Sockets
Illustration avec JAVA

Applications réparties

Plan

- Introduction, définition d'une "socket"
- Mode connecté (protocole TCP)
- Mode non-connecté (protocole UDP)
- Les API Réseau de JAVA
- Exemples

Bibliographie

- Gilles Roussel, Étienne Duris, …, "**Java et Internet : concepts et programmation**", Vuibert.
  - Tome 1 : ***côté client***, 2002, ISBN 2-7117-8689-7.
  - Tome 2 : ***côté serveur***, 2007, ISBN 2-7117-8690-0.
- Elliotte Rusty Harold, "Programmation Réseau avec Java", O'Reilly, 1997, ISBN 2-84177-034-6.
- Scott Oaks, Henry Wong, "Java Threads", 2nd Édition, O'Reilly, 1999, ISBN 1-56592-418-5, (existe en français).

Introduction

- Problème
  - Réaliser un service distribué/réparti en utilisant les protocoles de transport (TCP, UDP)
- Solutions
  - **Sockets** : mécanisme universel de bas niveau, utilisable depuis tout langage (exemple : C, Java)
  - Middleware (intergiciel)
    - *Appel de procédure à distance* (RPC), dans un langage particulier (exemple : C)
    - *Appel de méthode à distance*, dans les langages à objets (exemple : Java RMI, CORBA)
    - EJB, .Net, Web Services

Sockets

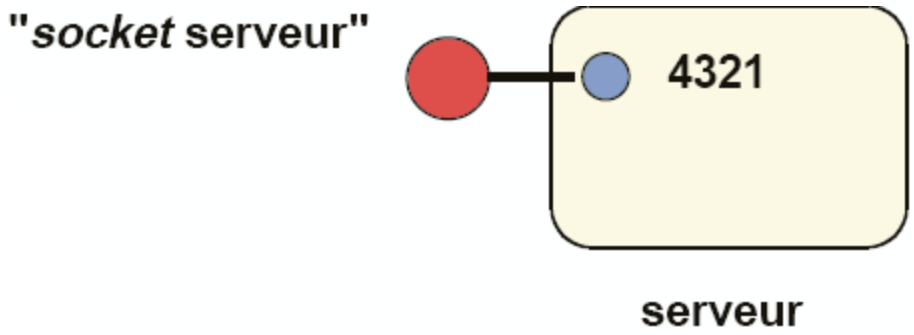
- Mécanisme de communication permettant d'utiliser l'interface de transport (TCP-UDP)
- Introduit dans Unix dans les années 80
  - Bill Joy, Projet "Berkeley Software Distribution" (BSD)
- Standard aujourd'hui.

Sockets

- Principe de fonctionnement : 3 phases

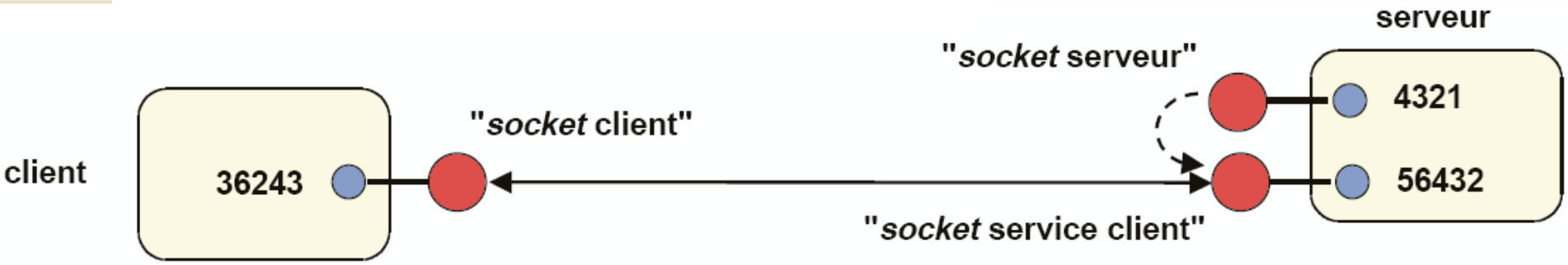
(illustration avec TCP, et plusieurs clients simultanés)

  - 1) Le serveur crée une "**socket serveur**" (associée à un port) et se met en attente



Sockets

- 2) Le client se connecte à la socket serveur
  - deux sockets sont alors créées : une "**socket client**", côté client, et une "**socket service client**" côté serveur. Ces sockets sont connectées entre elles.



- 3) Le client et le serveur communiquent par les sockets
  - L'interface est celle des fichiers (read, write). La socket serveur peut accepter de nouvelles connexions.

Deux réalisations possibles

- Mode connecté (protocole TCP)
  - Ouverture d'une liaison, suite d'échanges, fermeture de la liaison.
  - Le serveur préserve son état entre deux requêtes.
  - Garanties de TCP : ordre, contrôle de flux, fiabilité.
  - Adapté aux échanges ayant une certaine durée (plusieurs messages).

Deux réalisations possibles

- Mode non connecté (protocole UDP)
  - Les requêtes successives sont indépendantes.
  - Pas de préservation de l'état entre les requêtes.
  - Le client doit indiquer son adresse à chaque requête (pas de liaison permanente).
  - Pas de garanties particulières.
  - Adapté aux échanges brefs (réponse en 1 message).

Deux réalisations possibles

- Points communs
  - Le client a l'initiative de la communication ;

    Le serveur doit être à l'écoute.
  - Le client doit connaître la référence du serveur [adresse IP, n° de port]

    (annuaire si serveur enregistré, ou convention)
  - Le serveur peut servir plusieurs clients (1 thread par client).

Utilisation du mode connecté

- Caractéristiques
  - Établissement préalable d'une connexion (*circuit virtuel*) : le client demande au serveur s'il accepte la connexion.
  - Fiabilité assurée par le protocole (TCP).
  - Mode d'échange par flot d'octets : le *récepteur* n'a pas connaissance du découpage des données effectué par l'*émetteur*.
  - Possibilité d'émettre et de recevoir des caractères urgents (OOB : Out OF Band).
  - Après initialisation, le serveur est "passif" : il est activé lors de l'arrivée d'une demande de connexion du client.

