Programmation réseau avec Java

1/7 – Introduction

Olivier Ricou

19 avril 2010

On abordera dans ce cours :

- quelques principes de programmation réseau,
- les transferts de données,
- la programmation concurrentielle (Thread),
- les méthodes d'invocation distantes (RMI),
- les Web Services,
- la sécurité sous Java,
- J2EE, Globus.

Vous présenterez des bibliothèsques ou outils spécifiques (10 à 15 mn chaque groupe).

Les transparents sont disponibles sur `www.lrde.epita.fr/~ricou`.

1. Programmation réseau

Il s'agit d'un domaine vaste, interprété de façon différente suivant les personnes. Cela peut être

- la programmation des couches basses (TCP, UDP) via la mise en place de sockets,
- la mise en place de communications entre serveurs web,
- les agents,
- le calcul distribué.

On retiendra comme points communs :

L'envoi et la reception de données à travers le réseau.

La mise en place et l'invocation de services distants.

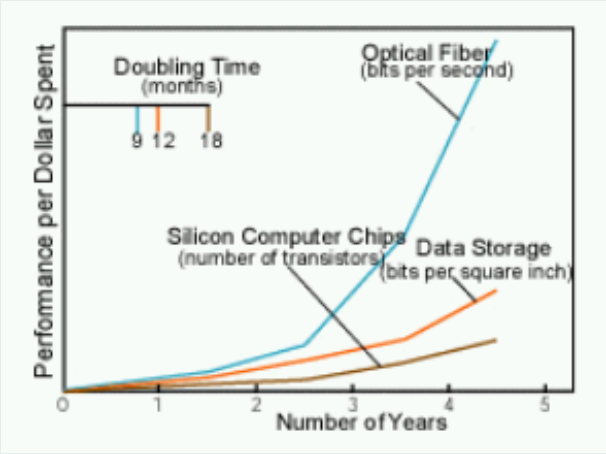
1.1. Pourquoi faire de la programmation réseau

Parceque c'est faisable, utile et rentable. La vitesse des réseaux augmente plus vite que celle des machines.

- le nombre de transistors dans un processeur double tous les 18 mois
- la vitesse des réseaux double tous les 9 mois

Entre 1986 et 2000 :

- les CPU : x 500
- les réseaux : x 340 000



graphe de Cleo Vilett (American Scientific janv. 2001)

Ce résultat a été établi en regardant la progression des débits des fibres optiques.

1.2. Quelques exemples

Toutes ces choses font intervenir de la programmation réseau :

ping

connexion à distance (ssh, telnet)

Seti

serveur web (apache)

eMule

ASP

serveur de fichier(nfs)

jeu de Go (IGS, cgoban)

BitTorrent

Corba

serveur d'impression (cups)

RMI

SOAP

serveur d'horloge (ntp)

irc

2. Différents modèles

– le modèle client-serveur
– le modèle à 2 niveaux, 3 ou plus (2-tiers, multi-tiers)
– le pair à pair (peer to peer)
– le modèle distribué
– le modèle du parallélisme

Note : ces modèles font intervenir des clients et des serveurs ce qui n'implique pas d'avoir plusieurs machines mais plusieurs processus.

Exemple : Ce portable a un serveur web qui tourne, serveur qu'on peut consulter depuis ce même portable avec un navigateur !

2.1. Le modèle client-serveur

Il y a principalement 2 modèles de client-serveur :

Fat server, thin client C'est l'ancien modèle avec les gros ordinateurs qui font tous les calculs et les petits clients qui ne servent qu'à afficher.

Exemple : les terminaux X

Fat client C'est le modèle le plus courant. Le client fait plus qu'afficher, il calcule aussi à partir des données que lui envoie le serveur.

Exemple : le web, encore plus visible avec les javascript et les applets

Problème : Le serveur peut rapidement devenir le goulot d'étranglement, que ce soit dans une entreprise ou sur Internet (serveur slashdoté).

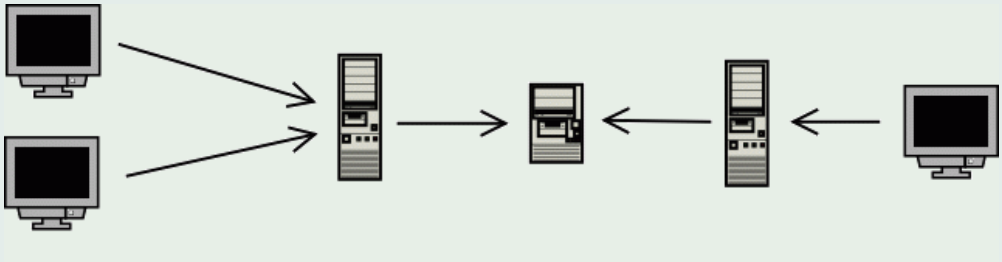
2.2. Les modèles multi-couches *(multi-tiers)*

2 couches : c'est le modèle client-serveur

L'architecture 3 couches

Avec trois niveaux on peut mieux répartir le travail ce qui permet de faire évoluer plus simplement le système.

1. interface homme-machine *Ex : un client lourd usuel*
2. application logique *Ex : un CGI qui va appeler une BD*
3. serveur de données *Ex : LDAP, SQL...*

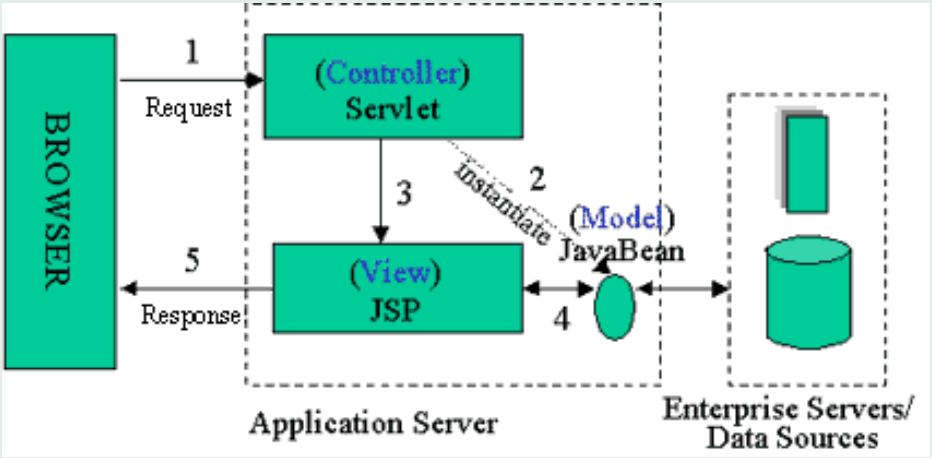


Exemple : La plupart des serveurs web dynamiques

L'architecture multi-couches

On continue à découper le travail en tranche, ce qui permet toujours une meilleure répartition de la charge et de distribuer le travail de programmation dans chaque niveau (chacun est responsable de son module/niveau), l'architecte des communications.

L'application du principe MVC (Modèle-Vue-Controleur) mène aussi à un système multi-niveaux.



2.3. Le pair à pair (peer to peer)

Il n'y a plus de client ni de serveur, chaque nœud joue les deux rôles.

Dans certains cas il existe néanmoins un nœud central pour guider les requêtes. C'était le cas de Napster.

Le *pur* pair à pair n'a pas de nœud central. Toutes les machines jouent le même rôle. C'est le cas de Gnutella.

