Master 2 Informatique I2A

Master DVL Master ITVL

- Finance
- Histoire
- Géographie
- Informatique
- Mathématiques
- Sciences Pour l'Ingénieur
- Français Langue Étrangère
- Administration Économique et Sociale
- Diplôme d'accès aux études universitaires

Master mention Informatique

Parcours Informatique Avancé et Applications (I2A)

CTU
Centre de Télé-enseignement Universitaire
http://ctu.univ-fcomte.fr

Filière Informatique

VVIXECSD

Communication dans les systèmes distribués

Mr PHILIPPE - LAURENT
laurent.philippe@univ-fcomte.fr

Mr MAZOUZI - KAMEL
kamel.mazouzi@univ-fcomte.fr

Université de Franche-Comté

UBFC Université Bourgogne Franche-Comté

UNIVERSITÉ DE FRANCHE-COMTÉ

Module INFO-CSD

Laurent Philippe

Communication dans les Systèmes Distribués

Master Informatique Avancée et Applications
2ème Année

COURS

Centre de Télé-enseignement Universitaire

Centre de télé enseignement
Filière Informatique
Domaine Universitaire de la Bouloie
25030 Besançon Cedex (France)

Table des matières

Introduction **5**

1 Communication avec les sockets **7**
1.1 Définition des sockets . . . 7
1.2 Les modes de communication . . . 8
1.2.1 Le mode connecté . . . 8
1.2.2 Le mode non-connecté . . . 9
1.3 L'identification . . . 9
1.4 Création des sockets en Java . . . 10
1.4.1 ServerSocket . . . 10
1.4.2 Socket . . . 11
1.4.3 Fermeture . . . 12
1.5 Les streams . . . 12
1.5.1 Flux d'octets . . . 13
1.5.2 Flux de données . . . 14
1.5.3 Flux d'objets . . . 15
1.5.4 Autres fluxs . . . 16
1.6 Un premier exemple en mode connecté . . . 16
1.6.1 Serveur . . . 16
1.6.2 Client . . . 17
1.7 Délais d'attente et test de réception . . . 18
1.8 DatagramSocket . . . 19
1.9 Un exemple en mode non-connecté . . . 20
1.9.1 Émetteur . . . 20
1.9.2 Récepteur . . . 21
1.10 Les serveurs et les sockets . . . 22
1.11 Vers des communications plus élaborées . . . 26
1.11.1 Protocole de communication . . . 27
1.11.2 Les modes de communication . . . 27
1.11.3 La structuration des données . . . 29
1.11.4 Les architectures client-serveur . . . 30

2 Appel de procédure ou invocation distante **31**
2.1 Introduction . . . 31
2.2 Java RMI : Remote Method Invocation . . . 33
2.3 Cycle de développement RMI . . . 34
2.3.1 Définir l'interface de l'objet distant . . . 35
2.3.2 Définir l'objet distant . . . 36
2.3.3 Serveur : créer et publier l'objet distant . . . 37
2.3.4 Client : utiliser l'objet à distance . . . 39
2.3.5 Restrictions du développement RMI . . . 41
2.4 L'appel de méthode à distance . . . 41
2.5 Le chargement dynamique de classes . . . 42
2.5.1 Principe de téléchargement dynamique de bytecode . . . 43
2.5.2 Sécurité en RMI . . . 45
2.6 L'activation d'objets distants . . . 46

3 Les bus de messages **47**
3.1 Introduction . . . 47
3.2 JMS . . . 49
3.2.1 Principes généraux . . . 49
3.2.2 Mise en place de la communication . . . 51
3.2.3 Programmation JMS . . . 53
3.2.4 Messages JMS . . . 57
3.2.5 Modes de session . . . 59
3.2.6 Cas d'erreur . . . 60
3.3 Synthèse . . . 60
3.4 Webographie . . . 61

4 Les services Web **63**
4.1 Généralités . . . 63
4.2 Architecture des services Web SOAP . . . 64
4.2.1 SOAP . . . 65
4.2.2 WSDL . . . 68
4.2.3 UDDI . . . 72
4.3 Développement de services Web SOAP . . . 72
4.3.1 Développer le service . . . 73
4.3.2 Consommer le service : le client . . . 76
4.4 Web Services REST . . . 79
4.4.1 Les URI . . . 79
4.4.2 Principales caractéristiques des web services REST . . . 80
4.4.3 Identification des ressources par les URI . . . 80
4.4.4 Ressource, opération et méthode HTTP . . . 81
4.4.5 Représentation des données . . . 81
4.5 Développement des web services REST . . . 83

4.5.1 Environnement de développement JAX-RS . . . 84
4.5.2 Développement JAX-RS . . . 85
4.5.3 L'annotation `@Path` . . . 86
4.5.4 Paramètres de requêtes . . . 88
4.5.5 Réponses . . . 92
4.5.6 UriBuilder . . . 93
4.5.7 Déploiement . . . 94
4.5.8 Développement du client . . . 96

5 Conclusion **99**
5.1 Synthèse sur les interfaces de communication . . . 99
5.2 Pour aller plus loin . . . 101

A Premiers pas avec des sockets **107**
A.1 Définition des sockets . . . 107
A.2 Les modes de communication . . . 108
A.2.1 Le mode connecté . . . 108
A.2.2 Le mode non-connecté . . . 109
A.3 L'identification . . . 109
A.4 Création et fermeture . . . 110
A.4.1 Création en mode non-connecté . . . 110
A.4.2 Création en mode connecté . . . 111
A.4.3 Fermeture des connexions et des sockets . . . 113
A.5 La communication . . . 113
A.5.1 Le mode non-connecté . . . 114
A.5.2 Le mode connecté . . . 116
A.5.3 Échange des données . . . 118

B Les sockets des experts **121**
B.1 Généralités . . . 121
B.2 Création des sockets . . . 122
B.3 Association d'une adresse . . . 123
B.4 Les adresses : l'API classique . . . 125
B.4.1 Adresses `AF_INET` . . . 125
B.4.2 Les numéros Internet . . . 126
B.4.3 Les noms de machine . . . 127
B.4.4 Les ports . . . 128
B.4.5 Les services . . . 130
B.4.6 Synthèse . . . 131
B.5 Établir une connexion . . . 133
B.6 La communication . . . 134
B.6.1 Données échangées . . . 134
B.6.2 Le mode non-connecté . . . 136

B.6.3 Le mode connecté 137
B.7 Autres fonctions 138
B.8 Sockets non bloquantes 138
B.9 Multiplexage des appels socket 139

Introduction

L'enseignement classique de la programmation vise généralement la conception d'un programme dont l'exécution se fait sur une seule machine, sous la forme d'un processus [1]. L'ensemble des éléments développés, fonction principale, variables locales ou globales, fonctions ou objets sert alors uniquement à ce programme. Lorsque le programme doit interagir avec un autre programme, éventuellement sur une autre machine, on parle alors d'application communicante ou distribuée. Le but de ce cours est de vous former à la programmation de la communication entre programmes.

Les supports utilisables pour la programmation des communications sont nombreux, mots-clefs ou package d'un langage, bibliothèques ou serveurs spécifiques, et offrent un niveau de fonctionnalité variable : depuis le niveau le plus bas, où les développements se font en langage de bas niveau (assembleur ou langage C), jusqu'à des niveaux d'abstraction élevés où les communications sont rendues transparentes à l'application par un environnement de programmation. Le niveau de fonctionnalité étant choisi en fonction des besoins des applications, il est nécessaire de maîtriser plusieurs types de supports pour avoir une bonne compréhension de la communication au sein d'une application. Ce cours aborde donc différentes solutions pour la mise en œuvre de la communication au sein d'un programme, sur la base des différentes technologies actuellement les plus répandues. Nous commençons par la programmation de sockets, support de communication le plus répandu dans le monde puisqu'il constitue la base de l'accès à Internet. Cette première expérience nous permet de mettre en évidence les principales problématiques liées à la communication. Nous étudions ensuite d'autres paradigmes de communication tels que les appels de procédures ou invocations distants à travers les *Remote Method Invocation* de Java, les bus de messages à travers JMS et les services Web. À noter que les choix des technologies qui illustrent les concepts ont été faits pour n'utiliser dans le cours que le langage de programmation Java et les approches les plus simples, pas forcément les plus utilisées.

Les pré-requis de ce cours sont une connaissance de la programmation en Java. Les concepts abordés sont assez techniques, basés sur l'utilisation d'API (*Application Programming Interface*), mais rarement abstraits. Ils peuvent être acquis sans difficulté à condition de s'appliquer à bien comprendre les programmes de correction des exercices. Ceci vous permettra de développer une compétence dans l'utilisation des fonctions de communication

1. La notion de processus est classique dans dans les systèmes linux, elle peut prendre l'appellation d'application sous d'autres systèmes. Dans tous les cas elle constitue l'entité du systèmes d'exploitation qui exécute un programme.

et dans la mise en œuvre d'applications communiquantes.

Le document de cours ne donne en général pas les instructions pour le développement, la compilation et l'exécution des programmes. Ces aspects sont abordés dans le document d'exercices.

Chapitre 1

Communication avec les sockets

Les sockets sont une API (Application Programming Interface, interface de programmation), c'est-à-dire un ensemble de fonctions normalisées, permettant de programmer des communications entre programme. Cette API a été initialement conçue à l'université de Berkeley en 1982 pour le système Unix BSD 4.1, pour permettre l'utilisation du protocole TCP/IP [1]. Cette interface a été étendue par la suite pour servir d'accès à de nombreux autres protocoles. Parmi ceux-ci, nous pouvons citer IPv6, Bluetooth, NFC, etc. L'API des sockets permet l'utilisation au niveau applicatif des fonctionnalités des protocoles TCP et UDP. Elle est assurément la plus utilisée pour les communications et elle dispose d'implémentations dans la plupart des langages. Dans ce cours nous abordons la programmation socket avec le langage Java.

1.1 Définition des sockets

Littéralement, une socket est une «prise» de communication depuis un processus vers l'extérieur : c'est l'entité qui permet au programme de «se brancher» pour communiquer avec le monde. Un programme qui souhaite communiquer commence donc par créer sa propre socket. L'entité socket est créée au cœur du système d'exploitation et du protocole.

Pour communiquer avec un autre programme possédant lui même une socket, les programmes doivent mettre leurs sockets en rapport. Pour cela une adresse est associée à chaque socket qui est utilisée pour l'identifier depuis l'extérieur. Les socket peuvent ensuite être utilisées pour envoyer et recevoir des données ; une même socket sert aussi bien à envoyer qu'à recevoir. Lorsque la communication est finie, le programme ferme la socket.

Un même programme peut créer plusieurs sockets pour communiquer avec plusieurs programmes. Au sein d'un programme, une socket est identifiée, en Java, par un objet. Cet objet sera toujours utilisé dans les fonctions d'accès aux sockets de manière à spécifier de quelle socket il s'agit.

1. pour de plus amples explications, vous pouvez revoir votre cours de licence ou consulter l'encyclopédie en ligne Wikipedia à l'adresse : http : //fr.wikipedia.org/wiki/Internet

La création d'une socket dépend du mode de communication avec lequel nous souhaitons utiliser la socket.

1.2 Les modes de communication

Pour envoyer des données avec une socket on utilise une adresse de communication, pour préciser qui est le destinataire. Lorsque des échanges répétés ont lieu, il est possible d'établir des connexions entre les programmes. Lorsqu'une connexion est établie, le programme qui envoie un message utilise la connexion et n'a plus besoin de préciser le destinataire puisque celle-ci conduit directement à l'autre programme. Ainsi, deux modes de communication sont définis : le mode connecté et le mode non-connecté. Une analogie simple permet de comprendre les deux modes :

— Le mode connecté est celui que vous utilisez avec le téléphone. Vous composez le numéro de votre correspondant et, une fois la connexion établie, les échanges se font dans les deux sens et uniquement entre les postes connectés.

— Le mode non-connecté est celui que vous utilisez avec le courrier postal ou électronique. Depuis une boîte aux lettre, vous pouvez émettre à destination de n'importe quelle boîte au lettres mais sur chacun des messages, vous devez préciser le destinataire.

1.2.1 Le mode connecté

Dans le **mode connecté**, deux sockets établissent une relation durable qui permet de ne pas préciser la socket destinatrice à chaque envoi de données, une fois la connexion réalisée. Au delà de l'analogie avec le téléphone, le mode opératoire est différent. Le modèle de connexion des sockets suppose une relation asymétrique où un des programmes prend le rôle du **serveur** et l'autre le rôle du **client**. Le serveur est vu comme un programme rendant des services à différents clients. Son exécution sous-entend une attente de demande de connexion de la part d'un client, l'ouverture de cette connexion, le traitement des messages échangés (service rendu), la fermeture de sa connexion avant de se mettre en attente du prochain client.

Pour sa part, le client se contente de demander une connexion avec le serveur, de lui envoyer les messages à traiter puis de fermer sa connexion. Pour mettre en place ce schéma, la notion de **socket de connexion** est introduite. En fait, le serveur utilise une socket spécifique uniquement pour recevoir des demandes de connexions, donc appelée socket de connexion. Ensuite, à chaque réception d'une demande de connexion de la part d'un client sur cette socket, le système crée chez le serveur une nouvelle socket, ou socket de communication, comme cela est illustré par la figure A.1. C'est cette socket qui est utilisée pour communiquer directement avec le client. Ce modèle permet au serveur de traiter simultanément plusieurs clients et de recevoir les requêtes de clients différents sur des sockets différentes.

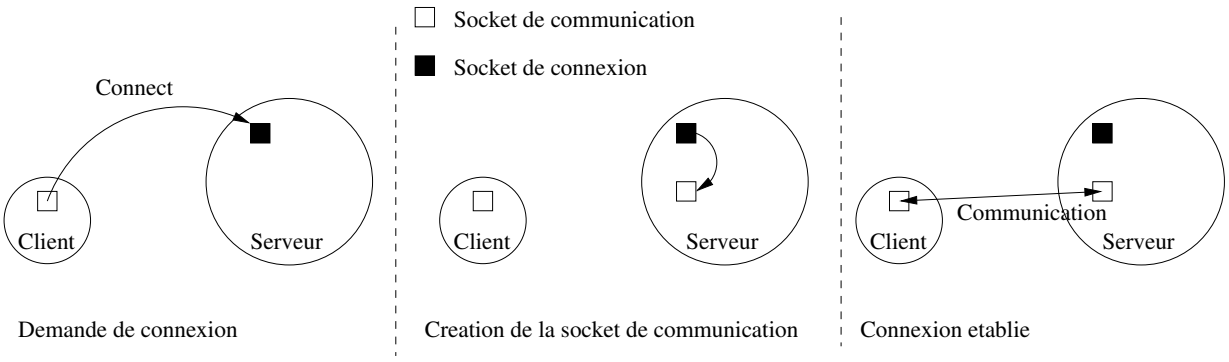


FIGURE 1.1 – Établissement d'une connexion

1.2.2 Le mode non-connecté

L'autre mode de communication utilisable avec les socket est le **mode non-connecté**. Comme son nom l'indique, il ne suppose pas l'établissement d'une connexion avant la mise en place d'une communication. Dans ce cas, le destinataire de la socket est précisé à chaque envoi. Il est possible d'établir une analogie entre le mode non-connecté et le courrier, qu'il soit postal ou électronique, pour lequel nous précisons systématiquement le destinataire. La socket peut être vue comme une boîte aux lettres par laquelle nous envoyons et recevons des données.

1.3 L'identification

Du point de vue du programme les sockets sont identifiées de deux manières : en externe par son adresse et en interne par un objet qui donne accès à la socket.

En externe, une socket est donc identifiée de manière unique à partir du **nom de la machine** (en fait son adresse IP) sur laquelle elle se trouve et d'un numéro local à la machine, appelé **numéro de port**. C'est la combinaison de ces deux informations qui garantit que deux sockets ne possèdent pas la même identification. Ainsi deux sockets créées sur la même machine possèdent forcément des numéros de port différents et deux socket créées sur des machines différentes peuvent avoir le même numéro de port.

Le nom de la machine est le nom internet de la machine, celui qui s'affiche généralement lors de la saisie du login ou qui peut être obtenu à l'aide de la commande `hostname` sur un système Unix/Linux. Si la communication dépasse les limites du réseau local, il est nécessaire d'adjoindre au nom les identifications de domaine. Par exemple, si je souhaite envoyer un message à une machine appelée `smith` dans le domaine `univ-fcomte.fr` alors que je ne m'y trouve pas, je dois donner le nom `smith.univ-fcomte.fr`, sinon le nom simple est généralement suffisant. Il est également possible pour une communication locale, si les programmes sont sur la même machine, d'utiliser le nom `localhost`.

La valeur donnée au numéro de port est généralement fixée par le programme à la création de la socket. Cette valeur doit être comprise entre 1024 et 65536, elle est donc

codée sur 16 bits et n'est pas signée. Attention certaines valeur de numéro de port [2] sont réservés à des applications.

Pour illustrer la programmation des sockets nous avons choisi le langage Java. Ce langage permet une programmation facile des sockets. Nous avons mis en annexe deux chapitres d'une version précédente du cours où nous présentons la programmation des sockets en C pour les étudiants qui souhaitent étendre leurs compétences pour un programmation bas niveau. A noter tout de même que le langage Java offre l'avantage de masquer l'hétérogénéité du codage des données, ce que ne fait pas le langage C.

1.4 Création des sockets en Java

La première chose à faire pour mettre en place une communication est donc de créer une socket dans notre programme. Pour créer une socket, nous devons d'abord savoir quel mode de communication, connecté ou non connecté, nous souhaitons utiliser avec cette socket, car le mode de communication est défini à la création d'une socket et ne peut être changé par la suite. Notons tout de même qu'un même programme peut faire des communications connectées et des communications non connectées à condition de posséder deux sockets, chacune créée avec un mode différent.

En Java, une socket est forcément un objet. Il existe donc une classe qui permet la création des sockets en implémentant un constructeur et la manipulation à l'aide des méthodes associées à la classe. Dans une communication en mode connecté, le client et le serveur utilisent des types de sockets différents. Les serveurs utilisent la classe `ServerSocket` alors que les clients utilisent la classe `Socket`. La communication en mode non connecté utilise, elle, l'objet `DatagramSocket`. Toutes les communications avec les sockets nécessitent l'importation des packages `java.net`.

Nous commençons, dans la suite, par traiter les sockets en mode connecté. Si nous utilisons le mode connecté, nous devons, en plus des informations sur la machine et le port, savoir quel est le rôle du programme : client ou serveur, car nous avons vu qu'ils ne travaillaient pas de la même façon.

1.4.1 ServerSocket

La classe `ServerSocket` est utilisée uniquement par le serveur. Elle correspond à une socket de réception des demandes de connexions. Les constructeurs de cette classe sont les suivantes :

```
public ServerSocket(int port) throws IOException
/* permet la creation d'une socket de serveur en fixant
* le numero de port qui lui est associe
*/
```

2. Sur les systèmes Unix/Linux les numéros réservés le sont généralement enregistrés dans le fichiers `/etc/services`

```
public ServerSocket(int port, int backlog) throws IOException
/* permet la creation d'une socket de serveur en fixant
* le numero de port et la taille de la file d'attente de
* demande de connexions qui lui est associee
*/

public ServerSocket(int port, int backlog, InetAddress addr)
throws IOException
/* permet la creation d'une socket de serveur en fixant
* le numero de port, la taille de la file d'attente de
* demande de connexions et l'adresse IP qui lui sont associes.
* Cette fonction presente un interet lorsqu'une machine dispose
* de deux adresses IP.
*/
```

A noter qu'il n'est pas nécessaire ici de passer de nom de machine car la socket est crée sur la machine où s'exécute le programme. Cette fonction n'est pas bloquante.

Une fois la socket créée, le programme attend des demandes de connexion à l'aide de la fonction :

```
public Socket accept() throws IOException
```

Cette fonction est bloquante, c'est-à-dire qu'elle ne rend pas la main au programme tant qu'elle n'a pas reçu une demande de connexion. Lorsqu'une demande de connexion arrive, cette fonction retourne une socket de type socket de communication qui est connectée avec le client.

Pour ces fonctions, une exception de type `IOException` est levée en cas d'erreur.

1.4.2 Socket

La classe `Socket` est utilisée pour la communication. Les différents constructeurs de la classe sont les suivants :

```
public Socket(String host, int port)
throws UnknwnHostException, IOException
/* permet la creation d'une socket connectee avec le serveur
* dont le nom et le numero de port sont donnes en parametre
*/

public Socket(InetAddress addr, int port) throws IOException
/* permet la creation d'une socket connectee avec le serveur dont
* l'adresse IP et le numero de port sont donnes en parametre
*/

public Socket(String host, int port, InetAddress localAddr,
int localPort) throws UnknwnHostException, IOException
/* permet la creation d'une socket connectee avec le serveur dont
* le nom et le numero de port sont donnes en parametre et
* donne l'adresse et le numero de port a la socket locale
*/
```

```
public Socket(InetAddress addr, int port, InetAddress localAddr,
int localPort) throws IOException
/* permet la creation d'une socket connectee avec le serveur dont
* l'adresse IP et le numero de port sont donnes en parametre et
* donne l'adresse et le numero de port a la socket locale
*/
```

Certaines méthodes de la classe `Socket` permettent d'obtenir des informations sur la connexion :

```
public InetAddress getInetAddress()
/* obtenir l'adresse IP distante */

public InetAddress getLocalAddress()
/* obtenir l'adresse IP locale */

public int getPort()
/* obtenir le numero de port distant */

public int getLocalPort()
/* obtenir le numero de port local */
```

1.4.3 Fermeture

Lorsqu'une socket n'est plus utilisée il est important de penser à la fermer. Par défaut cela est fait par la machine virtuelle Java, à la fin de l'exécution du programme mais le nombre de sockets ouvertes par un programme est limité. Un programme qui créé donc régulièrement des sockets sans les fermer finit par avoir une erreur quand cette limite est atteinte.

La méthode `close()` permet de fermer la connexion et de libérer les ressources qui y sont associées.

1.5 Les streams

Les communications en Java utilisent la notion de *stream* pour échanger des données entre programmes. Les *streams*, ou flux, sont des canaux de communication implémentés sous la forme d'objets encapsulant un flux d'octets. Généralement, les streams sont employés en Java pour la lecture ou l'écriture depuis un terminal, un fichier ou le réseau. Dans un *stream*, les données sont écrites ou lues les unes à la suite des autres. Les données ne sont pas accédées en fonction d'une position absolue, comme c'est le cas pour un tableau, mais séquentiellement, dans l'ordre du flux et relativement à la position courante. Les données sont lues dans l'ordre où elles ont été écrites mais les limites ne sont pas marquées, c'est-à-dire qu'il n'y a pas de marque de début et fin de donnée.

Les classes et interfaces de manipulation de flux sont regroupées dans le paquetage `java.io` et font partie de la spécification Java depuis la version 1.1. Il est donc nécessaire

d'importer ce package pour travailler avec les sockets. Dans la programmation socket, nous utilisons principalement deux types de stream : pour manipuler des octets et pour manipuler des objets.

A chaque socket est associé des stream. Une fois créé l'objet socket est utilisé pour obtenir les streams qui lui sont associés. On obtient les *streams* d'une socket à l'aide des méthodes :

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Toutes les méthodes sur les flux peuvent renvoyer `IOException`. Il est donc nécessaire de prévoir un traitement de cette exception à chaque utilisation des méthodes.

1.5.1 Flux d'octets

L'échange de données le plus basique repose sur des flux d'octets (`byte`), donc de données brutes. Dans le principe un émetteur envoie un buffeur de données, un tableau de `byte`, qui est reçu par le récepteur.

Toutes les classes qui manipulent des flots d'octets héritent de l'une des deux classes : `InputStream` et `OutputStream`.

Les méthodes d'`InputStream` sont :

```
int read()
/* retourne l'octet suivant dans le flot ou -1 si la fin du
* flot est atteinte
*/

int read(byte[] b)
/* lit dans le flot au plus b.length octets qui sont places
* dans le tableau b. Les octets sont stockes dans le tableau
* dans l'ordre de lecture, de la premiere case vers la derniere.
* Le nombre d'octets effectivement lus est retourne ou -1, si la
* fin du flot est atteinte.
*/

int read(byte[] b, int offset, int len)
/* est equivalente a la methode precedente si ce n'est que les
* octets sont stockes a partir de la case d'indice offset et
* qu'au plus len octets sont lus
*/
```

Les méthodes d'`OutputStream` sont :

```
void write(int b)
/* permet d'ecrire l'octet de poids faible de b dans le flot */

void write(byte[] b)
/* ecrit dans le flot les b.length octets stockes dans le tableau.
* L'ordre d'ecriture des octets dans le flot est celui du tableau
*/
```

```
void write(byte[] b, int offset, int len)
/* a le meme comportement que la methode precedente, mais ecrit
* len octets de b en partant de la case d'indice offset
*/
```

Dans un échange classique entre programmes avec des sockets, l'émetteur écrit des données sur un `OutputStream`, avec la fonction `write` tandis que le récepteur les lit sur son `InputStream` avec la fonction `read`. La fonction `read` est bloquante ce qui fait que le récepteur ne passera à l'instruction suivante qu'une fois des données reçues. En retour la fonction `read` donne le nombre d'octets reçus.

Les flux permettent un accès séquentiel aux données. Parfois, il est utile de modifier ce comportement. La méthode `skip(long n)` permet de consommer les `n` premiers octets d'un flux. Le nombre d'octets effectivement consommés est retourné par cette méthode.

A noter que les données ainsi échangées le sont sous leur forme brute, non typées. Ce mode convient bien par exemple à l'échange de données telles que des tableaux de types de base ou de texte. Pour les types élaborés, on utilise plutôt des formateurs de données. Nous abordons dans la suite l'échange d'objets sur des sockets mais il existe d'autres type de formateurs tels que les `PrintStream` ou `BufferReader` par exemple.

Pour finir, la méthode `close()` permet de libérer les ressources associées au flux. A noter donc qu'il est possible de fermer l'un des deux flux sur une socket tout en conservant l'autre.

1.5.2 Flux de données

Il existe plusieurs classes qui permettent l'utilisation des données de manière plus ou moins évoluée à partir des flux liés aux sockets. Parmi les solutions possibles, les classes `DataInputStream` et `DataOutputStream` permettent de lire/écrire des données de type de base sur les socket.

La création de ces objets se fait à partir du stream associé :

```
OutputStream os = s.getOutputStream();
InputStream is = s.getInputStream();

DataInputStream dis = new DataInputStream(is);
DataOutputStream dos = new DataOutputStream(os);
```

Les objet `DataStream` permettent l'envoi et la réception de tableau d'octet mais ajoute l'écriture et la lecture de données ayant un type de base. Voici quelques exemples de fonction :

```
// DataInputStream
readBoolean();
readChar();
readInt();
...
```

```
// DataOutputStream
writeBoolean();
writeChar();
writeInt();
...
```

Attention, en java les chaînes de caractères ne sont pas de simples suites d'octets. Pour échanger des chaînes de caractères sur une socket il est alors nécessaire d'utiliser des fonctions spécifiques. Les formatteurs de flux de données `DataInputStream` et `DataOutputStream` permettent ainsi un envoi et une lecture directe de chaîne de caractères sous la forme de tableaux d'octets à partir des méthodes `writeBytes` et `read`, mais la méthode `writeBytes` effectue un envoi octet par octet qui peut engendrer une réception en plusieurs fois. Dans ce cas il peut être préférable d'utiliser les méthodes `writeUTF` et `readUTF` qui vont préserver l'encodage de la chaîne et faire des envois/réceptions complets. A noter que les classes `InputStreamReader` et `OutputStreamWriter` permettent également de travailler directement avec des chaînes de caractères.

Pour gérer des envois de lignes, leur utilisation peut-être complétée par les classes `BufferedReader` et `BufferedWriter` qui sont initialisées de la manière suivante :

```
BufferedReader in = new BufferedReader(new InputStreamReader(is));

BufferedWriter out = new BufferedWriter(new OutputStreamWriter(os));
```

1.5.3 Flux d'objets

Les classes de lecture et écriture sur des flux peuvent être utilisée pour réaliser des échanges d'objets sur les sockets sur la base respectivement des classes `ObjectInputStream` ou `ObjectOutputStream`. Ces classes sont initialisées à partir des flux utilisés pour la transmission. L'exemple suivant montre la création de flux d'objets en lecture et en écriture à partir de flux initiaux.

```
OutputStream os = s.getOutputStream();
InputStream is = s.getInputStream();

ObjectInputStream ois = new ObjectInputStream(is);
ObjectOutputStream oos = new ObjectOutputStream(os);
```

Les objets écrits ou lus à partir des méthodes des classes `ObjectInputStream` ou `ObjectOutputStream` implémente l'interface `java.io.Serializable`. Cette interface suppose que l'objet ne contient pas de donnée dépendante du site sur lequel il se trouve, par exemple un descripteur de fichier. En effet, cette interface permet de sérialiser un objet (l'encoder sous la forme d'une suite d'octets) pour l'écrire sur un flux, ce qui permet soit de l'enregistrer dans un fichier, soit de l'envoyer à travers une socket. Si cet objet possède des données dépendante de la machine sur laquelle il s'exécute il ne pourra retrouver le même contexte sur une autre machine. La plupart des classes courantes sont potentiellement sérialisables, il est cependant nécessaire qu'elles implémentent cette interface pour

l'être vraiment. Pour cela, la déclaration suivante est suffisante :

```
class Truc implements Serializable {
  /* declaration de la classe */
}
```

Une fois les classes de flux initialisées, il est possible d'envoyer et de recevoir des objets en utilisant les méthodes suivantes sur ces classes :

```
void writeObject(Object o)
/* permet d'ecrire l'objet o sur le flux */

Object readObject()
/* permet de lire un objet sur le flux */
```

Lors de l'écriture, nous ne rencontrons pas de problème de type dans la mesure où la classe `Object` est la classe mère de tous les classes en Java. Elle surtype donc tous les objets Java et n'importe quel objet peut être utilisé avec le type `Object`. Dans notre cas, cela signifie que n'importe quel objet peut être passé en paramètre de la méthode `writeObject`.

À l'opposé, pour la lecture nous recevons un objet de type générique. Cet objet n'est pas sensé pouvoir être de n'importe quel type et il sera nécessaire de réaliser un transtypage pour affecter l'objet reçu à l'instance dans laquelle nous souhaitons mémoriser l'objet.

Attention, quelques petites précautions sont à prendre pour les flux d'objets :

— Il n'est pas possible de créer deux flux d'objet de même type sur une socket,

— Pour que le flux d'objet en entrée soit créé, il faut que le flux d'objets en sortie soit déjà créé à l'autre bout de la socket, sinon la fonction *getInputStream* bloque en attente de ce second flux. Cette propriété peut conduire à un blocage (deadlock) si vous faites la création des *ObjectInputStream* en premier de chaque côté de la socket. Il vaut donc prendre l'habitude de créer les *ObjectOutputStream* en premier.

— La fonction `readObject()` retourne des objets `final` donc qui ne peuvent pas être modifiés. On ne peut donc pas recevoir deux fois de suite dans le même objet.

1.5.4 Autres fluxs

La classe `PrintWriter` est une autre solution pour utiliser les flux de manière plus simple. Vous trouverez des informations plus complètes à l'adresse https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/io/PrintWriter.html.

1.6 Un premier exemple en mode connecté

L'exemple suivant montre l'utilisation de base des sockets en Java, en mode connecté.

1.6.1 Serveur

Listing 1.1 – Code d'un serveur socket

```
1  import java.net.* ;
2  import java.io.* ;
3
4  public class ServeurConnect {
5
6    public static void main(String[] args) {
7      ServerSocket srv;
8
9      try {
10       srv = new ServerSocket(5555) ;
11       Socket connexion = srv.accept();
12
13       OutputStream os = connexion.getOutputStream();
14
15       byte[] tabloServ = {1,2,3,4,5};
16
17       os.write(tabloServ);
18
19       connexion.close();
20     }
21   } catch(IOException ex) {
22     /* traitement de l'exception */
23   }
24 }
```

Ce listing donne le code d'un serveur simple qui crée une socket de connexion (ligne 10), puis se met en attente d'une demande de connexion (ligne11). Une fois qu'il obtient une socket connectée (`connexion`), il en prend le stream de sortie sur lequel il écrit un tableau de 5 chiffres. Ici la fonction `accept` est bloquante ce qui signifie que le programme de dépassera pas cette instruction tant qu'il n'aura pas reçu de demande de connexion. A la première demande reçue, une socket sera crée et connectée avec le programme demandeur. Cette socket pourra aussi bien être utilisée pour envoyer que pour recevoir. Il n'y a donc pas de lien entre le rôle de serveur et celui d'émetteur et/ou récepteur. C'est la logique de l'application qui détermine ce second rôle.

1.6.2 Client

```
1  import java.net.* ;
2  import java.io.* ;
3
4  public class ClientConnect {
5
6    public static void main(String[] args) {
7      Socket sock;
8      String machineServeur = "smith.univ-fcomte.fr";
9
10     try {
```

```
11          sock = new Socket(machineServeur, 5555);
12          InputStream is = sock.getInputStream();
13
14          byte[] tabloCli = new byte[5];
15          int recu = is.read(tabloCli);
16
17          sock.close();
18        }
19      } catch(IOException ex) {
20        /* traitement de l'exception */
21      }
22    }
23  }
```

Dans le listing précédent un programme client crée une socket connectée avec un serveur à l'adresse `smith.univ-fcomte.fr:5555` (ligne 11). A partir de la socket il obtient le stream de lecture (ligne 12) et reçoit le message envoyé (ligne 15).

Attention, comme vous pouvez le constater sur le listing, il est nécessaire d'allouer la structure de donnée pour recevoir les données avant de recevoir dans le cas des `OutputStream`. A la réception les données sont enregistrées dans la structure de données. Les données reçues ne dépassent jamais la limite du tableau. Si la taille du tableau alloué pour la réception n'est pas suffisante pour recevoir toutes les données reçues sur la socket, le surplus est laissé en attente. A l'inverse, la fonction `read` n'attend pas d'avoir remplit entièrement la structure de données de réception pour sortir. Dès qu'elle reçoit des données elle sort et donne le nombre d'octets qu'elle a reçu. Ainsi il est possible de prévoir un buffeur de grande taille pour recevoir un plus petit ensemble de données. Ceci est utile, par exemple, si la taille des données attendues n'est pas connue mais que seulement une borne maximale de la taille l'est. La taille des données peut-être connue à la réception par la valeur retournée par la fonction `read`.

Les données de flux ne sont pas structurées. C'est-à-dire qu'aucun marqueur n'est positionné entre les différents envois et il n'est pas possible de déterminer la taille d'un envoi à partir du buffeur de réception. Ainsi, il est possible de recevoir deux envois successifs de cinquante octets aussi bien sous la forme d'un buffeur de cent octets que de dix buffeurs de dix octets. C'est à l'application de réaliser la délimitation des données si elle en a besoin. Par contre, comme le mode connecté utilise le protocole TCP, nous avons la garantie que l'ordre des données est préservé à la réception.

Pour les formateurs d'objets `ObjectInputStream` et `ObjectOutputStream`, l'allocation de l'objet est faite par le stream.

1.7 Délais d'attente et test de réception

Comme nous l'avons précisé précédemment les fonctions `accept` et `read` sont bloquantes, ce qui signifie que le programme peut resté bloqué pendant un temps indéterminé s'il ne reçoit pas, soit une demande de connexion pour `accept`, soit des données pour `read`.

Si ce comportement ne convient pas au besoin du programme il est possible d'associer un délais d'attente à la socket quelque soit son type. La fonction utilisée est :

```
public void setSoTimeout(int timeout) throws SocketException
/* permet d'associer un timeout a la socket
* La valeur est donnee en millisecondes
*/
```

Lorsqu'un délais d'attente est positionné pour la socket, si elle n'a pas reçu de demande de connexion ou de données dans le temps du délais, l'attente est interrompue et la fonction lève une exception de type `SocketTimeoutException`. Le délais doit être appliqué avant que l'appel à la fonction bloquante soit réalisé.

La valeur de délais donnée doit être strictement positive. Si une valeur de 0 est donnée alors le délais est infini, donc la socket redevient bloquante.

Un effet de bord de cette fonction est la possibilité de tester s'il y a des données sur la socket, sans pour autant se mettre en attente s'il y en a pas. Cela peut-être faire de la manière suivante :

```
sock.setSoTimeout(1);
int recu = is.read(buffer);
if (recu == 0) {
  // aucune donnee sur la socket
} else {
  // traitement des donnees recues
}
```

L'ensemble des fonctions vues précédemment peuvent ensuite être combinées pour écrire des programmes répondant aux besoins des applications communiquantes. Dans les exercices vous pourrez ainsi faire la réponse à une requête, recevoir plusieurs messages de suite, etc.

1.8 DatagramSocket

Les sockets en mode non-connecté sont implémentées en Java par les objets `DatagramSocket`. Elles permettent l'envoi et la réception de données sous la forme de `DatagramPacket` dont la représentation est assez proche des paquets IP. Le mode d'utilisation est plus bas niveau que les sockets en mode connecté dans le sens où le programmeur doit préciser l'adresse IP du destinataire et le numéro de port pour chaque envoi.

