Université de Gabés
Faculté des Sciences de Gabés

Licence en génie logiciel et systèmes d'information
Niveau: 3eme année
Semestre 1

Chapitre 3: Communication dans un système réparti via les Sockets TCP/UDP

Dhikra KCHAOU
dhikrafsegs@gmail.com

2023-2024

Plan du chapitre

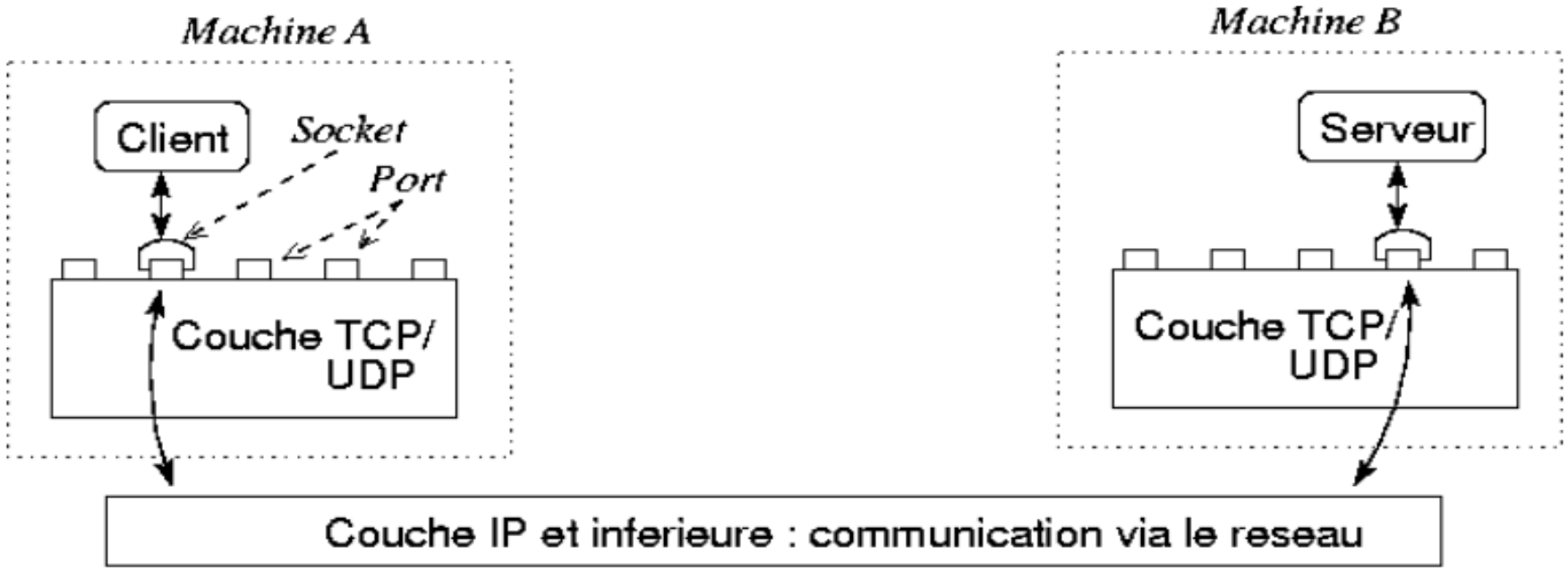
1. Présentation des sockets
2. Sockets UDP
   - La classe DatagramSocket
   - La classe DatagramPacket
3. Socket TCP
   - La classe Socket
   - La classe ServerSocket
4. Conclusion

Sockets

- La manière la plus ancienne de communiquer entre les applications réparties a été l'utilisation **des sockets.**
- Les sockets permettent de communiquer en point à point en mode client/serveur.
- Une socket est un point d'accès aux couches réseaux qui offre des services d'émission et de réception de données via un port.
- Une socket peut-être liée:
  - sur un port précis à la demande du programme
  - Sur un port quelconque libre déterminé par le système

Sockets

- Socket : prise, associée, liée à un port : c'est donc un point d'accès aux couches réseaux



Une socket est un point d'accès aux couches réseau TCP/UDP liée localement à un port.

