Communication distante

Introduction

- Ce chapitre aborde
  - la communication par socket
  - la communication par RMI

Réseau

- Java permet les habituelles utilisation de la communication par socket
  - application vers application
  - interrogation de serveur HTTP
  - création de clients et serveurs HTTP, FTP, SMTP, ...
    - de nombreux packages sont disponibles dans le "commons" du consortium Apache
  - lien entre applet et servlet
  - etc.

Réseau

- Le développement d'application communiquant par réseau est grandement facilitée par Java
  - `InetAddress` : une adresse IP
  - `URL` : une URL
  - `HttpURLConnection` : encapsule une connexion vers un serveur HTTP
  - `Socket` : encapsule une socket client
  - `SocketServer` : encapsule un comportement serveur par socket
  - et bien d'autres classes... cf. la documentation

Réseau

- Exemple de requête vers un serveur HTTP

```
public class HttpConnectionTest {

     public static void main(String[] args) throws IOException {
          URL urlServer = new URL("http://www.perdu.com");
          HttpURLConnection connection = (HttpURLConnection) urlServer.openConnection();
          connection.setRequestMethod("GET");
          connection.connect();
          BufferedReader reader = new BufferedReader(
                                        new InputStreamReader(connection.getInputStream()));

          String line;
          while((line = reader.readLine()) != null){
               System.out.println(line);
          }
          connection.disconnect();

     }

}
```

Réseau

- Exemple de récupération des adresses IP

```
public class InetAddressTest {

     public static void main(String[] args) {
          InetAddress LocaleAdresse ;
          InetAddress ServeurAdresse;

          try {
               LocaleAdresse = InetAddress.getLocalHost();
               System.out.println("L'adresse locale est : "+LocaleAdresse );

               ServeurAdresse= InetAddress.getByName("www.perdu.com");
               System.out.println("L'adresse du serveur www.perdu.com : "+ServeurAdresse);

          } catch (UnknownHostException e) {
               e.printStackTrace();
          }
     }
}
```

Socket

- Le socket est un point de terminaison pour les protocoles de transmission de données sous TCP/IP
  - lié à une adresse IP et un numéro de port
- Java propose deux types de sockets
  - le socket client – classe `Socket`
    - ce socket se connecte à un Socket serveur
  - le socket serveur – classe `ServerSocket`
    - ce socket écoute sur un port particulier et accepte des connexions provenant d'un client

Socket

- Java simplifie énormément la manipulation des sockets
- Côté serveur
  - instancier un `ServerSocket`
  - le mettre en attente de connexion de client
  - accepter le client et communiquer avec lui avec les flux d' E/S
- Côté client
  - instancier un `Socket` sur l'adresse + port du serveur
  - récupérer les flux d' E/S pour communiquer

Socket – côté serveur

- Exemple de serveur simple – 1/3

```
public class TimeServer {
     private int port = 12345;
     private ServerSocket server;

     public TimeServer() throws IOException{
          init();
     }

     public TimeServer(int port) throws IOException{
          this.port = port;
          init();
     }

     private void init() throws IOException{
          server = new ServerSocket(port);
     }


...

}
```

Socket – côté serveur

- Exemple de serveur simple – 2/3

```
public class TimeServer {
...

    private void waitConnection() throws IOException{
        Socket sockClient = null;
        InputStream in = null;
        OutputStream out = null;
        System.out.printf("Serveur démarré %s\n",server.getLocalSocketAddress());
        while(true){
            sockClient = server.accept();
            System.out.println(">>> Connexion du client "+sockClient.getInetAddress()+
                                                        " ["+sockClient.getPort()+"]");
            in = sockClient.getInputStream();
            out = sockClient.getOutputStream();
            handleConnection(in,out);
            sockClient.close();
        }
    }

...
}
```

