4-IF/ 4-IFA
Programmation Réseaux

Frédéric Prost
frederic.prost@insa-lyon.fr

INSA - Lyon

2023/2024

Intervenants

- Cours/TP: Frédéric Prost ; e-mail: frederic.prost@insa-lyon.fr
- TP/TD : Frédéric Prost - Sara Bouchenak - Romain Galle - Yacine Belal
- Pages du cours : Moodle INSA.
- Structure du cours :
  - Cours : 5 x 1h30.
  - TP : 4 x 4h.

Introduction

- Vision originale de l'informatique : un ordinateur résolvant un gros problème à la fois.
  - Cryptographie.
  - Bombes nucléaires, industrie spatiale.
  - Simulations numériques.
- Premières évolutions : échecs (CE Shannon).
- Développement de la capacité de stockage : Bases de données.
- Développement des communications : programmation réseau.
- Applications réseau : email, partage de documents (clouds), social media(FB, Twitter, ...), video on demand (Netflix)

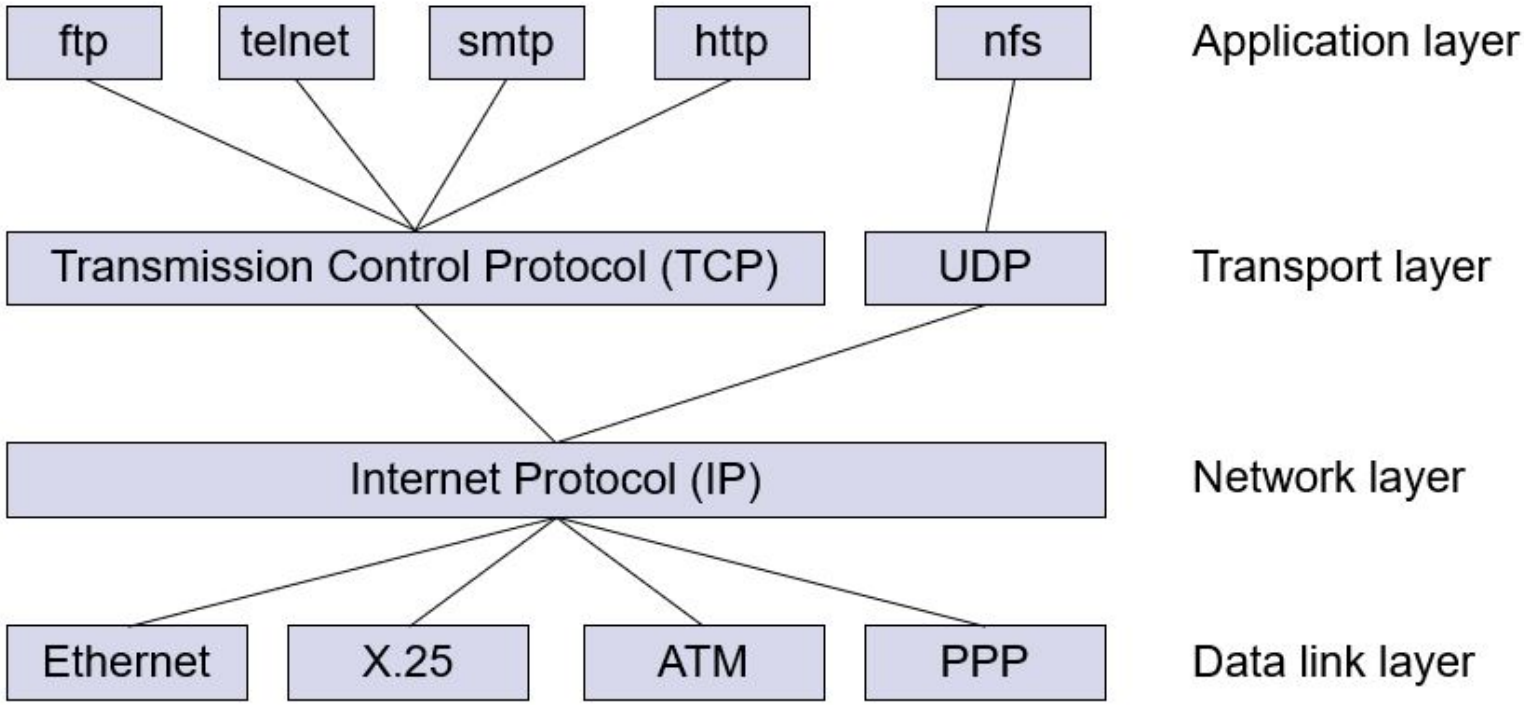
Introduction

- Le terme *réseau* est polysémique :
  - matériel : fibre optique, fils en cuivre, wifi, 4G/5G,
  - social : les anciens élèves de..., FB, Instagram, Youtube,
  - protocole : internet, bitcoin, ...
- Une manière de penser différente :
  - Synchronisation
  - Erreurs :

Open System Interconnection Layers

Layer	Function	Examples of protocols
7. Application	Services the actual applications used by the end user	SMTP, HTTP
6. Presentation	Formats data to be presented to the application layer	JPEG, GIF, ASCII
5. Session	Session establishment between processes running on different machines	RPC
4. Transport	Ensures that messages are delivered error-free, in sequence, and with no losses or duplication	TCP, UDP
3. Network	Controls the operantions of the subnet, deciding which physical path the data takes	IP
2. Data link	Error-free transfer of data frames from one node to another over the Physicla layer	PPP
1. Physical	Transmission and reception of unstructured raw bit stream over the physical medium	

Les couches de protocole



TCP, UDP et IP

- IP (Internet Protocol) :
  - Niveau Network du modèle OSI.
  - Adressage, routage et transport des paquets de données.
- TCP (Transmission Control Protocol) :
  - Un protocole sûr.
  - Guarantie la livraison des paquets.
  - Remet les paquest dans l'ordre.
- UDP (User Datagram Protocol) :
  - Un protocole pas sûr.
  - Très efficace.