Client/serveur avec sockets

- Il y a toujours différenciation entre une partie client et une partie serveur
  - Deux rôles distincts au niveau de la communication via TCP/UDP
  - Mais les éléments communiquant peuvent jouer un autre rôle ou les 2 en même temps
- Différenciation pour plusieurs raisons:
  - Identification : on doit connaître précisément la localisation d'un des 2 éléments communiquant
  - Dissymétrie de la communication/connexion
    - Le coté serveur communique via une socket liée à un port précis : port d'écoute
    - Le client initie la connexion ou la communication

Sockets UDP

- Mode datagramme
  - Envois de paquets de données (datagrammes)
  - Pas de connexion entre parties client et serveur
  - Pas de fiabilité ou de gestion de la communication
    - Un paquet peut ne pas arrivé (perdu par le réseau)
    - Un paquet P2 envoyé après un paquet P1 peut arriver avant ce paquet P1 (selon la gestion des routes dans le réseau)

Sockets UDP : Principe de communication

- La partie serveur crée une socket et la lie à un port UDP particulier
- La partie client crée une socket pour accéder à la couche UDP et la lie sur un port quelconque
- Le serveur se met en attente de réception de paquet sur sa socket
- Le client envoie un paquet via sa socket en précisant l'adresse du destinataire (partie serveur: Couple **@IP:port**)
  - Destinataire = @IP de la machine serveur et numéro de port sur lequel est liée la socket de la partie serveur
- Le paquet est reçu par le serveur (sauf problème réseau)
- Si le client envoie un paquet avant que le serveur ne soit prêt à recevoir : le paquet est perdu

Sockets UDP en Java

- Java intègre nativement les fonctionnalités de communication réseau au dessus de TCP-UDP/IP : **Package java.net**
- Classes utilisées pour communication via UDP
  - **InetAddress** : codage des adresses IP
  - **DatagramSocket** : socket mode non connecté (UDP)
  - **DatagramPacket** : paquet de données envoyé via une socket sans connexion (UDP)

Sockets UDP en Java : Classe InetAddress

- Constructeurs : Pas de constructeurs, on passe par des méthodes statiques pour créer un objet.
- Méthodes
  - **public static InetAddress getByName (String host) throws UnknownHostException**
    - Détermine l'adresse IP d'une machine dont le nom est passé en paramètre
    - L'exception est levée si le service de nom (DNS...) du système ne trouve pas de machine du nom passé en paramètre sur le réseau
    - Si l'adresse IP est indiquée sous forme de chaîne ("192.12.23.24") au lieu de son nom, le service de nom n'est pas utilisé

Sockets UDP en Java : Classe InetAddress

- Méthodes (suite)
  - **public static InetAddress getLocalHost() throws UnknownHostException**
    - Retourne l'adresse IP de la machine sur laquelle tourne le programme, c'est-à-dire l'adresse IP locale
  - **public String getHostName()**
    - Retourne le nom de la machine dont l'adresse est codée par l'objet InetAddress
  - **public String getCanonicalHostName()**
    - Retourne le nom de domaine complet pour cette adresse IP.

Sockets UDP en Java : Classe InetAddress

- Méthodes (suite)
  - **public byte[] getAddress()**
    - Retourne l'adresse IP d'un objet sous forme d'un tableau d'octet
  - **public String getHostAddress()**
    - Retourne l'adresse IP sous forme d'une chaine de caractères

Sockets UDP en Java : Classe DatagramPacket

- La classe DatagramPacket représente une structure des données en mode datagramme
- Constructeurs
  - **public DatagramPacket(byte[] buf, int length)**
    - Création d'un paquet pour recevoir des données (sous forme d'un tableau d'octets)
    - Les données reçues seront placées dans **buf**
    - **length** précise la taille max de données à lire
      - Ne pas préciser une taille plus grande que celle du tableau
      - En général, length = taille de buf