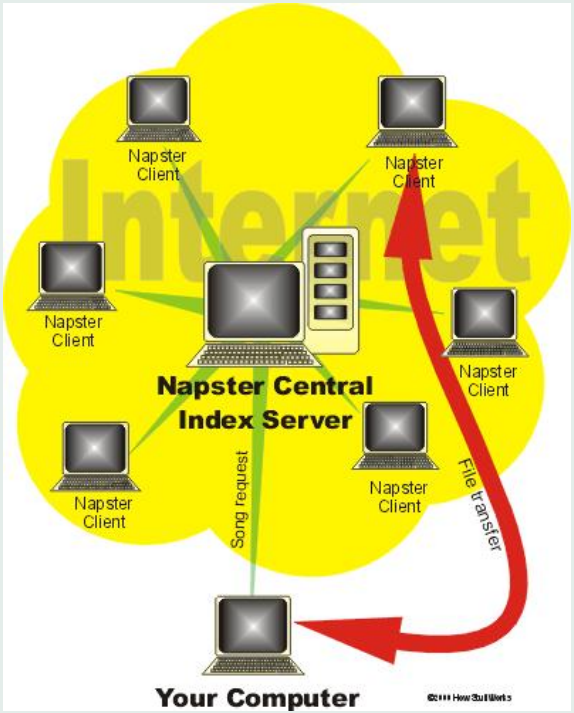
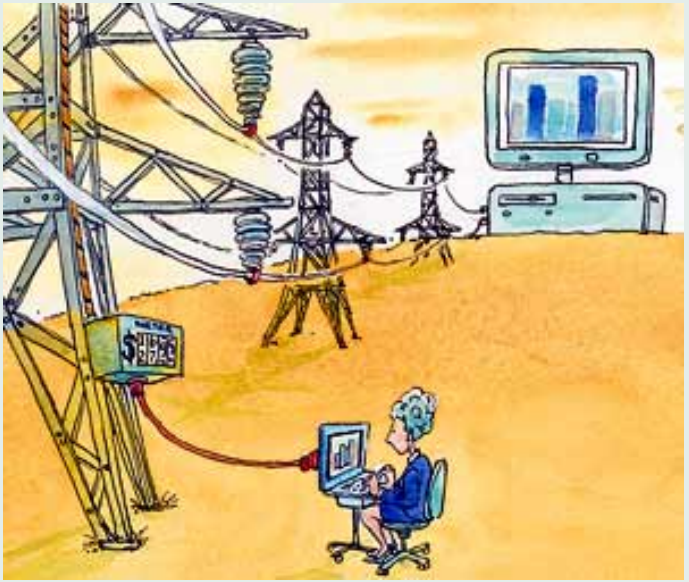


image du site HowStuffWorks

2.4. Le calcul distribué

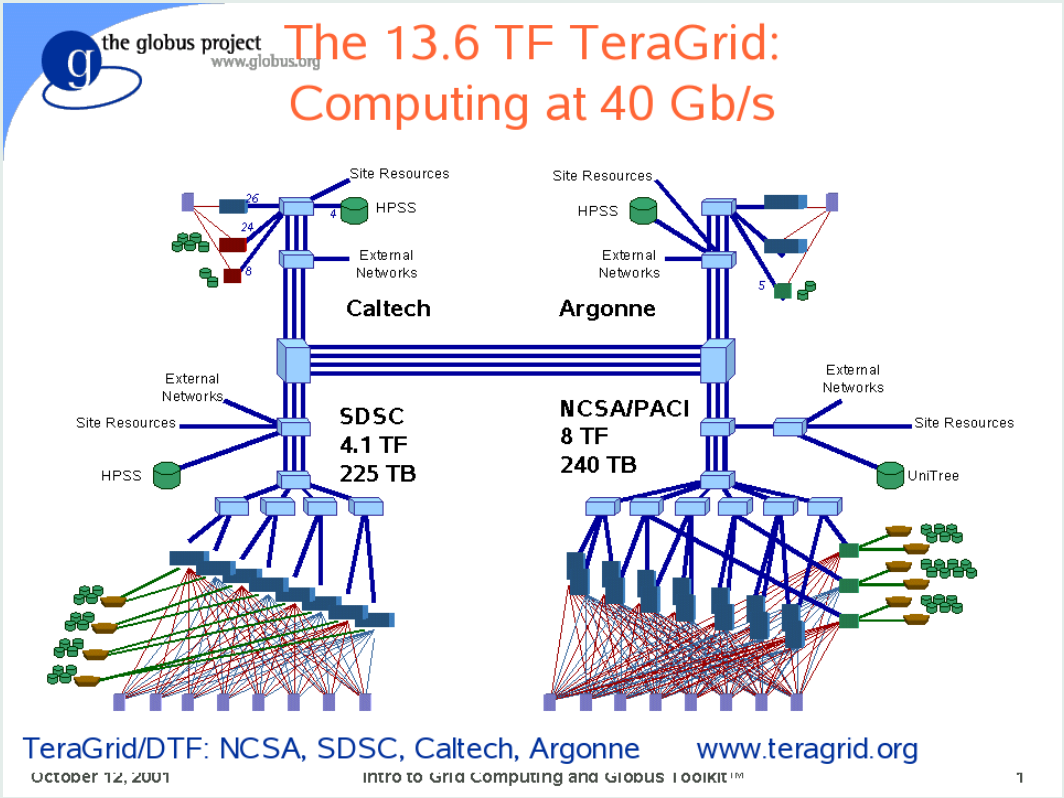
Des projets SETI@home Genome@home Distributed.net EGEE



source : The economist

Des logiciels Globus Condor Sun Grid Engine BOINC

Des protocoles Web Services Resource Framework (WSRF)
Open Grid Services Architecture (OGSA)



Une grille de calcul offre une puissance et disponibilité de calcul et du stockage qu'il est difficile/cher d'obtenir autrement.

2.5. Le calcul parallèle

Depuis pratiquement toujours, les ordinateurs le plus puissants comprennent plusieurs processeurs et un réseau pour les relier.

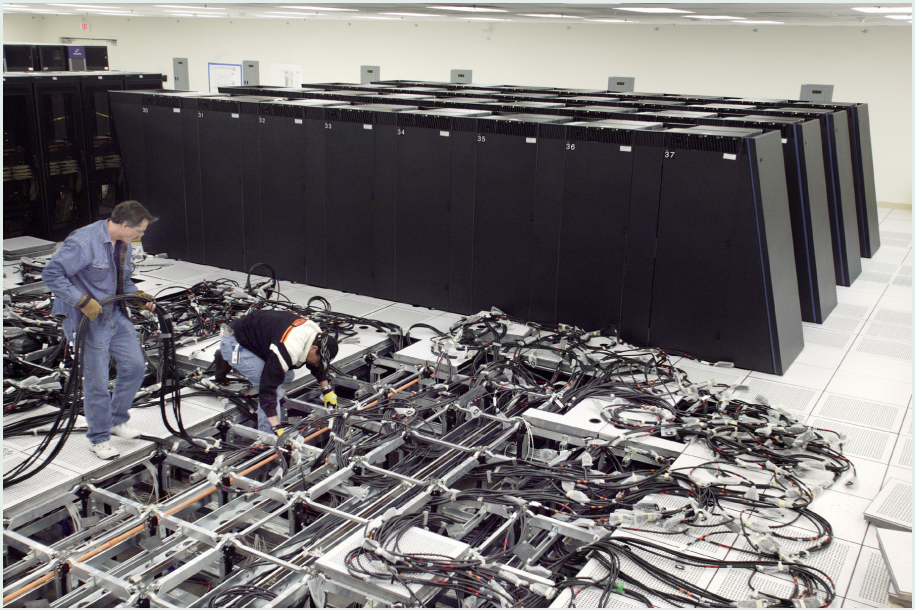


FIG. 1 – Le cablage du BlueGene du LLNL (106 496 processeurs)

Le calcul parallèle repose sur un réseau fiable et très rapide.

Les super ordinateurs actuels sont de MIMD (Multiple Instruction, Multiple Data) qui peuvent découpler le travail tant au niveau des tâches que des données à traiter.

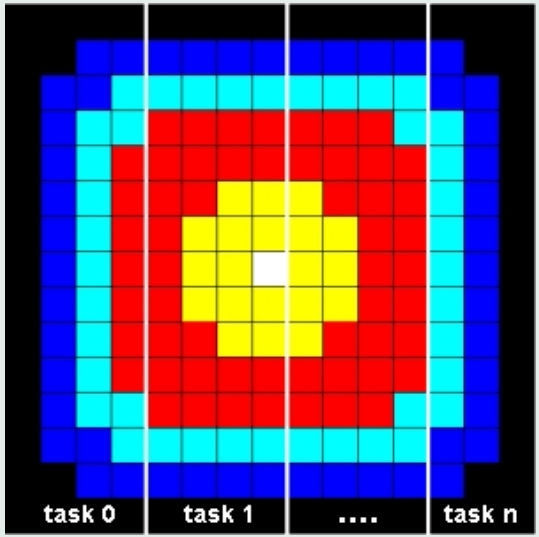


FIG. 2 – Partition d'un domaine de calcul

Java n'a pas de bibliothèque interne dédiée calcul parallèle actuellement.

