

Module 2

Communications inter-processus (IPC): Appel de procédures à distance (RPC)

• **RPC versus appel de fonction**

- Message de requête et de réponse; écritures et lectures réseau explicites ou appel de fonction distante (RPC).
- Paramètre d'entrée, de sortie ou les deux.
- Pas de passage de paramètre par pointeur.
- Pas de variable globale accessible à tous.
- Doit spécifier comment sérialiser les structures spéciales qui utilisent des pointeurs (chaîne de caractère, vecteur, liste, arbre...) par un langage de définition d'interface ou des attributs sur les déclarations.
- Si l'appelant et l'appelé sont programmés dans différents langages, il faut adapter les différences (e.g. ramasse-miette, exceptions...).
- Arguments supplémentaires: serveur à accéder, code d'erreur pour la requête réseau.

• Catégories et historique

- Protocoles simples : SMTP (1982), HTTP (1991), Remote GDB...
- RPC basés sur un langage d'interface (multi-langage): Sun RPC (1984), ANSA (1985), MIG (1985), CORBA (1991), gRPC (2015).
- RPC basés sur un langage spécifique (meilleure intégration): Cedar (1981), Argus, Modula-3 (1993), Java (1995).
- RPC basé sur une machine virtuelle multi-langage (Common Language Runtime) C# (2000).

- **Langage de définition des interfaces (IDL)**

- Déclaration des procédures (nom, et liste et type des arguments).
- Déclaration des types et des exceptions.
- Générateur de code à partir du IDL pour la librairie client (proxy), et la librairie serveur (squelette), dans le langage de programmation désiré.
- Le IDL et le générateur de code ne sont pas requis si la réflexivité permet d'obtenir cette information à l'exécution.

- **Sémantique des appels**

- Mécanismes: envoi simple, accusé de réception, retransmission, filtrage des requêtes dupliquées, mémorisation des réponses.
- Peut-être: en l'absence de réponse on ne peut savoir si la requête ou la réponse fut perdue.
- Au moins une fois: retransmission sans filtrage des requêtes jusqu'à obtention d'une première réponse.
- Au plus une fois: avec retransmission, filtrage des requêtes, et mémorisation du résultat, la procédure est exécutée exactement une fois si une réponse est obtenue et au plus une fois en l'absence de réponse.

- **Sun RPC**

- Définition d'interface et de types en XDR.
- Procédures avec un argument (entrée), une valeur de retour (sortie), et possiblement un numéro de version.
- **rpcgen** produit la librairie client (proxy/client stub) et le squelette (server stub) du programme serveur.
- Compléter le squelette du serveur avec l'implantation.
- Initialiser la connection dans le client et utiliser les fonctions proxy en vérifiant les erreurs.

- **Sun RPC**

- Le DNS effectue la localisation du serveur.
- Portmap effectue la localisation du service.
- Les appels sans valeur de retour peuvent être accumulés et envoyés d'un coup pour plus de performance.
- Authentification.
- Sémantique au moins une fois.
- Sur UDP, chaque requête ou réponse est limitée à 8Koctets.

```
const MAX=1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;

struct data {
    int length;
    char buffer[MAX];
};

struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};

struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
    version VERSION {
        void WRITE(writeargs)=1;
        data READ(readargs)=2
    }=2;
}=9999;
```



```
// Server
```

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "FileReadWrite.h"

void *write_2(writeargs *a) {
    /* supply implementation */
}

Data *read_2(readargs *a) {
    /* supply implementation */
}
```

```
// Client
```

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

```
#include <rpc/rpc.h>
```

```
#include "FileReadWrite.h"
```

```
main(int argc, char *argv[]) {
```

```
    CLIENT cl;
```

```
    struct data *d;
```

```
    struct readrags r;
```

```
    cl = clnt_create("server.polymtl.ca",FILEREADWRITE,VERSION,"tcp");
```

```
    if(cl == NULL) {...}
```

```
    d = READ(&r,cl);
```

```
    if(d == NULL) {...}
```

```
}
```

- **Appel de méthodes à distance**

- Modèle objet: références aux objets, interfaces, méthodes, exceptions, ramasse-miettes.
- Proxy: se présente comme un objet local, ses méthodes sérialisent les arguments, transmettent la requête au module de communication et retournent la réponse reçue.
- Répartiteur: reçoit les requêtes dans le serveur et les communique au squelette correspondant.
- Squelette: reçoit les requêtes pour un type d'objet, déséréalise les arguments, appelle la méthode correspondante de l'objet référencé et sérialise la réponse à retourner.
- Module de communication: envoi de requête client (type, numéro de requête, référence à un objet, numéro de méthode, arguments), la requête reçue par le serveur est envoyée au répartiteur et la réponse est retournée.

- **Module des références réseau**

- Table des objets importés (pour chaque objet distant utilisé, adresse réseau et adresse du proxy local correspondant),
- Table des objets exportés (pour chaque objet utilisé à l'extérieur, adresse réseau et adresse locale).
- Lorsqu'une référence réseau est reçue, son proxy ou objet local est trouvé ou un nouveau proxy est créé et une entrée ajoutée dans la table.
- Lorsqu'un objet réseau local est passé en argument, son adresse réseau le remplace et il doit se trouver dans la table des objets exportés.

- **Services pour l'appel de méthodes distantes**
 - Service de nom: permet d'obtenir une référence objet réseau à partir d'un identificateur/nom.
 - Contextes d'exécution: le serveur peut créer un fil d'exécution pour chaque requête.
 - Activation des objets: le serveur peut être démarré automatiquement lorsqu'une requête pour un objet apparaît (message augmentation de salaire pour le budget qui est stocké sur disque dans un fichier de chiffrier).
 - Stockage des objets persistents.
 - Service de localisation.

• Gestion de la mémoire en réparti

- Malloc et Free difficiles à appeler au bon moment en réparti!
- Chaque client s'enregistre lorsqu'il utilise un objet et avertit lorsqu'il en a terminé. Le serveur libère un objet si ni lui ni les clients ne l'utilisent.
- Ramasse-miette conventionnel qui travaille en réparti? Très difficile et inefficace puisqu'il faut tout arrêter en même temps lors du ramassage.
- Ramasse-miette local à chaque processus avec notification automatique au serveur lorsqu'un client cesse d'utiliser un proxy.
 - Une référence à un objet O est envoyée de A à B et A cesse de l'utiliser, le message de A (référence à O inutilisée) peut arriver avant celui de B (utilise O).
 - Cycle de références entre 2 clients ou plus.
 - Si le client disparaît, le serveur doit l'enlever de la liste (limite de validité pour les références, ou besoin de messages de maintien keepalive).

• Le Remoting en C#

- Fonctionne entre tous les langages supportés par le CLR: C#, C++, VB...
- Utilise la réflexivité pour générer le code client et serveur dynamiquement.
- Choix d'encodage binaire ou XML.
- Choix de canal TCP ou HTTP.
- Les objets qui héritent de MarshalByRef sont passés par référence à travers le réseau, les autres sont passés par valeur et doivent supporter l'interface ISerializable.
- Un objet déjà créé peut être exporté ou on peut enregistrer un type et l'objet est créé au moment de la requête selon un de deux modes (Singleton, Single call).
- Les références aux objets peuvent avoir une date d'expiration pour éviter d'avoir à vérifier pour les références inutilisées; passé cette expiration elles n'existent plus.