Socket – côté serveur

- Exemple de serveur simple – 3/3

```
public class TimeServer {

...
     private void handleConnection(InputStream in, OutputStream out) throws IOException {
          BufferedReader reader = new BufferedReader(new InputStreamReader(in));
          BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(out));
          String hello = reader.readLine();
          if(hello.equals("HELLO"))
               writer.write(new Date().toString()+"\n");
          writer.flush();
     }

     public static void main(String[] args) throws IOException {
          TimeServer server = new TimeServer();
          server.waitConnection();
     }

}
```

Socket – côté client

- Exemple de client - 1/2

```
public class TimeClient {

     private String hostName;
     private int port;
     private Socket client;

     public TimeClient(String hostName,int port){
          this.hostName = hostName;
          this.port = port;
     }

     public void connect() throws UnknownHostException, IOException{
          client = new Socket(hostName,port);
          System.out.println("=== Client connecté");
          BufferedReader reader = new BufferedReader(new InputStreamReader(client.getInputStream()));
          BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(client.getOutputStream()));
          writer.write("HELLO\n\n");
          writer.flush();
          String reponse = reader.readLine();
          System.out.println("=== "+reponse);
          client.close();
     }
...
}
```

Socket – côté client

- Exemple de client - 1/2

```
public class TimeClient {

...


    public static void main(String[] args) throws UnknownHostException, IOException {
        TimeClient client = new TimeClient("127.0.0.1", 12345);
        client.connect();
    }
}
```

```
Serveur démarré 0.0.0.0/0.0.0.0:12345
>>> Connexion du client /127.0.0.1 [49680]
>>> HELLO
>>> Connexion du client /127.0.0.1 [49681]
>>> HELLO
```

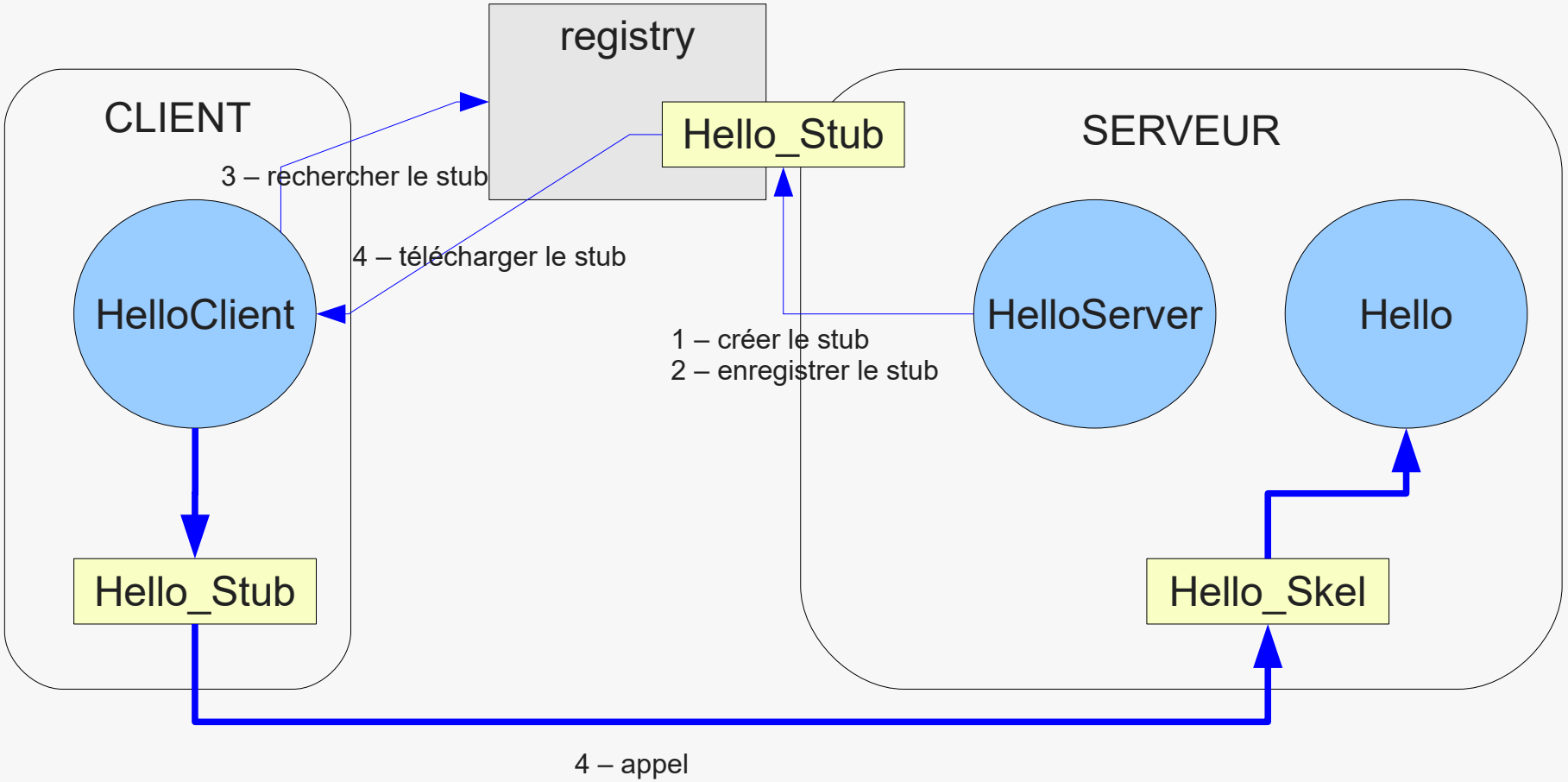
```
=== Client connecté
=== Sat Jan 07 11:48:28 CET 2012
```

