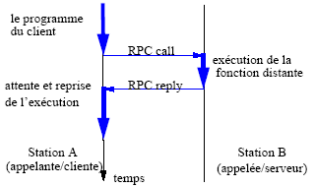
V.1. Généralités, RPC en C

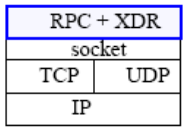
Appels de procédure (ou fonction) distante (RPC = Remote Procedure Call)

Le client appelle une procédure contenue dans le programme du serveur puis reçoit le résultat (appel à priori synchrone)



RPC et modèle OSI

La communication se fait via des Sockets (transmission de messages)



Problème RPC

Si on tient compte des erreurs (pertes, duplications) pouvant survenir lors des communications, on définit 3 sémantiques possibles pour l'appel de procédures distantes : "exactement une fois", "au moins une fois" ou "au plus une fois".

La sémantique choisie par l'implémentation sous RPC SUN est "au moins une fois" (réémission jusqu'à réception d'une réponse du serveur) ; il faut donc s'assurer que l'exécution d'une procédure distante est idempotente (par exemple, en utilisant le numéro de transaction (xid) disponible dans chaque message RPC).

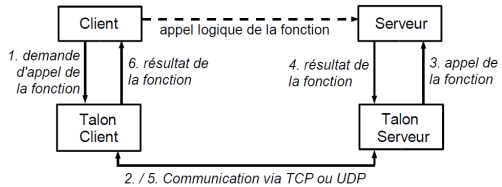
Autres problèmes RPC

- Problème du codage des données, de base (entiers, réels, ...) ou construits (chaînes, ...) en environnement hétérogène. Par exemple, les entiers peuvent être codés en little ou big endian, les caractères peuvent être codés en en ASCII ou autre (EBCDIC …), les tableaux, chaînes … peuvent être représentés différemment selon le compilateur utilisé.
Il y a alors nécessité d'une part de définir un format commun/d'échange (XDR, voir plus loin) et d'autre part de pouvoir réaliser l'encodage et le décodage des données selon ce format (format local <=> format commun)

- Autre problème, les pointeurs : une adresse mémoire perd sons sens sur une autre machine. Le problème peut être encore plus important dans le cas d'une arborescence de pointeurs : il faut alors pouvoir transformer cette arborescence en zone compacte. Nécessité de talons côté client (stub) et côté serveur (skeleton)

Talons côté client (stub) et serveur (skeleton)

Le client appelle localement la procédure distante sur le stub, lequel est chargé d'encoder les paramètres puis de relayer la demande vers le serveur. Côté serveur, le skeleton, décode les paramètres reçus puis aiguille l'appel vers la bonne procédure ; lorsqu'il reçoit le résultat, il l'encode puis l'envoie au skeleton ...



Le talon/souche client (stub) :
- reçoit l'appel de fonction en mode local et le transforme en appel distant (message envoyé au skeleton) après avoir encodé les paramètres
- reçoit les résultats en provenance du skeleton, les décode et les fournit au programme appelant

Le talon/souche serveur (skeleton) :
- reçoit le message d'appel et l'aiguille vers la bonne fonction après avoir décodé les paramètres
- récupère le résultat (après exécution de la fonction), l'encode puis l'envoie à destination du stub.

Langage RPC

Un programme (ou service) RPC est un ensemble de procédures (fonctions).

Chaque programme possède un nom et un numéro associé. Un programme peut exister en plusieurs versions. Chaque version est également identifiée par un nom et un numéro. Chaque version définit un ensemble de fonctions. Chaque fonction est également identifiée par un numéro.

Certains numéros de programme sont réservés pour certains services. Le programmeur doit choisir le sien dans l'intervalle allant de 0x20000000 à 0x3FFFFFFF

Numéros de programmes : 4 plages de valeurs, en hexadécimal :
- 0000 0000 à 1FFF FFFF : gérés par Sun
- 2000 0000 à 3FFF FFFF : programmeurs
- 4000 0000 à 5FFF FFFF : transient
- 6000 0000 à FFFF FFFF : réservé, non utilisé

Finalement, une fonction distante est identifiée de façon unique par un triplet : numéro de programme, numéro de version, numéro de fonction.

Il faut aussi décrire les types de données associés aux fonctions. Une fonction ne peut prendre qu'un seul paramètre en entrée et un seul en sortie : si on veut lui passer plusieurs paramètres en entrée ou récupérer plusieurs paramètres en sortie, on doit les regrouper dans une structure dédiée.

Exemple : service ping

```
program PING_PROG {
   version PING_VERS_PINGBACK {
      void PINGPROC_NULL(void) = 0; /* ping the caller, return the round-trip
      time in milliseconds. Return a minus one (-1) if operation times-out */
      int PINGPROC_PINGBACK(void) = 1; /* void - above is an argument to
      the call */
   } = 2;
   version PING_VERS_ORIG { /* original version */
      void PINGPROC_NULL(void) = 0;
   } = 1;
} = 0x20000001;

const PING_VERS = 2; /* latest version */
```

Remarques :

- On peut déclarer deux fonctions de même nom, signature et identifiant à condition qu'elles n'appartiennent pas à la même version dans le même programme.
- Par convention, on écrit les noms de fonctions, programmes et versions en majuscule.

Services RPC sous UNIX

Sous UNIX, le fichier des services RPC standards connus est /etc/rpc (liste des services RPC avec leurs alias). Exemple d'un service RPC connu : NFS.

Le service de nommage (= annuaire RPC = démon portmap ou portmapper)
- Il sert à enregistrer les services RPC tournant sur la machine serveur. Le service RPC s'enregistre auprès de son portmapper en précisant son numéro de programme, sa version et ses procédures.

- Le portmapper est lui-même un programme RPC : c'est le service rpcbind (dans /etc/rpc) qui utilise le numéro de port réservé 111. Les services RPC sont en général lancés par inetd, le portmapper doit être lancé avant.

- Les clients n'ont besoin de connaître que le seul numéro de port 111. Lorsqu'il reçoit une demande de la part d'un client, le portmapper la redirige alors vers le numéro de port correspondant au service.

La commande **rpcinfo** : Permet de connaître la liste des programmes RPC, versions et procédures disponibles sur un hôte.
- **rpcinfo -p [hote]** : donne la liste des services RPC enregistrés sur la machine hote (locale par défaut)
- **rpcinfo -[ut] hote pgme [vers]** : appel de la procédure 0 du service UDP (pour -u) ou TCP (pour -t) du programme *pgme* sur la machine *hote*. Remarque : la procédure 0 (procédure vide) doit toujours exister. Elle sert à vérifier l'existence du service.
- **rpcinfo -d hote pgme [vers]** : efface le service.