Sockets UDP en Java : Classe DatagramPacket

- Constructeurs (suite)
  - **public DatagramPacket(byte[] buf, int length, InetAddress address, int port)**
    - Création d'un paquet pour envoyer des données (sous forme d'un tableau d'octets)
    - **buf** : contient les données à envoyer
    - **length** : longueur des données à envoyer (Ne pas préciser une taille supérieure à celle de buf)
    - **address** : adresse IP de la machine destinataire des données
    - **port** : numéro de port distant (sur la machine destinataire) où les données vont être envoyées

Sockets UDP en Java : Classe DatagramPacket

- Méthodes « get »
  - **public InetAddress getAddress()**
    - Si paquet à envoyer : adresse de la machine destinataire
    - Si paquet reçu : adresse de la machine qui a envoyé le paquet
  - **public int getPort()**
    - Si paquet à envoyer : port destinataire sur la machine distante
    - Si paquet reçu : port utilisé par le programme distant pour envoyer le paquet
  - **public byte[] getData()**
    - Données contenues dans le paquet
  - **public int getLength()**
    - Si paquet à envoyer : longueur des données à envoyer
    - Si paquet reçu : longueur des données reçues

Sockets UDP en Java : Classe DatagramPacket

- Méthodes « set »
  - **public void setAddress(InetAdress adr)**
    - Positionne l'adresse IP de la machine destinataire du paquet
  - **public void setPort(int port)**
    - Positionne le port destinataire du paquet pour la machine distante
  - **public void setData(byte[] data)**
    - Positionne les données à envoyer
  - **public int setLength(int length)**
    - Positionne la longueur des données à envoyer

Sockets UDP en Java: Classe DatagramPacket

- Java n'impose aucune limite en taille pour les tableaux d'octets circulant dans les paquets UDP, mais
  - Pour tenir dans un seul datagramme IP, le datagramme UDP ne doit pas contenir plus de 65467 octets de données (Un datagramme UDP est rarement envoyé via plusieurs datagrammes IP)
  - Mais en pratique : il est conseillé de ne pas dépasser 8176 octets (Car la plupart des systèmes limitent à 8 Ko la taille des datagrammes UDP)
  - Pour être certain de ne pas perdre de données : 512 octets max
  - Si datagramme UDP trop grand : les données sont tronquées
- Si tableau d'octets en réception est plus petit que les données envoyées (Les données reçues sont généralement tronquées)

Sockets UDP en Java : Classe DatagramSocket

- Socket en mode datagramme
- Constructeurs
  - **public DatagramSocket() throws SocketException**
    - Crée une nouvelle socket en la liant à un port quelconque libre
    - Exception levée en cas de problème
  - **public DatagramSocket(int port) throws SocketException**
    - Crée une nouvelle socket en la liant au port local précisé par le paramètre port
    - Exception levée en cas de problème : notamment quand le port est déjà occupé

Sockets UDP en Java : Classe DatagramSocket

- Méthodes d'émission/réception de paquet
  - **public void send(DatagramPacket p) throws IOException**
    - Envoie le paquet passé en paramètre. Le destinataire est identifié par le couple @IP/port précisé dans le paquet
    - Exception levée en cas de problème d'entrée/sortie
  - **public void receive(DatagramPacket p) throws IOException**
    - Reçoit un paquet de données
    - Bloquant tant qu'un paquet n'est pas reçu
    - Quand un paquet arrive, les attributs de p sont modifiés
      - Les données reçues sont copiées dans le tableau passé en paramètre lors de la création de p et sa longueur est positionnée avec la taille des données reçues
      - Les attributs d'@IP et de port de p contiennent l'@IP et le port de la socket distante qui a émis le paquet

Sockets UDP en Java : Classe DatagramSocket

- Autres méthodes
  - **public void close()**
    - Ferme la socket et libère le port à laquelle elle était liée
  - **public int getLocalPort()**
    - Retourne le port local sur lequel est liée la socket
- En utilisant la classe DatagramSocket, on peut créer un canal (mais toujours en mode non connecté)
  - Pour restreindre la communication avec un seul destinataire distant
  - Car par défaut, une machine peut recevoir sur la socket des paquets venant de n'importe où

