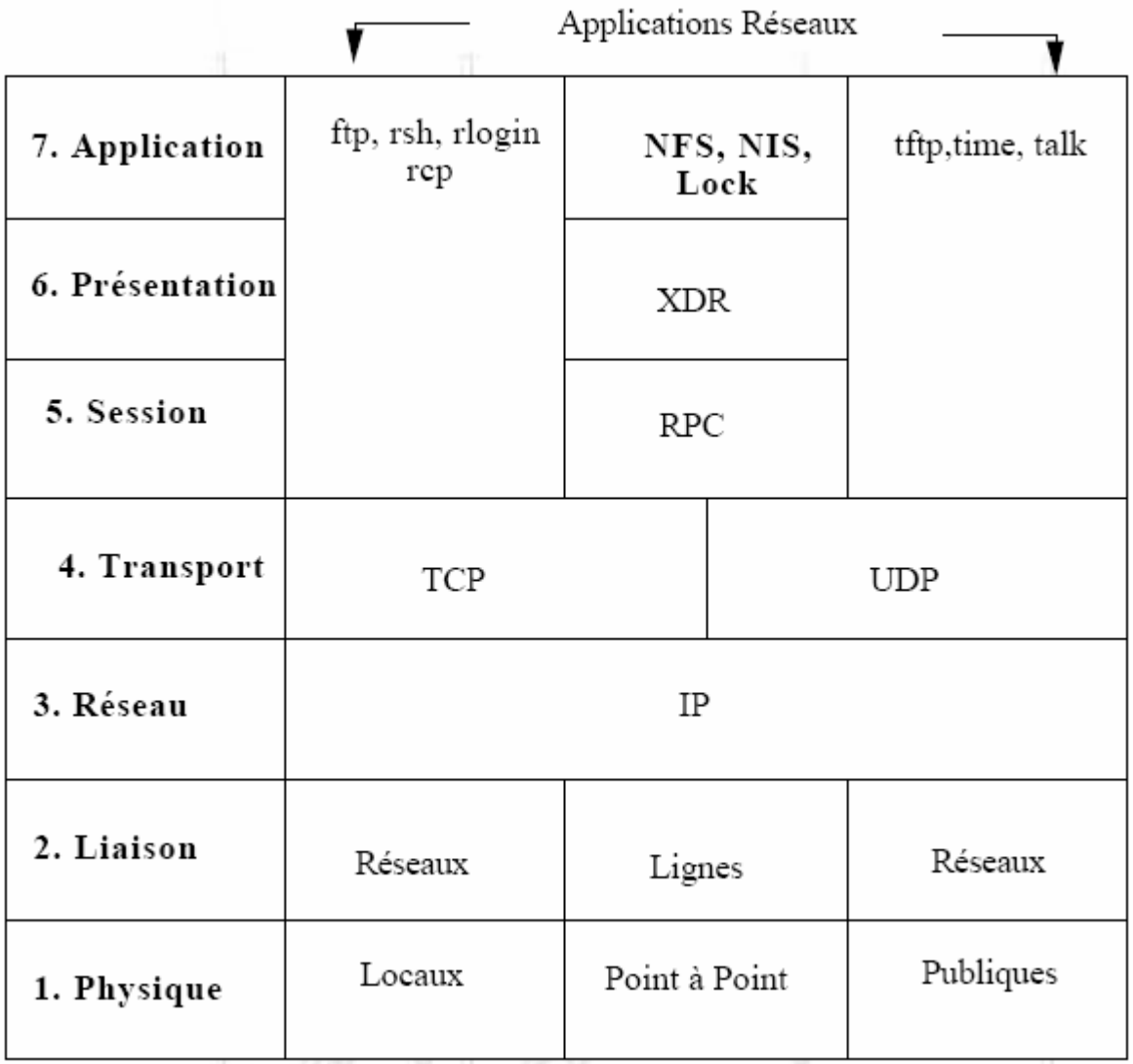
- Objectif : Garder la démarche de conception des applications centralisées
  - n Appel de procédure à distance ou *Remote Procedure Call* (*RPC*)
  - Introduit par Birrell & Nelson (1984)
  - Garder la sémantique de l'appel de procédure local ou *Local Procedure Call* (*LPC*)
  - Fonctionnement synchrone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

Historique

Bilan des modèles et outils de prog.



Environnement Internet / Modèle ISO



Introduction

- L’approche Client-Serveur
  - Mode de fonctionnement dissymétrique dans les systèmes répartis où
    - un ensemble de clients transmettent des requêtes,
    - ces requêtes sont exécutées par des serveurs qui en retournent les résultats aux clients.

- La communication a la forme "question-réponse", toujours à l’initiative des clients, jamais à celle du serveur
  - on peut utiliser l’échange de 2 messages en mode asynchrone (service offert par le niveau Transport).

Introduction

- L’approche Client-Serveur en RPC
  - **Mode de réalisation d’une interaction Client-Serveur**
    - l’opération à réaliser est présentée sous la forme d’une **procédure** que le **client** peut faire exécuter à distance par un autre site, **le serveur**.
  - **Service basique (API de RPC)**
    - côté client : invoque, génère l’appel distant et récupère le résultat.

      invoque (id_client, id_serveur, nom_procédure, paramètres)
    - côté serveur : reçoit, traite un appel et répond.

      traite (id_client, id_serveur, nom_procédure, paramètres)
  - **Service intégré objet**
    - côté client : on invoque une procédure localisée à distance.

      ref_objet_serveur.nom_procédure (paramètres)
    - côté serveur : on déploie l’objet qui implante la procédure.

      method nom_procédure (paramètres)

Introduction

- Avantages majeurs de l'approche C/Serveur en RPC

1. S'affranchir du côté basique des communications en mode message
   - ne pas avoir à programmer des échanges au niveau réseau en mode message.

2. Utiliser une structure familière : RPC
   - problème : ne pas ignorer les différences centralisé/réparti.

3. Disposer de mécanismes modernes de programmation
   - Vision modulaire des applications réparties (en approche objets répartis ou par composants sur étagère).

Introduction

- Les implantations des RPCs
  - Approches à RPC traditionnelles
    - SUN ONC/RPC
      Open Network Computing / Remote Procedure Call
    - OSF DCE
      Open Software Foundation - Distributed Computing Environnment
  - Approches à RPC intégrées dans les systèmes d'objets répartis
    - OMG CORBA
      Object Management Goup – Common Object Request Broker Achitecture
    - SUN Java RMI
      Remote Method Invocation
    - Microsoft DCOM
      Distributed Component Object Model

Introduction

- Les implantations des RPCs
  - Approches à RPC intégrées dans les systèmes de composants
    - SUN J2EE EJB

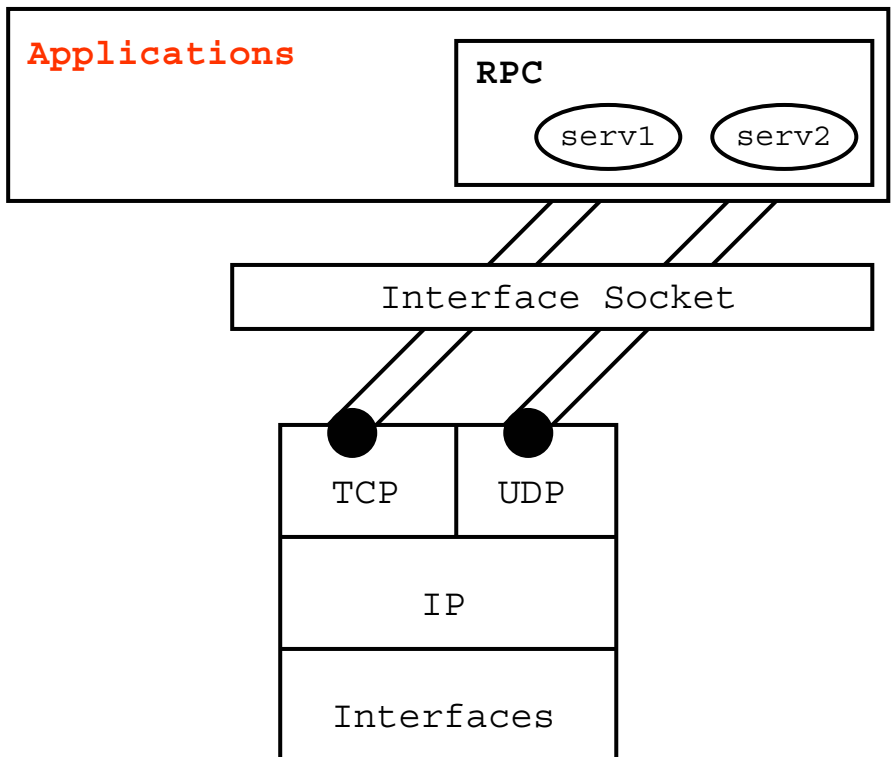
      Java 2 (Platform) Enterprise Edition – Enterprise Java Beans
    - OMG CCM

      CORBA Component Model
    - WS-SOAP

      Web Services - Simple Object Access Protocol

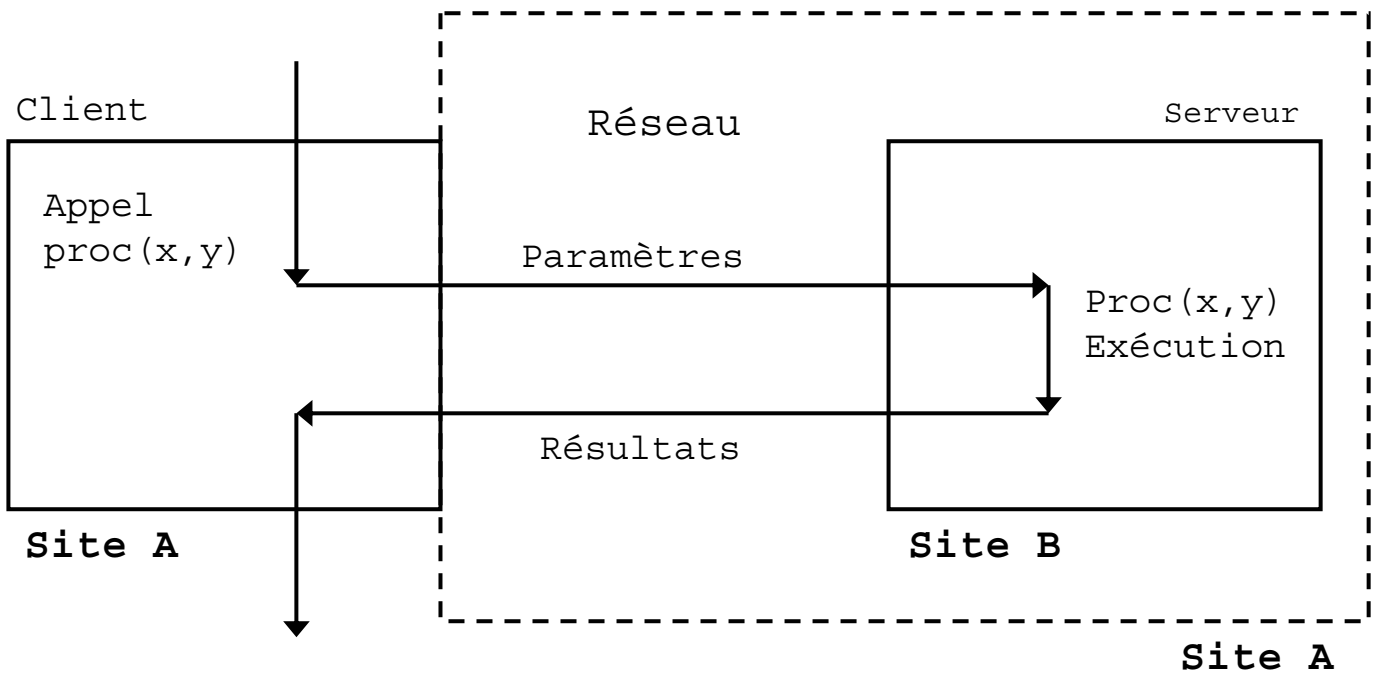
Introduction

- Définition
  - Ensemble de services s'appuyant sur l'interface des sockets
    - masquer l'interface socket
    - paradigme de l'appel de procédure en lieu et place des opérations send et receive.



Introduction

- Permettre l'appel de procédures à travers le réseau
  - Modèle Client-Serveur
  - Transparence (comme un appel en local)
  - Notion de souche (stub) client et serveur (cacher le fonctionnement du réseau)



Introduction

- Les difficultés :
  - Transmission des paramètres
    - multiples représentations.
    - passage par référence.
  - Localisation des serveurs
    - enregistrement des services.
    - localisation d’un service.
  - Pannes
    - défaillance du réseau.
    - de client, de serveur.

Mise en œuvre des RPC

- Notion de souches
  - Un mode de réalisation par interception ("wrapping")
    - une procédure intercepteur ("wrapper") intercepte l'appel d'un client vers un serveur et modifie le traitement serveur à sa guise.
    - décomposition en intercepteur côté client et intercepteur côté serveur.
    - décomposition en traitements avant et après le traitement serveur.
  - Souches
    - transformation d'un appel local en appel distant.
    - les détails de fonctionnement du réseau sont cachés au programme d'application, car inclus dans les procédures locales.
  - Très nombreuses terminologies dans ce cas
    - souches, stubs, talons, squelettes, procédures subrogées …

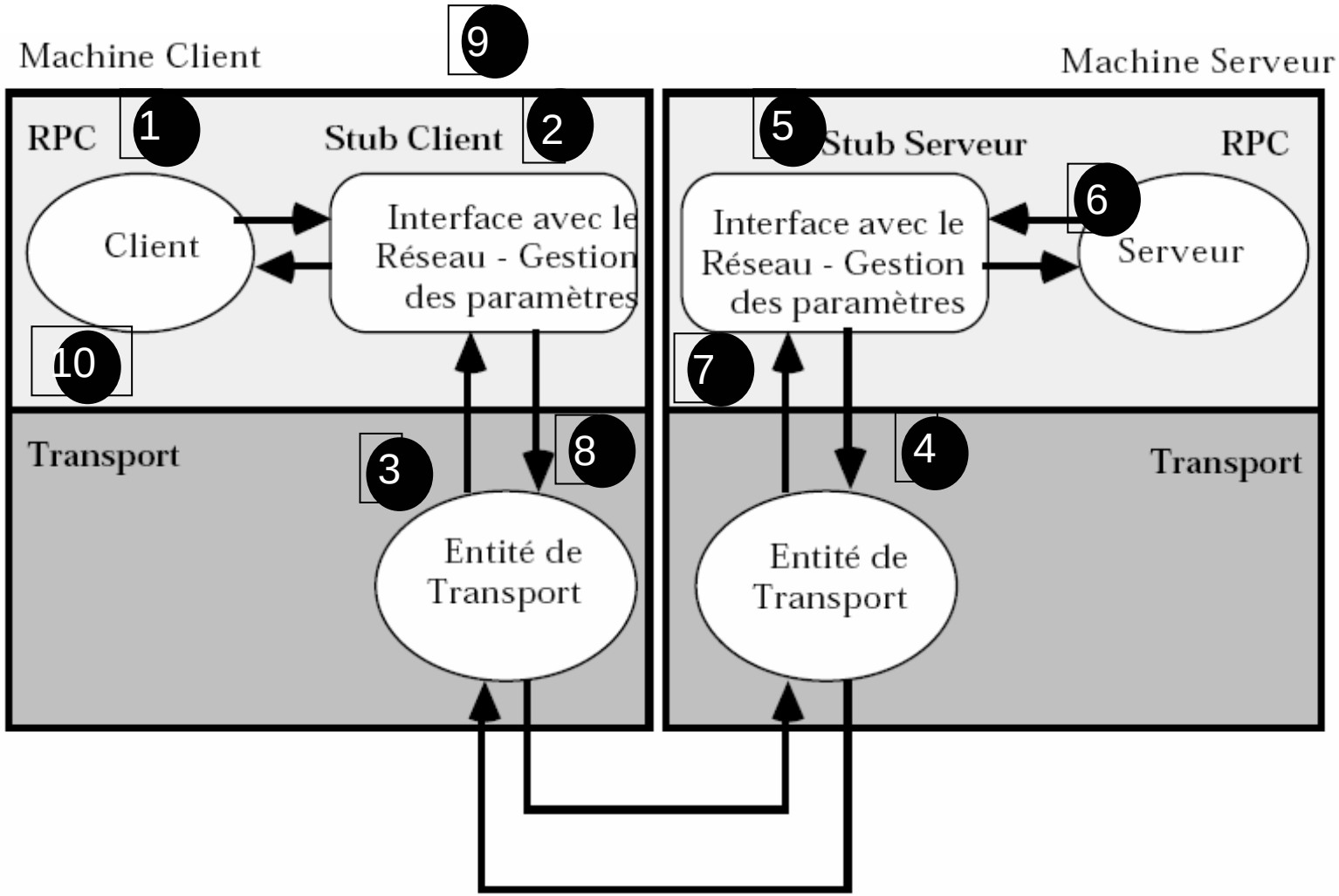
Mise en œuvre des RPC

- Souche Client ("Client Stub")
  - procédure côté client qui reçoit l’appel en mode local
  - le transforme en appel distant en envoyant un message
  - reçoit le message contenant les résultats après l’exécution
  - retourne les résultats comme dans un retour de procédure

- Souche Serveur ("Server Stub")
  - procédure côté serveur qui reçoit le message d’appel
  - fait réaliser l’exécution sur le site serveur par la procédure serveur
  - récupère les résultats et les retransmet par message