Un service basé sur les RPC : NFS

Utiliser des systèmes de fichiers distants. D'un côté (serveur), on exporte des répertoires et des fichiers, de l'autre (client), on monte ceux-ci dans sa propre arborescence. Initialement prévu pour les systèmes UNIX et étendu depuis à d'autres systèmes (Windows, Mac OS).

Plusieurs versions de NFS :

- *Versions 1, 2 et 3* : Communication initialement basée sur UDP (puis étendue progressivement à TCP), sans état (avec utilisation d'un cookie sur le client), gestion rudimentaire de la sécurité.
- *Version 4* : Gestion plus forte de la sécurité (authentification des users/groups et chiffrement des échanges avec Kerberos, certificats SPKM et LIPKEY), fonctionnement sur internet avec traversée des pare-feux, mécanisme de reprise sur incident, gestion d'état. Requiert TCP. La version 4 n'a plus besoin du portmapper.

Les processus démons :

- *rpc.mountd* : Gestion des requêtes de montage. N'est plus utilisé en version 4.
- *rpc.nfsd* : Serveur NFS. Démon principal, lancé en plusieurs exemplaires sous forme de threads (un pour chaque client).
- *rpc.statd* : Pour prévenir les clients d'un redémarrage du serveur (après un arrêt brutal). N'est plus utilisé en version 4.
- *rpc.lockd* : Processus facultatif qui permet au client de verrouiller des fichiers sur le serveur NFS. N'est plus utilisé en version 4.

- rpc.*rquotad* : Fournit des informations sur les quotas utilisateurs.
- *rpc.idmapd* : Utilisé par le serveur et le client pour traduire les identifiants d'utilisateurs et de groupes en noms et vice-versa (mécanisme d'appels dits "ascendants"). Nécessaire en version 4.
- rpc.*svcgssd* et rpc.*gssd* : Pour l'identification Kerberos respectivement côté serveur et côté client. Nécessaire en version 4.

Les fichiers serveur (UNIX) :

- /etc/hosts.allow et /etc/hosts.deny pour déterminer si un client est autorisé à utiliser le service NFS
- /etc/exports pour définir les dossiers partagés et les hôtes autorisés à y accéder

Les fonctions RPC :

Une requête NFS est implémentée sous forme d'une ou plusieurs fonctions RPC

Par exemple, une requête de copie est engendre les 5 appels de fonction suivantes :
- getattr(…)
- lookup(…)
- read(…)
- create(…)
- write(…)

Nom de la procédure	Rôle de la procédure
null	ne retourne rien, pour mesurer le temps d'un aller-retour vers un serveur
lookup (dirfh,nom)	retourne (fh, attr), c'est à dire un nouveau pointeur de fichier et les attributs de ce fichier
create (dirfh,nom,attr)	retourne (nouv_fh, attr), crée un nouveau fichier et renvoie son filehandle et ses attributs.
remove (dirfh,nom)	retourne (etat), retire un fichier d'un répertoire
getattr (fh)	retourne (attr), c'est à dire les attributs du fichier pointé par fh
read (fh depla,nbre)	renvoie (attr, données) et lit nbre octets à partir de l'octet depla
write (fh,depla,nbre,donnees)	renvoie (attr) et écrit nbre octets à partir de l'octet depla
link (dirfh,nom,tofh,tonom)	renvoie (etat)
symlink (dirfh,nom,chaine)	renvoie (etat)
readlink (fh)	renvoie (chaine)
mkdir (dirfh,nom,attr)	renvoie (fh, nouv_attr)
rmdir (dirfh,nom)	renvoie (etat)
readdir (dirfh,cookie,nbre)	renvoie (entrees)
statfs (fh)	renvoie (etat)

Langage XDR

But : éviter les problèmes de représentation des données d'une machine à l'autre (poids fort/faible, flottants IEEE/autre, structures complexes …)

Syntaxe issue du langage C, définie initialement dans le RFC 1014, plus récemment (Mai 2006) dans le RFC 4506. C'est le langage de description des types de données échangées lors du RPC. Toutes les données transitant via les sockets sous-jacentes sont codées/décodées via XDR

La conversion de la représentation locale vers XDR est appelée encodage ou sérialisation ou marshalling. L'opération inverse est appelée décodage ou désérialisation ou unmarshalling

Dans le modèle OSI, on peut considérer que RPC est de niveau 5 (couche session), XDR est de niveau 6 (couche présentation) et NFS (par exemple) de niveau 7 (couche application).

Utilisation et syntaxe proche du C : on peut utiliser des pointeurs, des énumérations, définir de nouveaux types avec typedef …

Types primitifs

- Booléen : **bool my_boolean;**
- Entier (32 bits) signé : **int my_int;**
  non signé : **unsigned int my_u_int;**
- Entier long (64 bits) signé : **hyper my_lg_int;**
  non signé : **unsigned hyper my_lg_u_int;**
- Flottants courts (32 bits/IEEE) : **float my_float;**
- Flottants longs (64 bits/IEEE) : **double my_lg_float;**
- Flottant très longs (128 bits/IEEE) : **quadruple my_lg_lg_float**;
- Rien (retourne toujours true) : **void;**

Types construits

- Tableau de taille fixe => comme en C : **int tab[10];**
- Tableau de taille variable :

```
int tab_var<10>;    /* tableau d'au plus 10 entiers */
int tab_var_2<>;    /* tableau d'entiers de taille quelconque */
```

- Chaine de caractères :

```
string chaine<32>;
string chaine_var<>;
```


- Données opaques :

```
opaque disk_block[512];
opaque file_data<1024>;
```

- Structure :

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

- Union : union tagguée (même principe qu'en pascal) : un tag indique quel champ/objet est en cours d'utilisation)

```
union read_result switch (int errno) {
    case 0:
        opaque data[1024];
    default:
        void;
};
```

Autres :

- Constante entière : **const max=100;** /* en C : #define max 100 */
- Enumération => Comme en C : **enum jour {ROUGE, VERT, BLEU};**
- Redéfinition de type (typedef) => comme en C : **typedef struct point point;**

Programmation des RPC : couche haute / couche basse

- Couche haute : Permet d'établir une communication RPC de façon simple (4 fonctions principales) mais moins de possibilités de gestion des communications que la couche basse. Elle n'utilise que le protocole UDP et de manière rigide. Elle ne permet pas de gérer les temporisations.

Couche basse : Plus complexe mais permet une gestion plus précise des communications ; elle permet de gérer le mode de transport (udp ou tcp), les appels non bloquants, les valeurs de timeout ou la fréquence des relances, la diffusion. RPCGEN (que l'on verra plus loin) utilise la couche basse.

