# Appel de procédures distantes

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

SRCS – Master 1 SAR 2019/2020

# Exemple fil rouge

## Objectif

Développer un service pour calculer des divisions.

## Spécifications de fonctionnalités

- Division classique entre deux flottants renvoyant un flottant
- Division entière entre deux entiers renvoyant deux valeurs : quotient et reste

# Ce que nous offre le réseau

## Protocole de transport

Couche réseau permettant :

- la communication de bout en bout entre deux processus distant
- le multiplexage des communications entre deux machines

## Identification d'un processus sur le réseau

- Identification de la machine : adresse IP ou nom DNS
- Un identifiant locale à la machine : numéro de port

# Les protocoles de transport

## User Datagram Protocol

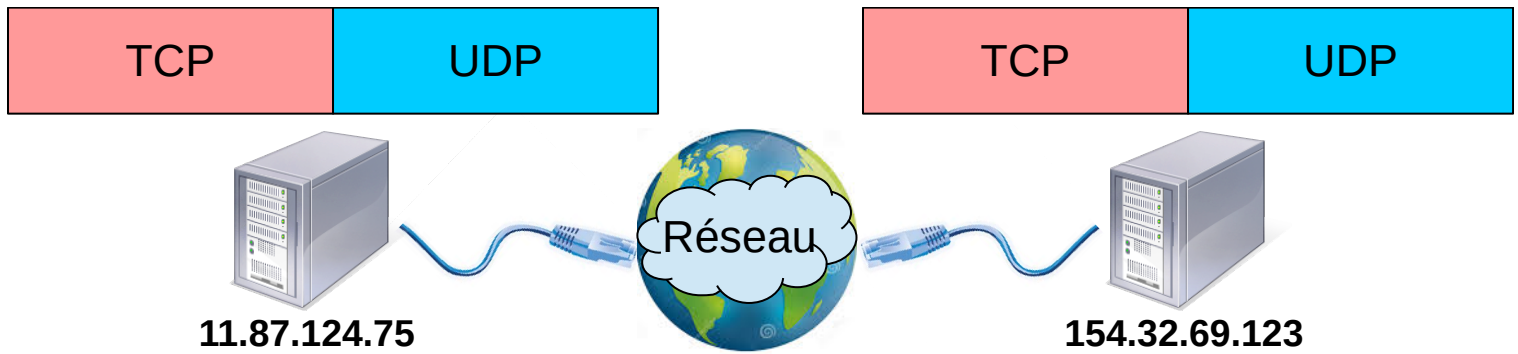
- Non connecté
- Non fiable

Services UDP connus : echo (1), daytime (13), DNS (53), sunrpc (111)

## Transport Control Protocol

- Connecté
- Communication fiable et FIFO

Services TCP connus : ftp (20,21), ssh (22), http (80), sunrpc (111)



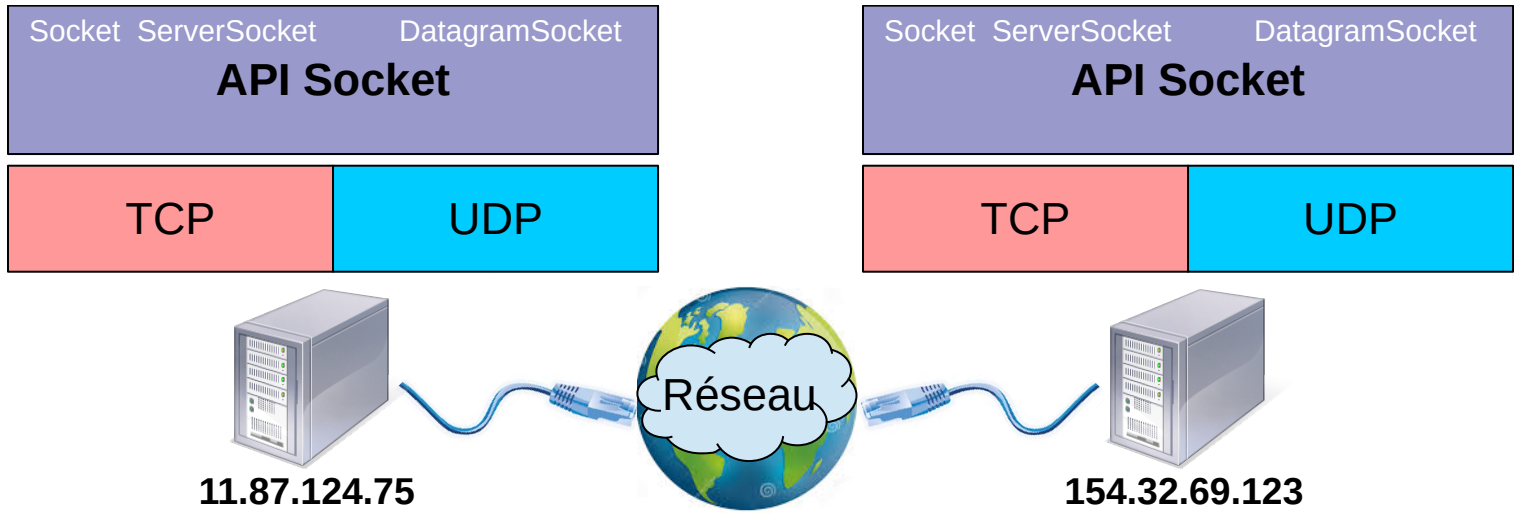
# Les sockets

## Définition

API système permettant la communication bidirectionnelle entre deux processus pouvant être distant.