```
// Interface mise en librairie et utilisée par le client
// et par le serveur.
```

```
using System;
```

```
abstract public class Server: MarshalByRefObject {
    abstract public string GetInfo();
}
```

```
// Client
```

```
using System;
using System.IO;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```

```
class Client
{
    public static void Main(string[] args) {
        int serverPort = 9090;
        host = "localhost";
        TcpChannel channel = new TcpChannel();
        ChannelServices.RegisterChannel(channel);
        Server server = (Server)RemotingServices.Connect(
            typeof(Server),
            "tcp://" + host + ":" + serverPort.ToString() + "/MyServer");
        string info = server.GetInfo();
    }
}
```

```
// Serveur
using System;
using System.IO;
using System.Threading;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```

```
class AServer: Server {
    public override string GetInfo() {
        return "A Server version 0.01"
    }
}
```

```
public static void Main(string[] args) {
    int serverPort = 9090;
    TcpChannel channel = new TcpChannel(serverPort);
    ChannelServices.RegisterChannel(channel);
    AServer server = new AServer();
```

```
// Un objet est créé et exporté explicitement.
// Beaucoup d'applications préfèrent laisser les objets être créés au
// besoin par un appel à
// RemotingConfiguration.RegisterWellKnownServiceType
// en spécifiant si la création est une fois (Singleton) ou
// à chaque appel (SingleCall).
```

```
ObjRef serverRef = RemotingServices.Marshal(server, "MyServer",
    typeof(Server));
```

```
// Ce serveur est disponible pour un temps limité...
Thread.Sleep(20000);
RemotingServices.Disconnect(server);
ChannelServices.UnregisterChannel(channel);
```