Sockets UDP en Java : Classe DatagramSocket

- Réception de données : via méthode receive
  - Méthode bloquante sans contrainte de temps : peut rester en attente indéfiniment si aucun paquet n'est jamais reçu
  - On peut préciser un délai maximum d'attente
    - **public void setSoTimeout(int timeout) throws SocketException**
    - L'appel de la méthode **receive** sera bloquante pendant au plus timeout millisecondes
    - Une méthode **receive** se terminera alors de 2 façons
      - Elle retourne normalement si un paquet est reçu en moins du temps positionné par l'appel de setSoTimeout
      - L'exception SocketTimeoutException est levée pour indiquer que le délai s'est écoulé avant qu'un paquet ne soit reçu (SocketTimeoutException est une sous-classe de IOException)

Sockets UDP Java – exemple coté client

```
InetAddress adr;
DatagramPacket packet;
DatagramSocket socket;
// adr contient l'@IP de la partie serveur
adr = InetAddress.getByName("scinfr222");
// données à envoyer : chaîne de caractères
byte[] data = (new String(« Bonjour")).getBytes();
// création du paquet avec les données et en précisant l'adresse du serveur
// (@IP et port sur lequel il écoute : 7777)
packet = new DatagramPacket(data, data.length, adr, 7777);
// création d'une socket, sans la lier à un port particulier
socket = new DatagramSocket();
// envoi du paquet via la socket
socket.send(packet);
```

Sockets UDP Java – exemple coté serveur

```
DatagramSocket socket;
DatagramPacket packet;
// création d'une socket liée au port 7777
DatagramSocket socket = new DatagramSocket(7777);
// tableau de 15 octets qui contiendra les données reçues
byte[] data = new byte[15];
 // création d'un paquet en utilisant le tableau d'octets
packet = new DatagramPacket(data, data.length);
// attente de la réception d'un paquet. Le paquet reçu est placé dans packet et ses données dans data.
 socket.receive(packet);
// récupération et affichage des données (une chaîne de caractères)
String chaine = new String(packet.getData(), 0, packet.getLength());
System.out.println(" recu : "+chaine);
```

Sockets UDP en Java – exemple suite

- La communication se fait souvent dans les 2 sens
  - Le serveur doit donc connaître la localisation du client
  - Elle est précisée dans le paquet qu'il reçoit du client
- Réponse au client, coté serveur

System.out.println(" ca vient de : "+packet.getAddress()+":"+ packet.getPort());

// on met une nouvelle donnée dans le paquet

(qui contient donc le couple @IP/port de la socket coté client)

packet.setData((new String("bien recu")).getBytes());

// on envoie le paquet au client

socket.send(packet);

Sockets UDP en Java – exemple suite

- Réception réponse du serveur, coté client

// attente paquet envoyé sur la socket du client

socket.receive(packet);

// récupération et affichage de la donnée contenue dans le paquet

String chaine = new String(packet.getData(), 0, packet.getLength());
System.out.println(" recu du serveur : "+chaine);