## Rappels API java

- `Socket` : socket de communication en mode connecté
- `ServerSocket` : socket d'écoute de connexion
- `DatagramSocket` : socket de communication en mode message



# Notre fil rouge avec les sockets

## Ma TODO list

- Décider d'un protocole applicatif :
  - Contenu des messages et de leur format
  - Enchaînement/séquence des messages
  - Cas d'erreur
- Coder la partie métier du serveur
- Coder la partie communication du serveur :
  - instanciation de la socket d'écoute
  - Réception des requêtes :
    - affecter la requête à un thread
    - désérialiser la requête
    - aiguillage vers le bon traitement
    - sérialiser et renvoyer la réponse au client
- Coder l'envoi de requêtes et la réception des réponses coté client

# Fil rouge : Messages du protocole applicatif

```java
public class Div implements
    Serializable {

 public final double a;
 public final double b;

 public Div(double a, double b){
  this.a=a;
  this.b=b;
 }
}
```

```java
public class DivEnt implements
     Serializable {

 public final int a;
 public final int b;

 public DivEnt(int a, int b){
   this.a=a;
   this.b=b;
  }
}
```

```java
public class RepDivEnt implements Serializable {

 public final int quotient;
 public final int reste;

 public RepDivEnt(int quotient, int reste){
  this.quotient=quotient;
  this.reste=reste;
 }
}
```

# Fil rouge : Code serveur (sans les try)

```
ServerSocket ss = new ServerSocket(4242);
while(!end) {
 final Socket client = ss.accept();
 new Thread(() ->{
   ObjectInputStream ois = new ObjectInputStream(client.
    getInputStream())
   ObjectOutputStream oos = new ObjectOutputStream(client.
    getOutputStream())
   Object request = ois.readObject();
   if(request instanceof Div) {
     Div d = (Div) request;
     oos.writeObject(new Double(d.a/d.b));
   }else if (request instanceof DivEnt) {
     DivEnt d = (DivEnt) request;
     oos.writeObject(new RepDivEnt(d.a/d.b, d.a%d.b));
   }else {
     oos.writeObject(new Exception("Inconnu"));
   }
   oos.flush();
 }).start();
}//fin while
```

# Inconvénients de l'API des Socket

## Une programmation trop lourde

- Peu intuitif
- Programmation dépendante des primitives réseaux (write, read)
- Gestion manuelle des fautes, des acquittements de messages
- Définition manuelle d'un protocole applicatif
- Pas de gestion de l'hétérogénéité $\Rightarrow$ code dépendant :
  - de la machine utilisée (architecture, endianness, nombre flottant)
  - du langage de programmation
  - de la technique de sérialisation
  - de l'OS (Unix, Windows, ...) si langage $\neq$ Java

**Besoin d'un modèle plus simple**

# Point de départ : Invocation de fonctionnalité locale

## Fonctionnement d'un appel de procédure/fonction

- Transfert des données : copie des arguments sur la pile
- Transfert de contrôle : jump sur la fonction appelée

## Un modèle simple

Appelant et appelé :

- même processus : en cas de faute appelant et appelé s'arrêtent
- même système : Homogénéité du langage, de la représentation des données
- même espace d'adressage : pas besoin d'I/O

# Vers un modèle d'invocation distante

## Objectif

Permettre à un processus d'invoquer un service en réalisant un appel de fonction dont l'exécution sera effectuée sur un processus distant.

## Problématiques

- Comment gérer les fautes ?
  - Que faire en cas de non-réponse ?
  - Quelle sémantique en cas ré-réémission ?
- Comment gérer l'hétérogénéité/l'interopérabilité ?
- Comment identifier les procédures sur le serveur ?

$\Rightarrow$ **Concept des Remote Procedure Call**

# Remote Procedure Call (RPC)

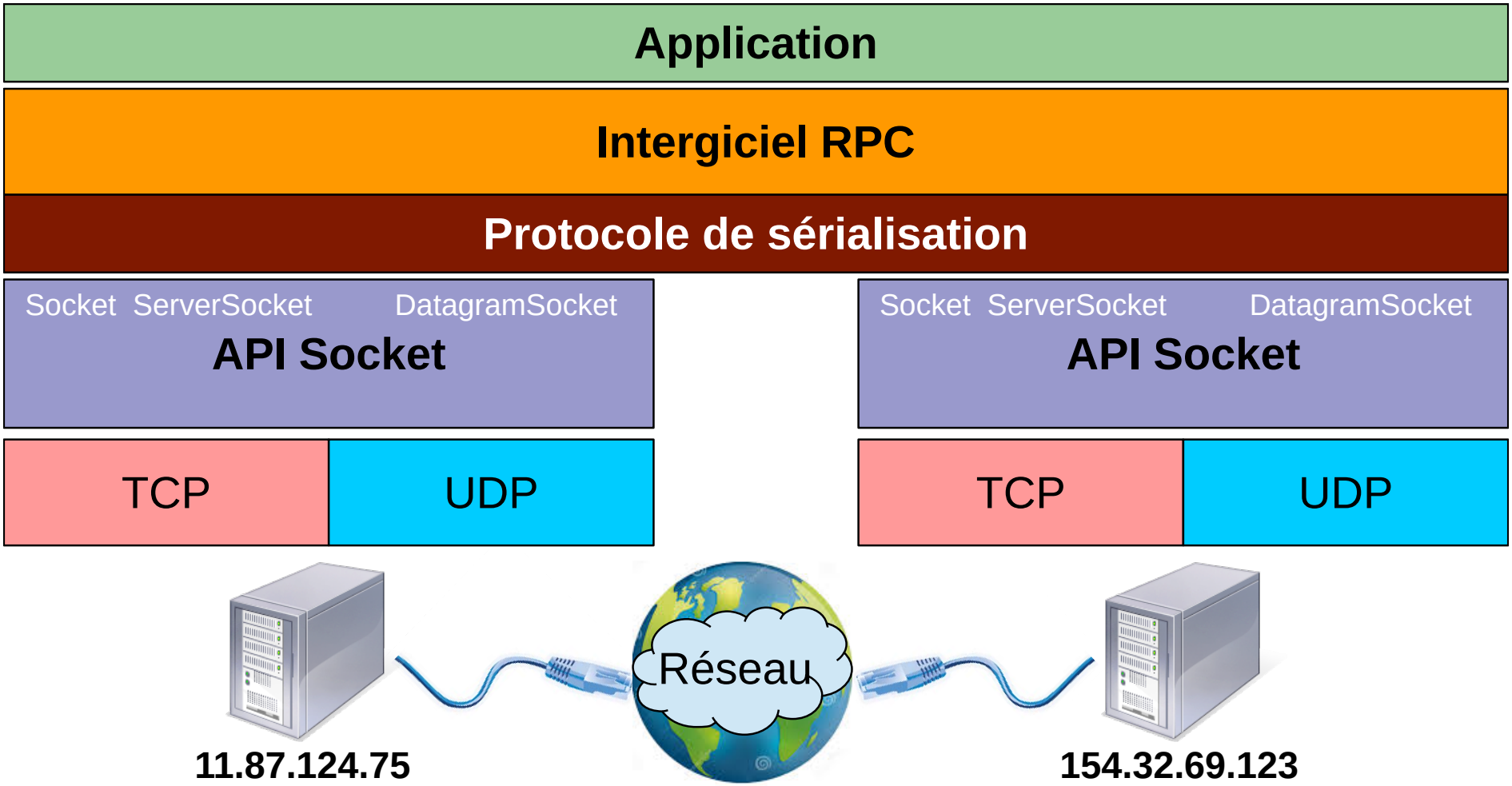
## Qu'est ce que c'est ?