```
}
}
```


• Java RMI

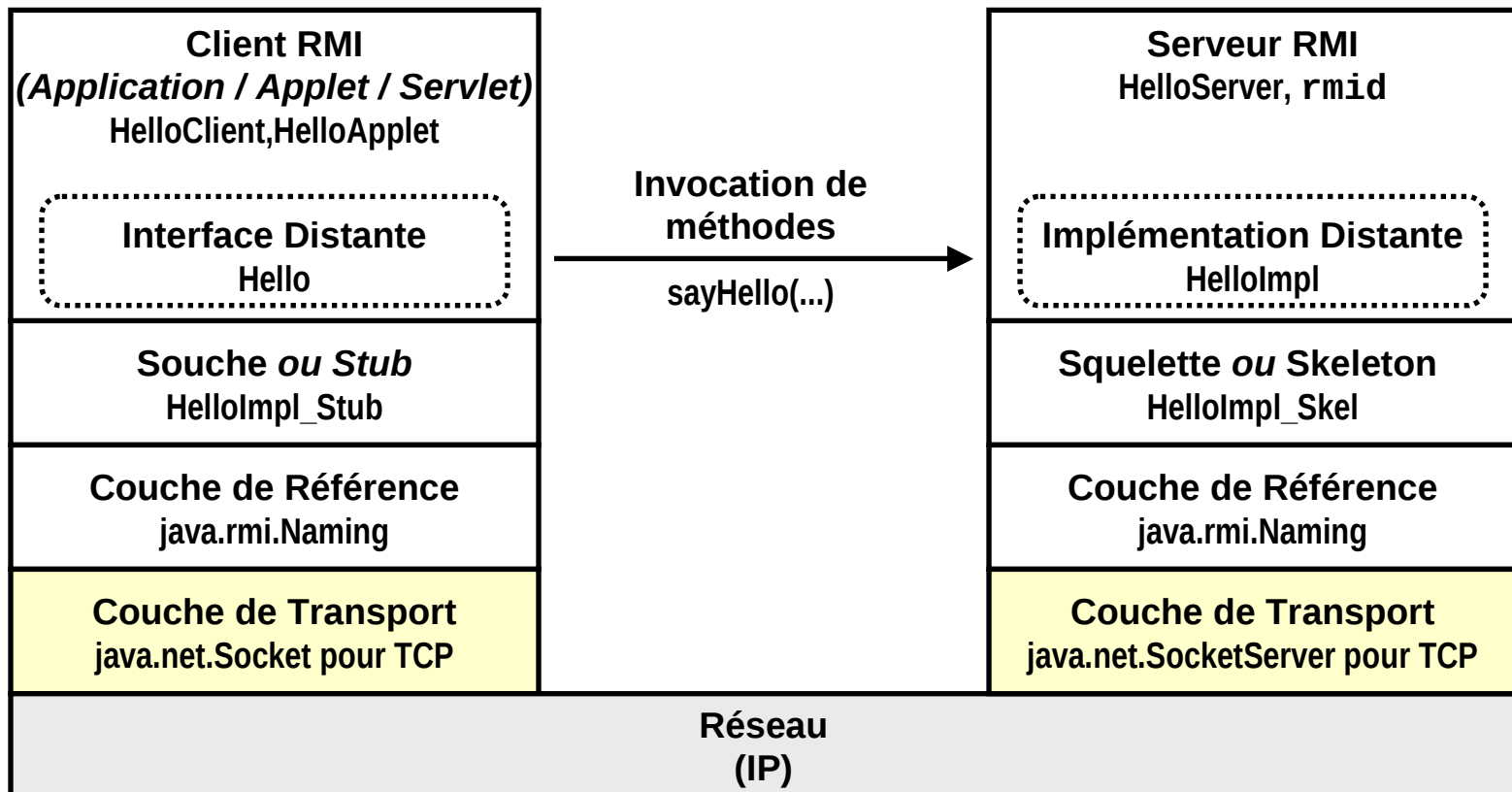
- Ne fonctionne que de Java à Java.
- S'applique à tout type qui implante l'interface Remote.
- Les méthodes doivent accepter l'exception RemoteException.
- Les arguments seront envoyés et la valeur retournée sert de réponse.
- Ces arguments doivent être des types primitifs, des objets réseau, ou des objets qui implantent l'interface Serializable (le graphe complet d'objets rejoints par un argument peut être transmis).
- Le type des objets sérialisés contient une référence à la classe. La classe peut être téléchargée au besoin par le récepteur.
- Le type des objets réseau contient une référence à leur classe proxy (stub) qui peut être téléchargée au besoin.

- **Java RMI**

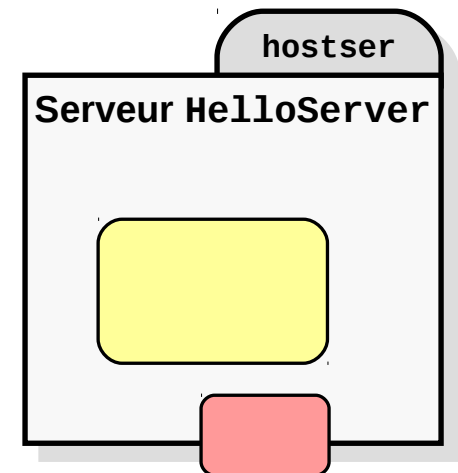
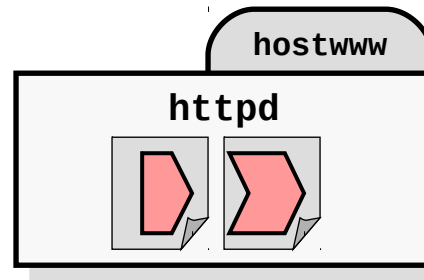
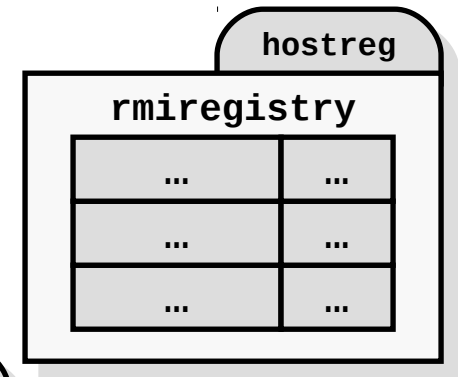
- Le RMIregistry maintient une table (nom, référence réseau) pour obtenir une référence à un objet désiré qui se trouve sur un ordinateur donné. Un nom a la forme:
//hostname:port/objectName.
- rmic peut être utilisé pour créer les proxy à partir du code des classes compilées. Le répartiteur est générique et fourni en librairie.
- Les appels distants peuvent être servis par des fils d'exécution différents, surtout s'il viennent de clients (connexions) différents.

Structure des couches RMI

l'architecture logique









La configuration

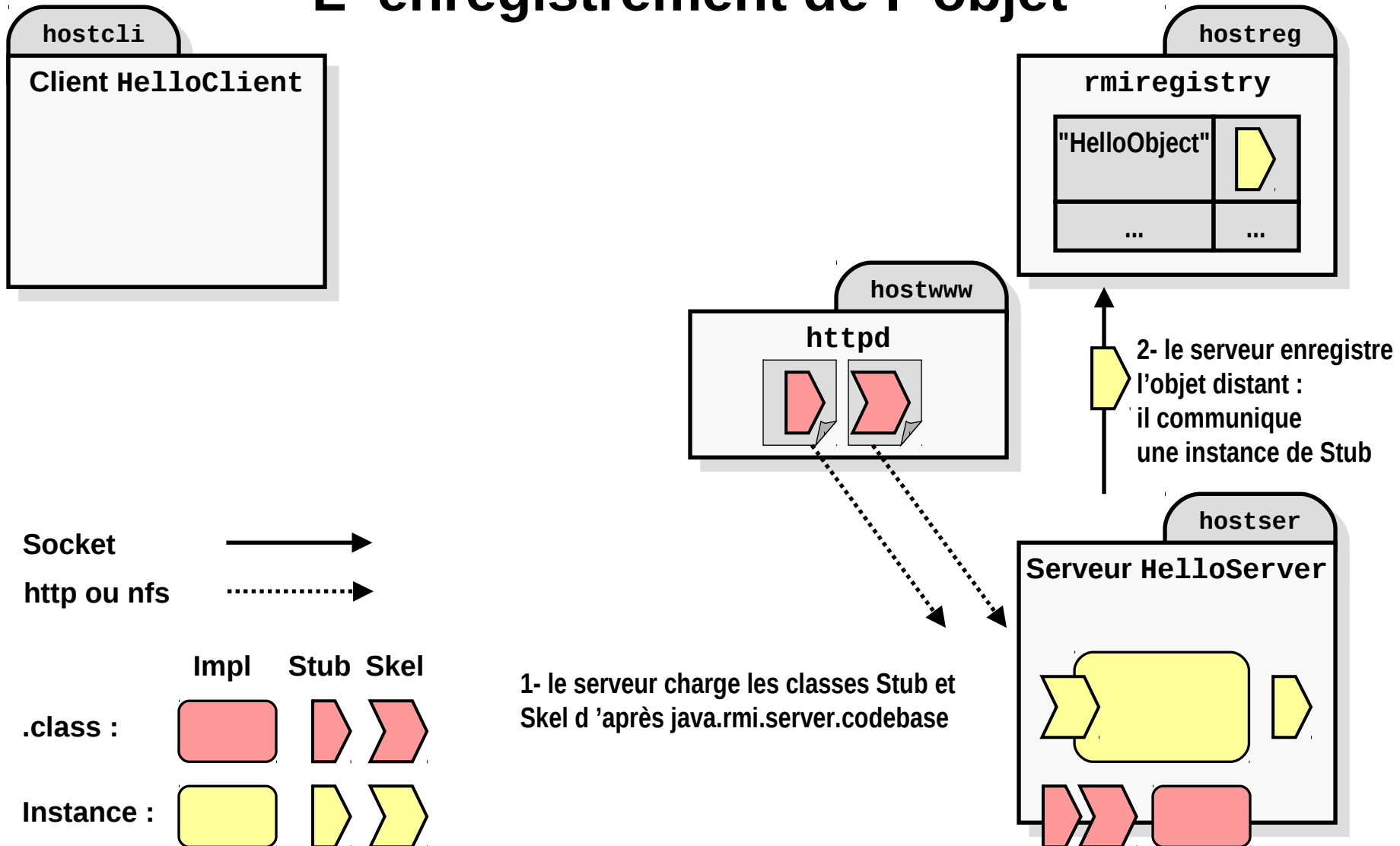


Socket 

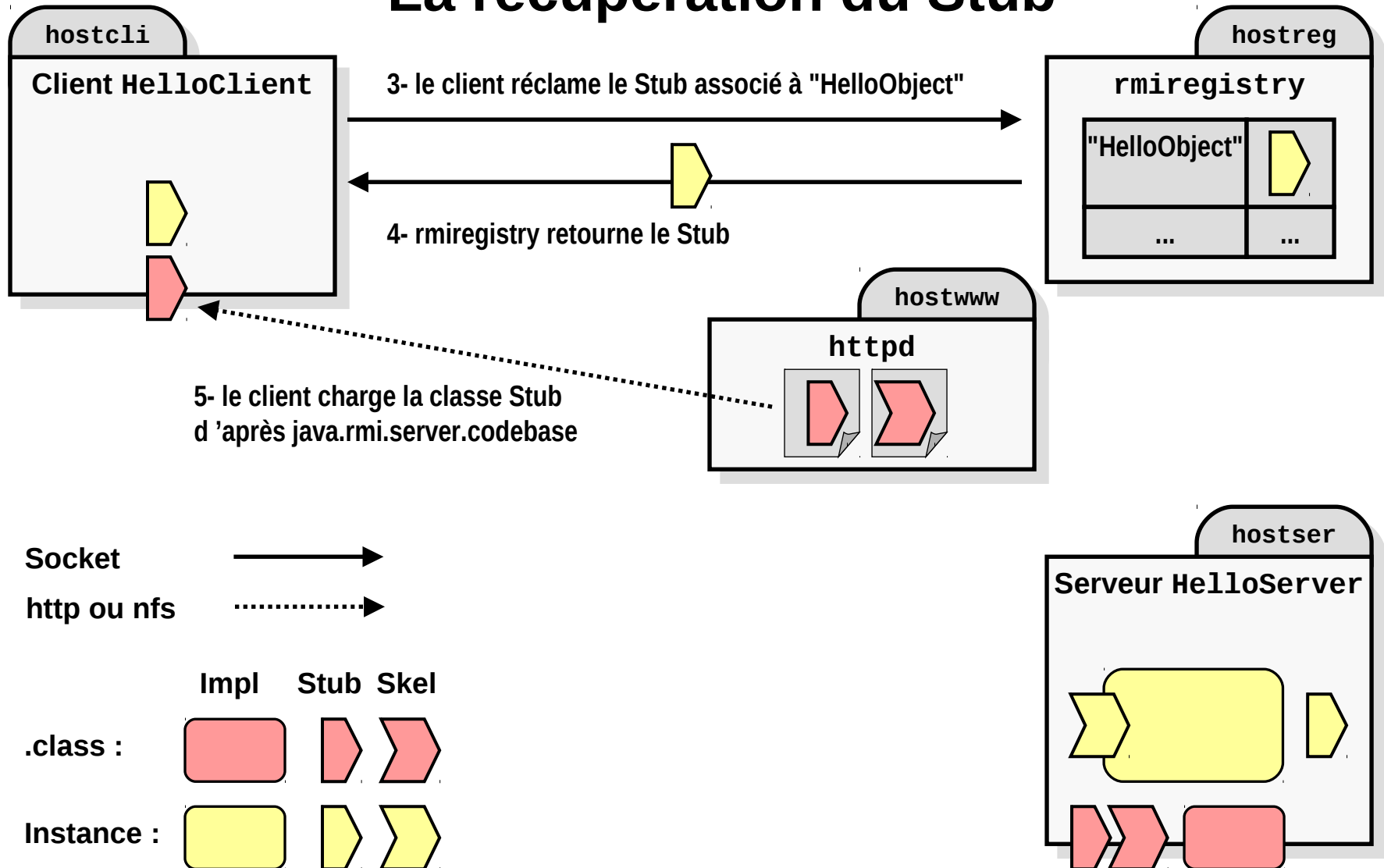
http ou nfs 

	Impl	Stub	Skel
.class :			
Instance :			

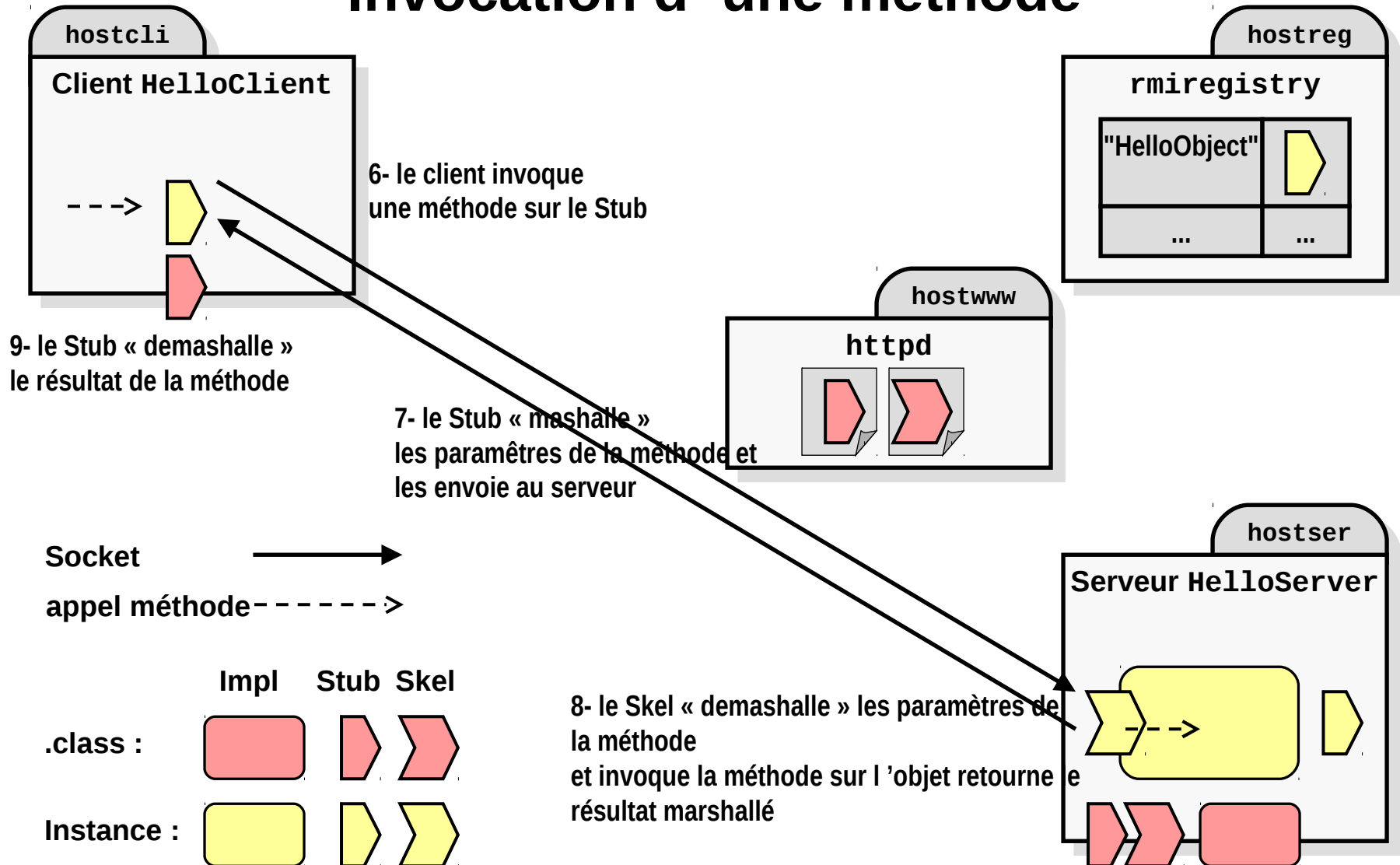
L'enregistrement de l'objet



La récupération du Stub



Invocation d'une méthode



Création et manipulation d'objets distants

- **5 Packages**

- java.rmi : pour accéder à des objets distants
- java.rmi.server : pour créer des objets distants
- java.rmi.registry : lié à la localisation et au nommage d'objets distants
- java.rmi.dgc : ramasse-miettes pour les objets distants
- java.rmi.activation : support pour l'activation d'objets distants

- **Étapes du développement**

- 1- Spécifier et écrire l'interface de l'objet distant.
- 2- Écrire l'implémentation de cette interface.
- 3- Générer les Stub/Skeleton correspondants.

- **Étapes de l'exécution**

- 4- Écrire le serveur qui instancie l'objet implémentant l'interface, exporte son Stub puis attend les requêtes via le Skeleton.
- 5- Écrire le client qui réclame l'objet distant, importe le Stub et invoque une méthode de l'objet distant via le Stub.

// Interface réseau Forme