Utilisation du mode connecté

- Caractéristiques (suite)
  - Un serveur peut répondre aux demandes de service de plusieurs clients : les requêtes arrivées et non traitées sont stockées dans une file d'attente.
- Contraintes
  - Le client doit avoir accès à l'adresse du serveur (adresse IP, n° de port).
- Modes de gestion des requêtes
  - **Itératif** : le processus traite les requêtes les unes après les autres.
  - **Concurrent** : création d'un processus fils (ou thread) pour chaque client. (serveur multi-threadé)

Serveur itératif en mode connecté

- Client ouvre connexion avec serveur avant adresser des appels, puis ferme connexion à la fin
  - Délimitation temporelle des échanges
  - Maintien de l'état de connexion pour la gestion des paramètres de qualité de service
  - Traitement des pannes, propriété d'ordre.
- Orienté vers :
  - Le traitement ordonné d'une suite d'appels
    - Ordre local (requêtes d'un client traitées dans leur ordre d'émission), global ou causal.
  - La gestion de données persistantes ou de protocoles avec état.

Gestion du parallélisme sur le serveur

- Avec schéma précédent : 1 seul processus serveur
  - Si clients multiples → traités en séquence
    - La longueur max. de la file d'attente peut être spécifiée lors de la création de la socket serveur.
- On peut aussi utiliser un schéma "multi-threads" :
  - Thread veilleur (*serveur*) doit créer explicitement un nouveau thread exécutant (*servant*) à chaque nouvelle connexion d'un client
  - Une socket différente "service client" est créée pour chaque client
  - Le thread veilleur se remet ensuite en attente.

Utilisation du mode non connecté

- Caractéristiques
  - Pas d'établissement préalable d'une connexion.
  - Pas de garantie de fiabilité.
  - Adapté aux applications pour lesquelles les réponses aux requêtes des clients sont courtes (1 message).
  - Le récepteur reçoit les données selon le découpage effectué par l'émetteur.

Utilisation du mode non connecté

- Contraintes
  - Le client doit avoir accès à l'adresse du serveur (adresse IP, port).
  - Le serveur doit récupérer l'adresse de chaque client pour lui répondre (primitives *sendto*/*send*, *recvfrom*/*receive*).
- Mode de gestion des requêtes
  - **Itératif** (requêtes traitées l'une après l'autre).
  - **Concurrent** (1 processus ou thread par client).

Serveur itératif en mode non connecté

- Le client peut envoyer une requête au serveur à tout moment.
  - Mode léger permettant
    - Le traitement non ordonné des appels.
    - L'absence de mémorisation entre requêtes successives (serveur sans données rémanentes et sans état).
  - Exemples :
    - Calcul de fonctions numériques,
    - DNS (service de noms),
    - NFS (service de fichiers répartis).

Les API Réseau de Java

`java.net`

`javax.net`

Principales classes

- Adresses IP
  - InetAddress
- Socket TCP
  - Socket, ServerSocket, JSSE (Java Secure Socket Layer)
- Sockets UDP
  - DatagramSocket, DatagramPacket
- Sockets MultiCast (cf. Annexe)
  - MulticastSocket, DatagramPacket
- Classes réseaux au niveau application (cf. Annexe)
  - URL, URI, URLConnection, HttpURLConnection, JarURLConnection

Adresses IP
`java.net.InetAddress`

- Représente une adresse IP
  - Utilisé par les classes Socket et DatagramSocket
- Qq. méthodes statiques pour la résolution DNS
  - `public static InetAddress` ***`getByName`***`(String hostname)` `throws UnknownHostException`
  - `public static InetAddress getLocalHost()` `throws UnknownHostException`
    - Donne l'adresse de l'hôte local

Exemple avec java.net.InetAddress

```
public static void main(String args[]) {
    InetAddress server;
    try {
          if(args.length > 0)
            server = InetAddress.getByName(args[0]);
          else server = InetAddress.getLocalHost();

          System.out.println(server);
    } catch (UnknownHostException e) {
          System.out.println("Introuvable !");
    }
}
// exemple de résultat : TOTO/192.168.1.2
```

Sockets en mode connecté

- Représente une connexion fiable TCP/IP entre 2 processus (~machines)
  - La connexion est fiable (contrôle d'erreur, …)
  - 2 flots de données sont établis entre les deux machines
- Connexion est asymétrique
  - Une machine est serveur : classe **`ServerSocket`**
    - Elle attend les demandes de connexion et les acceptent.
  - Une machine est client : classe **`Socket`**
    - Elle demande l'établissement de la connexion avec le serveur.

Sockets en mode connecté Coté serveur

- Serveur :
  - Crée la ServerSocket :

```
ServerSocket listenSocket = new ServerSocket(port);
```

  - Attend les demandes de connexion et crée une socket pour le dialogue :

```
Socket clientSocket = listenSocket.accept();
```

  - Récupère les flux d'entrée et de sortie :

```
InputStream in = clientSocket.getInputStream();
OutputStream out = clientSocket.getOutputStream();
```

  - Dialogue avec le client.

Sockets en mode connecté Coté client

- Client :
  - Crée la Socket :