Intergiciel permettant à des applications clientes d'exécuter des procédures sur une application serveur.

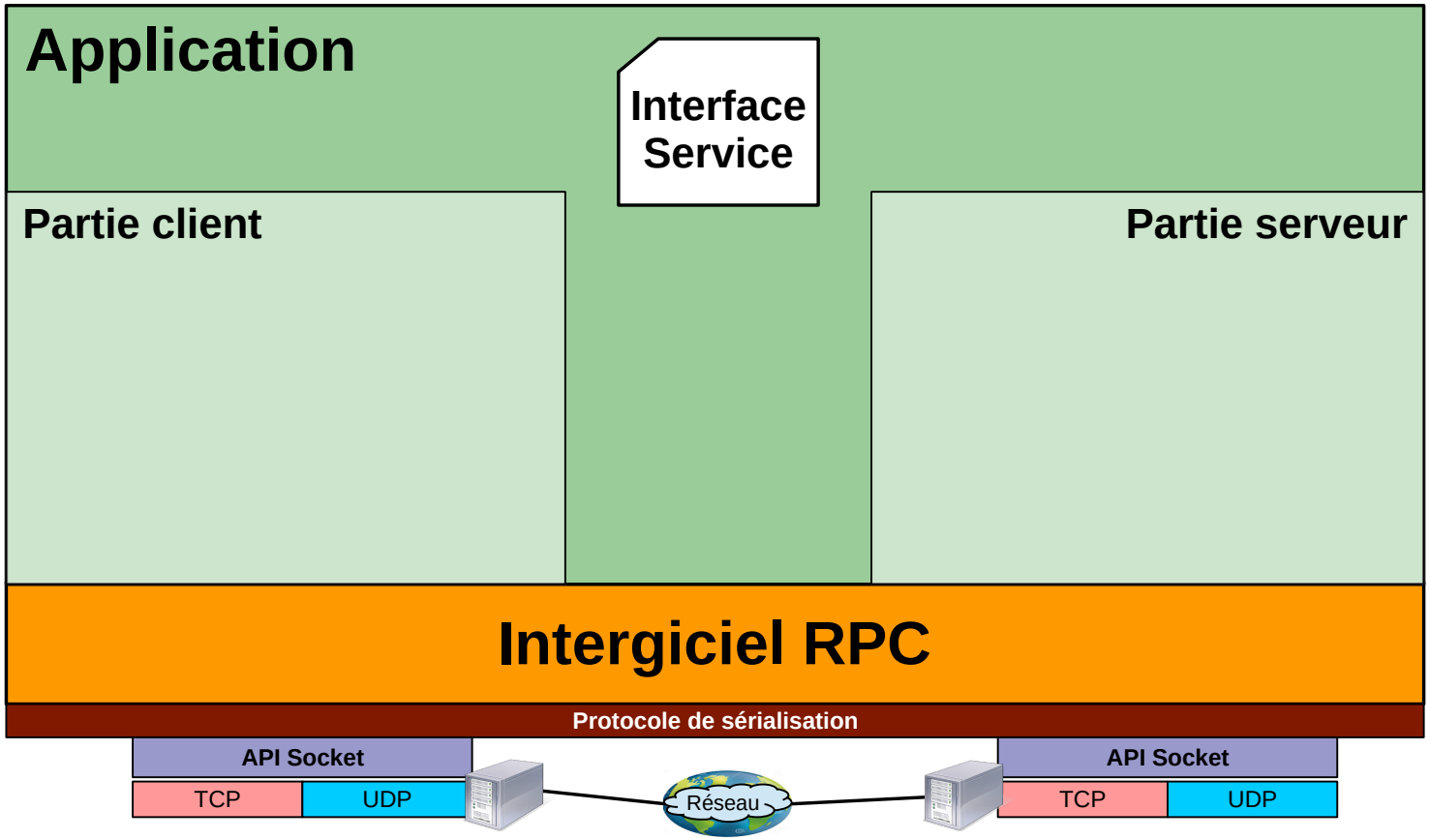
## Caractéristiques

- Séparation du code métier du code de communication
  $\Rightarrow$ Programmation simplifiée
- Associé à un outil de sérialisation
  $\Rightarrow$ Meilleure interopérabilité
  $\Rightarrow$ Standardisation de l'encodage
- Fournie des fonctionnalités complémentaires :
  ex : authentification, gestion de session, annuaire de service, etc.