Deux niveaux de temporisations :

- timeout : Temps maximal que l'on attend après une tentative d'appel avant de considérer le programme distant comme injoignable (fixé à 25 secondes par défaut)
- retry : Temps que l'on attend avant de réémettre un appel de fonction. Fixé à 5 secondes par défaut (si -1 : aucun nouvel essai). Valable uniquement avec UDP (avec TCP, pas de possibilité de perte une fois la connexion établie). Une même fonction peut donc être appelée plusieurs fois côté serveur pour un seul appel côté client.

Bibliothèque principale : **#include <rpc/rpc.h>**
On y trouve :

```
#include <rpc/svc.h>
#include <rpc/clnt.h>
etc.
```

Couche haute

int registerrpc(u_long prog_num, u_long vers_num, u_long fct_num, char *(*function)(), xdrproc_t xdr_param, xdrproc_t xdr_result);
/* Côté serveur. Enregistrement d'une fonction auprès du portmap. prog_num : numéro du programme. vers_num : numéro de version. fct_num : numéro de la fonction. function : pointeur sur la fonction à enregistrer. xdr_param : pointeur sur la fonction XDR de décodage des paramètres d'entrée. xdr_result : pointeur sur la fonction XDR d'encodage des résultats. registerrpc(...) retourne 0 si l'enregistrement a réussi ou -1 si erreur. */

void svc_run();
/* Côté serveur. Mise en écoute (boucle infinie) permettant d'attendre les requêtes client puis de les aiguiller vers les fonctions précédemment enregistrées. */

bool_t pmap_unset(u_long prog_num, u_long vers_num);
/* Côté serveur. Désenregistrement d'un couple programme-version complet (toutes ses fonctions) auprès du portmap (même si un exécutable est stoppé, il reste enregistré auprès du portmap). Retourne 1 si succès, 0 sinon. */

int callrpc(char *host, u_long prog_num, u_long vers_num, u_long fct_num, xdrproc_t xdr_param, char *in, xdrproc_t xdr_result, char *out);
/* Côté client. Appel d'une fonction distante. host : pointeur sur l'hôte. xdr_param : pointeur sur la fonction XDR d'encodage des paramètres d'entrée. in : pointeur sur les paramètres d'entrée. xdr_result : pointeur sur la fonction XDR de décodage des résultats. out : pointeur sur le buffer qui contiendra les résultats. callrpc(...) retourne une valeur de type enum clnt_stat (code d'erreur) castée en entier : valeur <> de 0 si problème (voir <rpc/clnt.h> pour le détail des codes d'erreurs de enum clnt_stat). */

Couche basse, fonctions côté serveur

SVCXPRT *svctcp_create(int sock, unsigned int send_buf_size, unsigned int recv_buf_size);
/* Création d'un objet SVCXPRT (structure contenant les caractéristiques du programme RPC et du client appelant une fonction ; voir sa définition dans rpc/svc.h). sock : descripteur de socket (si RPC_ANYSOCK, une nouvelle socket est créée et liée à un port quelconque). send_buf_size : taille du buffer d'envoi (si 0, taille par défaut). recv_buf_size : taille du buffer de réception (si 0, taille par défaut). Renvoie un pointeur sur la structure de type SVCXPRT créée ou NULL si erreur. */

SVCXPRT *svcudp_create(int sock);
SVCXPRT *svcudp_bufcreate(int sock, unsigned int send_buf_size, unsigned int recv_buf_size);
/* Même chose mais en mode UDP. Remarque : dans la première fonction, c'est la taille des buffers par défaut qui est utilisée. */

bool_t svc_register(SVCXPRT *svc, u_long num_prog, u_long num_vers, void (*dispatch)(svc_req *, SVCXPRT *), u_long protocol);
/* Enregistre un couple programme-version auprès du portmap en y associant une fonction d'aiguillage et un protocole. svc : pointeur sur la structure de type SVCXPRT initialisée. num_prog : numéro de programme. num_vers : numéro de version. dispatch : pointeur sur la fonction d'aiguillage. protocol : 0/IPPROTO_UDP/IPPROTO_UDP. svc_register(...) retourne 1 si succès et 0 sinon. */

void svc_unregister(u_long num_prog, u_long num_vers);
/* Désenregistre un couple programme-version auprès du portmap */

Couche basse, fonctions côté serveur (suite)

void dispatch (struct svc_req *req, SVCXPRT *svc); /* Fonction de dispatch à implémenter ou à générer avec RPCGEN (avec un nom différent de 'dispatch'). svc : pointeur sur la structure de type SVCXPRT initialisée. req : pointeur sur la structure de type svc_req contenant les caractéristiques de la requête d'appel + identification du client (voir sa définition dans rpc/svc.h). Fonctionnement : 1 : récupérer le numéro de la fonction à appeler dans la structure pointée par req ; 2 : décoder les paramètres d'entrée via la fonction svc_getargs(…) ; 3 : appeler la fonction demandée par le client avec les paramètres d'entrée ; 4 : encoder les résultats via la fonction svc_sendreply(...). */

bool_t svc_getargs(SVCXPRT *svc, xdrproc_t xdr_param, char *in); /* Appelée par la fonction de dispatch pour décoder les paramètres d'entrée d'une requête RPC. svc : pointeur sur la structure de type SVCXPRT initialisée. xdr_param : pointeur sur la fonction XDR de décodage des paramètres d'entrée. in : pointeur sur les paramètres d'entrée. Retourne 1 si succès, 0 si erreur. */

bool_t svc_sendreply(SVCXPRT *svc, xdrproc_t xdr_result, char *out); /* Appelée par la fonction de dispatch pour encoder et envoyer les résultats de la fonction appelée. svc : pointeur sur la structure de type SVCXPRT initialisée. xdr_result : pointeur sur la fonction XDR d'encodage des résultats. out : pointeur sur le buffer qui contiendra les résultats. Retourne 1 succès, 0 si erreur. */

struct sockaddr_in *svc_getcaller(SVCXPRT *svc); /* Pour récupérer l'adresse de l'appelant*/

void svcerr_decode(SVCXPRT *svc); /* Appelée par la fonction de dispatch si elle n'arrive pas à décoder les paramètres d'entrée. */

void svcerr_noproc(SVCXPRT *svc); /* Appelée par la fonction de dispatch si la fonction voulue par le client (celle dont il a indiqué le numéro) n'a pas été trouvée. */

void svc_destroy(SVCXPRT *svc); /* Destruction d'une structure de type SVCXPRT */

Couche basse, fonctions côté client

CLIENT *clnttcp_create(struct sockaddr_in *adr, unsigned long num_prog, unsigned long num_vers, int *sock, u_int send_buf_size, u_int recv_buf_size);
/* Création d'un client RCP pour un programme et une version en mode TCP. adr : pointeur sur la structure contenant les infos d'adressage du programme serveur. sock : pointeur sur le descripteur de socket local à utiliser ou RPC_ANYSOCK. send_buf_size : taille du buffer d'envoi. recv_buf_size : taille du buffer de réception. clnttcp_create(...) renvoie un pointeur sur l'objet CLIENT créé (structure contenant les caractéristiques de la liaison client/serveur ; voir sa définition dans rpc/clnt.h) ou NULL si erreur. */