3. Java et le réseau

La programmation réseau est incluse dans Java depuis la 1ère version.

3.1. L'évolution du réseau dans Java

Java 1.0 comprend les bases de la programmation réseau :

- la possibilité de se connecter à un ordinateur distant (aspect client) et d'attendre une nouvelle connexion (aspect serveur),
- la gestion des URL.

La programmation réseau reste frustre. Programmer un timeout est pénible.

Java 1.1

- ajout des RMI, Remote Method Invocation, qui permet d'executer des méthodes d'objets distants,
- ajout du multicast,
- les `Socket` et `SocketHandler` ne sont plus `final`.

Java 2.0 (1.2) étend la notion de programmation distribuée,

- introduction des Servlets et des JavaServer Pages (JSPs) pour faire du Web avancé,
- ajout des Enterprise JavaBeans (EJBs), la suite des RMI,
- les objets distants devenant des composants du serveur EJB, on peut profiter des services liés comme les transactions et la sécurité.

Java 1.4 s'interresse aux couches de bas niveau

- création du paquet `java.nio` qui complète le paquet d'E/S `java.io`,
- pattern matching à la perl ajouté,
- les `Channels` qui permettent des E/S non-bloquantes multiplexées (utile pour écrire un serveur qui tient la charge)

Java 1.5

- un paquetage d'utilitaire pour la concurrence, `java.util.concurrent`,
- réécriture des RMI avec la génération dynamique des *stub*s.
- ajouts dans les flux : `printf`, `Scanner`

Java 1.6

– intègre les Web Services

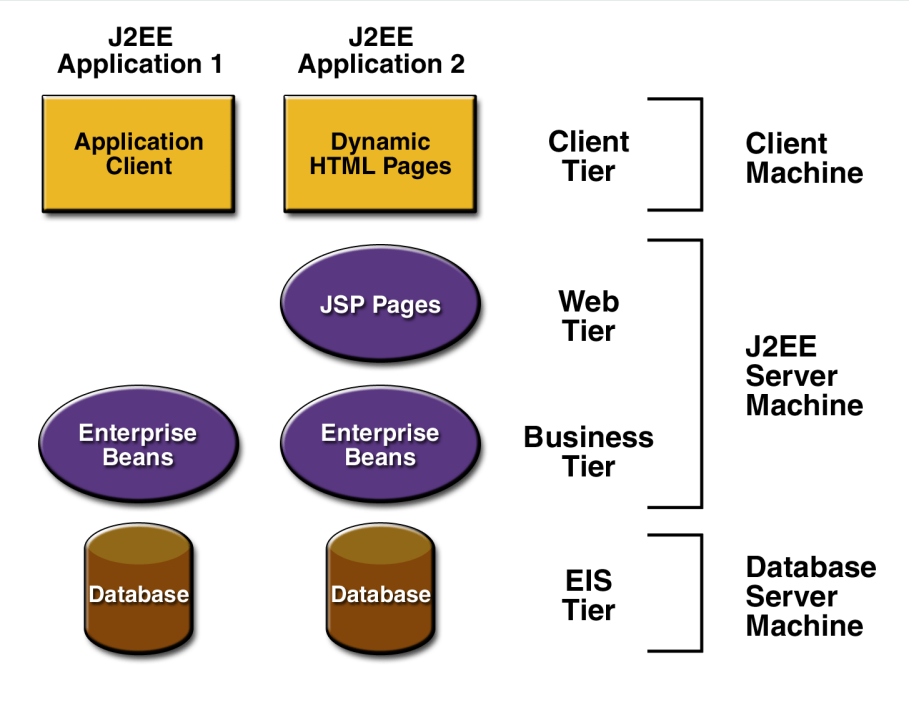
Il faut aussi tenir compte des paquets extérieurs ajoutés comme :

– `org.omg.CORBA` pour ce qui touche CORBA,
– `javax.rmi`, RMI-IIOP, pour l'interconnexion entre les RMI et CORBA,
– `JXTA` pour faire du peer-to-peer.

et tout ce qu'on trouve dans J2EE.

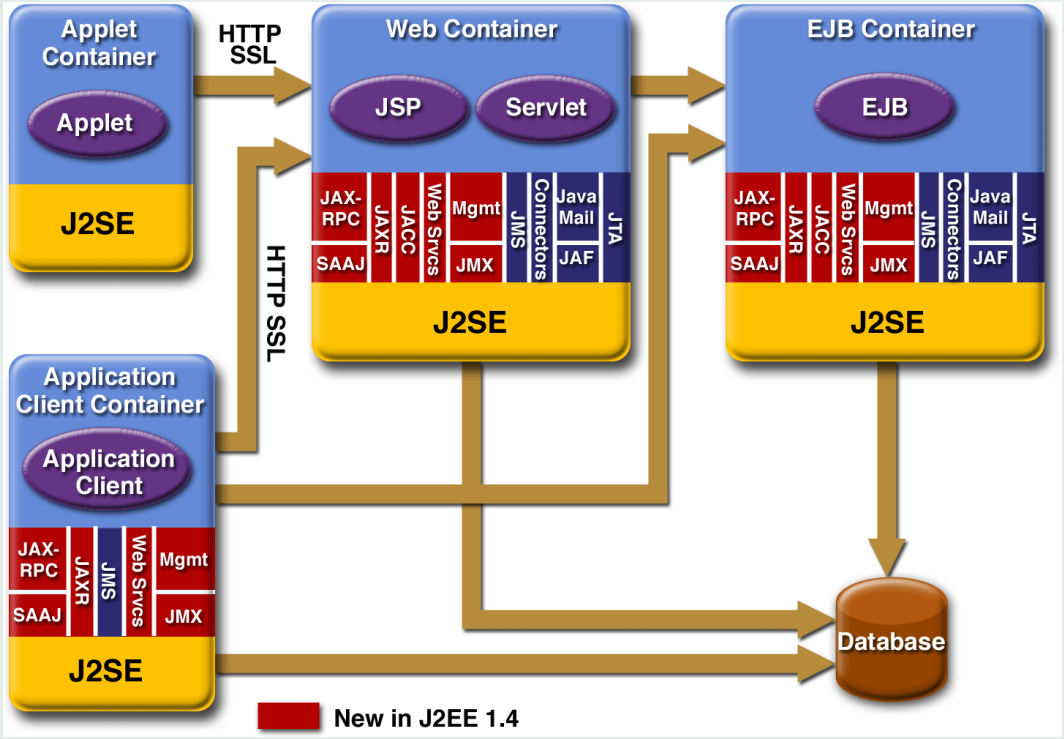
3.2. La Plateforme Java 2 Entreprise Edition (J2EE)

L'idée de la plateforme J2EE est de fournir un environnement simple et unifié pour développer des applications réseau.



Le multi-tiers avec J2EE

Ce qui donne dans les détails (en rouge les composants de Java 1.4)



APIs de J2EE pour notre exemple

3.3. Un exemple de programmation réseau directe (socket)

DayTime.java

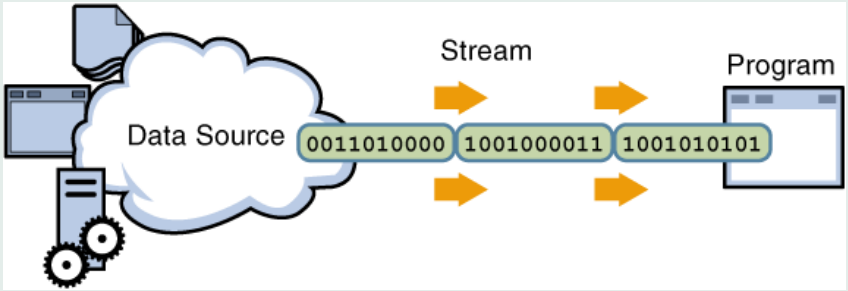
```
1  import java.io.*;
2  import java.net.*;
3
4  public class DayTime
5  {
6    String hostname = "toto.edu";
7    int portnumber = 13;
8
9    public static void main(String[] args)
10   {
11     DayTime time = new DayTime();
12
13     System.out.println("It's "+time.getTime() +
14                        " at "+time.hostname);
15   }
```


```
17    public String getTime()
18    {
19      String rep = "";
20
21      try
22      {
23        Socket sock = new Socket(hostname, portnumber)
24        InputStreamReader is =
25          new InputStreamReader(sock.getInputStream())
26        BufferedReader br = new BufferedReader(is);
27
28        rep = br.readLine();
29        sock.close();
30      }
31      catch (Exception e) { rep = "?"; }
32      return rep;
33    }
34  }
```

