



# SYSTEMES REPARTIS

Enseignante: Louhichi Soumaya

Année universitaire 2018 -2019

# PLAN

## Introduction aux systèmes répartis

- Présentation générale
- Quelques exemples des systèmes répartis
- Intérêts des systèmes répartis
- Propriétés des systèmes répartis

# Présentation générale

## Système distribué en opposition à un système centralisé

### ❖ Système centralisé

- Tout est localisé sur la même machine et accessible par le programme
- Système logiciel s'exécutant sur une seule machine
- Les applications accèdent localement aux ressources nécessaires (données, code, périphériques, mémoire ...)

### ❖ Système distribué

- Ensemble d'ordinateurs indépendants connectés en réseau et communiquent via ce réseau
- Cet ensemble apparaît du point de vue de l'utilisateur comme une seule entité
- Généralement, on utilise la même terminologie
- Systèmes répartis ⇔ Systèmes distribués (distributed systems)

# Présentation générale

- ❖ Vision matérielle d'un système distribué :  
architecture matérielle
  - Machine multiprocesseurs avec mémoire partagée
  - Cluster d'ordinateurs dédiés au calcul/traitement massif parallèle
  - Ordinateurs standards connectés en réseau
- ❖ Vision logicielle d'un système distribué
  - Système logiciel composé de plusieurs entités s'exécutant indépendamment et en parallèle sur un ensemble d'ordinateurs connectés en réseau

# Présentation générale

- **Pourquoi distribuer ?**
- Equilibrage de charge (load balancing) pour garantir la qualité de service (QoS)
- Economie:
- Partage de ressources : imprimante partagée
- Construire à partir de l'existant.

# Exemples des systèmes répartis

## ❖ Serveur de fichiers

- Accès aux fichiers de l'utilisateur quelque soit la machine utilisée

## ❖ Un serveur de fichiers

- Machines du département informatique: Sur toutes les machines : /home/ali est le « home directory » de l'utilisateur ali
- Physiquement : fichiers se trouvent uniquement sur le serveur
- Virtuellement : accès à ces fichiers à partir de n'importe quelle machine cliente en faisant « croire » que ces fichiers sont stockés localement

# Exemples des systèmes répartis

## ❖ Serveur de fichier (suite)

### ● Intérêts

- Accès aux fichiers à partir de n'importe quelle machine
- Système de sauvegarde associé à ce serveur
- Transparent pour l'utilisateur

### ● Inconvénients

- Si réseau ou le serveur plante : plus d'accès aux fichiers

# Exemples des systèmes répartis

- ❖ Autre exemple de système distribué : Web
  - Un serveur web auquel se connecte un nombre quelconque de navigateurs web (clients)
- Accès à distance à l'information
- Accès simple: Serveur renvoie une page HTML statique qu'il stocke localement
- Traitement plus complexe:
- Serveur interroge une base de données pour générer dynamiquement le contenu de la page
- Transparent pour l'utilisateur : les informations s'affichent dans son navigateur quelque soit la façon dont le serveur les génère



# Intérêts des systèmes répartis

- Utiliser et partager des ressources distantes
  - Un même service peut être utilisé par plusieurs acteurs, situés à des endroits différents
  - Système de fichiers : utiliser ses fichiers à partir de n'importe quelle machine
  - Imprimante : partagée entre toutes les machines
- Optimiser l'utilisation des ressources disponibles
  - Calculs scientifiques distribués sur un ensemble de machines
- Système plus robuste
  - Duplication pour fiabilité : deux serveurs de fichiers dupliqués, avec sauvegarde
  - Plusieurs éléments identiques pour résister à la montée en charge ...