# Positionnement du RPC dans la pile logicielle



# Développer une application : étape 1/4

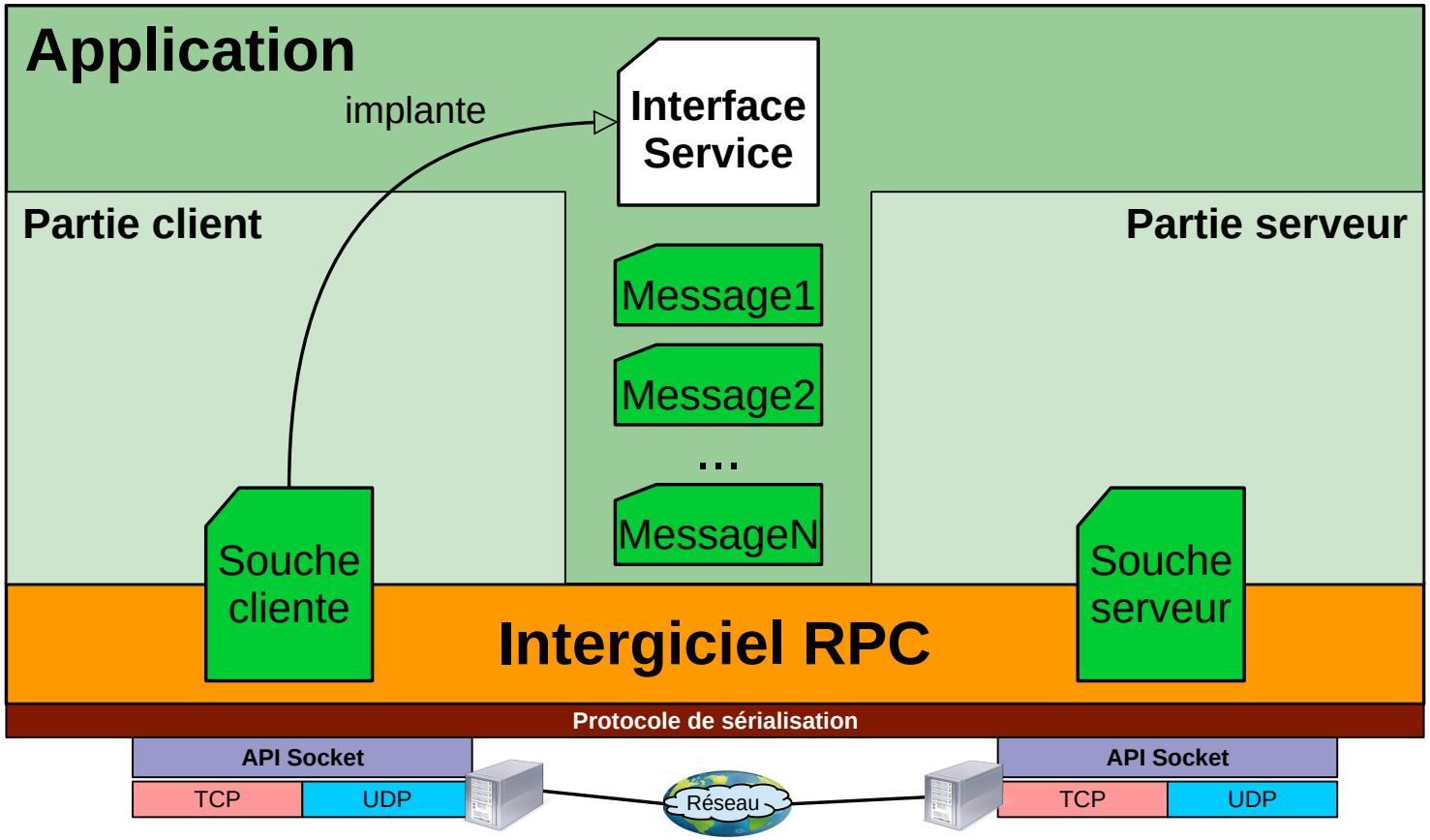



## Définir l'interface du service

À l'aide d'un **I**nterface **D**efinition **L**angage (IDL), définir :

- les prototypes des fonctions du service
- les types et structures des messages échangés (paramètres et retours)

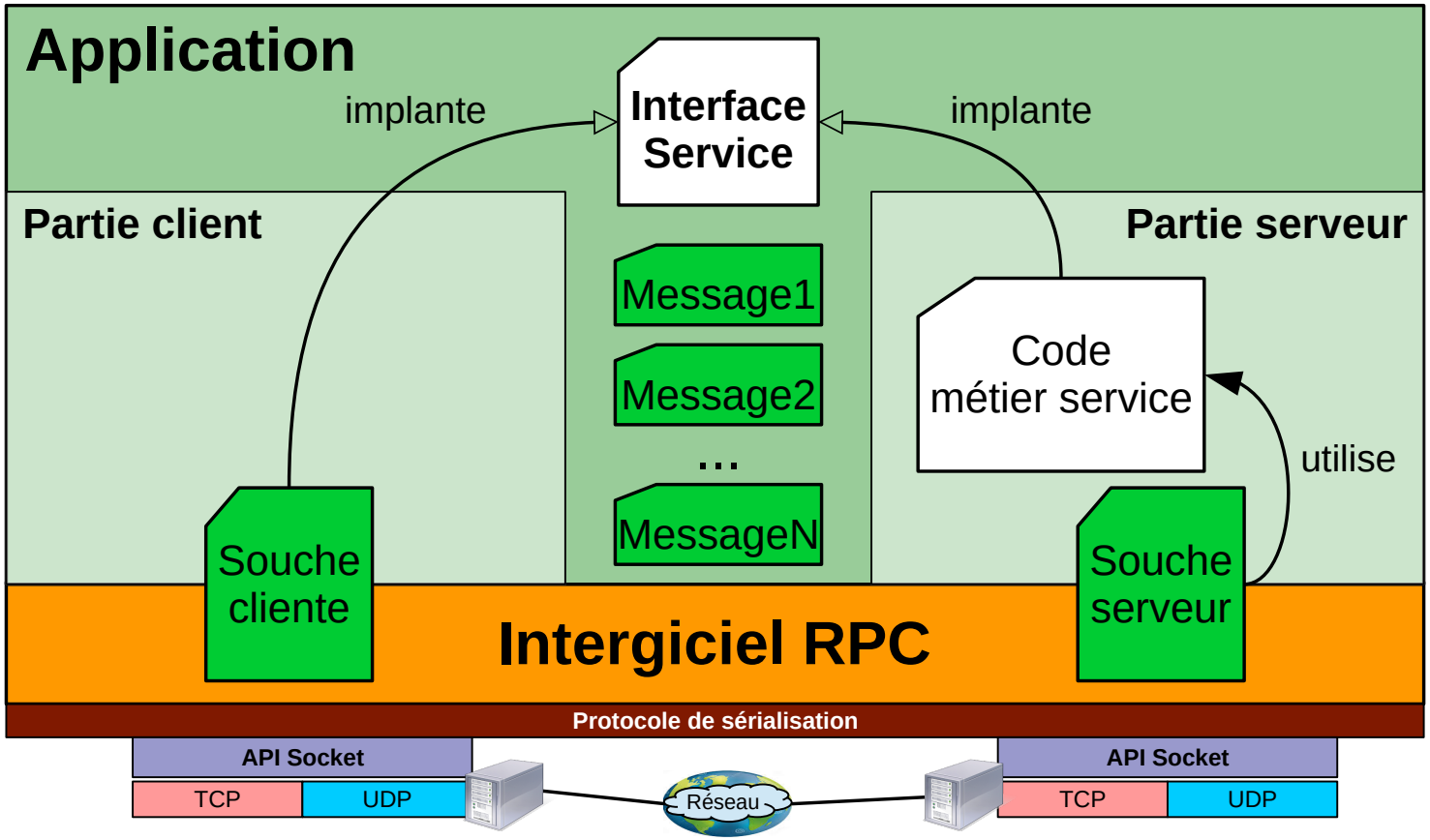
# Développer une application : étape 2/4