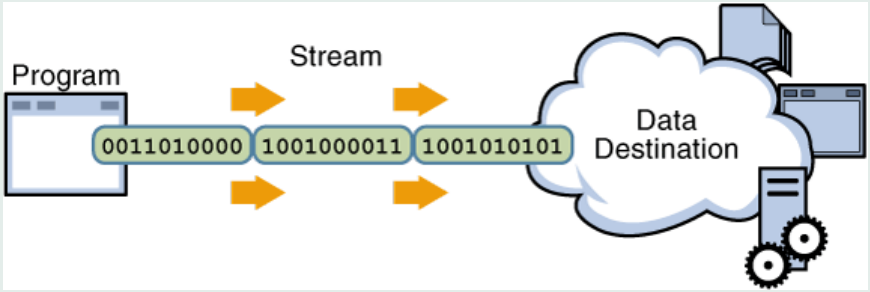
4. Les E/S en Java

Sous Java un flux est une suite continue d'octets ou de caractères.

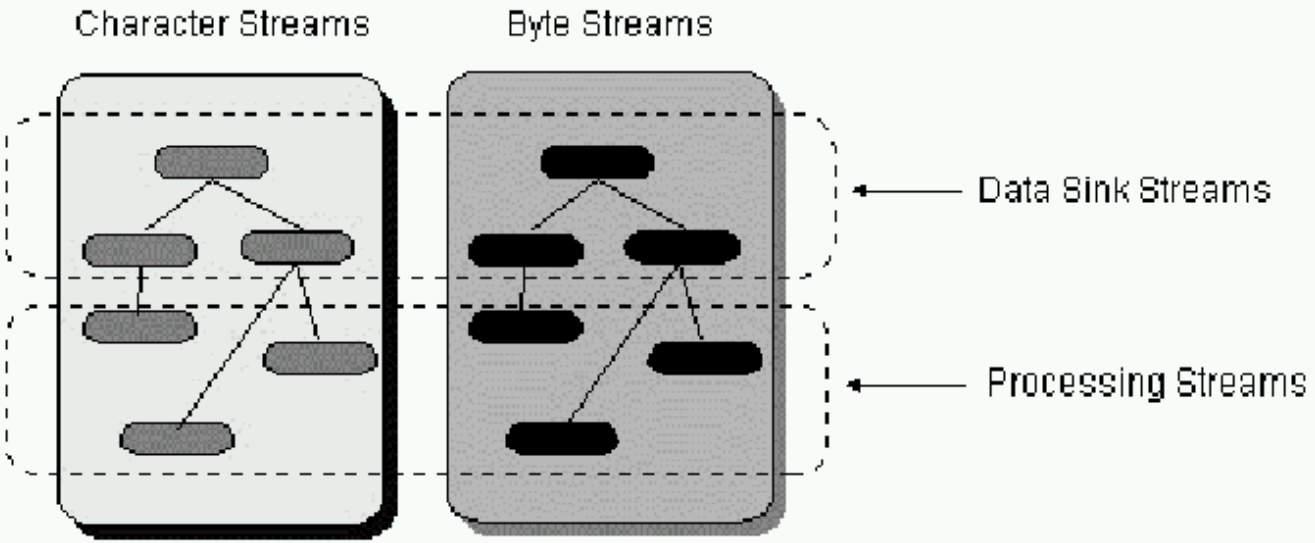
Un flux entrant dans votre programme est un `InputStream`



Un flux émit par votre programme est un `OutputStream`.

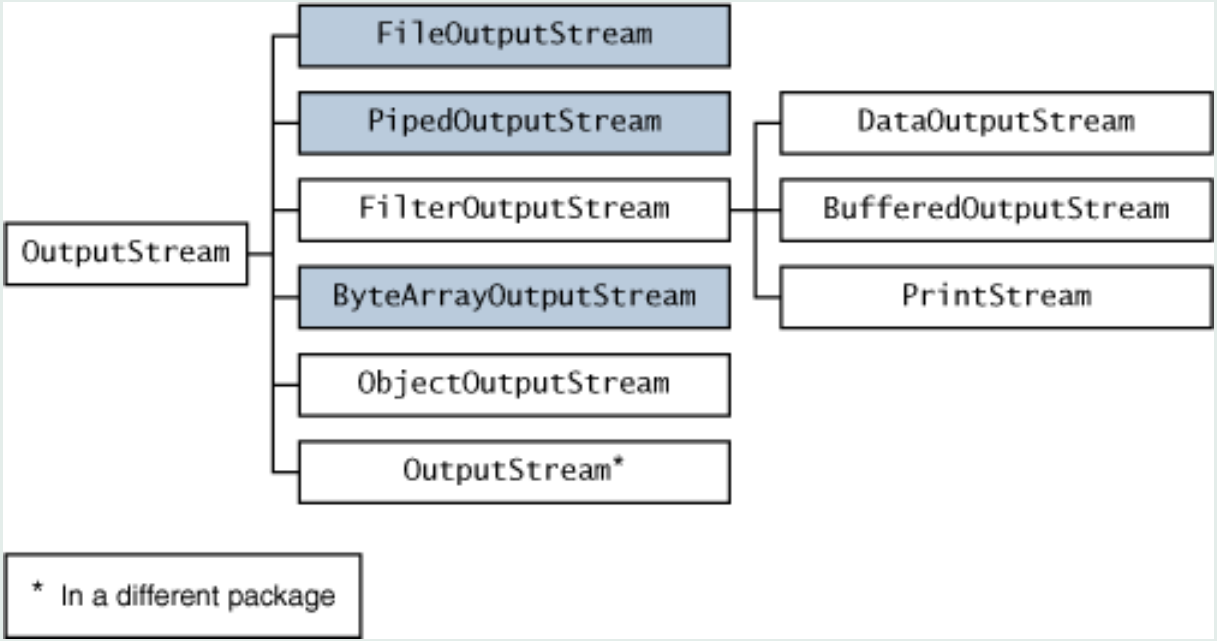


Il existe deux familles de flux, les flux d'octets et les flux de caractères. Pour les caractères, Java utilise l'Unicode stocké sur 16 bits.



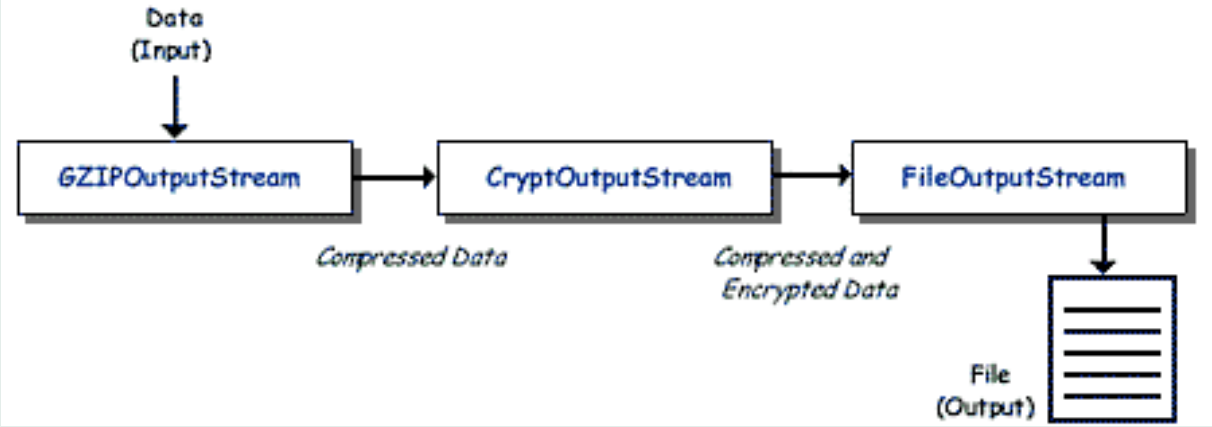
4.1. Les flux d'octets

Les `OutpuStream` finaux sont les puits (en grisé). Ceux qui s'y raccordent sont les fitres.