Programmation Réseaux

- Le cours porte sur les 4 dernières couches : Application, Présentation, Session, Transport.
- Objectifs du cours, introductions à différents paradigmes :
  - Sockets.
  - Remote Procedure Calls (RPC / Java RMI).
  - World Wide Web.
  - Introduction à l'informatique distribuée
  - Les TP porteront sur le niveau Application et l'utilisation de sockets et RPC en C.

Plan

Identifier et retrouver les ressources

- URI : Uniform Ressource Identifier.
  - Identifie une ressource.
  - Ne donne pas nécessairement le lieu où est la ressource.
- URL : Uniform Ressource Locator.
  - Représente le lieux où se situe la ressource.
  - Une URL dit comment on accède à la ressource, une URI pas forcément.
  - Le protocole utilisé pour localisé la ressource est connu à partir de l'URL.
- IPFS : adressage par le contenu. L'adresse est le hash du fichier.

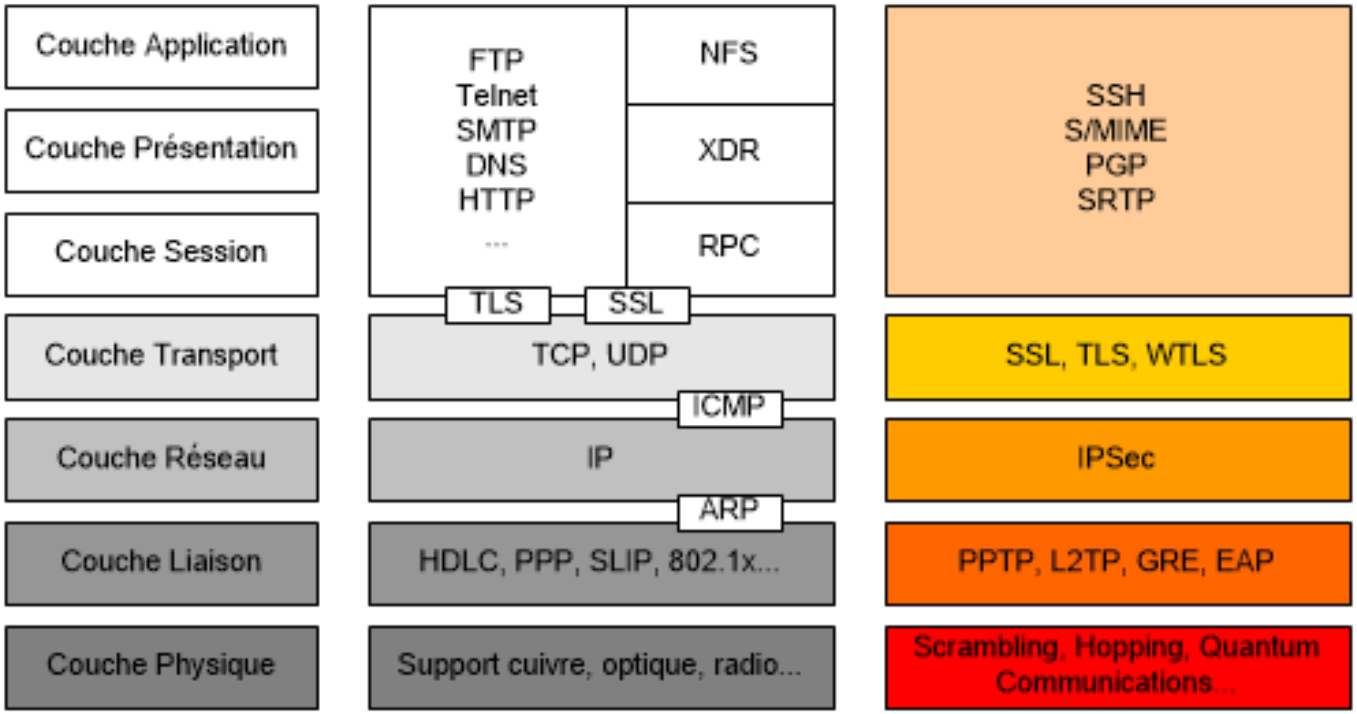
URL/URI exemples

- URI :
  - news:comp.infosystems.www.servers.unix : protocole news pour les newsgroups Usenet.
  - urn:isbn:0-395-36341-1 : ISBN (International Standard Book Number). Où est le livre est une autre histoire...
- URL :
  - mailto:frederic.prost@insa-lyon.fr
  - https::/google.com/
  - ssh:machin@machine.com
- Packages suivant les langages, par ex. Java : classes URLConnection et HttpURLConnection
  en C ... faire à la main.

Addresses IP

- Pour identifier chaque équipement connecté à un réseau utilisant le protocole IP.
- Les URL peuvent correspondre à des adresses IP : DNS.
- Adresse IPv4: Notation décimale avec 4 valeurs comprises entre 0 et 255, séparées par des points :
  192.168.1.10
  taille 32 bits = 4 octets
- Adresse IPv6: Notation hexadécimale avec 8 valeurs séparées par ":" :
  1987:0c02:0000:84c2:0000:0000:cf2a:9077
  128 bits = 16 octets
- 5 plages IP suivant l'utilisation : IP privées, publiques, multicast
  127.0.0.0 : localhost

Couche ISO modèle OSI - Résumé



Plan

Sockets

- Une socket est une abstraction/structure logicielle qui sert à un noeud d'un réseau à envoyer et recevoir des données.
- Provient des années 70 dans les premières version d'Unix avec la philosophie "tout est fichier".
- D'un point de vue du programmeur c'est le méchanisme qui permet de transférer des données d'un ordinateur à un autre.
- Aujourd'hui c'est compris par défaut comme *internet socket* du fait de la standardisation de TCP/IP.
- Trois caractéristiques :
  - Protocole de transport.
  - Adresse.
  - Numéro de port.

Sockets

- A l'origine (80) très système dépendant. Invoqués avec des librairies C/C++ spécifiques.