# Inconvénients des systèmes répartis

- S'il y a un problème au niveau du réseau
  - Le système marche mal ou plus du tout
- Bien souvent, un élément est central au fonctionnement du système : serveur
  - Si un serveur se plante : plus rien ne fonctionne
  - Goulot potentiel d'étranglement si débit d'information très important
- Sans élément central
  - Gestion du système totalement décentralisée et distribuée
  - Nécessite la mise en place d'algorithmes +/- complexes

# Propriétés des systèmes distribués

- Un **système réparti** (ou **distribué** de «distributed system» )
  - est un système comprenant un ensemble de **processus** et un **système de communication**
  - **Ensemble composé d'éléments reliés par un système de communication**
    - les éléments ont des fonctions de traitement (processeurs), de stockage (mémoire), de relation avec le monde extérieur (capteurs, actionneurs)
    - Les différents éléments du système ne fonctionnent pas indépendamment mais collaborent à une ou plusieurs tâches communes.
- ↳ Conséquence : une partie au moins de l'état global du système est partagée entre plusieurs éléments (sinon, on aurait un fonctionnement indépendant)

# Propriétés des systèmes distribués

- Système réparti = multiple ressources, autonomes, indépendants, de différents types, reliées par un réseau de communication.
  - Autonomes: peuvent travailler seules ou travailler en collaboration avec les autres
  - Indépendants : n'ont pas besoin d'informations sur le reste du système

# Propriétés des systèmes distribués

- **Hétérogénéité** : Les composants hétérogènes doivent être capables de « travailler » ensemble
  - Des machines utilisées (puissance, architecture matérielle...)
  - Des systèmes d'exploitation tournant sur ces machines
  - Des langages de programmation des éléments logiciels formant le système
  - Des réseaux utilisés : impact sur performances, débit, disponibilité ...
    - Réseau local rapide
    - Internet
    - Réseaux sans fil
- Exemple hétérogénéité des données : codage des entiers

# Propriétés des systèmes distribués

## ❖ **Fiabilité des systèmes distribués**

Nombreux points de pannes ou de problèmes potentiels:

- Réseau
  - Une partie du réseau peut-être inaccessible
  - Les temps de communication peuvent varier considérablement selon la charge du réseau
  - Le réseau peut perdre des données transmises
- Machine
  - Une ou plusieurs machines peut planter, engendrant une paralysie partielle ou totale du système
  - Peut augmenter la fiabilité par redondance, duplication de certains éléments
    - Mais rend plus complexe la gestion du système

# Propriétés des systèmes distribués

## ❖ **Fiabilité des systèmes distribués**

- **Tolérance aux fautes**
  - Capacité d'un système à gérer et résister à un ensemble de problèmes
  - Le système doit pouvoir fonctionner (au moins de façon dégradée) même en cas de défaillance de certains de ses éléments
  - Le système doit pouvoir résister à des perturbations du système de communication (perte de messages, déconnexion temporaire, performances dégradées)
  - Le système doit pouvoir facilement s'adapter pour réagir à des changements d'environnement ou de conditions d'utilisation

# Propriétés des systèmes distribués

## ❖ Sécurité des systèmes distribués

- Nature d'un système distribué fait qu'il est beaucoup plus sujet à des attaques
  - Communications à travers le réseau peuvent être interceptées
  - On ne connaît pas toujours bien un élément distant avec qui on communique
- Le système doit pouvoir résister à des attaques contre sa sécurité (violation de la confidentialité, de l'intégrité, usage indu de ressources, déni de service)
- **Solutions**
  - Connexion sécurisée par authentification avec les éléments distants
  - Cryptage des messages circulant sur le réseau



# Propriétés des systèmes distribués

## ❖ **Transparence**

- Fait pour une fonctionnalité, un élément est invisible ou caché à l'utilisateur ou un autre élément formant le système distribué
  - Devrait plutôt parler d'opacité dans certains cas ...
- But: cacher l'architecture, le fonctionnement de l'application ou du système distribué pour apparaître à l'utilisateur comme une application unique cohérente
- L'ISO définit plusieurs transparences (norme RM-ODP)
  - Accès, localisation, concurrence, réplication, mobilité, panne, performance, échelle