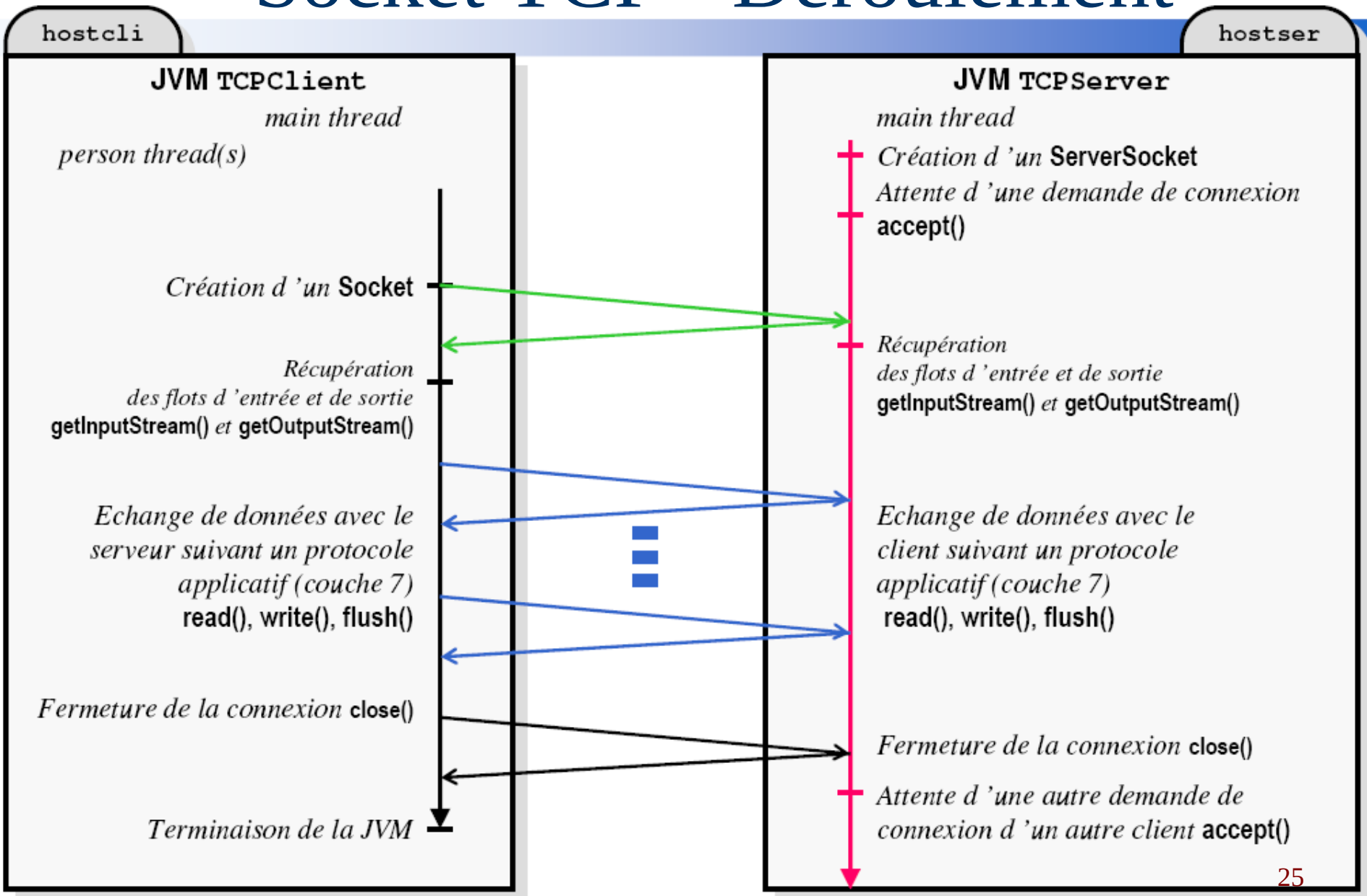
```
Socket server = new Socket(host,port);
```

  - Récupère les flux d'entrée et de sortie :

```
InputStream in = server.getInputStream();
OutputStream out = server.getOutputStream();
```

  - Dialogue avec le serveur.

Socket TCP - Déroulement



Socket TCP Remarques

- Le serveur :
  - Après la fermeture de la connexion, le serveur se met en attente du nouvelle connexion du même client ou d'un autre.
  - Si plusieurs demandes de connexion arrivent simultanément, une seule est acceptée et les autres sont mises en attente jusqu'au `accept()` suivant (modulo les timeouts).
  - Pour accepter et traiter plusieurs connexions simultanées, la solution est de "**multi-threader**" le serveur.

Socket TCP Remarques

- Le **protocole** :
  - Le client et le serveur doivent respecter un *protocole valide d'échange des données*
    - Le client envoie un entier mais le serveur attend un flottant.
    - Le client attend des données du serveur qui lui même attend des données du client → **interblocage** !
  - Attention à l'hétérogénéité des plateformes client et serveur
    - Un client sur Intel envoie un entier *little-endian*,
    - Le serveur sur Sparc reçoit cet entier et le traite en *big-endian*.

Socket TCP
Les échanges

- 2 types d'échanges possibles :
  - Échange en mode **Ligne** (chaînes de caractères)
  - Échange en mode **Bloc d'octets** (de bytes)

Socket TCP
Échange en mode Ligne

- Requêtes et réponses : une ou plusieurs lignes de texte (ASCII, UTF-8, UniCode, ...)
- Exemple de protocole en mode ligne : HTTP, FTP, SMTP, …
- Avantage pour le mise au point :
  - Le client peut être un client telnet

Socket TCP
Échange en mode Ligne

- En JAVA
  - Utilisation des Classes **BufferedReader / BufferedWriter**
- Attention :
  - Les terminaisons de ligne varient selon les OS
    - LF '\n', CR '\r', CRLF "\r\n", …
  - L'encodage n'est pas supporté par tous les langages
  - Certains protocoles (HTTP, SMTP, …) limitent la longueur des lignes
    - Sécurité : Risque d'attaque par "Buffer Overflow".

Socket TCP
Échange en mode bloc d'octets

- Requêtes et réponses : blocs d'octets *d'une taille et d'un format connus de l'autre*
  - Exemples : RPC, RMI, CORBA,...
  - Le bloc peut avoir une entête qui contient un *code* d'opération et la *longueur* du corps du bloc qui contient les données.

Socket TCP
Échange en mode Bloc d'octets

- En JAVA
  - Utilisation possible des Classes `DataInputStream`/ `DataOutputStream`
  - Mais impose que le client et le serveur soient des JVMs.
- Attention
  - Les données sont représentées différemment selon les architectures, les OS et les langages
    - *Big/Little-Endian*, …
  - Il faut alors un format pivot de représentation (XDR, IIOP …)

Exemple
Code d'un client (1/2)

```
public class EchoClient {
   public static void main(String[] args) {
    Socket theSocket = null;
    BufferedReader theInputStream = null;
    BufferedReader userInput = null;
    PrintStream theOutputStream = null;
    String theLine = null;

    try {
     theSocket = new Socket(args[0], Integer.parseInt(args[1]));
     theOutputStream = new PrintStream(theSocket.getOutputStream());
     theInputStream = new BufferedReader(
                new InputStreamReader(theSocket.getInputStream()));
     userInput = new BufferedReader(
                        new InputStreamReader(System.in));
```

Exemple
Code d'un client (1/2)

```
        while(true)
        {       theLine = userInput.readLine();
                if(theLine.equals(".")) break;
                theOutputStream.println(theLine);
                System.out.println(theInputStream.readLine());   }
      } // try
      catch (UnknownHostException e) { ... }
      catch (IOException e) { ... }
      finally
      { try {
              if(theSocket!=null) theSocket.close();
              if(theInputStream!=null) theInputStream.close();
              ...
        } catch (IOException e) { ... }  }
  } // main
} // EchoClient
```

Exemple
Code d'un serveur (1/2)

```
public class EchoServer {
   public static void main(String[] args) {
    ServerSocket listenSocket = null;
    try {
     listenSocket = new
   ServerSocket(Integer.parseInt(args[0]));
      while(true)
      { Socket clientSocket = listenSocket.accept();
        //System.out.println(clientSocket.getInetAddress());
        doService(clientSocket);
        clientSocket.close();
      }
    }
    catch (Exception e) { ... }
```

Exemple
Code d'un serveur (2/2)

```
finally{ try{if(listenSocket!=null) listenSocket.close();}
         catch (IOException e) { ... }   }
} // main

static void doService(Socket clientSocket) throws IOException
{BufferedReader in = new BufferedReader(
       new InputStreamReader(clientSocket.getInputStream()));
 PrintStream out =
       new PrintStream(clientSocket.getOutputStream());
 while(true)
 { String theLine=in.readLine();
   out.println(theLine); }
} // doService

} // class EchoServer
```

Exemple
Code d'un client SMTP

Exemple

- Remarque :
  - L'API ***JavaMail*** est préférable pour l'envoi de courriers électroniques !