  Portabilité limitée.

  Programmes qui devaient prendre en compte les différents OS etc.

- Aujourd'hui :
  - Les sockets sont disponibles sur toutes les plateformes et sont, d'un point de vue logiciel, le mécanisme fondamental de communication.
  - Différentes interfaces, eg Java, pour les sockets permettent de les abstraire du système sous-jascent.

  $\Longrightarrow$ Dans ce cours utilisation de C (illustratif).

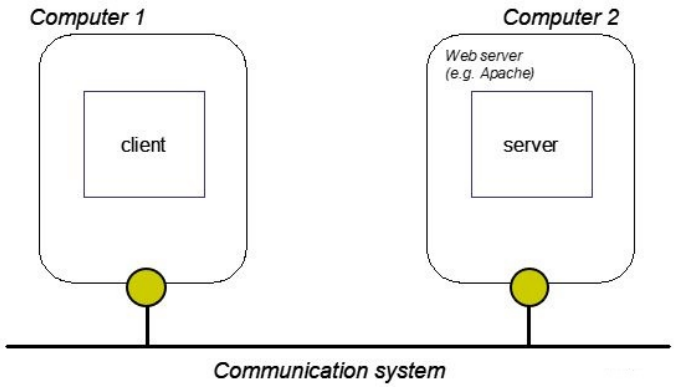
Ports - 1

- Un ordinateur connecté à un réseau TCP/IP est identifié par son adresse IP.
- Quand un processus veut accéder au réseau, il est identifé par :
  - L'adresse IP de l'ordinateur sur lequel il tourne
  - Un numéro de port associé avec le processus.
- C'est le système qui attribue les numéros de port.
- Beaucoup de services ont des numéros de port par défaut :
  - 23 pour telnet.
  - 25 pour SSH, SCP, SFTP (protocoles cryptés).
  - 15 pour SMTP (email).
  - ...

Ports - 2

D'un point de vue d'une application client/serveur

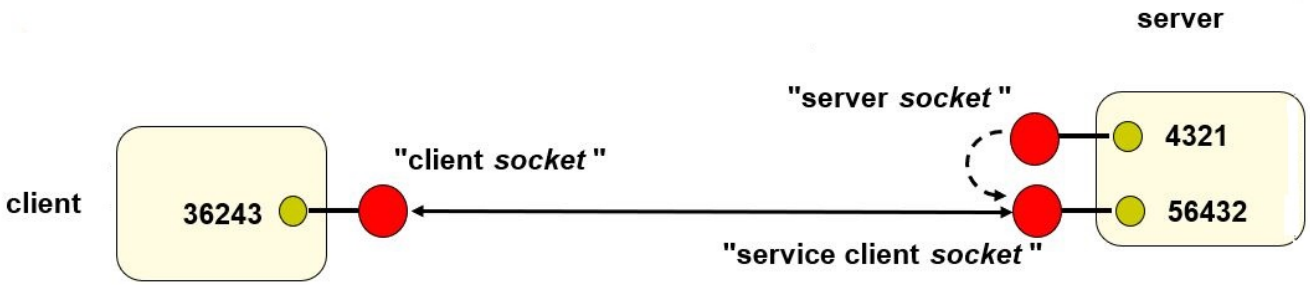
- Du côté serveur :
  - Quand le processus démarre l'OS lui assigne un port fixe.
  - Le serveur attend en processus de fond (un démon) pour les connections entrant à cette adresse.
- Du côté client :
  - Un processus client (sur un autre ordinateur) est assigné à un port sur son ordinateur.
  - Le serveur peut identifier le client de manière unique par le port et l'adresse sur le réseau.
- Les ports sont des points de communication. Graphiquement :



Sockets - Client/Serveur

Exemple TCP/IP en trois étapes.

1. Le serveur crée une "socket serveur", attend les connexions entrantes.
2. Le client se connecte à la "socket serveur". Deux sockets sont créées et connectées :
   - Une "socket client" du côté client.
   - Une "socket service client" du côté serveur.



3. Le serveur et le client communiquent par ces deux sockets.
   Le serveur peut recevoir d'autres connexions.

Plan

Sockets en langage C

- Suit la philosophie générale de C/Unix : une socket est un fichier dans lequel on peut lire/écrire.
- Très bas niveau : seuls des char sont transmis sur le réseau
  $\Longrightarrow$ Il faut coder/décoder (par exemple les int).
  prévoir un buffer de la bonne taille pour la réception des données...
- Un cas particulier l'architecture client/serveur.

Sockets en C - cheat sheet

Résumé des fonctions (et de la séquence) à utiliser:

TCP côté client	TCP côté serveur
socket	socket
connect	bind
send	listen
recv	accept
close	send
	recv
	close

$\Longrightarrow$ Il existe l'équivalent pour des communications UDP.

Exemples

- En local par TCP (1 client).
  - Exemple : client.c server.c.

Exemples

- En local par TCP (1 client).
  - Exemple : client.c server.c.
- Par internet (multiples clients).
  - Exemple : client2.c client2.h server2.c server2.h.

Multicasts

- Le multicast est disponible au niveau du protocole internet.
- Le multicast est basé sur le protocol de transport UDP.
- Un multicast utilise une adresse IP particulière (classe D). Ces adresses ne sont pas liés à des ordinateurs.
  1. Tous les participants s'enregistrent dans un groupe pour joindre la communication.
  2. Les messages peuvent être diffusés à (ou reçus de) tous les membres du groupe.
  3. Les participants doivent signaler quand ils cessent d'appartenir au groupe.
- Application: visio-conférences...

Autre implantation des sockets : Java.net

- La plateforme java donne une implantation standard pour utiliser des sockets, le package :

  `java.net`

- On pourrait tout implanter "à la main" mais utiliser des instance de la classe
  `java.net.Socket`
  est plus ... sûr.
- Classe `ServerSocket` permet de mettre en oeuvre une architecture Client/Serveur.
- Si l'application passe par internet, ce qui sera quasiment toujours le cas, une classe `URL` et les classes associées `URLconnection`, `URLEncoder` fournissent tout ce dont on a besoin.