Une `DatagramSocket` est créée à partir des ses constructeurs :

```
public DatagramSocket() throws SocketException
/* permet la creation d'une socket utilisant le protocole UDP
*/

public DatagramSocket(int port) throws SocketException
/* permet la creation d'une socket a laquelle est associe un numero
* de port. Cette fonction regroupe donc les fonctions socket et
* bind.
```

```
*/
```

Dans ce cas la socket n'est pas connectée. Il faut utiliser pour envoyer des données l'adresse complète de la socket : adresse IP de la machine et numéro de port.

La socket est ensuite utilisée pour échanger des `DatagramPacket`. Cet objet contient à la fois les données et l'adresse de destination. Les constructeurs sont :

```
public DatagramPaquet(byte[] buf, int length)
/* permet la creation d'un DatagramPacket a partir des donnees
* passees en parametre. Le destinataire de ce paquet n'est pas
* fixe.
*/

public DatagramPaquet(byte[] buf, int length, InetAddress address,
int port)
/* permet la creation d'un DatagramPacket a partir des donnees
* passees en parametre en fixant le destinataire du paquet
*/
```

Chaque paquet `DatagramPacket` possède donc des attributs `address` et `port` qui permettent de connaître le destinataire d'un message lors de l'envoi et l'émetteur lors de la réception. Les méthodes de la classe permettent de gérer les attributs associés à un paquet :

```
public InetAddress getAddress()
/* permet d'obtenir l'adresse contenue dans un DatagramPacket */

public int getPort()
/* permet d'obtenir le port contenu dans un DatagramPacket */

public byte[] getData()
/* permet d'obtenir les donnees */

public int getLength()
/* permet d'obtenir la taille des donnees */
```

L'envoi et la réception se fait, une fois le datagram constitué, à partir de la classe `Socket` à l'aide des méthodes :

```
public void send(DatagramPacket p) thows IOException
public void receive(DatagramPacket p) thows IOException
```

1.9 Un exemple en mode non-connecté

L'exemple donne une utilisation simple des sockets en mode non-connecté. Ici nous avons utilisé le nom `localhost` pour permettre l'exécution des programmes sur une même machine.

1.9.1 Émetteur

```
1  import java.net.* ;
2  import java.io.* ;
3
4  public class SenderNonConnect {
5
6      public static void main(String[] args) {
7
8          byte[] buf = {1,2,3,4,5};
9
10         try {
11
12             DatagramSocket socketSend = new DatagramSocket();
13             InetAddress address = InetAddress.getByName("localhost");
14             DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 5555);
15
16             socketSend.send(packet);
17
18         } catch ( UnknownHostException uhe ) { /* Exception */ }
19         } catch ( IOException ioe) { /* Exception */ }
20     }
21 }
```

L'émetteur crée une socket de type `DatagramSocket` (ligne 12) et prépare un `DatagramPacket` (ligne 14) dans lequel il donne l'adresse de la machine sur laquelle se trouve la socket destinatrice et le port de cette socket. A noter qu'il n'est pas besoin ici de préciser un numéro de port puisqu'aucune réception de message n'est faite.

1.9.2 Récepteur

```
1   import java.net.* ;
2   import java.io.* ;
3
4   public class ReceiverNonConnect {
5
6     public static void main(String[] args) {
7
8       try {
9         DatagramSocket socketRecv = new DatagramSocket(5555);
10
11        byte[] buf = new byte[256];
12        DatagramPacket packet = new DatagramPacket(buf, buf.length);
13        socketRecv.receive(packet);
14
15        System.out.println("Recu = " + packet.getLength() + " " + buf[2]);
16
17      } catch ( IOException ioe) { /* Exception */ }
18    }
19  }
```

Le récepteur crée une socket avec un numéro de port (ligne 9). Il alloue un buffeur pour la réception des données, qu'il passe en paramètre de la fonction de réception (ligne 13). Après la réception le buffeur contient les données reçues. Contrairement à ce qui se passe en mode connecté, le buffeur doit ici être de taille suffisante pour recevoir l'ensemble des données. Les données qui ne sont pas reçues dans le paquet sont perdues et il n'est donc pas possible de recevoir un `DatagramPacket` émis en deux `DatagramPacket`.

1.10 Les serveurs et les sockets

Nous avons vu jusqu'à maintenant comment mettre en place un échange simple entre deux programmes. Les communications sont souvent utilisées pour mettre en place des serveurs. Ces programmes sont accessibles à travers une socket et leur rôle est de répondre aux requêtes qui leur sont envoyées. Dans ce cas le schéma de communication côté serveur est un peu plus compliqué que dans le cas de l'échange simple puisque le serveur ne sert pas qu'un seul client mais un ensemble de clients.

La structure générale de ces serveur est la suivante :

Listing 1.2 – Code d'un serveur socket

```
1  import java.net.* ;
2  import java.io.* ;
3
4  public class ServeurConnect {
5
6    public static void main(String[] args) {
7      ServerSocket srv;
8      byte[] bufferRecv = new byre[BUF_SIZE];
9
10     try {
11
12       srv = new ServerSocket(5555) ;
13
14       while(true){
15         Socket connexion = srv.accept();
16
17         OutputStream os = connexion.getOutputStream();
18         int recu = os.read(bufferReq);
19         // Traitement de la requete et ecriture
20         // de la reponse dans bufferResp
21         os.write(bufferResp);
22
23         connexion.close();
24       }
25
26     } catch(IOException ex) { // traitement de l'exception }
27   }
28 }
```

Dans cette structure le serveur crée une socket pour les demandes de connexion, puis entre dans une boucle infinie où il attend des demandes de connexion de la part de ces clients. Lorsqu'une demande arrive le serveur reçoit la requête (ligne), le serveur traite ensuite la requête avant de transmettre la réponse au client.

Suivant le schéma des échanges entre le client et le serveur, cette structure simple peut ne pas être suffisante. Ainsi, si le traitement d'une requête comprends plusieurs échanges il est possible d'avoir un traitement plus complet au niveau du serveur. Le code précédent modifie la boucle infinie de réception des demandes de connexions pour permettre de recevoir plusieurs requêtes successives pour un même client :

Listing 1.3 – Code d'un serveur socket

```
1
2   while(true){
3     Socket connexion = srv.accept();
4
5     while ( !fin ) {
6
7       OutputStream os = connexion.getOutputStream();
8       InputStream is = connexion.getInputStream();
9
10      int recu = is.read(bufferReq);
11      // Traitement de la requete et ecriture
12      // de la reponse dans bufferResp
13      os.write(bufferResp);
14
15    }
16
17    connexion.close();
18  }
```

Dans l'exemple précédent, le serveur ne traite qu'un seul client à la fois. Si le traitement d'un seul client prend du temps, par exemple dans le cas où le client échange plusieurs requêtes, alors les autres clients peuvent avoir à attendre avant de voir leur requête être traitées par le serveur. Une solution à ce problème est d'utiliser des threads pour gérer les connexions. A la connexion d'un client un nouveau thread est créé qui va prendre en charge le client. Le thread principal reste lui disponible pour recevoir d'autres requêtes. De cette manière les clients sont traités en parallèle et un client ne bloque pas les autres.

Avec les threads deux possibilités peuvent être implémentées, soit avec une création dynamique des threads, à chaque demande de connexion, soit avec un pool de threads.

Le code suivant donne un exemple de la structure d'un serveur avec création dynamique de thread :

Listing 1.4 – Code d'un serveur socket multi-threadé avec création dynamique

```
1   class TraitReq extends Thread {
2
3     OutputStream os;
4     InputStream is;
```

```
5
6       TraitReq(Socket conn){
7
8         os = conn.getOutputStream();
9         is = conn.getInputStream();
10      }
11
12      void run() {
13
14        byte[] bufferReq = new byte[BUF_SIZE];
15        byte[] bufferResp = new byte[BUF_SIZE];
16
17        while ( !fin ){
18
19          int recu = is.read(bufferReq);
20          // Traitement de la requete et ecriture
21          // de la reponse dans bufferResp
22          os.write(bufferResp);
23        }
24
25        conn.close();
26      }
27    }
28
29    public class ServeurConnect {
30
31      ServerSocket srv;
32
33      public static void main(String[] args) {
34
35        try {
36
37          srv = new ServerSocket(5555) ;
38
39          while(true){
40            Socket connexion = srv.accept();
41
42            TraiteReq tr = new TraitReq( connexion );
43            tr.start();
44          }
45
46        } catch(IOException ex) { // traitement de l'exception }
47      }
48    }
```

Les problèmes de cette structure sont la possibilité de créer une infinité de threads et la nécessité de créer et détruire un thread à chaque nouvelle connexion. En effet, tant que le serveur reçoit des demandes de connexion, il crée de nouveaux threads. Si le nombre connexions est grand le serveur va créer beaucoup de thread qui, d'une part occuperont les ressources système, et d'autre part seront en concurrence pour leur exécution, donc

se ralentiront mutuellement. Cela ne sera pas sensible dans le cas de serveurs puissants mais peut l'être dans un environnement sensible. Dans le cas de serveurs très chargés il peut donc être intéressant d'utiliser une structure où le nombre de threads concurrents est limité, avec un pool de thread.

Le code suivant donne un exemple sommaire de l'architecture d'un serveur multi-threadé, avec un pool de threads :

Listing 1.5 – Code d'un serveur socket avec pool de threads

```
1  public class ServerPool extends Thread {
2
3    static Object lock;
4    static Socket conn
5
6    void run() {
7
8      Socket soc;
9
10     byte[] bufferReq = new byte[BUF_SIZE];
11     byte[] bufferResp = new byte[BUF_SIZE];
12
13     while(true) {
14
15       synchronized( lock ) {
16
17         lock.wait():
18         soc = conn;
19       }
20
21       OutputStream os = soc.getOutputStream();
22       InputStream is = soc.getInputStream();
23
24       while ( !fin ){
25
26         int recu = is.read(bufferReq);
27         // Traitement de la requete et ecriture
28         // de la reponse dans bufferResp
29         os.write(bufferResp);
30       }
31       soc.close();
32     }
33   }
34
35   public static void main(String[] args) {
36
37     lock = new Object();
38
39     try {
40
41       ServerSocket srv = new ServerSocket(5555) ;
42
```

```
43          ServerPool[] threadPool = new ServerPool[NB_THREADS];
44          for (ServerPool sp : threadPool){
45
46            sp = new ServerPool();
47            sp.start();
48          }
49
50          while(true) {
51
52            synchronized( lock ){
53              conn = srv.accpet();
54              lock.notify();
55            }
56          }
57          ...
58
59        } catch(IOException ex) { // traitement de l'exception }
60      }
61    }
```

Dans ce cas le serveur commence par créer un ensemble de threads qui se mettent toutes en attente de recevoir une demande de connexion et se met en attente d'une réception. Lorsqu'une demande arrive elle est prise en charge par un des threads, réveillé grâce à une synchronisation wait/notify. Les autres restent en attente tant qu'une connexion ne leur est pas attribuée.

Dans cette structure de code le nombre de requêtes traitées simultanément est limité par le nombre de threads. A noter que de toutes façons le parallélisme entre les threads est limité par la capacité du processeur. Le parallélisme réel dépend donc du nombre de cœeurs de la machine qui exécute le programme et du type de requêtes car, si une requête effectue des entrées/sorties une autre peut-être traitée pendant ce temps. Au final définir un nombre de threads supérieur au nombre de cœeurs sera suffisant pour bénéficier de la capacité de la machine la plupart du temps.

Cette dernière structure peut être programmée de manière plus simple et plus efficace en utilisant la notion de `ThreadPool`, directement prise en charge par le langage Java mais qui dépasse le contenu de ce cours.

1.11 Vers des communications plus élaborées

Ce chapitre a été l'occasion de vous familiariser avec la programmation de communications : nous avons vu comment réaliser des échanges et comment les synchroniser. Dans cette partie, grâce à l'expérience pratique acquise avec les sockets, nous revenons sur les problématiques générales de la communication et donnons quelques définitions.

1.11.1 Protocole de communication

La communication entre les programmes sert avant tout à les faire travailler ensemble pour rendre un service qui ne peut l'être par un seul programme. Or, à partir du moment où il y a plusieurs entités qui réalisent un travail en commun, ou qui partagent une ressource, il faut établir des règles de collaboration ou de partage. Des programmes différents ne savent en effet rien les uns des autres : leur état, leurs données, etc. Pour pouvoir travailler ensemble ils ont donc besoin d'échanger des informations et de se synchroniser, ce qui est fait en communiquant. Il faut alors définir des règles de communication, ce qu'on appelle un protocole.

Le protocole définit les règles d'échange : quelles données sont échangées et à quel moment, quelle primitive de communication est utilisée, qui prend l'initiative de communiquer, quelle est la signification des messages, etc. Il existe de très nombreux protocoles, certains sont normalisés et largement répandus, comme les protocoles HTTP (Hyper Text Transfert Protocol), NTP (Network Time Protocol) ou SMTP (Simple Mail Transfert Protocol). Il est également possible de définir un protocole spécifique à une application, par exemple en définissant les structures de données (les objets en Java) qui seront échangées, les constantes partagées, les codes d'erreur, etc. Regrouper ces déclarations dans un fichier spécifique permet d'avoir une vue d'ensemble sur le protocole de communication entre les programmes. Ce fichier peut, par exemple, servir de "mode d'emploi" pour l'accès à un serveur dont le client ne connaît pas l'implémentation : la structure des messages, que nous évoquons en 1.11.3, est suffisante pour que le client envoie des requêtes et reçoive des réponses du client.

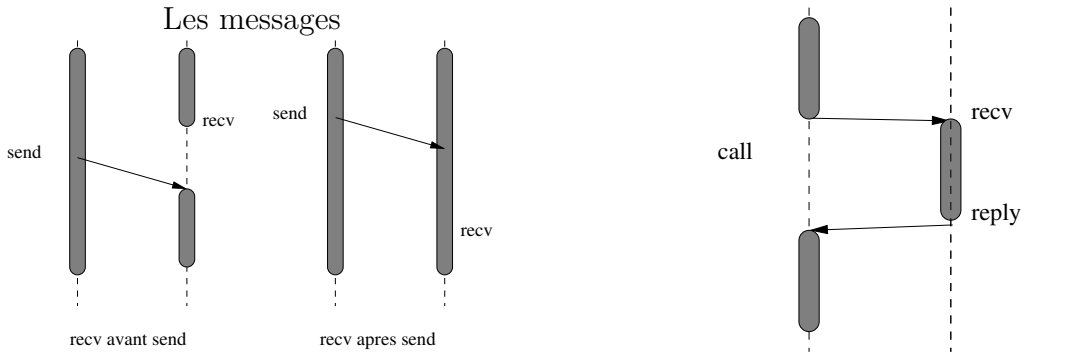
1.11.2 Les modes de communication

Nous avons vu que l'échange des données peut se faire de différentes manières en fonction des nécessités de synchronisation entre l'émetteur et le récepteur et le modèle de coordination ou de concurrence utilisé. On définit généralement deux modes principaux de communication : le mode asynchrone et le mode synchrone.

Les modes synchrone et asynchrone

Dans l'échange asynchrone, les données sont simplement envoyées de l'émetteur vers le récepteur, sans que le premier attende qu'elles soient reçues ou traitées. L'émetteur n'attend pas de réponse. Il ne connaît pas l'état du récepteur et de sa consommation des données. Comme nous le voyons sur la figure 1.2a, cet échange peut prendre deux formes suivant que la réception a lieu avant ou après l'envoi. C'est le mode utilisé lorsqu'on communique avec des sockets en faisant `send` et `recv`.

L'échange synchrone consiste en un aller-retour entre les programmes. Le message est envoyé au destinataire puis le client attend le retour de sa requête, comme cela est illustré sur la figure 1.2b. Puisque dans ce schéma l'appelant est bloqué pendant la durée du traitement et qu'il peut être pénalisé en cas d'appels multiples, ce mode de communication



(a) Schéma d'échange asynchrone (b) Schéma d'échange synchrone

FIGURE 1.2 – Synchronisation des échanges

est souvent couplé avec des possibilités de concurrence semblables à ce que nous avons vu avec les différentes structures de serveurs multi-threadés.

Pour ce mode, on parle aussi de rendez vous ou de point de synchronisation, car lorsque le récepteur reçoit le message, il sait où en est l'émetteur dans le déroulement de son programme, et lorsque l'émetteur reçoit la réponse, il sait où en est le récepteur. Ce qui n'est pas possible en mode asynchrone dans lequel la seule information possible est que la réception a lieu après l'émission.

Il est possible de réaliser un appel synchrone avec les sockets en faisant successivement appel à la fonction `send` puis à `recv` chez l'appelant et l'inverse chez le récepteur.

Le mode synchrone est le mode généralement utilisé en communication entre un client et un serveur car il supporte bien le schéma requête-réponse qui va avec la demande d'exécution d'un service et l'attente de la réponse, comme nous le voyons avec les appels de procédure distants ou les services Web. De plus, il permet la mise en place d'une gestion des erreurs dans le déroulement du traitement du côté du serveur. Ce mode sert parfois de base à un mode d'interrogation transactionnel, comme nous le voyons dans avec les bus de messages. Le traitement d'une requête est alors réalisé de manière atomique, c'est-à-dire soit pas du tout, soit complètement mais jamais stoppé en cours d'exécution.

Diffusion, diffusion restreinte

Les modes qui ont été présentés précédemment n'impliquent que deux programmes et ne sont pas suffisants pour tous les types de communication que nous pouvons avoir à mettre en œuvre dans les applications distribuées. La diffusion (ou *broadcast*) permet d'envoyer à tous les destinataires le même message. Il est possible d'avoir recourt à un envoi asynchrone multiple pour réaliser une diffusion mais en général un traitement spécifique permet d'optimiser les envois. Le protocole IP, s'appuyant sur les possibilités physiques des réseaux qu'il gère, permet par exemple d'optimiser les diffusions. Il est souvent plus avantageux de s'en remettre à une couche ou un service de communication pour réaliser la diffusion d'un message que de réaliser soi-même la distribution.

L'envoi d'un même message à un groupe de destinataires qui ne sont pas forcément réunis sur un réseau local ou sur une classe d'adresses est appelé diffusion restreinte (ou *multicast*). Elle recouvre des problématiques similaires à celles de la diffusion complète à travers la recherche de chemins communs aux messages et l'optimisation des ressources de communication. Le protocole IPv6 fournit un support au multicast. Ce type de communication peut être utilisé en support à la diffusion de données multimédia, dans la *video on demand* (VOD) par exemple. Nous voyons, au chapitre 3, avec les bus de messages, des paradigmes de diffusion restreinte basés sur des canaux (ou topic) et diffusé par un serveur de communication.

1.11.3 La structuration des données

Entre un émetteur et un récepteur ou un client et un serveur, les données peuvent être transmises sous différentes formes.

Les flux

Les flux sont le mode d'échange utilisé par les sockets. Les données sont échangées sous forme continue sans structuration, c'est-à-dire sans délimitation explicite entre les différentes émissions. Par exemple, avec des sockets, deux envois peuvent être reçus en une seule fois sans que le récepteur puisse le savoir. La couche réseau sous-jacente peut garantir l'ordre de réception pour que le contenu du buffeur respecte l'ordre de l'émission (TCP) ou pas (UDP) auquel cas la gestion de l'ordre des données doit être gérée explicitement au niveau des programmes.

Cette structuration des données est assez proche de la couche réseau IP (celle-ci ne se contente en général que d'ajouter un découpage en trames) ce qui permet d'offrir des performances de communication élevées car le système gère peu de choses.

Ce mode de transfert est bien adapté à des flux continus de type audio ou vidéo. Par contre, pour les autres types de communication, il implique que l'utilisateur prennent en charge le travail de structuration. Le travail n'est pas très complexe à réaliser : il peut reposer sur la mise en place d'un en-tête de message qui contient la taille du message transmis pour déterminer les frontières de messages, comme dans certaines requêtes HTTP, ou sur des balises comme CRLF inclues au message.

Les messages

À l'inverse des flux, les messages structurent les données échangées : le récepteur d'un message est assuré qu'il s'agit bien d'un seul envoi. Dans les messages les données sont également structurées en types de données. Cette structuration peut-être explicite, c'est-à-dire qu'une structure de données accompagne le message en donnant sa taille et éventuellement d'autres informations telles que la priorité, des droits d'accès, un type associé, etc., ou implicite, par convention entre les programmes qui communiquent. Les messages sont le mode d'échange utilisé par les bus de messages (par exemple JMS).

En général, ce mode de transfert est plus lourd que le modèle flux car il associe les informations de structuration des données mais il offre meilleur confort de programmation à travers la structuration, la possibilité de typer les message, la gestion de l'hétérogénéité, etc.

Pour des besoins spécifiques, il est possible de combiner les différents modes de transfert au sein d'une même application, en fonction des besoins. Par exemple, il est possible d'utiliser des messages pour le contrôle d'un flux multimédia (début, fin, réservation, etc), et un flux pour les données.

De plus, si la structuration en messages n'est prise en charge par le support de communication, elle peut être réalisée directement dans le programme, ou regroupée dans une bibliothèque supplémentaire, au dessus du support utilisé. C'est ce qui est fait par toutes les bibliothèques/supports de communication que nous voyons par la suite qui utilisent des sockets comme support de communication de base mais offre des fonctionnalités supplémentaires pour gérer ou faciliter la programmation des échanges.

1.11.4 Les architectures client-serveur

L'architecture client-serveur est très utilisée par les applications communicantes. Elle consiste à associer un rôle aux programmes qui communiquent, comme cela a été défini avec les sockets. La logique de l'architecture est que le serveur met à disposition du client un service : le client demande l'accès au service en envoyant une requête au serveur et celui-ci répond avec une réponse. Il s'agit donc d'un mode synchrone. Cette architecture est en particulier très utilisée sur Internet et nous avons déjà évoqué les protocoles (par exemple HTTP) qui définissent les structures des requêtes et des réponses. Cette structuration vient combler le manque de typage des flux de communication des sockets.

Dans ce contexte, l'utilisation de communications en mode synchrone répond à une logique proche de l'appel de procédure, à la différence près que l'exécution du code de la procédure est faite à distance (c.f. figure 1.2b). Les paramètres d'appel et de retour de la procédure peuvent facilement être copiés dans des messages, à la place d'être copiés sur la pile. On parle alors d'appel de procédures à distance, ou invocations distantes pour les langages objets, ce que nous voyons au chapitre 2.

Pour conclure ce chapitre, nous notons que l'interface (API) socket est celle qui sert d'accès à Internet et dans la plupart des communications. Elle est cependant de bas niveau et laisse au programmeur la responsabilité de la mise en œuvre d'un grand nombre de fonctionnalités : codage des données, synchronisation entre les processus, sûrete de fonctionnement, etc. Pour cette raison, de nombreuses propositions d'API plus évoluées, qui ajoutent des fonctionalités, existent. Nous voyons dans la suite du cours, les appels de procédures à distance avec RMI, au chapitre 2, mais aussi avec les services web SOAP et REST, au chapitre 4, et les bus de communication, au chapitre 3. Toutes ces propositions s'appuient sur des sockets mais elles sont transparentes pour le programmeur.

Chapitre 2

Appel de procédure ou invocation distante

2.1 Introduction

La communication à l'aide des sockets, bien que courante et efficace, est considérée comme une forme de communication de bas niveau entre des programmes. En effet, les sockets n'autorisent que l'échange de flux non structurés d'octets entre programmes communiquants et proposent une API réduite. Pour mettre en place une communication client-serveur, les programmeurs sont donc obligés de prendre en charge toutes les traitements liés à la communication (mise en forme des données dans un flux, gestion des erreurs, etc.[1]) alors que celles-ci pourraient être regroupées dans les bibliothèques ou services de communication.

Les modèles d'invocation distante reposent sur le constat que, lors d'une communication client/serveur, les messages échangés prennent souvent la forme d'un échange synchrone de requête/réponse. La requête a pour objectif d'appeler une fonctionnalité particulière du serveur avec les données nécessaires au traitement de la requête. Le client doit préciser dans le message le type de requête (par exemple dans l'exercice du serveur de calcul le client précise l'opération à effectuer) et les paramètres associés. En retour le serveur envoie au client les données de la réponse et les codes d'erreur, qu'il encapsule également dans un message. Cet échange est donc similaire à ce qui se passe lors d'un appel de procédure ou de fonction. Or un appel de procédure est un modèle de programmation simple à gérer, généralement maîtrisé par les programmeurs et qui permet de structurer les programmes. Le reproduire pour mettre en œuvre les communications permet donc de faciliter le travail du programmeur.

Le travail de génération des messages d'appel (requête) ou de retour (réponse) est toujours à peu près le même. Les modèles d'appel de procédure à distance (Remote Procedure

1. Dans ce contexte le langage Java est une exception car il gère, avec les objets de flux (tels que `ObjectInputStream`,`ObjectOutputStream`), un premier niveau de mise en forme des données. Ce qui n'est pas forcément le cas dans les autres langages.

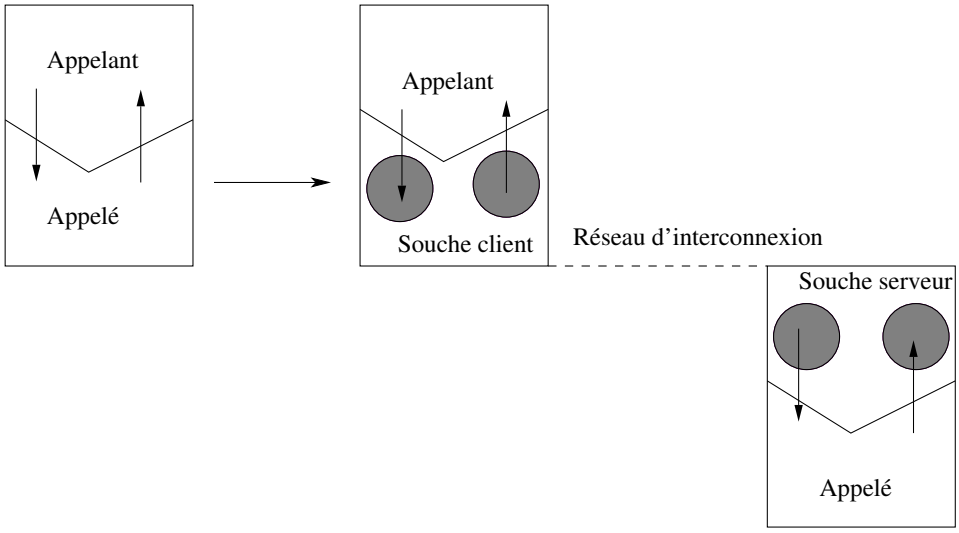


FIGURE 2.1 – Schéma d'invocation distance

Call ou RPC) pour les langages procéduraux et d'invocation de méthode distante (Remote Method Invocation ou RMI) pour les langages à orientés objets cherchent donc à automatiser ce travail. A partir d'un prototype de fonction un compilateur génère le code (souche ou *stub*) de création et d'envoi/réception des messages, comme cela est montré sur la figure 2.1. On parle d'**assemblage** ou *marshalling* des paramètres, c'est l'arrangement d'une collection de paramètres dans un message. À l'inverse, on parle de **désassemblage** ou *unmarshalling* pour la remise en forme (dans la pile) des paramètres.

On peut reprendre l'exemple du serveur de calcul traité dans l'exercice 6 pour illustrer de manière simple cette approche. Les fonctions accessibles sur le serveur de calcul pourrait aussi bien l'être à partir des fonctions suivantes :

```
1  int plus( int val1, int val 2);
2  int moins( int val1, int val 2);
3  int fois( int val1, int val 2);
4  int div ( int val1, int val 2);
```

Il est simple d'écrire le code de ces fonctions qui réalise l'invocation du serveur de calcul à partir des données passées en paramètre. Les fonctions ne ferait que d'initialiser les objets de requête, envoyer la requête attendre la réponse et retourner la valeur de résultat à l'appelant. Le code de ces fonctions peut même être automatiquement généré par un compilateur et mis dans une bibliothèque. En utilisant cette bibliothèque le programmeur n'a plus d'envoi/réception socket à faire puisqu'il ne fait que d'appeler ces fonctions et qu'elles font automatiquement l'invocation du serveur. De manière symétrique sur le serveur : le programmeur donne le code des fonctions et le serveur est généré de manière plus ou moins automatique.

En plus de l'automatisation cette approche permet d'ajouter des fonctionnalités, et donc de décharger le programmeur de certaines tâches. Par exemple, pour permettre les communications en environnement hétérogène, il est nécessaire de coder et typer les données

transmises, en plus de les encapsuler dans les messages. Les phases de codage/décodage peuvent être réalisées dans les souches.

L'ensemble des fonctions et services de communication constitue une sur-couche du système d'exploitation qui s'insère entre ce dernier et les application. Il est généralement appelé intergiciel ou *middleware*.

Pour le cours nous avons choisi d'illustrer la notion d'appel de procédure à distance ou d'invocation distante avec les Java RMI. Ce choix est fait pour des questions de simplicité dans la mesure où ils ne sont pas largement utilisés dans un contexte où les communications se font à travers Internet et où d'autres solutions s'avèrent plus adaptées (GRPC par exemple). Ils restent néanmoins une solution élégante et facile à mettre en œuvre qui permet un accès à ce concept sans un investissement trop conséquent.

2.2 Java RMI : Remote Method Invocation

RMI (*Remote Method Invocation*), ou invocation de méthodes à distance, permet de créer simplement des objets dont les méthodes peuvent être appelées à distance. Depuis la version 1.2, le JDK fournit un ensemble de classes prédéfinies (regroupées dans le paquetage `java.rmi` et ses sous-paquetages) et des outils qui rendent l'implémentation et l'utilisation des objets distants aussi simple que s'ils étaient locaux.

Schématiquement, le mécanisme RMI permet, dans un programme appelé **serveur RMI**, de créer un objet et de le rendre accessible aux autres programmes, pour lesquels il devient un **objet distant**. Dans l'un de ces programmes, dit **client RMI**, une application Java peut alors récupérer localement une représentation de l'objet distant, appelée **talon** (*stub*), sous la forme d'une référence d'objet distant. Il utilise alors cette référence comme il utiliserait une référence d'objet s'il était local sur à son programme. Notamment, il peut appeler, avec cette référence, les méthodes définies dans la classe de l'objet distant. Cet appel local de méthode sur la référence de l'objet distant est alors transmis par le talon du programme client jusqu'à la machine serveur où il engendre l'exécution, sur l'objet distant, de la méthode correspondant à celle qui a été appelée à distance. Grâce à RMI, ce mécanisme est transparent pour l'application cliente.

L'implémentation de RMI est basée sur le principe d'interface d'objet distant qui spécifie toutes les méthodes accessibles à distance sur l'objet. L'interface est l'entité partagée par les clients et le serveur. Elle est utilisée, comme tous les objets en Java, à partir de sa référence d'objet ; ici une référence d'objet distant. L'implémentation de l'objet interface repose sur deux objets souches : le **talon** (*stub*), du côté client, et le **squelette** (*skeleton*) du côté serveur comme illustré dans la figure 2.2. Du côté client, le talon simule l'objet distant : il offre les mêmes méthodes. Lors de l'appel d'une méthode sur un talon, ce dernier se connecte sur le serveur hébergeant l'objet distant et émet ensuite vers ce serveur une suite d'octets comprenant les identificateurs de l'objet distant et de la méthode appelée, suivis des arguments sérialisés : cette opération est appelée **assemblage** (*marshalling*). Du côté serveur, une fois l'objet localisé par le serveur, son squelette se charge de désérialiser les arguments et d'appeler la méthode souhaitée : cette opération est appelée **désambal-**

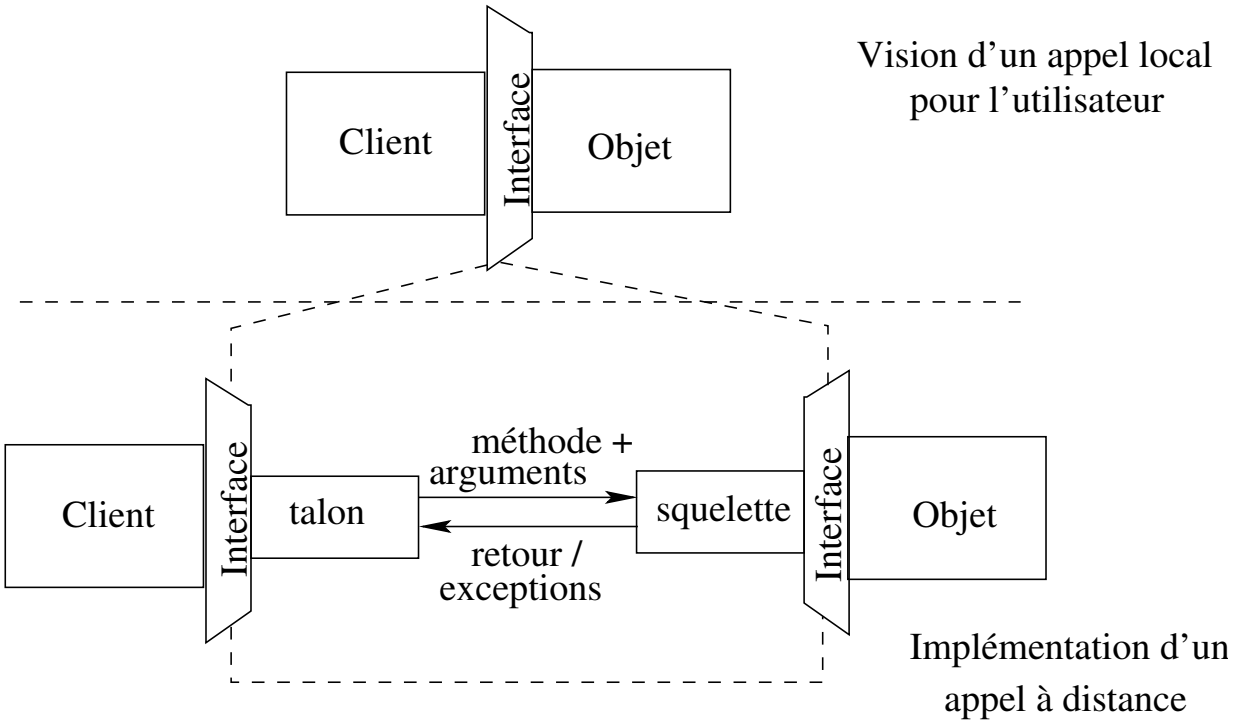


FIGURE 2.2 – Appel local versus appel à distance

lage (*unmarshalling*). Le squelette simule un appel local de côté du serveur et engendre l'exécution de la méthode sur l'objet. Une fois la méthode exécutée, c'est le squelette qui reçoit le résultat. Il se charge de le retourner en direction du talon qui lui a transmis l'appel. Cette transmission est réalisée par *marshalling* et la suite d'octets émise contient la forme du résultat, valeur ou exception, suivi de sa valeur sérialisée. Finalement, le talon du client reçoit ce résultat qu'il doit reconstruire (*unmarshalling*) avant de le retourner au client. Pour le client, tout s'est passé comme si l'appel sur le talon distant n'avait été qu'un simple appel à un objet local. L'ensemble de ces opérations est réalisé de manière totalement transparente aussi bien du côté client que du côté serveur.

Les **souches** (le talon et le squelette) masquent ainsi, au client et à l'objet distant, le fait qu'ils s'exécutent dans des programmes différents. Même s'ils agissent au même niveau pour la transmission des appels de méthodes, ces deux objets sont bien distincts l'un de l'autre : le talon implémente l'interface de l'objet pour le client et le squelette utilise la même interface pour transmettre les appels à l'objet distant. Ainsi, tout se passe comme si le client appelait classiquement (localement) la méthode via une interface d'objet.

2.3 Cycle de développement RMI

L'implémentation avec RMI d'une application comprenant un objet distant se décompose en cinq étapes :

1. **Écrire une interface pour l'objet distant**. Cette interface représente l'ensemble de méthodes, via lesquelles un objet distant pourra être manipulé.
2. **Écrire une implémentation de cette interface** du côté serveur. Cette classe servira à créer une instance (l'objet distant) respectant cette interface qui pourra être accédée à distance depuis la machine cliente. Cette implémentation expose, c'est-à-dire **rend accessible l'objet distant** dans un serveur d'objets, qui lui transmettra les appels distants de méthodes via son squelette.
3. **Générer, à partir de la classe implémentant l'interface, les classes souches** (du talon et du squelette) de l'objet distant [2].
4. **Publier l'existence de cet objet exposé** dans un serveur de noms, c'est-à-dire associer cet objet à un nom, à partir duquel des applications distantes clientes pourront récupérer un talon de cet objet (une instance de la classe du talon générée à l'étape 3).
5. **Appeler à distance des méthodes de l'objet distant** depuis la machine cliente, déclarées par l'interface, en utilisant le talon de l'objet exposé.

Ces étapes sont détaillées par la suite, sur la base d'un exemple classique : une applicationde de type `Hello`, mais en version distribuée. L'intérêtL'objectif est de réaliser une application distribuée qui permet au client d'envoyer une chaîne de caractères au serveur, qui l'affichera précédée du message «Hello».

2.3.1 Définir l'interface de l'objet distant

La première étape consiste à définir l'interface par laquelle les applications clientes auront accès aux services offerts par l'objet distant. Cette interface, comme toutes les interfaces en Java, est un objet virtuel dont nous devrons fournir une implémentation dans un objet (étape 2). Pour être accessible à distance cette interface doit obligatoirement hériter de l'interface `java.rmi.Remote`. De plus, les méthodes qui constituent cette interface doivent obligatoirement toutes déclarer lever une exception de la classe `java.rmi.RemoteException`. Cette classe d'exception est une sous-classe de `Exception`.