# Propriétés des systèmes distribués




- **Transparence d'accès**
  - Accès à des ressources distantes aussi facilement que localement
  - Accès aux données indépendamment de leur format de représentation
- **Transparence de localisation**
  - Accès aux éléments/ressources indépendamment de leur localisation
- **Transparence de concurrence**
  - Exécution possible de plusieurs processus en parallèle avec utilisation de ressources partagées
  - Offrir et gérer les accès concurrent aux ressources partagées:
    - Garantir un ordonnancement « juste » et performant
    - Eviter les inter blocages
- **Transparence de réplication**
  - Possibilité de dupliquer certains éléments/ressources pour augmenter la fiabilité

# Propriétés des systèmes distribués

- Transparence de mobilité
  - Possibilité de déplacer des éléments/ressources
- Transparence de panne
  - Doit supporter qu'un ou plusieurs éléments tombe en panne
- Transparence de performance
  - Possibilité de reconfigurer le système pour en augmenter les performances
- Transparence d'échelle
  - Doit supporter l'augmentation de la taille du système (nombre d'éléments, de ressources ...)
  - Le système doit préserver ses performances lorsque sa taille croît (nombre d'éléments, nombre d'utilisateurs, étendue géographique)

# Propriétés des systèmes distribués

- **Quelques difficultés**

Propriété	Difficulté engendrée
 <b>Asynchronisme</b> du système de communication (pas de borne supérieure stricte pour le temps de transmission d'un message)	⇒ Difficulté de détecter les défaillances
 <b>Dynamisme</b> (la composition du système change en permanence)	⇒ difficulté de définir un état global du système ⇒ Difficulté d'administrer le système
 <b>Grande taille</b> (nombre de composants, d'utilisateurs, dispersion géographique)	⇒ la capacité de croissance (scalability) est une propriété importante, mais difficile à réaliser

# Applications réparties

- ❖ Distinction entre “système” et “application”
  - Système : gestion des ressources communes et de l’infrastructure, lié de manière étroite au matériel sous-jacent
    - Système d’exploitation : gestion de chaque élément
    - Système de communication : échange d’information entre les éléments
    - Caractéristiques communes : cachent la complexité du matériel et des communications, fournissent des services communs de plus haut niveau d’abstraction
  - Application : réponse à un problème spécifique, fourniture de services à ses utilisateurs (qui peuvent être d’autres applications)
  - Utilise les services généraux fournis par le système
  - La distinction n’est pas toujours évidente, car certaines applications peuvent directement travailler à bas niveau (au contact du matériel). Exemple : systèmes embarqués

# Applications réparties

## Système réparti

- est un ensemble de systèmes calculatoires autonomes (exp: ordinateur, serveur, terminal, etc.) sans mémoire physique commune qui communiquent à travers un réseau quelconque.

## Application répartie

- est un ensemble de processus qui tournent sur un système réparti afin de fournir ou utiliser un service déterminé.

# Applications réparties

- **Application répartie:** c'est une application découpées en plusieurs unités (composants) où:
  - Chaque unité ou ensemble d'unités peut être placée sur une machine différente.
  - Chaque unité peut s'exécuter sur un système différent.
  - Chaque unité peut être programmé dans un langage différent.

# Applications réparties

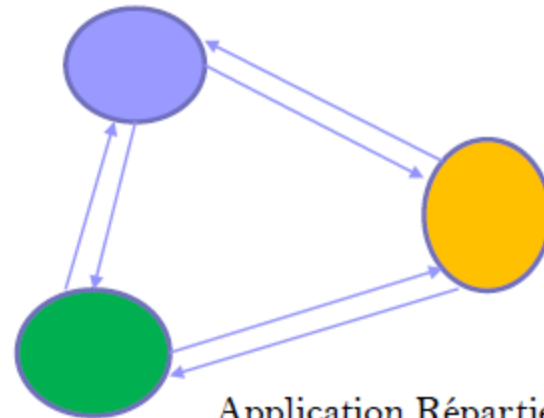
## ❖ Construction d'une application répartie

Les étapes de construction d'une application répartie sont :

1. Identifier les éléments fonctionnels de l'application pour les regrouper au sein des unités
2. Estimer les interactions entre les unités.
3. Définir le schéma organisationnel de l'application.