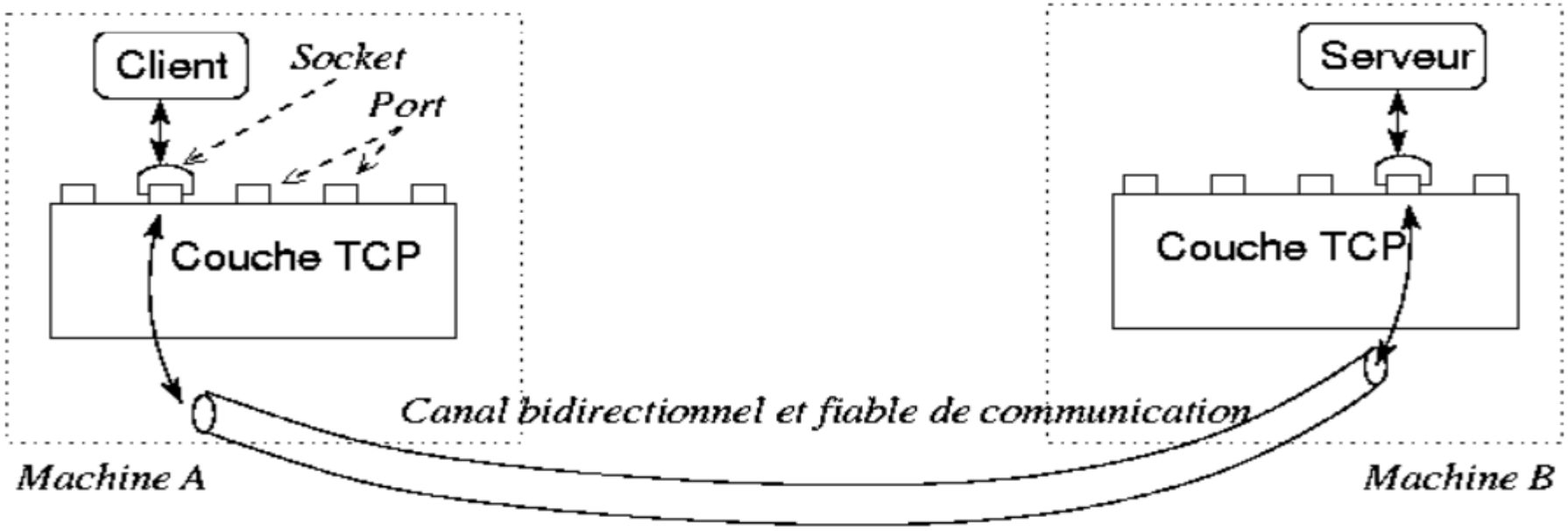
Critique sockets UDP

- Avantages
  - Simple à programmer (et à appréhender)
- Inconvénients
  - Pas fiable
  - Ne permet d'envoyer que des tableaux de bytes

Sockets TCP

Sockets TCP

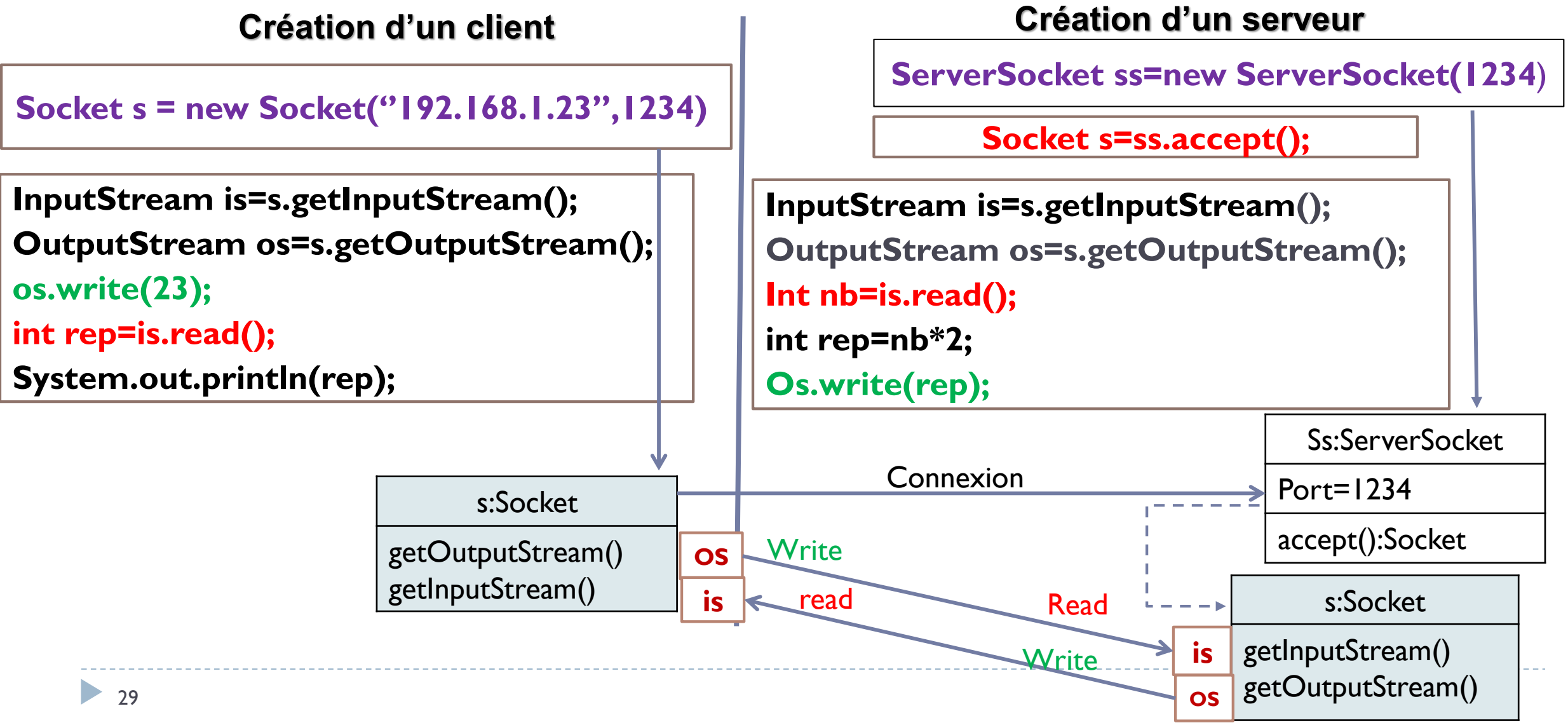
- Fonctionnement en mode connecté
  - Données envoyées dans un « tuyau » et non pas par paquet à travers des flux de données (Flux Java)
  - Fiable : la couche TCP assure que
    - Les données envoyées sont toutes reçues par la machine destinataire
    - Les données sont reçues dans l'ordre où elles ont été envoyées