```
package examples.RMIShape;
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws
    RemoteException;
}
```

// Interface réseau dessin (liste de Forme)

```
package examples.RMIShape;
import java.rmi.*;
import java.util.Vector;

public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws
    RemoteException;
    Vector allShapes()throws RemoteException;
    int getVersion() throws RemoteException;
}
```

// Serveur de dessin

```
package examples.RMIShape;
import java.rmi.*;

public class ShapeListServer {
    public static void main(String args[]){
        System.setSecurityManager(new
        RMISecurityManager());
        System.out.println("Main OK");
        try{
            ShapeList aShapelist = new
            ShapeListServant();
            System.out.println("After create");
            Naming.rebind("ShapeList", aShapelist);
            System.out.println("ShapeList server ready");
        }
        catch(Exception e) {
            System.out.println("ShapeList server main "
            + e.getMessage());
        }
    }
}
```

// Dessin exporté par le serveur

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ShapeListServant extends UnicastRemoteObject
    implements ShapeList{
    private Vector theList;
    private int version;

    public ShapeListServant()throws RemoteException{
        theList = new Vector();
        version = 0;
    }

    public Shape newShape(GraphicalObject g) throws
    RemoteException{
        version++;
        Shape s = new ShapeServant( g, version);
        theList.addElement(s);
        return s;
    }

    public Vector allShapes()throws RemoteException{
        return theList;
    }

    public int getVersion() throws RemoteException{
        return version;
    }
}
```

// Forme exportée par le serveur

```
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShapeServant extends
    UnicastRemoteObject implements Shape {
    int myVersion;
    GraphicalObject theG;

    public ShapeServant(GraphicalObject g, int
    version)throws RemoteException{
        theG = g;
        myVersion = version;
    }

    public int getVersion() throws RemoteException {
        return myVersion;
    }

    public GraphicalObject getAllState() throws
    RemoteException{
        return theG;
    }
}
```

```
// Client
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
import java.awt.Rectangle;
import java.awt.Color;

public class ShapeListClient{
    public static void main(String args[]){
        String option = "Read";
        String shapeType = "Rectangle";
        if(args.length > 0) option = args[0]; // read or write
        if(args.length > 1) shapeType = args[1];
        System.out.println("option = " + option + "shape = " + shapeType);
        if(System.getSecurityManager() == null){
            System.setSecurityManager(new RMISecurityManager());
        }
        else System.out.println("Already has a security manager");

        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList)
Naming.lookup("//test.shapes.net/ShapeList");
            System.out.println("Found server");
            Vector sList = aShapeList.allShapes();
            System.out.println("Got vector");
            if(option.equals("Read")){
                for(int i=0; i<sList.size(); i++){
                    GraphicalObject g = ((Shape)sList.elementAt(i)).getAllState();
                    g.print();
                }
            }
        }
    }
}
```

```
} else {
    GraphicalObject g = new
    GraphicalObject(shapeType,
        new
    Rectangle(50,50,300,400),Color.red,
        Color.blue, false);
    System.out.println("Created graphical
object");
    aShapeList.newShape(g);
    System.out.println("Stored shape");
}
}
catch(RemoteException e) {
    System.out.println("allShapes: " +
e.getMessage());
}
catch(Exception e) {
    System.out.println("Lookup: " +
e.getMessage());
}
}
}
```