App. Monolithique



Application Répartie



# Applications réparties

- ❖ Exemples d'applications réparties
  - Navigation web.
  - Guichets de banque (GAB (Guichet Automatique de Banque), DAB (Distributeur Automatique de Banque)).
  - Commerce électronique,

# Applications réparties

- Exemple : Télévision interactive
  - Systèmes de plus en plus présent
  - Programme à la demande
  - Jeux interactif
  - Pour cela il suffit d'un poste de télévision + un abonnement au réseau.
- Il s'agit d'une application « Grand public »
  - Le réseau doit être extensible
  - Et quelque soit le nombre des abonnées
  - La qualité de services doit rester inchangé
  - Ce qu'on appelle une application qui supporte le Passage à l'échelle

# Applications réparties

- ❖ Programmation classique versus programmation répartie
- La plupart des applications réparties sont de type client/serveur: le client demande des services à un serveur
- En programmation classique, lorsque un programme a besoin d'un service, il appelle localement une fonction/procédure/méthode d'une librairie, d'un objet, etc.
- En programmation répartie, l'appel de fonction/procédure/méthode peut se faire à distance
  - ➡ Proposer des méthodes/concepts/outils permettant de simplifier le développement d'application réseau client/serveur, en essayant de s'abstraire de l'aspect "distant"

# Applications réparties

## ❖ Programmation classique versus programmation répartie

Programmation classique	Programmation répartie
<ul style="list-style-type: none"><li>• L'utilisateur du service et le fournisseur de service se trouvent sur la même machine :</li><li>• Même OS</li><li>• Même espace mémoire</li><li>• Même capacité de calcul CPU</li><li>• Pas de problème de transport</li><li>• Disponibilité du service assuré (tant que l'on a accès à la librairie)</li></ul>	<ul style="list-style-type: none"><li>• L'utilisateur et le fournisseur de service ne se trouvent pas sur la même machine: deux machines différentes (sans compter celles traversées)</li><li>• OS différents</li><li>• Espace mémoire non unitaire : "passer un pointeur comme argument" ?</li><li>• Problème de transport : pare-feu (firewall), réseau, etc.</li><li>• Retrouver le service ? où se trouve-t-il ? qui le propose ?</li></ul>

# Applications réparties

## ❖ Programmation classique versus programmation répartie

Programmation classique	Programmation répartie
<ul style="list-style-type: none"><li>• Un même langage de programmation (sinon utilisation de binding)</li><li>• Même paradigme de programmation</li><li>• Même représentation des types de base</li><li>• Même représentation de l'information composite</li></ul>	<ul style="list-style-type: none"><li>• Deux langages différents</li><li>• Représentation de l'information composite différente</li><li>• Association des paramètres effectifs aux paramètres formels ?</li><li>• comment gérer les différents types de passage de paramètre ?</li><li>• Paradigmes de programmation différents : qu'est ce qu'un objet pour un langage procédural ? comment gérer les erreurs ?</li></ul>

# Modèles d'interaction dans un système distribué

- ◆ Les éléments distribués interagissent, communiquent entre eux selon plusieurs modèles possibles
  - ◆ Client/serveur
  - ◆ Diffusion de messages
  - ◆ Mémoire partagée
  - ◆ Pair à pair
  - ◆ ...
- ◆ Abstraction/primitive de communication basique
  - ◆ Envoi de message d'un élément vers un autre élément
  - ◆ A partir d'envois de messages, peut construire les protocoles de communication correspondant à un modèle d'interaction

# Modèles d'interaction dans un système distribué

## ◆ Rôle des messages

### ◆ Données échangées entre les éléments

- ◆ Demande de requête
- ◆ Résultat d'une requête
- ◆ Donnée de toute nature
- ◆ ...