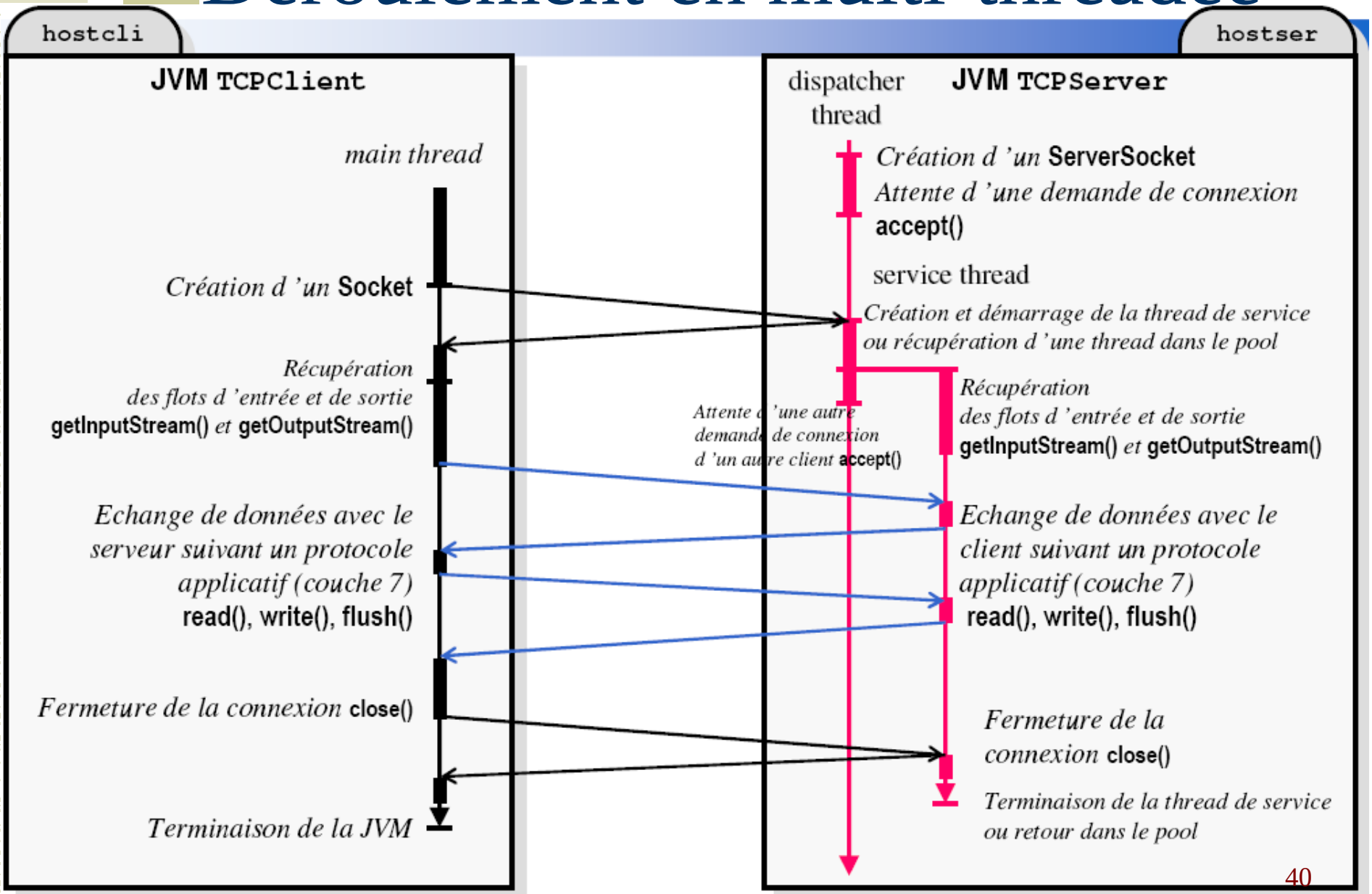
Exceptions

- java.net.*BindException*
  - Erreur de liaison à une adresse locale (le port est peut être déjà utilisé)
- java.net.*ConnectException*
  - Refus de connexion par l'hôte (aucun "process" écoute le port, …)
- java.net.*NoRouteToHostException*
  - L'hôte n'est pas joignable
- java.net.*ProtocolException*
  - Erreur du protocole (TCP, …)
- java.net.*SocketException*
  - Erreur du protocole (TCP, …)
- java.net.*UnknownHostException*
  - Erreur de DNS

Serveur multi-threadé

- Motivation : accepter (et servir) plusieurs clients.
- Méthode :
  - Un thread (veilleur/dispatcher) attend un client,
  - Récupère une socket dédiée à cette connexion avec le client,
  - Crée un thread (de service / servant) qui prend en charge le dialogue avec le client,
  - Retourne en attente d'une nouvelle demande de connexion.
- Remarque :
  - Les threads de service peuvent être tirés d'un pool
    - Avantage : limiter le coût d'initialisation des threads de service.