RMI

- RMI : Remote Method Invocation
  - permet la construction d'applications réparties en Java
    - comparable aux solution CORBA, DCOM+
  - même principe que le RPC (Remote Procedure Call)
- Cette API a énormément évoluée
  - JDK 1.2 – le skeleton est généré dynamiquement
  - JDK 1.3 – le stub est livré dynamiquement
    - mais le développeur doit quand même le créer
  - JDK 1.5 – élimination du stub
    - les proxys sont créés et délivrés dynamiquement

RMI

- Fonctionnement de base



RMI

- Nécessite
  - un côté serveur
    - une implémentation du service
    - un enregistrement dans un registry
  - un registry
    - qui enregistre les objets et les associe avec une chaîne de caractères
  - un côté client
    - qui utilise l'objet distribué distant, via les couches proxy
  - un réseau...

RMI

- L'objet distribué est accédé par un proxy
  - le proxy implémente la même interface que l'objet distribué
  - le proxy gère les appels vers les méthodes
  - les couches stub et skeleton sont les implémentations du proxy

RMI

- Les appels des méthodes distantes impliquent un envoi de paramètres et un retour de résultats
  - des objets peuvent être échangés
    - par valeur, car les JVM sont différentes
  - les objets échangés doivent être sérialisables
    - interface marqueur `Serializable`
    - les objets référencés dans les objets échangés doivent eux aussi être sérialisables
- RMI est basé sur le protocole JRMP
  - Java Remote Method Protocole

RMI

- Une classe accessible à distance doit implémenter l'interface `UnicastRemoteObject`
  - `package java.rmi`
  - les méthodes doivent renvoyer une exception de type `RemoteException`
- Les proxy stub et skeleton doivent être créés
  - avec le compilateur `rmic`
  - plus nécessaire depuis le JDK 1.5
    - utiliser la classe `UnicastRemoteObject`

RMI

- Développement
  - définir les packages
  - créer les interfaces des services
    - doivent étendre `Remote`
  - coder les classes passées par valeur
    - implémentent `Serializable`
  - coder les implémentations des services
    - doivent étendre `UnicastRemoteObject`
  - générer les stub
    - avec `rmic` – pas nécessaire avec le JDK 1.5

RMI

- Développement - suite
  - lancer le registry
    - utilitaire `rmiregistry`
    - peut être lancé par le code du serveur
  - lancer le serveur
    - les objets distribués sont enregistrés dans le registry
  - coder la partie cliente
    - coder la récupération des stub