### ◆ Gestion, contrôle des protocoles

- ◆ Acquiescement : message bien reçu
- ◆ Synchronisation, coordination ...



# Modèles d'interaction dans un système distribué

- Modèle client/serveur
  - ◆ 2 rôles distincts
    - ◆ Client : demande que des requêtes ou des services lui soient rendus
    - ◆ Serveur : répond aux requêtes des clients
  - ◆ Interaction
    - ◆ Message du client vers le serveur pour faire une requête
    - ◆ Exécution d'un traitement par le serveur pour répondre à la requête
    - ◆ Message du serveur vers le client avec le résultat de la requête
  - ◆ Exemple : serveur Web
    - ◆ Client : navigateur Web de l'utilisateur
    - ◆ Requêtes : récupérer le contenu d'une page HTML gérée ou générée par le serveur



# Modèles d'interaction dans un système distribué

- Modèle client/serveur
  - ◆ Modèle le plus répandu
    - ◆ Fonctionnement simple
    - ◆ Abstraction de l'appel d'un service : proche de l'appel d'une opération sur un élément logiciel
      - ◆ Interaction de base en programmation
  - ◆ Particularités du modèle
    - ◆ Liens forts entre le client et le serveur
    - ◆ Un client peut aussi jouer le rôle de serveur (et vice-versa) dans une autre interaction
    - ◆ Nécessité généralement pour le client de connaître précisément le serveur (sa localisation)
      - ◆ Ex : URL du site Web
    - ◆ Interaction de type « 1 vers 1 »

# Modèles d'interaction dans un système distribué

- Diffusion des messages
  - ◆ 2 rôles distincts
    - ◆ Emetteur : envoie des messages (ou événements) à destination de tous les récepteurs
      - ◆ Diffusion (broadcast)
        - ◆ Possibilité de préciser un sous-ensemble de récepteurs (multicast)
    - ◆ Récepteurs : reçoivent les messages envoyés
  - ◆ Interaction
    - ◆ Emetteur envoie un message
    - ◆ Le middleware s'occupe de transmettre ce message à chaque récepteur

# Modèles d'interaction dans un système distribué

## ❖ Diffusion des messages

- ❑ Une application produit des messages (producteur) et une application les consomme (consommateur)
- ❑ Le producteur (l'émetteur) et le consommateur (récepteur) ne communiquent pas directement entre eux mais utilisent un objet de communication intermédiaire (boîte aux lettres ou file d'attente)

## • Communication **asynchrone** :

- ❑ Les deux composants n'ont pas besoin d'être connectés en même temps grâce au système de file d'attente
- ❑ Emission non bloquante: l'entité émettrice émet son message, et continue son traitement sans attendre que le récepteur confirme l'arrivée du message
- ❑ Le récepteur récupère les messages **quand il le souhaite**

# Modèles d'interaction dans un système distribué

- Adapté à un système réparti dont les éléments en interaction sont faiblement couplés :
  - ☐ éloignement géographique des entités communicantes
  - ☐ possibilité de déconnexion temporaire d'un élément
- Deux modèles de communication :
  - ☐ Point à point: le message n'est lu que par un seul consommateur. Une fois lu, il est retiré de la file d'attente
  - ☐ Multi-points : le message est diffusé à tous les éléments d'une liste de destinataires

# Modèles d'interaction dans un système distribué

## ❖ Mémoire partagée

- Les éléments communiquent via une mémoire partagée à l'aide d'une interface d'accès à la mémoire
  - ❑ Ajout d'une donnée à la mémoire
  - ❑ Lecture d'une donnée à la mémoire
  - ❑ Retrait d'une donnée de la mémoire
- Particularité du modèle
  - ❑ Aucun lien, aucune interaction directe entre les participants