Mise en œuvre des RPC

1. Etape d'un RPC par message



Mise en œuvre des RPC

- Les étapes d'un RPC par message
  - Étape 1
    - le client réalise un appel vers la procédure souche client
    - la souche client collecte les paramètres, les code en format externe et les assemble dans un message (opération "parameters marshelling")
  - Étape 2
    - la souche client détermine l'adresse du serveur
    - la souche client demande à une entité de transport locale la transmission du message d'appel
  - Étape 3
    - le message est transmis sur un réseau au site serveur

Mise en œuvre des RPC

- Les étapes d'un RPC par message
  - Étape 4
    - le message est délivré à la souche serveur
  - Étape 5
    - la souche serveur désassemble les paramètres et réalise l'appel effectif de la procédure serveur
    - dans le cas de RPC synchrones, on réalise ici un rendez-vous d'activation.
  - Étape 6
    - la procédure serveur ayant terminé son exécution, transmet à la souche serveur, dans son retour de procédure, les paramètres résultats
    - la souche serveur collecte les paramètres retour et les assemble dans un message

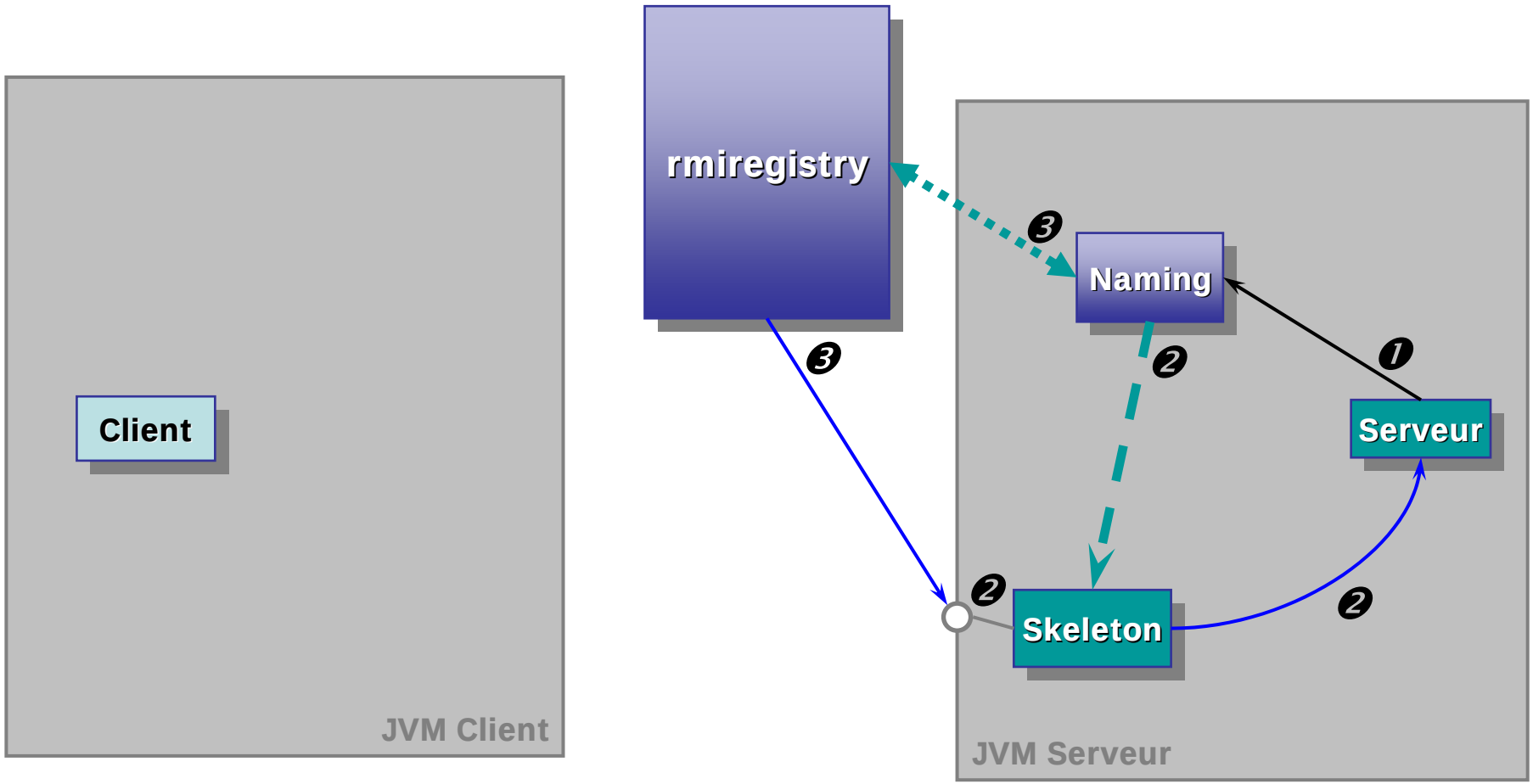
Mise en œuvre des RPC

- Les étapes d'un RPC par message
  - Étape 7
    - la procédure souche serveur demande à l'entité locale de transport la transmission du message
  - Étape 8
    - le message est transmis sur un réseau au site client
  - Étape 9
    - le message est délivré à la souche client,
    - la souche client désassemble les paramètres retour (opération "parameters unmarshelling")
      - à l'arrivée les résultats doivent être remis en format interne à partir du message en format externe reçu.
  - Étape 10
    - la procédure souche client transmet les résultats au client en effectuant le retour final de procédure

Exemple RPC : Java RMI Remote Method Invocation

- Positionnement :
  - Client-serveur « traditionnel »
    - DCE (Distributed Computing Environment)
      - Chaque fonction est utilisable via une interface normalisée.
  - Client-serveur « de données »
    - Requêtes SQL à une base de données.
  - Client-serveur « à objet »
    - Java RMI, Corba 2.x, COM
  - Client-serveur « à composant »
    - Corba 3.x, DCOM

Exemple RPC : Java RMI Remote Method Invocation

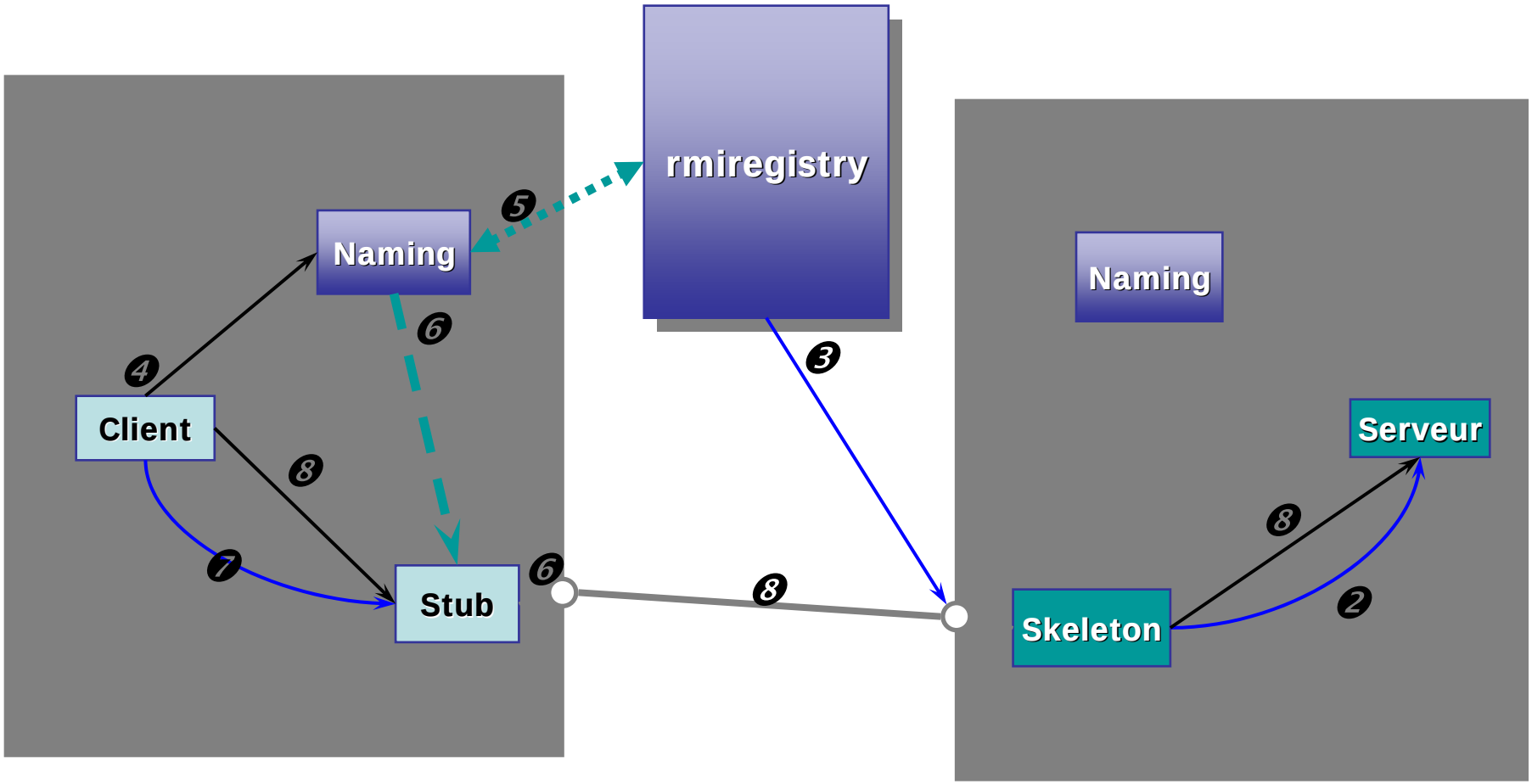


Java RMI
Mode opératoire coté serveur

- 1 - L'objet serveur s'enregistre auprès du Naming de sa JVM (méthode *rebind*)
- 2 - L ’objet skeleton est créé, celui-ci crée le port de communication et maintient une référence vers l'objet serveur
- 3 - Le Naming enregistre l'objet serveur, et le port de communication utilisé auprès du serveur de noms

- L'objet serveur est prêt à répondre à des requêtes

Exemple RPC : Java RMI Remote Method Invocation



Java RMI
Mode opératoire coté serveur

- 4 - L'objet client fait appel au Naming pour localiser l'objet serveur (méthode lookup)
- 5 - Le Naming récupère les "références" vers l'objet serveur, ...
- 6 - crée l’objet Stub et ...
- 7 - rend sa référence au client
- 8 - Le client effectue l'appel au serveur par appel à l’objet Stub

Mise en œuvre des RPC

2. Réalisation d'un RPC par migration

- Stratégie de migration
  - le code et les données de la procédure distante sont amenés sur le site appelant pour y être exécutés par un appel local habituel.
  - analogie avec une stratégie de **pré-chargement** en mémoire.

- Avantages
  - très efficace en cas de nombreux appels.

- Inconvénients
  - univers d'exécutions homogènes (i.e. machine virtuelle).
  - performance selon le volume de codes et de données.
  - problème de partage des objets (exclusion mutuelle entre exécutions).

Mise en œuvre des RPC

3. Réalisation d'un RPC en mémoire partagée répartie

- L'appel distant est réalisé en utilisant une mémoire virtuelle partagée répartie.
- La procédure est installée pour le client comme pour le serveur dans la mémoire partagée répartie
  - en fait, elle est dans l'espace réel du serveur.
- L'appel du client se fait comme si la procédure était locale, provoquant un premier défaut de page sur le début du code de la procédure.
- Le code et les données de la procédure distante sont amenés page par page sur le site appelant selon le parcours du code et des données
  - analogie avec une stratégie "page à la demande".

Mise en œuvre des RPC

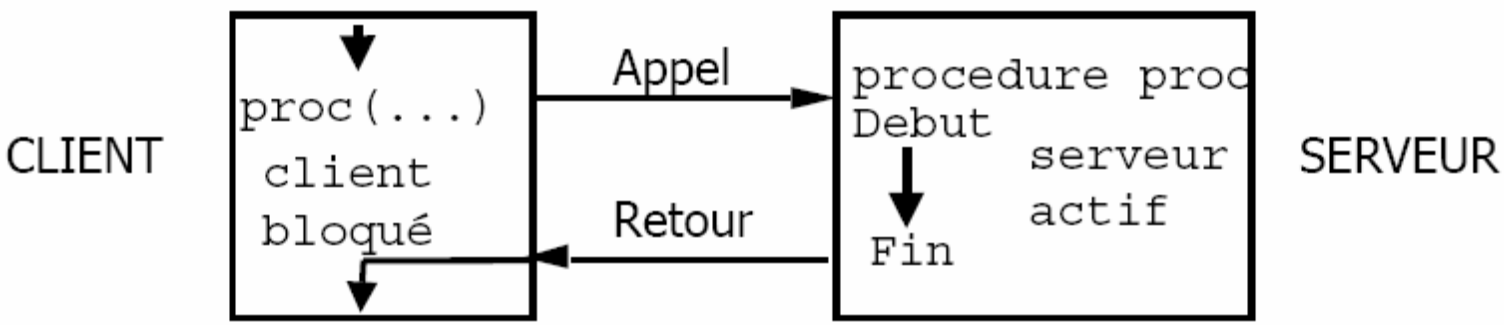
3. Réalisation d'un RPC en mémoire partagée répartie

- Avantages
  - efficace en cas de nombreux appels.
  - efficace si tout le code et les données ne sont pas visités.
  - résout le problème de l'utilisation des pointeurs (références d'adresses en mémoire).

- Inconvénients
  - univers d'exécutions homogènes.
  - volume de codes et de données à échanger page par page.
  - problèmes de partage selon cohérence de la mémoire répartie.

Appel de procédures en mode synchrone

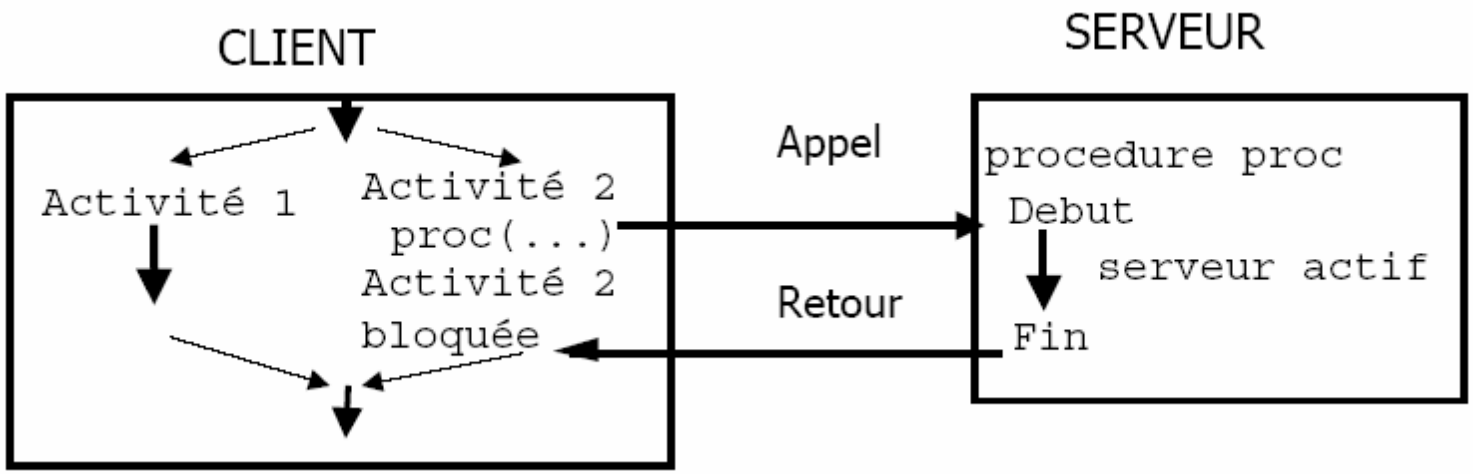
- Réalisation d'un RPC par messages de type synchrone
  - Deux messages (au moins) échangés
    - le premier message, correspondant à la requête, est celui de l'appel de procédure, porteur des paramètres d'appel.
    - le deuxième message, correspondant à la réponse, est celui du retour de procédure, porteur des paramètres résultats.



Inconvénient: le client reste inactif.

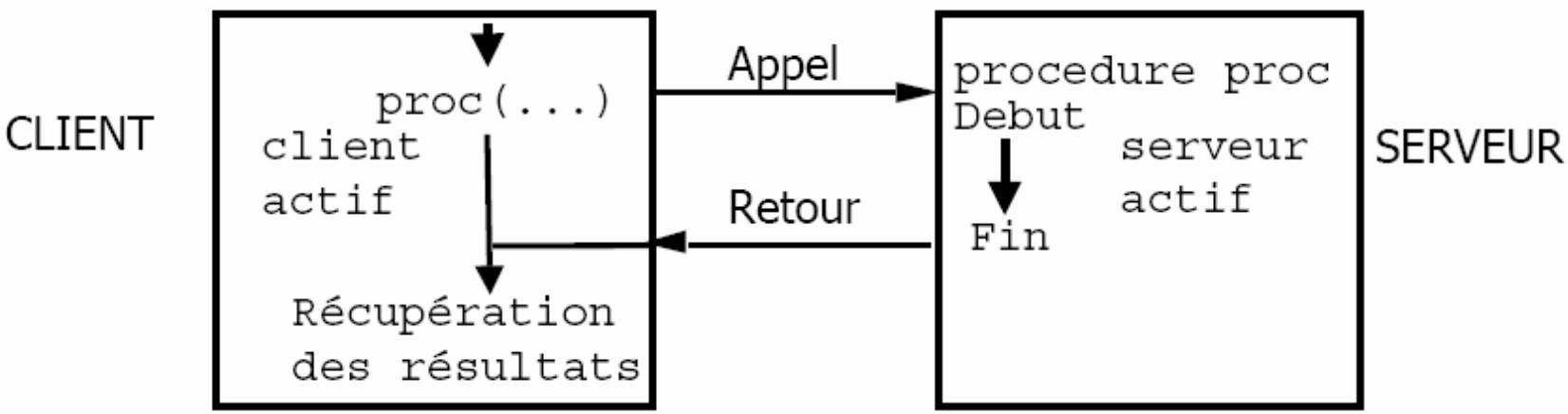
Appel de procédures en mode synchrone

- Solution au client inactif
  - Création de (au moins) deux activités (‘ threads ’) sur le site client
  - L'une occupe le site appelant par un travail à faire.
  - L'autre gère l’appel en mode synchrone en restant bloquée: Le fonctionnement est exactement celui d'un appel habituel.