CLIENT * clntudp_create (struct sockaddr_in *adr, unsigned long num_prog, unsigned long num_vers, struct timeval wait, int *sock);
CLIENT * clntudp_bufcreate (struct sockaddr_in *adr, unsigned long num_prog, unsigned long num_vers, struct timeval wait, int *sock, unsigned int send_buf_size, unsigned int recv_buf_size);
/* Même chose mais en mode UDP. wait : temps d'attente des résultats avant ré-appel.
struct timeval {int tv_sec; int tv_usec; }; /* tv_sec : temps en secondes. tv_usec : en microsecondes */

enum clnt_stat clnt_call(CLIENT *clnt, u_long proc_num, xdrproc_t xdr_param, char *in, xdrproc_t xdr_result, char *out, struct timeval tout);
/* Appel d'une fonction distante. clnt : pointeur sur une structure de type CLIENT initialisée. proc_num : numéro de la fonction à appeler. xdr_param : pointeur sur la fonction XDR d'encodage des paramètres d'entrée. in : pointeur sur les paramètres d'entrée. xdr_result : pointeur sur la fonction XDR de décodage des résultats. out : pointeur sur le buffer qui contiendra les résultats. tout : temps d'attente total des résultats. clnt_call (...) retourne une valeur de type enum clnt_stat (code d'erreur) castée en entier : valeur <> de 0 si problème (voir rpc/clnt.h pour le détail des codes d'erreurs de enum clnt_stat). */

Couche basse, fonctions côté client (suite)

void clnt_perrno(enum clnt_stat stat); /* Affiche un message sur la sortie erreur standard correspondant au code d'erreur indiqué par stat (le type enum clnt_stat est défini dans rpc/clnt.h). Cette fonction est utilisé après un appel de callrpc(...) ou de clnt_call(...) */

bool_t clnt_control(CLIENT *clnt, int req, char *info); /* Récupérer ou modifier des informations à propos du client. clnt : pointeur sur une structure de type CLIENT. req : type d'opération. info : pointeur sur l'information à lire ou à écrire. */

Exemples : valeur de req / type de info (pointeur sur une structure de type …).
CLSET_TIMEOUT / struct timeval* : mise à jour du timeout total
CLGET_TIMEOUT / struct timeval* : récupération du timeout total
CLGET_SERVER_ADDR / struct sockaddr_in* : récupération de l'adresse du serveur
CLSET_RETRY_TIMEOUT / struct timeval* : mise à jour du timeout de réessai (UDP uniquement)
CLGET_RETRY_TIMEOUT / struct timeval* : récupération du timeout de réessai (UDP uniquement)

void clnt_destroy (CLIENT *client); /* Destruction d'une structure de type CLIENT */

Fonctions XDR

Fonctions XDR. Nécessaires (côté client et serveur, en programmation couche haute et basse) pour le codage (C <=> XDR) des données (paramètres d'entrée, résultats).

bool_t xdr_int(XDR *xdrs, int *ip); /* Codage C <=> XDR d'un entier. ip pointeur sur l'entier en C. xdr : pointeur sur sa représentation XDR (structure). Retourne 1 si succès, 0 sinon. */

bool_t xdr_string(XDR *xdrs, char **sp, unsigned int maxsize); Codage C <=> XDR d'une chaîne. sp : pointeur sur pointeur sur la chaine en C. xdrs : pointeur sur sa représentation XDR (structure). maxsize : taille maximale de la chaîne. Retourne 1 si succès, 0 sinon.

etc. (voir la liste complète dans rpc/xdr.h)

rpcgen

rpcgen est un précompilateur de langage C (on peut parler de "compilateur RPC") permettant de générer automatiquement le code des talons (stub, skeleton) et les fonctions XDR de formatage des données à partir des structures de données (langage XDR) et des services (langage RPC) décrits dans un fichier .x (rappel : il ne peut y avoir qu'un seul paramètre d'entrée et un seul résultat (si plusieurs, il faut les regrouper dans une structure).

Fichier **geometrie.x**

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

struct rectangle { point p1; point p2; };

typedef int booleen;

struct coordonnees { int x1; int x2; int y1; int y2; };

struct param_inclus { struct rectangle rect; struct point p; };

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


rpcgen geometrie.x génère les fichiers suivants :

- geometrie.h : A inclure dans les fichiers serveur et client. Déclarations en C de constantes, structures et signatures de fonctions

- geometrie_xdr.c : A placer côté serveur et client. Contient les fonctions XDR d'encodage/décodage pour chaque type de données (structure) écrit dans le .x (encodage en format XDR avant transmission sur le réseau, décodage en format local à la réception).

- geometrie_clnt.c : Talon côté client (stub). Reçoit les appels localement et les transforme en appels distants

- geometrie_svc.c : Talon côté serveur (skeleton). Implémente le main(). Enregistrement des fonctions puis mise en écoute et aiguillage des requêtes RPC entrantes.

- geometrie_server.c (si RPCGEN est lancé avec l'option -a) : A placer côté serveur. Squelette des fonctions à implémenter.

- geometrie_client.c (si RPCGEN est lancé avec l'option –a) : A placer côté client. Programme client (contenant un main()) prêt à l'emploi. A compléter ou modifier pour correspondre aux besoins du programmeur.

geometrie.h

Définition des constantes, structures, signatures de fonctions. Fichier à inclure dans les fichiers serveur et client.

- Définition de constantes associant chaque nom de programme/version/fonction à son numéro (#define). Ces constantes permettent d'identifier le programme et les fonctions lors des appels distants.

Par exemple, avec :
```
rectangle CREER_RECTANGLE (coordonnees) = 2;
```
dans le .x, on trouve dans le .h :
```
#define CREER_RECTANGLE 2
```

- Pour chaque structure définie dans le .x, définition dans le .h de la même structure, du typedef associé et de la signature de la fonction XDR permettant le codage de ce type (C <=> XDR).

Par exemple, avec :
```
struct point { int x; int y; };
```
dans le .x, on aura dans le .h :
```
struct point {  int x;  int y; };
typedef struct point point;
bool_t xdr_point(XDR*, point*);
```

geometrie.h (suite)

- Pour chaque fonction définie dans le .x, on trouve dans le .h la signature de deux fonctions : la première (où le nom de fonction original est suffixé par _x (x est le numéro de version) est implémentée par le talon côté client, la seconde (suffixée par _x_svc) est à implémenter côté serveur (dans le _server.c).