# Modèles d'interaction dans un système distribué

## ❖ Mémoire partagée

- Complexité du modèle : dans la gestion de la mémoire
- Comment gérer la mémoire dans le contexte d'un système distribué ?
- Plusieurs solutions
- Déployer toute la mémoire sur un seul site
  - Accès simple mais un goulot potentiel d'étranglement
- Éclater la mémoire sur plusieurs sites
  - Avec ou sans duplication des données
  - Il faut des algorithmes +/- complexes de gestion de mémoire partagée

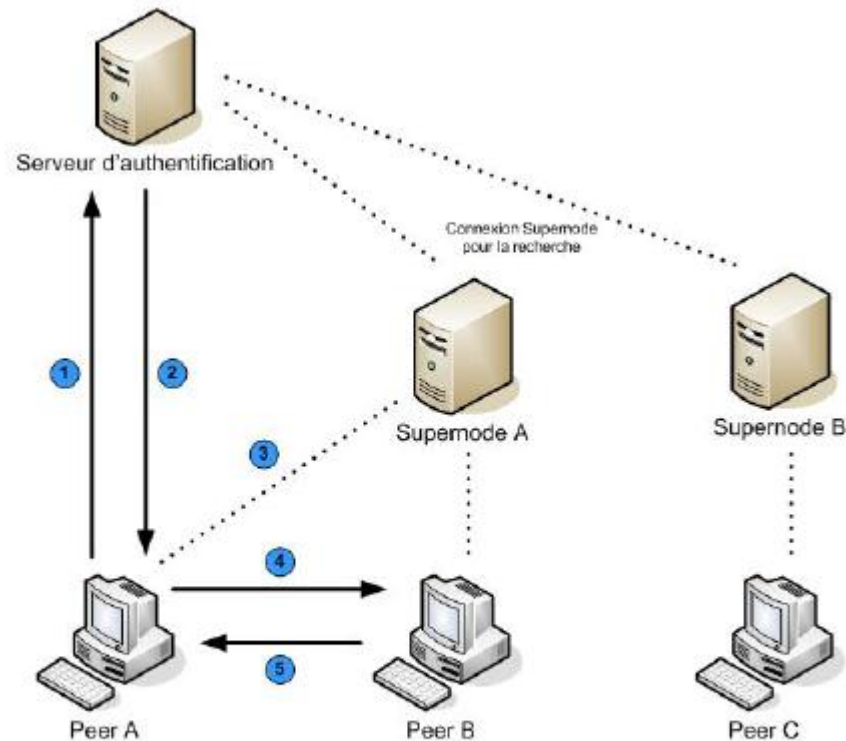


# Modèles d'interaction dans un système distribué

- ❖ Modèle pair à pair (peer to peer)
  - Un seul rôle : pas de distinction entre les participants
    - Chaque participant est connecté avec tous les participants d'un groupe et tout le monde effectue les mêmes types d'actions
  - Pour partager des données, effectuer des calculs,....
  - Exemples
  - Modèles d'échanges des fichiers (bit-torrent)
    - Avec parfois un modèle hybride client/serveur-P2P
    - Serveur sert à connaître la liste des fichiers et effectuer des recherches
    - Le mode P2P est utilisé ensuite pour les transferts
  - Algorithmes de consensus
    - Chacun mesure une valeur ( la même théorie), l'envoie aux autres,
    - Localement chacun exécute le même algorithme pour élire la bonne valeur.