- **Java Enterprise Edition (EE)**

- Au-dessus de Java Standard Edition (SE) et Java RMI, un API et environnement de déploiement pour les entreprises.
- J2EE 1.2 (1999), 1.3 (2001), 1.4 (2003), Java EE 5 (2006), 6 (2009), 7 (2013), 8 (2017).
- JavaServer Pages (JSP): interfaces pour HTTP.
- Unified Expression Language (EL): langage de script pour les expressions.
- JavaServer Faces (JSF): interface usager.
- Java API for RESTful Web Services (JAX-RS): support pour REST.
- Enterprise JavaBeans (EJB): support de composantes pour les *business objects*.
- Java Transaction API (JTA): support pour les transactions réparties.
- Java Persistence API (JPA): stockage de l'état dans une base de donnée.
- Bean Validation: annotations de contraintes.
- ...

• Enterprise Java Beans

- Programmer la logique de l'application séparément du reste de l'environnement: grappe pour le déploiement, interface usager, base de donnée, RPC, sécurité...
- Suppose une architecture classique à trois tiers (interface usager, logique de l'application, base de donnée).
- EJB 1.0 (1998): architecture de base.
- EJB 1.1 (1999): fichiers de méta-données en XML décrivant l'environnement, composantes (beans) de session et d'entité. Interface d'accès à distance.
- EJB 2.0 (2001): interface par message.
- EJB 2.1 (2003): Minuterie et support Web Service.
- EJB 3.0 (2006): POJO avec annotations remplace les méta-données XML.
- EJB 3.1 (2009): Quelques simplifications à l'architecture.
- EJB 3.2 (2013): Changements mineurs.

- **Conteneurs EJB**

- Logiciels comme JBoss (Red Hat), WebSphere (IBM), NetWeaver (SAP), WebLogic (Oracle), Geronimo (Apache), GlassFish (Sun).
- Reçoit les requêtes d'objets/clients locaux ou distants (RMI, RMI-IIOP, Web Services, JMS) qui fournissent l'interface usager.
- Les requêtes sont validées et dirigées vers les objets de session qui sont référencés ou créés au besoin (Business logic).
- Les objets entités sont accédés par les objets de session et leur état est géré et mis à jour dans la base de données selon ce qui a été spécifié dans les annotations (Persistence).

- **Les rôles selon EJB**

- Bean provider: fournisseur des composantes de l'application.
- Application assembler: concepteur de l'application qui assemble les composantes pour obtenir les fonctions désirées.
- Deployer: responsable du déploiement de l'application dans un environnement adéquat.
- Service provider: spécialiste des systèmes répartis qui s'assure du niveau de service désiré.
- Persistence provider: spécialiste des bases de données.
- Container provider: spécialiste de l'environnement d'exécution des composantes Java.
- System administrator: administrateur du système informatique qui s'assure que le système fonctionne selon ce qui a été conçu.

- **CORBA**

- Langage de définition d'interface avec constantes, structures, méthodes, et exceptions.
- Générateur de proxy et de squelette multi-langage.
- Possibilité d'appels asynchrones en l'absence de valeur de retour.
- Mécanisme pour la découverte et l'invocation dynamique de procédures.
- Divers services de notification...
- Semblable à Sun RPC mais plus sophistiqué et objet.

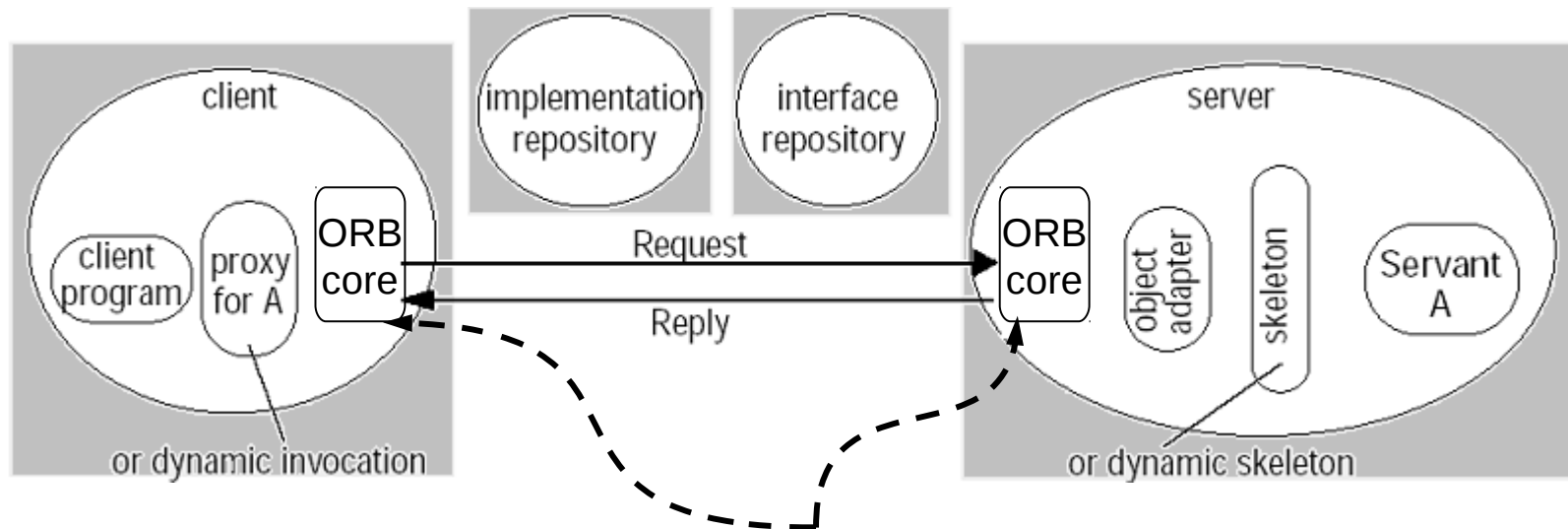
La norme CORBA

- Common Object Request Broker Architecture.
- L'OMG (Object management group) maintient la norme CORBA.
- C'est une architecture et infrastructure ouverte, indépendante du constructeur.
- CORBA **n'est pas** un Système Distribué mais sa spécification
 - spécifications principale de plus de 700 pages.
 - Plus 1200 pages pour spécifier les services naviguant autour
 - 1991: v1: standardisation de IDL et OMA
 - 1996: v2: spécifications plus robustes, IIOP, POA, DynAny.
 - 2002: v3: ajout du component model, QoS (asynchronisme, tolérance de panne, RT Corba- Corba temps réel).

- **Object Request Broker (ORB)**
 - Initialisation et finalisation.
 - Conversion de références aux objets.
 - Envoi de requêtes.
- **Portable Object Adapter (POA)**
 - Activation des objets.
 - Répartition des requêtes vers les objets via les squelettes.
 - Gestion des adresses réseau.
 - Portable: fonctionne avec des implantations CORBA différentes.