Appel de procédures en mode asynchrone

- Le client poursuit son exécution immédiatement après l'émission du message porteur de l'appel.
  - La procédure distante s'exécute en parallèle avec la poursuite du client.
  - Le client doit récupérer les résultats quand il en a besoin (primitive spéciale).



Récupération des résultats en mode asynchrone: notion de futurs explicites

- Un futur : une structure de donnée (un objet) permettant de récupérer des résultats.

- Notion de futur explicite
  - **Les structures de données sont construits par le client avant l'appel (le serveur les connaît et y dépose les résultats).**
  - Exemple: En ACT++ une boite à lettre définie par le client sert de moyen de communication pour les paramètres résultats.
  - BAL := factorielle.calculfact(n)
  - resultat := BAL.prélever()

Récupération des résultats en mode asynchrone: notion de futurs implicites

- Invocation asynchrone à futur implicite
  - Les structures de données de communication pour les réponses (ex boite à lettre) sont **implicitement créés par le prestataire du service** d‘appel à distance asynchrone. C**’est le mécanisme d’appel qui construit les objets résultats**
  - Approche générale: défaut d'information (analogie défaut de page en mémoire virtuelle).
  - **La lecture de la structure de donnée résultat bloque le client s'il accède à la réponse et que celle-ci n'est pas parvenue.**
  - **L’usager ne s’aperçoit de rien (si le résultat est parvenu ou s ’il n’est pas parvenu).**

Invocation à sens unique

- **Invocation asynchrone sans réponse (autre terminologie, "peut-être", "oneway" , "maybe")**
- Invocation asynchrone utilisé pour déclencher une procédure qui ne retourne pas de résultats. Pour obtenir un dialogue il faut prévoir d'autres procédures en sens inverse.

- **Avantage**: Utilisation d'un mode appel de procédure pour des communications sont en fait de mode message.

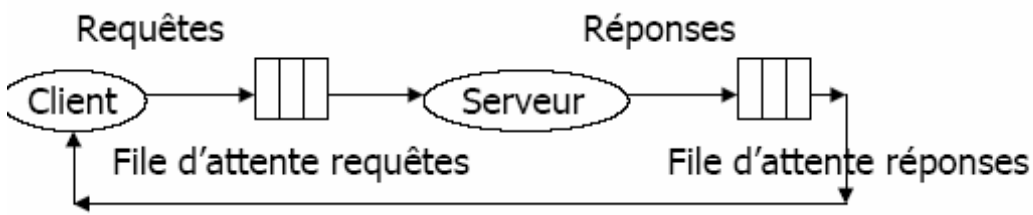
- **Inconvénient** : Uniquement possible en l'absence de retour de résultat, pas d'informations sur la terminaison du travail demandé.

- Exemples: CORBA oneway.

Gestion des appels : Exécution séquentielle

- Les requêtes d'exécution sont traitées l'une après l'autre par le serveur: exclusion mutuelle entre les traitements.

- Si la couche transport assure la livraison en séquence et que l'on gère une file d'attente premier arrivé premier servi, on a un traitement ordonné des suites d'appels.

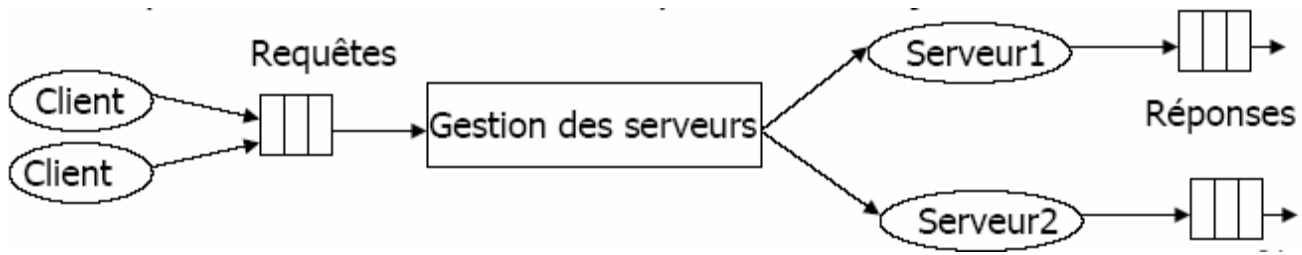


- Exemple RPC SUN : traitement séquentiel des requêtes mais utilisation de UDP => requêtes non ordonnées (mais mode synchrone le client attend la fin du traitement).

- Autres exemples: les RPC ont un mode séquentiel (ex CORBA)

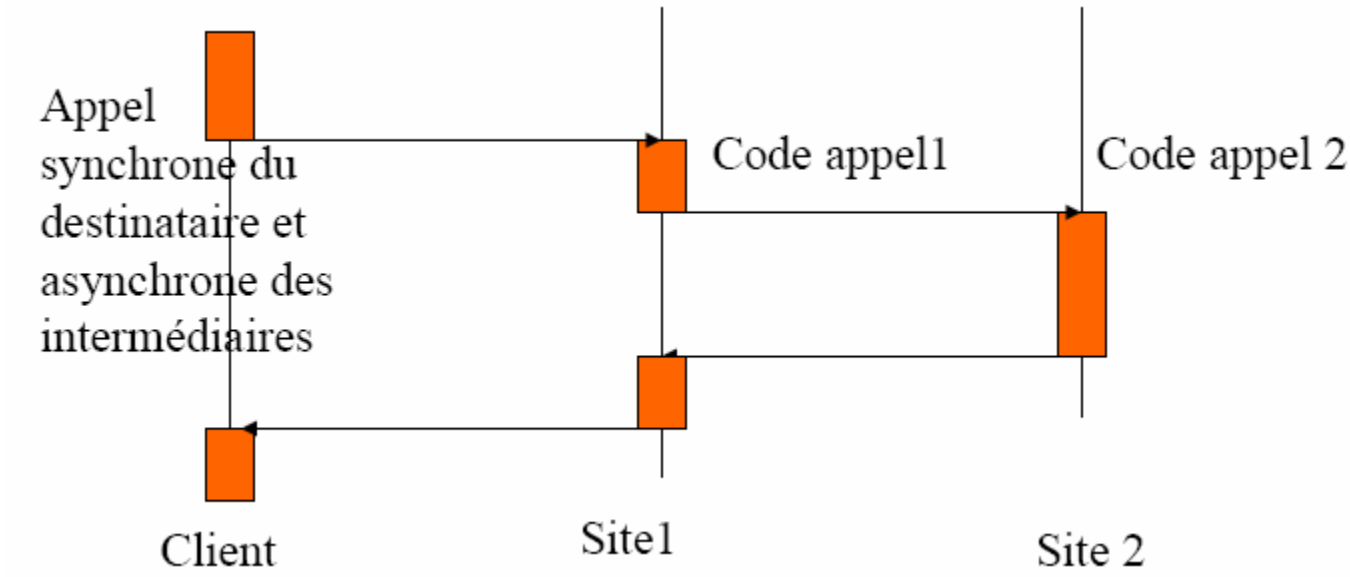
Gestion des appels : Exécution parallèle

- Dans ce mode, le serveur créé un **processus** ou une **activité** (un processus léger ou “thread”) par appel (gestion possible de pool de processus ou d ’activités).
- Les appels sont exécutés **concurremment.**
- Si les procédures manipulent des données globales rémanentes sur le site serveur, le **contrôle de concurrence doit être géré.**
- Exemple : Corba Notion d’adaptateur d’objets.



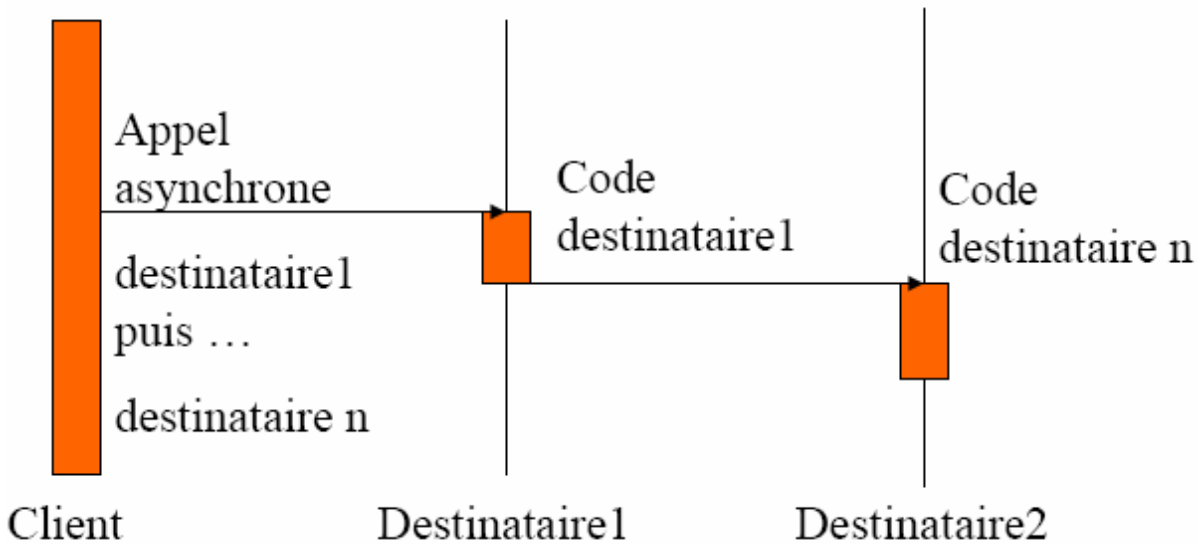
Appel à distance en séquence

- Appel à distance synchrone



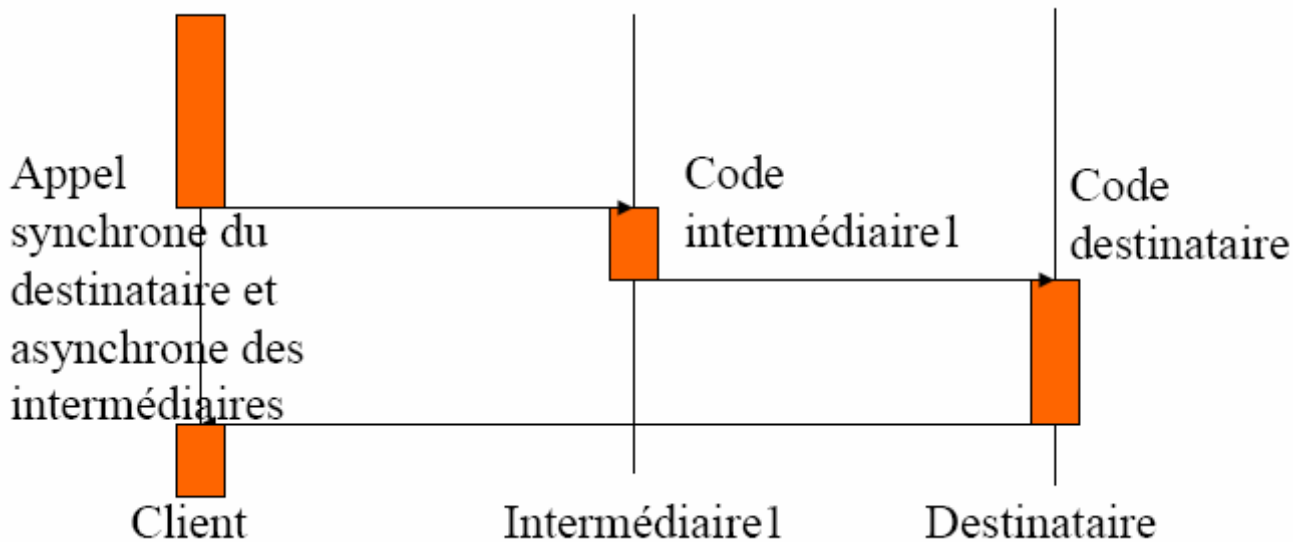
Appel à distance en séquence

- Baptisé **schéma à continuation en mode asynchrone**. L'émetteur prépare une liste de procédures destinataires à invoquer en mode asynchrone. Le message d'appel visite successivement les destinataires.
- Analogie avec le routage par la source en mode message
- Implantation: protocole SOAP en mode message.



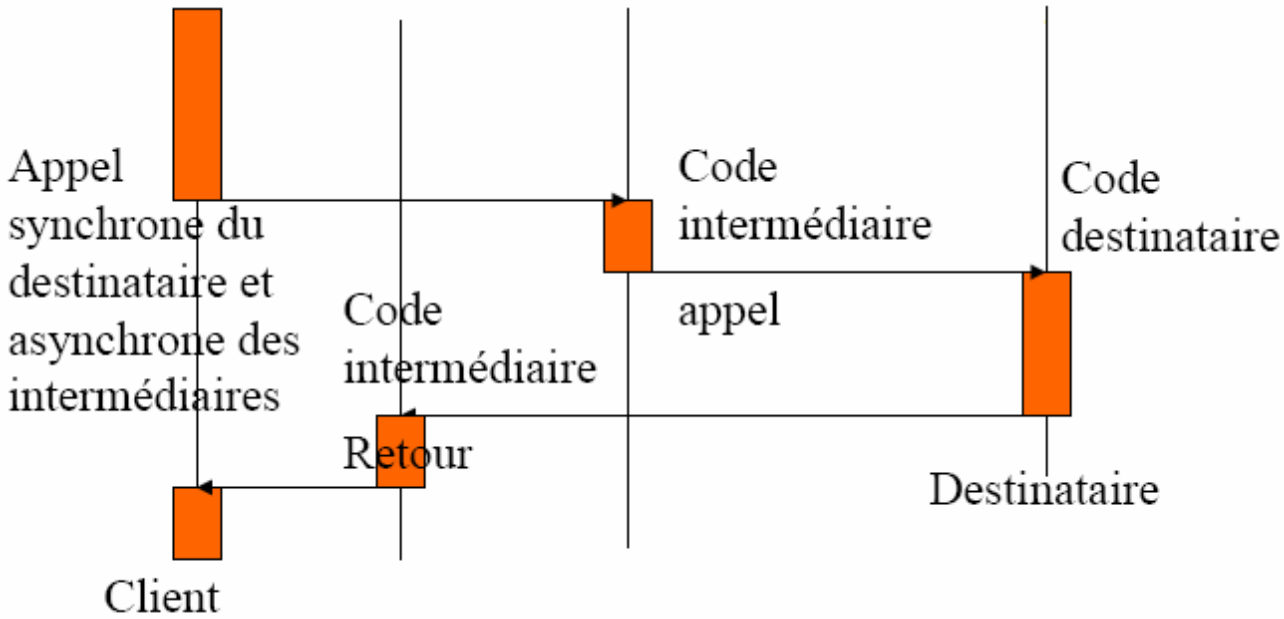
Continuation asynchrone avec appel synchrone pour le client

- Le premier schéma à continuation proposé. **Une liste d'intermédiaires en mode asynchrone et un destinataire final** : le tout en mode synchrone pour le client.
- Implantation: protocole SOAP en mode RPC.



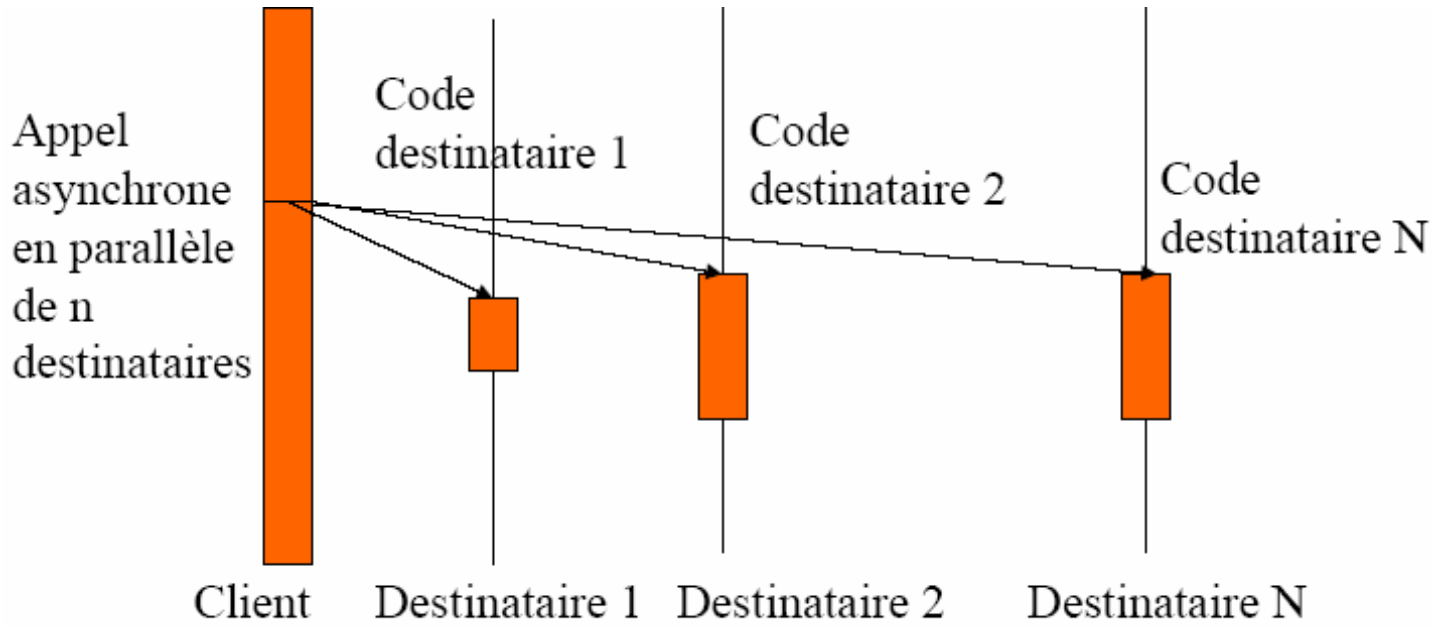
Continuation asynchrone en appel synchrone pour le client et avec réponse

- Une liste d'intermédiaires en mode asynchrone est possible à l'appel comme à la réponse : le tout en mode synchrone pour le client.
- Implantation: protocole SOAP en mode RPC.