Conclusion

- Les réseaux : une branche "à part" dans l'informatique.
  - Aspects très technologiques.
  - Aspects industriels : qu'est ce qui n'est pas connecté aujourd'hui ?
  - Aspects conceptuels uniques : applications distribuées, réseaux sociaux, cryptomonnaies...
- Les Sockets :
  - Mécanisme de base pour la communication d'un point de vue logiciel. $\Longrightarrow$ pour un programmeur c'est l'objet de base pour gérer les communications.
  - D'un point de vue logiciel assez bas niveau (RMI/RPC au cours suivant).

Plan

Introduction aux RPC/RMI

- Les sockets sont un moyen de communiquer les informations entre les applications dans les systèmes distribués.
  Simples, flexibles mais limitées à la transmission de données.
- Il n'y a pas d'aspects sémantiques au niveau des sockets.
- Ce sont les protocoles qui doivent fournir l'interprétation sémantique des données. Ils doivent êtres développés au niveau application.
- Le développement de tels protocols est complexe, prend du temps et est difficile à bien implanter.

Introduction aux RPC/RMI

- La programmation OO a déjà un cadre pour la sémantique des données : les objets.
- Localement les objets communiquent par des méthodes (appels de fonction).
- $\Longrightarrow$ On cherche la même chose pour des objets/fonctions distribués.
- $\Longrightarrow$ c'est toute l'idée du Remote Procedure Call - RPC.

C'est une version plus haut niveau que les sockets.

RPC - petit rappel historique

- Remote Procedure Call - développé dans les années 80 pour appeler des procédures à distance (avant la programmation objet).
- Le défi est l'appel de procédure dans un autre espace mémoire sur un ordinateur lointain : comment faire l'édition de lien, gérer les pointeurs, la pile etc. ?
- Techniquement:
  1. Différents espaces d'adressage.
  2. Machines et systèmes hétérogènes.

RPC - implantation

Différents espaces d'adressage.

- Cas local :
  - Les données peuvent êtres passées par référence.
  - La référence est une adresse physique dans une mémoire concrète.
  - La référence n'a aucun sens dans un autre espace mémoire.
- Cas distribué :
  - Les données doivent être passées en copie et codée (linéarisée) comme une suite de bits qui pourra être décodée par le correspondant !

RPC - implantation

Machines et systèmes hétérogènes.

- La représentation interne des données n'est pas forcément la même (integer sur 64 bits ou 32 bitsi - en little ou big endian).
- Il faut passer par une représentation des données qui soit indépendantes des plateformes (par ex XDR - eXtensible Data Representation).
- Les données reçues par RPC doivent êtres de nouveau reconverties pour l'utilisation locale.

Aperçu de solutions RPC 1/3

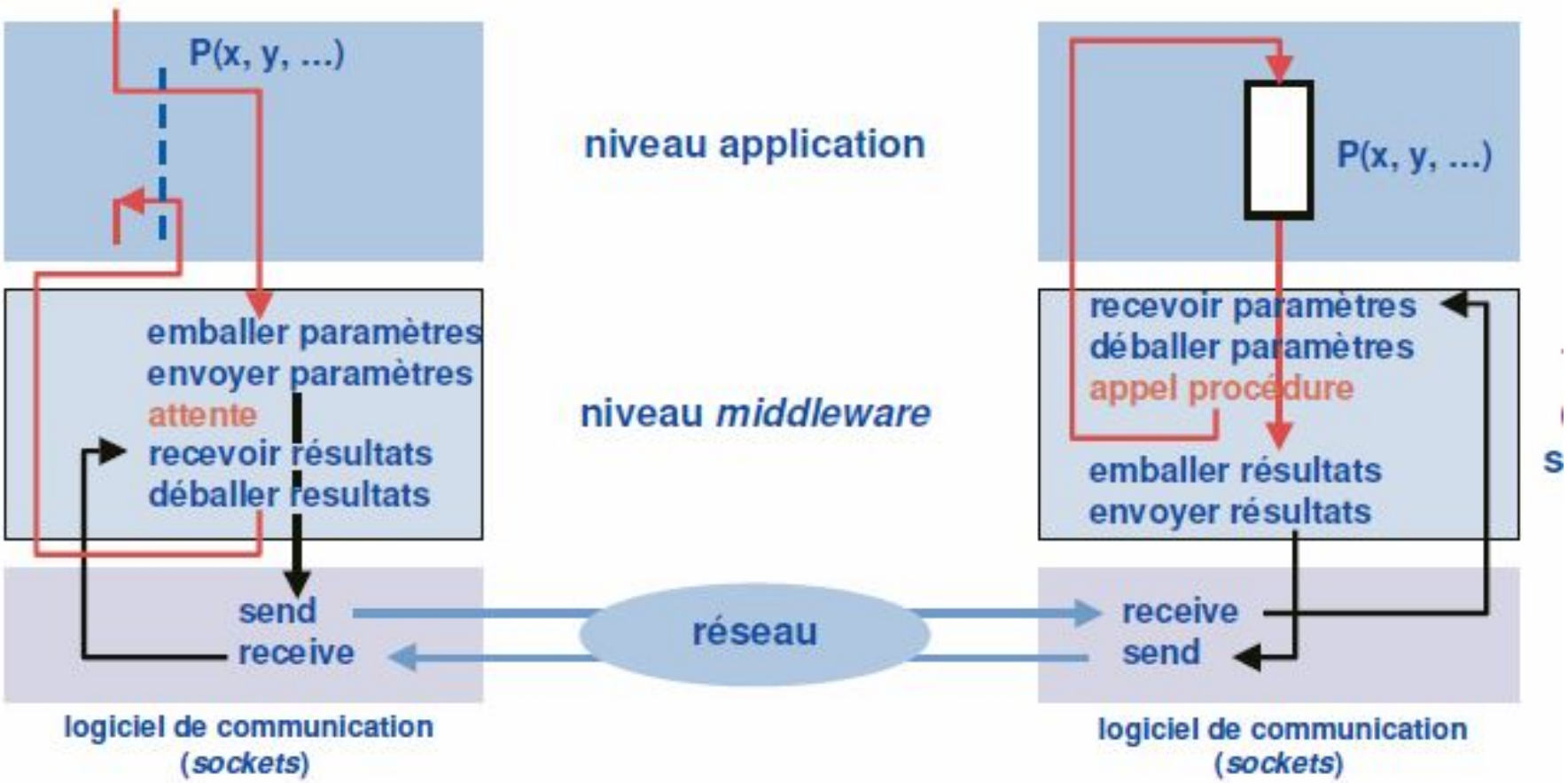
- Deux programmes : client et serveur.
- Programme serveur :
  - Définit les fonctions qui seront utilisées à distance.
  - attend des invocations de la part des clients.
    $\Longrightarrow$ c'est l'idée du "cloud computing".
- Programme client :
  - Ne voit qu'une interface de ce qui est implanté.
  - Invoque les fonctions sur le serveur.
- RPC est le cadre permettant de faire fonctionner la communication (d'un point de vue programmeur) entre client et serveur.
  Cela permet d'abstraire le comment les données seront partagées.