Déroulement en multi-threadée



Exemple
Code d'un serveur multi-threadé

```
public class EchoServer_multithread extends Thread {
 private Socket clientSocket;

 EchoServer_multithread(Socket clientSocket)
 { this.clientSocket = clientSocket; }

 public static void main(String[] args) {
   ServerSocket listenSocket = null;
   try {
        listenSocket = new ServerSocket(Integer.parseInt(args[0]));
        while(true) //le dispatcher est le thread qui exécute main()
        { Socket clientSocket = listenSocket.accept();
          //System.out.println(clientSocket.getInetAddress());
          EchoServer_multithread serviceThread =
                new EchoServer_multithread(clientSocket);
          serviceThread.start(); }
   } catch(Exception e){...} finally{... listenSocket.close();} ...}
}
```

Exemple
Code d'un serveur multi-threadé

```
public void run() {
   try { doService(clientSocket);  clientSocket.close(); }
   catch (IOException e) { ... } }

public void doService(Socket clientSocket) throws IOException
{ BufferedReader in;
  PrintStream out;
   in = new BufferedReader(
           new InputStreamReader(clientSocket.getInputStream()));
   out = new PrintStream(clientSocket.getOutputStream());
   while(true) {
     String theLine=in.readLine();
     if(theLine.equals(".")) break;//thread de service doit terminer
     out.println(theLine); }
} // doService

} // classe EchoServer_multithread
```

Personnaliser un type de Socket

- Motivation
  - Il est souvent nécessaire de traiter les données à envoyer ou reçues d'une socket (compression, conversion, filtrage, chiffrement, …)
    - Post-traitement des données après réception (exemple : Décompression, Déchiffrement)
    - Pré-traitement des données avant envoi (exemple : Compression, Chiffrement)

Personnaliser un type de Socket Etendre la classe de Socket

```
class CompressionSocket extends Socket {
 private int compressionrate;
 private InputStream in; private OutputStream out;
 public CompressionSocket(int compressionrate) {...}
 ...
 public InputStream getInputStream() throws IOException {...}
 public OutputStream getOutputStream() throws IOException {...}
 public synchronized void close() throws IOException {...}
}

class CompressionServerSocket extends ServerSocket {
  public CompressionServerSocket(int port, int compressionrate)
   throws IOException {...}
  public Socket accept() throws IOException {...}
  ...
}
```

Sockets en mode non connecté

- Rappel sur UDP
  - Couche de transport non fiable en mode non connecté.
  - Contrôle des erreurs mais pas de reprise sur erreur, pas d'ACK, séquencement des paquets non garanti.
  - Plus simple que TCP et plus efficace pour certaines applications (envoi de vidéo, de mesures, NFS, DNS, …).

Sockets en mode non connecté

- Principe
  - L'émetteur (*sender*) envoie un datagramme (paquets de données sur une socket que le récepteur (*receiver*) écoute.
- La classe `java.net.DatagramSocket`
  - Représente la socket locale ou distante en mode non connecté.
- La classe `java.net.DatagramPacket`
  - Représente un datagramme à envoyer ou reçu.

Sockets en mode non connecté Coté émetteur

- Émetteur :
  - Crée le DatagramSocket

```
DatagramSocket ms =
  new DatagramSocket(receiverInetAddress);
```

  - Construit un DatagramPacket

```
int len = 5;
byte[] data = new byte[len];
// data doit être "rempli" :
data = {'H', 'e', 'l', 'l', 'o'};
DatagramPacket outputPacket = new DatagramPacket(data,
  data.length, receiverInetAddress, port);
```

  - Envoie le DatagramPacket au recepteur

```
ms.send(outputPacket);
```

Sockets en mode non connecté Coté récepteur

- Récepteur
  - Crée le DatagramSocket lié à un port d 'écoute

```
DatagramSocket theSocket =
  new DatagramSocket(port);
```

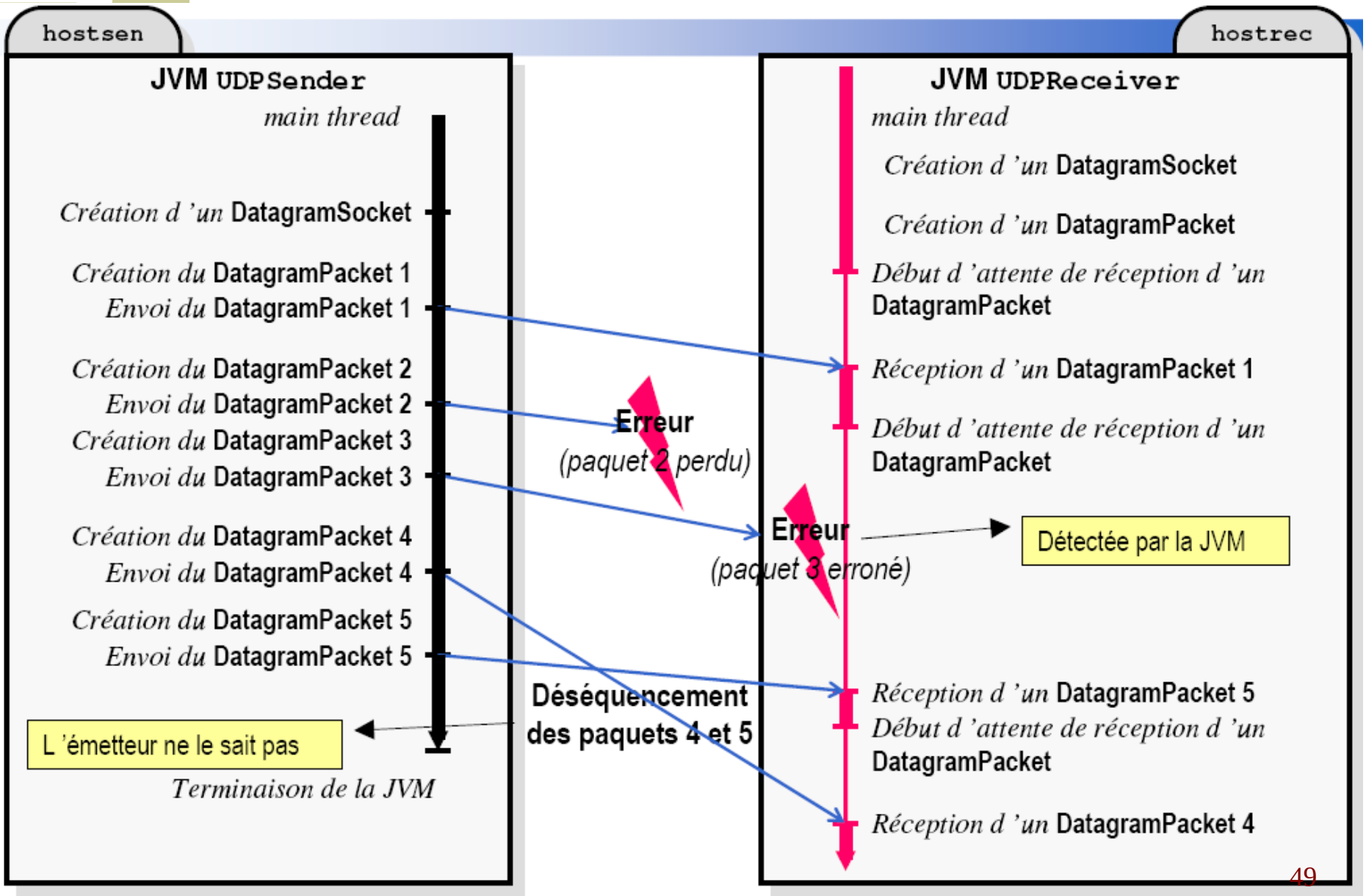
  - Construit un DatagramPacket pour la réception

```
byte[] incomingData = new byte[MAXLEN];
// ce buffer est vide
DatagramPacket incomingPacket =
   new DatagramPacket (incomingData ,
                       incomingData.length );
```