Pour l'exemple choisi, l'interface distante à déclarer, interface étendant l'interface `Remote` (ligne 5), contient une méthode distante (ligne 6) prenant en paramètre une chaîne de caractères, et levant l'exception `RemoteException`.

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3
4 public interface Hello extends Remote {
5   void sayHello(String param) throws RemoteException;
6 }
```

En tant qu'objet `public`, cette interface doit être déclarée dans un fichier indépendant, par exemple `Hello.java`.

2. Cette étape n'est plus nécessaire depuis Java 5.0 où les souches sont générés dynamiquement.

À noter que les exceptions de la classe `RemoteException` ou de l'une de ses sous-classes, sont levées dans trois cas :

— Erreur au niveau transport : par exemple, s'il n'est pas possible de se connecter sur le serveur d'objets distants ;
— Erreur au moment du passage de paramètres ou de retour de résultat : par exemple, si les classes des objets passés ou retournés ne sont pas sérialisables ;
— Erreur liée au protocole : par exemple si les versions des protocoles, JRMP (*Java Remote Method Protocol*) et sérialisation, utilisés entre le client et le serveur ne sont pas compatibles.

Ces deux contraintes interdisent l'utilisation immédiate d'un objet prévu pour une utilisation locale. Toutefois, il est possible d'utiliser l'héritage pour le rendre accessible à distance.

2.3.2 Définir l'objet distant

L'interface définit uniquement le prototype des méthodes accessibles sur l'objet distant, mais ne définit pas le comportement de ces méthodes car c'est un objet virtuel. C'est à l'objet distant de donner l'implémentation des méthodes de cette interface : donner le code exécuté lors de l'appel des fonctions.

Un objet distant Java doit être rendu accessible à distance pour qu'une application cliente puisse lui transmettre un appel de méthode à distance. Cette opération, appelée **exposition**, peut être réalisée grâce à l'héritage de classes. Actuellement, il existe deux classes principales qui permettent d'exposer des objets :

— la classe `java.rmi.server.UnicastRemoteObject` ;
— la classe abstraite `java.rmi.activation.Activatable`.

Dans cette section, nous nous concentrons sur `UnicastRemoteObject` et nous reviendrons sur la classe `Activatable` à la section 2.6.

L'appel (ligne 8)au constructeur de la classe mère `UnicastRemoteObject` (héritage ligne 5) entraîne implicitement et automatiquement l'exposition de l'objet. Au besoin, l'exposition peut être effectuée explicitement grâce à la méthode `exportObject` dans après la création de l'objet distant.

```
1  import java.rmi.server.UnicastRemoteObject;
2  import java.rmi.RemoteException;
3
4  public class HelloImpl extends UnicastRemoteObject
5                                          implements Hello {
6
7    public HelloImpl() throws RemoteException { }
8
9    public void sayHello(String param) throws RemoteException {
10     System.out.println("Hello " + param + "!");
11   }
12 }
```

La classe `UnicastRemoteObject` permet d'exposer des objets du côté serveur afin de les rendre accessibles, à distance, aux applications clientes. Elle encapsule, de façon transparente pour le développeur, la prise en charge des objets auprès d'un serveur d'objets. Ce serveur d'objets, exécuté dans un processus léger, attend les appels de méthodes des clients et les aiguille vers les instances concernées via leur squelette. L'objet distant est uniquement accessible si le serveur d'objets est actif. En particulier, si la commande qui a lancé le serveur d'objets est interrompue, l'objet n'est plus accessible à distance.

Hériter de la classe `UnicastRemoteObject` simplifie beaucoup l'écriture et l'utilisation des objets distants. Toutefois, cette approche a deux inconvénients. D'une part, la classe de l'objet distant ne peut pas hériter d'une autre classe. D'autre part, un objet de la classe de l'objet distant ne peut pas être utilisé uniquement en local, car l'objet est toujours exposé au moment de sa construction. Ces inconvénients sont palliés par l'utilisation de la méthode `exportObject`.

Toutes les méthodes déclarées dans l'interface doivent être implémentées dans la classe de l'objet distant, afin de pouvoir instancier des objets distants (ligne 10). Une implémentation d"objet distant peut cependant proposer plus de méthodes que celles qui sont définies dans l'interface.

2.3.3 Serveur : créer et publier l'objet distant

Le serveur est chargé essentiellement de créer et publier l'objet distant pour que les clients puissent y accéder.

Créer un objet distant

Un objet distant est créé comme tout autre objet d'une application, instance de la classe de l'implémentation. L'exception `RemoteException` doit être traitée au cas où une erreur se produit lors de l'instanciation.

Publier un objet distant

Pour faire connaître le nouvel objet exposé, ce dernier peut alors être *publié* dans un *serveur de noms* (un objet distant à son tour). Le serveur de noms maintient une table d'associations entre des noms et des références des objets distants qui y sont déclarés. Les objets situés sur la même machine que le serveur de noms peuvent y publier des références distantes. Ces références distantes seront ensuite récupérées, localement ou à distance, à partir de leur nom.

La classe `java.rmi.Naming` encapsule l'accès aux objets serveurs de noms. Pour cette classe, un nom est une URL, au format `//machine:port/nom`.

Les champs *machine* et *port* permettent de repérer l'objet serveur de noms, attaché à une adresse Internet et un port TCP particulier. Ces derniers peuvent être omis, dans ce cas la machine locale (`localhost`) et le port par défaut (1099) sont utilisés. Chaque fois

qu'une URL doit être passée en argument, si elle ne peut pas être interprétée correctement, une exception `MalformedURLException` est levée par la méthode correspondante.

Les méthodes de la classe `Naming`, toutes statiques, sont les suivantes :

- `bind(url, objet)` : tente de publier dans le serveur de noms repéré par les parties `machine` et `port` de l'URL, une association entre la partie `nom` de l'url et une référence distante correspondant à l'objet passé en paramètre ;
- `rebind(url, objet)` : une alternative de la méthode `bind`, permet d'associer une référence distante à un nom même si ce dernier est utilisé ;
- `unbind(url)` : efface une association du serveur de noms, à partir d'un nom de l'url donné en paramètre ;
- `lookup(url)` : permet de récupérer une référence distante associée à un nom donné sous la forme d'un URL ;
- `list(url)` : retourne un tableau de chaînes de caractères contenant la liste des noms actuellement publiés dans le serveur de noms dont l'URL de base est passé en paramètre.

Les instructions minimales de l'application serveur sont les suivantes :

- la création de l'objet distant, ayant le type de l'implémentation de l'objet distant car le serveur dispose de sa classe (ligne 9) ;
- la publication de l'objet distant, dans le serveur de noms, sous le nom «Hello» (ligne 10).

```
1  import java.rmi.Naming;
2  import java.net.MalformedURLException;
3  import java.rmi.RemoteException;
4
5  class HelloServer {
6    public static void main(String[] args) {
7      try {
8        HelloImpl helloObj = new HelloImpl();
9        Naming.rebind("Hello", helloObj);
10     } catch (RemoteException e) {
11       System.out.println("HelloServer exception " +
12                                  e.getMessage());
13     } catch (MalformedURLException e) {
14       System.out.println("HelloServer : url malformed " +
15                                  e.getMessage());
16     }
17   }
18 }
```

La création de l'objet distant et sa publication peuvent engendrer des erreurs (d'exposition ou d'accès au serveur de noms), sous forme d'exceptions. Ces exceptions nécessitent d'être traitées par un bloc `try/catch` (lignes 8, 11–17).

Le schéma des actions réalisées par le serveur est illustré dans la figure 2.3, en indiquant les lignes du code serveur à l'origine des actions.

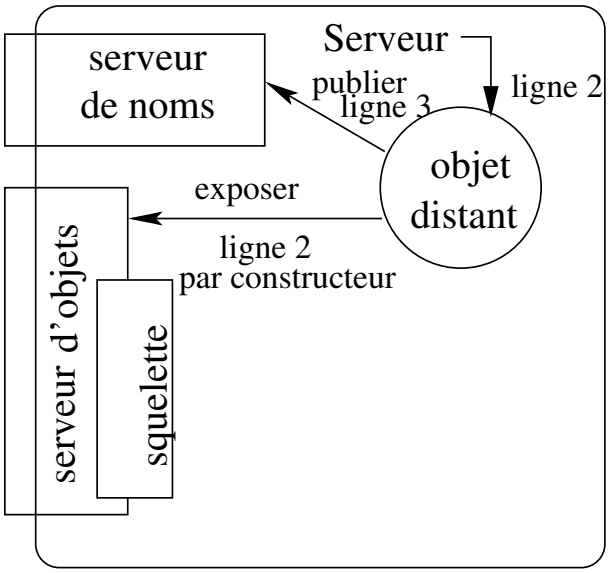


FIGURE 2.3 – Actions réalisées par l'application serveur

Création du serveur de nom depuis le serveur

Un serveur de nom peut également être démarré directement depuis un serveur avec les classes `Registry` et `LocateRegistry`. Dans ce cas il faut ajouter l'appel à la fonction `createRegistry(int port)` dans le code du serveur avant de faire le `rebind`.

```
1 import java.rmi.registry.LocateRegistry;
2 import java.rmi.registry.Registry;
3
4           ...
5           HelloImpl helloObj = new HelloImpl();
6           Registry registry = LocateRegistry.createRegistry(1099);
7           registry.rebind( "serveur", objServ ) ;
8           ...
```

2.3.4 Client : utiliser l'objet à distance

Pour utiliser l'objet à distance, nous construisons une application cliente. Celle-ci commence par récupérer un objet (le talon) correspondant à l'objet distant, via le serveur de noms qui se trouve sur la machine serveur. Pour cela, la méthode statique `lookup` de la classe `java.rmi.Naming` est utilisée. Cette méthode retourne un objet de la classe `java.rmi.Remote` qu'il faut transtyper dans l'interface de l'objet distant utilisé.

Les instructions minimales de l'application cliente sont les suivantes :

— la récupération de la référence de l'objet distant (son talon) par l'interrogation du serveur de noms, en connaissant le nom de l'objet publié (ligne 10). L'URL donné (en argument dans la ligne de commande) doit contenir l'adresse de la machine sur laquelle tourne le serveur de noms (sous forme d'adresse IP ou de nom de machine,

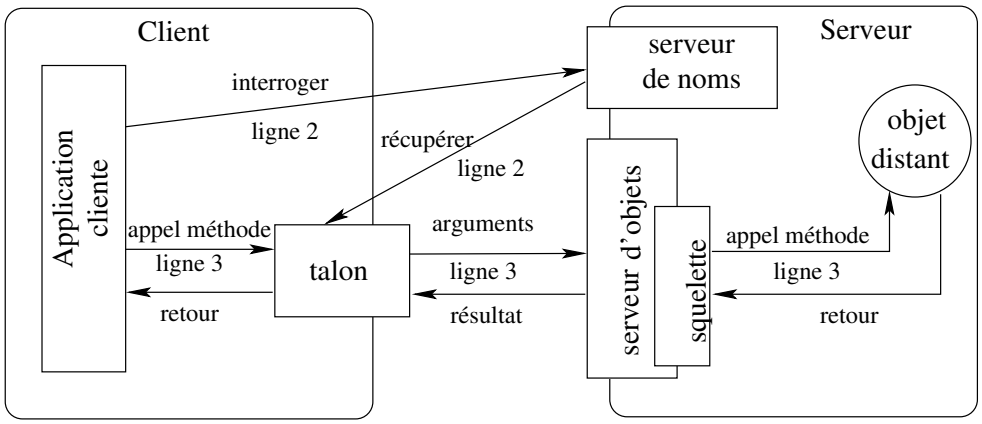


FIGURE 2.4 – Actions réalisées par l'application cliente

complété éventuellement par le numéro de port, si un autre que le port par défaut est utilisé). L'objet récupéré est de type `Remote`, qui sera transtypé vers le type de l'objet distant, en l'occurrence `Hello`. Le client manipule uniquement des objets distants ayant le type de l'interface, jamais le type de l'implémentation, car le client ne connaît pas l'implémentation des services.

— l'appel de méthode à distance, sur l'objet distant récupéré (ligne 12).

La récupération de la référence de l'objet distant et l'appel de méthode à distance peuvent engendrer des erreurs (d'accès à distance, de mauvais URL), traduites par l'apparition d'exceptions. Ces exceptions nécessitent d'être traitées par un bloc `try/catch` (lignes 9, 13–22).

```
1  import java.rmi.Naming;
2  import java.rmi.RemoteException;
3  import java.rmi.NotBoundException;
4  import java.net.MalformedURLException;
5
6  class HelloClient {
7    public static void main(String[] args) {
8      try {
9        Hello helloObject = (Hello)Naming.lookup("//" +
10                                          args[0] + "/Hello");
11       helloObject.sayHello("Vio");
12     } catch (RemoteException e) {
13       System.out.println("HelloClient exception " +
14                                  e.getMessage());
15     } catch (NotBoundException e) {
16       System.out.println("HelloClient : object not bound " +
17                                  e.getMessage());
18     } catch (MalformedURLException e) {
19       System.out.println("HelloClient : url malformed " +
20                                  e.getMessage());
21     }
22   }
```

```
23 }
```

Le schéma des actions réalisées par le client est illustré dans la figure 2.4, en indiquant les lignes du code client à l'origine des actions.

Comme pour tout appel de méthode, des paramètres peuvent être spécifiés. Les aspects liés à l'appel de méthode à distance seront détaillés dans la section 2.4.

2.3.5 Restrictions du développement RMI

Le développeur des applications réparties RMI doit respecter un certain nombre de règles de construction d'applications :

- définir des interfaces (étendant l'interface `Remote`) pour les objets distants ;
- toute méthode de l'interface doit obligatoirement déclarer lever l'exception `RemoteException` ;
- l'implémentation de l'objet distant doit obligatoirement étendre une classe d'exposition de l'objet, ici `UnicastRemoteObject` ;
- l'implémentation de l'objet distant doit contenir un constructeur vide qui lève l'exception `RemoteException` ;
- tout appel distant (invocation d'une méthode à distance) doit être encadré dans un bloc `try/catch` ;
- tous les objets non-distants passés en paramètre aux méthodes distantes doivent être sérialisables.

Le développeur **ne peut pas** :

- invoquer des méthodes statiques d'une classe distante ;
- accéder directement aux champs statiques d'une classe distante ;
- publier des objets dans un serveur de noms distant.

2.4 L'appel de méthode à distance

Lorsqu'un client effectue un appel à distance de méthode, cet appel entraîne normalement l'exécution d'une méthode, du côté serveur, sur l'objet exposé. Si un autre client appelle au même instant une méthode du même objet, il se peut que les deux méthodes soient exécutées en concurrence, dans deux processus légers du côté serveur. Il faut que le programmeur prévoit, lors de l'implémentation des objets distants, la gestion de cette concurrence, en utilisant des blocs d'instructions synchronisés.

L'appel à distance d'une méthode n'est pas équivalent à l'appel local de cette méthode. Dans le cas d'un appel distant, les paramètres, la valeur de retour et les exceptions sont transmis par copie. Par exemple, le paramètre vu par l'objet exposé est une **copie** (*clone*) de l'argument passé par le client lors de l'appel à distance. Si ce comportement est le même que le comportement usuel pour les variables de type primitif, il est différent du comportement pour les objets qui sont passés par référence lors des appels locaux des méthodes. Cela implique que, dans le cas général, une modification faite à distance sur un paramètre n'est pas répercutée sur l'argument passé par le client.

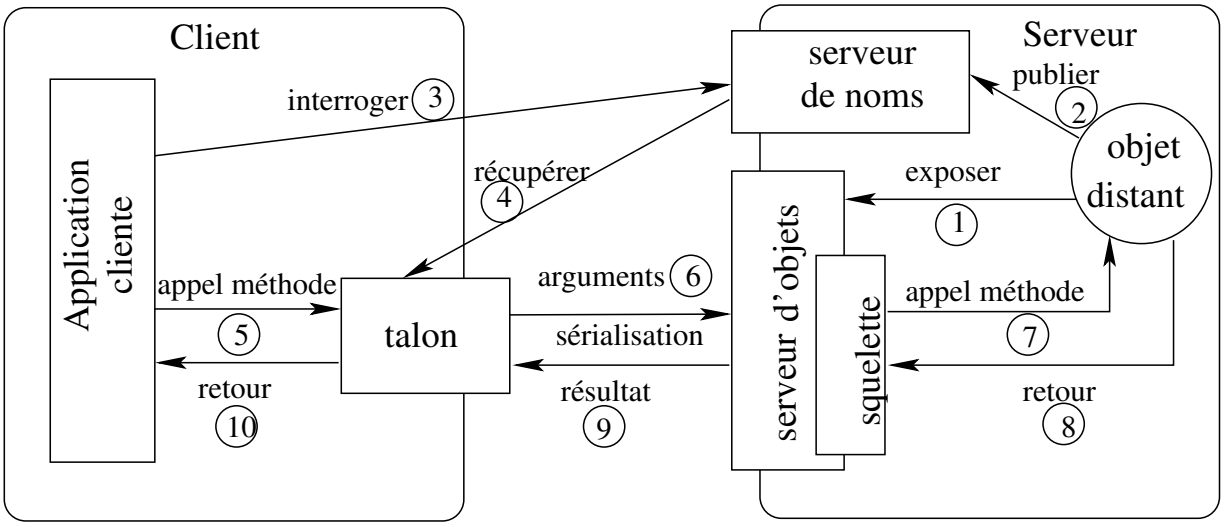


FIGURE 2.5 – Différentes étapes d'un appel de méthode à distance RMI

La sémantique du passage par copie est due à l'utilisation de la sérialisation pour transmettre les arguments, les exceptions et la valeur de retour. Un objet désérialisé (par exemple le paramètre reçu par l'objet exposé) est une copie de l'objet sérialisé (l'argument envoyé lors de l'appel sur le talon). Le mécanisme de sérialisation assure toutefois que, si un même objet est accessible par deux arguments (ou de deux façons), une seule copie est transmise.

Afin que la sérialisation soit possible, les paramètres, les exceptions ou la valeur de retour doivent être donc sérialisables. Les types primitifs et les objets distants le sont par défaut. Par contre, les types utilisateur ne sont pas généralement sérialisables. Ils peuvent le devenir si les classes correspondantes implémentent l'interface `java.io.Serializable`.

Les différentes étapes de l'appel à distance RMI sont résumées dans la figure 2.5. L'appel à distance en RMI est bloquant : les instructions suivant un appel à distance ne peuvent s'exécuter qu'après la terminaison du traitement distant. Ce comportement empêche donc que les cycles CPU soient utilisés par l'application cliente pendant un appel à distance. Pour pallier à cet inconvénient, l'application cliente est généralement rendue multithreadé : un thread réalise l'appel à distance, pendant que l'application continue à effectuer d'autres traitements. Une synchronisation est possible afin de s'assurer que le traitement a été effectué, ou que le résultat calculé par l'appel distant est disponible. Ce modèle de programmation distante multithreadée (avec les contraintes de synchronisation qui s'imposent) doit entièrement être géré par le programmeur.

2.5 Le chargement dynamique de classes

Dans l'exemple précédent, le bytecode de toutes les classes utilisées par les clients, les serveurs d'objets ou le serveur de noms devaient être accessibles par le chargeur de classes

courant, c'est-à-dire que les classes devaient être accessibles via la variable d'environnement CLASSPATH. En effet, rappelons que lors de la sérialisation d'un objet, le bytecode de sa classe n'est pas placé dans le flot. Il faut donc récupérer ce bytecode, lors de la désérialisation, par un autre moyen.

RMI propose un mécanisme permettant de charger automatiquement, depuis un URL de base, le *codebase*, et grâce aux protocoles FILE, HTTP ou FTP, le bytecode des classes désérialisées, dans le cas où le bytecode n'est pas accessible par le chargeur de classes courant.

Un codebase est un emplacement à partir duquel les classes sont chargées dans la machine virtuelle Java. Pour RMI, le codebase est utilisé afin de spécifier un ou plusieurs URL à partir desquels les talons des classes distantes (et d'autres classes aussi) peuvent être chargés. RMI précise cette information grâce à la propriété `java.rmi.server.codebase`. Généralement, les classes nécessaires à l'exécution des appels à distance doivent être rendues accessibles à partir d'une ressource réseau, comme un serveur HTTP ou FTP.

2.5.1 Principe de téléchargement dynamique de bytecode

Lors de la demande d'exécution du bytecode, l'URL placé dans le flot d'objets au moment de la sérialisation est lu, puis la méthode tente de charger le bytecode de la classe. La classe est d'abord recherchée par le chargeur de classes courant. Si elle n'est pas trouvée, le bytecode de la classe est chargé, si possible, depuis l'URL trouvé dans le flot. Dans ce cas, le chargement est réalisé par la classe `java.rmi.server.RMIClassLoader` et, plus précisément, par sa méthode statique `loadClass`. Celle-ci prend en arguments l'URL trouvé dans le flot et le nom de la classe recherchée ; si la classe est trouvée, elle retourne un objet de classe `Class` correspondant et sinon, elle lève une exception de la classe `ClassNotFoundException`.

La figure 2.6 illustre le principe de téléchargement dynamique de classes en RMI. Dans celle-ci, un client reçoit (désérialise) deux objets de classes `A` et `B` provenant de l'objet distant `Objet1`, qui les a lui-même obtenus de l'objet distant `Objet2`. On suppose que les classes `A` et `B` sont disponibles sur la `machine3`, mais que seule la classe `A` est disponible sur la `machine2`.

Lors de la transmission sérialisée de chacun de ces objets entre l'objet `Objet2` et l'objet `Objet1`, la valeur de `codebase` a été placée dans le flot au moment de la sérialisation, sur la `machine3`. Au moment de la désérialisation, sur la `machine2`, de l'objet de la classe `A`, l'objet `Objet1` charge la classe avec le chargeur de classes courant car elle est trouvée localement. En revanche, la classe `B` n'est pas trouvée localement sur la `machine2` et l'URL de base trouvé dans `codebase2` est donc utilisé pour charger le bytecode de la classe à distance.

Conformément à ce que nous avons dit plus haut, lorsque ces objets doivent être sérialisés par `Objet1` pour les transmettre au client de la `machine1`, la classe `A` est annotée avec `codebase1` (puisqu'elle a été chargée par le chargeur de classe courant de l'objet `Objet1`) tandis que la classe `B` est annotée avec `codebase2` (qui contient l'URL depuis lequel son bytecode a été chargé).

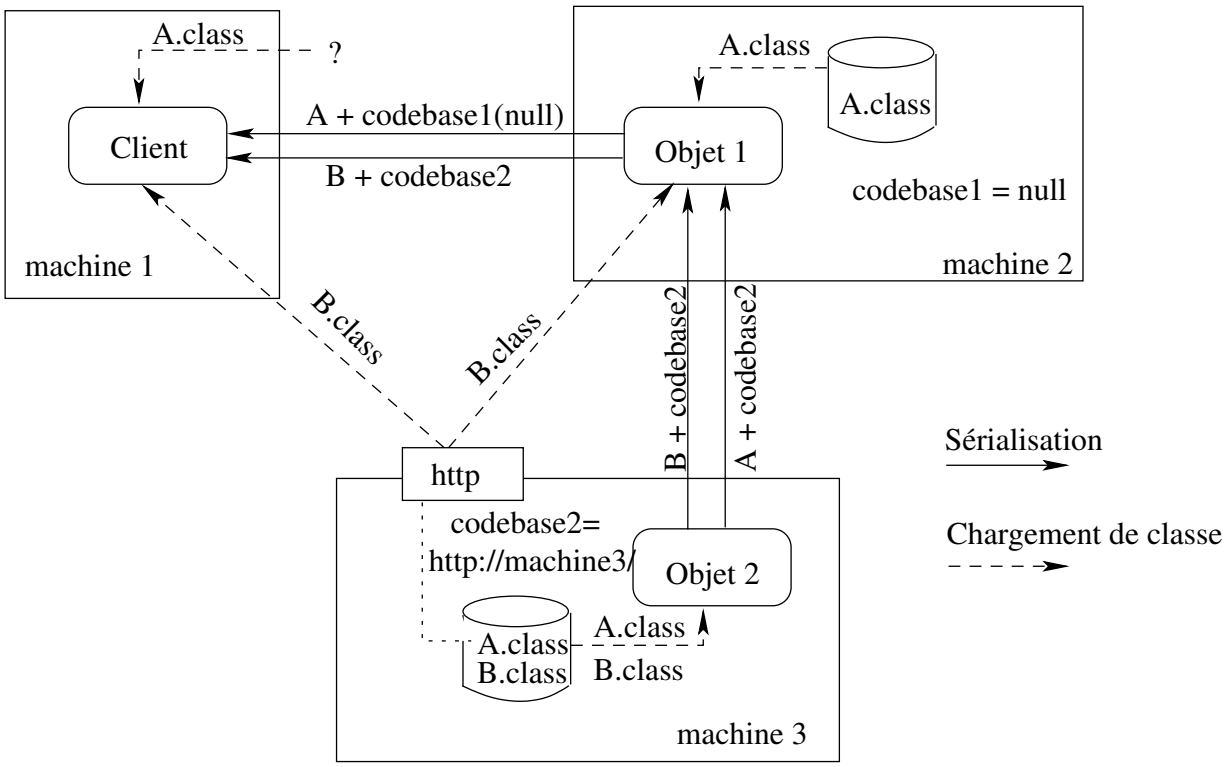


FIGURE 2.6 – Chargement dynamique de classes en RMI

Finalement, le client peut charger la classe `B` depuis la `machine3` mais il échoue lors du chargement de la classe `A` alors que si ces objets avaient directement été reçus de l'objet `Objet2`, les deux classes auraient été chargées avec succès à partir du même codebase.

Cette remarque permet de comprendre (et d'éviter) une erreur classique liée au serveur de noms. Supposons que le serveur de noms dispose dans son CLASSPATH de la classe du talon de l'objet distant. Au moment de la récupération du talon par le client, via la méthode `lookup`, l'URL de chargement trouvé dans le flot sera toujours nul si l'objet distant avait spécifié un URL de chargement. En effet, la propriété `java.rmi.server.codebase` de l'objet serveur de noms créé par `rmiregistry` est vide (null). Le client ne pourra donc pas charger les classes à distance comme cela aurait été possible si elles n'avaient pas fait partie du CLASSPATH du serveur de noms. Il est donc important de lancer le serveur de noms (par la commande `rmiregistry`) depuis un répertoire qui ne contient aucune des classes qui doivent être chargées à distance.

Nous avons détaillé le mécanisme de chargement dans le sens serveur vers client, mais le serveur peut également charger les classes des objets passés en arguments par le client. Pour cela, le client doit préciser une URL de base pour le chargement des classes de ces arguments, grâce à la propriété système `java.rmi.server.codebase`.

2.5.2 Sécurité en RMI

Afin de préserver la sécurité, les applications cliente et serveur RMI doivent créer et installer un gestionnaire de sécurité, afin de protéger l'accès aux ressources système du code non fiable chargé, qui s'exécute dans la machine virtuelle. Le gestionnaire de sécurité utilisé classiquement est une instance de la classe `java.rmi.RMISecurityManager`. Dans la mesure où un gestionnaire de sécurité est installé, il faut également prévoir un fichier politique de sécurité qui permette de charger le bytecode des classes.

Le code qui permet la création et l'installation d'un gestionnaire de sécurité est le suivant (à utiliser en début du code du client et du serveur) :

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

Le fichier de sécurité `java.policy` suivant permet au code téléchargé de réaliser deux opérations :

- se connecter ou accepter des connexions sur les ports à partir de 1024 au 65535 sur tout hôte ;
- se connecter sur le port 80 (le port HTTP).

```
grant {
  permission java.net.SocketPermission "*:1024-65535",
        "connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
};
```

Le lancement des applications (serveur et cliente) dans le cas d'utilisation de la politique de sécurité est réalisé grâce à la propriété `java.security.policy` :