Aperçu de solutions RPC 2/3

- Avantages attendus
  - Facilité de programmation
    $\Longrightarrow$ complexité des protocoles de communication est cachée
  - Facilité de mise au point : une application peut être mise au point sur un site unique, puis déployée sur plusieurs sites
  - Portabilité : résulte de l'usage d'un langage de haut niveau
    $\Longrightarrow$ Indépendance par rapport au système de communication
- Problèmes de réalisation
  - Transmission des paramètres (conversion entre la forme interne et une qui circule sur le réseau : codé via xdr)
  - Gestion des processus
  - Réaction aux défaillances

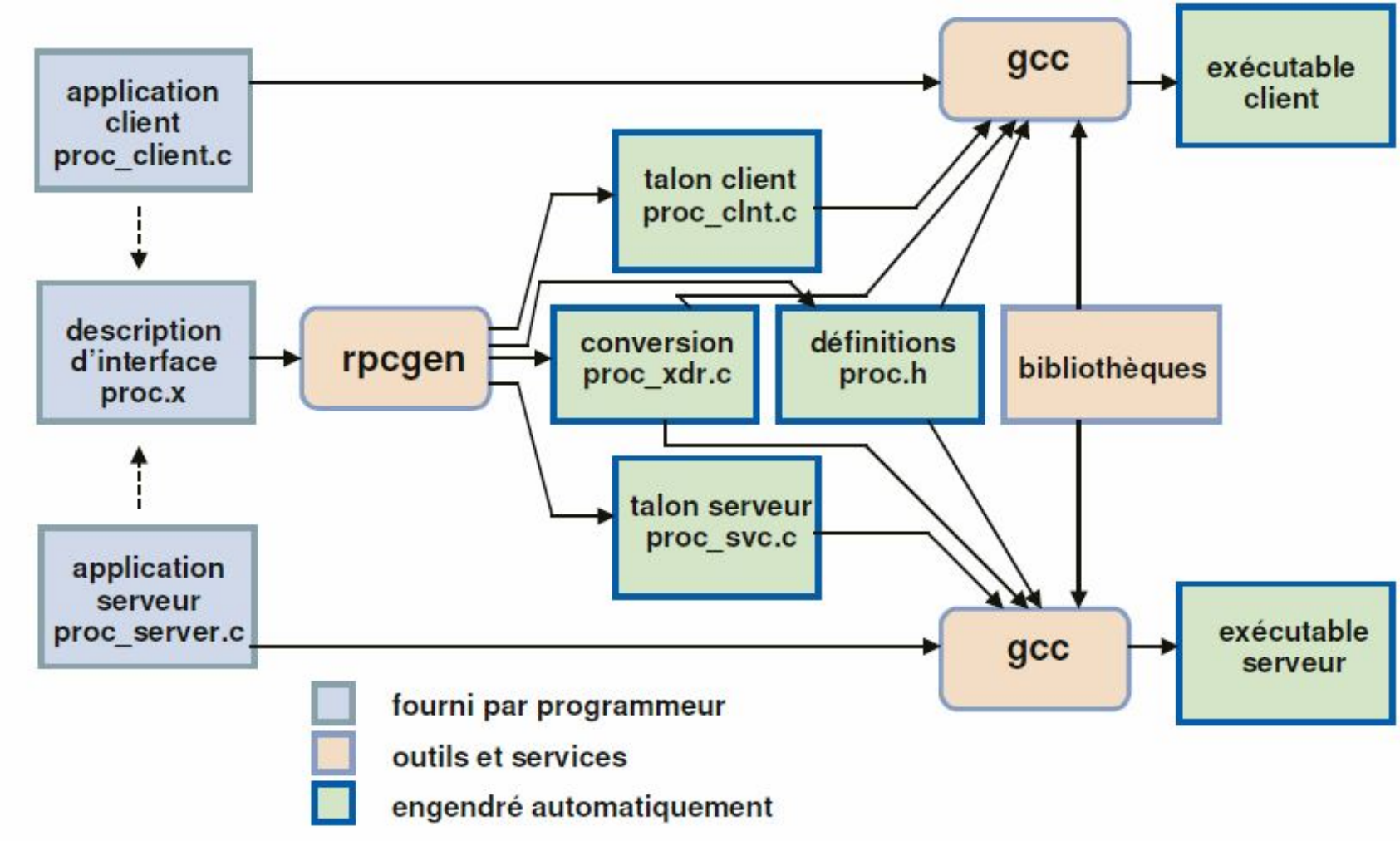
$\Longrightarrow$ Trois modes de défaillance indépendants : client, serveur, réseau

Principe d'une solution RPC 3/3



Les talons clients et serveurs sont créés à partir d'une description d'interface.