  - Réception le DatagramPacket du paquet puis utilisation

```
theSocket.receive(incomingPacket);
```

Socket UDP - Déroulement



Exemple
Code de l'émetteur

```
public class UDPSender {
 public static void main(String[] args) {
   try { InetAddress receiver = InetAddress.getByName(args[0]);
        int port = Integer.parseInt(args[1]);
        DatagramSocket theSocket = new DatagramSocket();
        BufferedReader userInput = new BufferedReader(
                                   new InputStreamReader(System.in));
        while (true) {
         String theLine = userInput.readLine();
         if (theLine.equals(".")) break;
         byte[] data = theLine.getBytes();
         DatagramPacket theOutput =
           new DatagramPacket(data, data.length, receiver, port);
         theSocket.send(theOutput); }
        } catch (Exception e) { ...}
   }
} // class socketsUDP_UDPSender
```

Exemple
Code du récepteur

```
public class socketsUDP_UDPReceiver {
 public static void main(String[] args) {
  try {//construction d'un DatagramSocket
   int port = Integer.parseInt(args[0]);
   DatagramSocket ds = new DatagramSocket(port);
   //construction d'un DatagramPacket
   int len = Integer.parseInt(args[1]);
   byte[] buffer = new byte[len];
   DatagramPacket incomingPacket =
        new DatagramPacket(buffer, buffer.length);
   while(true) {
        ds.receive(incomingPacket);
        String s = new String(incomingPacket.getData());
        System.out.println(incomingPacket.getAddress()
          + " at port " + incomingPacket.getPort() + " says " + s); }
   } catch (Exception e) { ... }
} // socketsUDP_UDPReceiver
```

Remarques

- Fragmentation IP
  - Le DatagramPacket peut être fragmenté pour les couches inférieur (IP, Ethernet, …) par la couche UDP
    - La taille maximun d'un paquet IP est de 65535 octets.
  - Il est ré-assemblé par la couche UDP du récepteur avec d'être remis.
- Fiabilité sur UDP
  - La fiabilité peut être garantie au niveau applicatif par le développeur
    - Par l'ajout de numéro de séquence, de date (horloge de Mattern), de timeout pour les reprises ...

      dans les messages contenant les données ou de service.

La classe `java.net.DatagramPacket`

- Représente un datagramme à envoyer ou reçu par UDP ou IP MultiCast
  - Contient un tableau de bytes qui peut être récupéré ou positionné avec les méthodes `getData()` / `setData()`, `setLength()` / `getLength()`.
  - Contient également l'adresse et le port de l'émetteur méthodes `getAddress()`, `getPort()`.
  - Contient également l'adresse et le port du récepteur méthodes `setAddress()`, `setPort()`.

La classe `java.net.DatagramPacket`

- Peut être structuré pour l'envoi de messages structurés
  - DatagramPacket ne peut être dérivé : *classe finale*
  - La "structure" peut contenir un numéro de séquence, horloge de Mattern, … pour assurer un niveau de fiabilité
  - Peut contenir un ***objet sérialisé*** uniquement si l'émetteur et le récepteur sont des JVMs.

Conclusion

- API Réseau de Java
  - `Socket/ServerSocket`, `DatagramSocket`, `MulticastSocket` fournissent une API réseau bas niveau au dessus de TCP/IP, UDP/IP et IP MultiCast.
- Cependant
  - Les messages et flux de données ne sont pas structurés.
  - Solution 1
    - Des outils plus adaptés (RPC, RMI, CORBA) masquent au développeur les détails de la connexion et le codage des messages.
  - Solution 2
    - `java.net` offre aussi des classes `URL` et `URLConnection` pour la récupération de données au dessus de protocoles comme HTTP, FTP, …

Quelques conseils…

- Prévoir la terminaison propre des programmes
- Prévoir les cas d'erreur
- Gérer correctement les exceptions
- Lorsqu'ils ne sont plus utilisés, penser à :
  - Fermer les flux,
  - Fermer les sockets
- Importer les packages utiles
- Commenter le code, …

ANNEXE
Sécurité

Gestionnaire de sécurité

Relai SOCKS

JSSE (Java Secure Socket Extension)

Gestionnaire de sécurité

- Exemple

```
grant {
 permission java.net.SocketPermission
  "localhost:3003", "listen";
 permission java.net.SocketPermission "serveur",
  "connect,accept";
 permission java.net.SocketPermission
  "*.univmed.fr:1000-2000", "connect";
 permission java.net.SocketPermission
  "*.univmed.fr:3000-", "connect";
}
```

- Remarque
  - Pas de distinction entre UDP et TCP.

Relai SOCKS

- Relai sur des connexions TCP au travers d'un pare-feu
  - SOCKS (RFC1928 pour V5)
- Propriétés
  - socksProxyHost, socksProxyPort
  - `java -DsocksProxyHost=socks.cloud9.net -DsocksProxyPort=1080 MyClass`

JSSE
Java Secure Socket Extension

- Extension `javax.net` permettant de sécuriser une socket avec **SSL**
  - Package `javax.net.ssl`
  - Fournit aussi JCE pour les algorithmes crypto.
  - Utilisable avec les RMI, CORBA, HTTP, ...
  - Désormais distribuée dans J2SE 1.4+

JSSE
Java Secure Socket Extension

- Classes
  - Voir la documentation…

    %JAVA_HOME%\docs\guide\security\jsse\JSSERefGuide.html
  - `SSLContext` est initialisée avec un `KeyManager`
  - `SSLSocketFactory` et `SSLServerSocketFactory` sont créés avec `SSLContext`
  - `SSLServerSocket` est créé par `SSLServerSocketFactory`
  - `SSLSocket` est créé par `SSLSocketFactory` ou `SSLServerSocket`