Remarque : Pour les paramètres (paramètre d'entrée, résultat) un niveau de pointeur est rajouté par rapport au .x (les valeurs deviennent des adresses). De plus, chaque fonction contient maintenant un paramètre supplémentaire (de type CLIENT * pour la première et de type struct svc_req * pour la seconde).

Par exemple, avec :

```
version GEOM_VERSION_1 {
    ...
    int SURFACE_RECTANGLE (rectangle) = ...;
    ...
} = 1;
```

dans le .x, on aura dans le .h :

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


geometrie_xdr.c (fonctions XDR)

Contient une fonction d'encodage/décodage (conversion C <=> XDR) pour chaque type de donnée définie dans le .x

Par exemple, avec :

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

dans le .x, on aura dans le _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);
}
```

Remarque : il n'y a à chaque fois qu'une seule fonction pour l'encodage et le décodage. La structure XDR (définie dans rpc/xdr.h) contient un champ x_op qui détermine le sens du codage (valeurs XDR_ENCODE/0 ou XDR_DECODE/1)

geometrie_svc.c (talon côté serveur).

Contient la fonction main() qui enregistre le programme comme service RPC auprès du portmapper et lance la mise en écoute.

Extraits :

```
/* fonction de dispatch implémentée */
static void geom_prog_1(struct svc_req *rqstp, register SVCXPRT *transp){
    .....
}

int main(int argc, char **argv){

    register SVCXPRT *transp;
    pmap_unset(GEOM_PROG, GEOM_VERSION_1);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){...}
    if (!svc_register(transp, GEOM_PROG, GEOM_VERSION_1, geom_prog_1, IPPROTO_UDP)){...}

    transp=svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL){...}
    if (!svc_register(transp, GEOM_PROG, GEOM_VERSION_1, geom_prog_1, IPPROTO_TCP)){...}

    svc_run();
}
```


geometrie_clnt.c (talon côté client)

Implémente les fonctions surface_rectangle_1(…), creer_rectangle_1(…) et est_inclus_1(…). Rappel : fonctions appelables en local côté client et qui relaient l'appel vers le serveur (clnt_call(…)) après avoir encodé les paramètres.

Extrait : implémentation de la fonction surface_rectangle_1(…) :

```
static struct timeval TIMEOUT = {25, 0 };

int *surface_rectangle_1(rectangle *argp, CLIENT *clnt){
    static int clnt_res;
    memset ((char*)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, SURFACE_RECTANGLE, (xdrproc_t) xdr_rectangle, (caddr_t) argp,
(xdrproc_t) xdr_int, (caddr_t) &clnt_res, TIMEOUT) != RPC_SUCCESS){
        return NULL;
    }
    return (&clnt_res);
}

.....
```

Remarque : la variable devant contenir le résultat est déclarée en static (elle doit continuer à exister après l'appel de la fonction).

geometrie_server.c (squelette des fonctions côté serveur à implémenter)

Généré par rpcgen avec l'option -a. On y trouve le squelette des fonctions surface_rectangle_1_svc(...), creer_rectangle_1_svc(...) et est_inclus_1_svc(...) que le programmeur doit maintenant compléter.

Extrait : squelette de fonction surface_rectangle_1_svc (…) et implémentation par le programmeur :

```
int *surface_rectangle_1_svc (rectangle *rect, struct svc_req *req){
    static int result ;
    result=(rect -> p1.x – rect -> p2.x) * (rect -> p1.y – rect -> p2.y) ;
    if (result <0)
        result=-result;
    return &result ;
}

.....
```

Remarque : là encore, la variable devant contenir le résultat est déclarée en static.

Il reste à écrire la partie client

Un code par défaut (à compléter/modifier) peut être généré par rpcgen avec l'option –a.

Exemple : client.c

```
#include “geometrie.h“
#include <rpc/rpc.h>
#include <stdio.h>
#include <stdlib>

/* Fonction main(…) : le nom d’hôte du serveur est passé en paramètre */
int main(int argc, char *argv[]) {
    CLIENT *client;
    rectangle *rect;
    int *surface;

    client = clnt_create(argv[1], GEOM_PROG, GEOM_VERSION_1, “udp”);
    if (client==NULL){
        perror(“erreur creation client\n”);
        exit(1) ;
    }

    rect -> p1.x = 12;
    rect -> p1.y = 15;
    rect -> p2.x = 2;
    rect -> p2.y = 20;
    surface=surface_rectangle_1 (rect, client);
    printf("Rectangle de surface %d\n", *surface);
}
```


Compilation / édition de liens / exécution

Côté client et serveur :

```
gcc –c geometrie_xdr.c
```

Côté client :

```
gcc –c geometrie_clnt.c
gcc –c client.c
gcc –o client client.o geometrie_clnt.o,  geometrie_xdr.o
./client
```

Côté serveur :