Chaîne de production



RPCGEN

- `RPCGEN`
  - RPC Generator
  - Utilitaire permettant de générer automatiquement le code des talons et des fonctions XDR associées aux données utilisées par les fonctions
- Principe d'utilisation
  - On décrit dans un fichier (d'extension `.x`)
  - Les structures de données propres aux fonctions
  - Les fonctions appelables à distance
  - RPCGEN génère ensuite un ensemble de fichiers

Par l'exemple

- Programme de traitement de formes géométriques.
  $\Longrightarrow$ les fonctions sont appelables par les clients à distance.

```
int surface_rectangle(struct rectangle rect);
struct rectangle creer_rectangle(int x1, y1, x2, y2);
booleen inclus(struct rectangle r, struct point p);
```

- Avec les types suivants :

```
struct point { int x, y; };

struct rectangle { struct point p1, p2; };

typedef int booleen;
```

fonctions d'origine

```
int surface_rectangle(struct rectangle rect) {
 return abs((rect.p1.x - rect.p2.x) *
            (rect.p1.y - rect.p2.y));}

struct rectangle creer_rectangle(int x1, int x2, int y1,
                                 int y2) {
struct rectangle rect;
rect.p1.x = x1; rect.p1.y = y1; rect.p2.x = x2; rect.p2.y = y2
return rect;}

booleen inclus(struct rectangle rect, struct point p){
return ( (p.x >= rect.p1.x) && (p.x <= rect.p2.x)
         && (p.y >= rect.p1.y) && (p.y <= rect.p2.y));}
```

Ecriture du fichier .x

- Fichier décrivant les fonctions et données
  - Pour les données
    - Décrit les structures presque comme en C (voir suite pour détails)
  - Pour les fonctions une règle fondamentale :
    - Une fonction ne prend qu'*UN* seul paramètre
      $\Longrightarrow$ On doit donc définir une structure de données dédiée à la fonction si on veut passer plusieurs valeurs en paramètre
- Définition d'un programme RPC
  - Programme = ensemble de fonctions/services
  - Chaque programme possède un nom avec un numéro associé
  - Un programme peut exister en plusieurs versions
    - Chaque version est identifiée par un nom et un numéro associé
    - Chaque version définit une liste de fonctions
    - Chaque fonction est identifiée par un numéro unique

fichier "geometrie.x"

```
struct point { int x; int y; };
struct rectangle { struct point p1; struct point p2; };
struct coordonnees { int x1; int x2; int y1; int y2; };
struct param_inclus { struct rectangle; struct point p; };
typedef int booleen

program GEOM_PROG {
    version GEOM_VERSION_1 {
      int SURFACE_RECTANGLE(rectangle) = 1;
      rectangle CREER_RECTANGLE(coordonnees) = 2;
      booleen INCLUS(param_inclus) = 3;
    } = 1;
} = 0x20000001;
```

geometrie.x : explications

- Nom du programme : `GEOM_ROG` d'identifiant 20000001 (en hexa)
- Nom de version : `GEOM_VERSION_1` de numéro 1
- Les 3 fonctions sont numérotées 1, 2 et 3
- Les structures coordonnees et `param_inclus` ont été créées pour les fonctions nécessitant plus de 2 paramètres
- Par convention, on écrit les noms de fonctions, programmes et versions en majuscule

Utilitaire rpcgen

- Pour générer les fichiers avec rpcgen
  `$ rpcgen geometrie.x`
- Fichiers générés à partir du `.x`
  - `geometrie.h`
    $\Longrightarrow$ Définition de toutes les structures et des signatures des opérations
  - `geometrie_xdr.c`
    $\Longrightarrow$ Fonctions de codage XDR des structures de données
  - `geometrie_clnt.c` et `geometrie_svc.c`
    $\Longrightarrow$ Talons cotés client et serveur
- Avec l'option `-a` de `rpcgen`
  - Génère en plus les fichiers `geometrie_server.c` et `geometrie_client.c`
    $\Longrightarrow$ Squelettes pour l'écriture de clients et du serveur

Contenu du `fichier.h` - 1

- Définition de la structure équivalente en C
- Définition d'un type du même nom correspondant à la structure
- Signature de la méthode XDR correspondant à ce type
- Exemple, dans le `.x`

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

- Devient dans le .h

```
struct point { int x; int y;};
typedef struct point point;
bool_t xdr_point(XDR*, point*);
```

Contenu du `fichier.h` - 2

- Pour chaque méthode
  - Dans le `.x`
  - Pour chaque opération nommée `fonction`, de version numérotée `y`, qui prend un `type_param` en paramètre et retourne un `type_retour`
  - On aura dans le `.h`, signature de 2 méthodes correspondantes

```
type_retour *fonction_y(type_param *, CLIENT *)
type_retour *fonction_y_svc(type_param *, struct svc_req *)
```

- Dans les 2 cas
  - Le nom de la fonction est suffixée par la version (et `svc` pour la seconde)
  - Un niveau de pointeur en paramètre et retour est ajouté par rapport au `.x`

Contenu du `fichier.h` - 3

- Première version : `fonction_y`
  - Fonction qui est implantée par le talon client
  - C'est cette fonction que la partie client appelle localement pour demander l'appel de la fonction associée coté serveur
  - Paramètre `CLIENT*`
    $\Longrightarrow$ Caractéristiques de la communication avec la partie serveur
- Deuxième version `fonction_y_svc`
  - Fonction à implanter coté serveur
    $\Longrightarrow$ Fonction qui est appelée par les clients
  - Paramètre `svc_req*` : caractéristiques de la requête d'appel de service et identification du client
- Exemple
  - `geometrie.x` : `int SURFACE_RECTANGLE(rectangle) = 1;`
  - `geometrie.h` :

```
int surface_rectangle_1(rectangle *, CLIENT *);
int surface_rectangle_1_svc(rectangle * , svc_req *);
```