Sockets TCP: Principe de communication

- Le serveur lie une socket d'écoute sur un certain port bien précis et appelle un service d'attente de connexion de la part d'un client
- Le client appelle un service pour ouvrir une connexion avec le serveur : Il récupère une socket (associée à un port quelconque par le système)
- Du coté du serveur, le service d'attente de connexion retourne une socket de service (associée à un port quelconque) : C'est la socket qui permet de dialoguer avec ce client
- Comme avec sockets UDP : le client et le serveur communiquent en envoyant et recevant des données via leur socket

Sockets TCP: Principe de communication



Sockets TCP en Java

- Les classes du package **java.net** utilisées pour communiquer via TCP:
  - **InetAddress** : codage des adresses IP : Même classe que celle décrite dans la partie UDP et l'usage est identique
  - **Socket** : socket mode connecté du coté client
  - **ServerSocket** : socket d'attente de connexion du coté server

Sockets TCP en Java: la classe socket

- Cette classe implémente des sockets client (également appelés simplement « sockets »).
- Constructeurs
  - **public Socket(InetAddress address, int port) throws IOException**
    - Crée une socket locale et la connecte à un port distant d'une machine distante identifié par le couple adresse/port
  - **public Socket(String address, int port) throws IOException,UnknownHostException**
    - Idem mais avec nom de la machine au lieu de son adresse IP codée
    - Lève l'exception UnknownHostException si le service de nom ne parvient pas à identifier la machine

Sockets TCP en Java: la classe socket

- Méthodes d'émission/réception de données
  - Contrairement aux sockets UDP, les sockets TCP n'offre pas directement de services pour émettre/recevoir des données
  - On récupère les flux d'entrée/sorties associés à la socket
    - **OutputStream getOutputStream()** : Retourne le flux de sortie permettant d'envoyer des données via la socket
    - **InputStream getInputStream()** : Retourne le flux d'entrée permettant de recevoir des données via la socket
  - Fermeture d'une socket
    - **public close() :** Ferme la socket et rompt la connexion avec la machine distante