RMI

- Créer l'interface du service

```
public interface Calcul extends Remote {
     int add(int a, int b) throws RemoteException;
}
```

- Coder l'implémentation du service

```
public class CalculImpl extends UnicastRemoteObject implements Calcul {

     protected CalculImpl() throws RemoteException {
          super();
     }

     @Override
     public int add(int a, int b) throws RemoteException {
          return a + b;
     }
}
```

RMI

- Une fois les classes compilées il faut générer les stubs
  - utilitaire rmic

```
rmic -classpath . org.antislashn.formation.CalculImpl
```

    - à lancer dans le répertoire contenant les classes compilées
      - répertoire *bin* si Eclipse est utilisé
      - cette étape peut être automatisée sous Eclipse
    - un fichier `CalculImpl_Stub.class` est créé

RMI

- Coder le code du serveur
  - la classe `Naming` gère l'enregistrement des objets dans le registry
    - les objets sont enregistrés sous forme d'URL
      - //host:port/name
    - par défaut le registry écoute sur le port 1099
  - méthodes de `Naming`
    - `bind(String name, Object obj)` : enregistrement de l'objet
    - `rebind(String name, Object obj)` : ré-enregistrement de l'objet
    - `unbind(String name, Object obj)` : dés-enregistrement de l'objet

RMI

- méthodes de `Naming`
  - `list(String name)` : liste les noms des objets
  - `lookup(String name)` : retourne l'objet lié au nom
- Pour des raison de sécurité de chargement de classe il est nécessaire de préciser le dépôt des classes à charger
  - propriété `java.rmi.server.codebase`

```
public class Server {

    public static void main(String[] args) throws RemoteException, MalformedURLException,
                                                                    AlreadyBoundException {
        Calcul calcul = new CalculImpl();
        System.setProperty("java.rmi.server.codebase",
            Calcul.class.getProtectionDomain().getCodeSource().getLocation().toString());
        Naming.rebind("calcul", calcul);
        System.out.println("Calcul est lancé");
    }
}
```

RMI

- Coder le côté client
  - nota : la partie client doit avoir dans son classpath l'interface du service
  - la méthode `lookup` permet de récupérer l'objet

```
public class TestCalculRMI {


    public static void main(String[] args) throws MalformedURLException, RemoteException,
                                                   NotBoundException {
        Calcul calcul = (Calcul) Naming.lookup("rmi://127.0.0.1:1099/calcul");
        System.out.println("Resultat = "+calcul.add(2, 2));
    }
}
```

RMI

- Pour tester
  - lancer le registry
    - utilitaire `rmiregistry`
  - lancer le serveur
  - lancer la partie client

RMI

- Simplification avec l'utilisation de la classe `UnicastRemoteObject`
  - l'interface du service étend toujours `Remote`
    - les méthodes renvoient des `RemoteException`
  - l'implémentation n'a plus besoin d'étendre `UnicastRemoteObject`
  - le serveur utilise la methode `exportObject` pour récupérer un stub
    - le stub sera ensuite enregistré classiquement

RMI

- Codage des classes devant être distribuées

```java
public interface Hello extends Remote{
    String sayHello(String name) throws RemoteException;
}
```

```java
public class HelloImpl implements Hello {

    public HelloImpl() throws RemoteException{}

    @Override
    public String sayHello(String name) {
        return "Hello, "+name;
    }

}
```

RMI

- Codage du serveur
  - le registry est récupéré par la classe `Registry`
    - permet aussi de lancer le registry

```
public class HelloServer {

    public static void main(String[] args) {
        try {
            HelloImpl impl = new HelloImpl();
            System.setProperty("java.rmi.server.codebase",
                HelloImpl.class.getProtectionDomain().getCodeSource().getLocation().toString());
            Hello stub = (Hello) UnicastRemoteObject.exportObject(impl,0);

            Registry registry = LocateRegistry.getRegistry();
            registry.rebind("Hello", stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
        }
    }
}
```