Architecture de CORBA

Object Request Broker (ORB)

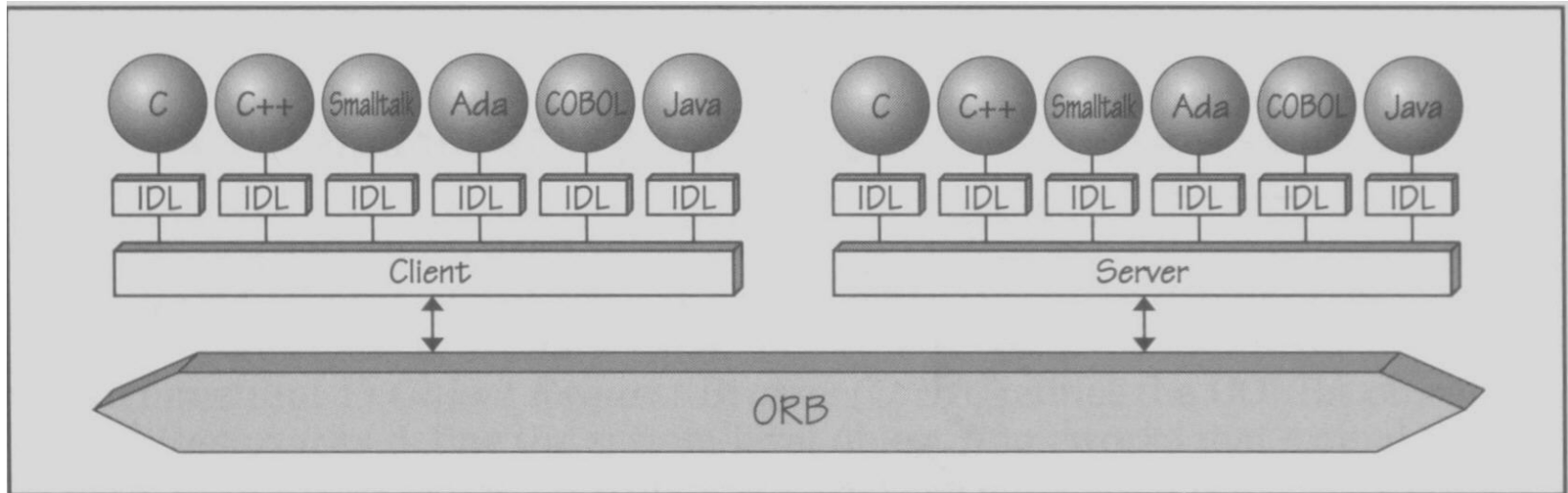


Fournit une interface incluant des opérations

- **Langage de définition d'interface**

- Module: définit un espace de nom.
- Interface: liste de méthodes, attributs et déclarations de types ou exceptions. Héritage simple ou multiple.
- Méthode: nom de fonction, arguments in/out/inout, valeur de retour, exceptions. Attribut oneway pour invocation asynchrone.
- Types: short, long, unsigned short, unsigned long, float, double, char, boolean, octet, any, struct, enum, union (tagged), array, string, sequence, object.
- Exception: peut contenir des variables.
- Attribut: variable de l'interface. Peut être *readonly*.

Le concept



- **Référence à un objet**

- IOR: Interoperable Object Reference.
- Format: identificateur du serveur, IIOP, hostname, port, nom de l'adaptateur, nom de l'objet.
- IIOP: protocole particulier, interoperable et basé sur TCP/IP.
- IOR transitoire ou persistant.
- Plusieurs champs serveur/port sont possibles pour les services avec réplication.
- Un serveur peut répondre avec une redirection.

- **Support pour différents langages**

- Chaque langage doit avoir son générateur idl et son interface de programmation CORBA.
- En C, un argument est utilisé pour vérifier les exceptions.
- En Java les struct, enum, et unions sont implantés avec des classes.
- En C++ la relation entre les constructeurs/destructeurs et la gestion de la mémoire est subtile.

- **Services CORBA**

- Noms: service hiérarchique (contexte, liste de (nom, contexte ou objet)). Part du contexte initial *racine*.
- Canal d'événement. Fournisseurs et clients qui peuvent chacun être appelant ou appelé (push/pull). Plusieurs fournisseurs et clients peuvent se connecter au même canal. Un canal peut être un fournisseur, ou un client pour un autre.
- Notification: service enrichi pour un canal d'événements. Les clients peuvent spécifier les événements d'intérêt et interroger les types d'événements offerts. Les fournisseurs peuvent spécifier les types qu'ils offrent et savoir lesquels sont d'intérêt pour un ou plusieurs clients.
- Sécurité: authentification de l'envoyeur et de la réponse, journal, liste de contrôle des accès.
- Transactions: transactions et transactions imbriquées (begin/commit/abort).
- Persistance.

- **CORBA Common Data Representation (CDR):**

- Permet de représenter tous les types de données qui peuvent être utilisées comme arguments ou valeurs de retour dans un appel à distance CORBA ;
- Il existe 15 types primitifs: Short (16bit), long(32bit), unsigned short, unsigned long, float, char, ...
- CDR offre la possibilité d'avoir des « *constructed types* » qui sont des types construits à partir de types primitifs.
- Le type d'un item de données n'est pas donné avec sa représentation dans le message: l'émetteur et le récipiendaire du message connaissent l'ordre et le type des données du message.

- Exemple du CDR :

```
Struct Person {
    string name;
    string place;
    long year;
};
```

*index dans la
séquence de bytes*

<div style="display: flex; align-items: center; justify-content: center;"> ← 4 bytes → </div>	
0–3	5
4–7	"Smit"
8–11	"h____"
12–15	6
16–19	"Lond"
20–23	"on____"
24–27	1934

Longueur du string
‘Smith’

Longueur du string
‘London’

unsigned long

```
interface DocumentServer
{
    void getServerInfo(out string version, out string date);
};
```

```

/* Programme client pour un serveur de document CORBA. */

/* Introduction des définitions requises pour utiliser CORBA. */

#include "stdio.h"
#include "orb/orbit.h"
#include "DocumentServer.h"

DocumentServer server;

int
main (int argc, char *argv[])
{
    CORBA_Environment ev;
    CORBA_ORB orb;
    CORBA_char    *version, *date;
    FILE * ifp;
    char * ior;
    char filebuffer[1024];

    /* Initialisation du service CORBA. */

    CORBA_exception_init(&ev);
    orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", &ev);

    /* Lecture dans un fichier de l'identificateur du serveur de
       document à accéder. */

    ifp = fopen("doc.ior","r");
    if( ifp == NULL ) {
        g_error("No doc.ior file!");
        exit(-1);
    }

    fgets(filebuffer,1024,ifp);
    ior = g_strdup(filebuffer);
    fclose(ifp);

```

```

/* Connexion avec le serveur. */

server = CORBA_ORB_string_to_object(orb, ior, &ev);

if (!server) {
    printf("Cannot bind to %s\n", ior);
    return 1;
}

/* Appel d'une fonction du serveur. */

DocumentServer_getServerInfo(server, &version, &date, &ev);

/* Vérification du code d'erreur. */

if(ev._major != CORBA_NO_EXCEPTION) {
    printf("we got exception %d from echoString!\n", ev._major);
    return 1;
}
else {
    /* Tout va bien, impression de la réponse et libération des objets. */
    printf("Version: %s, Date %s\n", version, date);
    CORBA_free(date);
    CORBA_free(version);
}

/* Fin du programme, libération de la connexion et du service
CORBA. */

CORBA_Object_release(server, &ev);
CORBA_Object_release((CORBA_Object)orb, &ev);

return 0;
}

```

```
/* Introduction des définitions requises pour utiliser CORBA. */
```

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "signal.h"
#include "orb/orbit.h"
#include "DocumentServer.h"
```

```
DocumentServer server = CORBA_OBJECT_NIL;
```

```
/* Prototype de la fonction qui implante le service offert. */
```

```
static void
do_getServerInfo(PortableServer_Servant servant,
                 CORBA_char **version, CORBA_char **date,
                 CORBA_Environment *ev);
```

```
/* Initialisation des objets CORBA servant à décrire le service. */
```

```
PortableServer_ServantBase__epv base_epv = {
    NULL,
    NULL,
    NULL
};
```

```
POA_DocumentServer__epv documentServer_epv = { NULL,
do_getServerInfo };
```

```
POA_DocumentServer__vepv poa_documentServer_vepv =
{ &base_epv, &documentServer_epv };
```