JSSE
Exemple de client

```
public class Jsse_client {
 public static void main(String[] args) {
   String host = args[0];
   int port = Integer.parseInt(args[1]);
   try {
    SSLSocketFactory sslFact =
        (SSLSocketFactory)SSLSocketFactory.getDefault();
    SSLSocket s = (SSLSocket)sslFact.createSocket(host, port);
    s.startHandshake();
    OutputStream out = s.getOutputStream();
    InputStream in = s.getInputStream();
    // Send messages to the server through the OutputStream
    // Receive messages from the server through the InputStream
   } catch (IOException e) { ... }
   }
}
```

JSSE
Exemple de serveur

```
public class Jsse_serveur {
 public static void main(String[] args) {
   int port = Integer.parseInt(args[0]);
   try {
    SSLServerSocketFactory sslSrvFact =
        (SSLServerSocketFactory)
   SSLServerSocketFactory.getDefault();
    SSLServerSocket s =
        (SSLServerSocket)sslSrvFact.createServerSocket(port);
    SSLSocket c = (SSLSocket)s.accept();
    OutputStream out = c.getOutputStream();
    InputStream in = c.getInputStream();
    // Send messages to the client through the OutputStream
    // Receive messages from the client through the InputStream
   } catch (IOException e) {...}
 }
}
```

JSSE
Exemple de client avec authentification

```
public class Jsse_client_authentification {
 public static void main(String[] args) {
  SSLSocketFactory factory = null;
  try {
   String host = args[0];
   int port = Integer.parseInt(args[1]);
   char[] passphrase = "passphrase".toCharArray();
   KeyStore ks = KeyStore.getInstance("JKS");
   ks.load(new FileInputStream("testkeys"), passphrase);
   KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
   kmf.init(ks, passphrase);
   SSLContext ctx = SSLContext.getInstance("TLS");
   ctx.init(kmf.getKeyManagers(), null, null);
   factory = ctx.getSocketFactory();
   SSLSocket socket = (SSLSocket)factory.createSocket(host, port);
   socket.startHandshake();
  } catch (Exception e) {...}
 }
}
```

ANNEXE
Programmation MultiCast

Rappel sur l'IP MultiCast

- Couche de transport non fiable en mode diffusion restreinte
  - Contrôle des erreurs mais pas de reprise sur erreur, pas d'ACK, séquencement des paquets non garanti.
  - Utilisé pour les applications de diffusion restreinte.
    - Vidéo/audio-conférence, cours de la bourse, …

Rappel sur l'IP MultiCast

- UDP Unicast vs IP MultiCast
  - UDP envoie un paquet de données d'un point (hôte) vers un autre (hôte,port).
  - IP Multicast envoie une suite de paquets de données d'un point (hôte) vers un groupe d'N hôtes qui souhaitent recevoir ces données.
- Groupe MultiCast
  - Spécifié par une adresse de classe D (de 224.0.0.1 à 239.255.255.255) et par un port (UDP)
- Voir : *http://www.ipmulticast.com*

Utilisation des sockets en mode diffusion multicast

- Principe
  - Le diffuseur (*multicaster*) rejoint un groupe multicast et envoie un datagramme sur ce groupe
  - Les récepteurs (*receiver*) qui ont rejoint le groupe reçoivent les datagrammes envoyés par les diffuseurs.
- La classe `java.net.MulticastSocket`
  - Dérive de java.net.DatagramSocket
  - Représente le socket en diffusion.
- La classe `java.net.DatagramPacket`
  - Représente un packet (ou Datagramme) à envoyer ou reçu.

Sockets en mode diffusion restreinte

- Coté diffuseur
  - Crée le MulticastSocket

```
MulticastSocket ms = new MulticastSocket();
ms.joinGroup(groupInetAddress);
```

  - Construit un DatagramPacket d'émission

```
byte[] data = new byte[len];
// data doit être "rempli"
// data = {'H', 'e', 'l', 'l', 'o'};
DatagramPacket outputPacket = new
DatagramPacket(data, data.length,
groupInetAddress, port);
```

  - Envoie le DatagramPacket au groupe de diffusion

```
ms.send(outputPacket);
```

Sockets en mode diffusion restreinte

- Coté récepteur
  - crée le MulticastSocket et rejoindre le groupe

```
MulticastSocket ms = new
MulticastSocket(port);
ms.joinGroup(groupInetAddress);
```

  - construit un DatagramPacket de réception

```
byte[] data = new byte[len];
DatagramPacket incomingPacket = new
DatagramPacket(data, data.length);
```

  - reçoit le DatagramPacket diffusé

```
ms.receive(incomingPacket);
```

Code du Diffuseur Exemple

```
public class Multicast_Sender {
 public static void main(String[] args) {
  try { InetAddress ia = InetAddress.getByName(args[0]);
    int port = Integer.parseInt(args[1]);
    MulticastSocket ms = new MulticastSocket();
    ms.joinGroup(ia);
    BufferedReader userInput =
        new BufferedReader(new InputStreamReader(System.in));
    while(true) {
       String theLine = userInput.readLine();
       if (theLine.equals(".")) break;
       byte[] data = theLine.getBytes();
       DatagramPacket dp =
               new DatagramPacket(data,data.length,ia,port);
       ms.send(dp); }
   ms.leaveGroup(ia);
   ms.close();  } catch (Exception e) { ... }
 } }
```

Code du Receveur Exemple

```
public class Multicast_Receiver {
 public static void main(String[] args) {
  try { //paquet de réception
   byte[] buffer = new byte[65509];
   DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
   InetAddress ia = InetAddress.getByName(args[0]);
   int port = Integer.parseInt(args[1]);
   MulticastSocket ms = new MulticastSocket(port);
   ms.joinGroup(ia);
   while(true) {
        ms.receive(dp);
        String s = new String(dp.getData());
        System.out.println(s);
   }
  } catch (Exception e) { ... }
 }
}
```

Remarques sur MulticastSocket

- Un socket n'a pas besoin être membre d'un groupe multicast pour y envoyer de message.
- Adresses de classe D : 224.0.0.0-239.255.255.255
- Adresses de classes D déjà réservées :

  224.0.18.000-224.0.18.255 Dow Jones

  224.0.19.000-224.0.19.063 Walt Disney Company

  ...
- Les applets ne sont pas autorisées à utiliser les sockets multicast.