```
java -Djava.security.policy=java.policy hello.HelloServer
java -Djava.security.policy=java.policy hello.HelloClient m1:1099
```

2.6 L'activation d'objets distants

Lors de l'implémentation d'objets distants tels que ceux exposés grâce à la classe `UnicastRemoteObject`, le développeur est confronté à deux problèmes contradictoires :

- si les objets distants sont exposés (en attente de connexions) en permanence, les ressources mémoire et réseau de la machine qui les héberge sont monopolisées inutilement en cas d'utilisation peu fréquente ;
- si l'exposition des objets distants est interrompue (c'est-à-dire si le serveur d'objets qui les expose est arrêté), volontairement ou involontairement, puis reprise, un talon obtenu par un client avant la nouvelle exposition n'est plus valable pour accéder à la nouvelle instance de l'objet distant ainsi réexposée.

Les classes du paquetage `java.rmi.activation` et la commande `rmid` ont été introduites, dans la version 1.2 de Java, afin d'apporter des solutions à ces problèmes. Plus particulièrement, la classe abstraite `Activatable` doit être utilisée pour le développement des objets destinés à être accessibles de façon permanente par les clients, l'utilisation de la classe `UnicastRemoteObject` étant désormais réservée au développement d'objets distants "temporaires".

L'idée de ces nouvelles implémentations est de permettre de créer (ou de recréer) les objets distants, dynamiquement, lorsqu'un client souhaite y accéder. Ces objets distants sont appelés *activables*, mais leur activation (création) est transparente pour le client pour qui tout se passe comme si l'objet distant avait toujours été exposé (actif).

Avec les objets de la classe `UnicastRemoteObject`, l'exposition d'un objet distant impliquait nécessairement que cet objet était actif, c'est-à-dire prêt à recevoir des appels distants. L'utilisation de la classe `Activatable` ajoute un niveau d'indirection dans l'accès aux objets distants : les appels de méthodes distantes des clients sont maintenant transmis à un objet représentant l'objet distant, que ce dernier soit actif ou non, c'est-à-dire créé et exposé ou non. L'activation éventuellement nécessaire pour accéder à l'objet distant est gérée, de manière transparente pour le client, par le représentant de cet objet qui lui, est toujours accessible.

Le cœur du mécanisme d'activation est la commande `rmid` qui prend en charge l'activation des objets distants. Néanmoins, ce mécanisme n'étant un mécanisme générique de la communication (on parle plus de déploiement dans ce cas) nous ne développons pas le fonctionnement des objets activatable.

Chapitre 3

Les bus de messages

3.1 Introduction

La communication événementielle n'est pas, à proprement parler un mode de transfert de données. Il s'agit plutôt d'un modèle de communication qui, plutôt que de s'appuyer sur des messages, repose sur le concept d'évènement. Ce modèle est, par exemple, implémenté par la couche de communication par *Topics* de JMS (*Java Messaging Service*).

Un événement est une entité proche du message mais destinée à être publiée (envoyée) sur un **bus d'événements**. Les émetteurs d'évènements s'inscrivent donc sur le bus en tant que publieurs. De manière symétrique, les programmes en attente d'évènements s'inscrivent sur le bus en tant que récepteurs. Le bus d'évènements est alors chargé de faire suivre le message aux destinataires soit en mode *push* soit en mode *pull*. Dans le mode *push*, les événements sont directement diffusés (poussés) depuis le bus vers les récepteurs dès leur émission. Dans le mode *pull*, les événements sont enregistrés par le bus et c'est aux récepteurs de venir les chercher eux-mêmes. Les événements en mode *pull* peuvent utiliser de la communication synchrone, le récepteur vient périodiquement consulter les événements qui ont été publiés.

Dans ce chapitre nous nous intéressons à la communication par messages. Un message est un ensemble de données, groupées pour être envoyées puis reçues ensemble. Dans une communication, le processus classique consiste donc d'abord à préparer un message (marshalling), à l'envoyer puis à le recevoir avant d'en extraire les données (unmarshalling). À l'opposé de l'approche RMI qui cache la notion de messages sous une interface et une invocation distante, la communication par messages conserve cette notion qu'elle utilise comme support de la communication. Elle se positionne cependant à un niveau plus haut que la communication par socket car les messages sont typés et structurés : c'est-à-dire que deux messages sont deux objets distincts qui seront forcements reçus indépendemment, contrairement aux sockets ou deux envois des données peuvent être reçus sous la forme d'un seul.

L'autre point important de la communication par message est la notion de bus. La communication par message est généralement implantée par un bus de messages ou *MOM :*

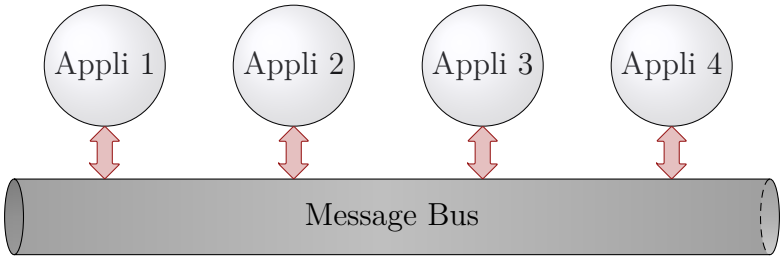


FIGURE 3.1 – Architecture d'un bus de messages

Message Oriented Middleware[1]. La figure 3.1 donne un aperçu de l'architecture d'un bus de messages. Le bus dans ce cas cache aux applications l'implémentation au dessus du réseau et l'hétérogénéité de celui-ci.

Une des différences principales de la communication par MOM vient du fait que les échanges ne sont plus directs mais passent par des objets intermédiaires, gérés par le bus. Un objet intermédiaire est un peu comme des boîtes aux lettres : les messages sont envoyés sur l'objet intermédiaire et le récepteur doit venir le chercher, ou est informé de leur arrivée. Nous parlons alors d'adressage indirect puisque le message n'est plus envoyé directement à son destinataire. Nous parlons alors de *couplage faible* entre les composants d'une application distribuée. Ce mode permet d'introduire plus de souplesse et flexibilité dans l'application et sa gestion. Dans un contexte de modularité les MOM peuvent apporter une solution élégante à l'interaction entre modules comme nous le verrons.

L'utilisation du bus de messages et de ces objets intermédiaires permet également de mettre en place des services qui n'existent pas en communication directe :

— Transaction : envois et réceptions groupés.
— Persistance : retransmission des messages en cas de panne.
— Multicast : communication de groupe.
— Trace des échanges (*log*).
— Gestion des droits d'accès.
— Transparence des échanges : protocole de communication facilitant l'encapsulation d'objets ou d'autres données typées.

A noter tout de même que ces MOM s'appuient généralement, comme la plupart des couches de communication actuelles, sur le protocole TCP/IP et donc sur les sockets. De ce fait ce mode de communication a aussi un coût plus élevé puisqu'il ajoute des relais, le MOM, et des couches logicielles, les bibliothèques d'échange avec le MOM.

Un grand nombre d'applications peuvent bénéficier du support des bus de messages, à condition qu'elles soient distribuées bien sûr. Parmi celles-ci nous pouvons noter les applications dites de type workflow. C'est-à-dire où un ensemble de composants agissent de concert pour rendre un même service. Nous pouvons illustrer ce type d'applications par la gestion d'une commande pour une fabrication à la demande comme sur la figure 3.2 par exemple.

1. Couche logicielle intermédiaire située entre l'application et le système d'exploitation de la machine

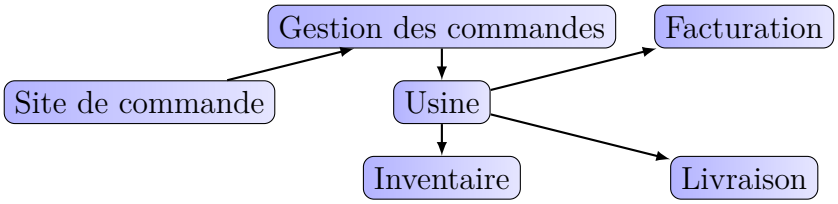


FIGURE 3.2 – Exemple de workflow

De nombreuses plateformes implémentent la communication par message. Chacune de ces plateformes est accessible à travers une interface (API ou Application Programming Interface) qui lui est propre. Dans ce chapitre nous étudions principalement l'interface JMS avec l'objectif d'en donner une première vue pour comprendre la différence de la communication par message par rapport aux autres couches de communication. Nous n'en donnons donc qu'un aperçu partiel. Les étudiants souhaitant approfondir le sujet pour se référer aux documents donnés en bibliographie.

3.2 JMS

JMS (Java Message Service) est une spécification qui unifie, dans l'environnement Java, la programmation de l'accès aux plateformes de messages. JMS n'est donc pas un logiciel mais ce sont les plateformes de messages qui offrent une interface compatible JMS. Ainsi les programmes qui s'exécutent sur une plateforme compatible JMS doivent pouvoir s'exécuter sur une autre plateforme compatible JMS et les développeurs JMS peuvent passer d'une plateforme à l'autre sans avoir à acquérir de nouvelles compétences.

Nous détaillons dans la suite les principes généraux de JMS avant de détailler la façon de programmer.

3.2.1 Principes généraux

La conception de la spécification JMS a été réalisée sur la base de plateformes existantes avec pour objectif de les unifier, et non pas indépendemment de l'existant. Ceci justifie en partie que l'architecture ou certains concepts peuvent paraître complexes ou peu intuitifs.

La figure 3.3 [2] montre le positionnement de JMS dans l'écosystème logiciel Java. Nous pouvons noter que la spécification différencie la gestion des objets administrés et des droits d'accès de la gestion des échanges de messages. Parmi les rôles définis nous avons :

— Le fournisseur JMS (*provider*) qui est la plateforme de messagerie.
— Les clients JMS : composants produisant et consommant des messages.
— Messages : contient les informations communiquées entre les clients.
— Objets administrés : objets JMS configurés par un administrateur. Rque : Les objets administrés ne sont pas manipulés par un programme.

2. Les figures de cette partie proviennent principalement du site web d'Oracle

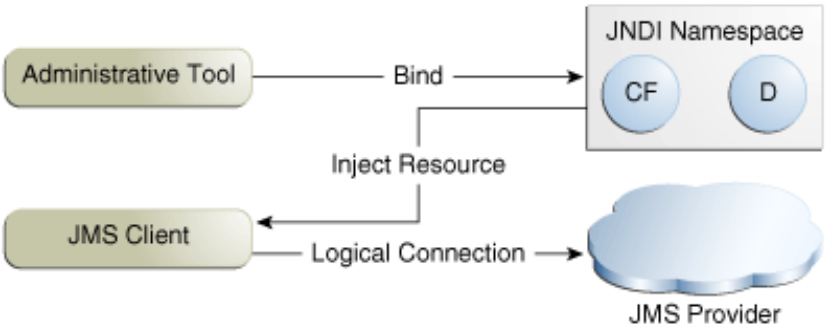


FIGURE 3.3 – Positionnement général de JMS

L'intérêt d'un MOM par rapport à une programmation socket de plus bas niveau est de permettre une gestion avancée des messages et plusieurs modes de communication. JMS repose sur deux modes de communication principaux qui sont la communication point-à-point et la communication événementielle ou par topics. Chacun de ces modes de communication utilise des objets intermédiaires par lesquels transitent les messages. De ce fait il n'y a pas besoin d'établir une connexion entre un émetteur et un récepteur pour réaliser un échange de messages. Les objets intermédiaires permettent de plus la mise en place de services de communication avancés.

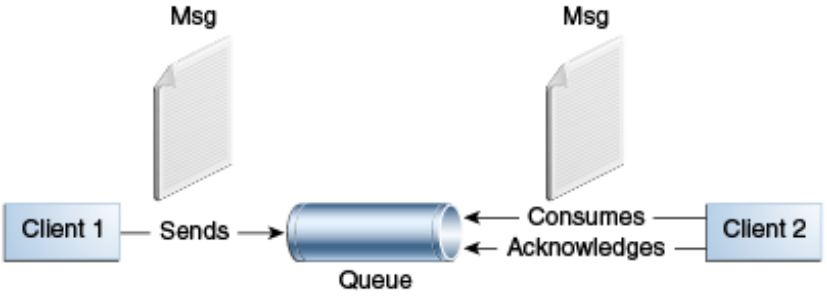


FIGURE 3.4 – Communication point-à-point dans JMS

La figure 3.4 détaille le principe général de la communication point-à-point dans JMS. Dans la communication point-à-point l'objet intermédiaire est une file appelé (*queue*). Un émetteur envoie donc ses messages sur la queue. Plusieurs émetteurs peuvent envoyer sur la même queue mais un seul destinataire peut consommer le message sur la queue. Nous parlons de communication N à 1. La communication est asynchrone (l'émetteur n'a pas besoin d'attendre le récepteur) mais mémorisée. Le récepteur peut ainsi consommer le message même s'il était hors-ligne lorsque ce dernier a été émis.

La figure 3.5 détaille le principe général de la communication événementielle (*publish/-subscribe*) dans JMS. Dans la communication événementielle l'objet intermédiaire est un canal ou sujet appelé (*topic*). Les émetteurs envoient, publient, donc leurs messages sur un topic. Les récepteurs s'inscrivent au topic pour recevoir les messages qui sont publiés sur ce

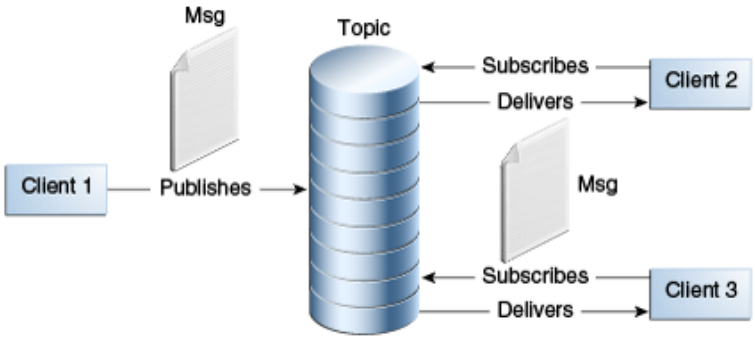


FIGURE 3.5 – Communication événementielle dans JMS

topic. Plusieurs émetteurs peuvent envoyer sur le même topic et plusieurs consommateurs peuvent recevoir les messages du topic sur lequel ils sont inscrits. Il est également possible de s'inscrire à plusieurs topics simultanément. Un client qui a souscrit à un sujet ne peut consommer que les messages émis après la souscription.

A noter que, quelque soit l'objet intermédiaire utilisé dans la communication, le consommateur de message dispose de deux modes pour recevoir les messages :

— le mode synchrone : un consommateur peut explicitement recevoir un message de la destination avec la méthode bloquante `receive`.

— le mode asynchrone : une application peut enregistrer un *listener* avec un consommateur. Un listener est un objet spécifique de Java qui se met en écoute d'un évènement. L'objet listener définit une méthode spécifique qui est appelée au déclenchement de l'évènement auquel il est associé. Pour la communication la méthode est `onMessage`. Lorsqu'un message sera reçu, la méthode `onMessage` est donc appelée.

3.2.2 Mise en place de la communication

L'interface JMS repose sur un ensemble de concepts, implémentés sous la forme d'objets, qui permettent la création et l'administration des objets de communication, la définition des propriétés des échanges et la communication elle-même.

La version actuelle de la spécification JMS est la version 2.0, qui date de 2013. Par rapport à la spécification 1.0, la version 2.0 introduit un certain nombre de simplifications.

Les principaux objets disponibles dans l'interface JMS sont les suivants :

Objets administrés : les fabriques (`ConnectionFactories`) qui représentent le provider et les destinations (`Destination`) qui représentent les queues et les topics. Les objets administrés ne sont pas créés par le programme JMS mais plutôt par le provider. Le programme initie une instance à partir d'objets existants.

Connections : cet objet représente la connexion de notre programme au provider JMS. En général la création de la connexion est l'occasion de s'identifier auprès du provider, à l'aide un nom d'utilisateur et d'un mot de passe. Pour pouvoir être utilisée, c'est-à-dire envoyer et recevoir des messages, une connexion doit être dé-

marrée. Elle est bien sûr arrêtée en fin d'utilisation. Une connexion est un objet lourd il n'est donc pas recommandé de l'arrêter tant que le programme a encore des communications.

Sessions : un provider offre plusieurs modes d'échange comme le mode transactionnel et l'auto-acquitement. Ces modes sont fixés à la création de la session. La session sert ensuite à une suite d'échanges. Pour changer le mode de communication il faut arrêter la session courante et en démarrer une autre avec les paramètres désirés.

JMSContext : les objets `Connection` et `Session` sont, pour des raisons de simplification, regroupés au sein de `JMSContext`. Attention, bien que la spécification 2.0 regroupe les connections et les sessions, il est important de comprendre ces deux notions qui restent sous-jacentes et permettent de définir différents modes d'accès.

Producteur de messages : l'objet `JMSProducer` permet l'envoie des messages sur une destination.

Consommateur de messages : l'objet `JMSConsumer` permet de recevoir des messages à partir d'une destination.

Messages : objet qui contient les information échangées. Un message est composé de plusieurs champs que nous détaillons dans la suite.

L'ensemble de ces objets sont en fait, dans la définition JMS, des interfaces donc des objets virtuels. Ce sont les bibliothèques liées au provider qui implémentent ces interfaces en objets. Pour cette raison, les programmes utilisent parfois directement l'objet d'implémentation plutôt que l'interface générique.

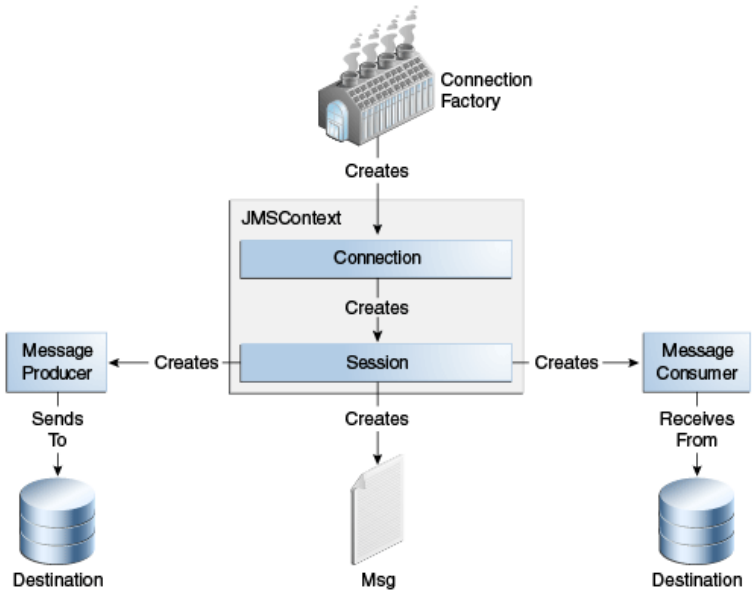


FIGURE 3.6 – Organisation générale de JMS

La figure 3.6 donne l'architecture générale de l'accès à un service JMS. Le principe de programmation à retenir est le suivant :

1. un programme qui souhaite accéder à une plateforme JMS doit d'abord s'adresser à une fabrique `ConnectionFactory`
2. le programme demande à la fabrique un contexte (`JMSContext` lui permettant de s'identifier si nécessaire.
3. L'objet context permet de créer des objets de communication `JMSProducer` ou `JMSConsumer`, avec lequels il est possible d'envoyer et recevoir des messages, et des messages.

A noter que cette architecture, permet de décoréler la gestion (création, nommage) des objets de communication de l'interface de programmation : ce ne sont pas les programmes JMS qui s'occupent de créer et gérer le cycle de vie des objets de communication.

3.2.3 Programmation JMS

Dans cette partie nous expliquons comment sont utilisés chacun des objets JMS et comment ils sont accèdes dans les programmes[3]. Nous présentons la version 2.0 qui est plus simple d'utilisation et profite mieux des avancées du langage Java. Les exemples pratiques que nous montrons ont été réalisés avec le bus de messages Artemis (ActiveMQ). Certaines dépendances à ce bus sont toujours visibles dans le code, car ce bus ne propose pas de support jndi.

Accès aux fabriques de connexions

La logique de programmation JMS suppose l'accès à un MOM, ou *provider*, identifié grâce à une *Connection factories* pour l'initialisation. Les `ConnectionFactory` sont les objets qui permettent de créer une connexion au *provider*. L'accès se fait à partir d'une implémentation de la classe. Dans ce cas d'Artemis :

```
1  ConnectionFactory factory = (ConnectionFactory) new ActiveMQConnectionFactory()
```

Si le serveur n'a pas été lancé sur l'URL par défaut, il est possible de préciser celle-ci en paramètre. L'adresse peut avoir une forme comme `"tcp://localhost:61616"`, classiquement pour le MOM ActiveMQ. Dans un serveur d'application[4], l'accès peut aussi se faire à partir d'une injection JNDI telle que :

```
1  @Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
2  private static ConnectionFactory connectFactory;
```

où la valeur donnée à `lookup` dans l'annotation `@Resource` est définie dans les propriétés de l'application (ou dans un fichier `jndi.properties`). Ceci permet de décoréler le code des dépendances au contexte d'exécution (l'URL de connexion).

3. Pour une description complète de l'interface de programmation JMS il est recommandé de se référer à la javadoc correspondante

4. Les serveurs d'application sont des infrastructures offrant un contexte d'exécution pour des composants applicatifs. Jboss est un exemple de serveur d'application.

Création et accès aux destinations

Une fois l'accès à la `ConnectionFactory` obtenu il est nécessaire d'initialiser la destination, c'est-à-dire soit la queue soit le topic qui est utilisé pour l'échange de messages. Cela se fait avec la fonction `createQueue`, que cette queue existe déjà ou pas.

```
1  Queue queue = jmsc.createQueue("CsdQueue");
```

Comme précédemment nous pouvons, avec un serveur d'application, préciser le nom de la destination visée à partir d'annotations, par exemple dans l'exemple suivant :

```
1  @Resource(lookup = "jms/CsdQueue")
2  private static Queue queue;
3
4  @Resource(lookup = "jms/CsdTopic")
5  private static Topic topic;
```

Le bus ActiveMQ que nous utilisons applique une politique dite paresseuse des création des destinations. C'est-à-dire que, par défaut, il crée une destination pour laquelle un accès est demandé.

Connections et sessions

Pour des raisons de simplification, l'interface 2.0 regroupe les notions de connexion et session au sein d'un objet `JMSContext`. Le `JMSContext` permet alors de créer des producteurs et des consommateurs de messages, des messages et des destinations temporaires. La création du `JMSContext` se fait, avec la fonction `createContext` à partir de la connectionFactory, de la manière suivante :

```
1  JMSContext context = connectFactory.createContext();
```

À noter de le contexte ne contient qu'une seule session. Si le programme a besoin de plusieurs sessions, par exemple pour avoir des propriétés de communication différentes telles que la persistance, il doit créer un nouveau contexte. C'est également le cas si le contexte doit être accédé par des threads différents car les sessions ne supportent pas le mulit-thread.

Dans le `JMSContext` la connexion au bus de messages est activée par défaut mais son activation peut être gérée (méthode `start`) ou désactivée (méthode `stop`) par le programme. Une connexion désactivée ne délivre plus de messages au programme ni aux objets en attente tels que les listeners. L'activation de la connexion n'a aucun effet sur l'envoi de messages.

Envoi/réception de messages

Les envois et réception de messages sont réalisés à l'aide des objets `JMSProducer` et `JMSConsumer`.

Pour envoyer des messages, il est nécessaire de créer un objet `JMSProducer`. Par exemple de la manière suivante :

```
1  JMSProducer producer = context.createProducer(dest);
```

où la variable `dest` est soit une queue soit un topic. L'objet `JMSProducer` permet de positionner les propriétés associées au message. À noter que nous ne précisons pas, dans sa création, quelle est la destination des messages qui seront envoyés. Cette destination est précisée au moment de l'envoi.

De manière équivalente, pour recevoir un message nous avons besoin d'un `JMSConsumer`.

```
1  JMSConsumer consumer = context.createConsumer(dest);
```

Pour sa création nous précisons cette fois l'objet destination utilisé, contrairement au `JMSProducer`. Ainsi il est nécessaire d'avoir plusieurs `JMSConsumer` pour accéder à différentes destinations alors qu'un seul `JMSProducer` est suffisant. La raison de cette asymétrie provient de la possibilité de réception asynchrone que nous voyons plus loin.

À partir des producteurs et consommateurs définis précédemment il est possible de réaliser l'échange des messages. Voici un exemple simple d'envoi de message :

```
1  producer.send(queue, message);
```

Où l'objet `message` est du type `Message` que nous détaillons plus loin.

Et, pour la partie réception, nous avons :

```
1  Message message = consumer.receive();
```

Comme pour les sockets, la fonction `receive` est bloquante. Il s'agit alors d'une communication synchrone telle que nous l'avons définie dans la partie 3.2.1. Le programme sera donc bloqué tant qu'il n'aura pas reçu un message. À noter que cette fonction peut prendre un temps maximal d'attente en paramètre.

À noter que l'objet `JMSProducer` est léger et peut donc être recréé à chaque utilisation comme dans le code suivant.

```
1  context.createProducer().send(dest, message);
```

Pour ne pas bloquer un client dans une réception bloquant, JMS propose une fonctionnalité de communication asynchrone. Pour cela il faut créer un objet *listener*. Cet objet doit implémenter l'interface `MessageListener` qui ne contient qu'une méthode : `onMessage` qui prend en paramètre un objet de type `Message`. La méthode `onMessage` est invoquée à la réception de chaque message. Le message passé en paramètre de la fonction est celui qui est reçu, comme dans l'exemple suivant :

```
1  class MyListener implements MessageListener {
2    public void onMessage(Message msg) {
3       System.out.println(((TextMessage)msg).getText());
4    }
5  }
```

Auparavant l'objet *listener* est déclaré dans la partie principale du programme, à partir de l'objet `JMSConsumer` :

```
1  Listener myListener = new Listener();
2  consumer.setMessageListener(myListener);
```

Les communications par topic et par queue se passent de la même manière : un message est envoyé puis reçu avec les mêmes primitives mais sur des objets destination différents. Cependant la sémantique de consommation des messages est différente en fonction du type de destination. Sur les queues, la sémantique est semblable à celle des sockets : un message envoyé attend dans la queue jusqu'à être consommé. Puisque la queue ne peut être accédée que par un seul consommateur, le message est délivré à lui seul. Pour les topics la sémantique est un peu plus complexe.

FIGURE 3.7 – Réception sur un topic

En mode normal, c'est-à-dire en faisant appel à la fonction `createConsumer` seuls les messages qui sont envoyés sur le topic pendant la connexion du consommateur sont reçus par ce dernier. Sur la figure 3.7 une application JMS reçoit des messages en provenance d'un topic. Six messages sont successivement envoyés sur le topic concerné mais l'application n'est pas toujours à l'écoute du topic : elle ne crée de consommateur que sur les deux intervalles de temps. Puisque l'application n'est pas connectée lorsque les messages *M3* et *M4* arrivent sur le topic, celle-ci ne les reçoit pas.

Pour le cas où une application souhaite recevoir tous les messages, même si elle ne possède pas de consommateur connecté au topic, JMS offre la possibilité de créer un consommateur durable à l'aide de la fonction `createDurableConsumer`. Un nom est associé à la création d'un consommateur durable pour permettre de retrouver cette connexion lors de sa réouverture. Ce nom doit être associé au `JMSContext` tout de suite après sa création. Un consommateur durable nommé `MySub` sera donc, par exemple, créé avec le code suivant :

```
1  JMSContext contex = connectionFactory.createContext();
2  context.setClientID( id );
3  ...
4  String subName = "MySub";
5  JMSConsumer consumer = context.createDurableConsumer(dest, subName);
```

Le comportement du consommateur durable est illustré par la figure 3.8. Nous voyons sur cette figure qu'il y a une différence entre la création de la session vers le topic et la création du consommateur. Après avoir fermé un consommateur il est possible d'en créer

un autre, avec le même nom, qui reprend la sessions précédente et permet donc de ne pas perdre de message. Par exemple, dans le cas de la figure 3.8, le message *M2* sera reçu par le second consommateur.

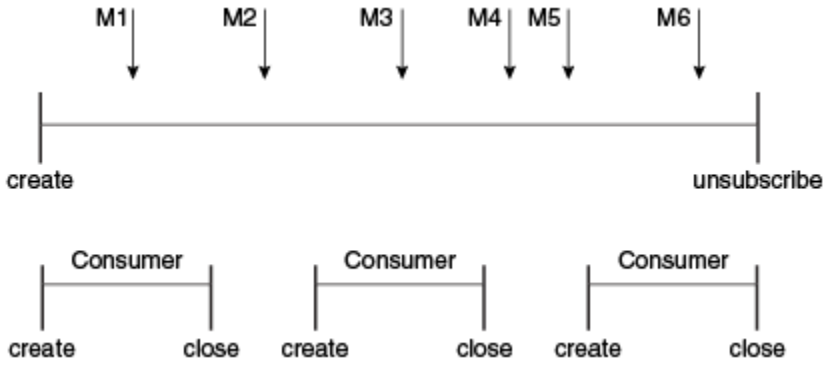


FIGURE 3.8 – Consommateur durable sur un topic

Dans le cas d'un consommateur durable, puisque la connexion est maintenue, il est nécessaire de pouvoir l'arrêter lorsque cela devient nécessaire. Cela est fait avec la fonction `unsubscribe` du contexte. Par exemple de la manière suivante :

```
1 consumer.close();
2 context.unsubscribe(subName);
```

3.2.4 Messages JMS

L'objet de communication lui-même est un objet `Message`. Un message est composé de trois parties.

— L'en-tête : un ensemble de champs possédant des valeurs déterminés par le client ou le fournisseur.
— Les propriétés : des champs spécifiés par l'application et servant aux sélecteurs.
— Le corps : la partie données proprement dite, qui peut être de 6 types différents).

Les en-têtes des messages JMS sont composés de champs qui permettent au provider et/ou au récepteur d'avoir des information additionnelles pour gérer les messages. La table suivante détaille les principaux champs de l'entête, leur nom, leur contenu et qu'elle est l'entité qui remplit ce champ.

Champ	Description	Défini par
JMSDestination	Destination du message.	provider
JMSPriority	Priorité du message.	provider
JMSMessageID	Identifiant du message.	provider
JMSTimestamp	La date de réception du message par le provider.	provider
JMSCorrelationID	L'identifiant du message relatif.	application
JMSReplyTo	Destination à utiliser pour la réponse.	application
...		

D'autres champs existent (JMSDeliveryMode et JMSExpiration par exemple) mais ce sont les plus communs.

Le corps des messages JMS peuvent prendre six types différents. Chacun de ces types détermine un type de message différent, classe fille de la classe principale `Message`. Le tableau suivant décrit les six types de messages JMS.

Type	Description
TextMessage	Une chaîne de caractères.
MapMessage	Un ensemble de paires clé-valeur.
BytesMessage	Un flot de données binaires.
StreamMessage	Un flot de données primitives.
ObjectMessage	Un objet sérialisé.
Message	Ne contient rien.

Le type `Message` ne contient donc pas de données. Les données ne sont disponibles que dans ses classes filles. Il peut néanmoins être utilisé comme un outil de synchronisation, pour signaler la fin d'une transmission par exemple.

Le type de message peut servir à la réception à différencier les messages reçus et les traitements associés. Le code suivant donne un exemple de traitement pour un échange d'un message de type `TextMessage`.

```
1 TextMessage message = context.createTextMessage();
2 message.setText(msg_text);      // msg_text is a String
3 context.createProducer().send(message);
```

```
1 Message m = consumer.receive();
2 if (m instanceof TextMessage) {
3     String message = m.getBody(String.class);
4     System.out.println("Reading message: " + message);
5 } else {
6     // Handle error or process another message type
7 }
```

Lorsqu'un consommateur peut potentiellement recevoir différent types de messages il est alors nécessaire de savoir identifier le type du message reçu. Dans l'exemple précédent nous voyons donc que, après avoir reçu un message dans la classe générale `Message`, le consommateur peut identifier son type à partir du test `m instanceof TextMessage`. Ce test lui permet d'identifier les messages de différents types et d'appliquer le traitement correspondant.

Pour finir les messages peuvent posséder des propriétés, fixées par l'émetteur. L'exemple suivant associé une propriété *NewsType* au message. Cette information vient en complément du contenu du message.

```
1  message = context.createTextMessage(``Hello World !'');
2  message.setStringProperty("NewsType", "Sports");
```

Ces propriétés peuvent ensuite être utilisées à la réception pour filtrer les messages à l'aide de sélecteur. La syntaxe utilisée pour les sélecteurs repose sur un sous-ensemble de SQL92. Le code suivant donne un exemple de sélection des messages à l'arrivée. Parmi l'ensemble des messages disponibles sur une destination, seuls ceux pourtant une propriétés *NewsType* initialisée à *Sports* ou *Opinion* seront reçus.

```
1  String sel = "NewsType = 'Sports' OR NewsType = 'Opinion'";
2  JMSConsumer consumer = context.createConsumer(sel);
```

À noter qu'il est possible (mais est-ce souhaitable ?) de ne réduire les échanges de messages qu'au strict minimum, comme dans l'exemple simplifié suivant :

```
1 String message = "This is a message";
2 context.createProducer().send(dest, message);
```

```
1 String message = consumer.receiveBody(String.class);
```

3.2.5 Modes de session

Nous avons présenté la classe `JMSContext` précédemment et évoqué le fait qu'il est possible, à partir de la session associée à cette classe, de définir des modes de communication associés. Le mode par défaut est un mode qui permet une communication dite *ACKNOWLEDGE* dans laquelle nous nous contentons d'envoyer et de recevoir les messages. L'intérêt d'utiliser un bus de messages peut également être de profiter des services qui sont proposés par le bus. De ce fait la session propose plusieurs modes de communication avec des fonctionnalités supplémentaires par rapport aux échanges que nous avons étudiés jusque là. Pour configurer un nouveau mode de communication associé à la session, il est nécessaire de créer un nouveau `JMSContext`, à partir du contexte initial. Par exemple, la création d'un nouveau contexte en mode transactionnel se fait de la manière suivante :

```
1    ConnectionFactory cf = (ConnectionFactory) new ActiveMQConnectionFactory();
2    JMSContext context = cf.createContext();
3    JMSContext jmsc = context.createContext( JMSContext.SESSION_TRANSACTED );
```

Les différents modes proposés par les sessions sont les suivants :

AUTO_ACKNOWLEDGE Un accusé de réception est transmis automatiquement à chaque réception. C'est le mode par défaut que nous avons utilisé jusqu'ici.

CLIENT_ACKNOWLEDGE Un accusé de réception est transmis par le consommateur. Pour cela il fait appel à la méthode `acknowledge` de l'objet message. Il n'a pas a rédiger un message explicite.

DUPS_OK_ACKNOWLEDGE Un accusé de réception est transmis paresseusement. Ce mode est moins coûteux que les modes précédents mais, suivant le provider, il peut arriver qu'un message soit reçu plusieurs fois.

TRANSACTED Spécifie que la session est en mode transactionnel.

Le mode transactionnel permet de garantir qu'un ensemble de message est reçu soit au complet soit pas du tout, comme pour des transactions dans une base de données. Il permet de grouper l'envoi et la réception de messages en les terminant par l'appel de la méthode `commit`. Exemples :

— Un client envoie deux messages mais ne souhaite pas qu'en cas de défaillance (du client), seul le premier ne soit reçu par le fournisseur.

— Un client reçoit deux messages mais ne souhaite pas qu'en cas de défaillance (du client), seul le premier ne soit reçu et traité.

Le mode permet de gérer la tolérance aux pannes dans l'échange des messages.

En cas d'erreur, une transaction peut être annulée par l'appel de la méthode `rollback`.

Attention, il ne faut pas grouper un envoi et une réception qui soit en attente d'une réponse du message envoyé. Cela provoque un interblocage.

En lien avec la tolérance aux pannes il est également possible d'associer des propriétés de persistance aux messages. Seul un message persistant survit à la défaillance du fournisseur. La durabilité est une propriété liée au souscripteur alors que la persistance est une propriété liée au fournisseur.

3.2.6 Cas d'erreur

Pour permettre une gestion correcte des erreurs au sein d'une application utilisant JMS des classes d'exceptions sont définies. En voici quelques unes :

```
1  IllegalStateException // file creee sur un TopicSession
2  InvalidDestinationException // mauvaise destination
3  InvalidSelectorException // probleme de syntaxe du selecteur
4  ...
```

3.3 Synthèse

Pour conclure cette partie nous pouvons énoncer les caractéristiques clés des MOM.

— Couplage faible entre les participants (ils n'ont pas tous besoin d'être connectés au même moment).

— Fiabilité (`durable`, `persistent`).
— Performance (distribution de la charge, débit important).

Un des aspects que nous n'avons pas développé ici est le protocole de communication utilisé pour échanger les données dans les MOM. A bas niveau, l'échange de données repose la plupart du temps sur le protocole TCP/IP. Mais TCP ne gère pas le codage des données, or, si nous voulons pouvoir faire interagir des programmes différents, éventuellement codés dans des langages différents il est alors nécessaire de représenter les données dans un format compatible avec ces langages. Le provider ActiveMQ propose ainsi plusieurs protocoles différents pour encoder les données : openwire, qui est son protocole natif, ampq, un protocole largement utilisé par d'autres MOM et qui tend à devenir un standard, ou stomp, un protocole qui encode les données sous forme de texte.

À titre indicatif voici l'historique de JMS :
— JMS 1.0 (2001).
— JMS 1.1 (2002).
— JMS 2.0 inclus dans Java EE 7 (avril/mai 2013).
— JMS 2.1 inclus dans Java EE 8 (2016).

Comme nous l'avons déjà évoqué il existe plusieurs implémentations pour JMS. Celles que nous utilisons pour les exercices est ActiveMQ Artemis [5].

Autres implémentations connues :

JORAM JMS 1.1, JMS 2.0 (depuis 2014).

OpenMQ (GlassFish) Implémentation de référence de Oracle.

3.4 Webographie

À partir des liens ci-dessous vous pourrez avoir des éléments de références à partir des documentations officielles et compléter vos connaissances en JMS.

Documentations officielles :
— Documentation de l'interface JMS 2.0
— Spécification JMS 2.0
— Tutoriel sur les concepts
— Tutoriel sur les exemples

Discussions sur des questions liées :
— Real world use of JMS/message queues?
— What is Java Message Service (JMS) for?
— ActiveMQ Use Cases
— Why should I use JMS and not RMI+Queue?
— Message Queue vs. Web Services?

5. https://activemq.apache.org/components/artemis/

Chapitre 4

Les services Web

4.1 Généralités

Les services Web sont un ensemble de technologies qui permettent l'invocation de méthodes distantes présentes sur des systèmes distribués et hétérogènes. La communication et l'échange de données entre les applications sont basés sur les standards Web tels que HTTP et XML.

Le standard W3C définit les services web comme étant des applications auto-descriptives, modulaires et faiblement couplées. Elles fournissent un modèle de programmation et de déploiement d'applications, basé sur des normes, et s'exécutant au travers de l'infrastructure web.

Les principaux avantages des services web par rapport aux technologies comme RPC ou RMI sont :

— L'interopérabilité entre langages, plates-formes et systèmes d'exploitation,
— L'utilisation des standards et protocoles ouverts,
— L'utilisation des formats texte dans les échanges de données, facilitant ainsi la réalisation des interactions faiblement couplées,
— La flexibilité et l'extensibilité.

Il existe deux grandes familles de services Web :

— SOAP,
— REST.

Nous voyons dans ce chapitre d'abord les Web Services SOAP, les plus anciens historiquement, avant d'aborder les Web Services Rest.

4.2 Architecture des services Web SOAP

Les services Web de type SOAP font un usage intensif de XML, des espaces de nom XML et des schémas XML. Ces technologies font la force des services Web car cela permet leur utilisation par des clients et des serveurs hétérogènes. XML est notamment utilisé pour stocker et organiser les informations des requêtes et des réponses mais aussi pour décrire le service Web.

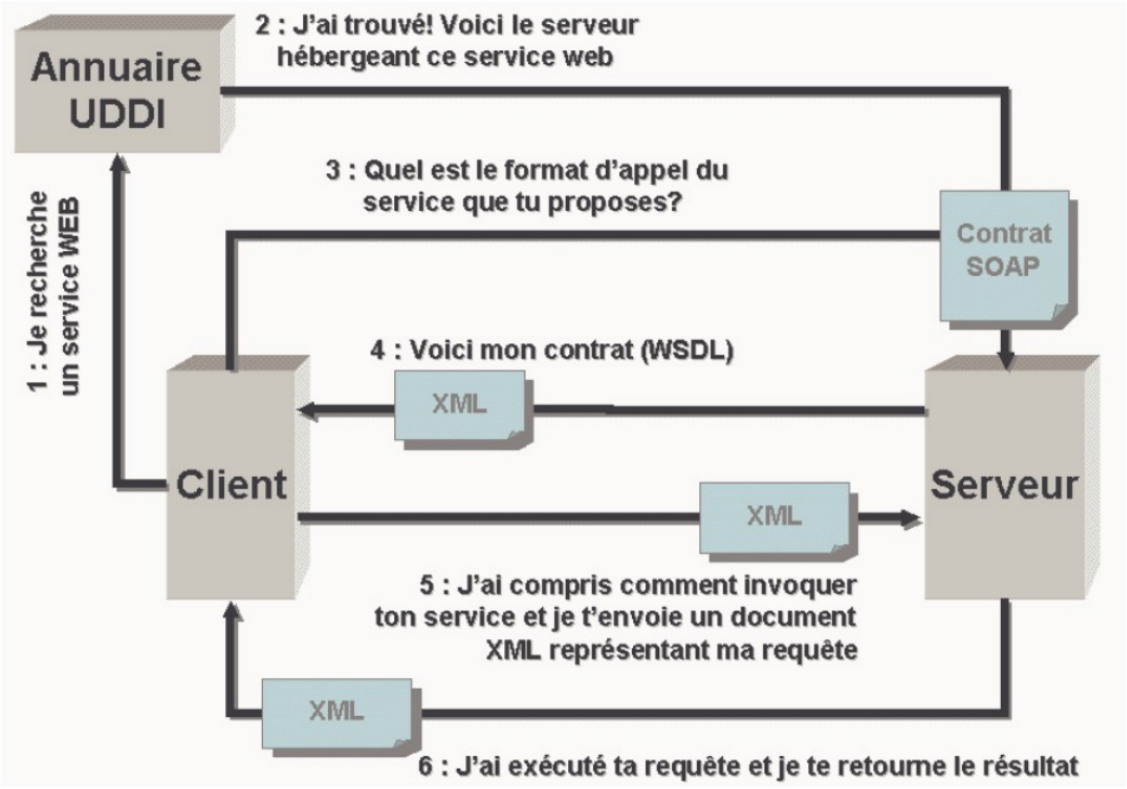


FIGURE 4.1 – Scénario général d'utilisation d'un service Web

La figure illustre les étapes à suivre pour mettre en œuvre un service Web.

1. Le client envoie une requête à l'annuaire de services pour trouver le service Web dont il a besoin,
2. L'annuaire recherche le service Web approprié et renvoie le serveur qui détient le service,
3. Le client envoie alors une requête à ce serveur pour obtenir le contrat de normalisation de ses données,
4. Le serveur envoie sa réponse en XML (forme établie par WSDL),
5. Le client peut maintenant rédiger sa requête pour traiter les données dont il a besoin,
6. Le serveur exécute la requête du client et renvoie sa réponse sous la même forme normalisée.

L'architecture des services Web SOAP se base sur trois composants standardisés :
— Un protocole de transport, pour envoyer et de recevoir des informations formatées sous forme de messages sur Internet, (**SOAP**),
— Un protocole pour décrire le service qui permet entre autres d'énumérer les méthodes disponibles, (**WSDL**),
— Un protocole de localisation pour la découverte de services, (**UDDI**).

4.2.1 SOAP

SOAP (ancien acronyme de Simple Object Access Protocol) est un protocole de RPC (Remote Procedure Call) orienté objet, basé sur XML, il permet la transmission de messages entre objets distants. Le transfert se fait le plus souvent via le protocole HTTP, mais peut également se faire par un autre protocole comme SMTP. SOAP est une spécification non propriétaire, il n'est pas lié à un protocole particulier et il n'est pas non plus lié à un système d'exploitation ni à un langage de programmation.

Un message SOAP est un document XML composé (figure 4.2) d'une enveloppe avec un en-tête et le corps du message.

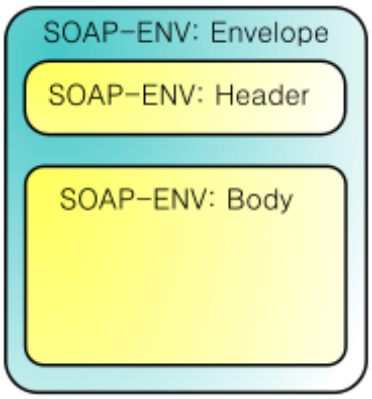


FIGURE 4.2 – Format du message SOAP

L'enveloppe est obligatoire dans un message SOAP, elle permet de spécifier la version de SOAP utilisée et les règles d'encodage (sérialisation et désérialisation). L'enveloppe englobe l'entête et le corps (figure 4.3).

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
   soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
<!– Entête et corps-->
</soap:Envelope>
```

FIGURE 4.3 – Exemple d'une enveloppe SOAP

L'en-tête est optionnel et peut être utilisé pour compléter les informations nécessaires à une requête, plus généralement des métadonnées relatives au message. L'en-tête reconnaît plusieurs attributs spécifiques :

- `actor`, indique le destinataire de l'information. Cela permet de viser une application intermédiaire spécifique via une URL,
- `mustUnderstand`, prend une valeur booléenne et indique si le traitement de l'élément est obligatoire ou non (figure 4.4),
- `encodingStyle`, spécifie les règles d'encodage s'appliquant à l'élément.

```
<m:Trans xmlns:m="http://www.w3schools.com/transaction/"
soap:mustUnderstand="1">234</m:Trans>
```

FIGURE 4.4 – Exemple d'élément d'un en-tête SOAP

Le corps (body) doit contenir en envoi, le nom de la méthode appelée, ainsi que les paramètres à appliquer. En retour, il contiendra une réponse à sens unique ou un message d'erreur détaillé en cas d'exception.