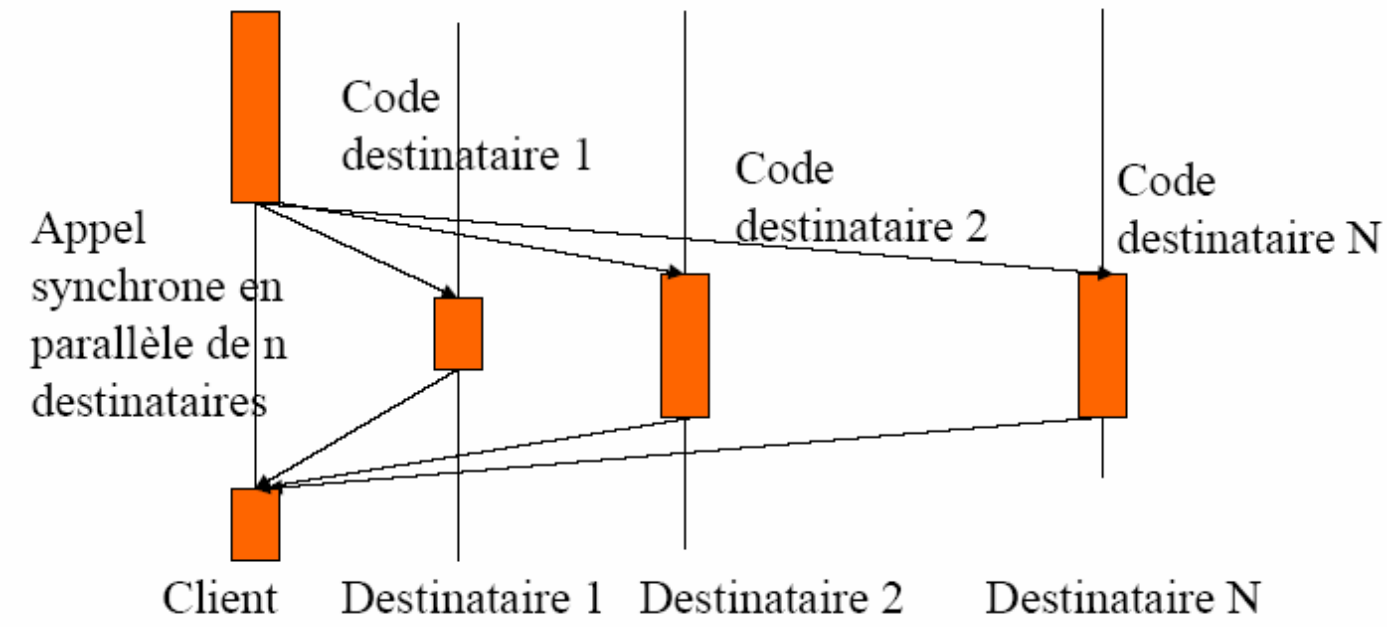
Continuation asynchrone contrôle par composition parallèle d'appels distants

- **Notion de RPC asynchrone sur groupe**



Continuation asynchrone contrôle par composition parallèle d'appels distants (2)

- Notion de RPC synchrone sur groupe (autre nom RPC diffusé, RPC parallèle)



Propriétés d’ordre dans les communications

- L'appel de procédure distante peut comporter des spécifications de propriétés d'ordre.
  - Le respect d'une propriété d'ordre peut porter:
    - sur le lancement de la procédure (peut utilisable)
    - sur la totalité de son exécution.

- Ordre local: Les exécutions pour un client sont réalisées dans l'ordre d'émission.

- Ordre global: Les exécutions pour un client sont réalisées dans le même ordre sur tous les destinataires (cas des communications de groupe).

- Ordre causal: Les exécutions sont effectuées en respectant la relation de causalité qui existe entre les requêtes.

Avantages / Inconvénients

- Réalisation d'un RPC par messages asynchrones

- **Avantages**
  - volume de code ou de données serveur quelconque.
  - applicable en univers hétérogènes moyennant des conversions.
  - partage d'accès sur le site serveur analogue au transactionnel.

- **Inconvénients**
  - pas d'usage des pointeurs dans les paramètres.
  - échange de données complexes, de grande taille délicat.
  - peu efficace dans le cas de très nombreux appels.

RPC Local

- **Problème de performance**
  - quand on invoque un serveur qui se trouve sur la même machine, la traversée des couches réseaux est inutile et coûteuse.
  - Si le serveur se trouve dans le même processus (même domaine)
    - pas de problème, appel local.
  - Si le serveur se trouve dans un autre processus (autre domaine)
    - la communication réseau est réalisée par un segment de mémoire partagée entre le client et le serveur qui contient un tas pour les paramètres d'appel et de réponse.
- **Avantages**
  - transmission d'appel très performant comme mode de RPC local.
- **Inconvénients**
  - uniquement applicable aux RPC du même site.

Conclusion 1

- L’appel est d’abord développé en invocation distante par messages
- Supporte l’hétérogénéité.
- Finalement, le plus simple à réaliser.
- RPC, DCE, CORBA, RMI, DCOM, SOAP.
- Des optimisations peuvent être obtenues par l’usage opportun des autres solutions
  - exemple : Chorus a développé les 4 solutions.
  - exemple : DCOM RPC par messages + RPC léger

Sémantique des RPC

- Existence de très nombreuses implantations différentes des RPC, ayant des sémantiques variées.
- L'objectif affiché par la plupart des implantations serait pour le programmeur :
  - de retrouver la sémantique habituelle de l'appel de procédure en mode centralisé.
  - sans se préoccuper de la localisation de la procédure exécutée (sauf s'il le souhaite explicitement).

Gestion des données: sans Persistance

- Mode sans donnée partagée persistante (rémanente)
  - Données locales à la procédure : pas de problème.
  - Données applicatives partagées (variables d'instance, fichiers, bases de données) : problème de persistance.

- Sans donnée rémanente
  - Situation idéale du cas où l'appel de procédure s'exécute en fonction uniquement des paramètres d'entrée
    - en produisant uniquement des paramètres résultats.
  - Pas de modification de données rémanentes sur le site serveur.
  - Pas de problème pour la tolérance aux pannes et pour le contrôle de concurrence.

Gestion des données: Persistance

- Les exécutions successives manipulent des données rémanentes sur le site serveur.
  - Une telle exécution modifie le contexte sur le site distant
    - serveur de fichier réparti, de bases de données.
    - Opérations d'écriture de données persistantes, la structure de donnée manipulée par les méthodes d'un objet
  - problème de contrôle de concurrence.
  - difficultés en cas de pannes en cours d'exécution.

- Solution
  - Le couplage d'une gestion transactionnelle avec une approche RPC (ou système d'objets répartis).

  - Exemple EJB

RPC

Partie 2

Rappel

- RPC sans réponse : demande un échange synchrone.
  - Le programme appelant reste en attente de la réponse du serveur.
  - Une fois que le traitement est terminé, le serveur transmet la réponse attendue par le programme client en un seul flot.
    - Caractère exclusivement synchrone de l'échange
    - Fiabilité médiocre (repose sur une exécution et une seule; si elle échoue, il n'y a pas de reprise)
    - Pas de synchronisation entre le client et le serveur au cours de l'échange
    - Pas de gestion du flux de retour (la réponse arrive en une fois quel qu'en soit le volume)

- **Le RPC avec réponse:** appels avec réponse.
  - compte rendu d'exécution sur deux cas, le cas de réussite et celui d'échec de l'exécution de la procédure.
- **Le RPC avec réponses multiples** (appelé Broadcast RPC). Il entraîne plusieurs comptes rendus pendant l'échange. Par exemple, la procédure peut envoyer des résultats à des intervalles réguliers. Ils sont idéals pour des calculs exécutés à intervalle fixe.

Persistance des données

1. **Avec ou sans état**
2. **Avec ou sans connexion**

Rappel

- Objectifs :

a) Distribution:
   - Répartir une application sur plusieurs machines

b) Transparence:
   - Appeler une procédure distante comme si elle était locale

c) Facilité de conception:
   - Transformer des applications locales en applications réparties

- Indépendance du protocole de transport

- Usage :

- Convertir une application développée localement en une application distribuée.

- Les procédures sont exécutées sur un site central:
  - Facilité d'administration
  - Facilité de mise à jour
  - Renforcer les règles d'intégrité pour les BD

Gestion des données: Persistance

- Autre aspect de la rémanence des données sur le serveur:
  - La terminologie **avec ou sans état** porte sur **l'existence ou non d'un descriptif** pour chaque relation client serveur au niveau du serveur.
  - **Notion d'état** : un ensemble de données rémanentes au niveau du protocole pour chaque relation client serveur.
    - Permettrait de traiter les requêtes dans l'ordre d'émission.
    - Permettrait de traiter une requête en fonction des caractéristiques de la relation client serveur (qualité de service).

Gestion des données: Persistance

- Il existe plusieurs possibilités pour rendre un objet persistant :

  - Par instanciation à partir d'une classe persistante
    - Une classe souvent située a la racine du graphe d'héritage définie la propriété de persistance. Tout objet qui est une instance de cette classe ou de ces sous-classes hérite alors de cette propriété et est rendu persistant.
  - A la création de l'objet
    - Soit l'on crée l'objet avec un paramètre supplémentaire pour indiquer qu'il doit être rendu persistant. Ou par commande spéciale
  - Par envoi de message rendant l'objet persistant.
    - redéfinir la persistance de l'objet par envoi de messages.

  - Par stockage dans une zone définie comme persistante.
    - On définit certaines variables comme persistantes, tout objet référencé par cette variable devient alors persistant.
  - Par attachement de l'objet à une structure persistante
    - Dans ce schéma, il existe en général un objet racine persistant. Tous les objets que l'on peut atteindre à partir de cette racine, deviennent eux mêmes persistants : tout objet attaché à une structure de donnée persistante devient lui-même persistant

Mode sans Etat

- Les appels successifs d'une même procédure s'exécutent **sans liens** entre eux.
  - Il peut y avoir modification de données globales rémanentes sur le site serveur mais **chaque opération** du point de vue du protocole s'effectue sans référence au **passé** (indépendamment de toutes celles qui ont précédé).
  - Exemples:
    - **NFS** ' Network File System ' de SUN système de fichier réparti basé sur RPC sans état. Lecture/Écriture du nième article d'un fichier dont toutes les caractéristiques utiles (nom, droit d'accès) sont passées au moment de l'appel.
    - **HTTP** ' HyperText Transfer Protocol ' protocole d'exécution de méthodes distantes sans état.

Mode avec Etat

- Les appels successifs s'exécutent en fonction d'un état de la relation client serveur laissé par les appels antérieurs.
  - Exemple d'utilisation : la gestion de l'ordre de traitement des requêtes, la gestion de caractéristiques du client.
  - Exemple système de fichier en RPC: Lecture d'article de fichier sur le pointeur courant.

Mode avec ou sans connexion

- Gestion des connexions :
  - Délimitation temporelle des requêtes et des exécutions de procédures entre des opérations d'ouverture et de fermeture.
  - Maintien d'un descriptif de connexion sur le client et sur le serveur pour la gestion de paramètres caractéristiques de la connexion (QoS, traitement des pannes, propriétés d'ordre, …).
  - Définition d'une référence (désignation) de connexion.

Appel de procédure distante sans connexion

- Dans ce mode le client peut envoyer des appels au serveur à n'importe quel moment.
  - Le client comme le serveur ne gèrent pas de descriptif de la relation client serveur : absence de mémoire entre appels successifs.
  - Le mode sans connexion est donc un mode orienté
    - Vers un traitement sans gestion de qualité de service des appels.
    - Vers le traitement d'un grand nombre de clients: la gestion de connexions est jugée trop coûteuse.
- Exemple : tous les RPC.

Transmission des arguments

I) La transmission par valeur

II) Les autres modes de passage

Transmission des arguments

- Problème du passage des paramètres dans l'appel de procédure distante

  - Existence de sémantiques de transmission variées :
    - par valeur ,
    - par adresse (copie/restauration) ,
    - par référence/nom

Transmission des arguments : par valeur

- Le seul mode de transmission des données dans les messages en réseau.
- Si le site client et le site serveur utilisent des formats de données différents : **problème syntaxique**: une conversion est nécessaire.
- Solution:
  - notion de présentation des données au moyen d'une syntaxe de transfert => une représentation lexicale des données simples et une convention d'alignement des données commune au client et au serveur.

- Exemple : *little endian* vs *big endian*
  - Sens des octets d'une chaîne de caractères

  - Poids fort des nombres à gauche ou à droite

  - Alignement des données sur les frontières de mots
- Conventions de représentation des nombres flottants
- Conventions de représentation des structures complexes

Transmission des arguments : par valeur

- **Définir une syntaxe abstraite de données pivot**:
  - analogue des langages de description de données des langages évolués, facile à générer pour un développeur d'application (en mode message ASN1 « Abstract Syntax Notation 1 »).
  - A partir de la syntaxe abstraite: **codage/décodage** de la **syntaxe de transfert** (technique de compilation, notion de compilateur de protocole).
  - Défini dans la recommandation X.208 du **CCITT**, et la norme 8824 de l'**ISO**, **ASN.1** est un langage utilisé pour décrire des syntaxes abstraites.
  - ASN.1 (*Abstract Syntax Notation One*) est un langage formel qui permet de spécifier les données transmises par les **protocoles de télécommunications** indépendamment des langages informatiques et de la représentation physique de ces données, pour toutes sortes d'applications communicantes et de données aussi complexes (ou aussi simples) soient-elles.
  - ASN.1 appartient au domaine des protocoles de communications, et peut être assimilé à un langage de programmation de haut niveau, tout du moins en ce qui concerne la partie spécification des données.

Construction d'application ASN1

1. **Développement des spécifications**
   - Définir quels types de messages que l'application doit échanger (envoyer/recevoir
   - Sur la base des besoins, de nouvelles spécifications ASN.1 peuvent être construites ou bien directement utilisées si elles existent déjà.
   - Quand de nouvelles spécifications sont conçues, il faut décidé quelles sont les règles de transferts qui doivent être utilisées (BER, DER, PER..)
2. **Vérification de la syntaxe et compilation:**
   - Code cible peut être produit en C, C++, Java, pascal ou tout autre langage selon le compilateur ASN.1 utilisé
3. **Ecriture de l'application :**
   - Ecriture de structures de données produites par le compilateur ASN.1.
   - Utilisation des librairies natives pour coder, décoder les données de l'application.
4. **Mis en œuvre de l'application**

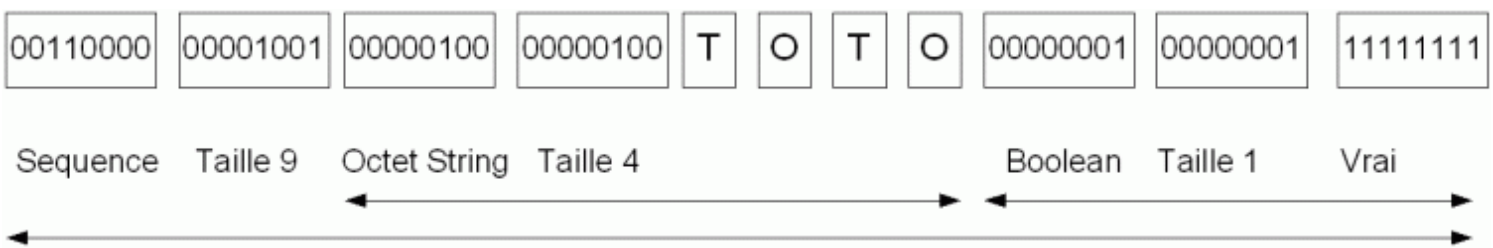
Syntaxe de transfert

- Se caractérise par un ensemble de règles expliquant comment coder/décoder les types simples et complexes, définis au niveau de la syntaxe abstraite. Appliquer la syntaxe de transfert permet alors d'obtenir des chaînes d'octets transmissibles sur le réseau et décodables sans ambiguïté.
- ***BER, Basic Encoding Rules :***
  - La syntaxe de transfert définie en conjonction avec le langage ASN.1 est BER, issues de la recommandation X.209 du **CCITT**, et de la norme 8825 de l'**ISO**. BER est un algorithme qui, à partir des valeurs ASN.1, encode les bits en format approprié pour la transmission sur le réseau, et de façon à en optimiser la représentation.