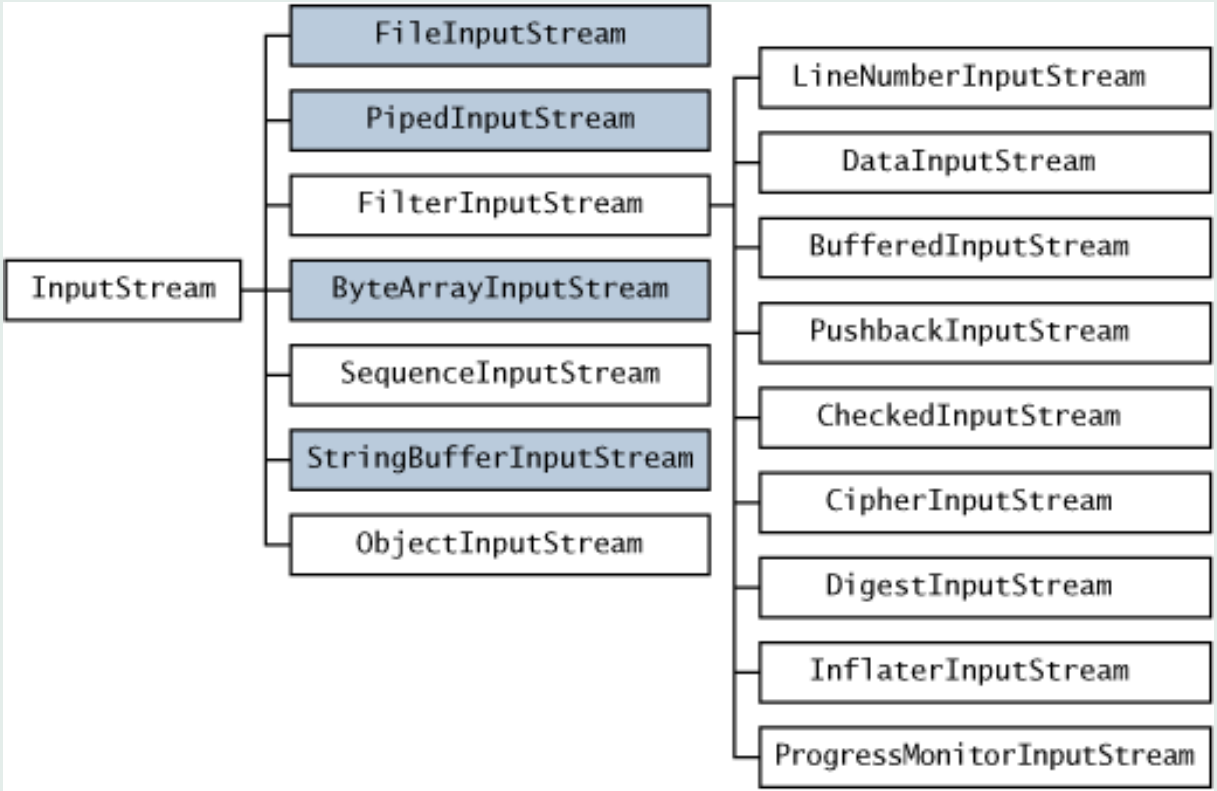


Ainsi on emboite les flux les uns aux autres :

```
FileOutpuStream fos = new FileOutpuStream(toto)
EncryptedOutputStream eos = new EncryptedOutputStream(fo
GZIPOutputStream gos = new GZIPOutputStream(eos);
```

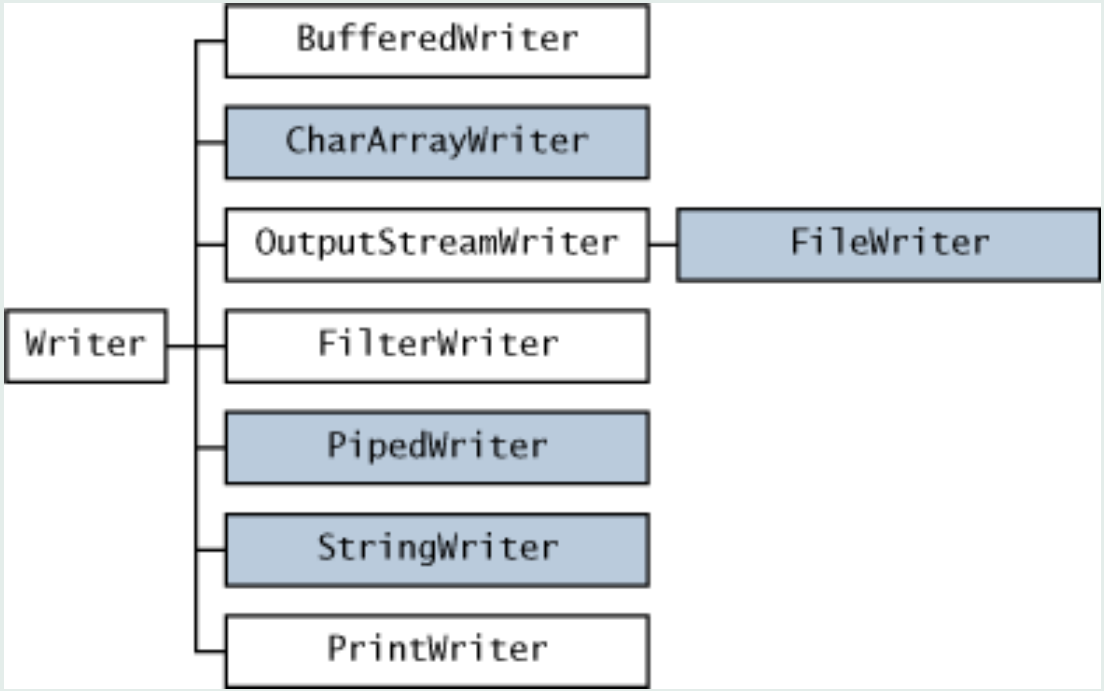


Les `InputStream` initiaux sont les sources (en grisé). Ceux qui s'y raccordent sont les filtres.

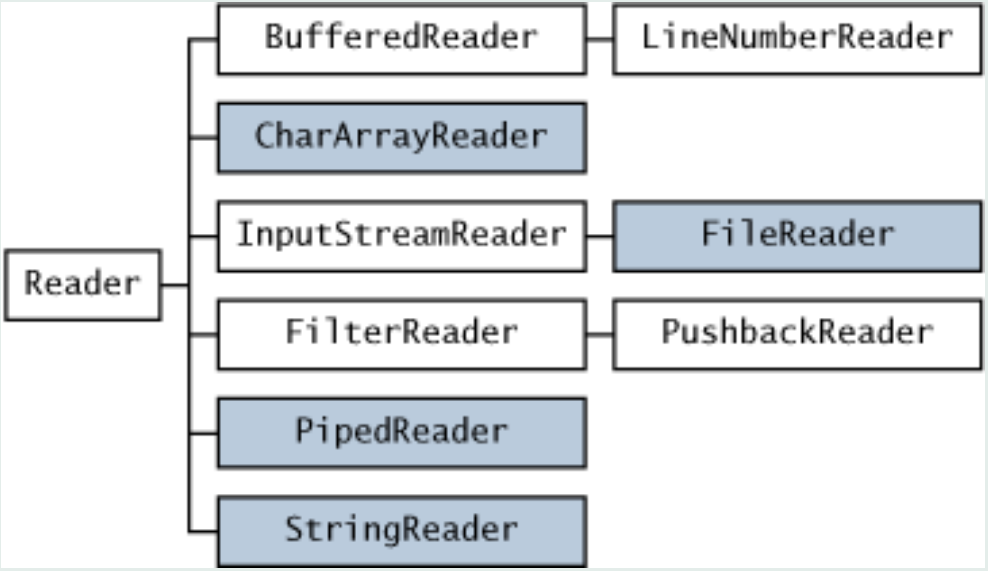


4.2. Les flux de caractères

Les `Writer` finaux sont les puits (en grisé). Ceux qui s'y raccordent sont les fitres.