Ce dernier message utilise le sous-élément `Fault`, qui lui-même dispose de quatre sous-éléments possibles :

- `faultcode`, identifiant l'erreur par un code.
- `faultstring`, une explication de l'erreur.
- `faultactor`, l'origine de l'erreur.
- `detail`, des détails spécifiques.

Par exemple, pour invoquer la méthode distante `getPrice()` (pour obtenir le prix d'un produit) avec le paramètre `Apple`, le corps du message SOAP contient les informations suivantes :

```
<?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>
      <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
        <m:Item>Apples</m:Item>
      </m:GetPrice>
   </soap:Body>
</soap:Envelope>
```

FIGURE 4.5 – Message SOAP contenant une requête

Si tout se passe bien, le serveur répond avec le message SOAP suivant :

```
<?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>
    <m:GetPriceResponse xmlns:m="http://www.w3schools.com/prices">
      <m:Price>1.90</m:Price>
   </m:GetPriceResponse>
</soap:Body>
</soap:Envelope>
```

FIGURE 4.6 – Message SOAP contenant une réponse

En cas d'erreur, le serveur répond avec un message SOAP suivant :

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
             soap:encodingStyle="http://schemas.xmlsoap.org/soap/
                                 encoding/">
<soap:Body>
<soap:Fault>
        <faultcode>soap:MustUnderstand</faultcode>
        <faultstring>Mandatory Header error.</faultstring>
        <faultactor>http://www.wrox.com/heroes/endpoint.asp</faultactor>
        <detail>
           <w:source xmlns:w="http://www.wrox.com/">
              <module>endpoint.asp</module>
              <line>203</line>
           </w:source>
        </detail>
     </soap:Fault>
  </soap:Body>
</soap:Envelope
```

FIGURE 4.7 – Message SOAP contenant des erreurs

Une des forces de SOAP est de permettre l'interopérabilité entre différentes plate-formes. Il est donc important d'avoir des règles de codage des types de données, pour qu'elles soient encodées/décodées sans difficultés. On distingue deux types de données :

— simple : une chaîne de caractère par exemple,
— composé : structures ou tableaux.

Dans le cas où des données binaires devraient transiter (comme une image par exemple), il est également possible d'envoyer un message SOAP avec attachement en utilisant un message MIME (Multimedia Internet Mail Extension).

4.2.2 WSDL

Pour consommer un service Web, le client a besoin de sa description détaillée avant de pouvoir interagir avec lui. WSDL (Web Service Description Langage) fournit cette description dans un document XML (figure 4.8). cette description comprend une définition du service, les types de données utilisés notamment dans le cas de types complexes, les opérations utilisables, le protocole utilisé pour le transport et l'adresse d'appel.

WSDL est un contrat entre un client et un serveur qui fait état :

— des spécifications d'interfaces qui décrivent toutes les méthodes publiques,
— des spécifications relatives aux types de données de messages mis en oeuvre dans les questions-réponses,
— des informations liées au protocole de transport utilisé,
— des informations d'adresse permettant de localiser le service décrit.

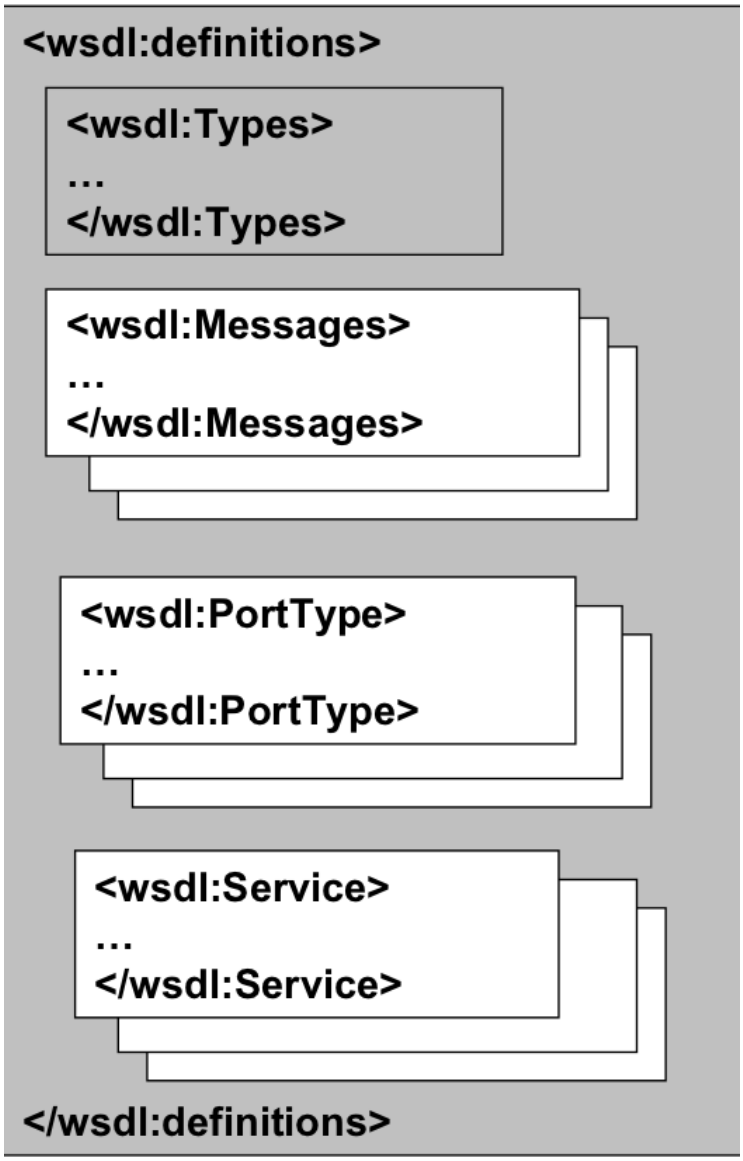


FIGURE 4.8 – Structure d'un document WSDL

L'élément types décrit tous les types de données utilisés entre le client et le serveur. Pour cela il emploi les schéma XML.

L'exemple suivant (figure 4.9) définit deux types : `symbol` de type `string` et `price` de type `float`.

```
<types>
  <schema targetNamespace=http://advocatemedia.com/GetStockQuote.xsd
   xmlns="http://www.w3.org/2000/10/XMLSchema">
<element name="StockQuoteRequest">
   <complexType>
  <all>
    <element name="symbol" type="string"/>
  </all>
 </complexType>
</element>

<element name="StockQuoteResponse">
  <complexType>
  <all>
    <element name="price" type="float"/>
  </all>
</complexType>
</element>
</schema>
</types>
```

FIGURE 4.9 – Exemple définition de types

l'élément message décrit les données échangées entre le client et le serveur. Cela peut être comparé aux paramètres d'un méthode distante.

Par exemple (figure 4.10), la méthode `GetStockQuote` prend en paramètre un élément de type `StockQuoteRequest` et retourne un élément de type `StockQuoteResponse`

```
<message name="GetStockQuoteRequest">
   <part name="body" element="myXSD:StockQuoteRequest"/>
</message>

<message name="GetStockQuoteResponse">
   <part name="body" element="myXSD:StockQuoteResponse"/>
</message>
```

FIGURE 4.10 – Exemple de l'élément message

l'élément portType définit les opérations du service Web et les messages impliqués (de type input, output ou fault). Peut être comparé à une interface Java.

Par exemple l'interface `GetStockQuotePort` (figure 4.11) contient une seule méthode (opération) `GetStockQuote`

```
<portType name="GetStockQuotePort">
  <operation name="GetStockQuote">
    <input message="myns:GetStockQuoteRequest"/>
    <output message="myns:GetStockQuoteResponse"/>
 </operation>
</portType>
```

FIGURE 4.11 – Exemple de l'élément portType

l'élément binding définit le format des messages et le protocole utilisé par chaque type de port. C'est l'implémentation de l'interface (figure 4.12).

```
<binding name="GetStockQuoteBindingName"
  type="GetStockQuotePort ">
  <soap:binding style="rpc"
  transport=" http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetStockQuote">
  <soap:operation soapAction="http://advocatemedia.com/GetStockQuote"/>
    <input> <soap:body use="literal"/> </input>
    <output> <soap:body use="literal"/> </output>
  </operation>
</binding>
```

FIGURE 4.12 – Exemple de l'élément binding

l'élément port définit un point de terminaison identifié de manière unique par la combinaison d'une adresse internet et d'une liaison.

```
<service name="AdvocateMediaGetStockQuotes">
 <documentation>Simple Web Service to Retrieve
a stock quote</documentation>
<port name="GetStockQuotePort" binding="myns:GetStockQuoteBindingName">
<soap:address location="http://advocatemedia.com/GetStockQuote"/>
</port>
</service>
```

FIGURE 4.13 – Exemple de l'élément service

l'élément service contient une collection d'élément port (figure 4.13).

4.2.3 UDDI

L'annuaire des services UDDI (Universal Description, Discovery and Integration) est un standard pour la publication et la découverte des informations sur les services Web. Il permet à une entreprise de s'inscrire dans l'annuaire, d'y enregistrer et de publier ses services Web. Il est alors possible d'accéder à l'annuaire [1] et de rechercher un service particulier.

Une entrée du répertoire UDDI est constituée d'un fichier XML qui décrit une entreprise et les services qu'elle offre. Chaque entrée du répertoire UDDI est constituée de trois parties :

1. Les pages blanches décrivent l'entreprise qui offre le service : nom, adresse, contacts, etc,
2. Les pages jaunes comportent les catégories industrielles,
3. Les pages vertes décrivent l'interface vers le service avec suffisamment de détail pour qu'il soit possible d'écrire une application permettant d'utiliser le service Web.

L'UDDI n'est pas un élément indispensable à la mise en œuvre des services web comme peut l'être XML, WSDL ou SOAP.

4.3 Développement de services Web SOAP

Plusieurs plates-formes permettent de développer des services Web, on peut citer par exemple : GSOAP (C/C++), Guzzle (PHP), JAX-WS, Apache CFX, Axis2 (Java).

Nous utilisons l'API JAX-WS (intégré dans Java à partir de la version 1.6) pour développer un simple service Web. JAX-WS est basée sur les annotations ce qui simplifie le développement des services web.

1. le site Web officiel de UDDI est http://uddi.xml.org

4.3.1 Développer le service

Le développement d'un service Web avec JAX-WS se fait en plusieurs étapes :

— Développer une interface publique : les méthodes exposées (distantes) et leurs paramètres,
— Implémenter l'interface : le code métier du service,
— Publier et déployer le service.

La première étape consiste à décorer une interface Java (listing suivant) avec l'annotation obligatoire (`@WebService`) qui peut prendre les paramètres suivants :

— **endpointInterface** : l'interface définissant le service
— **name** : le nom du webservice
— **portName** : le nom du port
— **serviceName** : le nom du service
— **targetNamespace** : URL pour les namespaces
— **wsdlLocation** : la localisation du WSDL

```
1  package ws;
2
3  import javax.jws.WebMethod;
4  import javax.jws.WebService;
5  import javax.jws.soap.SOAPBinding;
6  import javax.jws.soap.SOAPBinding.Style;
7
8  //Service Endpoint Interface
9  @WebService
10 @SOAPBinding(style = Style.RPC) //optional
11 public interface HelloWorld{
12
13         @WebMethod //optional
14         String getHelloWorldAsString(String name);
15
16 }
```

Toutes les méthodes publiques de l'interface sont disponibles dans le service Web. L'annotation `@WebMethod` est optionnelle, elle permet par exemple de personnaliser le nom de la méthode et ces paramètres dans le WSDL généré.

L'annotation optionnelle `@SOAPBinding` permet de spécifier le type des messages échangés et le type d'encodage utilisé. On distingue deux de styles de message :

— `style=style.RPC` : les messages SOAP contiennent les noms des méthodes et leurs paramètres,
— `style=style.Document` : les messages SOAP ne contiennent pas les noms des méthodes, un document XML en entrée et en retour est employé.

Par défaut JAX-WS utilise le style document pour les échanges.

La deuxième étape consiste à implémenter l'interface :

```
1  package ws;
2
3  import javax.jws.WebService;
4
5  //Service Implementation
6  @WebService(endpointInterface = "ws.HelloWorld")
7  public class HelloWorldImpl implements HelloWorld{
8
9          @Override
10         public String getHelloWorldAsString(String name) {
11                 return "Hello World JAX-WS " + name;
12         }
13 }
```

Et enfin, publier et déployer l'application localement :

```
1
2  package ws;
3
4  import javax.xml.ws.Endpoint;
5
6  //Endpoint publisher
7  public class HelloWorldPublisher{
8
9          public static void main(String[] args) {
10            Endpoint.publish("http://localhost:9999/ws/hello",
11                             new HelloWorldImpl());
12     }
13
14 }
```

Le WSDL généré est disponible à l'adresse suivante : http://localhost:9999/ws/hello?wsdl

Note : Dans la pratique, les services Web sont souvent déployés dans les serveurs d'applications tels que Tomcat et JBoss.

```
- <definitions targetNamespace="http://ws/" name="HelloWorldImplService">
    <types/>
  - <message name="getHelloWorldAsString">
      <part name="arg0" type="xsd:string"/>
    </message>
  - <message name="getHelloWorldAsStringResponse">
      <part name="return" type="xsd:string"/>
    </message>
  - <portType name="HelloWorld">
    - <operation name="getHelloWorldAsString">
        <input message="tns:getHelloWorldAsString"/>
        <output message="tns:getHelloWorldAsStringResponse"/>
      </operation>
    </portType>
  - <binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
    - <operation name="getHelloWorldAsString">
        <soap:operation soapAction=""/>
      - <input>
          <soap:body use="literal" namespace="http://ws/"/>
        </input>
      - <output>
          <soap:body use="literal" namespace="http://ws/"/>
        </output>
      </operation>
    </binding>
  - <service name="HelloWorldImplService">
    - <port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
        <soap:address location="http://localhost:9999/ws/hello"/>
      </port>
    </service>
  </definitions>
```

FIGURE 4.14 – Structure du document WSDL généré

JAX-WS et les types complexes

JAX-WS permet de transférer les données complexes comme par exemple les objets Java (paramètres d'une méthode) ou des données binaires (fichiers, images).

Les objets comme paramètres les objets Java sont sérialisés/désérialisés en XML grâce à l'annotation `@XmlRootElement`, et sont ensuite englobés dans le corps de l'enveloppe SOAP.

Le listing suivant illustre le paramètre `Message` d'un service.

```
1  package ws;
2  import javax.xml.bind.annotation.XmlRootElement;
3
4  @XmlRootElement
5  public class Message {
6      private String message;
7      private int id;
8
9      //getters and setters
10
11 }
```

Les données binaires sont envoyées en attachement dans l'enveloppe SOAP. L'annotation `@MTOM` disponible dans l'implémentation du service permet de transformer les paramètres de type binaire de méthodes distantes en pièce jointe.

Voici un exemple de transfert d'image :

```
1  package ws;
2  ...
3  @MTOM
4  @WebService(endpointInterface = "ws")
5  public class ImageServerImpl implements ImageServer{
6
7          @Override
8          public Image downloadImage(String name) {
9          ...
10         }
11 }
```

4.3.2 Consommer le service : le client

Client Java

L'utilisation de JAX-RPC est similaire à celle de RMI : le code du client appelle les méthodes distantes (les services) à partir d'un objet local nommé `stub`. Cet objet se charge de dialoguer avec le serveur et de coder/ décoder les messages SOAP échangés.

L'outil `wsimport` de JAX-WS permet de générer automatiquement le `stub` de l'application en se basant sur le WSDL fourni par le serveur.

— Se placer dans un nouveau répertoire, pour ne pas mélanger le code du serveur avec celui du `stub` généré,

— Lancer la commande suivante :

```
1 wsimport -keep http://localhost:9999/ws/hello?wsdl
```

Cette commande génère les classes du `stub` et les compile automatiquement. Nous nous basons sur le code généré pour créer la classe principale du client :

```
1  import ws.*;
2
3  public class HelloClient {
4
5  public static void main(String args[]) {
6
7       //get service
8       HelloWorldImplService service = new HelloWorldImplService();
9       //get port or interface
10      HelloWorld helloWorld = service.getHelloWorldImplPort();
11      //invovcate service
12      String response = helloWorld.getHelloWorldAsString("Plop");
13      System.out.println("response: "+ response);
14  }
15 }
```

Le code consiste à instancier l'objet service `HelloWorldImplService`, de récupérer l'interface (le port) `HelloWorld` et enfin d'appeler la méthode du service Web `getHelloWorldAsString()`.

Client Python

Nous utilisons le paquet python `suds`[2] pour invoquer le service Web précédemment déployé :

```
1  from suds.client import Client
2
3  url = "http://localhost:9999/ws/hello?wsdl"
4
5  client = Client(url)
6
7  response = client.service.getHelloWorldAsString("Plop")
8
9  print response
```

Et c'est aussi simple que cela !

2. Pour l'insatller : `pip install suds-community`

4.4 Web Services REST

Les services WEB de type SOAP sont à l'heure actuelle considérés comme des web services lourds. Ils supposent la mise en place des schémas SOAP pour pouvoir être mis en œuvre et, une fois le schéma fixé, sont relativement peu souples aux évolutions. Une autre technologie de web service s'est développée depuis qui donne plus de souplesse dans l'évolution du service. Il s'agit des web services REST, pour Representational State Transfer. Il ne s'agit pas à proprement parlé d'une technologie mais plutôt d'un style architectural. Dans les web services REST il n'y a pas de contrat explicite, de type interface, comme nous l'avons vu pour JavaRMI et les web services SOAP. L'idée fondatrice qui est à la base des web services REST est que, plutôt que d'utiliser des protocoles de communication complexes, l'utilisation de HTTP permet de mettre en œuvre tous les échanges entre un client et un service.

Dans les web services REST, les ressources sont identifiées par des URI (Uniform Resource Identifier) et peuvent être soit des données, comme un fichier ou un résultat de requête sur une base de données, soit des fonctionnalités, c'est-à-dire des procédures à appliquer. Ces ressources sont exploitées avec un protocole de communication sans état, généralement HTTP. Les clients et les serveurs échangent des représentations de ressources en utilisant une interface et un protocole normalisé. L'approche légère offerte par les web services REST sert de base aux architectures micro-services où une application est décomposée en services de petite taille, communiquant entre eux à travers des requêtes HTTP. L'intérêt de cette architecture réside surtout dans sa souplesse due à sa modularité, par exemple pour faire de l'intégration continue.

Ce cours s'est largement inspiré, pour sa partie sur Jax-RS, de celui de Mickaël Baron, accessible sur https://mbaron.developpez.com/cours/soa/jaxrs. Pour pratiquer il y a quelques exemples de programmes dans la partie exercices mais vous pouvez aussi trouver des exemples de programmes, qui vous permettront de vous faire une idée plus précise de la programmation Jax-RS à l'adresse http://www.mkyong.com/tutorials/jax-rs-tutorials/.

4.4.1 Les URI

Les URI proposent un modèle de désignation largement utilisé dans le web. Leur objectif est de définir des identificateurs pour désigner les ressources de toute nature, aussi bien physiques (un serveur) que logiques (un fichier). Ils font l'objet d'une spécification du Network Working Group de l'IETF (Internet Task Force) identifiée sous l'appellation RFC 2396.

Les URI sont des chaînes de caractères, ce qui fait qu'ils peuvent être utilisés dans des environnements hétérogènes. L'uniformité de leur représentation leur permet également de décrire des ressources hétérogènes. Suivant le type de l'URI une interprétation sémantique de la chaîne d'identification est possible.

Les URI regroupent deux sous-types d'identificateurs, les URN et les URL.

Les URL, ou Uniform Resource Locator, sont bien connus des personnes qui utilisent le

web puisqu'ils comprennent les chemins d'accès aux pages ou ressources accessibles. En fait un URL est constitué deux deux parties : un protocole d'accès et un identificateur de la ressource. Ainsi une adresse internet est précédée du protocole `HTTP`. Nous savons par exemple que `http://moodle.univ-fcomte.fr` nous permet d'accéder, à travers le protocole `http` à la plate-forme moodle de l'université de Franche-comté. De la même manière certains URL permettent d'envoyer un mail (par exemple : `mailto@patrick.dupont@monmail.com`), de télécharger un fichier (par exemple : `ftp://leserveur/chemin/nomfichier`).

Les URN, Uniform Resource Name, permettent de définir une identification unique, globale et persistance pour une de ressources. Dans notre contexte, unique signifie que deux ressources ne porteront jamais le même nom, que la désignation est reconnue sur l'ensemble du réseau internet et persistante signifie cette désignation peut avoir une durée de vie au delà de l'instance d'une ressource. Les URN suivants sont des exemple d'URN qui peuvent identifier un disque dur (`uuuid:85dsfjdpj269708307363` , UUID signifie Universally Unique IDentifier), un livre (`ISBN:0-395-36341-1`) ou un document émis par l'IETF (`ietf:rfc:2141`). On remarquera que, comme pour les URL, les URI peuvent contenir une référence explicite à un système d'identification existant (par exemple : uuid, isbn, doi, ...).

4.4.2 Principales caractéristiques des web services REST

La définition d'un service web de type REST suppose les caractéristiques suivantes :

- Identification de ressources via des URI : le web service expose un ensemble de ressources (via leur URI) pour identifier les cibles de l'interaction avec les clients
- Interface uniforme : les ressources sont manipulées exclusivement avec 4 méthodes PUT, GET, POST et DELETE
- Messages auto-descriptifs : les ressources sont séparées de leur représentation (dans divers formats, HTML, XML, texte, PDF, JPEG, JSON, ou d'autres)
- Pour les interactions sans état : les requêtes sont auto-suffisantes, l'état est maintenu par la représentation de la ressource
- Pour les interactions à états (statfull) : les interactions à états se basent sur un transfert d'état explicite. Cela simplifie à mise en œuvre du service mais augmente le volume de la communication

4.4.3 Identification des ressources par les URI

Un URI identifie une ressource de manière unique, c'est-à-dire qu'à un instant donné, une seule ressource sera identifiée par cet URI, ce qui nous permet d'être sûr que la ressource ne sera pas confondue avec une autre. Une ressource peut cependant avoir plusieurs URI pour l'identifier et la représentation de la ressource peut évoluer avec le temps. Par exemple `http://localhost:8080/banque/comptes/snoopy/PEL` et `http://localhost:8080/banque/comptes/sn` peuvent être deux identification distinctes d'un compte en banque. Nous dirons ici que `/PEL` et `/1` sont les identifiants primaires d'une même ressource

Les ressources peuvent être hiérarchiques, c'est-à-dire qu'une ressource peut être une collection de ressources. Par exemple `/banque/comptes/snoopy` est une ressource de type

collection qui regroupe tous les comptes en banque de Snoopy.

Dans le développement d'une application, c'est donc aussi la manière dont nous allons la concevoir qui va en faire un web service REST. Pour cela il faut que l'identification soit basée sur des URI. Ces identificateurs sont adaptés à la mise en place d'une identification hiérarchique. De ce fait si les ressources de l'application s'y prêtent il sera facile d'utiliser cette désignation. Par exemple, tout les données stockées dans les tables d'une base peuvent facilement être identifiées de manière hiérarchique.

4.4.4 Ressource, opération et méthode HTTP

Dans une architecture REST, l'exécution d'une procédure sur une ressource s'appelle une opération. Une ressource peut subir 4 opérations identifiée par l'acronyme CRUD pour Create, Retrieve, Update, Delete, chacune correspondant à un type de requête, ou méthode HTTP :

- **Create** : pour la création de la ressource, qui se traduit par une méthode `POST`,
- **Retrieve** : pour la lecture du contenu de la ressource, qui se traduit par une méthode `GET`,
- **Update** : pour la mise à jour, qui se traduit par une méthode `PUT`,
- **Delete** : pour supprimer la ressource, qui se traduit par une méthode `DELETE`,

Rappel :

Nous rappelons que le protocole HTTP repose, pour l'exécution de ces requêtes, sur un échange en mode requête-réponse à l'initiative du client. Ce qui signifie qu'à chaque requête du client le serveur envoie un message de réponse. Le format de ces messages est normalisé par le protocole HTTP.

Les requêtes sont composées d'une ligne de commande, d'une entête de requête et d'un corps de message. La ligne de commande contient le *VERB*, c'est-à-dire le type de commande (par ex : POST, GET, etc.), l'URI et la version du protocole HTTP. dans notre cas, l'URI donné dans cette ligne de commande sert à la description de la ressource REST sur laquelle l'opération sera appliquée. L'entête de requête est utilisée pour fournir des informations sur le client et le corps pour fournir les données associées à la requête.

Les requêtes sont composées d'une ligne de d'état, d'une entête et d'un corps de message. La ligne d'état contient un code de réponse (par ex : 404 quand l'URI donné dans la requête n'est pas trouvé) et la version du protocole HTTP utilisé. L'entête de réponse est utilisée pour donner des informations complémentaires et les données de la réponse sont transmises dans le cors du message.

4.4.5 Représentation des données

Le type des données transmises dans une requête HTTP est fixé par le type MIME associé (audio, vidéo, image, texte, etc.). Du point de vue des requêtes de web services nous avons généralement un ensemble de données structurées à envoyé, comme dans toutes les communications que nous avons vues dans le cours. Le codage de ces données se fait

principalement en mode texte avec des formats tels que XML ou plus souvent dans le cas des web services REST, Json.

Nous avons vu dans la partie sur les web services SOAP l'utilisation du XML. Nous nous intéressons maintenant à la représentation Json.

Json, pour JavaScript Object Notation, est une représentation implicite de données en mode texte normalisée par l'IETF sous la référence RFC 7159. La représentation choisie par Json rend les données moins verbeuses que le XML ce qui en facilite la lecture. Cette représentation reste néanmoins lisible par l'humain et simple à programmer.

Le code suivant donne un exemple de représentation Json pour la définition d'une structure de donnée associée à une personne :

Listing 4.1 – Exemple de représentation Json

```
{
  "firstName": "Patrick",
  "lastName": "Dupond",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "22 rue des lilas",
    "city": "Venexia",
    "state": "WonderLand",
    "postalCode": "1234567"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "444 555 666"
    },
    {
      "type": "office",
      "number": "111 222 333"
    }
  ]
}
```

Une représentation implicite signifie que le type de données échangé entre l'émetteur et le récepteur est convenu. Ainsi, dans la programmation socket, nous sommes obligés de savoir, à la réception d'une donnée de quel type elle est pour l'utiliser correctement. À l'inverse une donnée explicite embarque sa définition, c'est-à-dire aussi bien le type de la donnée (entier, char, string, ...) que sa structuration. En Json nous ne définissons pas nom plus les types de données transmis. Par exemple dans le code Json donné au listing 4.1 certains données sont des chaînes de caractères, d'autres sont des entiers, des structures de données, etc. À la réception d'une donnée telle que celle-là il est nécessaire de connaître le type de donnée associé à l'âge pour l'interpréter correctement. Notons que Json, en incluant le titre des champs de la structure permet néanmoins de les expliciter et ainsi de pourvoir, par exemple, retrouver un champ même sans avoir la même organisation de la structure de donnée.

1. Les chaînes de caractères : ce sont des séquences de 0 ou plus caractères Unicode, obligatoirement délimitées par des guillemets.
2. Les nombres : ce sont des nombres décimaux signés qui peuvent contenir une part fractionnelle ou élevée à la puissance (notation E). Le Json ne fait aucune distinction entre un entier et un flottant.
3. Les booléens : qui peuvent prendre les valeurs true ou false.
4. Le type null : c'est une valeur vide, utilisant le mot clé null.
5. Les tableaux : qui sont des listes ordonnées de valeurs dont le type peut-être différent. Un tableau commence par un crochet ouvrant (`[`) et se termine par un crochet fermant (`]`). À l'intérieur d'un tableau les données sont séparées par des virgules
6. Les objets : ce sont des collections non ordonnées de paires nom/valeur, dans lesquelles les noms sont des chaînes de caractères qui sont séparées de la valeur par le caractère `:`. Un objet commence par une accolade ouvrante (`{`) et se termine par une accolade fermante (`}`).

Dans l'exemple du listing 4.1, nous avons donc un objet composé des paires `firstName`, `lastName`, ..., d'un objet `adresse`, lui-même composé des paires `streetAddress` et d'un objet `phoneNumbers`, lui-même composé d'un tableau d'objets numéro de téléphone.

L'utilisation du format Json est relativement simple en Java car nous disposons de classes facilitant le codage de classes Java (au moins de leurs données associées) et le décodage d'objets Json pour l'instantiation de classes. Parmi les packages existants nous pouvons citer Jackson, qui a la réputation d'être rapide, ou encore Gson, développé par Google.

Le listing 4.2 montre l'utilisation du package Gson en Java.

Listing 4.2 – Exemple d'utilisation du package Gson

```
1 import com.google.gson.Gson;
2
3 Gson gson = new Gson();
4
5 MyInData mid = new MyINData(45, 0.10, ``Hello'');
6 String toJsonString = gson.toJson( mid );
7 ...
8 MyInData fromJsonString = gson.fromJson( jsonString,
9                                 MyInData.class );
```

4.5 Développement des web services REST

Plusieures plateformes aident au développement des web services REST en prenant en charge l'analyse et le traitement des requêtes côté serveur et les réponses côté client. Nous pouvons citer Spring, Jersey, Apache CXF, etc. Pour illustrer le développement de

services web nous avons choisi de rester dans l'environnement Java et nous nous reposons sur Jersey [3], l'implémentation de référence fournie par Oracle.

4.5.1 Environnement de développement JAX-RS

L'interface Java pour implémenter des web services RESTFull s'appelle JAX-RS. Elle est définie par la spécification JSR 339 [4]. La version courante de cette spécification, sur laquelle nous nous reposons, est la version 2.0 et elle fait partie intégrante de la spécification Java pour les entreprises, Java Enterprise Edition ou EE, depuis la version 6 de Java EE. Elle ne décrit que le côté serveur de la mise en œuvre des web services RESTFull. La partie client étant plus généralement prise en charge directement au sein d'un navigateur. Dans cette interface, et comme pour les spécifications des autres interfaces d'accès à des services dans Java, le développement des services repose sur l'utilisation de classes Java et d'annotations. Cette interface Java est fournie actuellement par différentes implémentations : Jersey, l'implémentation de référence, Apache CXF, RESTEasy, fournie par le serveur JBoss, ou encore RESTlet un des premiers framework implémentant les services web REST en Java.

L'impléméntation de référence [5] des Jax-RS s'appelle JERSEY [6]. La version actuelle est la 2.26 et répond aux spécification JAX-RS 2.0. Elle est intégrée dans les framework Glassfish et Java EE et les outils de développement associés sont par exemple proposés dans l'environnement de développement intégré (IDE) NetBeans [7]. Pour la construction d'une partie serveur avec l'outil de construction de projet Maven il faut utiliser les éléments suivants :

— groupId : com.sun.jersey
— artifactId : jersey-server
— Version : 2.26

Il faut noter que l'approche proposée ici, si elle est légère car elle se contente du minimum et n'implique pas le lancement d'un framework, n'est pas pour autant la plus simple. L'approche pédagogique choisie vise à se concentrer sur les spécificités des services web Rest, sans les mélanger avec d'autres fonctionnalités spécifiques aux framework de services. Cependant, si vous avez l'occasion d'utiliser des frameworks tels que Spring vous verrez que le développement et le déploiement sont simplifiés. Par ailleurs, les services web REST ne sont pas tous implémentés en Java et qu'ils peuvent dans d'autres environnements et/ou langages (Python, Php, C++, C#, ...).

3. jersey.java.net
4. La spécification peut être trouvée sur le site officiel jcp.org.
5. Pour les JSR (Java Specification Request), Oracle fournit des implémentations de référence qui servent de référence pour les autres développement implémentant la spécification.
6. jersey.java.net
7. netbeans.org

4.5.2 Développement JAX-RS

Le développement de services web avec JAX-RS est basé sur des POJO (Plain Old Java Object) en utilisant, sur ces objets, des annotations spécifiques à JAX-RS[8]. Il n'y a alors pas de description requise dans des fichiers de configuration. Seule la configuration de la Servlet «JAX-RS» est requise pour réaliser le pont entre les requêtes HTTP et les classes Java annotées. Le service web REST est ensuite déployé dans une application web.

Contrairement aux services web étendus il n'y a pas de possibilité de développer un service REST à partir du fichier de description WADL (web Application Description Language) et seule l'approche Bottom/Up est possible pour créer un service. Il faut donc, dans l'ordre, créer et annoter un POJO, le compiler puis le déployer au sein de la plateforme web qui l'exposera et le tester. Pour la validation il est néanmoins possible d'accéder au document WADL associé au service car ce fichier peut-être généré automatiquement par JAX-RS. Il est alors accessible à partir de l'adresse : *http ://host/context/application.wadl*.

Listing 4.3 – Exemple simple de service REST

```
1  @Path("helloworld")
2  public class HelloWorldResource {
3
4      public static  final String MESSAGE = "Hello World!";
5
6      @GET
7      @Produces("text/plain")
8      public String getHello() { return MESSAGE; }
9  }
```

Le listing 4.3 donne un exemple simple d'objet Java annoté pour définir une ressource JAX-RS. Nous pouvons identifier dans ce code trois annotations : `@Path`, `@GET` et `@Produces`. Ces annotations complètent l'objet pour en faire un objet invocable à distance, à travers une méthode HTTP, et accessible à l'aide d'un URI.

Il faut noter que le nom donné à la classe n'est pas utilisé dans l'appel du client qui ne verra que le contexte `helloworld` donné par l'annotation `@Path`. Ce nom n'a donc d'importance qu'à l'intérieur de la définition du serveur. Il en va de même pour le nom de la méthode qui ne répond qu'à une requête `GET` de la part du client.

L'identification et le traitement des différentes méthodes HTTP se fait par l'annotation des méthodes avec les annotations `@GET`, `@POST`, `@PUT`, `@DELETE` et `@HEAD` sur les méthodes de l'objet. A noter ici qu'il n'y a aucun lien obligatoire entre l'annotation de la méthode, le type de requête HTTP, et le travail effectivement réalisé par la méthode. Cette correspondance est laissée à l'appréciation du programmeur.

8. Les annotations sont des ajouts qui complètent le code des objets. Elle sont préfixées par le caractère @

4.5.3 L'annotation `@Path`

L'annotation `@Path` permet d'indiquer que l'objet doit être exposé comme un service, qu'il va répondre à des requêtes REST. Cette annotation précise le chemin d'accès à la ressource donc l'URI qui sera utilisé pour y accéder. A noter que cet URI est relatif au contexte dans lequel va être déployé l'objet. Cette annotation peut également marquer des méthodes de la classe. Dans ce cas, l'URI d'accès spécifique à la méthode est la concaténation de l'expression du `@Path` de la classe et du `@Path` de la méthode.

La valeur d'URI associée à l'annotation `@Path` a la structure suivante :

$$http://localhost:8080/restws/helloworld$$

Où la première partie `localhost:8080` donne le serveur et le port du service. `restws` donne le contexte dans lequel le service web est exposé. La ressource racine de l'objet est donnée par `helloworld`, elle peut elle-même être composée d'un chemin d'accès si les objets sont organisés de manière hiérarchique. Le chemin d'accès à une ressource ne décrit pas forcément directement l'objet. L'URI peut aussi définir l'accès à une méthode. Ce serait, par exemple, le cas si la méthode `getHello` avait été annotée.

L'annotation `@Path` peut prendre la forme d'un template URI. C'est-à-dire un URI incluant des variables qui seront substituées au moment de l'exécution, lors d'un appel à une opération. Il ne s'agit pas à proprement parlé de paramètres de fonction car leur porté est limitée aux URIs. Cela permet une construction dynamique des URI. Dans la définition du `@Path`, ces templates sont donnés entre accolades. Par exemple, dans le listing 4.4, un nom d'utilisateur est donné en variable pour composer le template URI. La valeur est ensuite obtenue dans l'objet à l'aide de l'annotation `@PathParam`. Ainsi dans le listing la variable `user` est obtenue en paramètre de la fonction `getHello` et utilisée pour composer le message de retour.

Listing 4.4 – Exemple de templateURI

```
1  @Path("helloworld")
2  public class HelloWorldResource {
3      public static final String MESSAGE = "Bonjour ";
4
5      @GET
6      @Path("/{user}")
7      @Produces("MediaType.TEXT\_PLAIN")
8      public String getHello(@PathParam("user") String user) {
9          return MESSAGE + user;
10     }
11
12 }
```

L'appel à ce web service se fait alors avec l'URI suivant :

$$http://localhost:8080/restws/helloworld/mon\_nom$$

A noter que, dans le listing 4.4, nous avons fait porter la variable par la méthode `getHello`, elle aurait également pu être donnée dans l'annotation `@Path` de l'objet lui-même.

Il est également possible d'utiliser des expressions régulières pour engendrer des templates URI. Les expressions régulière sont définies de la même manière que ce que nous utilisons dans une commande shell telle que `egrep`. L'expression permet alors de filtrer les requêtes pour ne réaliser que les requêtes dont le modèle correspond à l'expression régulière donnée dans le template URI. Par exemple l'annotation `@Path("users/username:[a-zA-Z][a-zA-Z]*_[0-9]")` limite les URI à ceux qui contiennent une chaîne composée d'au moins un caractère (minuscule ou majuscule) puis un sous-tiret puis un chiffre, après la partie `users/`.

L'annotation `@Path` permet également de déléguer une partie du traitement d'une requête à un autre objet. La méthode qui délègue s'appelle alors un *sub-resource locator*. Cette méthode doit respecter un certain nombre de conditions :

— Elle doit être annotée avec l'annotation `@Path`,
— Elle ne doit porter aucune des annotations `@GET`, `@POST`, `@PUT` ou `@DELETE`,
— Elle doit retourner une sous-ressource, donc un type d'objet.

Dans ce cas le développement suit un schéma classique et il n'y a pas d'obligation de placer une ressource racine. Le listing 4.5 donne un exemple où la méthode `getHello` est un sub-resource locator qui fait appel à l'objet `MessagePersonnel`.

Listing 4.5 – Exemple de sub-resource locator