Exemple BER : Syntaxe de transfert

- Toutes les primitives sont codées sous la forme (type,longueur,valeur)
- Les types construits sont codés dans un TLV avec les sous éléments stockés dans la partie V. Les parties T,L et V sont chacune codées sur un nombre entier d'octets. La partie T identifie sans ambiguïté le type de l'élément au sein du contexte dans lequel il va être utilisé, elle est constituée éventuellement du tag que l'utilisateur a spécifié afin de garantir que les éléments T correspondent aux contraintes suivantes :

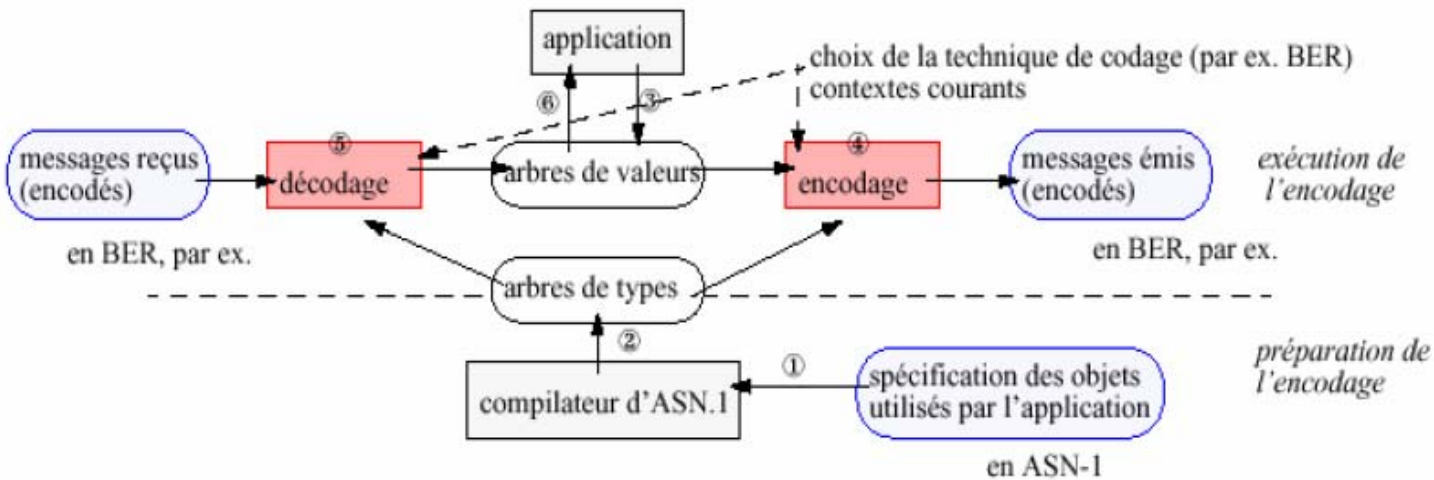
Exemple de codage pour :
Participant ::= SEQUENCE {
nom OCTET STRING
type BOOLEAN
}



Utilisation

- Règles utilisées pour transformer les données spécifiées en un format standard qui peut être décodé sur n'importe quel système.
- Le choix des règles est laissé au concepteur du protocole.
- Les règles d'encodage ASN.1 standardisées sont
  - Basic Encoding Rules (**BER**) début 1980
  - Distinguished Encoding Rules (**DER**)
  - Canonical Encoding Rules (**CER**)
  - Packed Encoding Rules (**PER**)

.



Utilisation

- BER dans un grand champ d'applications :
  - SNMP Simple Network Management Protoco
  - MHS Message Handling Services
  - TSAPI (contrôle des interactions telephone/computer)
  - DER = un BER spécialisé pour les applications sécurisées,e-commerce

- CER similaire à DER, pour des très gros messages à encoder au fur et à mesure.

- PER plus récent avec un algorithmes qui implique un résultat rapide et plus compact dans des applications style contrôle aérien et télécommunications audio-visuelle.

- Et Lean ED encodage de petite taille avec de hautes performances au détriment des possibilités de diagnostic

.

RPC : représentation des paramètres

- Solution normalisée (ASN.1)
  - Syntaxe abstraite (Abstract Syntax Notation) pour représenter des
  - structure de données
  - Codage indépendant des machines
  - Outils de génération disponibles pour des machines spécifiques
- Autres solutions
  - Représentation externe commune. Exemple : Sun XDR (External Data Representation) utilisée dans NFS
  - Conversions inutiles si client et serveur de même type
  - Conversion explicite (par le serveur) à partir d'une représentation locale au client
  - Négociation entre client et serveur

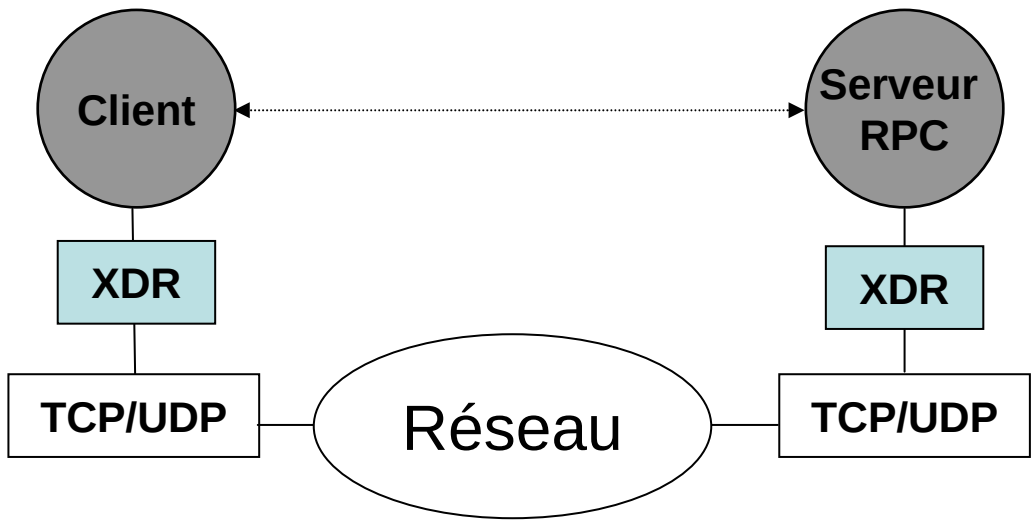
Un exemple de convention de représentation de paramètres : XDR

- XDR : External Data Representation (Sun)
  - Format de transmission sur réseau
  - Bibliothèque de programmes de conversion
- Ensemble de routines disponibles pour les types de base
  - Données élémentaires
  - Structures simples (tableaux, chaînes, unions)
  - Traitement de pointeurs, gestion de mémoire
- Utilisation : programmes d'emballage/déballage
  - Courante : par les générateurs automatiques de talons, dans tous les cas usuels
  - Avancée : directement par le programmeur, pour le traitement de structures de données complexes

XDR

- Bibliothèque de codage qui définit comment les données doivent être véhiculées sur le réseau.

- Permet d'échanger des données entre machines ayant des représentations internes différentes.



XDR

type	taille	description
int	32 bits	entier signé de 32 bits
u_int	32 bits	entier non signé de 32 bits
bool_t	32 bits	valeur booléenne (0 ou 1)
enum_t	arb.	type énuméré
hyper	64 bits	entier signé de 64 bits
u_hyper	64 bits	entier non signé de 64 bits
float	32 bits	virgule flot. simple précision
double	64 bits	virgule flot. double précision
opaque	arb.	donnée non convertie
fixed array	arb.	tableau de longueur fixe de n'importe quel autre type
struct_t	arb.	agrégat de données
discriminated union	arb.	structure implémentant des formes alternatives
symbolic constant	arb.	constante symbolique
void	0	utilisé si pas de données
string	arb.	chaîne de car. ASCII

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

opaque : Donnée transmise telle quelle (stockée mais non consultée par un programme donné)

XDR : exemple

```
#include <rpc/types.h>
#include <rpc/xdr.h>
#define ARITH_PROG 0x33333333
#define ARITH_VERS1 1
#define ADD_PROC 1
#define MULT_PROC 2
#define SQRT_PROC 3
struct couple {float e1, e2;};
int xdr_couple();
```

Exemple de flux :

```
#include "exemple.h"

int xdr_couple(XDR * xdrp, struct couple * p)
{
  return (xdr_float(xdrp, &p->e1)&&xdr_float(xdrp, &p->e2));
}
```

XDR : exemple Serveur

```
#include <stdio.h>
#include "exemple.h"

char *add();
char * mult();
char * rac();

main(){
  int rep;
  rep=registerrpc(ARITH_PROG, ARITH_VERS1, ADD_PROC, add,
   xdr_couple, xdr_float);
  if (rep==-1){
    fprintf(stderr, "errreur registerrpc (add)\n");
    exit(2);
  }
```

XDR : exemple Serveur

```
rep=registerrpc(ARITH_PROG, ARITH_VERS1, MULT_PROC, mult,
  xdr_couple, xdr_float);
  if (rep==-1){
    fprintf(stderr, "errreur registerrpc (mult)\n");
    exit(2);
  }
  rep=registerrpc(ARITH_PROG, ARITH_VERS1, SQRT_PROC, rac,
  xdr_couple, xdr_float);
  if (rep==-1){
    fprintf(stderr, "errreur registerrpc (rac)\n");
    exit(2);
  }

  svc_run();
  fprintf(stderr, "erreur sur svc_run\n");
  exit(3);
}
```

XDR : Fonctions

```
#include "exemple.h"
char * add (struct couple *p){
  static float res;
  res=p->e1+p->e2;
  return ((char *)&res);
}
```

XDR : Clients

```
#include <stdio.h>
#include "exemple.h"

main (int n, char * v []){
  float x;
  struct couple don, res;
  int op, m;

  don.e1=13.4;
  don.e2=17.1;

  m=callrpc(v[1], ARITH_PROG, ARITH_VERS1, ADD_PROC,
   xdr_couple, &don, xdr_float, &x);
  if (m==0){
    printf("%f + %f = %f\n", don.e1, don.e2, x);
  }else{
    fprintf(stderr, "erreur : %d\n", m);
  }
```

Transmission des arguments : par valeurs

- **Les IDL**
  - **Insuffisance des normes** de syntaxe abstraite et de transfert en mode message asynchrone: ASN1/BER.
    - Définition de nouveaux langages de syntaxe abstraite adaptés aux appels de procédure distante
  - **IDL « Interface Definition Language »**
    - En appel de procédure distante : génération automatique du code des souches à partir de la syntaxe abstraite
    - Les souches fabriquent la syntaxe de transfert en réalisant l'alignement des paramètres dans les messages 'marshalling'
    - Pour chaque IDL en général redéfinition d'une syntaxe de transfert **.**

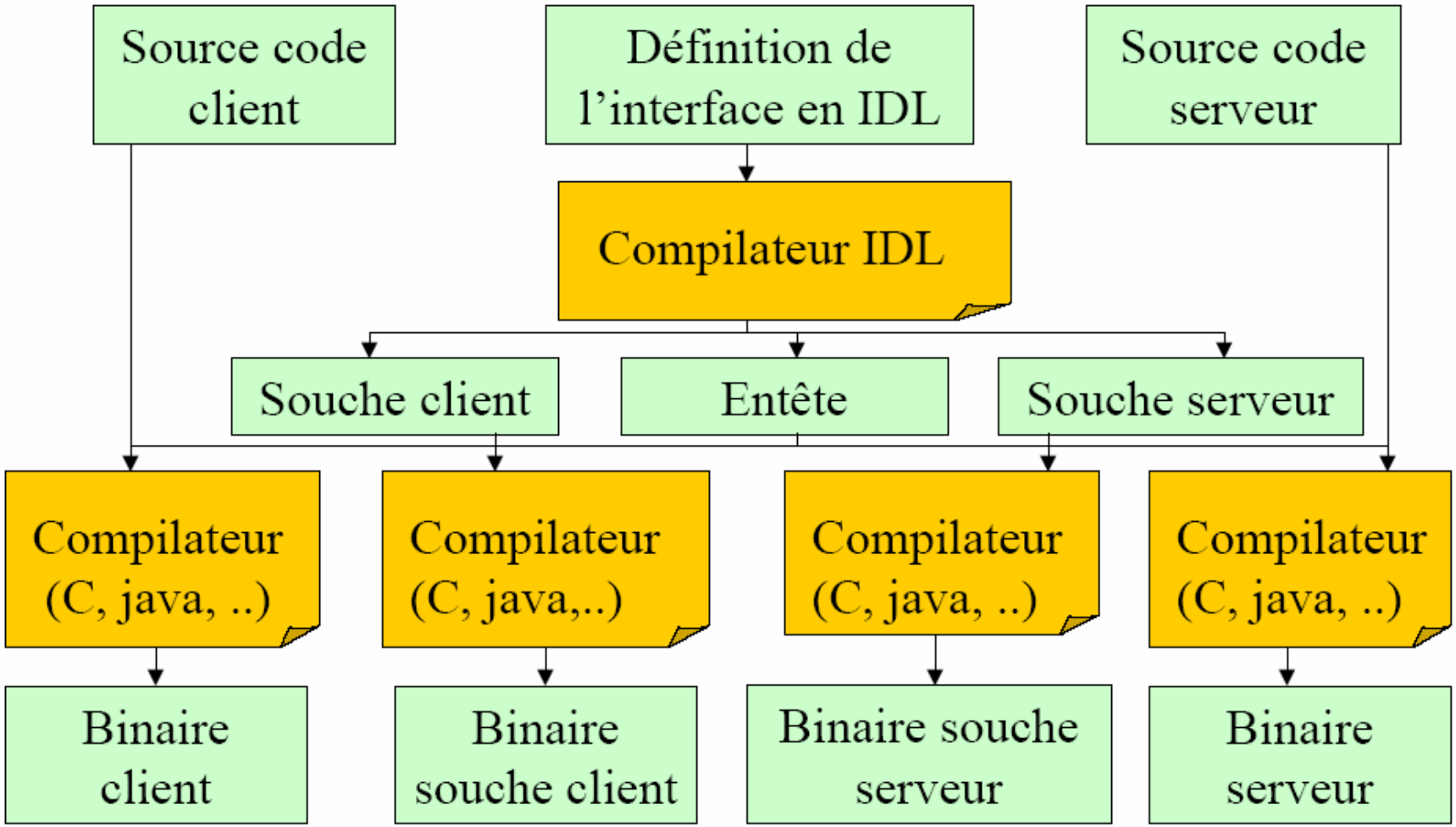
Transmission des arguments : par valeurs

- **Les IDL**
  - Être **indépendant** des langages évolués utilisant le RPC.
    - Permettre l'invocation distante **avec tout langage évolué**
    - Définition d'un **langage pivot de description de données** ayant des fonctionnalités assez riches pour les langages les plus récents.
    - Définition d'un langage pivot qui permette de **corriger les ambiguïtés** et les insuffisances des langages anciens (comme C).
    - Notion de **correspondance entre les types** retenus dans **l'IDL** et les types des différents **langages existants** ("mappings").
    - Permettre aux utilisateurs de définir un **identifiant unique** pour chaque interface afin de l'utiliser .
    - Supporter les **caractéristiques particulières des langages objets**
    - Exemple : permettre de définir des relations d'héritage entre définitions d'interfaces.
    - Éventuellement **structurer les interfaces en modules**.

Exemple IDL Corba

```
module StockObjects        {
  struct Quote {
    string symbol;
    long at_time;
    double price;
    long volume;};
  exception Unknown{};
  interface Stock {
    Quote get_quote() raises(Unknown); // lit la cotation.
    void set_quote(in Quote stock_quote); // écrit la cotation
                        };
  interface StockFactory {
    Stock create_stock(
      in string symbol,
      in string description ); };
                                  };
```

Génération des souches



Quelques exemples d'IDL et de format de présentation en RPC

- SUN ONC/RPC
  - XDR eXternal Data Representation
- OSF DCE
  - IDL DCE - Format NDR Network Data Representation
- OMG CORBA
  - IDL Corba - Format CDR Common Data Representation, Protocole IIOP
- SUN Java RMI
  - Langage Java - Protocole JRMP Java Remote Method Protocol
- Microsoft – DCOM
  - MIDL Microsoft IDL - DCOM Protocole ORPC Object RPC Format NDR
- WS-SOAP
  - WSDL Web Services Definition Language - XML.

Transmission des arguments : par adresse

- **Le passage par adresse (par pointeur)** utilise une adresse mémoire centrale du site de l'appelant qui n'a aucun sens sur l'appelé (sauf cas particulier).
- **Indispensable de traiter ce problème.**
- **Les pointeurs perdent leur signification (difficulté pour passer des structures complexe**

Interdiction totale des pointeurs

- Dans la plupart des RPC, **pas de passage des paramètres** par adresse ou de structures de données contenant des pointeurs.
- Introduit une **différence** dans le développement de procédures locales et celles destinées à un usage distant.

Passage par nom (références ou pointeurs d'objets)

- En approche objets répartis **utilisation des références d'objets.**
- Le passage d'un argument se fait en transmettant une **référence** sur l'objet en univers réparti (en CORBA IOR).
- L'accès s'effectue au **moyen des méthodes implantées par l'objet**. Par exemple accès à un attribut en lecture ou en écriture.
- **Avantages inconvénients**
  - Mécanisme universel.
  - Coûteux pour des accès fréquents.

Problème de mise en œuvre :

Désignation Liaison

Problème de mise en oeuvre

- Comment le client connaît-il l'adresse du serveur?
- Comment fonctionne la transmission des paramètres et résultats?
  - Modes de passage (valeur, référence ?)
  - Structures complexes
  - Représentation, emballage/déballage
  - Traitement de l'hétérogénéité
- Comment sont traitées les erreurs!?
  - Hypothèses sur les erreurs, mode de détection
  - Sémantique" de l'appel en cas d'erreur
- Construction des talons
  - Génération automatique
- Gestion à l'exécution
  - Lancement et arrêt du serveur, etc.