## Générer les souches et messages

- Compilation de l'IDL vers un langage de programmation cible
- La souche cliente implante l'interface et fait office de mandataire du serveur pour le code client (DP proxy)

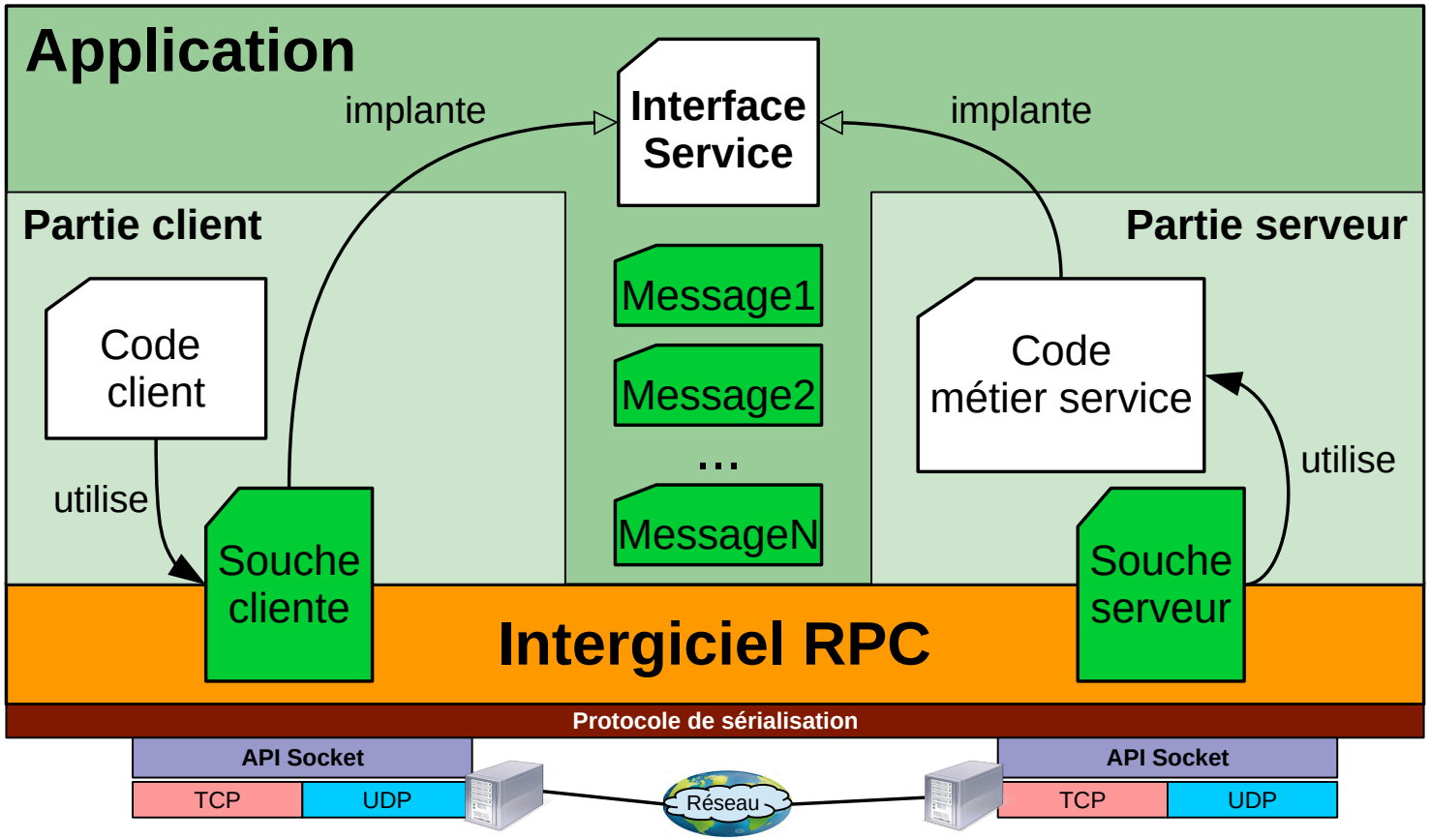
# Développer une application : étape 3/4




## Définir le code serveur

- Définir le code métier du service (implantation de l'interface)
- Lier la souche serveur à l'implémentation