Les `Reader` initiaux sont les sources (en grisé). Ceux qui s'y raccordent sont les filtres.



Les classe établissant les connexions réseaux ne sont pas des classes du paquet des I/O mais elles génèrent des `InputStream` et `OutputStream`.

Pour l'instant on peut imaginer qu'on a un flux sortant allant vers l'autre ordinateur et un entrant en venant.

4.3. FileInputStream et FileOutputStream

FileS.java

```
1  import java.io.*;
2
3  public class FileS
4  {
5    public static void main(String[] args)
6    {
7       try { // il peut y avoir des pbs à l'ouverture
8         FileOutputStream fos =
9                         new FileOutputStream("test");
10        byte[] data = {1,2,3,4,5,6,7,8,9};
11        fos.write(data);
12        fos.close();
13      }
14      catch (IOException e) {System.out.println(e);}
15   }
16 }
```

4.4. `DataInputStream` et `DataOutputStream`

Les `FileInputStream` et `FileOutputStrem`, travaillant sur des octets, ne sont pas pratiques à l'usage.

Il est plus pratique de travailler avec des filtres qu'on raccorde aux puits et aux sources.

Ainsi on peut raccorder un `DataOutputStream` à un `FileOutputStream` ce qui permet d'utiliser les méthodes de la classe `DataOutputStream` pour dialoguer avec le puit à savoir le fichier dans ce cas.

```
FileOutputStream fos = new FileOutputStream("titi");
DataOutputStream dos = new OutputStream(fos);

dos.writeInt(7);
for (int i = 0; i < 7; i ++)
  dos.writeDouble(data_x[i]);  // array of double
dos.flush();
```

4.5. Buffered Filter Stream

Les flux avec tampon permettent d'optimiser la vitesse de transfert et jouer avec le tampon pour revenir en arrière.

La taille du tampon peut être spécifiée lors de la création du `BufferedOutpu` Elle est de 512 octets par défaut. Pour faire partir un paquet avant que le tampon soit rempli, on peut utiliser la méthode `flush()`.

Pour un `BufferedInputStream bis`, on peut se servir du tampon pour revenir en arrière. Cela peut servir pour faire du parsing par exemple.

```
int read_byte = bis.read();
bis.mark(16); // mark the position and keep it for
              // until 16 bytes are read
read_byte = bis.read();
read_byte = bis.read();
bis.reset();              // return to the marker
read_byte = bis.read(); // same as 2nd read
```

4.6. `PrintStream` et `PrintWriter`

PrintStream est certainement le premier flux utilisé par tous les programmeurs de Java. En effet, `System.out` est un `PrintStream` dirigé vers l'écran.

Ces classes ont la particularité de ne pas faire remonter les exceptions d'E/S et de permettre d'ajouter des retours à la ligne dans le flux.

```
PrintStream ps = new PrintStream(
                     new FileOutputStream("test"));
ps.println("Le premier resultat est" + res1);
```

Pour lire ce fichier lisant les octets (l'encodage est choisi en fonction de votre variable d'environnement) :

```
String res;
BufferedReader in =
    new BufferedReader(new InputStreamReader("test"));
while(null != (res = in.readLine())) {
```

L'équivalent en utilisant des flux de caractères donne :

```
PrintWriter pw = new PrintWriter(new FileWriter("test"))
pw.print("Bonjour \n voici une premiere serie de mots");
```

4.7. Pipons

Les tubes, *pipes*, sont un sytème de communication unidirectionnel entre deux processus. On verra ultérieurement comment avoir 2 processus.

```
class Pipons extends Thread
{
  PipedOutputStream po;
  PipedInputStream pi;
  byte delay;

  private void connect(Pipons other) throws IOException
      po.connect(other.pi);
      pi.connect(other.po);
  }
...  // for each thread :
  po.write(val);
  b = (byte) pi.read();
```

On peut bien sûr brancher des filtres sur ces flux.

4.8. Les équivalences

On a plus ou moins l'équivalent avec les flux de caractères ce que résume le tableau suivant :

Caractère	Définition	Octet
Reader	Classe abstraite pour les flux entrant	InputStream
BufferedReader	Tampons d'entrée, peut lire une ligne	BufferedInputStream
LineNumberReader	Compte les numéros de ligne	(rien)
CharArrayReader	Lit depuis un tableau	ByteArrayInputStream
InputStreamReader	Transforme un flux d'octets en un flux de caractères	(rien)
FileReader	Lit un fichier	FileInputStream
FilterReader	Classe abstraite pour filter le flux	FilterInputStream
PushbackReader	Permet de remettre les données lues dans le flux ce qui permet de les relire	PushbackInputStream
PipedReader	Lit depuis un "pipe"	PipedInputStream
StringReader	Lit depuis un String	(rien)

La correspondance pour les flux sortants étant :

Caractère	Définition	Octet
Writer	Classe abstraite pour les flux sortant	OutputStream
BufferedWriter	Tampon de sortie, utilise le `return` de du système d'exploitation	BufferedOutputStream
CharArrayWriter	Ecrit dans un tableau	ByteArrayOutputStream
FilterWriter	Classe abstraite pour filter le flux	FilterOutputStream
OutputStreamWriter	Transform un flux de caractères en flux d'octets	(rien)
FileWriter	Ecrit en binaire sur un fichier	FileOutputStream
PrintWriter	Ecrit en ASCII sur un fichier	PrintStream
PipedWriter	Ecrit dans un "pipe"	PipedOutputStream
StringWriter	Ecrit dans un String	(rien)

Il existe d'autres filtres permettant de manipuler les flux :

- le `Scanner` qui remplace le `StreamTokenizer`
- le `java.securityDigestOutputStream` pour faire un condensat
- les flux de compression du paquet `java.util.zip`
- et d'autres...

Scanner.java

```
1  public static void main(String[] args)
2  {
3    Scanner scan = new Scanner(new File("myData"));
4    scan.useDelimiter("|");     // une regex marche
5
6    while scan.hasNext()
7      System.out.println(scan.next());
8  }
```

4.9. Le paquet New I/O, `java.nio`

Les New I/O sont la partie la plus importante de la version 1.4 de Java. Ils amméliorent grandement les performances et offrent de nouvelles possibilités dont :

- des Buffers pour les types primitifs (float, int...),
- des encodeurs et décodeur pour ensemble de caractères (UTF, ISO...),
- un pattern-matching à la perl,
- les canaux, *Channels*, concept représentant une connexion plus proche du système que les I/O classique (les I/O classiques ont une méthode `getChannel`) :
  - la possibilité de poser des verrous sur les fichiers et d'y faire des copies conforme de la mémoire,
  - des I/O multiplexées non bloquantes pour écrire des gros serveurs.
  - fermeture asynchrone du cannal, interruption possible,

Certaines nouveautés ont été inclues dans les anciennes classes (cf `String` par exemple).

Enfin Java 1.5 ajoute les `SSLSocket`s.

5. Communications à distance

5.1. La synchronisation des communication

On a soit

- des communication synchrones $\longrightarrow$ bloquantes
- des communication asynchrones $\longrightarrow$ non bloquantes

Les communications synchrones

- send envoie un paquet et attend un accusé de réception,
- receive bloque le processus jusqu'à ce qu'il reçoit des données.

Les communications asynchrones

- send envoie un paquet et le processus continue immédiatement,
- receive ne bloquant pas le processus, il faut utiliser un système d'interruption ou de scrutation (polling) pour traiter le message reçu.

Java utilise un send asynchrone et un receive synchrone.

Java permettant d'avoir plusieur processus tournant en parallèle. Bloquer un processus pour attendre un message n'est donc pas gênant[1].

Bloquer un processus après un envoi n'est que rarement utile et surtout il suffit de faire suivre le send par un receive synchrone pour avoir un send synchrone.

[1] sauf si on se retrouve à avoir trop de processus. Dans ce cas cf les flux multiplexés de `java.nio`

5.2. Destination du message

Une destination est une adresse + un port.