```
1  @Path("helloworld")
2  public class HelloWorldResource {
3    public static final String MESSAGE = "Bonjour ";
4
5    @Path("personnalise")
6    public String getHello() {
7      return new MessagePersonnel(MESSAGE);
8    }
9
10 }
11
12 public class MessagePersonnel {
13
14   @GET
15   @Path("{user}")
16   public String getHello(@PathParam("user") String user) {
17     return MESSAGE + user;
18   }
19
20 }
```

A noter qu'une méthode sub-resource locator supporte le polymorphisme (retourne des sous types).

4.5.4 Paramètres de requêtes

Il est possible de passer des paramètres aux requêtes HTTP. Vous avez probablement déjà vu des URI contenant des `?` et des `&` comme dans l'exemple suivant ;

$$https://www.google.com/search?client=ubuntu\&channel=fs\&q=truc$$

Dans cet URI la partie search contient des paramètres qui sont passés à une méthode. Cette formulation des paramètres n'est pas spécifique au web services REST mais ceux-ci l'utilisent.

Dans Jax-RS les annotations sont utilisées pour extraire les paramètres de l'URI et injecter le contenu dans les objets. Les annotation utilisables pour extraire les paramètres sont les suivantes :

- @DefaultValue : valeur par défaut
- @PathParam : extraire les valeurs des Template Parameters
- @QueryParam : extraire les valeurs des paramètres d'une requête
- @FormParam : extraire les valeurs des paramètres d'un formulaire
- @HeaderParam : extraire les paramètres de l'en-tête
- @CookieParam : extraire les paramètres des cookies
- @Context : extraire les informations liées aux ressources de contexte

Le listing 4.6 montre un exemple de passage de deux paramètres de type chaîne à la fonction `getHello`.

Listing 4.6 – Exemple de passage de paramètres

```
1  @Path("helloworld")
2  public class HelloWorldResource {
3      public static final String MESSAGE = "Bonjour ";
4
5      @GET
6      @Path("{user}")
7      @Produces("MediaType.TEXT_PLAIN")
8      public String getHello(
9                  @DefaultValue("Monsieur") @QueryParam("name") String name,
10                 @DefaultValue("X") @QueryParam("surname") String surname {
11
12         return MESSAGE + name + surname;
13     }
14
15 }
```

Nous voyons ici la principale annotation utilisée pour le passage de paramètres : `@QueryParam`. Elle permet d'extraire de l'URI les paramètres dont le nom est donné en paramètre de l'annotation et de les affecter aux paramètres annotés de la fonction. Par exemple, dans le listing 4.6, la méthode `getHello` prend deux paramètres `name` et `surname`, ils sont affectés aux paramètres `name` et `surname` qui sont annotés par `@QueryParam`.

L'invocation du service par le client prendra alors la forme :

$$/helloworld/queryparameter?name = Foo\&surname = Bar$$

Dans cette invocation la valeur des paramètres est donnée explicitement à partir du nom du paramètre, leur ordre dans l'URI invoquée n'a donc pas d'importance.

Il est possible d'affecter des valeurs par défaut aux paramètres, au cas où le client ne donnerait pas l'ensemble des paramètres attendus. L'affectation par défaut se fait par l'annotation `@DefaultValue`. Ainsi, dans le listing 4.6, le paramètre `name` de la méthode `getHello` prend la chaîne `MONSIEUR` par défaut si aucun paramètre `name` n'est donné dans l'URI d'invocation du service. Cela permet de renforcer la robustesse du service. Néanmoins, si aucun paramètre n'est passé dans l'URI à l'invocation, la méthode est bien réalisée mais avec des valeurs non garanties.

Jusqu'ici les paramètres que nous avons utilisés sont des chaînes de caractères mais il est bien sûr possible de donner d'autres types de données en paramètre ou en retour des fonctions. La représentation des données passées en paramètres ou retournées par les fonctions est donnée par les annotations `@Consumes` et `@Produces`. L'annotation `@Consumes` spécifie le ou les types MIME accepté(s) par une méthode d'une ressource et l'annotation `@Produces` spécifie le ou les types MIME produit(s) par une méthode d'une ressource. Par exemple nous avons utilisé `@Produces(MediaType.TEXT_PLAIN)` dans le listing 4.6.

Quelques remarques sur les définitions de type MIME :

- une même méthode peut accepter ou retourner des données avec des types MIME différents. Donc il peut y avoir plusieurs types dans une même annotation
- les annotations peuvent être portées par une classe ou une méthode mais l'annotation sur une méthode surcharge celle de la classe
- Si aucune annotation n'est associée à une méthode celle-ci accepte tous les types
- La liste des types possibles est donnée par la classe `MediaType`.

A partir du MediaType, JAX-RS effectue des opérations de sérialisation et dé-sérialisation vers un type Java spécifique. Par exemple, les types suivants sont traduits de la manière suivante :

- */* → byte[] (InputStream, File)
- text/* : → String
- text/xml, application/xml, application/*+xml → JAXBElement
- application/x-www-form-urlencoded → MultivalueMap<String,String>

Les listings 4.7, 4.8 et 4.9 donnent des exemples de passage de paramètres de différents types pour un service web librairie.

Listing 4.7 – Exemple d'utilisation des paramètres `InputStream`

```
1  @Path("/contentbooks")
2  public class BookResource {
3    @PUT
4    @Path("inputstream")
5    public void updateContentBooksInputStream(InputStream is) throws IOException {
6      byte[] bytes = readFromStream(is);
7      String input = new String(bytes);
```

```
8         System.out.println(input);
9       }
10      private byte[] readFromStream(InputStream stream) throws IOException {
11        ByteArrayOutputStream baos = new ByteArrayOutputStream();
12        byte[] buffer = new byte[1000]; int wasRead = 0;
13        do {
14          wasRead = stream.read(buffer);
15          if (wasRead > 0) { baos.write(buffer, 0, wasRead); }
16        } while (wasRead > -1);
17        return baos.toByteArray();
18      }
19      @Path("inputstream")
20      @GET
21      @Produces(MediaType.TEXT_XML)
22      public InputStream getContentBooksInputStream() throws FileNotFoundException {
23        return new FileInputStream("c:\\example.xml");
24      }
25    }
```

Listing 4.8 – Exemple de requête et réponse avec un fichier

```
1   @Path("/contentbooks")
2   public class BookResource {
3     @Path("file")
4     @PUT
5     public void updateContentBooksFile(File file) throws IOException {
6       byte[] bytes = readFromStream(new FileInputStream(file));
7       String input = new String(bytes);
8       System.out.println(input);
9     }
10    @Path("file")
11    @GET
12    @Produces(MediaType.TEXT_XML)
13    public File getContentBooksFile() {
14      File file = new File("example.xml");
15      return file;
16    }
17  }
```

Listing 4.9 – Exemple de requête et réponse avec un String

```
1   @Path("/contentbooks")
2 public class BookResource {
3 @Path("string")
4 @PUT
5 public void updateContentBooksString(String current) throws IOException {
6 System.out.println(current);
7 }
8 @Path("string")
9 @GET
```

```
@Produces(MediaType.TEXT_XML)
public String getContentBooksWithString() {
  return "<?xml version=\"1.0\"?>" +
         "<details>Ce livre est une introduction sur la vie" +
         "</details>";
}
```

Les types que nous avons vu jusque là sont des types MIME standardisés. Il est également possible d'utiliser des types personnalisés en s'appuyant sur la spécification JAXB (JSR 222). Cette spécification définit la correspondance entre les classes Java et leur représentation XML ou XML Schema. L'avantage est de pouvoir manipuler des objets Java sans passer par une représentation abstraite XML. Pour cela, la classe est annotée pour décrire la correspondance entre XML Schema et les informations de la classe : `XmlRootElement`, `XmlElement`, `XmlType`, ...

Jax-RS met en œuvre la spécification JAXB et prend donc en charge la sérialisation et la dé-sérialisation de classes annotées par `@XmlRootElement`, `@XmlType` en les « enveloppant » dans un objet `JAXBElement`. Le contenu d'une requête et d'une réponse peut alors être représenté par du XML ou du JSON. La manipulation de types personnalisés oblige de préciser dans le service le type MIME à traiter et à retourner. Les définitions correspondantes doivent être faites par des annotations `@Produces` et `@Consumes` avec les valeurs MIME `text/xml`, `application/xml` ou `application/*+xml` pour une représentation XML et `application/json` pour JSON.

Les listing 4.10 et 4.11 montrent la création d'un objet JAXB et son passage en paramètre.

Listing 4.10 – Exemple de définition d'un objet JAXB

```
@XmlRootElement(name = "book")
public class Book {
  protected String name;
  protected String isbn;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getIsbn() {
    return isbn;
  }
  public void setIsbn(String isbn) {
    this.isbn = isbn;
  }
  public String toString() {
    return name;
  }
}
```

Listing 4.11 – Exemple de passge d'un objet JAXB en paramètre

```
1  @Path("/contentbooks")
2  public class BookResource {
3    @Path("jaxbxml")
4    @Consumes("application/xml")
5    @PUT
6    public void updateContentBooksWithJAXBXML(Book current) throws IOException {
7      System.out.println("Name: " + current.getName() +
8                         ", ISBN: " + current.getIsbn());
9    }
10   @Path("jaxbxml")
11   @GET
12   @Produces("application/xml")
13   public Book getContentBooksWithJAXBXML() {
14     Book current = new Book();
15     current.setIsbn("123-456-789");
16     current.setName("Harry Toper");
17     return current;
18   }
19   @Path("jaxbxml")
20   @Consumes("application/xml")
21   @POST
22   public void updateContentBooksJAXBElmt(JAXBElement<Book> currentJAXBElemnt) {
23     Book current = currentJAXBElemnt.getValue();
24     System.out.println("Name: " + current.getName() +
25                         ", ISBN: " + current.getIsbn());
26   }
27 }
```

4.5.5 Réponses

La réponse au client utilise également le protocole HTTP. Elle retourne donc le code standard qui donne l'état, ou status, du traitement. Pour les réponses sans erreur les codes de réponse possibles vont de 200 à 399. Par exemple, le status 200 signifie `OK`, le contenu est non vide alors que le status 204 signifie `No Content`, le contenu est vide. Lorsque la réponse est une erreur le statut peut prendre les valeurs de 400 à 599. Par exemple, le status 404 signifie `Not Found`, la ressource n'est pas trouvée, le status 406 signifie `Not Acceptable`, le type MIME n'est pas supporté et le status 405 signifie `Method Not Allowed`, la méthode HTTP n'est pas supportée,

La spécification JAX-RS facilite la construction de réponses pour choisir un code de retour, fournir des paramètres dans l'en-tête, retourner une URI, etc. Les réponses sont faites à partir de la classe `Response`. Les méthodes de la classe `Response` sont les suivantes (liste non exhaustive) :

- `ResponseBuilder created(URI location)` : Modifie la valeur de Location dans l'en-tête, à utiliser pour une nouvelle ressource créée
- `ResponseBuilder notModified()` : Statut à « Not Modified »

— `ResponseBuilder ok()` : Statut à « Ok »
— `ResponseBuilder serverError()` : Statut à « Server Error »
— `ResponseBuilder status(Response.Status)` : défini un statut particulier défini dans Response.Status

Les informations de ces méthodes sont obtenues par des méthodes statiques retournant des objets `ResponseBuilder` qui supposent l'utilisation du patron de conception `Builder`.

Les méthodes de la classe `ReponseBuilder` sont les suivantes (liste non exhaustive) :

— `Response build()` : crée l'instance de l'objet `Response`,
— `ResponseBuilder entity(Object value)` : modifie le contenu du corps,
— `ResponseBuilder header(String, Object)` : modifie un paramètre de l'en-tête.

Ces méthodes sont utilisées, à partir de la création de l'objet `ReponseBuilder` par l'invocation des méthodes de la classe `Response`, pour compléter les informations contenues dans la réponse. Le listing 4.12 donne un exemple de composition d'une réponse à une opération `GET`.

Listing 4.12 – Exemple de réponse

```
1  @Path("/contentbooks")
2  public class BookResource {
3    ...
4    @Path("response")
5    @GET
6    public Response getBooks() {
7      return Response
8      .status(Response.Status.OK)
9      .header("param1", "Bonjour")
10     .header("param2", "Hello")
11     .header("server", "keulkeul")
12     .entity("Ceci est le message du coprs de la reponse")
13     .build();
14   }
15 }
```

Ici l'objet `Response` est créé par l'appel à la méthode `status`. Cette méthode retourne un objet `ReponseBuilder`, sur lequel est invoquée la méthode `header` pour compléter le contenu de l'entête. Le retour de cette fonction est lui-même un objet `ReponseBuilder` qui est utilisé pour compléter l'entête puis le corps, jusqu'à la création elle-même de la réponse.

4.5.6 UriBuilder

La création d'une réponse suppose fréquemment la construction d'un URI. La spécification JAX-RS propose des outils pour la construction des URI. La classe `UriBuilder` permet de construire des URI complexes, soit à partir de la la classe `UriInfo` pour construire des URI relatifs au chemin de la requête, soit *"From scratch"*, c'est-à-dire construire un nouvel

URI. La logique d'utilisation de la classe `UriBuilder` est identique à celle de la classe `ResponseBuilder`.

Les principales méthodes pour obtenir un objet `UriBuilder` à partir d'un objet `UriInfo` sont les suivantes :

- `UriBuilder getBaseUriBuilder()` : relatif au chemin de l'application
- `UriBuilder getAbsolutePathBuilder()` : relatif au chemin absolu (base + chemins)
- `UriBuilder getRequestUriBuilder()` : relatif au chemin absolu incluant les paramètres

Les principales méthodes de la classe `UriBuilder` sont les suivantes :

- `URI build(Object... values)` : construit une URI à partir d'une liste de valeurs pour les Template Parameters
- `UriBuilder queryParam(String name, Object...values)` : ajoute des paramètres de requête
- `UriBuilder path(String path)` : ajout un chemin de requête
- `UriBuilder fromUri(String uri)` : nouvelle instance à partir d'une URI
- `UuiBuilder host(String host)` : modifie l'URI de l'hôte

Listing 4.13 – Exemple de création d'URI

```
1  @Path("/contentbooks")
2  public class BookResource {
3    ...
4    @Consumes("application/xml")
5    @POST
6    @Path("uribuilder")
7    public Response createBooksFromURI(Book newBook, @Context UriInfo uri) {
8      URI absolutePath = uri
9      .getAbsolutePathBuilder()
10     .queryParam("param1", newBook.getName())
11     .path(newBook.getIsbn())
12     .build();
13     return Response.created(absolutePath).build();
14   }
15 }
```

Le listing 4.13 montre un exemple de création d'URI absolu à partir de l'objet `UriInfo` obtenu par l'annotation `@Context`. L'objet `UriInfo` permet d'obtenir un objet `UriBuilder` initialisé avec le chemin absolu d'invocation de la méthode par la fonction `getAbsolutePathBuilder`. Ce chemin est ensuite complété par un paramètre de requête, ici le nom du livre, et un path, donné par le numéro ISBN. Pour finir, l'URI est créé par la méthode `build`.

4.5.7 Déploiement

Le service, une fois conçu doit être déployé au sein du support d'exécution choisi, c'est-à-dire un serveur d'application ou un serveur HTTP. Pour un déploiement en ser-

veur d'application, les applications JAX-RS sont déployées sous le format WAR. Il faut alors déclarer les classes ressources dans le fichier de déploiement `web.xml`. Deux types de configurations sont autorisées : une sous-classe de la classe `Application` ou de la classe `Servlet`, fournies par l'implémentation Jax-RS.

Comme nous l'avons dit précédemment l'objet de ce cours est de vous donner un aperçu de ce que sont les services web REST. Nous nous contentons donc de voir leur développement en Java, sans aborder la partie serveur d'application (application framework) qui, si elle est le cas le plus fréquent, demanderait d'ajouter une part conséquente à ce cours. Le déploiement abordé ici, même s'il est moins fréquemment utilisé, permet de limiter notre propos au cadre des services web REST et d'avoir un environnent de test simple, ne nécessitant pas l'installation d'un framework. Pour la même raison nous avons limité cette partie de déploiement au strict minimum vous permettant de tester des services web REST facilement.

Dans le contexte de déploiement, la classe `Application` fournit une implémentation à vide du contexte d'exécution. Il faut ensuite compléter les classes ressources à l'aide des fonctions :

— `Set<Class> getClasses()` : classes des ressources
— `Set<Object> getSingletons()` : instances des ressources

Pour des raisons de simplicité nous lui préférons donc la classe `ResourceConfig` qui fournit une implémentation complète et peut être initialisée directement à partir de la classe de notre service. Ensuite JAX-RS peut être utilisée avec Java 6 sans avoir à déployer une application web, à la différence de JAX-WS. L'implémentation JERSEY nécessite néanmoins l'ajout d'un serveur WEB en mode embarqué, nous utilisons ici le serveur jetty et les containers de servlet.

Listing 4.14 – Exemple d

```
1  import org.eclipse.jetty.server.*;
2
3  public class App {
4    private static final URI BASE_URI =
5                                  URI.create("http://localhost:8080/restws/");
6    public static final String ROOT_PATH = "/helloworld/*";
7
8    public static void main(String[] args) {
9
10     try {
11
12       final ResourceConfig resourceConfig =
13                               new ResourceConfig(HelloWorldResource.class);
14
15       Server server = new Server(port);
16       ServletContextHandler ctx =
17             new ServletContextHandler(ServletContextHandler.NO_SESSIONS);
18       ctx.setContextPath("/");
19       server.setHandler(ctx);
20       ctx.addServlet(new ServletHolder(new ServletContainer(
```

```
21                                                  resourceConfig ) ), ROOT_PATH);
22             server.start();
23
24             System.out.println("Server started on " + BASE_URI + ROOT_PATH));
25
26         } catch (IOException ex) {
27             Logger.getLogger(App.class.getName()).log(Level.SEVERE, null, ex);
28         }
29     }
30 }
```

Le listing 4.14 donne le code de déploiement du service `HelloWorldResource`. Il ne comprends que la création de la ressource à travers l'objet `ResourceConfig`, la création et le démarrage du serveur web Jetty. Il peut être repris et adapté au besoin du déploiement et test d'un service.

Ce type de déploiement suppose tout de même d'avoir accès à l'ensemble des classes nécessaires pour la mise à disposition du services. ces classes sont regroupées sous la forme de fichiers jar qui vous sont données dans la partie exercices.

Dans les anciennes versions de Java, pour la compilation de notre application, comme pour son exécution, il sera nécessaire de préciser le `CLASSPATH` dans lequel se trouve ces fichiers jar. En supposant que ces fichiers se trouvent dans le répertoire `jarsServer` les commandes de compilation et d'exécution pourrait être les suivantes :

```
javac -cp ".:../jarsServer/*" helloworld/server/*.java

java -cp ".:../jarsServer/*" helloworld.server.App
```

Pour les versions plus récentes (> 8) il est préférable d'avoir recours à des outils de compilation tels que maven pour faciliter la recherche et l'utilisation de dépendences. Dans les corrections de l'exercice sur le "ContactBook" vous pourrez trouvez un ficher `pom.xml"` utilisable pour compiler et déployer un service.

4.5.8 Développement du client

Les clients de services web REST sont des clients web classiques puisqu'il ne vont générer que des requêtes des méthodes HTTP. Ainsi un browser peut facilement jouer le rôle du client pour accéder à vos services. Il suffit par exemple de taper dans la barre d'URL l'URI que nous souhaitons accéder pour réaliser un appel `GET` sur le service. L'invocation manuelle ne convient cependant pas à tous les cas et le développement d'un programme client est souvent souhaité. Ce développement peut se faire avec un grand nombre de bibliothèques, l'utilisation d'une API cliente ne suppose pas que les services web soient développés avec JAX-RS. On peut par exemple utiliser .NET, PHP, python, etc. Par contre, pour des invocations contenant des objets Java, il peut-être plus simple d'utiliser une bibliothèque Java pour éviter d'avoir à faire une transformation explicite des objets en XML.

Comme cela a été dit précédemment, la spécification Jax-RS ne se préoccupe pas de la partie client. L'existence de la partie client dépend donc de l'implémentation de la spécification choisie. Dans notre cas, Jersey propose une API cliente facilitant l'échange sous la forme de requêtes et réponse, que nous détaillons brièvement dans la suite. Elle nous permettra d'écrire des programmes client pour l'accès à nos services déployés.

Listing 4.15 – Exemple de client pour un web service REST

```
1  public class HelloClient {
2
3      public static void main(String[] args) {
4          try {
5
6              Client client = ClientBuilder.newBuilder().newClient();
7              WebTarget target = client.target("http://localhost:8080/restws");
8              target = target.path("helloworld");
9
10             Invocation.Builder builder = target.request("text/plain");
11             Invocation resp = builder.buildPost(Entity.text("Bonjour"));
12             resp.invoke();
13
14             builder = target.request("text/plain");
15             Invocation response = builder.buildGet();
16             String msg = response.invoke(String.class);
17             System.out.println("Recu " + msg);
18
19         } catch (Exception ex) {
20             ex.printStackTrace();
21         }
22     }
23 }
```

Le listing 4.15 donne un exemple de client pour le service web `helloworld` vu dans la partie serveur. La première partie du code consiste à créer le client puis à générer l'URI sur lequel nous allons invoquer le service. C'est à partir de cet URI que sera constitué la requête au service. Nous avons ensuite deux appels au service, un en mode Post et l'autre en mode Get. La construction d'un appel repose ici sur les objets `Invocation` et `Invocation.Builder` et consiste principalement à fournir les données nécessaires à ces objets pour construire l'appel.

Pour réaliser une invocation il est donc nécessaire de partir d'un URI. Cet URI peut être obtenu à partir de la classe `WebTarget`. Cette classe possède plusieurs méthodes permettant de constituer la base de l'URI (méthode `target`), de la compléter (méthode `path`), d'ajouter un paramètre (méthode `queryParam`) et d'obtenir l'objet `Invocation.Builder` qui sert à préparer la requête (méthode `request`).

La classe `Invocation.Builder` sert à préparer la requête. Les principales méthodes de cette classe sont les suivantes :

— `buidDelete`, `buidGet`, `buidPost`, `buidPut` pour construire chacun des types de requêtes

— `header` pour ajouter des informations dans l'entête de la requête

La classe `Invocation` servent à réaliser l'appel au service une fois la requête constituée. Les principales méthodes de cette classe sont les suivantes :

— `invoke`, qui fait un appel synchrone au service. Si la méthode invoquée doit retourner une valeur, le type de la valeur doit être donné en paramètre de l'appel. C'est, par exemple, le cas de l'appel de la méthode `GET`.

— `submit`, qui fait un appel asynchrone au service. Comme pour la méthode `invoke`, le type du retour doit être spécifié.

Chapitre 5

Conclusion

Comme nous l'avons vu depuis le début de ce cours, la communication est un élément primordial des systèmes distribués. Les propriétés associées au support de communication déterminent en effet les possibilités d'interaction entre les composants d'un système. Ainsi les propriétés de la bibliothèque (ou du logiciel) utilisée pour communiquer ont une incidence sur la manière concevoir et implémenter les applications distribuées et il est donc nécessaire de prêter attention au choix du couple bibliothèque-interface de communication lors de la conception d'une application.

5.1 Synthèse sur les interfaces de communication

Les évolutions successives de l'informatique permettent de masquer la complexité des mécanismes mis en œuvre derrière une approche de plus en plus conceptuelle. Ainsi, les langages de programmation ont tout d'abord été très proches de la machine (langage machine de bas niveau), puis procéduraux et, enfin, orientés objets. Le succès du langage Java a véritablement popularisé ce mode de programmation.

Les protocoles réseau ont suivi le même type d'évolution. Ils furent d'abord très proches de la couche physique du réseau, avec les mécanismes de sockets orientés octets pour les données et peu de garanties pour la synchronisation ou le transfert. La communication par les sockets implique de prendre en compte l'ensemble des propriétés de la communication pour le développement. Ceci suppose évidement une connaissance maîtrisée de l'interface et des développements conséquents. De plus, comme en assembleur, le développement de nouvelles applications permet peu de réutilisation de code : nous sommes obligés d'adapter la programmation de la communication à chaque nouveau contexte. Cependant, comme toute couche de bas niveau, elle a son utilité dans les environnements suivants :

— contraints : pas assez de mémoire, de capacité de calcul ou de communication,
— spécifiques : transfert de données brutes,
— hérités : pour permettre l'évolution d'applications existantes basées sur des sockets.

L'interface socket, couche d'accès au protocole TPC/IP, est aujourd'hui largement utilisée puisqu'elle constitue la base de tous les développements pour l'Internet et de nombreux

services sur le réseau. Du fait de son positionnement dans les couches basses, elle sert souvent de base au développement de couches de communication de niveau supérieur. Elle reste également un recours lorsque le développeur doit avoir un contrôle précis des échanges tant au niveau de leur synchronisation que du transfert de données brutes. Notons qu'il est possible d'associer plusieurs niveaux de communication pour le développement d'une même application en réalisant l'administration sur la base de fonctions de haut niveau alors que les transferts bruts le sont directement avec des sockets.

Ensuite, la notion de RPC a permis de faire abstraction des protocoles de communication et, ainsi, a facilité la mise en place d'applications client-serveur. Aujourd'hui, l'utilisation d'un modèle de communication «orienté objet», où la communication est réalisée grâce à des invocations de méthodes, permet une totale transparence des appels distants et permet de manipuler un objet distant comme s'il était local. Le flux d'informations fut donc initialement constitué d'octets, puis de données et, enfin, de messages.

Une application à objets répartis utilise des objets qui résident sur des machines distinctes d'un réseau. Ces objets sont sollicités par des appels de méthodes, initiés à distance, qui transitent via ce réseau pour être appliqués sur l'objet, là où il réside. Du point de vue du programme appelant de telles méthodes, les objets auxquels elles s'appliquent sont appelés *objets distants*. Nous avons étudié RMI, JMS et les services Web.

RMI RMI permet à un programme Java d'invoquer des méthodes sur des objets Java dans d'autres machines virtuelles, qui peuvent s'exécuter sur des hôtes différents. Un objet talon local gère l'invocation des méthodes sur les objets distants. RMI utilise la sérialisation afin d'assembler et désassembler les paramètres de ces appels. La sérialisation d'objets est une spécification par laquelle les objets peuvent être convertis vers un flux d'octets, et reconstitués à partir du flux. Le flux inclut suffisamment d'informations pour reconstituer les champs dans le flux vers des versions compatibles pour la classe.

Afin d'offrir le support RMI, Java utilise le modèle d'objets distribué, qui diffère du modèle d'objets de base en plusieurs points, incluant : les arguments non distants passés en paramètres ou retournés d'un appel RMI sont passés par copie : un objet distant est passé par référence et non en copiant la réelle implémentation de l'objet ; les clients interagissent avec les objets distants grâce aux interfaces distantes, et non pas avec leur implémentation ; la sémantique de certaines méthodes définies dans la classe `Object`, `equals`, `hashCode`, `toString`, `clone` et `finalize` ont des sémantiques particulières pour les objets distants, et des exceptions additionnelles peuvent apparaître lors d'un appel RMI.

JMS La communication par messages permet de relier les composants d'une application distribuée en conservant l'expression explicite des échanges. La spécification JMS permet un accès standardisé, depuis le langage Java, aux middlewares de communication par messages, les MOMs. Ces MOMs agissent comme une entité intermédiaire dans l'échange de messages et permettent ainsi d'étendre la sémantique des communications, avec les modes de communication point-à-point et événementiel, et les services qui sont associés, persistance, transactions.

L'intérêt de la communication par messages est de relier des composants d'applications pour leur permettre d'interagir. Les MOMs supportent facilement l'hétérogénéité. Par exemple ActiveMQ est accessible à partir de différents langages : java, C++, python. L'interopérabilité sur les données peut alors être obtenue en utilisant un codage intermédiaire tel que Json. La portée des MOMs est cependant limitée à un réseau interne car ils ne permettent pas de franchir les firewalls.

Services Web SOAP Les services Web de type SOAP permettent l'appel des méthodes distantes (services) en utilisant un protocole Web (HTTP en général) pour le transport et XML pour l'encodage des données. Ils permettent l'interopérabilité entre plates-formes, langages de programmation et systèmes d'exploitation. Le protocole SOAP est utilisé pour la communication et l'échange de données, WSDL pour la description des services et enfin des annuaires (UDDI) pour référencer les services et les entreprises.

Une des forces de SOAP est de permettre l'interopérabilité entre différentes plateformes. Il est donc important d'avoir des règles de codage des types de données, pour qu'elles soient encodées/décodées sans difficultés. On distingue deux types de données, simple (chaîne de caractères par exemple) et composé (structures et tableaux). Dans le cas où des données binaires devraient transiter (comme une image par exemple), il est également possible d'envoyer un message SOAP avec attachement en utilisant un message MIME (Multimedia Internet Mail Extension.

Services Web REST Les services Web de type REST utilisent le protocole HTTP comme couche de base pour la communication. Les différents types de requêtes HTTP sont utilisés pour Depuis quelques années les services web REST ont tendance

5.2 Pour aller plus loin

L'objectif de ce cours est de vous montrer les principaux concepts de communication. Vous donner des connaissances sur toutes les couches possibles de communication est hors de portée d'un seul cours mais les concepts présenter ici devraient vous permettre de vous en sortir avec la plupart d'entre elles.

Pour les personnes intéressées nous donnons dans la suite des pointeurs vers plusieurs outils qui peuvent répondre à leur besoins ou à leur curiosité.

Programmation socket :
- en C : voir annexes
- en C#
- en python

Encodage des messages :
- Protobuf
- Json
- Thrift

Protocoles de communication :
- AMQP
- OpenWire
- Stomp
- MQTT

Bus de messages :
- ZeroMQ
- RabbitMQ

Communication par RPC :
- GRPC
- XML-RPC
- Thrift

Annexes

Nous donnons en annexe deux chapitres sur la programmation socket en C, tels qu'ils étaient lorsque nous avons décidé de les enlever du cours pour laisser plus de place aux chapitres sur d'autres modes/couches de communication. Le premier chapitre fait référence à une bibliothèque de communication `Socket_EAD` que nous ne fournissons plus. Le second chapitre donne toutes les informations complémentaires pour se passer de cette bibliothèque.

Annexe A

Premiers pas avec des sockets

Après avoir pris conscience de la signification de la communication entre programmes, nous passons à une phase plus pratique de la communication : les sockets. L'utilisation de l'interface de programmation complète des sockets n'étant pas très facile d'accès, nous avons découpé cette utilisation en deux chapitres : le premier permet de se familiariser avec les principes de base tandis que le second donne une vision plus complète des possibilités. Dans ce chapitre, nous décrivons donc comment utiliser les sockets pour mettre en place une communication entre des programmes sur la base d'une interface de programmation simplifiée de manière à limiter les difficultés technologiques.

A.1 Définition des sockets

Littéralement, une socket est une «prise» de communication depuis un processus vers l'extérieur : c'est l'entité qui permet au programme de «se brancher» pour communiquer avec le monde. Un programme qui souhaite communiquer en utilisant TCP/IP commence donc par créer sa propre socket. Pour communiquer avec un autre processus possédant lui même une socket, le programme doit ensuite mettre les deux sockets en rapport. Il peut alors utiliser la sienne pour envoyer et recevoir des données. Une même socket sert aussi bien à envoyer qu'à recevoir. Lorsque la communication est finie, le programme ferme la socket.

L'entité socket utilisée par le programme est créée au cœur du système d'exploitation et du protocole. Pour cette raison, nous faisons appel, dans les programmes, à des fonctions systèmes pour toute manipulation qui concerne les sockets. La représentation des sockets au sein du programme peut varier suivant les systèmes d'exploitation ou le langage de programmation. Nous utilisons ici le langage C et une interface de système Unix.

Un même programme peut créer plusieurs sockets pour communiquer avec plusieurs programmes. Dans notre programme, une socket est alors identifiée à partir d'un descripteur. Ce descripteur sera toujours utilisé dans les fonctions d'accès aux sockets de manière à spécifier de quelle socket il s'agit. Ce descripteur est une valeur entière qui sera parfois, par abus de langage, assimilée à la socket.

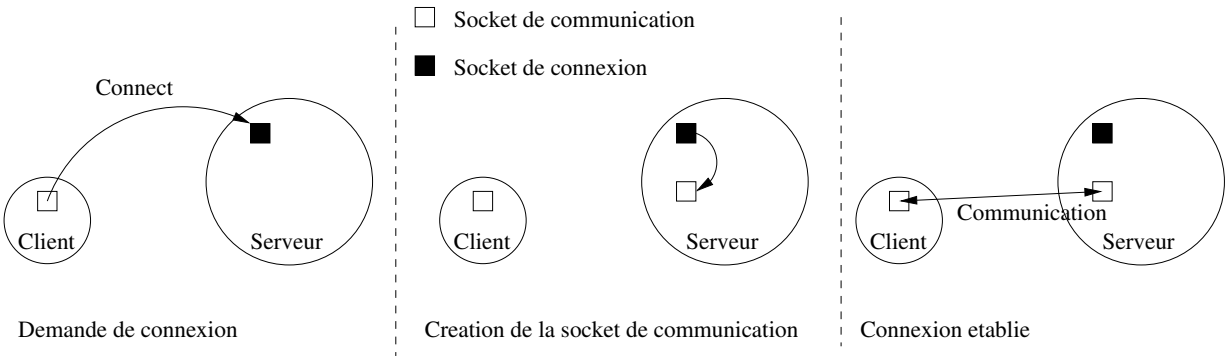


FIGURE A.1 – Établissement d'une connexion

La création d'une socket dépend du mode de communication avec lequel nous souhaitons utiliser la socket.

A.2 Les modes de communication

Comme nous l'avons vu au chapitre précédent, il existe différents modes de communication dont les principaux sont le mode connecté et le mode non-connecté. Ces deux modes sont proposés avec les sockets.

A.2.1 Le mode connecté

Dans le **mode connecté**, deux sockets établissent une relation durable qui permet de ne pas préciser la socket destinatrice à chaque envoi de données, une fois la connexion réalisée. Il est possible d'établir une analogie (relative) entre le mode connecté et l'utilisation du téléphone : vous composez un numéro pour établir une connexion pendant laquelle, tant que vous n'avez pas raccroché, tout ce que vous dites est transmis à votre interlocuteur.

La philosophie du mode connecté est cependant différente de celle du téléphone. Le modèle de connexion des sockets suppose une relation asymétrique où un des programmes prend le rôle du **serveur** et l'autre le rôle du **client**. Le serveur est vu comme un programme rendant des services à différents clients. Son exécution sous-entend une attente de demande de connexion de la part d'un client, l'ouverture de cette connexion, le traitement des messages échangés (service rendu), la fermeture de sa connexion avant de se mettre en attente du prochain client.

Pour sa part, le client se contente de demander une connexion avec le serveur, de lui envoyer les messages à traiter puis de fermer sa connexion. Pour mettre en place ce schéma, la notion de **socket de connexion** est introduite. En fait, le serveur utilise sa socket initiale, celle qu'il crée, uniquement pour recevoir des demandes de connexions. Ensuite, à chaque réception d'une demande de connexion de la part d'un client sur cette socket, le système crée chez le serveur une nouvelle socket, comme cela est illustré par la figure A.1. C'est cette socket qui est utilisée pour communiquer directement avec le client. Ce modèle

permet au serveur de traiter simultanément plusieurs clients et de recevoir les requêtes de clients différents sur des sockets différentes.

A.2.2 Le mode non-connecté

L'autre mode de communication utilisable avec les socket est le **mode non-connecté**. Comme son nom l'indique, il ne suppose pas l'établissement d'une connexion avant la mise en place d'une communication. Dans ce cas, le destinataire de la socket est précisé à chaque envoi. Il est possible d'établir une analogie entre le mode non-connecté et le courrier, qu'il soit postal ou électronique, pour lequel nous précisons systématiquement le destinataire. La socket peut être vue comme une boîte aux lettres par laquelle nous envoyons et recevons des données.

A.3 L'identification

Comme nous l'avons évoqué précédemment, il est nécessaire de savoir précisément à quelle socket nous nous adressons dans plusieurs cas :

— En mode connecté, pour demander la connexion d'un client à un serveur, il est nécessaire de pouvoir spécifier à quel serveur nous nous connectons.

— En mode non connecté, pour envoyer un message, il est nécessaire de pouvoir spécifier quel est le destinataire du message.

Une socket est donc identifiée de manière unique à partir du **nom de la machine** (en fait son adresse IP) sur laquelle elle se trouve et d'un numéro local à la machine, appelé **numéro de port**. C'est la combinaison de ces deux informations qui garantit que deux sockets ne possèdent pas la même identification. Ainsi deux sockets créées sur la même machine possèdent forcément des numéros de port différents et deux socket créées sur des machines différentes peuvent avoir le même numéro de port.

Le nom de la machine est le nom internet de la machine, celui qui s'affiche généralement lors de la saisie du login ou qui peut être obtenu à l'aide de la commande `hostname`. Si la communication dépasse les limites du réseau local, il est nécessaire d'adjoindre au nom les identifications de domaine. Par exemple, si je souhaite envoyer un message à une machine appelée `smith` dans le domaine `univ-fcomte.fr` alors que je ne m'y trouve pas, je dois donner le nom `smith.univ-fcomte.fr`, sinon le nom simple est généralement suffisant. Il est également possible pour une communication locale, si les programmes sont sur la même machine, d'utiliser le nom `localhost`.

La valeur donnée au numéro de port est généralement fixée par le programme à la création de la socket. Cette valeur doit être comprise entre 1024 et 65536, elle est donc codée sur 16 bits et n'est pas signée.

Attention Le numéro de port ne doit pas être confondu avec le descripteur de la socket qui sert à l'identification de la socket à l'intérieur du programme.

A.4 Création et fermeture

La première chose à faire pour mettre en place une communication est donc de créer une socket dans notre programme. Une part importante de la complexité de l'interface socket concerne la création des sockets et leur identification, nous avons donc simplifié cette partie en proposant des fonctions plus faciles d'accès. **Ces fonctions ne sont pas standardisées !** Elles ont été créée pour ce cours et ne sont donc pas disponibles dans les interfaces de programmation socket usuelles. Elles ont donc un suffixe «`_EAD`» pour bien les distinguer de l'API socket standard.

Pour créer une socket, nous devons d'abord savoir quel mode de communication, connecté ou non connecté, nous souhaitons utiliser avec cette socket, car le mode de communication est défini à la création d'une socket et ne peut être changé par la suite. Notons tout de même qu'un même programme peut faire des communications connectées et des communications non connectées à condition de posséder deux sockets, chacune créée avec un mode différent.

Pour utiliser les fonctions de création simplifiée des sockets, il est nécessaire d'inclure dans vos programmes le fichier définissant les types et valeurs associés aux sockets :

```
1 #include "fonctionsSocket.h"
```

A.4.1 Création en mode non-connecté

En mode non-connecté, nous pouvons créer une socket avec la fonction `socketUDP_EAD` dont le prototype est le suivant :

```
1 int socketUDP_EAD(unsigned short port);
```

La fonction prend en paramètre le numéro de port qui doit être associé à la socket. Elle rend une valeur entière qui est le descripteur de socket alloué. Il nous sert de référence à chaque fois que nous devons utiliser la socket. Si la socket ne peut être créée, la valeur de retour est négative.

Par exemple, le programme suivant crée une socket et mémorise son descripteur dans la variable `desc_sock`.