```
gcc –c geometrie_svc.c
gcc –c geometrie_server.c
gcc –o serveur geometrie_svc.o geometrie_server.o geometrie_xdr.o
./serveur &
```


V.2. RMI

Généralités

- RMI est une API intégrée au JDK depuis la version 1.1
- Elle permet l'appel de méthodes appartenant à des objets Java s'exécutant dans la JVM d'une machine distante.
- L'appel peut aussi se faire sur la même machine (localhost).

Fonctionnement

- Le serveur dispose d'un registre de nom (annuaire) dans lequel il enregistre les objets accessibles à distance (= objets distants = objets distribués).
- Le client télécharge depuis ce registre un représentant local de l'objet distant : le stub. C'est avec cet objet que le client va interagir. Le stub a pour rôle de sérialiser les données à envoyer vers le serveur (paramètres de méthodes) et de désérialiser les réponses (exceptions éventuellement) reçues du serveur.
- Le stub côté client interagit avec le skeleton côté serveur. Outre le rôle d'écoute du réseau, le skeleton désérialise les paramètres, appelle la bonne méthode (aiguillage) puis sérialise la réponse et l'envoie vers le client.
- RMI utilise le protocole JRMP, lui-même basé sur les sockets et TCP/IP
- JRMP est spécifique à Java mais RMI peut aussi s'appuyer sur le protocole RMI-IIOP qui est interopérable avec Corba : l'interaction distante peut alors concerner des objets construits dans des langages différents

Sérialisation / désérialisation

Dans leur espace d'adressage (JVM), les données sont structurées de façon plus ou moins complexe (objets contenant d'autres objets …). Il faut pouvoir les aplatir en vue de leur transmission sur le réseau (passer d'un graphe d'objet à un flux d'octets). C'est la sérialisation (= marshalling = pliage). L'opération inverse s'appelle désérialisation (=unmarshalling = dépliage).

En java, tout objet qui doit être sérialisé doit implémenter l'interface **java.io.Serializable**

Les types primitifs ainsi que la plupart des objets de l'API Java (composants AWT …) sont sérialisables. Certains objets ne sont pas sérialisables : fichiers, threads (objets qui ont des références sur le système d'exploitation).

Autres problèmes à gérer (dans d'autres langages, comme par exemple le C) :

- Gestion des pointeurs (références d'objets…).
- Portabilité des encodages (big endian / little endian …).

Stub / skeleton

Les classes (compilées) stub et skeleton sont générés ensembles par la commande **rmic** à partir de la classe (compilée) proposant les méthodes distantes.

Les classes stub et skeleton ne sont plus indispensable (la première depuis Java 1.5, la seconde depuis la version 1.2).

Packages

java.rmi : contient les définitions d'interfaces (**Remote** …), de classes (**Naming**, **RMISecurityManager** …), et d'exceptions (**RemoteException** …) utiles côté client et côté serveur

java.rmi.server : interfaces, classes et exceptions plus spécialement utiles côté serveur : classe **UnicastRemoteObject**, **RMIClassLoader** ….

java.rmi.registry : relatif au rmiregistry : classe **LocateRegistry** …

java.rmi.dgc : relatif au garbage collector distribué

java.io : interface **Serializable** (à utiliser si passage de paramètre objet par copie)

Registre de nom (rmiregistry)

C'est l'annuaire des objets accessibles à distance (équivalent du portmapper des RPC). Il contient un ensemble d'associations URL - objet.

D'un côté, on y accède pour enregistrer un objet distant (**Naming.rebind(…)**), de l'autre pour obtenir une référence sur cet objet distant (**Naming.lookup(…)**)

Il doit être lancé sur la machine qui héberge ces objets accessibles à distance. Cela peut être fait en ligne de commande (**rmiregistry &** sous UNIX, **start rmiregistry** sous Windows) ou depuis le programme (**LocateRegistry.createRegistry(...)**).

Il utilise par défaut le port 1099 mais il est possible d'utiliser un autre port.

Il y a un rmiregistry unique par JVM.

Partie serveur (résumé)

La "classe distante" (classe de l'objet distant) doit obligatoirement hériter de la classe **UnicastRemoteObject** (laquelle contient les méthodes permettant de rendre un objet accessible à distance).

Toutes les méthodes qui sont accessibles à distance doivent se déclarer prêtes à lever une **RemoteException**.

Les signatures de ces méthodes doivent être décrites dans une interface ("interface distante"). Cette interface doit hériter de l'interface **Remote** (laquelle ne définit aucune méthode mais permet simplement d'indiquer que les méthodes seront accessibles à distance). La "classe distante" doit implémenter cette interface.

L'objet distant doit être enregistré dans le rmiregistry : **Naming.rebind(…)**.

Il faut lancer le rmiregistry : **rmiregistry &** (lancement en tâche de fond sous UNIX).

Partie client (résumé)

Avant java 1.5, le client devait disposer localement de la "classe stub" compilée (générée côté serveur grâce à la commande **rmic** à partir de la "classe distante" compilée). Ce n'est plus le cas. Par contre, il doit toujours disposer de l'"interface distante" compilée.

Il faut obtenir une référence sur l'objet distant. En pratique, cela se fait en téléchargeant depuis le rmiregistry (**Naming.lookup(…)**) un "objet stub".

Le client peut ensuite appeler les méthodes distantes.

Téléchargement de code à distance

• Les classes (compilées) nécessaires au client ou au serveur sont en général placées dès le départ à l'endroit désigné par leur variable d'environnement **classpath**. Si ce n'est pas le cas, RMI prévoit leur téléchargement dynamique depuis un serveur Web en utilisant le protocole HTTP (http://...) ou NFS (file://...). La désignation du "codebase" peut se faire en ligne de commande (**java –Djava.rmi.server.codebase=http:// ….**) ou depuis le programme (**System.setProperty("java.rmi.server.codebase", "http://…");**).

• Par exemple, en utilisant le principe du polymorphisme lors de l'appel de méthodes, on peut décider côté serveur de récupérer à distance les classes (compilées) des objets passés en paramètre (cela évite d'avoir beaucoup classes à gérer localement).

• Pour télécharger des classes, il est obligatoire d'utiliser un gestionnaire de sécurité : celui-ci peut être lancé en ligne de commande (**java –Djava.security.manager**) ou depuis le programme (**System.setSecurityManager(new RMISecurityManager();**).

• Le gestionnaire de sécurité vérifie alors que le chargement dynamique n'enfreint pas les règles de sécurité qui sont définies dans un fichier **.policy** (**java.policy** par défaut mais on peut en choisir un autre). Désignation en ligne de commande (**java –Djava.security.policy =monFichier.policy**) ou depuis le programme (**System.setProperty("java.security.policy", "monFichier.policy");**)

• Exemple d'un fichier **.policy** (très permissif) :