Sockets TCP en Java: la classe socket

- Méthodes « get »
  - **int getPort()** : Renvoie le port distant avec lequel est connecté la socket
  - **InetAddress getAddress()** : Renvoie l'adresse IP de la machine distante
  - **int getLocalPort()** : Renvoie le port local sur lequel est liée la socket
  - **public void setSoTimeout(int timeout) throws SocketException**
    - Positionne l'attente maximale en réception de données sur le flux d'entrée de la socket
    - Si temps dépassé lors d'une lecture : exception **SocketTimeoutException** est levée (Par défaut : temps infini en lecture sur le flux)

Sockets TCP en Java: la classe ServerSocket

- Socket d'attente de connexion, coté serveur uniquement
  - Constructeurs
  - **public ServerSocket(int port) throws IOException**
    - Crée une socket d'écoute (d'attente de connexion de la part du client)
    - La socket est liée au port dont le numéro est passé en paramètre : L'exception est levée notamment si ce port est déjà lié à une socket
  - Méthodes
  - **Socket accept() throws IOException**
    - Attente de connexion d'un client distant
    - Quand la connexion est faite, retourne une socket permettant de communiquer avec le client : socket de service
  - **void setSoTimeout(int timeout) throws SocketException**
    - Positionne le temps maximum d'attente de connexion sur un accept
    - Si temps écoulé, l'accept lève l'exception SocketTimeoutException

Sockets TCP Java: exemple coté client

Envoi d'une chaîne par le client et réponse sous forme d'une chaîne par le serveur

- Coté client

// adresse IP du serveur

InetAddress adr = InetAddress.getByName("scinfr222");

// ouverture de connexion avec le serveur sur le port 7777

Socket socket = new Socket(adr, 7777);

// construction de flux objets à partir des flux de la socket

ObjectOutputStream output = new ObjectOutputStream(socket.getOutputStream());

ObjectInputStream input = new ObjectInputStream(socket.getInputStream());

Sockets TCP Java : exemple coté client

// écriture d'une chaîne dans le flux de sortie : c'est-à-dire envoi de données au serveur

```
output.writeObject(new String("Bonjour"));
```

// attente de réception de données venant du serveur (avec le readObject), on sait qu'on attend une chaîne, on peut donc faire un cast directement

```
String chaine = (String)input.readObject();
System.out.println(" reçu du serveur : "+chaine);
```

Sockets TCP Java – exemple coté serveur

// serveur positionne sa socket d'écoute sur le port local 7777

ServerSocket serverSocket = new ServerSocket(7777);

// se met en attente de connexion de la part d'un client distant

Socket socket = serverSocket.accept();

// connexion acceptée : récupère les flux objets pour communiquer avec le client qui vient de se connecter

ObjectOutputStream output = new ObjectOutputStream(socket.getOutputStream());

ObjectInputStream input = new ObjectInputStream(socket.getInputStream());

// attente les données venant du client

String chaine = (String)input.readObject();

System.out.println(" recu : "+chaine);

Sockets TCP Java – exemple coté serveur

// affiche les coordonnées du client qui vient de se connecter
System.out.println(" ca vient de : " +socket.getInetAddress()+":"+socket.getPort());

// envoi d'une réponse au client

output.writeObject(new String("bien recu"));

- Quand manipule des flux d'objets, il est souvent utile de vérifier le type de l'objet reçu) en utilise **instanceof**
  - Exemple

```
String chaine; Personne pers;
Object obj = input.readObject();
if (obj instanceof String) chaine = (String)obj;
if (obj instanceof Personne) pers = (Personne)obj;
```

Critique sockets TCP

- Avantages
  - Niveau d'abstraction plus élevé qu'avec UDP
    - Mode connecté avec phase de connexion explicite
    - Flux d'entrée/sortie
  - Fiable
- Inconvénients
  - Plus difficile de gérer plusieurs clients en même temps
    - Nécessite du parallélisme avec des threads
    - Mais oblige une bonne structuration coté serveur

Bibliographie

- Pollet Yann, Architectures logicielles réparties - Du client-serveur au cloud computing, 2019, ellipses edition.
- Systèmes distribués Sockets TCP/UDP et leur mise en oeuvre en Java, Eric Cariou, Université de Pau et des Pays de l'Adour.
- Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification, https://docs.oracle.com/en/java/javase/21/docs/api/index.html