```
1  int main(int argc, char *argv[]) {
2
3    int desc_sock; /* descripteur de socket */
4
5    desc_sock = socketUDP_EAD(2609);
6    if (desc_sock < 0) {
7      printf("Erreur %d a la creation socket\n", desc_sock);
8    }
9
10   return 0;
11 }
```

En mode non-connecté, tous les programmes créent les sockets de la même manière. Attention, si plusieurs programmes sont exécutés sur la même machine, vous devez leur donner des numéros de port différents. Une fois la socket créée, elle est directement utilisable pour mettre en place une communication.

A.4.2 Création en mode connecté

Si nous utilisons le mode connecté, nous devons, en plus des informations sur la machine et le port, savoir quel est le rôle du programme : client ou serveur, car nous avons vu qu'ils ne travaillaient pas de la même façon. Nous utilisons les fonctions suivantes pour réaliser la création des différents types de socket en mode connecté. Les fonctions post-fixées «_EAD» sont directement inspirées de ce qui est proposé dans Java pour la programmation socket et ne sont pas dans l'API socket standard. À l'inverse, la fonction `accept` est une fonction standard.

```
1 int socketServeur_EAD(unsigned short port);
2 int socketClient_EAD(const char *host, unsigned short port);
3
4 int accept(int sock_desc, struct sockaddr *, socklen_t);
```

Le principe général à mettre en œuvre pour la création des sockets en mode connecté est le suivant :

1. Le serveur crée une socket destinée à recevoir les demandes de connexion avec la fonction `socketServeur_EAD`.
2. Ensuite, le serveur se met en attente d'une connexion avec la fonction `accept`.
3. Enfin, le client demande à se connecter au serveur à l'aide de la fonction `socketClient_EAD`.

Pour créer la socket du serveur, nous précédons de la même manière que pour le mode non-connecté pour associer un numéro de port à la socket. La fonction utilisée s'appelle `socketServeur_EAD`. Elle rend un descripteur de socket qui est utilisé dans la fonction `accept` pour attendre les demandes de connexions de la part du ou des clients. Cette structure un peu complexe permet alors à plusieurs clients de se connecter avec un seul serveur. Ainsi le serveur attend toujours sur la même identification de socket (numéro de port et nom de machine), connue de l'ensemble de ses clients, et traite les demandes de connexions successives. La fonction `socketServeur_EAD` n'est pas bloquante, elle retourne donc dès que la socket est créée. Cette fonction retourne une valeur négative en cas de problème de création.

La fonction `accept` est bloquante et ne sort que lorsque le serveur a reçu une demande de connexion de la part d'un client. En retour, elle rend le descripteur de socket qui a été créé pour la connexion avec le client ou -1 en cas d'erreur. Les deux derniers paramètres de la fonction `accept` servent à recevoir une identification de la socket client. Pour le moment, nous mettons systématiquement ces valeurs à NULL. L'exemple suivant illustre la création de la socket `desc_sock_conn`, pour recevoir les demandes de connexion, puis de la socket `desc_sock_trans` connectée avec un client.

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

  /* descripteur de la socket de connexion */
  int desc_sock_conn;

  /* descripteur de la socket de transmission */
  int desc_sock_trans;


  desc_sock_conn = socketServeur_EAD(2609);
  if (desc_sock_conn < 0) {
    printf("Erreur creation socket\n");
    return 1;
  }

  desc_sock_trans = accept(desc_sock_conn, NULL, NULL);
  if (desc_sock_trans == -1) {
    printf("Erreur connexion socket\n");
    return 2;
  }

  return 0;
}
```

Le client utilise la fonction `socketClient_EAD` pour créer sa socket. Les paramètres donnés à l'appel de la fonction sont le nom de la machine et le port de la socket du serveur auquel il souhaite se connecter. Ici il est donc important que le numéro de port passé en paramètre soit le même pour le client et le serveur, de même le nom de machine donné doit être celui du serveur. Cette fonction est bloquante tant que la connexion avec le serveur ne peut être obtenue ou qu'une durée limite (*timeout*) n'est pas dépassée. L'exemple suivant illustre la création de la socket du client.

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

  int desc_sock; /* descripteur de socket */

  desc_sock = socketClient_EAD("smith", 2609);
  if (desc_sock < 0) {
    printf("Erreur creation socket\n");
  }

  return 0;
}
```

Une fois les sockets créées, il est possible de passer à la communication entre les programmes.

A.4.3 Fermeture des connexions et des sockets

Lorsque l'échange entre un client et un serveur en mode connecté se termine, il faut normalement fermer la connexion qui les relie en utilisant la fonction `shutdown` dont le prototype est :

```
1 int shutdown(int desc_sock, int how);
```

Le premier paramètre `desc_sock` correspond au descripteur de la socket concernée. Le second `how` permet de fixer sur quel mode est fermée la connexion. La valeur 0 permet de fermer en réception, 1 ferme en envoi et 2 ferme complètement la connexion. Les différents modes permettent soit de rendre la connexion unidirectionnelle (modes 0 et 1), soit de couper celle-ci. L'appel à la fonction `shutdown` pour des sockets en mode non-connecté est une erreur !

Pour terminer un programme proprement, quel que soit le mode de communication utilisé, il faut penser à fermer les sockets qui ont été ouvertes en utilisant la fonction `close`, dont le prototype est :

```
1 int close(int desc_sock);
```

Le paramètre `desc_sock` correspond au descripteur de la socket que l'on veut fermer. La fonction retourne un code d'erreur à -1 en cas de problème.

Attention, le fait d'appeler la fonction `shutdown` ne ferme pas la socket mais seulement la connexion, il est donc nécessaire de faire également appel à la fonction `close`. Notons tout de même que si ni l'une ni l'autre des fonctions ne sont appelées avant la fin du programme, le système se charge de réaliser ces fermetures pour terminer proprement le programme. Il est cependant plus correct de réaliser ces fermetures soi-même dès que possible dans le programme pour éviter de consommer inutilement des ressources.

A.5 La communication

La communication socket se fait sur la base d'échanges de données grâce à des fonctions d'envoi et des fonctions de réception. Les fonctions d'envoi prennent en paramètre les données à envoyer sous la forme d'un *buffer* : une zone de mémoire caractérisée par son adresse de début et sa taille. Le système copie le contenu de cette zone dans le message à envoyer. De la même manière, le récepteur reçoit les données du message dans une zone de mémoire. Cette zone de mémoire devra être pré-allouée avant de demander à recevoir pour que le système dispose, dans votre programme, d'une zone de mémoire pour recopier les données, celle-ci n'étant pas allouée par le système.

Suivant le mode de communication associé à la socket, différentes fonctions seront utilisées. Dans le mode non-connecté, la socket de l'émetteur n'est pas reliée à celle du récepteur, il est donc nécessaire de préciser le destinataire à chaque envoi. Dans le mode connecté, deux sockets sont liées, les données émises sur une socket sont envoyées à la socket connectée.

A.5.1 Le mode non-connecté

Le prototype général de la fonction d'envoi `sendto` est le suivant :

```
ssize_t sendto(
  int desc_sock,     /* descripteur socket */
  const void *buf,   /* pointeur sur zone de memoire */
  size_t len,        /* taille de la zone de memoire */
  int flags,         /* options d'envoi */
  const struct sockaddr *to, /* id socket destinatrice */
  socklen_t addrlen /* taille de l'adresse */
);
```

Le premier paramètre `desc_sock` est le descripteur de la socket, obtenu à sa création.

Les deux paramètres suivants définissent les données à envoyer. Comme nous l'avons expliqué précédemment, les données sont envoyées sous la forme d'un *buffer* identifié à partir du pointeur `buf` de type `void *` (pointeur) et de la taille `len` de type `size_t`. Le type `size_t` accepte ici une valeur entière.

La valeur qui est passée en quatrième paramètre `flags` correspond à des propriétés de notre envoi (prioritaire, limité, etc.). Pour des raisons de simplicité, nous utilisons toujours la valeur 0 pour ce paramètre.

Les deux derniers paramètres servent à passer l'identification de la socket destinatrice. L'adresse est donnée par un pointeur `to` de type `struct sockaddr *` et sa taille `addrlen` de type `socklen_t`. Pour simplifier cette partie, nous utilisons la fonction `socketAddr_EAD` qui rend directement l'identification à partir des informations dont nous disposons, c'est-à-dire le nom de la machine et le numéro de port de la socket. Le paramètre de taille de l'adresse est donné par la fonction `tailleAddr_EAD`.

Attention, les fonctions `socketAddr_EAD` et `tailleAddr_EAD` ne sont pas des fonctions standards, elles fonctionnent bien pour nos exercices mais ne conviennent pas pour une utilisation avancée des sockets.

La fonction `sendto` est bloquante tant que les données ne sont pas parties de la machine. Il est donc possible de réutiliser le buffeur du message dès que le programme est sorti de l'appel. À son retour, la fonction donne le nombre d'octets envoyés ou -1 si l'exécution de la fonction s'est mal déroulée. L'exemple suivant montre un envoi d'une donnée contenue dans une variable entière.

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

  int desc_sock; /* descripteur de socket */
  int val = 42;
  int envoyes;

  /* creation de la socket */
  ...

  /* envoi des donnees */
  envoyes = sendto(desc_sock, &val, sizeof(int), 0,
    socketAddr_EAD("smith", 2610), tailleAddr_EAD());
```

```
13
14    if (envoyes == -1) {
15      printf("Erreur a l'envoi\n");
16    }
17
18    close(desc_sock);
19    return 0;
20  }
```

Ici le *buffer* de donnée est constitué d'une seule donnée, notre valeur entière. Le buffeur est donc identifié par l'adresse en mémoire de la variable et la taille de la variable. Ces deux paramètres sont passés à la fonction comme deuxième et troisième paramètres.

Pour recevoir un un message nous utilisons la fonction `recvfrom` dont le prototype est le suivant :

```
1  ssize_t recvfrom(
2    int desc_sock,        /* descripteur socket */
3    void *buf,            /* pointeur sur zone de memoire */
4    size_t len,           /* taille de la zone de memoire */
5    int flags,            /* options de reception */
6    struct sockaddr *from, /* id envoyeur */
7    socklen_t *addrlen    /* taille de l'adresse */
8  );
```

Comme pour la fonction d'envoi, le premier paramètre `desc_sock` est le descripteur de la socket qui est utilisée pour recevoir. Les deuxième et troisième paramètres décrivent le buffeur utilisé pour recevoir les données. Comme nous l'avons expliqué précédemment, les données sont reçues dans le buffeur identifié à partir du pointeur `buf` de type `void *` et de la taille du buffeur `len` de type `size_t`. Comme pour la fonction `sendto`, le quatrième paramètre `flags` donne des options de réception, nous le mettons systématiquement à 0 pour avoir une réception standard. Les derniers paramètres permettent de recevoir l'identification de la socket émettrice mais, pour simplifier la programmation, nous leur donnons la valeur `NULL`.

La fonction `recvfrom` rend -1 en cas de problème d'invocation. Sinon, elle est bloquante tant que rien n'est reçu et elle retourne un résultat dès la réception d'un premier message, même si le buffeur prévu pour recevoir le message n'est pas rempli. En retour, la fonction rend alors la taille du message recu. Donc, si le récepteur ne connaît pas à priori la taille du message à recevoir, il doit allouer avant l'appel à la fonction un buffeur de taille suffisante pour recevoir différents messages mais il peut quand même traiter le message reçu même s'il ne remplit pas entièrement le *buffer*. L'exemple suivant illustre la réception de la donnée entière envoyée précédemment.

```
1  int main(int argc, char *argv[]) {
2    int desc_sock; /* descripteur de socket */
3
4    int val;
5    int recus;
```

```
6
7   /* creation de la socket */
8   ...
9
10  /* reception des donnees */
11  recus = recvfrom(desc_sock, &val, sizeof(int), 0, NULL, NULL);
12
13  if (recus == -1) {
14    printf("Erreur a la reception\n")
15  }
16
17  printf("J'ai recu la valeur : %d\n", val);
18
19  close(desc_sock);
20  return 0;
21 }
```

A.5.2 Le mode connecté

Dans le mode connecté, on relie temporairement deux descripteurs de socket sur le principe énoncé à la section A.2.1. Il n'est donc plus nécessaire de préciser l'adresse à chaque échange.

En utilisant la socket connectée, les messages sont envoyés grâce à la fonction `send` dont le prototype est le suivant :

```
1 ssize_t send(
2   int desc_sock,     /* descripteur socket */
3   const void *buf,   /* pointeur sur zone de memoire */
4   size_t len,        /* taille de la zone de memoire */
5   int flags,         /* options d'envoi */
6 );
```

La fonction `send` est une version réduite de la fonction `sendto` dans laquelle il n'est pas nécessaire de préciser le destinataire. Son utilisation en est donc simplifiée. De la même manière que la fonction `sendto`, cette fonction prend une valeur de `flags` en paramètre, nous la mettons à 0.

Comme la fonction `sendto`, la fonction `send` bloque tant que le message n'est pas parti de la machine locale. Ceci garantit que le buffeur du message peut être réutilisé sans risque après l'appel à la fonction. Le retour de la fonction correspond au nombre d'octets envoyés ou à un code d'erreur -1. L'exemple suivant illustre l'utilisation de cette fonction, une fois la connexion établie, pour l'envoi d'une chaîne de caractères :

```
1 int main(int argc, char *argv[]) {
2
3   int desc_sock; /* descripteur de socket */
4   char *chaine = "Hello world";
5   int envoyes;
6
```

```
7   /* creation de la socket */
8   ...
9
10  /* envoi des donnees */
11  envoyes = send(desc_sock, chaine, 12, 0);
12
13  if (envoyes == -1) {
14    printf("Erreur a l'envoi\n");
15  }
16
17  close(desc_sock);
18  return 0;
19 }
```

Ici, la chaîne de caractère est donnée à partir de l'adresse en mémoire `chaine` du premier caractère de cette chaîne et de la taille de la chaîne. Notons que cette chaîne n'est composée que de 11 caractères donc de 11 octets. Le douzième octet que nous demandons de passer à la fonction `send` correspond au délimiteur de la chaîne, soit le caractère '\0', qui doit être également transmis si nous voulons conserver à la chaîne ses propriétés.

Les messages sont reçus grâce à la fonction `recv` dont le prototype est le suivant :

```
1 ssize_t recv(
2   int desc_sock,       /* descripteur socket */
3   void *buf,           /* pointeur sur zone de memoire */
4   size_t len,          /* taille de la zone de memoire */
5   int flags,           /* options de reception */
6 );
```

Cette fonction est la version réduite de la fonction `recvfrom`. De la même manière nous mettrons le paramètre `flags` à 0.

Comme la fonction `recvfrom`, cette fonction est bloquante tant que nous n'avons pas reçu de message mais elle n'attend pas d'avoir reçu la taille des données spécifiée dans `len` pour terminer son attente. Le retour de la fonction donne le nombre d'octets effectivement reçus par la fonction ou un code d'erreur -1. L'exemple suivant illustre la réception de la chaîne de caractères envoyée précédemment.

```
1  int main(int argc, char *argv[]) {
2    int desc_sock; /* descripteur de socket */
3    char buf[20];
4    int recus;
5
6    /* creation de la socket */
7    ...
8
9    /* reception des donnees */
10   recus = recv(desc_sock, buf, sizeof(buf), 0);
11
12   if (recus == -1) {
13     printf("Erreur a la reception\n");
```

```
14    }
15
16    printf("J'ai recu : %s\n" buf);
17    close(desc_sock);
18
19    return 0;
20  }
```

Cet exemple illustre également le fait que les fonctions de réception n'attendent pas d'avoir reçu autant de caractères que demandé dans le paramètre `len`. Ici, nous demandons à recevoir 20 octets alors que nous n'en envoyons que 12 (les 12 caractères de la chaîne `"Hello world"`). Cette valeur `len` correspond donc à la taille maximale qui peut être reçue.

Remarque importante Il est important de noter qu'il n'y a aucune relation entre les rôles de client et serveur et les rôles d'émetteur et récepteur. Les rôles de client et serveur servent à déterminer, avant l'établissement de la connexion, qui fait appel à la fonction `socketClient_EAD` et qui fait appel aux fonctions `socketServeur_EAD` et `accept`. Une fois cette connexion établie, ces rôles s'effacent au profit des rôles d'émetteur et récepteur pour savoir qui fait appel à la fonction `send` et qui fait appel à la fonction `recv`.

A.5.3 Échange des données

Pour bien maîtriser les envois et les réceptions sur les sockets, il est peut-être nécessaire de rappeler quelques informations sur la manière de transmettre des données. Nous avons vu que, globalement, l'envoi se fait à partir de l'adresse en mémoire et de la taille des données alors que la réception se fait sur la base d'un buffeur alloué par le programme avant la réception. Nous allons voir comment transférer différents types de données.

En ce qui concerne l'envoi des **types de base** défini par le langage C, Les données de ce type sont mémorisées dans des variables. Il est alors possible d'utiliser l'adresse de la variable (`&nom_variable`) comme pointeur de début de message et la taille de la variable, obtenue avec la fonction `sizeof`, comme taille de message. Ceci est illustré par l'exemple que nous avons donné précédemment pour l'envoi d'une valeur entière.

De la même manière, pour la réception de ces types de base, il nous suffit de disposer d'une variable de ce type et de l'utiliser comme buffeur de réception. Dans ce cas, comme pour l'envoi, l'adresse de la variable (`&nom_variable`) est utilisée comme pointeur de début de *buffer* et la taille de la variable, obtenue avec la fonction `sizeof`, comme taille de buffeur.

Ces règles s'appliquent également pour les envois de types ou de structures de données définis par l'utilisateur sauf si ceux-ci contiennent des pointeurs, des tableaux ou des chaînes de caractères.

En ce qui concerne les **tableaux**, nous rappelons que la définition d'un tableau, en langage C, consiste en la réservation d'une zone de mémoire correspondant à la taille du tableau et à l'initialisation de la variable identifiant le tableau avec l'adresse de cette zone de mémoire. La fonction `sizeof` est alors utilisable pour obtenir la taille totale du tableau.

La définition de message à envoyer ou du *buffer* de réception peut donc se faire en utilisant ces deux données, comme cela est illustré avec la réception en mode connecté. Notons que la définition d'un tableau est un moyen aisé de définir un buffeur de réception d'une taille voulue.

En ce qui concerne les **chaînes de caractères**, nous avons rappelé que leur définition consiste en :

- une zone de mémoire dans laquelle les caractères composants la chaîne sont écrits les uns après les autres,
- la variable (de type `char *`) qui l'identifie et qui est initialisée à l'adresse de début de cette zone,
- un caractère '\0' qui marque la fin de la chaîne.

Nous disposons donc, à travers la variable identifiant la chaîne, de son adresse en mémoire. Cette adresse peut être utilisée pour l'envoi, comme cela est réalisé dans l'exemple d'envoi en mode connecté. Par contre, en ce qui concerne sa taille, il n'est pas possible d'utiliser la fonction `sizeof` car cette fonction ne donne que la taille associée au type de la donnée or le type de la variable est `char *`, ce qui vaut 4 octets pour `sizeof`. Pour cette raison, la taille des chaînes doit être calculée avec la fonction `strlen`. Cette fonction donne cependant la taille de la chaîne sans tenir compte du caractère '\0' qui en marque la fin. Pour préserver les propriétés de la chaîne dans l'envoi, il est alors nécessaire d'ajouter 1 à la valeur retournée par `strlen`.

Pour la réception des chaînes de caractères, nous avons vu dans l'exemple de réception en mode connecté que nous pouvions prévoir un tableau de caractères. La difficulté réside alors dans le fait que la taille d'une chaîne n'est pas limitée. Il est donc difficile de savoir combien de caractères réserver dans le tableau qui, lui, doit être de taille fixe. La meilleure solution consiste généralement a établir une convention entre l'émetteur et le récepteur pour garantir que le nombre de caractères envoyé ne dépasse pas la capacité de réception. L'établissement de cette convention n'est cependant pas toujours possible et il faut alors avoir recours à d'autres solutions.

En ce qui concerne les **pointeurs**, nous déconseillons vivement de les passer dans un message. Un pointeur correspond à une adresse dans la mémoire locale de l'émetteur. Il n'a donc aucune signification dans la mémoire du récepteur. Une analogie nous permettra de terminer ce chapitre. Chez moi, pour accéder aux fourchettes, je peux vous donner le pointeur qui dit «cuisine, deuxième tiroir à droite». Si je vous envoie ce pointeur dans un message, il y a peu de chances qu'il référence la même chose. Donc, ce qui est référencé par un pointeur dans un contexte a peu de chance d'être référencé par le même pointeur dans un autre contexte.

Annexe B

Les sockets des experts

Après avoir pris conscience de la signification de la communication entre programmes, nous décrivons dans ce chapitre l'interface de programmation complète. Cette partie permet d'acquérir une connaissance approfondie de la programmation des sockets. Elle est donc plus technique que la précédente mais votre connaissance de manipulation des sockets vous permettra de mieux appréhender les nouveaux concepts présentés ici. Nous reprenons chacune des étapes de l'établissement d'une communication par socket pour la détailler et présenter l'ensemble des fonctions. Comme pour toute autre fonction des système Unix, vous pourrez lire la documentation à travers les commandes `man nom_fonction` ou `info nom_fonction`.

B.1 Généralités

Toutes les communications Internet entre les ordinateurs reposent sur le protocole TCP/IP qui englobe les protocoles IP (Internet Protocol), TCP (Transport Control Protocol) et UDP (Unreliable Datagram Protocol). Le but de ce cours n'est pas de décrire le protocole TCP/IP. Aussi, pour de plus amples explications, vous pouvez revoir votre cours de licence ou consulter l'encyclopédie en ligne Wikipedia à l'adresse : `http://fr.wikipedia.org/wiki/Internet`. Il est cependant nécessaire de rappeler qu'un protocole de communication général, comme Internet, est lui-même composé de plusieurs sous-protocoles. On parle cependant aussi de protocole pour les sous-protocoles. Un protocole complet est également appelé suite de protocoles ou pile de protocoles.

Les sous-protocoles sont généralement organisés en couches : les protocoles de plus haut niveau utilisant ceux des couches basses. Ainsi un protocole de niveau $N + 1$ utilise les fonctions et propriétés du protocole de niveau N mais n'accède pas directement à ceux de niveau $N - 1$. Ce schéma d'organisation à été défini dans la norme OSI (*Open Systems Interface*) de l'ISO (*International Organization for Standardization*). Il est partiellement mis en œuvre dans TCP/IP où les protocoles TCP et UDP utilisent le protocole IP. L'accès à ces protocoles se fait à travers une interface de programmation.

L'API (*Application Programming Interface*, interface de programmation) du protocole

TCP/IP des systèmes Unix repose principalement sur l'utilisation de **sockets** de communication. L'API des sockets permet l'utilisation au niveau applicatif des fonctionnalités du protocole. Cette API a été définie par l'université de Berkeley en 1982 pour le système Unix BSD 4.1, un des premiers systèmes Unix à inclure une implémentation de TCP/IP. Cette interface a été étendue par la suite pour servir d'accès à de nombreux autres protocoles. Parmi ceux-ci, nous pouvons citer X.25, IPX, Netbeui, IPv6, Bluetooth, etc.

Le mode connecté de programmation des sockets permet de mettre en place des communications fiables puisqu'il repose sur le protocole TCP qui offre des garanties quant à l'acheminement des messages. Dans ce mode, nous avons la garantie de ne perdre aucune donnée et l'ordre de réception est identique à celui d'émission. Le choix du mode connecté doit donc se faire lorsque la qualité de l'échange doit être assurée.

Pour sa part, le mode de communication non-connecté repose sur le protocole UDP. Ce protocole ne donne pas la garantie qu'un message envoyé arrive à sa destination. Ainsi, si vous envoyez un message sur une adresse erronée cela ne produira pas d'erreur. De même, si le trafic du réseau est important votre message peut tout simplement être supprimé, sans que vous en soyez informé. Dans la réalité, cela arrive rarement et vous aurez l'impression que ce protocole délivre aussi bien vos messages que le protocole TCP. Il est cependant nécessaire de tenir compte de cette éventualité si votre programme est très sensible à la perte de données. Le risque est plus important si vous envoyez de grosses quantités de données.

B.2 Création des sockets

Les fonctions que nous avons définies pour la création des sockets reposent sur les fonctions de l'interface standard. Pour la création d'une socket, nous utilisons la fonction `socket`. Cette fonction utilise la notion de domaine et d'un type de protocole.

Le domaine définit le type de réseau et la famille de protocole qui supporteront la communication. Les familles de protocoles utilisables par les sockets sont nombreuses, elles concernent un grand nombre des types de réseaux locaux, des réseaux sans fils, etc. Dans ce cours, nous ne travaillons que sur les réseaux internet donc avec la famille de protocole TCP/IP qui est définie à partir du nom de domaine `AF_INET`. Lorsque nous utilisons d'autres protocoles, la valeur est généralement trouvée dans le fichier `<sys/socket.h>`.

Le type de protocole définit un mode d'utilisation spécifique de la famille de protocoles. Ces modes permettent de définir des propriétés de communication généralement liées aux garanties offertes par la communication. Nous avons déjà vu les modes connectés et non-connecté dans le chapitre précédent. D'autres modes permettent des envois en mode datagramme ou message sur des sockets avec plus ou moins de garanties. Nous nous limitons aux deux protocoles déjà vu. Pour le mode connecté, nous utilisons les sockets de type `SOCK_STREAM` et pour les sockets non-connectées, nous utilisons le type `SOCK_DGRAM`.

On crée une socket grâce à la fonction `socket` dont le prototype est :

```
1 int socket(int domain, int type, int protocol);
```

Le paramètre `domain`, pour nous, prendra toujours la valeur `AF_INET`. Le paramètre `type` est le type de communication qui est utilisé par la socket, Pour le protocole TCP on met `type` à `SOCK_STREAM` et pour le protocole UDP on met `type` à `SOCK_DGRAM`. Le paramètre `protocol` vaut toujours à 0. La fonction renvoie un descripteur de socket qui nous sert de référence à chaque fois que nous devons utiliser la socket. Si la socket ne peut être créée, la valeur de retour est -1.

Le résultat de l'appel à la fonction `socket` est la création d'une socket à laquelle aucun nom n'est associé, simplement nous avons la prise à laquelle nous devons, par la suite, attribuer une identification. Le retour de la fonction retourne donc le descripteur de la socket.

Du point de vue de la programmation, nous voyons une socket comme un fichier. C'est à dire que le descripteur qui nous est retourné par la fonction `socket` est en fait un descripteur de fichier. Ce descripteur de fichier est une entrée dans la table des descripteurs de fichier du processus qui exécute notre programme. Il correspond au numéro alloué dans la table à la création de la socket : c'est donc une valeur entière (le système alloue la première entrée libre de la table des descripteurs). La taille de la table des descripteurs de fichiers étant limitée, nous aurons donc un nombre limité de sockets créées à un instant donné dans notre programme. D'où l'intérêt de penser à fermer celles dont nous ne nous servons plus.

Puisque la socket est accédée par le programme sous la forme d'un descripteur de fichier, la plupart des fonctions utilisables sur un fichier le seront également sur la socket. C'est le cas des fonctions de lecture et d'écriture mais aussi de manipulation et de contrôle. Par contre, une socket n'est créée/ouverte qu'avec une fonction spécifique.

Pour utiliser cette fonction, il est nécessaire d'inclure les fichiers définissant les types et valeurs associés aux sockets :

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
```

B.3 Association d'une adresse

La socket que nous avons créée constitue notre «prise» d'accès pour la communication entre processus. Pour reprendre l'analogie entre la communication par sockets et le téléphone ou le courrier, une socket peut être comparée à un combiné téléphonique ou une boîte aux lettres (la boîte elle-même). Pour mettre en place une communication, il est alors nécessaire de définir l'identification de la socket, l'équivalent d'un numéro de téléphone ou d'une adresse de courrier et de l'associer à la socket. En communication socket, cette identification est appelée une adresse. Elle est définie dans une structure `struct sockaddr`.

Attention, le vocable d'adresse est utilisé plusieurs fois avec des significations différentes dans notre contexte de travail. En effet, une adresse peut être une adresse en mémoire, l'adresse d'une socket et l'adresse d'une machine sur le réseau Internet. Nous essayerons donc, par la suite, de conserver la notion d'adresse pour parler de l'adresse associée à une

socket. Pour les adresses en mémoire, nous parlons de pointeur et pour les adresses de machines sur le réseau, nous parlons de numéro Internet ou IP. Lorsque nous ne pouvons faire autrement, nous précisons adresse mémoire ou adresse réseau.

Nous associons une adresse à une socket à l'aide de la fonctions `bind` dont voici le prototype :

```
1 int bind(
2   int desc_sock,
3   const struct sockaddr *addr,
4   socklen_t addrlen
5 );
```

Le paramètre `desc_sock` est le descripteur de la socket à laquelle nous voulons associer une adresse. Le paramètre `addr` est un pointeur sur l'adresse qui doit être associée à la socket et le paramètre `addrlen` est la taille de la structure d'adresse, parce que les adresses n'ont pas la même taille dans tous les domaines de communication. Il est possible d'utiliser `sizeof(addr)`. La valeur de retour est un code d'erreur permettant de savoir comment l'appel s'est passé. En cas de problème, la valeur de retour est -1 et 0 sinon.

Remarque L'interface des sockets a été conçue pour supporter plusieurs protocoles différents. L'adresse que nous souhaitons associer à notre socket traduit en général la position dans le réseau de la socket. Cette adresse est donc dépendante de la famille de protocole utilisée. Or la fonction `bind` est, elle, générique, c'est à dire qu'elle est utilisée quel que soit le protocole accédé par la socket. Pour cette raison, les concepteurs de l'interface ont utilisé une petite subtilité permettant d'utiliser la même fonction quel que soit le protocole de communication : l'adresse est passée sous la forme d'un pointeur, qui référence l'emplacement en mémoire de l'adresse, et d'une valeur donnant la taille de l'adresse. La fonction `bind` peut ainsi récupérer l'adresse, quelle que soit sa taille et sa structure, en accédant directement à la zone de mémoire correspondante. Pour résoudre les problèmes de type des paramètres, un type générique d'adresse `struct sockaddr` est défini. Il recouvre l'ensemble des types d'adresse possibles. Le programmeur passe le paramètre sous la forme d'un pointeur sur l'adresse qui est définie par son protocole. Ainsi, la structure de données passées en paramètre est bien équivalente puisqu'il s'agit d'un pointeur dans les deux cas. Le pointeur donné par le programmeur est transtypé en `struct sockaddr *` pour obtenir la correspondance des types. La fonction `bind` peut ensuite déterminer quel est le type d'adresse utilisé grâce au premier champ de la structure d'adresse. Quel que soit le protocole, donc le type d'adresse, le premier champ de la structure est toujours un entier de deux octets `short` qui contient un identificateur de la famille du protocole de communication. Dans notre cas, ce premier champ sera toujours à `AF_INET`, puisque nos adresses sont des adresses Internet.

B.4 Les adresses : l'API classique

Une adresse sert au protocole de communication pour déterminer à qui doit être délivré un message. Une adresse est l'équivalent d'un numéro de téléphone ou d'une adresse de courrier, qu'il soit postal ou électronique. Une adresse est utilisée, soit pour être associée à une socket, soit pour être associée à l'envoi d'un message de manière à en préciser le destinataire.

B.4.1 Adresses `AF_INET`

Dans le domaine de communication `AF_INET`, nous utilisons des adresses décrites par une structure `struct sockaddr_in`. Les adresses `AF_INET` permettent d'accéder à toutes les machines d'un domaine Internet. Dans une adresse `AF_INET`, il y a deux identificateurs : un pour la machine et un pour la socket sur la machine. Cette double identification permet d'obtenir une adresse unique pour chaque socket. En effet, dans le réseau Internet, chaque machine est identifiée par un numéro (adresse réseau) IP unique, décrit dans une structure `struct in_addr`. C'est ce numéro qui est utilisé dans la partie machine de l'adresse d'une socket et non le nom de la machine comme nous l'avons fait au chapitre précédent. Ensuite, le numéro de port permet une identification sur la machine. Le contenu d'une adresse `AF_INET` est donc le suivant (voir `ip(7)`) :

```
1 struct sockaddr_in {
2   sa_family_t sin_family; /* famille d'adresses : AF_INET */
3   in_port_t sin_port;     /* port dans l'ordre des octets reseau */
4   struct in_addr sin_addr; /* adresse Internet */
5 };
```

L'utilisation de cette structure et des fonctions qui la manipule nécessite l'inclusion des fichiers d'entête du protocole Internet :

```
1 #include <netinet/in.h>
```

Cette structure contient en fait des champs supplémentaires permettant, pour des raisons de sécurité, d'aligner la taille de toutes les adresses réseaux. Donc, avant d'initialiser cette structure, il est nécessaire de la mettre complètement à 0 de la manière suivante :

```
1 struct sockaddr_in addr;
2 memset(&addr, 0, sizeof(addr));
```

Pour initialiser une adresse `AF_INET`, il est nécessaire d'initialiser un numéro IP et un numéro de port. Le champ `sin_addr` doit donc être initialisé avec le numéro IP de la machine sur laquelle se trouve la socket et le champ `sin_port` avec le numéro de port qui doit être associé à la socket.

Cependant, dans le cas d'une adresse destinée à être associée à une socket (utilisation de la fonction `bind`), il n'est pas toujours nécessaire d'initialiser ce champ `sin_addr` avec l'adresse d'une machine. Le champ peut être initialisé à `INADDR_ANY`, et dans ce cas c'est l'adresse de la machine courante (au moment de l'exécution) qui est utilisée. Ceci permet

d'écrire des programmes qui attribuent bien une adresse locale à la socket sans avoir à connaître le numéro IP de la machine au moment où nous écrivons le programme.

De la même manière, il est possible de ne pas fixer de numéro de port dans l'adresse lors de l'association d'une socket et son adresse, en positionnant le champ `sin_port` à 0. Dans ce cas, le numéro de port est attribué par le système en fonction des numéros déjà utilisés par les autres sockets.

Dans la suite, nous voyons différentes manières d'initialiser le numéro IP et le numéro de port.

B.4.2 Les numéros Internet

Dans un domaine Internet, chaque machine est identifiée à l'aide d'un numéro IP. Étant donné la large utilisation du protocole IP et donc la variété des implémentations, la description de ce numéro varie. Il est en fait toujours composé de quatre octets. Mais, il peut être décrit de la manière suivante. Dans des implémentations anciennes, il est généralement décrit par :

```
1 struct in_addr {
2     union {
3         struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
4         struct { u_short s_w1,s_w2; } S_un_w;
5         u_long S_addr;
6     } S_un;
7 };
8 #define s_addr S_un.S_addr
```

Dans les versions récentes de Linux, il est décrit par :

```
1 typedef uint32_t in_addr_t;
2 struct in_addr
3   {
4     in_addr_t s_addr;
5   };
```

En fait, les deux représentations sont les mêmes puisqu'elles reposent sur une même structure de quatre octets. La première représentation donne trois visions différentes : sous la forme de quatre caractères non signés (en Unix `u_char` équivaut à `unsigned char`) séparés, sous la forme de deux entiers de deux octets (`u_short`) ou d'un entier de quatre octets. Cette représentation est justifiée par la définition des différentes classes d'adresse utilisées dans le routage Internet. Ainsi, nous voyons souvent le numéro IP d'une machine écrit sous la forme de quatre octets. Par exemple la machine `moodle` a pour adresse `194.57.91.49`. La seconde représentation donnée ne définie le numéro IP que sous la forme d'un entier de 32 bits.

Tout ce discours est un peu complexe. Pour ceux qui ne maîtrisent pas les structures complexes en langage C, il est suffisant de retenir que le numéro IP est un entier et qu'il peut être, dans tous les cas, représenté par le champ `s_addr` de la structure `struct in_addr`.

B.4.3 Les noms de machine

Ce numéro IP n'étant pas très facile à mémoriser, il est possible, grâce au DNS, de donner un nom aux machines et d'établir une correspondance entre les noms et les numéros. Pour obtenir l'adresse `AF_INET` (Internet) d'une machine, on peut utiliser la fonction `gethostbyname`. Cette fonction permet d'obtenir des informations sur une machine à partir de son nom. Le prototype de la fonction est le suivant :

```
1 struct hostent *gethostbyname(const char *name);
```

Le paramètre `name` est une chaîne de caractère donnant le nom de la machine dont nous cherchons l'adresse. La structure `struct hostent`, valeur de retour de la fonction est définie de la manière suivante :

```
1 struct hostent {
2     char  *h_name;       /* nom officiel de l'hote */
3     char **h_aliases;    /* liste d'alias */
4     int    h_addrtype;   /* type d'adresse de l'hote */
5     int    h_length;     /* longueur de l'adresse */
6     char **h_addr_list;  /* liste d'adresses */
7 }
8 #define h_addr h_addr_list[0] /* pour compatibilite */
```

Cette fonction nécessite l'inclusion du fichier :

```
1 #include <netdb.h>
```

Comme l'indique le prototype, cette fonction retourne une structure de données de type `struct hostent` qui décrit les informations associées à la machine du point de vue du DNS. À l'appel de cette fonction, le système recherche dans ses tables locales puis dans le DNS, les informations liées à la machine suivant la politique de résolution de noms définie par le système, dans le fichier `/etc/nsswitch.conf`.

La structure `struct hostent` comprend :

- le nom principal de la machine dans le champ `h_name`.
- une liste d'alias dans le champ `h_aliases`. Une machine peut avoir plusieurs noms, par exemple `moodle` et `moodle.univ-fcomte.fr` sont deux noms différents. Dans ce cas, la machine possède un nom principal et les autres noms sont mémorisés dans le champ des alias.
- le type des adresses réseaux, pour nous des numéro IP, associées à la machine dans le champ `h_addrtype`. La fonction ayant été conçue pour fonctionner avec différents protocoles, ce champ permet de savoir comment traiter les adresses réseaux qui sont mémorisées dans le champs `h_addr_list`.
- la longueur d'une adresse réseau dans le champ `h_length`. De la même manière que pour la fonction `bind`, les adresses réseaux sont ici gérées sur la base de la zone où elles sont enregistrées en mémoire. C'est à dire à partir d'un pointeur sur le début de la zone et de la taille occupée par l'adresse réseau.
- la liste des adresses réseaux de la machine dans le champ `h_addr_list`. De la même manière qu'une machine peut avoir plusieurs noms, elle peut aussi avoir plusieurs

adresses réseau (c'est surtout le cas pour les machines passerelles, à la frontière entre deux réseaux). L'adresse réseau principale est toujours stockée en premier. Elle est donc accessible à partir de la simplification `h_addr`. Il faut remarquer que le tableau obtenu n'est pas constitué de pointeurs sur un type d'adresse réseau générique mais plutôt de pointeurs sur des caractères. Cette définition était classique lorsque l'interface socket à été définie, la définition `char *` était utilisée pour les pointeurs génériques. Il sera donc nécessaire de transtyper («caster») le pointeur donné par `h_addr` en un pointeur sur une structure `struct in_addr`.

La structure `struct hostent` est allouée par le système et la fonction retourne un pointeur sur la structure associée. Il n'est donc pas nécessaire d'allouer une structure avant l'appel à la fonction mais il faut déclarer le pointeur qui sert en retour de la fonction.

Encore une fois, cette structure n'est pas très simple à comprendre pour les programmeurs qui ne maîtrisent pas bien le langage C. Pour cette raison, nous donnons la «formule magique» qui permet d'obtenir le numéro IP d'une machine.