```
POA_DocumentServer poa_documentServer_servant =
{ NULL, &poa_documentServer_vepv };
```

```
int main (int argc, char *argv[])
{
    PortableServer_ObjectId objid =
        {0, sizeof("myDocumentServer"), "myDocumentServer"};
```

```
PortableServer_POA poa;
```

```
CORBA_Environment ev;
char *retval;
CORBA_ORB orb;
FILE * ofp;
```

```
/* Sortir proprement en cas d'interruption du programme */
```

```
signal(SIGINT, exit);
signal(SIGTERM, exit);
```

```
/* Initialisation du service CORBA */
```

```
CORBA_exception_init(&ev);
orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", &ev);
```

```
/* Initialisation du service de document */
```

```
POA_DocumentServer__init(&poa_documentServer_servant, &ev);
```

```
/* Contacter le localisateur d'objet pour lui donner une référence
à notre service de document. */
```

```
poa = (PortableServer_POA)CORBA_ORB_resolve_initial_references(orb,
    "RootPOA", &ev);
PortableServer_POAManager_activate(
    PortableServer_POA__get_the_POAManager(poa, &ev), &ev);
PortableServer_POA_activate_object_with_id(poa,&objid,
    &poa_documentServer_servant, &ev);
```

```
/* Obtenir une référence à notre service et la convertir en chaîne  
de caractères. */
```

```
server = PortableServer_POA_servant_to_reference(poa,  
        &poa_documentServer_servant, &ev);
```

```
if (!server) {  
    printf("Cannot get objref\n");  
    return 1;  
}
```

```
retval = CORBA_ORB_object_to_string(orb, server, &ev);
```

```
/* Ecrire la référence dans un fichier. */
```

```
ofp = fopen("doc.ior", "w");
```

```
fprintf(ofp, "%s", retval);  
fclose(ofp);
```

```
/* Libération de la référence produite par CORBA sous forme de  
chaîne de caractères. */
```

```
CORBA_free(retval);
```

```
/* Impression d'un message et attente des requêtes. */
```

```
fprintf(stdout, "En attente de requêtes...\n");  
fflush(stdout);  
CORBA_ORB_run(orb, &ev);
```

```
return 0;
```

```
}
```

```
/* Implantation de la fonction offerte en service CORBA */
```

```
static void  
do_getServerInfo(PortableServer_Servant servant,  
        CORBA_char **version, CORBA_char **date,  
        CORBA_Environment *ev)  
{  
    (*version) = CORBA_string_dup("0.99");  
    (*date) = CORBA_string_dup("24 janvier 2000");  
    return;  
}
```

```
// IDL
```

```
struct Rectangle{  
    long width;  
    long height;  
    long x;  
    long y;  
};
```

```
struct GraphicalObject {  
    string type;  
    Rectangle enclosing;  
    boolean isFilled;  
};
```

```
interface Shape {  
    long getVersion() ;  
    GraphicalObject getAllState() ;  
};
```

```
typedef sequence <Shape, 100> All;
```

```
interface ShapeList {  
    exception FullException {};  
    Shape newShape(in GraphicalObject g) raises  
(FullException);  
    All allShapes();  
    long getVersion();  
};
```

// Serveur

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Properties;

public class ShapeListServer
{
    public static void main(String args[]) {
        try{
            // create and initialize the ORB
            Properties props = new Properties();
            props.put("org.omg.CORBA.ORBInitialPort", "1500");
            ORB orb = ORB.init(args, props);

            // create servant and register it with the ORB
            ShapeListServant shapeRef = new ShapeListServant(orb);
            orb.connect(shapeRef);
            // get the root naming context
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            // bind the Object Reference in Naming
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, shapeRef);
            // wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }
        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

// Dessin offert par le serveur

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
class ShapeListServant extends _ShapeListImplBase
{
    private Shape theList[];
    private int version;
    private static int n=0;

    ORB theOrb;

    public ShapeListServant(ORB orb){
        theList = new Shape[4];
        version = 0;
        theOrb = orb;
    }

    public Shape newShape(GraphicalObject g)
        throws ShapeListPackage.FullException {
        version++;
        Shape s = new ShapeServant( g, version);
        if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        theOrb.connect(s);
        return s;
    }

    public Shape[] allShapes(){
        return theList;
    }

    public int getVersion() {
        return version;
    }
}
```


// Client

```
import org.omg.CosNaming.*;
import
org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.Properties;

import java.util.Vector;

public class ShapeListClient {
    public static void main(String args[]) {
        String option = "Read";
        String shapeType = "Rectangle";
        if(args.length > 0) option = args[0]; // read or write
        if(args.length > 1) shapeType = args[1];
        try{
            // create and initialize the ORB at port number
1500
            Properties props = new Properties();
            props.put("org.omg.CORBA.ORBInitialPort",
"1500");
            ORB orb = ORB.init(args, props);
```

```
        try{
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef =
NamingContextHelper.narrow(objRef);
            // resolve the Object Reference in Naming
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            try{
                ShapeList shapeListRef =
ShapeListHelper.narrow(ncRef.resolve(path));
                if(option.equals("Read")){
                    Shape[] sList = shapeListRef.allShapes();
                    for(int i=0; i<sList.length; i++){
                        GraphicalObject g = sList[i].getAllState();
                        g.print();
                    }
                } else {
                    GraphicalObject g = new GraphicalObject(shapeType,
                        new Rectangle(50,50,300,400), false);
                    try {
                        shapeListRef.newShape(g);
                    } catch(ShapeListPackage.FullException e) {}
                }
            } catch(Exception e){}
        } catch(org.omg.CORBA.ORBPackage.InvalidName e){}
    } catch (org.omg.CORBA.SystemException e) {
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
    }
}
}
```

- **SOAP**

- Simple Object Activation Protocol.
- Requête HTTP, action POST, contenu XML (nom de la fonction et arguments d'entrée), réponse XML avec arguments de retour.
- Facile à déployer car réutilise toute l'infrastructure Web.
- Relativement inefficace.

POST /InStock HTTP/1.1

Host: www.example.org

Content-Type: application/soap+xml;
charset=utf-8

Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="http://www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body

xmlns:m="http://www.example.org/stock">

<m:GetStockPrice>

<m:StockName>IBM</m:StockName>

</m:GetStockPrice>

</soap:Body>

</soap:Envelope>

HTTP/1.1 200 OK

Content-Type: application/soap+xml;
charset=utf-8

Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="http://www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body

xmlns:m="http://www.example.org/stock">

<m:GetStockPriceResponse>

<m:Price>34.5</m:Price>

</m:GetStockPriceResponse>

</soap:Body>

</soap:Envelope>

• RESTful API for Web Services

- Requête sans contexte (stateless), pour accéder une ressource représentée par un URL, en utilisant un certain format (e.g. JSON) et les méthodes HTTP (GET, PUT...).
- La méthode GET n'a aucun effet sur la ressource.
- Les méthodes PUT et DELETE sont idempotentes.
- Pas une norme, simplement un style architectural de service Web.
- De plus en plus populaire en raison de sa simplicité.
- Pas toujours bien utilisé (manques au niveau de la récupération d'erreurs, pas complètement sans contexte...).

GET /api/StockPrice/IBM.json HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: nnn

```
{ "CompanyName": "IBM",  
  "Price": "34.5"  
}
```

gRPC

- Acronyme pour **gRPC Remote Procedure Call**.
- Rendu disponible par Google en 2015: “A high performance, open-source universal RPC framework”.
- Communication avec HTTP2 (compression des entêtes).
- Générateurs de code pour une dizaine de langages populaires (C++, Java, Python, Go, Ruby, C#...).
- Permet différents formats pour les messages mais implante initialement Protobuf (Protocol Buffers).
- Permet d’insérer le support pour l’équilibrage de charge, le traçage, le monitoring et l’authentification.
- Utilisé par Google, Netflix, Twitter, Cisco, Juniper...

HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path =
/google.pubsub.v2.PublisherService/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc+proto
grpc-encoding = gzip
authorization = Bearer
y235.wef315yfh138vh31hv93hv8h3v

DATA (flags = END_STREAM)
<Length-Prefixed Message>

HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip
content-type = application/grpc+proto

DATA
<Length-Prefixed Message>

HEADERS (flags = END_STREAM,
END_HEADERS)
grpc-status = 0 # OK
trace-proto-bin = jher831yy13JHy3hc

- **Protocol Buffers**

- Semblable à SUN RPC, encodage binaire.

```
message Person {
  required string name = 1;
  optional string email = 3;
  enum PhoneType { MOBILE = 0; HOME = 1; WORK = 2; }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}

Message PersonInfo { int32 age = 1; int32 salary = 2; }

service Contact {
  rpc Check (Person) returns (PersonInfo) {}
}
```


- **Protocol Buffers**

```
Person person;  
person.set_name("John Doe");  
person.set_email("jdoe@example.com");  
person.SerializeToOstream(&output);  
  
person.ParseFromIstream(&input);  
cout << "Name: " << person.name() << endl;  
cout << "E-mail: " << person.email() << endl;
```

• Protocol Buffers

- Format binaire compatible vers l'avant et l'arrière.
- Chaque champ peut être obligatoire ou optionnel et vient avec un numéro (tag) pour l'identifier. Ce numéro permet de retrouver les bons champs même si un nouveau champ est inséré au milieu.
- Un message est une séquence de clé, type, valeur. La clé est le numéro de champs, le type est un entier de 3 bits (varint, float, length delimited...). La clé et le type sont groupés en un varint et prennent ensemble usuellement 1 octet.
- Les varint sont des entiers de longueur variable (0-127 ou $1b+0-127 + \text{varint}$). Un premier bit à un indique qu'un octet supplémentaire est utilisé. Petit boutien.
- Fonctionne entre les langages et entre les plates-formes.
- Il est possible de traiter un message même s'il contient des champs inconnus.
- Protoc génère le code pour initialiser, sérialiser et lire les types décrits de même que pour faire des RPC (un type pour l'envoi et un pour la réponse).

```
// route_guide.proto, (tiré de https://grpc.io/docs/tutorials/basic/c.html)
```

```
syntax = "proto3";  
package routeguide;
```

```
service RouteGuide {  
  rpc GetFeature(Point) returns (Feature) {}  
  rpc ListFeatures(Rectangle) returns (stream Feature) {}  
}
```

```
message Point {  
  int32 latitude = 1;  
  int32 longitude = 2;  
}
```

```
message Rectangle {  
  Point lo = 1;  
  Point hi = 2;  
}
```

```
message Feature {  
  string name = 1;  
  Point location = 2;  
}
```

```
// route_guide.pb.h, the header which declares your generated message classes  
// route_guide.pb.cc, which contains the implementation of your message classes  
// route_guide.grpc.pb.h, the header which declares your generated service classes  
// route_guide.grpc.pb.cc, which contains the implementation of your service classes
```

```
// route_guide_server.cc
...
class RouteGuideImpl final : public RouteGuide::Service {
public:
    explicit RouteGuideImpl(const std::string& db) {
        routeguide::ParseDb(db, &feature_list_);
    }

    Status GetFeature(ServerContext* context, const Point* point,
                     Feature* feature) override {
        feature->set_name(GetFeatureName(*point, feature_list_));
        feature->mutable_location()->CopyFrom(*point);
        return Status::OK;
    }

    Status ListFeatures(ServerContext* context,
                       const routeguide::Rectangle* rectangle,
                       ServerWriter<Feature>* writer) override {
        for (const Feature& f : feature_list_) {
            if (inside(f.location(), rectangle)) {
                writer->Write(f);
            }
        }
        return Status::OK;
    }
private:

    std::vector<Feature> feature_list_;
};
```

```
void RunServer(const std::string& db_path) {
    std::string server_address("0.0.0.0:50051");
    RouteGuideImpl service(db_path);

    ServerBuilder builder;
    builder.AddListeningPort(server_address,
                             grpc::InsecureServerCredentials());
    builder.RegisterService(&service);
    std::unique_ptr<Server>
        server(builder.BuildAndStart());
    server->Wait();
}

int main(int argc, char** argv) {
    std::string db = routeguide::GetDbFileContent(argc, argv);
    RunServer(db);
    return 0;
}
```

```

class RouteGuideClient {
public:
    RouteGuideClient(std::shared_ptr<Channel>
        channel, const std::string& db)
        : stub_(RouteGuide::NewStub(channel)) {}

    void ListFeatures() {
        routeguide::Rectangle rect;
        Feature feature;
        ClientContext context;

        rect.mutable_lo()->set_latitude(4000000000);
        rect.mutable_lo()->set_longitude(-7500000000);
        rect.mutable_hi()->set_latitude(4200000000);
        rect.mutable_hi()->set_longitude(-7300000000);

        std::unique_ptr<ClientReader<Feature> >
            reader(stub_->ListFeatures(&context, rect));

        while (reader->Read(&feature))
            std::cout << "Found feature called " <<
                feature.name() << std::endl;

        Status status = reader->Finish();
    }
}

```

```

bool GetFeature(const Point& point,
    Feature* feature) {
    ClientContext context;

    Status status = stub_->GetFeature(&context,
        point, feature);

    if (!status.ok() || !feature->has_location() ||
        feature->name().empty()) return false;

    std::cout << "Found feature called " <<
        feature->name() << std::endl;
    return true;
}

std::unique_ptr<RouteGuide::Stub> stub_;
std::vector<Feature> feature_list_;
};

int main(int argc, char** argv) {
    RouteGuideClient guide(
        grpc::CreateChannel("localhost:50051",
            grpc::InsecureChannelCredentials()),db);
    Feature feature;

    guide.GetFeature(MakePoint(0,0), &feature);
    guide.ListFeatures();
    return 0;
}

```

- **Exemples d'utilisation des RPC**

- Quelques services comme NFS sont basés sur les Sun RPC.
- Les principaux programmes dans le bureau GNOME (chiffrier, fureteur, éditeur, panneau...) offraient une interface CORBA mais se tournent maintenant vers D-BUS.
- CORBA est souvent utilisé par les compagnies de télécommunications pour leurs applications réparties de gestion de réseau.
- Java RMI est souvent utilisé dans des applications réparties internes.

- **Autres mécanismes**

- D-Bus : commun à GNOME et KDE, (en remplacement de CORBA et DCOP). Bus système (e.g., notification de batterie, réseau, clé USB...) et bus de session (e.g., sélection).
- Websockets : communication duplex entre client et serveur avec entête HTTP.
- AMQP, ZeroMQ : solutions de remplacement plus performantes et plus légères pour la communication inter-processus et l'appel de procédures à distance (Request–reply, Publish–subscribe, Push–pull, Exclusive pair), avec option at-least-once, at-most-once, exactly-once. Utilisé pour le courtage, OpenStack...

• Conclusion

- Les RPC sont un mécanisme intéressant pour formaliser les interfaces de manière à permettre la communication entre composantes possiblement distantes et dans un langage de programmation différent.
- Les Sun RPC sont encore assez répandus pour certains services.
- Java RMI est simple d'utilisation et est utilisé dans des systèmes homogènes Java.
- Le Remoting est simple d'utilisation et fonctionne avec plusieurs langages (C#, C++, VB). Il remplace très avantageusement DCOM.
- CORBA est depuis plusieurs années déployé à grande échelle mais des solutions plus simples et performantes sont populaires comme gRPC, AMQP et ZeroMQ.
- SOAP, REST, Ajax... sont très utilisés sur le Web.