Contenu du `fichier.h` - 4

- Définitions de constantes
  - Numéros de programme et de versions en associant chaque numéro au nom précisé dans .x
  - Exemple

```
#define GEOM_PROG 0x20000001
#define GEOM_VERSION_1 1
```

- Pour chaque fonction, définition d'une constante associant son nom à son numéro
  - Exemple
    $\Longrightarrow$ Dans `.x` : `rectangle CREER_RECTANGLE(coordonnees) = 2;`
    $\Longrightarrow$ Dans `.h` : `#define CREER_RECTANGLE 2`
- Toutes ces constantes serviront à identifier le programme et les fonctions lors des appels à distance

Contenu de geometrie_xdr.c

- Contient les méthodes de codage XDR pour chaque type de données décrit dans le .x
- Exemple

  $\Longrightarrow$ Dans geometrie.x:

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

  $\Longrightarrow$ Dans geometrie_xdr.c:

```
bool_t xdr_point(XDR* xdrs, point* objp) {
 if (!xdr_int(xdrs, &objp->x)) {
 return (FALSE);
 }
 if (!xdr_int(xdrs, &objp->y)) {
 return (FALSE);
 }
 return (TRUE);
}
```

Implantation des fonctions

- Côté serveur, dans un fichier à part :
  - Ou dans le squelette coté serveur si on l'a généré
  - Pour chacune des fonctions du programme, on implante le code en prenant la signature du `.h`
- Exemple :

  $\Longrightarrow$ Dans le `.h`

```
extern int *surface_rectangle_1_svc(rectangle*, CLIENT*);
extern rectangle *creer_rectangle_1_svc(coordonnees*,
 CLIENT*);
extern bool_t *inclus_1_svc(param_inclus*, CLIENT*);
```

  $\Longrightarrow$ Implantation de ces fonctions dans fichier `geometrie_server.c`

Fichier geometrie_server.c

```
#include "geometrie.h"
int *surface_rectangle_1_svc(rectangle *rect, svc_req *req) {
static int result;
result = (rect -> p1.x { rect -> p2.x) *
(rect -> p1.y { rect -> p2.y);
return &result;}
rectangle creer_rectangle_1_svc(coordonnees *coord,
          svc_req *req) {  static rectangle rect;
rect.p1.x = coord -> x1; rect.p1.y = coord -> y1;
rect.p2.x = coord -> x2; rect.p2.y = coord -> y2;
return &rect;}
booleen inclus_1_svc(param_inclus *param, svc_req *req) {
static booleen result;
result = (param -> p.x >= param -> rect.p1.x) &&
(param -> p.x <= param -> rect.p2.x) &&
(param -> p.y >= param -> rect.p1.y) &&
```

Remarques implantation du serveur

- Pas d'utilité à manipuler le paramètre `svc_req`
- Les variables `result` sont déclarées en `static`
  - Il faut retourner un pointeur sur une donnée qui existera toujours après l'appel de la fonction.
  - Cette donnée sera retournée par copie au client.

Côté client

- Appel d'un service `fonction` sur la partie serveur
  $\Longrightarrow$ Appelle simplement la fonction `fonction_x` coté client
  $\Longrightarrow$ Cette fonction est appelée sur le talon client qui relaie la requête coté serveur sur lequel on appellera `fonction_x_svc`
- Avant de pouvoir appeler une fonction sur une machine distance
  $\Longrightarrow$ Il faut identifier le programme RPC tournant sur cette machine

```
CLIENT *clnt_create(
char *machine,                  nom de la machine serveur
long numero_programme,          id. du programme RPC
long numero_version,            id. de version du programme
char *protocole);               "udp"  ou  "tcp"
```

- Identificateurs programme et version : utiliser les noms associés du `.h`
- Retourne un identificateur de communication coté client à utiliser pour appel des fonctions ou `NULL` si problème
- Exemple : fichier client.c

Fichier client.c

```
#include "geometrie.h"
int main() { CLIENT *client; rectangle *rect; point p;
 coordonnees coord; param_inclus p_inc; int *surface; i
 booleen *res_inclus;
 client = clnt_create("xxx.yy.zz.tt.uu", GEOM_PROG,GEOM_VERSIO
 if (client == NULL) { perror(" erreur creation client\n");
     exit(1); }
 coord.x1=12; coord.x2=20; coord.y1=10;coord.y2=15; p.x=14; p.
 rect = creer_rectangle_1(&coord, client);
 p_inc.rect = rect; p_inc.p = p;
 surface = calculer_surface_1(rect, client);
 res_inclus = inclus_1(&p_inc, client);
 printf(" rectangle de surface %d\n", *surface);
 if (*res_inclus) { printf(" p inclus dans rect\n");
}
```

Commentaires sur `client.c`

- Pour la connexion avec le programme RPC distant :
  - Le programme s'exécute sur la machine xxx.yy.zz.tt.uu
  - On utilise les constantes définies dans `geometrie.h` pour identifier le programme et sa version :
    `GEOM_PROG` et `GEOM_VERSION_1`
  - On choisit une communication en UDP
- Fonctions `creer_rectangle_1`, `calculer_surface_1` et `calculer_surface_1`

  $\Longrightarrow$ Vont engendrer l'appel des fonctions associés sur le serveur (sur la machine xxx.yy.zz.tt.uu)

  $\Longrightarrow$ Aucune différence avec appel d'une fonction locale

  Transparence totale de la localisation distante du code de la fonction
- Gestion des erreurs possibles lors des appels `RPC`
  - Les fonctions retournent `NULL`
  - `clnt_perror(CLIENT*,char *msg)` : affiche l'erreur i (avec msg avant)

Compilation client et serveur

- Pour les 2 parties, besoin des fonctions de codage XDR

```
$ gcc -c geometrie_xdr.c
```

- Coté client :

```
$ gcc -c geometrie_clnt.c
$ gcc -c client.c
$ gcc -o client client.o geometrie_clnt.o
          geometrie_xdr.o
```