```
1  struct hostent *host;      /* description de la machine serveur */
2  struct sockaddr_in addr;   /* adresse de la sochet du serveur */
3
4  /* Recherche de l'adresse de la machine */
5  host = gethostbyname(nom_machine);
6  if (host == NULL) {
7    /* traitement de l'erreur */
8  }
9
10 /* Recopie de l'adresse IP */
11 addr.sin_addr.s_addr =
12     ((struct in_addr *) (host->h_addr_list[0]))->s_addr;
```

Il existe la fonction réciproque `gethostbyaddr` qui permet d'obtenir le nom d'une machine à partir de son adresse.

Dans certains cas, il n'est possible ni de connaître le nom de la machine sur laquelle s'exécute le programme, ni de l'obtenir par un passage de paramètre ou à partir d'un fichier de configuration. Pour obtenir le nom de la machine locale (pour obtenir ensuite son adresse ou faire `bind` sur une socket), on peut alors utiliser la fonction `gethostname`, définie dans l'en-tête `unistd.h` et dont le prototype est :

```
1  int gethostname(char *name, size_t len);
```

B.4.4 Les ports

Pour localiser une socket sur une machine le système utilise un identificateur local appelé port. Le numéro de port associé à une socket est un entier codé sur deux octets. Les numéros de port inférieurs à 1024 sont réservés aux processus superviseurs (*root*).

Il y a deux manières pour obtenir un numéro de port : le fixer par le programme ou en demander l'attribution au système. Notons que, pour faire communiquer deux programmes, il est nécessaire d'une part qu'un des programmes donne une adresse, donc un numéro de

port, à sa socket et, d'autre part, que l'autre programme connaisse cette adresse, donc ce numéro de port, pour pouvoir contacter le destinataire. De même que pour pouvoir établir une conversation téléphonique, il est nécessaire d'avoir un combiné avec un numéro et un interlocuteur qui connaisse le numéro à appeler.

En fonction du mode d'attribution du numéro de port, les programmes qui communiquent n'ont pas les mêmes garanties, ce qui suppose des interactions différentes. En effet, si nous choisissons de fixer le numéro de port utilisé, la même valeur doit être définie dans les deux programmes. Elle sera définie statiquement dans le code du programme ou donné à partir d'un fichier de configuration. Nous définissons ainsi un point de rencontre entre deux programmes. Si le numéro de port est attribué dynamiquement par le système, il faut alors trouver une solution pour donner la valeur attribuée au programme qui communiquera avec cette socket.

Pour fixer un numéro de port, il suffit de positionner une valeur dans le champ `sin_port` avant de faire appel à la fonction `bind`. Si le programme est appelé à s'exécuter avec des droits étendus (superviseur), le numéro de port peut être choisi de manière quelconque, sinon, pour une exécution en mode utilisateur, il doit être supérieur à 1024. Le problème lié au choix de ce numéro de port consiste alors à être sûr que le numéro de port ne soit pas déjà associé à une socket, ce qui génère une erreur de la fonction `bind`. Généralement, ce n'est pas le cas pour les numéros de port utilisateurs, sauf si des applications particulières s'exécutent sur la machine.

Pour attribuer dynamiquement un numéro de port, le champ `sin_port` est initialisé à 0 avant l'appel à la fonction `bind`. Pour connaître le numéro de port attribué par le système, il faut demander au système qu'elle est l'adresse associée à la socket, après l'appel à la fonction `bind` et en extraire le numéro du port. Ceci se fait à l'aide de la fonction :

```
1 int getsockname(
2   int desc_sock,
3   struct sockaddr *addr,
4   socklen_t *addrlen
5 );
```

Au retour de l'appel, la structure `addr` est initialisée avec l'adresse de la socket, y compris le numéro de port attribué. La fonction retourne une valeur -1 en cas d'erreur.

Une fois ce numéro de port connu, il est nécessaire de le faire parvenir aux interlocuteurs de notre programme. Cela peut se faire simplement en affichant le numéro de port à l'écran. L'utilisateur lit cette valeur et la donne à un autre programme. L'autre programme peut obtenir cette valeur par saisie, lecture dans un fichier, etc.

Attention, le mode de codage utilisé par Internet est de type MSB alors que le codage sur les machines PC est de type LSB. Dans le cas d'un entier sur deux octets comme le numéro de port, le codage est donc réalisé dans l'ordre inverse. De la même manière, les ordinateurs utilisés dans la communication n'utilisent pas forcément le même codage. On peut alors utiliser les fonctions de conversion de l'ordre des octets :

```
1 uint16_t htons(uint16_t hostshort);
2 uint16_t ntohs(uint16_t netshort);
```

La fonction `htons`, «host to network», permet de convertir un numéro de port depuis une représentation machine vers la représentation IP. La fonction `ntohs`, «network to host», permet de convertir un numéro de port depuis une représentation IP vers une représentation machine. Ces fonctions sont dépendantes de la machine sur laquelle elles s'exécutent. Elles peuvent ne rien faire sur certaines machines (par ex. MSB) mais sur d'autres machines (par ex. LSB), elles inversent l'ordre des octets. Sur des machines de type PC, il est impératif de les utiliser même si le programme peut marcher sans y faire appel dans la mesure où, en utilisant le même codage faux sur les deux machines qui communiquent, la valeur est bien la même. Ceci peut cependant générer des erreurs dans la mesure où la valeur donnée à la socket ne correspond pas à la valeur attendue.

Un numéro de port ne peut pas être affiché avec un simple format `"%d"` puisque le type du `port` est `unsigned short`. Il faut donc utiliser le format d'affichage `"%hu"`.

B.4.5 Les services

Les sockets sont souvent utilisées pour mettre en place des applications accessibles à travers le réseau, nous parlons alors de service. La notion est service est bien connue dans le réseau Internet, elle est illustrée par les serveurs web ou service HTTP, les services de messagerie, etc. De la même manière qu'un nom est associé aux machines pour les identifier à la place du numéro IP, des noms de services sont définis pour éviter d'avoir à mémoriser le numéro de port lui-même. Un certain nombre de services sont définis et associés à des numéros de ports fixés. Par exemple, le numéro de port 80 est toujours associé au service HTTP, le numéro 100 au service POP3, etc. Ces associations sont définies le plus souvent sur chaque machine dans le fichier `/etc/services` ou parfois dans des bases de données définies sur le réseau. Un protocole d'accès peut aussi être défini pour un service. Par exemple, certains services ne sont accessibles qu'à travers le protocole UDP.

Le numéro de port associé à un service peut être obtenu à l'aide de la fonction `getservbyname`, dont l'utilisation est similaire à la fonction `gethostbyname`. Le prototype de cette fonction est le suivant :

```
1 struct servent *getservbyname(const char *name, const char *proto);
```

Les chaînes de caractères `service` et `proto` donnent respectivement le service recherché et le protocole utilisé par le service. La structure `struct servent`, valeur de retour de la fonction, est définie par :

```
1 struct servent {
2     char  *s_name;     /* nom officiel du service */
3     char **s_aliases; /* liste d'alias */
4     int    s_port;     /* numero de port */
5     char  *s_proto;    /* protocole a utiliser */
6 }
```

Cette fonction nécessite l'inclusion du fichier :

```
1 #include <netdb.h>
```

De manière similaire à ce qui a été fait avec le nom et le numéro IP de la machine, cette fonction retourne un pointeur sur une structure `struct servent`, allouée par le système. Nous n'aurons donc qu'à déclarer une variable du type pointeur sur cette structure dans notre programme pour lui affecter le retour de la fonction. Les paramètres qui composent la structure nous renseignent sur le nom du service et la liste des alias qui peuvent lui être associés, le numéro de port utilisé par le service et le nom du protocole utilisé pour y accéder. À noter tout de même qu'ici le numéro de port est défini par un type `int` alors qu'il est défini par un type `u_short` dans l'adresse de la socket, difficile de trouver une explication à cette définition. Heureusement les numéros de port sont bien définis en entiers de deux octets dans les structures du protocole IP.

B.4.6 Synthèse

Après toutes les informations que nous venons de voir, il paraît nécessaire de faire une petite synthèse sur la manière de créer une socket, pour remplacer les fonctions `socketUDP`, `socketAddr`, `socketServeur` et `socketClient` utilisées au chapitre précédent. Cette synthèse sera d'autant plus utile que la méthode de création des sockets est généralement toujours la même et que, une fois que nous disposons du code nécessaire à une création, il est souvent suffisant de le reprendre et d'en modifier quelques valeurs pour obtenir un résultat satisfaisant.

Pour créer une socket identifiée, il est donc nécessaire de suivre les étapes suivantes :

1. Créer la socket avec la fonction `socket`.
2. Initialiser une structure de données correspondant à l'adresse qui doit être donnée à la socket. Pour cela, il nous faut obtenir l'entier qui correspond à l'adresse IP de la machine et il faut fixer le numéro de port.
3. Attribuer l'identification à la socket avec la fonction `bind`.

La création de sockets identifiées se fait de la même manière que la socket soit en mode connecté ou en mode non-connecté.

L'exemple suivant illustre la création d'une socket identifiée en mode non-connecté sur le port 2609 en respectant le codage Internet pour le numéro de port :

```
1  int main(int argc, char *argv[]) {
2    int sock; /* descripteur socket */
3    int err;  /* code erreur */
4    struct sockaddr_in addr;  /* adresse de la sochet */
5
6    /* Creation de la socket, protocole UDP */
7    sock = socket(AF_INET, SOCK_DGRAM, 0);
8    if (sock < 0) {
9      printf("Erreur creation de socket\n");
10     return(-1);
11   }
12
13   /* Initialisation de l'adresse de la socket */
```

```
14    memset(&addr, 0, sizeof(addr));
15    addr.sin_family = AF_INET;
16    addr.sin_port = htons(2609);
17    addr.sin_addr.s_addr = INADDR_ANY;
18
19    /* Attribution de l'adresse a la socket */
20    err = bind(sock, (struct sockaddr *)&addr, sizeof(addr));
21    if (err < 0) {
22      printf("Erreur sur le bind");
23      return(-2);
24    }
25
26    return 0;
27  }
```

Dans l'exemple précédent, le numéro IP associé à la socket n'a pas besoin d'être précisé car il s'agit de la machine locale. Par contre, si je désire établir une connexion en mode connecté ou envoyer une donnée en mode non-connecté le champ `s_addr` doit être correctement initialisé. L'exemple suivant montre l'initialisation d'une adresse avec le port 2610 et la machine `smith` :

```
1   int main(int argc, char *argv[]) {
2     int sock; /* descripteur socket */
3     int err;  /* code erreur */
4     struct sockaddr_in addr;  /* adresse de la sochet */
5     struct hostent *host;
6
7     /* Creation de la socket */
8     ...
9
10    /* Initialisation de l'adresse de la socket */
11    memset(&addr, 0, sizeof(addr));
12    addr.sin_family = AF_INET;
13    addr.sin_port = htons(2610);
14
15    /* Recherche de l'adresse de la machine */
16    host = gethostbyname("smith");
17    if (host == NULL) {
18      printf("Erreur gethostbyname \n");
19      return(-2);
20    }
21
22    /* Recopie de l'adresse IP */
23    addr.sin_addr.s_addr =
24        ((struct in_addr *) (host->h_addr_list[0]))->s_addr;
25
26    /* Suite du programme */
27    ...
28
29    return 0;
30  }
```

Dans le cas où la communication se fait en mode non-connecté, la création d'une socket identifiée est suffisante. Pour le mode connecté nous avons, en plus à établir la connexion.

B.5 Établir une connexion

Dans le mode connecté on relie temporairement deux descripteurs de socket pour ne plus préciser l'adresse à chaque échange. Cela implique donc une petite gymnastique préliminaire pour établir la connexion : ce qui était fait automatiquement avec les fonctions `socketClient` et `socketServeur` doit maintenant être réalisé explicitement.

Une fois une socket créée, le client se limite à demander sa connexion au serveur grâce à la fonction `connect` dont le prototype est le suivant :

```
1 int connect(
2   int desc_sock,
3   const struct sockaddr *addr,
4   socklen_t addrlen
5 );
```

L'appel à cette fonction suppose que le serveur ait donné une adresse à sa socket au moment où le client exécute cette fonction. Par contre, il n'est pas indispensable que le client associe une adresse à la sienne pour la connecter avec celle du serveur.

À l'appel de la fonction, le système envoie une demande de connexion au serveur. Cet appel est bloquant tant que la connexion n'a pas été acceptée ou refusée. En cas de problème la fonction retourne un code d'erreur.

La fonction `socketClient` contient donc principalement une création de socket, l'initialisation d'une structure d'adresse identifiant le serveur et l'appel de la fonction de connexion.

Du côté du serveur et donc du côté de la fonction `socketServeur`, nous avons vu qu'il devait disposer d'une socket de connexion pour ne pas mélanger les demandes de connexion et les données envoyées par les clients. Ceci est réalisé en créant une socket par la procédure normale puis en la déclarant comme dédiée aux demandes de connexion grâce à la fonction `listen` dont le prototype est le suivant :

```
1 int listen(
2   int desc_sock, /* descripteur socket a transformer */
3   int backlog    /* nb max demandes en attente */
4 );
```

Cette fonction est appliquée à la socket principale du serveur, créée précédemment avec la fonction `socket`. Cette fonction a pour effet de transformer cette socket en socket de connexion et de créer, associée à la socket, une file d'attente de demandes de connexion. La taille de cette file d'attente est fixée à la valeur du paramètre `backlog`, de taille maximale huit, ce qui devrait permettre de limiter les demandes de connexion en attente. Par exemple, si la valeur de `backlog` est positionnée à deux et que trois clients font appel à `connect` sans que le serveur ne fasse appel à `accept` alors le dernier client devrait recevoir une erreur. Attention ceci ne marche pas sur tous les systèmes, en particulier sous les systèmes Linux

pour lesquels cette valeur n'est pas prise en compte. L'appel à `listen` est non-bloquant. Il rend -1 en cas d'erreur. Une fois l'appel à `listen` réalisé sur une socket, il ne faut plus l'utiliser pour recevoir des données.

La fonction `socketServeur` contient donc principalement une création de socket, l'initialisation d'une structure d'adresse qui est associée à la socket par la fonction `bind` et l'appel à la fonction `listen` sur la socket pour la dédier aux connexions.

Nous avons vu qu'ensuite, le serveur se met en attente de demandes de connexion grâce à la fonction `accept` dont le prototype est le suivant :

```
1 int accept(
2   int desc_sock_conn,     /* socket connexion */
3   struct sockaddr *addr, /* socket demandant la connexion */
4   socklen_t *addrlen      /* pointeur taille de l'adresse */
5 );
```

La fonction `accept` a pour effet de créer une nouvelle socket dont le descripteur est donné en valeur de retour. Cette socket est connectée avec celle qui a demandé la connexion. C'est donc cette nouvelle socket qui sert pour échanger des données avec le client. La socket `desc_sock_conn` reste affectée à la réception de demandes de connexion. Cet appel est bloquant tant que le programme n'a pas reçu une demande de connexion, en attente d'une demande de connexion.

Les derniers paramètres, que nous n'avions pas utilisé jusque là, permettent de recevoir l'adresse de l'émetteur dans une structure d'adresse, que nous devons pré-allouer. Cette structure est identifiée par le pointeur `addr` de type `struct sockaddr *` et un pointeur sur une variable de type `socklen_t`, contenant la taille `addrlen` de l'adresse pré-allouée. En retour de la fonction, cette variable contient la taille de l'adresse reçue, raison pour laquelle il est nécessaire de faire un passage de paramètre par adresse (au sens pointeur). Si nous n'avons pas besoin de cette adresse, il est possible de passer la valeur `NULL` pour les paramètres `addr` et `addrlen`.

B.6 La communication

Les principales difficultés dans l'utilisation de l'interface des sockets proviennent de la création, nous venons donc de les passer, et les fonctions que vous avez utilisé au chapitre précédent pour la communication avec les sockets sont déjà celles de l'interface, vous avez donc connaissance du principal. Nous allons cependant revoir la partie de communication dans le but de donner quelques détails complémentaires.

B.6.1 Données échangées

Nous avons vu que la communication sockets ne se fait que sur la base d'échanges de données vues par les programmes sous la forme d'un *buffer*. Cela signifie que le message est constitué du contenu d'une zone de mémoire, recopiée dans le message envoyé. De la même manière, ce que reçoit le récepteur est recopié dans une zone de mémoire.

Les données ainsi échangées le sont sous leur forme brute, non typées. Si vous avez, par exemple, défini une structure dans votre programme émetteur et qu'il l'envoie au récepteur, celle-ci est reçue sans indication de typage sur les champs de la structure. En fait, la structure est reçue telle qu'elle était mémorisée chez l'émetteur. Ceci implique qu'une donnée émise avec un certain type peut être reçue avec un type différent sans erreur. Il est donc nécessaire d'être prudent sur le type de réception des messages. Nous verrons au chapitre sur le client/serveur comment mettre en place les procédures nécessaires au typage des structures.

En ce qui concerne la réception, il est nécessaire de préparer une zone de mémoire pour la réception du message. Cette zone de mémoire est passée à la fonction de réception sous la forme d'un pointeur sur le début de la zone et de la taille de la zone. Pour garantir l'intégrité de vos données, la fonction de réception ne recopie jamais en dehors de cette zone de mémoire. Cela signifie que, si la taille des données arrivées dépasse cette zone, seules la partie qui peut être stockée dans la mémoire l'est, à partir du début des données. Le reste est laissé dans un tampon du système en attendant une demande de réception ultérieure. Ainsi, si l'émetteur a envoyé une chaîne de cent caractères et que je ne demande qu'à en recevoir cinquante, seuls les cinquante premiers caractères de la chaîne sont reçus, les autres restent en attente

Attention, nous avons vu que les chaînes de caractères sont, en langage C, codées sous la forme d'une suite de caractères terminée par un octet nul. Cet octet nul est utilisé dans les fonctions de manipulation de chaînes comme point de fin de traitement. Dans notre exemple, la chaîne de cent caractères sera suivie d'un 101ième caractère, un octet nul. Si donc nous ne recevons que cinquante caractères, l'octet nul n'en fera pas partie et nous ne pouvons utiliser les fonctions de traitement des chaînes de caractères sans risque d'erreur. Il est cependant possible d'ajouter manuellement cette valeur.

Le mode d'échange des données diffère suivant le protocole de communication utilisé :

Avec le protocole TCP : les données échangées sont transmises sous la forme d'un flux. C'est-à-dire qu'aucun marqueur n'est positionné entre les différents envois et il n'est pas possible de déterminer la taille d'un envoi à partir du *buffer* de réception. Ainsi, il est possible de recevoir deux envois successifs de cinquante octets aussi bien sous la forme d'un *buffer* de cent octets que de dix *buffeurs* de dix octets. C'est à l'application de réaliser la délimitation des données si elle en a besoin. En utilisant le protocole de communication TCP, nous avons la garantie que l'ordre des données est préservé à la réception.

Avec le protocole UDP : les données échangées sont transmises sous la forme de DataGrammes, ou paquets : elles sont envoyées et reçues sous cette forme. Du point de vue de l'envoi cela signifie que la taille des données est limitée à la taille maximale d'un paquet (généralement $\simeq$ 65Ko). Du point de vue de la réception, cela signifie que deux envois successifs sont reçus indépendamment et qu'une réception reçoit un paquet ; si le buffeur de réception est trop petit alors les données du paquet qui ne rentrent pas dans le buffeur sont perdues. Avec le protocole UDP nous n'avons pas de garantie sur l'ordre de réception des parquets ni sur leur bonne réception puisque

des données peuvent être perdues.

B.6.2 Le mode non-connecté

Dans le chapitre précédent, nous avons évité au maximum l'utilisation des adresses dans la communication. Ainsi, dans les fonctions `sendto` et `recvfrom`, les champs d'adresse ont été remplis à partir de fonctions pré-définies, ou ignorés. Pour une utilisation avancée de ce mode de communication, il peut être utile de savoir se servir de ces champs.

Rappelons que la fonction `sendto` a pour prototype :

```
1 ssize_t sendto(
2   int desc_sock,     /* descripteur socket */
3   const void *buf,   /* pointeur sur zone de memoire */
4   size_t len,        /* taille de la zone de memoire */
5   int flags,         /* options d'envoi */
6   const struct sockaddr *to, /* id socket destinatrice */
7   socklen_t addrlen /* taille de l'adresse */
8 );
```

Nous pouvons maintenant voir que les paramètres cinq et six doivent être remplis avec une structure d'adresse équivalente à celle utilisée dans la fonction `bind`. Il s'agit de la structure d'adresse identifiant la socket qui doit recevoir le message.

D'autre part, nous avions vu que la fonction est bloquante tant que les données ne sont pas parties de la machine. Il faut cependant préciser que si la taille du message est trop importante pour être envoyé en une seule fois, la fonction rend une erreur.

Reprenons également la fonction `recvfrom` dont le prototype est le suivant :

```
1 ssize_t recvfrom(
2   int desc_sock,       /* descripteur socket */
3   void *buf,           /* pointeur sur zone de memoire */
4   size_t len,          /* taille de la zone de memoire */
5   int flags,           /* options de reception */
6   struct sockaddr *from, /* id envoyeur */
7   socklen_t *addrlen  /* taille de l'adresse */
8 );
```

De même que dans la fonction `accept` les derniers paramètres permettent de recevoir l'adresse de la socket avec laquelle le serveur est connecté. Le mode de passage des paramètres est donc identique et la taille de l'adresse doit être initialisée avant l'appel à la fonction. Si nous n'avons pas besoin de cette adresse, il est possible de passer la valeur `NULL` pour les paramètres `from` et `addrlen`.

Remarque L'interface initiale des sockets, qu'il n'est pas rare de rencontrer encore sur certains systèmes, utilise des types `char *` à la place des types `void *` pour le pointeur sur les données, et des types `int` à la place de `size_t`, `ssize_t` et `socklen_t`.

B.6.3 Le mode connecté

Du fait que les adresses ne sont pas utilisées dans les échanges en mode connecté, nous avons utilisé les fonctions de l'interface dans le chapitre précédent. Nous pouvons simplement ajouter que la fonction `send` peut rester bloquée tant que l'ensemble des données passées en paramètre ne sont pas envoyées, même si cela doit se faire en plusieurs fois. Ce n'est pas le cas de la fonction `sendto` qui rend une erreur si elle ne peut envoyer en une fois.

Généralement on choisit de travailler en mode connecté si l'accès au serveur demande plusieurs requêtes, peut-être anonymes, doit être sûr, etc. Si l'accès au serveur est ponctuel ou si on communique avec différents processus, il vaut mieux utiliser le mode non-connecté qui est plus rapide.

Remarques et mises en garde

- L'ensemble des fonctions de l'interface est constitué par des fonctions système. Le déroulement de ces fonctions dépend donc des structures de données internes au système et il n'est pas possible de prédire leur bon déroulement. Par exemple, la fonction de création des sockets fait appel à des structures de données de taille fixe, ce qui fait que si d'autres programmes ont déjà consommé ces ressources, il ne sera pas possible de créer la socket. Il est donc indispensable de tenir compte de ce fait dans vos programmes. Cela doit se traduire par le test systématique des retours des fonctions systèmes et par un traitement d'erreur qui tient compte de ce risque. L'appel à la fonction `exit` constitue un traitement sommaire mais efficace dans ce cas.
- Les fonctions `recvfrom` et `accept` retournent l'adresse de l'émetteur, il faut donc bien se souvenir que la taille de l'adresse est passée par pointeur en paramètre et qu'elle doit être initialisée.
- Dans la mesure où les sockets sont identifiées dans le système comme des fichiers, il est possible d'utiliser certaines fonctions de manipulation des fichiers sur les sockets. Ainsi, `read` fonctionne comme `recv` et `recvfrom` avec trois arguments et `write` fonctionne comme `send` avec trois arguments. `write` ne peut pas s'utiliser à la place de `sendto` car il faut pouvoir préciser le destinataire. Cette notation peut être rencontrée dans des programmes utilisant des sockets. Il est cependant recommandé d'utiliser les fonctions spécifiques pour plus de clarté.
- Les fonctions d'interface des socket sont des fonctions systèmes. Elles retournent donc toutes un code d'erreur dans la mesure où nous n'avons aucune garantie de leur bon déroulement. Un retour d'erreur d'une fonction système est toujours (sous Unix) caractérisé par la valeur -1. Le code d'erreur spécifique est stockée dans la variable `errno`. Il est possible de traiter les erreurs systèmes de trois manières différentes. Dans tous les cas il faut inclure le fichier système `errno.h` :

  1. `fprintf(stderr, "%d", errno)`, permet de faire afficher le code d'erreur puis d'aller voir dans `/usr/include/errno.h` à quoi il correspond.

2. `void perror(const char *msg)`, affiche un message d'erreur et la chaîne donnée en paramètre, en fonction de la valeur contenue dans `errno`
3. `char *strerror(int errno)`, rend une chaîne qui contient un message d'erreur en fonction de `errno`.

B.7 Autres fonctions

En cas de besoin, on peut connaître l'adresse de la socket avec laquelle on est connecté grâce à la fonction `getpeername` dont le prototype est le suivant :

```
1 int getpeername(
2   int desc_sock,  /* descripteur socket connectee */
3   struct sockaddr *addr,
4   socklen_t *addrlen
5 );
```

Au retour de la fonction les variables `addr` et `addrlen` contiennent respectivement l'adresse de la socket avec laquelle la socket identifiée par `desc_sock` est connectée et la longueur de cette adresse.

B.8 Sockets non bloquantes

Certains schémas de communication ont besoin de pouvoir tester si des données sont disponibles sur la socket sans bloquer lorsqu'il n'y en a pas. C'est, par exemple, le cas d'un serveur qui veut traiter simultanément plusieurs clients : s'il demande à recevoir sur une socket avec la fonction `recv` et que le client qui y est connecté n'a rien envoyé alors le programme va être bloqué tant que ce client n'envoie rien. Cela peut aussi être le cas lorsque un récepteur reçoit une suite de données dont il ne connaît pas la taille, à chaque réception, il ne sait pas s'il reste des données à lire dans la socket. Pour résoudre ce problème, il est possible de rendre une socket non bloquante. C'est-à-dire que les opérations effectuées dessus ne bloquent plus le processus.

Les appels bloquants aux socket peuvent être évités en utilisant la fonction `ioctl` avec le flag `FIONBIO`. Cette fonction est une fonction de contrôle des entrées-sorties en général. Elle travaille sur les fichiers. Dans ce cas, elle s'utilise de la manière suivante :

```
1 int block;
2 /*
3  * block = 1, pour rendre non bloquante,
4  * block = 0, pour rendre bloquante
5  */
6
7 err = ioctl(desc_sock, FIONBIO, &block);
```

Dans le cas où la socket ne contient pas de données ni de demande de connexion ou lorsque le serveur n'accepte pas immédiatement une connexion, les appels aux fonctions bloquantes retournent une erreur :

— pour `accept`, `send`, et `recv` on a `err` égal à -1 et et l'erreur `errno` est `EWOULDBLOCK` ;
— pour `connect`, on a `err` égal à -1 et l'erreur `errno` est `EINPROGRESS`.

Par exemple :

```
1 int block = 1;
2 err = ioctl(desc_sock, FIONBIO, &block);
3
4 err = recv(...);
5 if ((err < 0) && (errno == EWOULDBLOCK)) {
6
7   /* rien dans la socket */
8
9 }
```

L'utilisation des sockets non-bloquantes permet de se mettre en attente sur plusieurs sockets en même temps par scrutation successives. Attention, dans ce cas le processus reste actif et, s'il passe son temps uniquement à scruter les sockets non-bloquantes, il consomme de la puissance CPU pour rien. Il est donc possible de mettre un processus en attente sur plusieurs sockets.

B.9 Multiplexage des appels socket

Le multiplexage des appels socket consiste à mettre en attente un programme sur plusieurs sockets simultanément et de savoir, après l'attente, celle qui est prête à envoyer ou à recevoir. Il est réalisé avec la fonction `select`. Encore une fois, cette fonction a été conçue pour les fichiers mais peut être utilisée sur les sockets puisque leur interface de programmation est intégrée à celle des fichiers.

```
1 int select(
2   int nfds,             /* = FD_SETSIZE */
3   fd_set *readfds,      /* ensemble des desc lecture */
4   fd_set *writefds,     /* ensemble des desc ecriture */
5   fd_set *exceptfds,    /* ensemble des desc exception */
6   struct timeval *timeout  /* temps d'attente */
7 );
```

L'utilisation de cette fonction suppose les inclusions suivantes :

```
1 #include <sys/types.h>
2 #include <sys/time.h>
```

L'appel à la fonction `select` suppose la constitution d'ensembles de descripteurs de fichiers : un pour les descripteurs en lecture, un pour les descripteurs en écriture et un pour les descripteurs d'exception.

Le premier paramètre de la fonction `select` est la taille maximale du plus grand des ensembles de descripteurs passés en paramètre, plus 1. En fait un ensemble de descripteurs peut être de taille différente en fonction du système hôte. Cette taille dépend du nombre

de descripteurs de fichiers ouverts maximum par processus, ce qui est défini par un paramétrage du système d'exploitation et donné par la constante `FD_SETSIZE`. Or cette taille peut être grande[1] et il vaut mieux éviter de tester le maximum possible pour chacun des ensembles. Pour cette raison, le premier paramètre de la fonction `select` donne la taille maximale des ensembles `fd_set`.

Les descripteurs de lecture sont ceux sur lesquels nous appelons des fonctions de lecture : `read` pour les fichiers, `recv`, `recvfrom` ou `accept`. Les descripteurs d'écriture sont ceux sur lesquels nous appelons des fonctions d'écriture : `write` pour les fichiers, `send` ou `sendto` pour les sockets. Les descripteurs d'exceptions sont les sorties d'erreur par exemple mais nous ne les utiliserons pas avec les sockets. Le type associé aux ensembles de descripteurs est `fd_set`.

Pour constituer les ensembles des descripteurs on utilise les fonctions suivantes :

```
1  int fd;
2  fd_set fdset;
3
4  FD_ZERO (&fdset)
5  /* initialise l'ensemble */
6
7  FD_SET (fd, &fdset)
8  /* met le descripteur fd dans l'ensemble fdset */
9
10 FD_CLR (fd, &fdset)
11 /* supprime le descripteur fd de l'ensemble fdset */
12
13 FD_ISSET (fd, &fdset)
14 /* teste si le descripteur est pret apres un appel a select */
```

Il faut noter que les ensembles de descripteurs sont passés par adresse car ils sont modifiés par la fonction. L'utilisation de la fonction `FD_ZERO` est ainsi recommandée avant chaque utilisation et avant chaque ré-utilisation de l'ensemble pour le "nettoyer" puisqu'il est modifié par l'appel.

Il n'est pas nécessaire d'avoir un ensemble de chaque type pour utiliser la fonction `select`. Une valeur `NULL` peut-être passée en paramètre pour les ensembles que nous ne souhaitons pas tester.

En fonction du code de retour on peut savoir par quoi l'attente de `select` a été interrompue. Si la valeur de retour vaut zéro, elle est sortie par timeout, -1 indique une erreur, et $n > 0$ le nombre de descripteurs prêts.

En sortie de l'appel à `select`, si au moins l'un des descripteurs est prêt (valeur retournée positive), alors seuls les descripteurs prêts sont maintenus dans les ensembles passés en paramètre. L'utilisation de la fonction `FD_ISSET` permet alors, en testant successivement tous les descripteurs initialement positionnés, de savoir lequel est prêt.

1. Limitée à 1024 sous linux mais sans limite dans la norme POSIX. A noter que, dans les applications un peu anciennes, il était courant d'utiliser systématiquement la valeur `FD_SETSIZE` comme taille des ensembles car le nombre maximum de fichiers ouverts étaient plus petit, de l'ordre de quelques dizaines.

Puisque la fonction `select` modifie le contenu des ensembles de descripteurs, il est nécessaire de réinitialiser ces ensembles avant chaque nouvel appel à la fonction.

La structure `timeout` de type `struct timeval` donne le temps maximum d'attente dans `select`.

```
1 struct timeval {
2   long tv_sec;  /* secondes */
3   long tv_usec; /* microsecondes */
4 };
```

Si le paramètre `timeout` a la valeur `NULL`, la fonction `select` n'a pas de délai de garde. Si la structure `struct timeval` est initialisée à 0, il n'y a pas d'attente mais les descripteurs sont quand même testés.