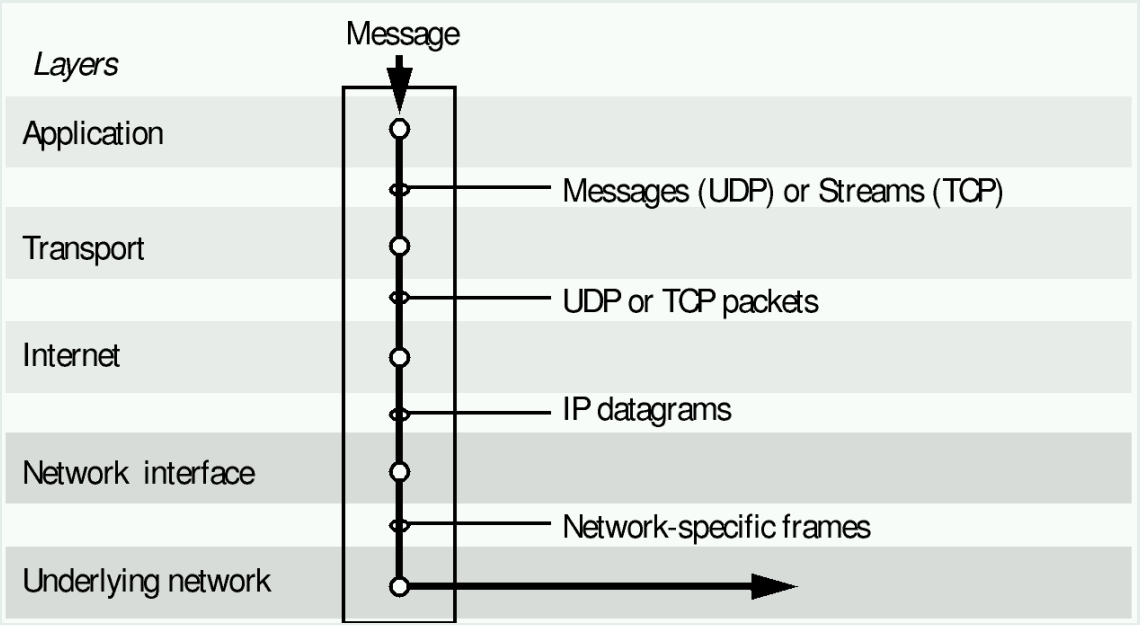
Quelques points à prendre en compte :

- Si l'adresse n'est pas un nom de machine mais une adresse IP, alors déplacer le serveur impliquera de modifier tous les clients.
- Avec certains OS comme Mach, il est possible de basculer des adresses en direct et donc de glisser[2] un serveur vers une autre adresse sans que les clients ne s'en apperçoivent.
- Il est possible d'envoyer un message à tout le monde[3]. Il s'agit de broadcast.
- Il est possible dans certains système comme IPv6, d'envoyer un message à un groupe de personnes. Il s'agit de multicast.

On peut faire du multicast en IPv4 en utilisant pour les personnes du groupe des adresses de classe D réservées. Il s'agit des adresses allant de 224.0.0.0 à 239.255.255.255.

[2] déplacer = arrêter puis relancer, glisser n'implique pas d'arrêt
[3] tout le monde sur le même brin

5.3. Niveau de communication



On utilisera les classes `DatagramSocket` et les `DatagramPacket` pour UDP et les `Socket` pour TCP.

Il est possible d'avoir des classes gérant les commnunications au niveau applicatif comme la classe `URLConnection`.

UDPClient.java

```
1  // from "Distributed Systems, concept & design"
2  import java.net.*;
3  import java.io.*;
4
5  public class UDPClient
6  {
7    public static void main(String args[])
8    {
9      // args give message contents and destination ho
10     DatagramSocket aSocket = null;
11
12     try
13     {
14       aSocket = new DatagramSocket();
15       byte [] m = args[0].getBytes();
16       InetAddress aHost =
17         InetAddress.getByName(args[1]);
18       int serverPort = 6789;
```

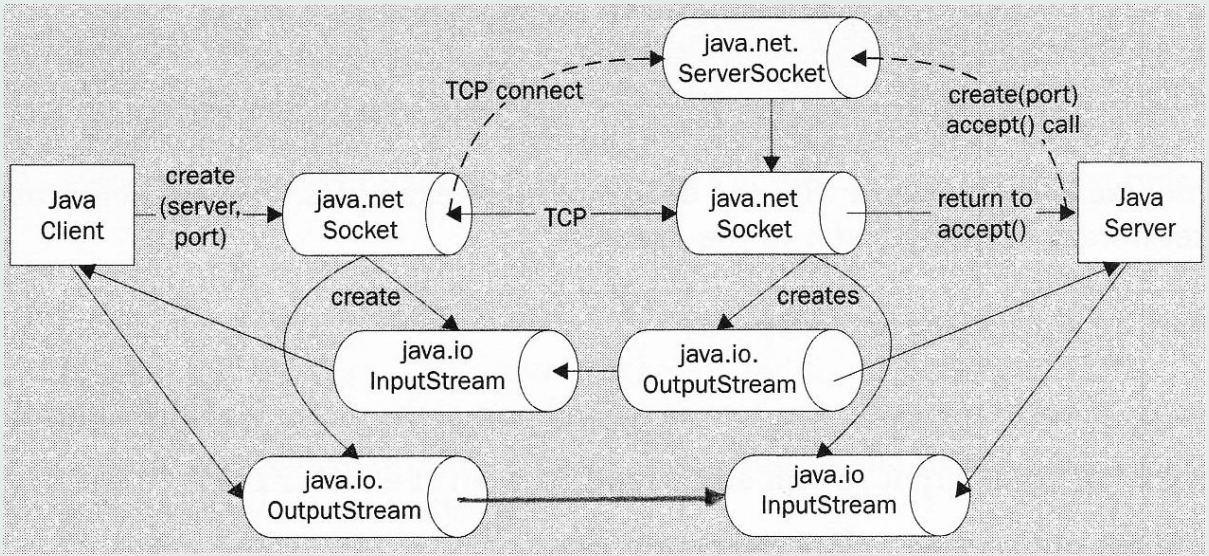

```
20          DatagramPacket request =
21             new DatagramPacket(m,  args[0].length(),
22                                aHost, serverPort);
23          aSocket.send(request);
24          byte[] buffer = new byte[1000];
25          DatagramPacket reply =
26            new DatagramPacket(buffer, buffer.length);
27          aSocket.receive(reply);
28          System.out.println("Reply: " +
29                          new String(reply.getData()));
30       }
31       catch (SocketException e)
32       { System.out.println("Socket: "+e.getMessage());
33       catch (IOException e)
34       { System.out.println("IO: " + e.getMessage());}
35       finally {if(aSocket != null) aSocket.close();}
36    }
37 }
```

UDPServer.java

```
1  // from "Distributed Systems, concept & design"
2  // by G.Coulouris, J.Dollimore and T.Kindgerg
3  // cf http://www.cdk3.net/
4  import java.net.*;
5  import java.io.*;
6
7  public class UDPServer
8  {
9    public static void main(String args[])
10   {
11     DatagramSocket aSocket = null;
12
13     try
14     {
15       aSocket = new DatagramSocket(6789);
16       // create socket at agreed port
17       byte[] buffer = new byte[1000];
```


```
19          while(true)
20          {
21             DatagramPacket request =
22              new DatagramPacket(buffer, buffer.length);
23            aSocket.receive(request);
24            DatagramPacket reply =
25              new DatagramPacket(request.getData(),
26                                 request.getLength(),
27                                 request.getAddress(),
28                                 request.getPort());
29            aSocket.send(reply);
30          }
31       }
32       catch (SocketException e)
33       { System.out.println("Socket: " + e.getMessage()
34       catch (IOException e)
35       { System.out.println("IO: " + e.getMessage()); }
36       finally {if(aSocket != null) aSocket.close();}
37    }
38 }
```

En utilisant les flux TCP on peut y brancher les flux d'E/S de Java ce qui permet d'écrire et de lire via le réseau comme sur un fichier.



dessin extrait de "Beginning Java Networking" chez Wrox

TCPClient.java

```
1  import java.net.*;
2  import java.io.*;
3
4  public class TCPClient
5  {
6    public static void main (String args[])
7    {
8      // arguments supply message and hostname
9      Socket s = null;
10     try{
11       int serverPort = 7896;
12       s = new Socket(args[1], serverPort);
13       BufferedReader in =
14         new BufferedReader(new InputStreamReader(s.g
15       PrintWriter out =   // true for autoFlush
16         new PrintWriter( s.getOutputStream(),true);
```


```
18          out.println(args[0]);
19          System.out.println(in.readLine());
20        }
21        catch (UnknownHostException e)
22        {System.out.println("Socket:"+e.getMessage());}
23        catch (EOFException e)
24        {System.out.println("EOF:"+e.getMessage());}
25        catch (IOException e)
26        {System.out.println("readline:"+e.getMessage());
27        finally
28        {
29          if(s!=null)
30            try {s.close();}
31          catch (IOException e)
32          {System.out.println("close:"+e.getMessage());}
33        }
34      }
35    }
```

6. Les processus légers (threads)

Puisque les communications entrantes sont bloquantes, tout serveur se doit d'avoir plusieurs processus qui tournent en parallèle pour écouter l'ensemble des clients.