- Coté serveur :

```
$ gcc -c geometrie_svc.c
$ gcc -c geometrie_server.c
$ gcc -o serveur geometrie_svc.o geometrie_server.o
          geometrie_xdr.o
```

Coté serveur - Lancement

- Le talon coté serveur (`geometrie_svc.c`) contient un main qui enregistre automatiquement le programme comme service RPC
- Avec l'outil système `rpcinfo`, on peut connaître la liste des services RPC accessibles sur une machine

```
$ /usr/sbin/rpcinfo -n scinfe222
program no_version protocole no_port
100000 2 tcp 111 portmapper
100000 2 udp 111 portmapper
100024 1 udp 32768 status
100024 1 tcp 32770 status
391002 2 tcp 32771 sgi_fam
536870913 1 udp 32916
536870913 1 tcp 38950
```

- Notre programme est bien lancé en version 1
- Numéro programme : $(536870913)_{10} = (20000001)_{16}$
- Il est accesible via UDP (port 32916) ou TCP (port 38950)

Langage RPC compléments - Base

- RPC Language :
  - Langage des fichiers `.x`
  - Langage de type IDL : Interface Definition Language
    Interface : ensemble des opérations qu'offre un élément logiciel
- Compléments sur la définition des éléments dans les structures de données
  $\Longrightarrow$ On peut utiliser des pointeurs, des énumérations et définir des types avec typedef
  Utilisation et syntaxe comme en C
- Pour définir des constantes $\Longrightarrow$ `const NAME = val;`
- Exemple
  Si dans le `.x` on a `const DOUZAINE = 12;`
  c'est traduit dans le `.h` en : `#define DOUZAINE 12`

Langage RPC compléments - Tableaux

- Tableaux de taille fixe et connue : comme en C, avec [ ]
  Exemple : `int data[12];`
- Tableaux de taille variable : Utilise les < > au lieu des [ ]
  on peut préciser la taille maximale du tableau entre les < >
- Exemple
  - `int data` < > taille quelconque
  - `double valeurs`< 10 > tableau d'au plus de 10 double
- Traduction en C
  - Définition d'un tableau de taille variable : structure de 2 éléments
    $\Longrightarrow$ Le pointeur correspond au tableau
    $\Longrightarrow$ La taille du tableau

```
struct {
u_int data_len;
int *data_val;
} data;
```

```
struct {
  u_int valeurs_len;
  double *valeurs_val;
} valeurs;
```

Langage RPC Compléments

- Chaînes de caractères
  - Utilise le type string
  - Taille de la chaîne : quelconque ou taille maximale
  - Utilise aussi la notation en < >
  - Exemples
    - $\Longrightarrow$ `string nom< 20 >;` chaine de taille d'au plus 20 caractères
    - $\Longrightarrow$ `string message< >;` chaine de taille quelconque
  - Traduits dans le `.h` en `char*` tous les deux
    - $\Longrightarrow$ `char *nom;`
    - $\Longrightarrow$ `char *message;`
- Données opaques
  - Utilise le type opaque
  - Exemple
    - $\Longrightarrow$ `opaque id_client[128];`

Langage RPC Compléments

- Unions

  Syntaxe d'utilisation

```
union nom_union switch(type discrim) {
case value : ... ;
case value : ... ;
...
default : ...;
};
```

  Exemple: Selon le code d'erreur d'un calcul, on veut stocker le résultat normal (un flottant) ou le code de l'erreur (un entier)

```
union res_calcul switch(int errno) {
case 0:
 float res;
default:
 int error;
};
```

Langage RPC Compléments

- Exemple union (suite). Traduction dans le `.h` en :

```
struct res_calcul {
  int errno;
  union {
   float res;
   int error;
   } res_calcul_u;
  };
typedef struct res_calcul res_calcul;
```

- Notes sur les numéros de programmes RPC : 4 plages de valeurs, en hexa
  - 0000 0000 à 1FFF FFFF : gérés par Sun
  - 2000 0000 à 3FFF FFFF : programmes utilisateurs
  - 4000 0000 à 5FFF FFFF : transient
  - 6000 0000 à FFFF FFFF : réservé, non utilisé

Traitement des défaillances (résumé)

- Difficultés du traitement des défaillances
  - Client sans réponse au bout d'un délai fixé. 3 possibilités :
    1. Le message d'appel s'est perdu sur le réseau
    2. L'appel a été exécuté, mais le message de réponse s'est perdu
    3. Le serveur a eu une défaillance

       c1: Avant d'avoir exécuté la procédure

       c2 : Après la procédure mais avant d'avoir envoyé la réponse
  - Si le client envoie de nouveau la requête
    - Pas de problème dans le cas 1.
    - L'appel sera exécuté 2 fois dans le cas 2

      Remède : associer un numéro unique à chaque requête
    - Divers résultats possibles dans le cas 3 selon remise en route du serveur
- Conséquences pratiques sur la conception des applications
  - Construire des serveurs "sans état" (donc toute requête doit être self-contenue, sans référence à un "état" mémorisé par le serveur)
  - Prévoir des appels idempotents (2 appels successifs ont le même effet qu'un appel unique). Exemple : déplacer un objet
    - move(deplacement_relatif) : non idempotent
    - move_to(position absolue) : idempotent

Conclusion

- Avantages:
  - Abstraction (les détails de la communication sont cachés)
  - Intégration dans un langage : facilite portabilité, mise au point
  - Outils de génération, facilitent la mise en oeuvre
- Limitations:
  - La structure de l'application est statique : pas de création dynamique de serveur, pas de possibilité de redéploiement entre sites
  - Pas de passage des paramètres par référence
  - La communication est réduite à un schéma synchrone
  - La persistance des données n'est pas assurée (il faut la réaliser explicitement par sauvegarde des données dans des fichiers)
  - Des mécanismes plus évolués visent à remédier à ces limitations
    - Objets répartis (ex : Java RMI)
    - Composants répartis (ex : EJB, Corba CCM, .Net)