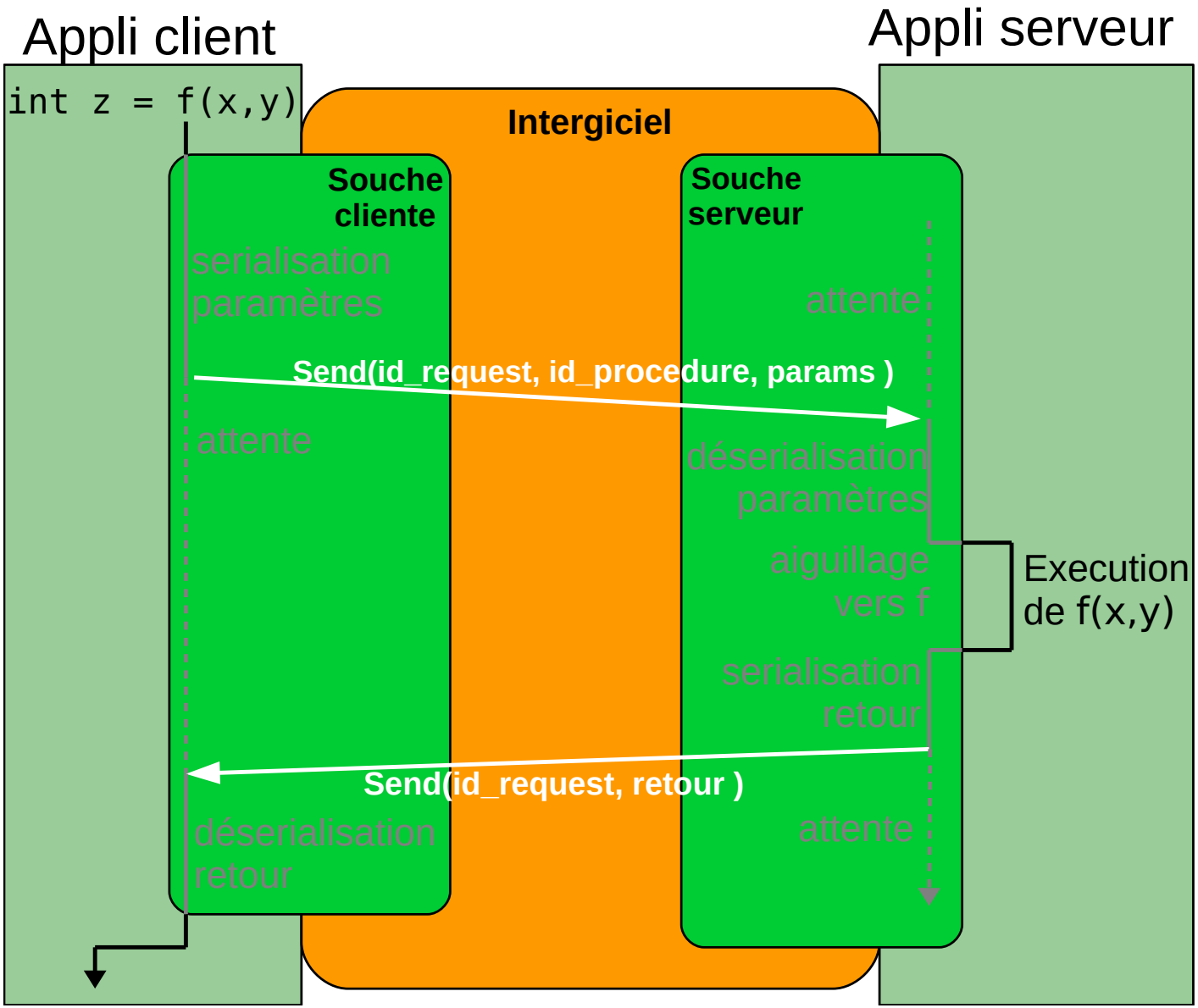
# Développer une application : étape 4/4




## Définir le code client

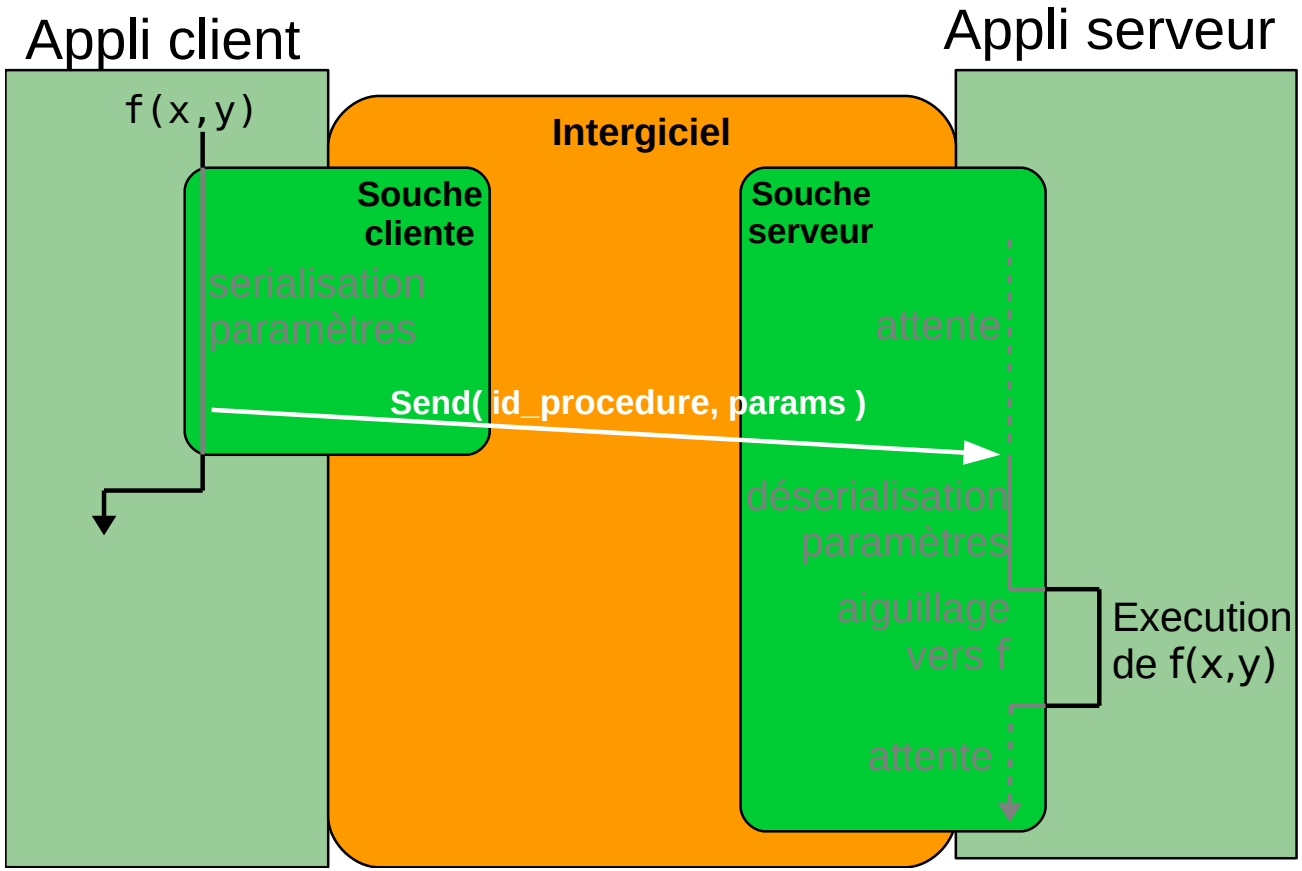
- Initialiser une souche cliente
- Appeler les services souhaités.