```
grant { permission java.security.AllPermission;   } ;
```

Le passage des paramètres et retour de méthode

Passage de paramètres par valeur à la méthode distante

• Les types primitifs sont passés par valeur
• Les objets non distants sont également passés par valeur. Ils doivent donc être sérialisables (leur classe doit implémenter l'interface **Serializable**).
• Le passage par valeur consiste en une copie de l'objet

Passage de paramètres par référence à la méthode distante

• Les objets distants (du point de vue du serveur cette fois-ci) sont passés par référence. Leur classe doit alors implémenter une "interface distante" (une interface qui hérite de Remote) et étendre **UnicastRemoteObject**.
• En pratique, pour permettre la manipulation d'un objet distant, c'est un stub de l'objet distant qui est sérialisé et envoyé depuis le client vers le serveur.

Le retour des méthodes et les exceptions

• La méthode retourne un résultat (sérialisé) au client ou bien une exception (elle aussi sérialisée)
• Il existe plusieurs types d'exceptions selon le problème rencontré :
- Erreurs liées à l'annuaire : **NotBoundException**, **AlreadyBoundException**,etc…
- Erreurs liées à l'appel et au retour de méthodes : **RemoteException**, **MarshalException**, **UnmarshalException** …
- Erreurs plus générales sur le serveur : **ServerException** …

Le ramasse miette réparti (Distributed Garbage Collector)

• Le but d'un Garbage Collector (GC) est de supprimer de la mémoire locale (JVM du programme), les objets qui ne sont plus référencés.

• C'est plus compliqué avec les objets distribués car il faut prendre en compte les références distantes, c'est-à-dire les références sur les stubs du côté des clients : nécessité d'un Distributed Garbage Collector (DGC)

• Fonctionnement d'un GC classique

Un compteur de référence est associé à chaque objet : ce compteur est incrémenté lorsqu'une référence est ajouté à l'objet et décrémenté lorsqu'une référence est perdue (associée à un autre objet ou à Null). Lorsque ce compteur atteint la valeur 0, l'objet est candidat à sa suppression par le GC.

• Fonctionnement d'un DGC

La gestion des références distantes est assurée par une coordination des GC de chaque machine. Nombre de références sur un objet distribué sur le serveur = nombre de stubs référencés sur les différents clients.

Problème : comment prévenir le DGC de la perte de référence sur un stub si une des machines est plantée ou inaccessible (problème sur le réseau).
Solution : utiliser une durée de bail (lease) : si au bout d'un certain temps, un objet distribué n'est plus sollicité par un stub distant, on décrémente son compteur.

Concurrence sur le serveur par des threads

• Lorsque plusieurs clients interagissent en même temps avec le serveur, ce dernier créé à priori un thread pour chacun ; la spécification RMI est cependant floue quant à leur gestion (FIFO, pool de threads …)

• Dans cet environnement multithread, il faut faire attention aux ressources partagées. Pour les rendre critiques, on peut utiliser le principe du verrouillage (mot clé **synchronized**). Toutefois, il ne faut pas abuser du verrouillage car cela ralentit les traitements (attente de verrou) et cela peut générer des interblocages.

Appel synchrone / appel asynchrone

• Un appel RMI est à priori synchrone (attente du retour de la méthode). Il devient asynchrone si la méthode distante retourne **void** (le client n'attend rien).

• Une autre façon de faire de l'asynchronisme est d'utiliser un thread spécifique pour l'appel distant (le client qui peut dans le même temps exécuter d'autres threads n'est ainsi pas bloqué).

Call-back sur le client

• Une communication asynchrone est également possible grâce au mécanisme du call-back : le serveur répond au client en appelant à son tour une méthode sur ce dernier. La classe contenant cette méthode doit alors être distante (hériter de **UnicastRemoteObject**, implémenter une "interface distante"). Il peut s'agir de la classe contenant la méthode d'appel initial ou bien d'une autre.

• Il faudra alors cette fois-ci faire attention aux ressources partagées sur le client

Mise en œuvre d'une communication simple avec RMI

I. Ecriture des classes et interfaces

1) "Interfaces distantes" : pour chaque objet distribué, définir une interface qui décrit les signatures des méthodes offertes à distance. Cette interface doit hériter de **Remote** et chaque méthode doit relever une **RemoteException**.
2) "Classes distantes" : implémenter chaque interface précitée par une classe qui hérite de **UnicastRemoteObject**.
3) "Classe serveur" (serveur): définir une classe contenant le programme principal (**main(…)**) qui enregistre les objets distribués (instances des classes précitées) dans le registre de noms et attend les appels distants.
4) "Classe client" (client) : définir une classe contenant le programme principal (**main(…)**) qui récupère les "objets stub" (représentations locales des objets distants) depuis le registre de noms puis appelle les méthodes distantes.

Remarque : dans l'exemple qui va suivre, il n'y a pas de classe pour les paramètres (car on n'utilise que des types simples)

II. Compilation des classes et interfaces, génération des stubs et skeletons

1) Utiliser la commande **javac** pour compiler les différentes classes et interfaces du du I. (on part du principe que tout est au départ sur le serveur).
2) Utiliser la commande **rmic** pour générer les classes stubs et skeletons (compilées) à partir des classes du I.2) compilées. (plus nécessaire depuis java 1.5)
3) Copier sur le client les "interfaces distantes" compilées, la classe client compilée et éventuellement (si avant java 1.5) les classes stubs compilées.

III. Lancement du serveur et du client

1) Côté serveur, lancer le **rmiregistry** (en tâche de fond avec **&**) à moins que cela ne soit fait depuis le code du serveur (**LocateRegistry.createRegistry(..)**).
2) Côté serveur et côté client, utiliser la commande **java** pour lancer l'un et l'autre (à partir de leurs classes respectives compilées). De chaque côté, certaines options peuvent être précisées, comme par exemple en cas de téléchargement de classes distantes l'utilisation d'un gestionnaire de sécurité (**-Djava.security.manager**), l'emplacement du fichier de sécurité (**-Djava.security.policy**) et l'emplacement des fichiers à télécharger sur un serveur Web (**-Djava.rmi.server.codebase=http://...**). Là encore, on peut aussi lancer ces options depuis le programme.

Téléchargement de classes/interfaces distantes (compilées),codebase et classpath

• Le téléchargement de classes distantes peut concerner le client et le serveur
• On peut par exemple télécharger la classe du client (client dynamique) et/ou les classes des objets distribués (serveur dynamique)

• Dans tous les cas, il faut faire attention au **classpath** (codebase local). Le client commence par rechercher les classes dans son classpath; si il ne les trouve pas, il essaye alors de les récupérer à partir de son **codebase** (à distance). Même principe sur le serveur : si on souhaite que celui-ci récupère des classes dynamiquement, il faut s'assurer avant de lancer **rmiregistry** que celles-ci ne sont pas dans son classpath, cet emplacement étant prioritaire sur le codebase.

Remarque : on peut détruire l'accès au **classpath** par la commande :
unset classpath.

I.1) Interface de l'objet distribué

• Elle doit contenir toutes les signatures des méthodes appelables à distance (pas celles qui sont appelables localement). Elle doit donc être **public** et ses méthodes doivent être **public** également.

• Elle est donc construite sur le serveur mais doit être connue du client : sa forme compilée est soit copiée "en dur" soit téléchargée dynamiquement sur le client.

• Elle doit hériter de l'interface **Remote**.

• L'invocation à distance pouvant échouer pour diverses raisons (crash du serveur, connexion refusée à l'hôte distant, objet n'existant plus, problème lors de la sérialisation ou de la désérialisation...) chaque méthode distante doit se déclarer prête à lever une **RemoteException**.

• Le plus souvent, on propose une interface par objet distribué mais il est possible de proposer plusieurs interfaces pour un même objet (définir différentes vues en fonction des clients).

Exemple : interface distante déclarant une seule méthode distante

```
import java.rmi.*;
public interface RessourceDistanteInterface extends Remote{
    public String chaineMajuscule(String ch) throws RemoteException;
}
```


I.2) Classe de l'objet distribué

• Elle doit implémenter l'interface précédente et donc compléter le corps des méthodes.

• Elle doit hériter de la classe **UnicastRemoteObject** (qui elle-même implémente l'interface **Remote**).

• Ses méthodes appelables à distances (comprenant le constructeur de la classe) doivent indiquer qu'elles peuvent lever une **RemoteException**.

• Elle pourrait contenir une méthode **main(…)** qui crée l'objet puis l'enregistre dans le rmiregistry. Cependant, dans notre exemple (et c'est ce qui est fait le plus souvent) on utilise une "classe serveur" dédiée qui gère (créé et enregistre dans le rmiregistry) l'ensemble des objets distribués.

Exemple : Classe de l'objet distribué