ANNEXE
Les classes de Connexion

`java.net.URL`

`java.net.URLConnection`

`java.net.HttpURLConnection`

`java.net.JarURLConnection`

java.net.URL

- Définit une URL.
- Méthodes
  - `String getProtocol(), String getHost(), String getPort(), String getRef()`
  - `InputStream openStream()` permet d'obtenir un `InputStream` d'une connexion au serveur
  - `URLConnection openConnection()` permet d'obtenir un `URLConnection`

java.net.URL Constructeurs

- `URL localfile = new URL("/users/donsez/pages/cours/index.html");`
- `URL gamelan = new URL("http://www.gamelan.com/pages/");`
- `URL gamelanGames = new URL(gamelan, "Gamelan.game.html");`
- `URL gamelanNetwork = new URL(gamelan, "Gamelan.net.html");`
- `URL gamelanNetworkBottom = new URL(gamelanNetwork, "#BOTTOM");`
- `URL gamelanNetwork2 = new URL("http", "www.gamelan.com", "/pages/Gamelan.net.html");`

java.net.URL
Méthodes

```
public class ParseURL {
 public static void main(String[] args) throws Exception
  {
  URL aURL = new
  URL("http://java.sun.com:80/docs/books/tutorial/index.
  html#DOWNLOADING");
  System.out.println("protocol = " +
  aURL.getProtocol());
  System.out.println("host = " + aURL.getHost()+ "
  filename = " + aURL.getFile());
  System.out.println("port = " + aURL.getPort() + " ref
  = " + aURL.getRef());
  }
}
```

Utilitaires

- Classe URI
  - Uniform Resource Identifier.
- Encodeur/Décoder
  - Méthodes statiques de conversion :

`x-www-form-urlencoded` vers/depuis `String`

```
static String URLDecoder.decode(String
  urlencoded)
static String URLEncoder.encode(String str)
```

Exemple de récupération d'un document

```
public class URL_Recup_document {
 public static void main(String[] args) {
   URL aurl = new URL(args[0]);
   BufferedReader in = new BufferedReader(
       new InputStreamReader(aurl.openStream()));
   String inputLine;
   while ((inputLine = in.readLine()) != null)
       System.out.println(inputLine);
   in.close();
   }
}
```

La classe de connexion `java.net.URLConnection`

- Définit une connexion à une URL
  - Super-classe abstraite indépendante du schéma de connexion (http, ftp, file, mailto, …)
- Sous-classes
  - `HttpURLConnection`, `JarURLConnection`

java.net.URLConnection

- Utilisation
  - 1- l'instance est récupérée de URL.openConnection()
  - 2- configuration des paramètres de la connexion
    - setAllowUserInteraction, setDoInput, setDoOutput, setIfModifiedSince, setUseCaches, setRequestProperty
  - 3- connexion connect() et obtention d'un InputStream
  - 4- récupération des champs d'entête (header fields)
    - getContent, getHeaderField, getContentEncoding, getContentLength, getContentType, getDate, getExpiration, getLastModifed

Exemple de récupération d'un document

```
public class URLConnection_Recup_document {
 public static void main(String[] args) {
   URL aurl = new URL(args[0]);
   URLConnection aurlcnx = aurl.openConnection();
   BufferedReader in = new BufferedReader(
       new
   InputStreamReader(aurlcnx.getInputStream()));
   String inputLine;
   while ((inputLine = in.readLine()) != null)
       System.out.println(inputLine);
   in.close();
   }
}
```

Exemple d'envoi de mail

```
public class URLConnection_envoi_mail {
 public static void main(String[] args) {
   URL mailto = new URL("mailto:"+args[0]);
   URLConnection mailtocnx= mailto.openConnection();
   mailtocnx.connect();
   PrintStream p =
       new PrintStream(mailtocnx.getOutputStream());
   p.println("Subject: Test\r\n\r\nSalut !");
   p.close();
   }
}
```

java.net.HttpURLConnection

- Sous interface de URLConnection
  - Récupération d'un document via une connexion HTTP.
- Méthodes :
  - Positionnement des champs de la requête HTTP
    - setRequestedMethod()
  - Récupération des champs de la réponse HTTP
    - getResponseCode(), getResponseMessage()
  - Clôture de la session HTTP
    - disconnect() pour les connexions keep-alive
- Usage :
  - Robot de récupération, ...
  - Dialogue client-serveur en tunneling HTTP/TCPIP avec un serveur HTTP.

Exemple de récupération d'un document

```
public class URL_HttpURLConnection_recup_document {
 public static void main(String[] args) {
   URL httpurl = new URL(args[0]);
   HttpURLConnection httpcnx = (HttpURLConnection)httpurl
   .openConnection();
   if(httpcnx.getResponseCode()== HttpURLConnection.HTTP_OK) {
    BufferedReader in = new BufferedReader(
        new InputStreamReader(httpcnx.getInputStream()));
    String inputLine;
    while ((inputLine = in.readLine()) != null)
      System.out.println(inputLine);
    in.close(); }
   else { System.err.println(httpcnx.getResponseMessage()); }
   }
}
```

java.net.JarURLConnection

- Sous interface de URLConnection
  - Récupération d'un Jar file ou d'une de ses entrées.
- Syntaxe des URL
  - jar:<url>!/{entry} !/ est un séparateur
  - un Jar file : jar:http://www.foo.com/bar/baz.jar!/
  - un Jar file : jar:file:/local/dev/bar/baz.jar!/
  - un répertoire : jar:http://www.foo.com/bar/baz.jar!/COM/foo/
  - un entrée : jar:http://www.foo.com/bar/baz.jar!/COM/foo/Quux.class
- Exemple :

```
url = new URL("jar:http://www.foo.com/bar/baz.jar!/");
JarURLConnection
  jarConnection=(JarURLConnection)url.openConnection();
Manifest manifest = jarConnection.getManifest();
```

Les gestionnaires pour URL

- Architecture logicielle ouverte autour d'URL
  - Ajout de nouveaux schémas (https, vod, mcast,...)
  - Ajout de nouveaux types de ressources (image/fract)
  - Personnalisation des gestionnaires existants (http, ...).
- 3 classes d'objets :
  - Gestionnaire de schéma
    - sous-classe de URLStreamHandler
    - par défaut sun.net.www.protocol.<schema>.Handler
  - Gestionnaire de connexion
    - sous-classe de URLConnection
  - Gestionnaire de contenu (type/soustype MIME)
    - sous classe de ContentHandler
    - par défaut : sun.net.www.content.<type>.<soustype>

Les gestionnaires pour URL

- Personnalisation des Gestionnaires
  - Constructeurs d'URL
  - Méthodes d'URL
  - Gestionnaire de schéma
    - setURLStreamHandlerFactory(URLStreamHandlerFactory factory)
  - Gestionnaire de contenu (type/soustype MIME)
    - setContentHandlerFactory(ContentHandlerFactory factory)
- Lire
  - http://java.sun.com/developer/onlineTraining/protocolhandlers/