# Appel synchrone de procédure




**Le client attend la réponse du serveur**

# Appel asynchrone de procédure




**Le client n'attend pas de réponse du serveur**

## Conséquences

- La fonction appelée ne doit renvoyer aucun résultat
- Aucune garantie applicative que l'appel soit reçu et exécuté

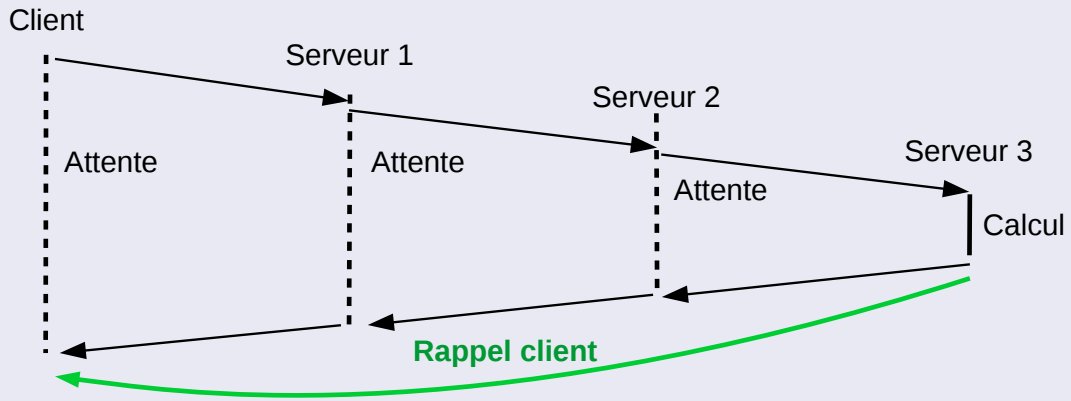
# Mécanisme de rappel

## Définition

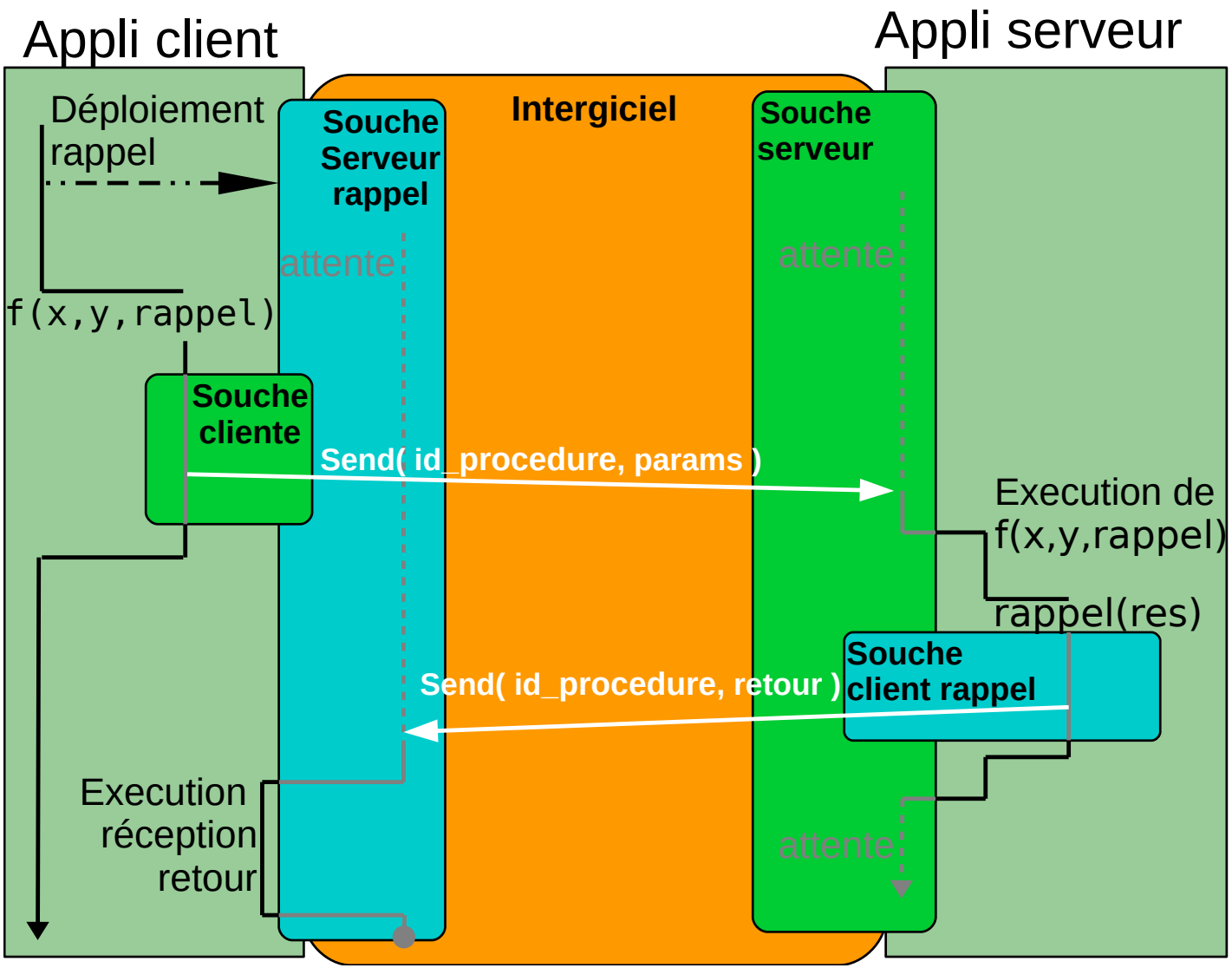
Service éphémère offert par le client pour qu'un serveur puisse le contacter ultérieurement