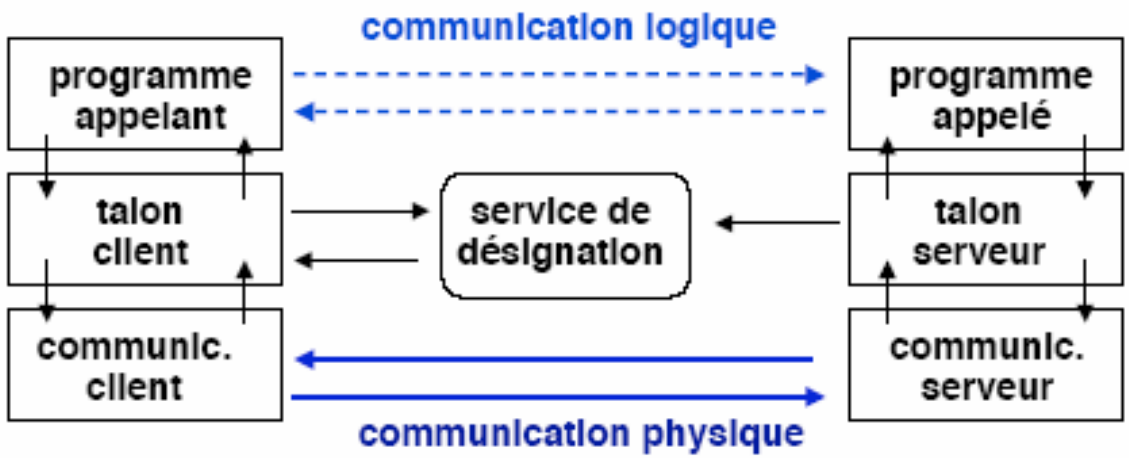
RPC : Désignation et liaison

- Objets à désigner
  - Procédure appelée, site serveur
  - Propriétés souhaitées : désignation indépendante de la localisation
    - Possibilité de reconfiguration (pannes,régulation de charge)

- Moment de liaison
  - Liaison précoce (statique) ou tardive (dynamique)
  - Statique : localisation du serveur connue à la compilation
  - Dynamique : localisation non connue à la compilation
    - Désignation symbolique des services (non liée à un site d'exécution
    - Possibilité d'implémentation ou de sélection retardée

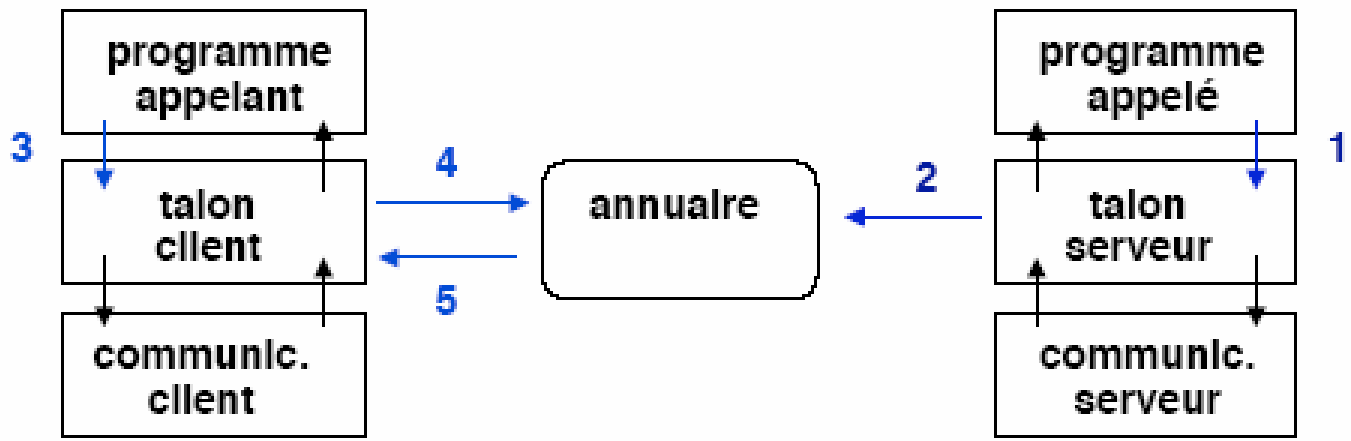
RPC : Désignation et liaison

- **Liaison statique** : pas d'appel à un serveur de noms (ou appel à la compilation
- **Liaison au premier appel** : consultation du serveur de noms au premier appel seulement
- **Liaison à chaque appel** : consultation du serveur de noms à chaque appel



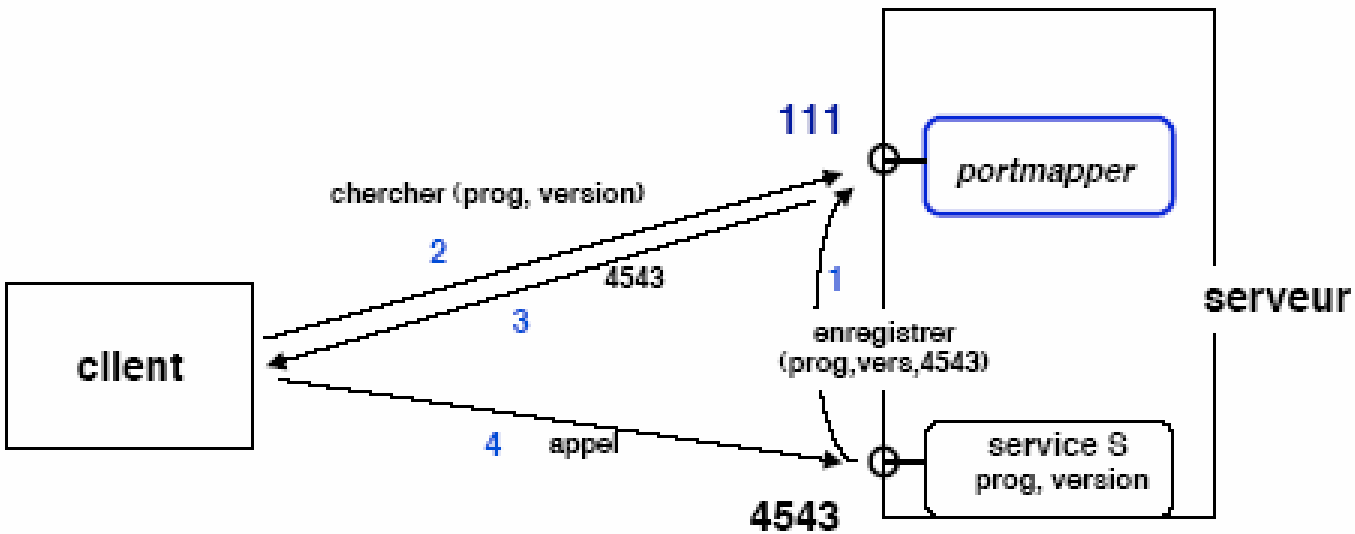
RPC : désignation et liaison utilisant un annuaire

- 1, 2 : le serveur s'enregistre auprès de l'annuaire avec <nom,adr.serveur, n° port>
- 3, 4, 5 : le client consulte l'annuaire pour trouver <adr.serveur, n° port> à partir de <nom>
- L'appel peut alors avoir lieu
- Schémas plus élaborés : attributs (critères de choix)



RPC : désignation et liaison utilisant le *portmapper*

- Si le serveur est connu (cas fréquent): on peut utiliser un service de nommage local au serveur, **le portmapper**
- Un service enregistre le n° de port de son veilleur auprès du portmapper et le veilleur se met en attente sur ce port
- Le portmapper a un n° de port fixé par convention (111)



Adresse de 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 : adresses libres
0x00000000 - 0x1FFFFFFF : serveurs officiels

RPC : “sémantique” en cas de panne

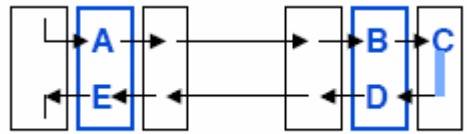
- Détection de panne = expiration de délai de garde.
- On tente de réexecuter l’appel. Combien de fois la procédure a-t-elle été exécutée?
- La sémantique dépend des hypothèses de pannes et du mécanisme de reprise :
  - Indéfini (pas d’hypothèses)
  - Au moins une fois (appel répété, plusieurs exécutions possibles, au moins une réussit)
    - acceptable si opération idempotente (2 appels successifs ont le même effet que 1 appel)
  - Au plus une fois (0 ou 1 fois)
    - on évite les exécutions répétées, mais on ne garantit pas le succès
  - Exactement une fois (c’est l’idéal, difficile à atteindre)

RPC : traitement des défaillances

- Hypothèses de défaillances
- Système de communication
  - Congestion du réseau, messages retardés
  - Perte de messages
  - Altération de messages
- Serveur
  - Défaillance avant l'exécution de la procédure
  - Défaillance pendant l'exécution de la procédure
  - Défaillance après l'exécution de la procédure
  - Défaillance définitive ou reprise possible
- Client
  - Défaillance pendant le traitement de la requête

RPC : traitement des défaillances

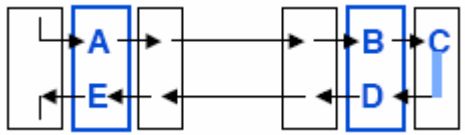
- Congestion du réseau ou du serveur
  - Panne transitoire (ne nécessite pas d'action)
  - Détection : expiration du délai de garde A ou D
  - Reprise (délai de garde A)
  - Le talon client (A) réémet l'appel (même identificateur) sans intervention de l'application
  - Le service d'exécution (C) détecte que c'est une réémission
    - Appel en cours : aucun effet
    - Retour déjà effectué : réémission du résultat



- Reprise (délai de garde D) : réémission du résultat
- Sémantique
  - Si défaillance transitoire : exactement une fois
  - Si défaillance permanente : détection, exception vers application

RPC : traitement des défaillances

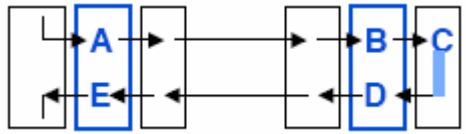
- Panne du serveur après émission de l'appel
  - Plusieurs moments possibles
    - Avant B, entre C et D, entre fin traitement et D
    - Traitement : pas fait, partiel, total



  - Détection : expiration du délai de garde A
  - Reprise
    - Le client réémet l'appel dès que le serveur redémarre
    - Sémantique : au moins une fois
      - Le client ne connaît pas l'endroit de la panne
    - On peut faire mieux avec un service transactionnel
      - Mémorise identificateur de requête et état avant exécution

RPC : traitement des défaillances

- Panne du client après émission de l'appel



  - L'appel est correctement traité
    - Changement d'état du serveur
    - Le processus exécutant courant est déclaré "orphelin"
  - Détection : expiration du délai de garde D, n réémissions infructueuses
    - Action du serveur : élimination des orphelins
      - Consomment des ressources
  - Reprise (après redémarrage du client)
    - L'application cliente réémet l'appel (id. différent)
      - Sémantique : au moins une fois
      - Le serveur ne peut pas détecter qu!'il s!'agit d'une répétition
      - Pas d!'incidence si idempotent
      - On peut faire mieux, si le client a un service de transactions

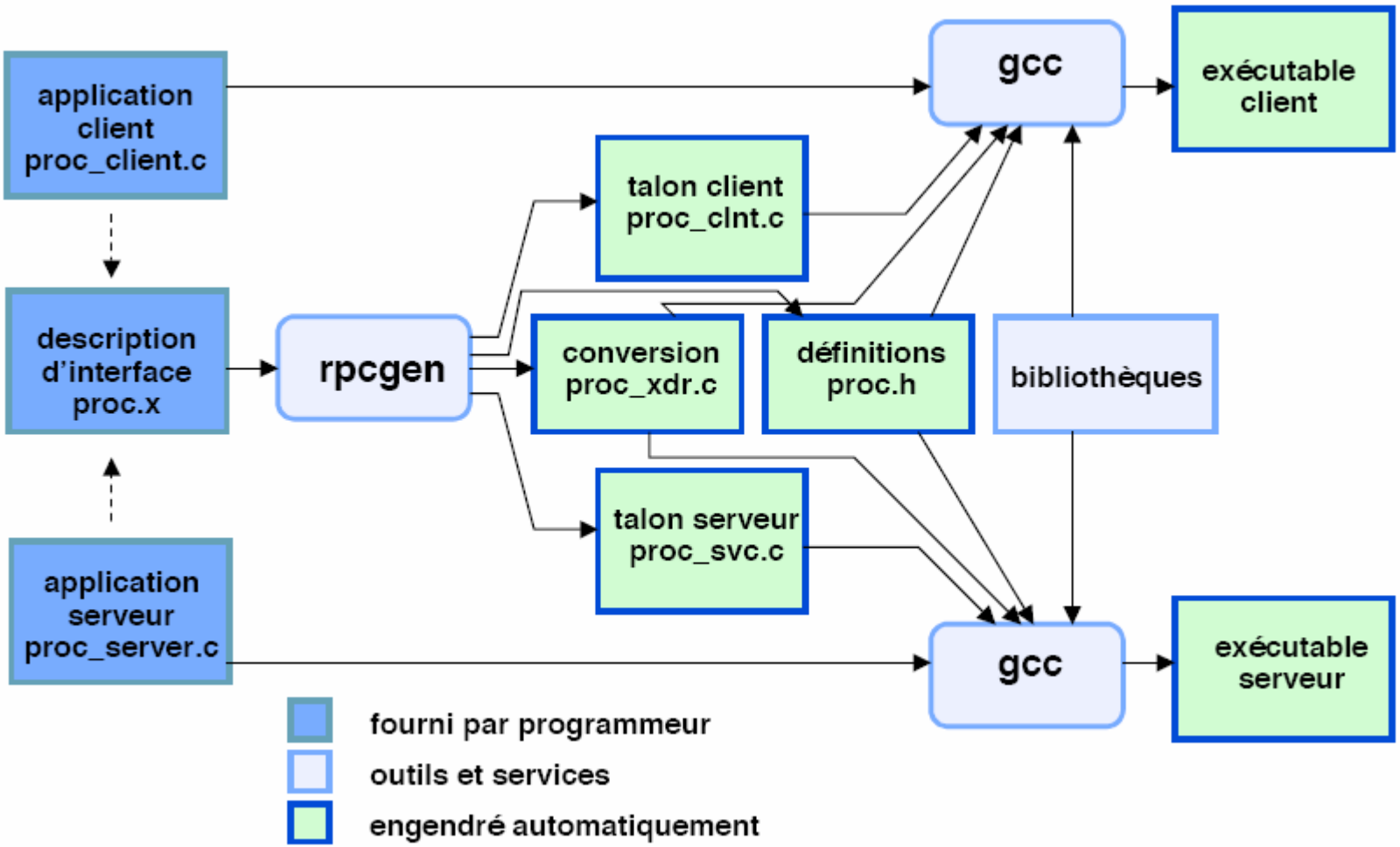
RPC : traitement des défaillances

- Solution de Nelson :
  - Extermination : Le client utilise un journal de trace et tue les orphelins : solution coûteuse en espace et complexe
  - Réincarnation : Définition de périodes d'activité incrémentale du client. Après une panne, il diffuse un message indiquant unenouvelle 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 ?!!!

Génération des talons

- Principe
  - La structure des talons est bien définie et ils peuvent être créés automatiquement
  - Renseignements nécessaires
    - Dépendants de l'environnement local : procédures de conversion, protocole de communication
    - Dépendants de l'application : formats des paramètres et résultats (pour l'emballage-déballage); c'est l'interface de la procédure
- Mise en oeuvre
  - Les informations nécessaires dépendantes de l'application sont décrites dans un langage de définition d'interfaces (IDL)

Génération des talons



Génération des interfaces

- Interface = “contrat” entre client et serveur
  - Définition commune abstraite
    - Indépendante d’un langage particulier (adaptée à des langages multiples)
    - Indépendante de la représentation des types
    - Indépendante de la machine (hétérogénéité)
  - Contenu minimal
    - Identification des procédures (nom, version)
    - Définition des types des paramètres, résultats, exceptions
    - Définition du mode de passage (IN, OUT, IN-OUT)
  - Extensions possibles
    - Procédures de conversion pour types complexes
    - Propriétés non-fonctionnelles (qualité de service), non standard)

Exemple : Génération des interfaces

fichier “annuaire.x”

```
/* constantes et types */

const max_nom = 20 ;
const max_adr = 50 ;
const max_numero = 16 ;

typedef string typenom<max_nom>
typedef string typeadr<max_adr>
typedef string
        typenumero<max_numero>

struct personne {
        typenumero numero ;
        typenom nom ;
        typeadr adresse ;
        } ;

typedef struct personne personne ;
```

```
/* description de l 'interface */

program ANNUAIRE {
   version V1 {
        void INIT(void) = 1;
        int AJOUTER(personne p) = 2 ;
        int SUPPRIMER (personne p) = 3 ;
        personne CONSULTER (typenom nom) = 4 ;
   } = 1 ;
} = 0x23456789 ;
```

Exemple : Génération des interfaces

Fichiers engendrés à partir de annuaire.x :

annuaire.h	include
annuaire_clnt.c	talon client
annuaire_svc.c	talon serveur
annuaire_xdr.c	proc. conversion

Aides à la construction :

annuaire_client.c	modèle de programme client
annuaire_server.c	modèle de programme client
makefile.annuaire	modele de makefile

Illustration : annuaire_xdr.c

```
bool_t xdr_typeadr(xdrs, objp)
   register XDR *xdrs;
   typeadr *objp;
{
   register long *buf;

   if (!xdr_string(xdrs, objp, max_numero))
            return (FALSE) ;
   return (TRUE) ;
}
```

```
bool_t xdr_typenom(xdrs, objp)
...
bool_t xdr_typenumero(xdrs, objp)
...
bool_t xdr_personne(xdrs, objp)

  register XDR *xdrs;
  typeadr *objp;
{
  register long *buf;

   if (!xdr_typenumero(xdrs, &objp->numero))
            return (FALSE) ;
   if (!xdr_typenom(xdrs, &objp->nom))
            return (FALSE) ;
   if (!xdr_typeadr (xdrs, &objp->adresse))
            return (FALSE) ;
   return (TRUE) ;
}
```

Utilisation de rpcgen

Exemple : modèle de programme client : annuaire_client.c

```
...
main(argc, argv)
int argc ; char * argv ;
{
        CLIENT *clnt ;
        char * host
...
        if (argc <2) {
                printf("usage: %s server_host\n", argv[0]) ;
                exit(1)
        {
        host = argv[1] ;

        clnt = clnt_create (host, ANNUAIRE, V1, "netpath") ;    /* "poignée" d 'accès au serveur */
        if (clnt == (CLIENT *) NULL {
                {clnt_pcreateerror(host) ;
                exit(1) ;
        {
...                                                             /* saisir paramètres */
        result_2 = ajouter_1(&ajouter_1_arg, clnt)
        if (result_2 == (int *) NULL) {
                clnt_perror(clnt, "call failed") ;
        {
...                                                             /* afficher résultats */
}
```

Service d'enregistrement des services RPC

- Le démon **portmap** (**rpcbind**) est responsable de l'enregistrement des services RPC
  - service RPC de numéro 100000 associé statiquement aux ports UDP et TCP 111.