```
import java.rmi.*;
import java.rmi.server.*;

public class RessourceDistante extends UnicastRemoteObject implements
RessourceDistanteInterface {
    public RessourceDistante () throws RemoteException {
        super(); /* constructeur de la classe mère */
    }
    public String chaineMajuscule(String chaine) throws RemoteException {
        return chaine.toUpperCase();
    }
}
```


I.3) "Classe serveur"

C'est la classe qui contient la méthode **main(…)**. En plus d'un éventuel lancement du **rmiregistry**, de la mise en place d'un **SecurityManager** (si nécessaire), etc. elle est surtout dédiée à la création des objets distribués (instanciation des classes) et à leur enregistrement dans le rmiregistry.

A coder :

- Lancement du **rmiregistry** (on pourrait aussi le lancer en ligne de commande)
- Si téléchargement dynamique de classes, mise en place d'un **SecurityManager**, désignation de l'emplacement de ces classes (**codebase**), et éventuellement désignation du fichier **.policy** à utiliser (si autre que celui par défaut). Là encore, on pourrait aussi faire tout ça en ligne de commande.
- Création des objets distribués (**new(…)**) et enregistrement de ceux-ci dans le rmiregistry (**Naming.rebind(…)**).

Lancement du rmiregistry depuis le programme

Le rmiregistry doit toujours être lancé avant l'enregistrement des objets distants (ce qui est logique).

```
java.rmi.registry.Registry r = LocateRegistry.createRegistry(1099) ; /* lancement sur le
port par défaut */
```

Remarque : pour récupérer une référence sur un rmiregistry (local ou distant) :
LocateRegistry.getRegistry(…)

Mise en place d'un SecurityManager etc. depuis le programme

• Le téléchargement de classes distantes pose des problèmes de sécurité car le serveur va exécuter du code provenant d'une autre machine : la mise en place d'un **SecurityManager** est donc obligatoire dans ce cas (sans SecurityManager, le serveur doit trouver toutes les classes dont il aura besoin dans son classpath).
• Le SecurityManager applique la politique de sécurité définie dans un fichier **.policy** (**java.policy** par défaut).

Mise en place d'un SecurityManager (depuis le programme) :

```
if (System.getSecurityManager() == null)
    System.setSecurityManager(new RMISecurityManager());
```

Désignation d'un codebase (depuis le programme) :

```
System.setProperty ( "java.rmi.server.codebase", "http://localhost:8080/rmi/");
```

On peut forcer le téléchargement d'une classe distante en utilisant la méthode statique **loadClass(…)** de la classe **RMIClassLoader** :

```
Class RessourceDistante = RMIClassLoader.loadClass("http://localhost:8080/rmi/",
"RessourceDistante");
```

Désignation du fichier .policy à utiliser (depuis le programme) :

```
System.setProperty("java.security.policy", "file:/C:/java.policy");
```


Serveur statique

Les classes des objets distribuées sont présentes localement (dans le classpath)

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public class ServeurStatique {
    public static void main(String[] args) {
        try {
            java.rmi.registry.LocateRegistry.createRegistry(1099);
            RessourceDistante ressourceDistante = new RessourceDistante();
            String url="rmi://"+InetAddress.getLocalHost().getHostAddress()+"/ressource";
            Naming.rebind(url, ressourceDistante);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Serveur dynamique

Les classes des objets distribuées sont téléchargées depuis un site distant

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Properties;
import java.net.*;
public class ServeurDynamique {
    public static void main(String[] args) {
        try {
            java.rmi.registry.LocateRegistry.createRegistry(1099);
            if (System.getSecurityManager() == null)
                System.setSecurityManager(new RMISecurityManager());
            Properties p=System.getProperties();
            String url=p.getProperty("java.rmi.server.codebase");
            String urlLoc="rmi://"+InetAddress.getLocalHost().getHostAddress()+"/ressource";
            Class RessourceDistante = RMIClassLoader.loadClass(url, "RessourceDistante");
            Naming.rebind(urlLoc, (Remote) RessourceDistante.newInstance());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


I.4) "Classe client"

Comme pour le serveur, on peut télécharger la classe du client : on parle alors de client dynamique. Ce n'est pas le cas dans l'exemple suivant.

Exemple : Classe client (client statique)

```
import java.rmi.*;
public class Client {

   /* En argument, la chaîne à convertir en majuscules */
   public static void main(String[] args) {
      try {
         RessourceDistanteInterface ressDistInterface = (RessourceDistanteInterface)
Naming.lookup("rmi://134.59.56.62:1099/ressource");
         String premArgMaj=ressDistInterface.chaineMajuscule(args[0]);
         System.out.println("Premier Argument du main en majuscule:"+ premArgMaj) ;
      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```


II. Compilation des classes et interfaces, génération des amorces

Remarques : 1) Dans notre exemple, pas de téléchargement de code (ni côté client, ni côté serveur) 2) On part du principe que tout se trouve au départ côté serveur (à l'endroit désigné par la variable classpath)

1) Compilation des classes et interfaces
javac *.java

2) Génération des amorces (skeleton et stub). Inutile depuis java 1.2 resp. java 1.5.
rmic RessourceDistante
Si on veut générer les codes sources des amorces :
rmic –keepgenerated RessourceDistante

3) Placer les versions compilées du client (Client.class), de l'interface distante (RessourceDistanteInterface.class) et éventuellement (si java avant version 1.5) de la classe stub (RessourceDistante_stub.class) sur la machine cliente à l'endroit désigné par son classpath. Attention à laisser une copie de RessourceDistanceInterface.class sur le serveur.

III. Lancement du serveur et du client

1) Dans notre exemple, pas besoin de lancer le rmiregistry en ligne de commande car il est lancé depuis le programme.

2) Lancer le serveur en tâche de fond
java ServeurStatique &

3) Lancer le client
java Client

"Classes paramètres"

Dans notre exemple, nous avons utilisé des paramètres de types simples (passage par copie) et pas des objets construits, lesquels peuvent être passés par copie ou par référence.

- Si passage par copie, la classe du paramètre objet doit implémenter **Serializable** et sa version compilée doit être connue à la fois du client et du serveur (elle y est placée au départ ou téléchargée dynamiquement).

- Si passage par référence, la classe doit être considérée comme une "classe distante" (mais cette fois-ci du point de vue du serveur). Elle doit étendre **UnicastRemoteObject** et implémenter une "interface distante" (qu'il faut écrire). La version compilée de cette classe doit être placée sur le client (là encore, elle peut éventuellement être téléchargée) tandis que l'interface compilée doit être placée à la fois du client et du serveur.

Exemples en TP