Pour pouvoir *forker* en Java on dérive sa classe de `Thread`[4].

Il faut ensuite implémenter la méthode `run()` qui sera lancée dans un nouveau processus par la méthode `start()`.

Comme il s'agit d'une méthode liée à un objet, on garde l'accès à toutes les variables de classe et de l'objet.

(on verra plus en détail les threads au prochain cours)

[4] on verra une autre méthode plus tard

DamnFork.java

```
1  class DamnFork extends Thread
2  {
3    private int id;
4
5    public DamnFork(int i) { // will never finish
6        id = i;
7        this.start();
8        DamnFork next = new DamnFork(++i);
9    }
10
11   public void run() {
12     while(true) System.out.println(id);
13   }
14
15   static public void main(String args[]) {
16       DamnFork d = new DamnFork(0);
17   }
18 }
```

TCPServer.java

```
1  import java.net.*;
2  import java.io.*;
3  public class TCPServer
4  {
5    public static void main (String args[]) {
6      try {
7        int serverPort = 80; // the server port
8        ServerSocket listenSocket =
9          new ServerSocket(serverPort);
10       while(true) {
11         Socket clientSocket = listenSocket.accept();
12         Connection c = new Connection(clientSocket);
13       }
14     }
15     catch(IOException e)
16     {System.out.println("Listen socket:"+e.getMessag
17   }
18 }
```


```
20  class Connection extends Thread
21  {
22    BufferedReader in;
23    PrintWriter out;
24    Socket sock;
25
26    public Connection (Socket a_client_socket)
27    {
28      try {
29        sock= a_client_socket;
30        in = new BufferedReader(new InputStreamReader(
31                                  sock.getInputStream()
32        out =new PrintWriter(sock.getOutputStream(),tr
33        this.start();
34      }
35      catch(IOException e)
36      {System.out.println("Connection:"+e.getMessage()
37    }
```


```
39    public void run()
40    {
41      try {        // an echo server
42          out.println(in.readLine());
43      }
44      catch (EOFException e)
45      {System.out.println("EOF:"+e.getMessage());}
46      catch(IOException e)
47      {System.out.println("readline:"+e.getMessage());
48      finally{
49        try {sock.close(); out.close(); in.close();}
50        catch (IOException e){/*close failed*/}
51      }
52    }
53  }
```

6.0.0.1. Testons avec telnet.

6.0.0.2. Définissons le port dans `/etc/services`.

7. La sérialisation

http ://java.sun.com/j2se/1.5.0/docs/guide/serialization/

La sérialisation est l'art d'empaqueter les données décrivant un objet pour pouvoir le recréer ensuite.

Cela peut servir à sauvegarder l'état d'un programme (un jeu) dans un fichier ou à envoyer un objet à distance via le réseau.

Pour *sérialiser* un objet, il suffit que sa classe implémente l'interface `Serializable`. Cette interface étant vide, il suffit de déclarer qu'on l'implémente pour l'implémenter.

```
public class MonObjet implements Serializable { }
```

La sérialisation se fait ensuite via un flux sortant et la désérialisation via un flux entrant (des byte streams) :

```
MonObjet mo;
...
FileOutputStream fos = new FileOutputStream("toto");
(new ObjectOutputStream(fos)).writeObject(mo);
```

La sérialisation est récursive à savoir qu'elle sérialiste aussi les objets variables d'un objet. Bien sûr, les objets sauvés doivent tous être sérialisables.

Pour éviter d'empaqueter 2 fois un objet référencé par 2 objets, la sérialisation maintient une liste des objets empaquetés.

7.1. `transient`

Il est courant qu'il ne soit pas pertinent de sauver certaines variables d'un objet. Pour indiquer qu'une variable ne devra pas être prise en compte lors de la sérialisation, on utilise le mot `transient`.

```
public class Square implements Serializable {
  Point origine;
  int largeur;
  transient int surface;
```

7.2. Customisation

La sérialisation ne gère ni les variables `transient`, ni les variables `static`, ni les objets non sérialisables.

Pour donner des valeurs à ces variables lors du dépaquetage on peut écrire la méthode `readObject`. `writeObject` sert à personnaliser l'écriture de la sérialisation.

MySerializable.java

```
1  import java.io.*; import java.util.*; import java.ne
2
3  public class MySerializable implements Serializable
4  {
5    private static int globalID;
6    private transient int ID;
7    String sToday = "Today: ";
8    private transient Date dtToday = new Date();
9    private transient Socket sock;
```


```
11    public MySerializableClass() {
12      ID = getID();
13      try { sock = new Socket("SomeHost", 1234); }
14      catch (IOException e) {}
15    }
16
17    private synchronized static int getID() {
18      return globalID++;
19    }
20
21    private void writeObject(ObjectOutputStream os)
22            throws IOException
23    {
24      // automatic serialization of serializable data
25      os.defaultWriteObject();
26
27      // custom serialization of the socket object.
28      os.writeObject(sock.getInetAddress());
29      os.writeInt(sock.getPort());
30    }
```


```
32    private void readObject(ObjectInputStream is)
33          throws IOException, ClassNotFoundException
34    {
35      // automatic serialization of serializable data
36      is.defaultReadObject();
37
38      // custom serialization of the socket object.
39      InetAddress sock_address =
40                        (InetAddress) is.readObject();
41      int sock_port = is.readInt();
42      sock = new Socket(sock_address, sock_port);
43
44      // custom serialization of the data members
45      ID = getID();
46      dtToday = new Date();
47    }
48  }
```

Remarque : la méthode `defaultWriteObject` appartient à l'`ObjectOutputStream oos` et ne prend pas d'argument, même pas `this` pour savoir ce qu'il faut sérialiser.

En fait cette méthode sera informée de l'objet à sérialiser lorsqu'on sérialisera l'objet à l'aide de `oss.writeObject(toto)`, `toto` étant notre objet.

`oos.writeObject(toto)` note dans la variable `curObj` d'`oss` l'objet à sérialiser puis examinera sa classe. Si elle découvre que la classe de `toto` a une méthode `writeObject` alors elle invoquera cette méthode (cf la classe `Method`) en lui indiquant que `toto` est `this` et que l'argument est le flux courant, à savoir `oss`.

Arrivé à la méthode `defaultWriteObject`, cette méthode regarde si la variable `curObj` existe et si oui, elle la sérialise.

7.3. Le clonage fort

Java utilise les références partout, aussi écrire `a = b;` ne crée pas un nouvel objet `a` mais simplement une référence sur `b`.

Pour recopier `b` il faut utiliser la méthode `clone` soit `a = b.clone();`

Mais le clonage ne recopie pas les variables objets de `b` mais leur références. Aussi après clonage, `a.x` et `b.x` est le même objet.

Grace à la sérialisation, il est possible d'empaqueter l'objet `b` et tout ce qu'il contient dans un tableau de byte, puis de créer `a` à partir de ce tableau. On a ainsi une copie complète de `b`.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream()
(new ObjectOuptputStream(baos)).writeObject(b);
byte[] array = baos.toByteArray();

ByteArrayInputStream bais=new ByteArrayInputStream(array
ObjectInputStream ois = new ObjectInputStream(bais);
a = (MonObjet) ois.readObject();
```