## Intérêts

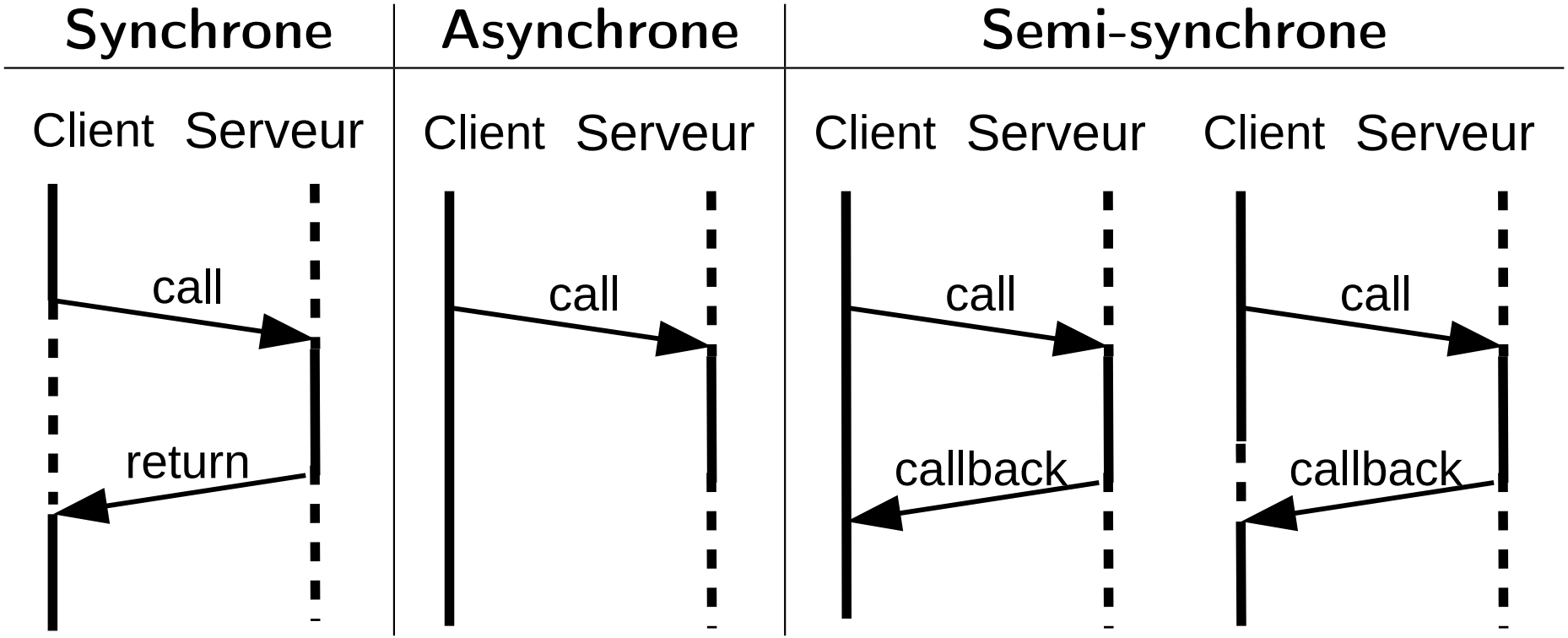
- **Appel semi-synchrone** : Le serveur peut retourner au client le résultat d'un appel même si c'est un appel asynchrone.
- Un serveur $S$ peut transférer à un autre serveur $S'$ la requête du client
  $\Rightarrow$ $S'$ peut contacter directement le client sans repasser par $S$



# Appel semi-synchrone de procédure



# Résumé des différents types d'appel



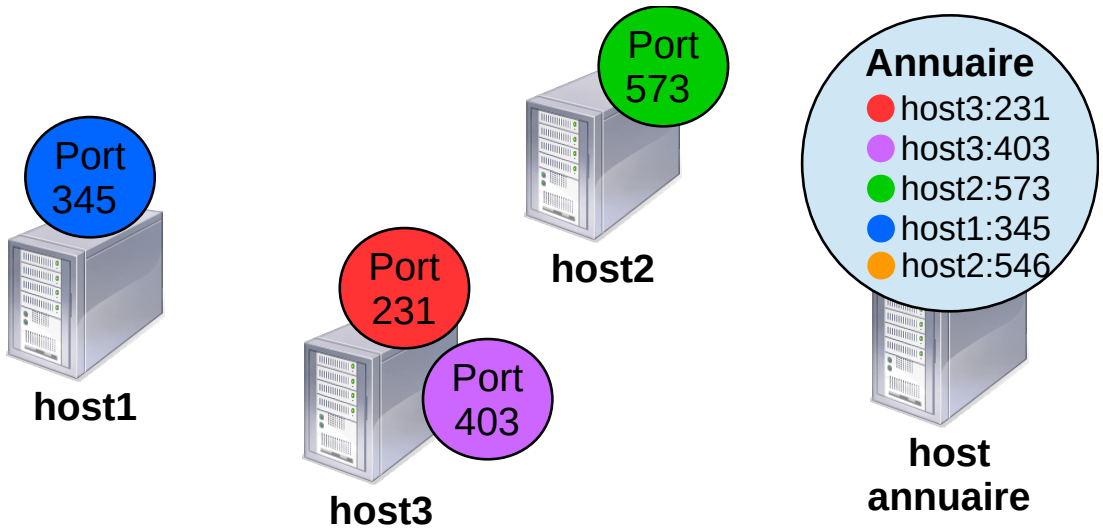
# Annuaire

## Définition

Service associant un nom de service à son adresse sur le système

## Intérêt

- Abstraction complète du réseau :
  $\Rightarrow$ il suffit juste de connaître une seule adresse, celle de l'annuaire
- Le nommage de service est plus intuitif
- Offrir un catalogue de l'ensemble des services du système



# Exemples d'intergiciels pour l'invocation distante

| Nom | Sérialisation | Paradigme | Particularités |
|---|---|---|---|
| **Sun RPC** | XDR | procédural | langage C uniquement |
| **gRPC** | Protobuf | procédural | Google |
| **Java RMI** | Java | objet | pas d'IDL |
| **CORBA** | CDR | objet | Standard reconnu |
| **EJB** | Java | composant | pas d'IDL |
| **XML-RPC** | XML | procedural | standard Web |
| **SOAP** | XML | procedural | standard Web |
| **Thrift** | interne | procedural | fondation Apache |