- Le fichier /etc/rpc
  - contient la liste des programmes RPC référencés sur la machine

nom	**numéro**	**alias**
rpcbind	100000	portmap sunrpc rpcbind
rstatd	100001	rstat rup perfmeter
rusersd	100002	rusers
nfs	100003	nfsprog
mountd	100005	mount showmount

Service d'enregistrement des services RPC

- La commande rpcinfo
  - réalise des compte-rendus sur les services RPC.
  - effectue un appel vers un serveur RPC est rapporte toutes les informations trouvées.

  - la commande rpcinfo –p
    - permet de visualiser la table de correspondance géré par le démon portmap

      $ rpcinfo –p [host]

program	**vers**	**proto**	**port**	**service**
100000	4,3,2	tcp	111	rpcbind
100000	4,3,2	udp	111	rpcbind
100007	3	tcp	32772	ypbind
100007	3	udp	32774	ypbind
100011	1	udp	32783	rquotad
100005	1	tcp	32862	mountd
100005	1	udp	32884	mountd

Service d'enregistrement des services RPC

- La commande rpcinfo –u[udp] –t[tcp] host prognum [vers]
  - permet de tester le service spécifié

```
$ rpcinfo –u sunserv nfs
program 100003 version 2 ready and waiting
program 100003 version 3 ready and waiting

$ rpcinfo –t sunserv nfs
program 100003 version 2 ready and waiting
program 100003 version 3 ready and waiting

$ rpcinfo –u sunserv rquota
program 1000011 version 1 ready and waiting

$ rpcinfo –t sunserv rquota
program not registred
program 1000011 is not available
```

Service d'enregistrement des services RPC

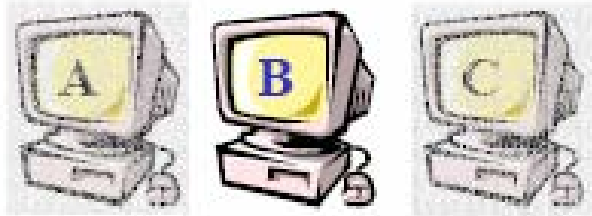
- La commande rpcinfo –u[udp] –t[tcp] host prognum [vers]
  - permet de tester le service spécifié

```
$ rpcinfo –u sunserv nfs
program 100003 version 2 ready and waiting
program 100003 version 3 ready and waiting

$ rpcinfo –t sunserv nfs
program 100003 version 2 ready and waiting
program 100003 version 3 ready and waiting

$ rpcinfo –u sunserv rquota
program 1000011 version 1 ready and waiting

$ rpcinfo –t sunserv rquota
program not registred
program 1000011 is not available
```

DCE RPC

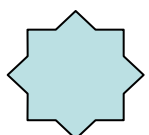
Opération souhaitée :

« res = gasp(12,v); »

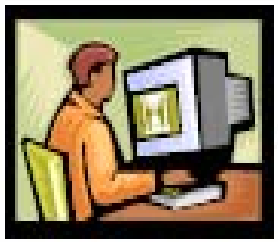


• **SUN RPC (DCE RPC) – pseudo code client (utilisant rpcgen) :**

```
CLIENT *clnt;

/* operations d'identification du serveur et                    */
/* du service « le_service_gasp »                               */
clnt = clnt_create(server_host,SERVICE_PGM_ID,
                   SERVICE_VERSION_ID,"udp");

/* prepare les arguments et appelle "le_service_gasp" distant*/
args.val1 = 12;
args.val2 = v;
res = le_service(&args,clnt);

/* detruit la connexion au service distant                      */
clnt_destroy(clnt);
```

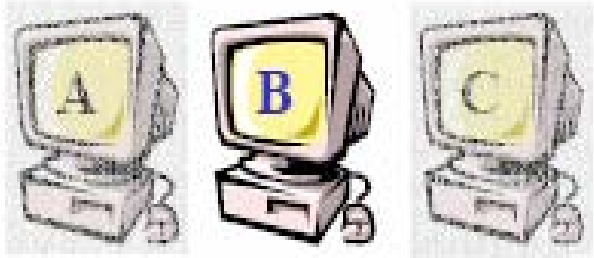



RPC Corba



Opération souhaitée :

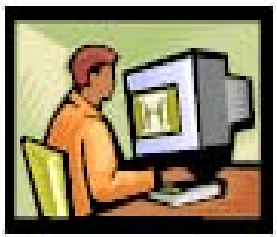
« res = gasp(12,v); »



- **RPC : Corba – pseudo code client :**

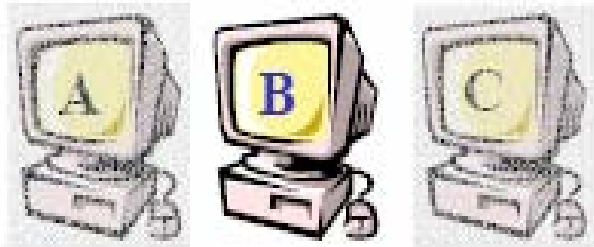
```
/* Identification du service de naming global sur le bus Corba*/
ORB orb = ORB.init();
NamingContext NameServer = NamingContextHelper.narrow(
                    orb.resolve_initial_references("NameService"));
/* operation d'identification de l'objet réalisant le service */
ClasseService ObjetService =
        ClasseServiceHelper.narrow(NameServer.resolve("s-gasp"));
/* objet (service) distant utilisé comme un objet local       */
res = ObjetService.gasp(12,v);
```

Java RMI



Opération souhaitée :

« res = gasp(12,v) ; »

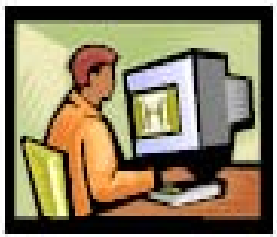


• **RPC : Java-RMI – pseudo code client :**

```
/* Recuperation de l'adresse du serveur hébergeant le service*/
/* Interrogation d'un serveur de nom global (API java        */
/* normalisée, réalisation par LDAP, DNS, …).                */
/* Besoin connaître serveur de nom (ok: un seul serveur)     */
server_address = Annuaire.GetServer("s-gasp"); /* pseudo code*/
```

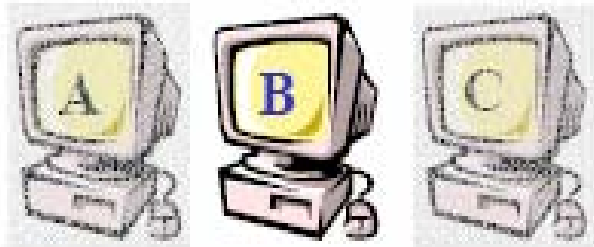
```
/* operation d'identification de l'objet réalisant le service*/
ClasseService ObjetService;    /* herite de « remote »       */
ObjetService = Naming.lookup(server_address,"s-gasp");

/* objet (service) distant utilisé comme un objet local      */
res = ObjetService.gasp(12,v);
```

Java RMI



Opération souhaitée :

« res = gasp(12,v) ; »

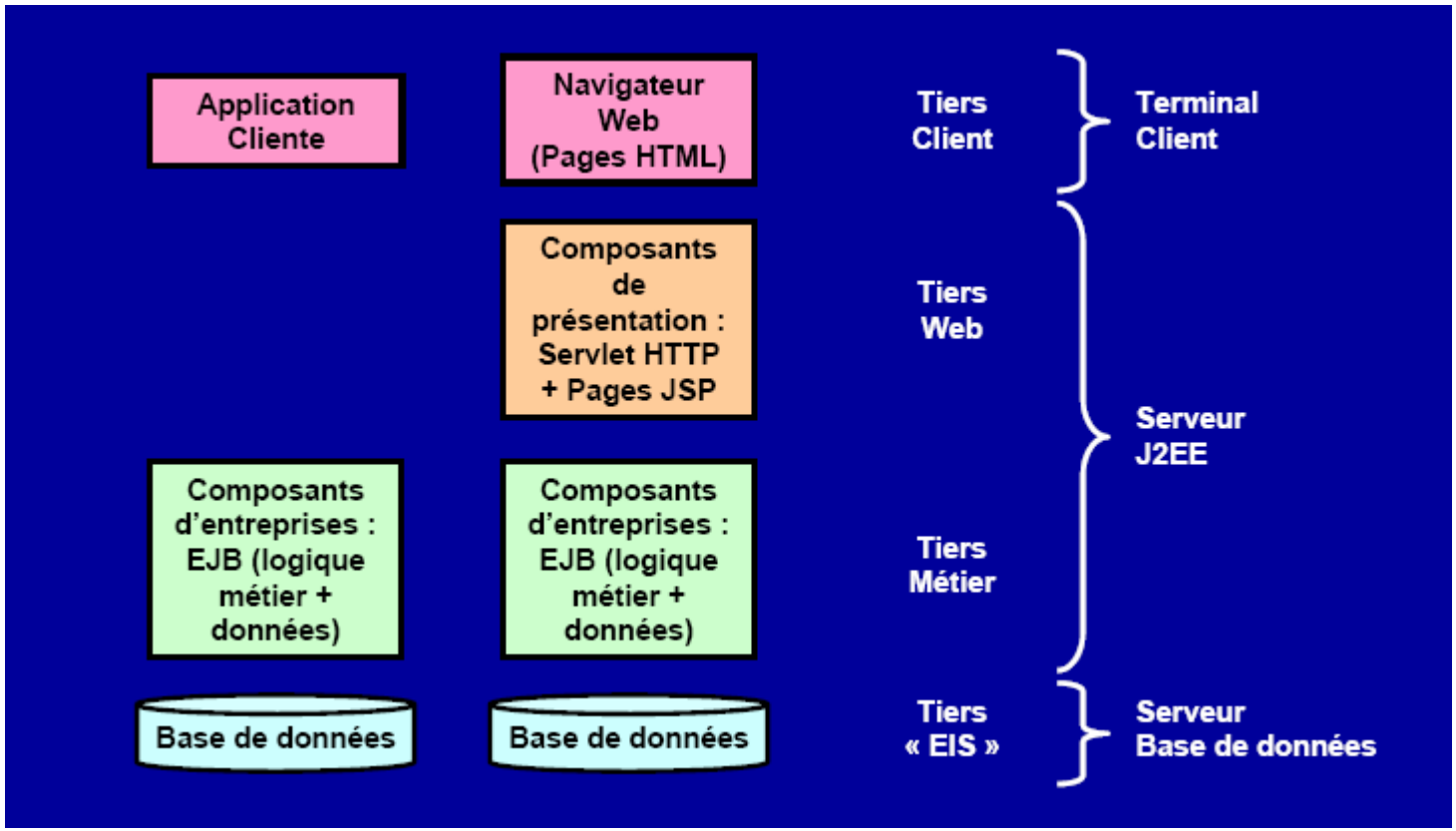


• **RPC : Java-RMI – pseudo code client :**

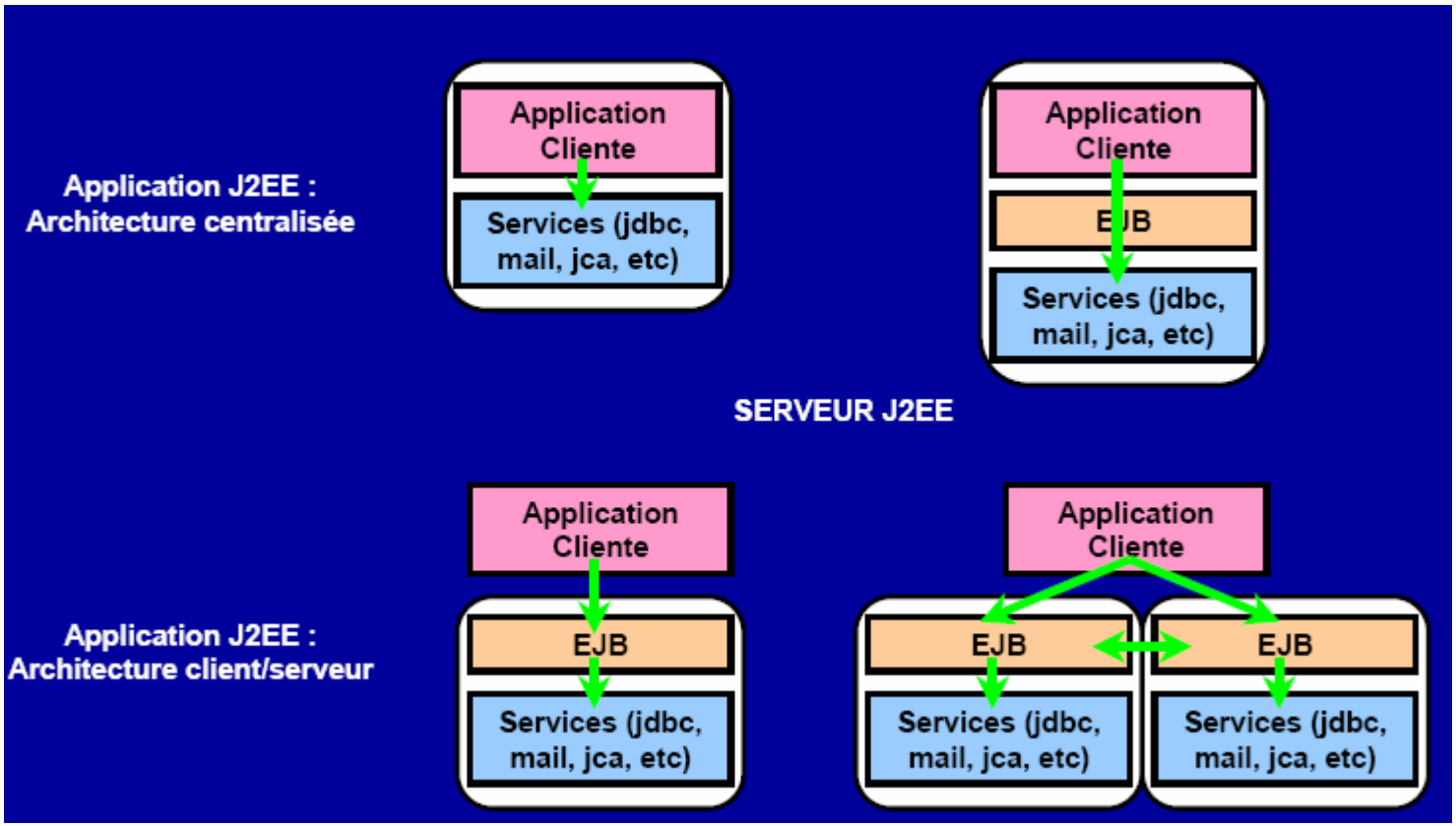
```
/* Recuperation de l'adresse du serveur hébergeant le service*/
/* Interrogation d'un serveur de nom global (API java        */
/* normalisée, réalisation par LDAP, DNS, …).                */
/* Besoin connaître serveur de nom (ok: un seul serveur)     */
server_address = Annuaire.GetServer("s-gasp"); /* pseudo code*/
```

```
/* operation d'identification de l'objet réalisant le service*/
ClasseService ObjetService;    /* herite de « remote »       */
ObjetService = Naming.lookup(server_address,"s-gasp");

/* objet (service) distant utilisé comme un objet local      */
res = ObjetService.gasp(12,v);
```

J2EE

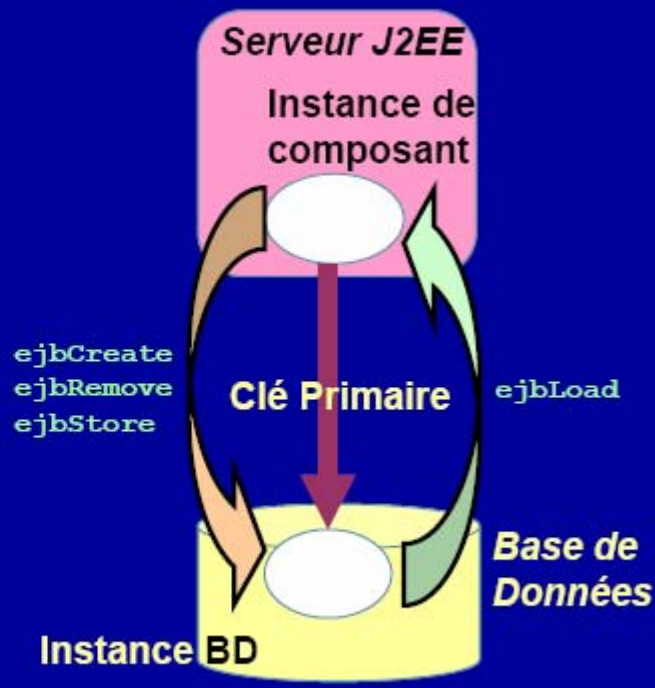

Application Cliente
Navigateur Web (Pages HTML)
Composants de présentation : Servlet HTTP + Pages JSP
Composants d'entreprises : EJB (logique métier + données)
Composants d'entreprises : EJB (logique métier + données)
Base de données
Base de données
Tiers Client
Tiers Web
Tiers Métier
Tiers « EIS »
Terminal Client
Serveur J2EE
Serveur Base de données

J2EE


Application J2EE :
Architecture centralisée
Application Cliente
Services (jdbc, mail, jca, etc)
Application Cliente
EJB
Services (jdbc, mail, jca, etc)
SERVEUR J2EE
Application J2EE :
Architecture client/serveur
Application Cliente
EJB
Services (jdbc, mail, jca, etc)
Application Cliente
EJB
EJB
Services (jdbc, mail, jca, etc)
Services (jdbc, mail, jca, etc)

Modèle de persistance des EJB entité

- **Notion de « clé primaire »**
  - Nom persistant
  - Désignation de l'instance BD
  - Définie par une classe
    - De base (String, Integer, …)
    - Composite = {champs de l'instance}
- **Notion de « finder »**
  - Définit par une interface « home »
  - Retourne une instance ou une collection d'instances
  - BMP => ejbFinder (retourne une instance ou une collection d'instances de clé primaire)



Rôle des bibliothèques

- Fonctions de bas niveau pour le *(un)marshalling*
- Etablissement des connexions réseau
- Mise en oeuvre du protocole :
  - identification du programme
  - authentification (éventuelle)
  - gestion de certaines erreurs (de protocole, de version, etc.)
  - transmission effective des données (en particulier la gestion de la mémoire)
  - publication et découverte d'un service

- Fournies par le "vendeur" de la solution RPC

Rôle des bibliothèques

- Il reste quand même des choses à écrire (!) : dépend de la solution retenue
- RPC ONC :
  - le code de la fonction pour le serveur (le code de démarrage est engendré par les outils)
- le code du client :
  - connexion au serveur
  - appel de la fonction
- les outils RPC ONC proposent en général une version à compléter de tout ce que le programmeur doit fournir
- autres solutions : en gros la même chose

Rôle des bibliothèques

- Les RPC ONC (Sun) :
- Solution classique disponible sur tous les Unix (incluse dans le système en général) et sous windows comparable aux DCE RPC en “gratuit”
- langage de description des données : XDR (rfc 1832)
- technologie utilisée par *Network File System*
- outil de base rpcgen :
  - engendre des convertisseurs (*(un)marshalling*)
  - engendre les souches (serveurs et clients)
  - propose des modèles pour le client et le serveur
  - engendre un Makefile
  - utilise un *lookup service* pour la publication et la découverte d’un service (rfc 1833)
- les RPC ONC peuvent être implantées sur n’importe quelle couche transport (en général TCP et UDP)

Rôle des bibliothèques

- Les RPC ONC (Sun) :
- Solution classique disponible sur tous les Unix (incluse dans le système en général) et sous windows comparable aux DCE RPC en “gratuit”
- langage de description des données : XDR (rfc 1832)
- technologie utilisée par *Network File System*
- outil de base rpcgen :
  - engendre des convertisseurs (*(un)marshalling*)
  - engendre les souches (serveurs et clients)
  - propose des modèles pour le client et le serveur
  - engendre un Makefile
  - utilise un *lookup service* pour la publication et la découverte d’un service (rfc 1833)
- les RPC ONC peuvent être implantées sur n’importe quelle couche transport (en général TCP et UDP)

Protocole

- Principes de base :
  - un service réseau est fournis par plusieurs programmes
  - un programme fournis plusieurs fonctions (ou procédures)
  - un client appelle une procédure d'un programme
  - une procédure est identifiée par trois entiers (positifs) :
    - le numéro de programme
    - le numéro de version du programme
    - le numéro de procédure au sein du programme
  - affectation des numéros de programme :
    - de 0x0 à 0x1fffffff : affectation par SUN
    - de 0x20000000 à 0x3fffffff : affectation par le programmeur
    - de 0x40000000 à 0xffffffff : réservé
  - le protocole est basé sur l'échange de messages décrits en XDR

Librairie RPC

- Elle se décompose en deux couches, la couche haute et la couche basse.
- La couche haute :
  - Implémentation au-dessus d'UDP.
  - Taille des messages fixée à UDPMSGSIZE
  - Pas d'authentification
  - Faible paramètrage (*timeout* de l'appel de fonction = 25 sec)
  - Peu de fonctions de manipulation (svc_run, register_rpc, callrpc)
- La couche basse :
  - Très paramétrable
  - Beaucoup de fonctions de manipulation

Couche haute coté serveur

- **int registerrpc** (u_long_ no_pg, u_long no_vers, u_long no_proçvoid * (* fonction) ( ), xdrproc_t xdr_param, xdrproc_t xdr_result);

- Paramètres :
  - no_pg : numéro du programme où enregistrer la fonction
  - no_vers : numéro de version
  - no_proc : numéro de procédure à donner à la fonction
  - fonction : pointeur sur la fonction à enregistrer
  - xdr_param : fonction d'encodage/décodage des paramètres
  - xdr_result : fonction d'encodage/décodage du résultat

- Retour :
  - 0 : en cas de succès
  - -1 : en cas d'erreur et envoi d'un message d'erreur sur stderr

- Chaque fonction est enregistrée séparément.
- Attente d'appel de fonction
  - void svc_run();

Couche haute coté client

- int **callrpc** (char * machine, u_long no_prog, u_long no_vers, u_long no_proc,
- xdrproc_t xdr_param, void *param, xdrproc_t xdr_result, void *result);
- Paramètres :
  - Avant appel
    - machine : nom de la machine où se trouve la fonction à exécuter
    - no_prog, no_vers, no_proc : identifie la fonction à appeler
    - xdr_param : filtre XDR pour les paramètres
    - param : pointeur sur le paramètre à passer à la procédure
    - xdr_result : filtre XDR pour le résultat
    - result : pointeur sur zone réservée pour stocker le résultat de la RPC
  - Aprés appel
    - result : pointe sur le résultat de la fonction distante.
    - Retour :
      1. 0 : en cas de succès
      2. autre en cas d'erreur (casté en clnt_stat et passé à la fonction clnt_perrno, il y a affichage de l'erreur sur stderr)

Couche basse coté serveur

- Le type SVCXPTR
  - Type opaque (comme FILE) qui contient les informations concernant le serveur :
    - xp_sock : descripteur de la socket
    - xp_port : port d'attachement
    - xp_ops : pointeur sur structure contenant les fonctions de gestion des appels
    - xp_addrlen, xp_verf : champ de sécurité
  - Création d'un service UDP
    - SVCXPRT * svcudp_create(int desc);
    - Permet de d'initialiser un structure SVCXPRT pour un nouveau service UDP.
    - Paramètres :
    - desc : descripteur d'une socket préalablement créée ou RPC_ANYSOCK (création + bind automatique)
    - Retour :
      - pointeur sur structure permettant de manipuler le service
      - NULL : en cas d'erreur

Couche basse coté serveur

- Le type SVCXPTR
  - Création du service TCP
    - **SVCXPRT *svctcp_create** (int desc, u_int snd_siz, u_int rcv_siz);
    - Permet de d'initialiser un structure SVCXPRT pour un nouveau service TCP.
    - Paramètres :
      - desc : descripteur d'une socket TCP déjà créée ou RPC_ANYSOCK (socket, bind, listen automatique)
      - snd_siz, rcv_siz : taille des buffers de réception et d'envoi pour xdrrec_create() (si 0 => 4000par défaut)
    - Retour :
      - pointeur sur structure permettant de manipuler le service
      - NULL : en cas d'erreur
  - **Destruction d'un service**
    - void svc_destroy(SVCXPRT *svc_pt);
    - Permet de libérer les ressources associées à la structure SVCXPRT.
    - Paramètres :
    - svc_pt : pointeur sur service à détruire
    - Remarque : ferme la socket si elle a été ouverte par la librairie RPC

Couche basse coté serveur

- Enregistrement d'un service
  - bool_t svc_register(SVCXPRT *svc_pt,u_long no_prog,u_long no_vers,void (*dispatch)(), u_long proto);
  - Enregistre le service auprès du deamon portmap
  - Paramètres :
    - svc_pt : pointeur sur le service à enregistrer
    - no_prog, no_vers : numéro du programme et version du service à enregistrer
    - dispatch : fonction de répartition à appeler après la réception d'une requête
    - proto : IPPROTO_TCP ou IPPROTO_UDP entraînent la création du service et l'enregistrement dans le deamon portmap
  - Retour :
    - TRUE : en cas de succès
    - FALSE : en cas d'échec
    - Appeler pmap_unset avant, afin de désenregistrer le service auprès du *deamon* portmap afin d'éviter les erreur
  - Le lancement du serveur
    - void svc_run(void);
    - Démarre l'attente du serveur.

Couche basse coté client

- Le type CLIENT est un type opaque contenant des informations concernant le client, en particulier :
  - cl_auth : champ de sécurité
  - cl_ops : pointeur sur structure contenant les fonctions de gestion des appels.
- **Création générique d'un client**
- CLIENT *clnt_create(char *machine,u_long no_progu_long no_vers,char* protocole);
- Permet de créer un client attaché à un service particulier sur une machine particulière.
- Paramètres :
  - machine : nom de la machine sur laquelle se trouve le service
  - no_prog, no_vers : numéro de programme et de version du service
  - protocole : nom du protocole (" udp " ou " tcp ") ` utilisé par le service
- Retour :
  1. pointeur sur structure permettant d'utiliser le client
  2. NULL : en cas d'échec

Couche basse coté client

- **Création d'un client UDP**
  - CLIENT *clntudp_create(struct sockaddr_in *adresse_pt,u_long no_prog,u_long no_vers,struct timeval retry,int *socket_pt);

Paramètres :

1. adresse_pt : pointeur sur l'adresse du service si le champ sin_port==0 ce champ est rempli automatiquement grâce au deamon portmap
2. no_prog, no_vers : numéro de programme et de version du service
3. retry : temps d'attente avant réemission <sys/time.h>
4. socket_pt : pointeur sur zone mémoire contenant avant l'appel le descripteur d'une socket UDP déjà créée ou RPC_ANYSOCK, après appel elle contient le descripteur de la socket du client

Couche basse coté client

- **Création d'un client TCP**
- CLIENT *clnttcp_create(struct sockaddr_in *adresse_pt,u_long no_prog,u_long no_vers,int *socket_pt,u_int snd_siz, u_int rcv_siz);

- Paramètres :
  - adresse_pt : pointeur sur l'adresse du service si le champ sin_port==0 ce champ est rempli
  - automatiquement grâce au deamon portmap
  - no_prog, no_vers : numéro de programme et de version du service
  - socket_pt : pointeur sur zone mémoire contenant avant l'appel le descripteur d'une socket UDP déjà créée ou RPC_ANYSOCK, après appel elle contient le descripteur de la socket du client
  - snd_siz, rcv_siz : tailles des buffers d'envoie et de réception, si 0 valeur par défaut (4000)

Couche basse coté client

- Appel d'une procédure distante
- enum clnt_stat clnt_call(CLIENT * cl_pt, u_long no_proc, xdrproc_t
- xdr_param, void *param, xdrproct xdr_result, void * result, struct timeval
- timeart);
- Paramètres :
  - 1. cl_pt : pointeur sur le client
  - 2. no_proc : numéro de la procédure à appeler
  - 3. xdr_param : filtre XDR pour l'encodage du paramètre
  - 4. param : pointe sur le paramètre
  - 5. xdr_result : filtre XDR pour décoder le résultat
  - 6. result : pointe sur une zone mémoire réservée pour recevoir le résultat
  - 7. timeout : temps d'attente avant échec, ignoré si positionné avec clnt_control()

  - RPC_SUCCESS : en cas de succès…

RPCGEN

- Génère des fichiers serveurs/client /XDR à partir d'une spécification minimale
- Par convention on utilise des fichiers avec l'extension .x -> rpcgen prog.x
- L'outil rpcgen produit :
  - un fichier d'en-tête : prog.h
  - deux stubs : prog_clnt.c et prog_svc.c
  - un filtre XDR : et prog_xdr.c
- Le programmeur produit :
  - serveur.c
  - client.c
- Le compilateur produit :
  - le client avec : client.c, prog_clnt.c, prog_xdr.c
  - le serveur avec : serveur.c, prog_svc.c, prog_xdr.c.

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*

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 Général d'un 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

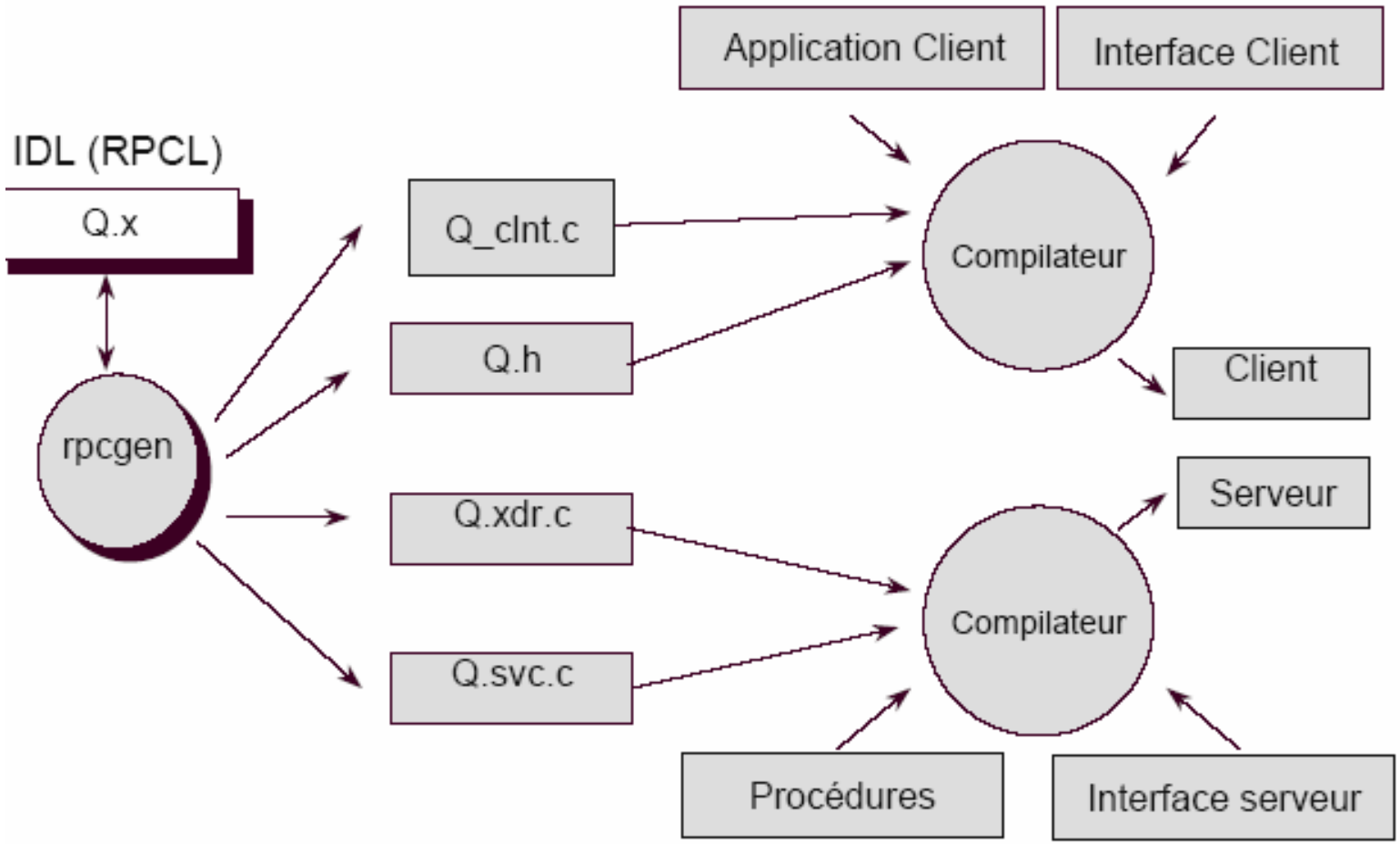
- Ecrire le contrat toto.x dans le langage RPCL
  - Compiler le contrat avec RPCGEN
    - toto.h : déclarations des constantes et types utilisés dans le code généré pour le client et le serveur
    - toto_xdr.c : procédures XDR utilisés par le client et le serveur pour encoder/décoder les arguments,
    - 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

Méthodologie de développement

- 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
  - Client «host» «liste d'arguments»

RPCGEN


Application Client
Interface Client
IDL (RPCL)
Q.x
Q_clnt.c
Compilateur
Q.h
Client
rpcgen
Q.xdr.c
Serveur
Compilateur
Q.svc.c
Procédures
Interface serveur