# Modèles d'interaction dans un système distribué

- Modèle pair à pair hybride



1. Authentification auprès du serveur
2. Serveur renvoie le *Supernode* le plus près
3. Effectue la recherche du fichier sur le *Supernode*
4. Contacte le peer possédant le fichier
5. Transfère le fichier

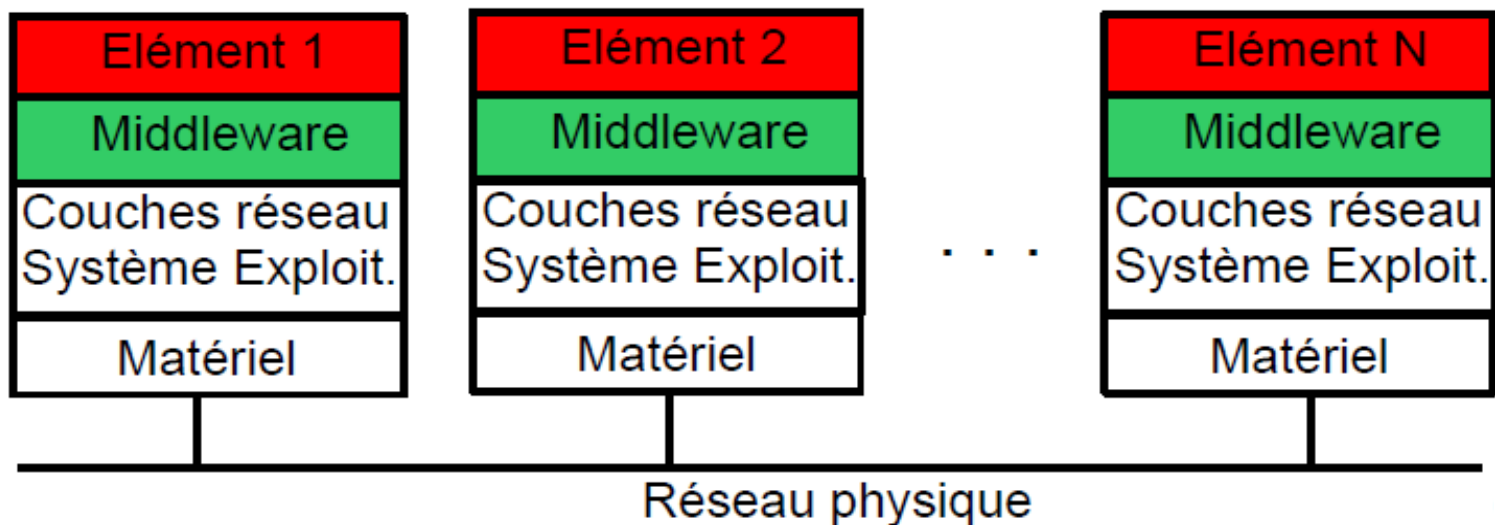


# Notion de middleware dans un système réparti

- ◆ Système distribué
  - ◆ Ensemble d'entités logicielles communiquant entre-elles
- ◆ Entités logicielles s'exécutent sur des machines reliées entre elles par un réseau
- ◆ Communication entre entités logicielles
  - ◆ Le plus basique : directement en appelant les services des couches TCP ou UDP
  - ◆ Plus haut niveau : définition de couches offrant des services plus complexes
    - ◆ Couche réalisée en s'appuyant sur les couches TCP/UDP
    - ◆ Exemple de service : appel d'une procédure chez une entité distante
    - ◆ Notion de middleware (intergiciel)

# Notion de middleware dans un système réparti

- ◆ Middleware ou intergiciel : couche logiciel
  - ◆ S'intercale entre le système d'exploitation/réseau et les éléments de l'application distribuée
  - ◆ Offre un ou plusieurs services de communication entre les éléments formant l'application ou le système distribué



# Notion de middleware dans un système réparti

- ◆ But et fonctionnalités d'un middleware
  - ◆ Gestion de l'hétérogénéité
    - ◆ Langage de programmation, systèmes d'exploitation utilisés ...
  - ◆ Offrir des abstractions de communication de plus haut niveau
    - ◆ Appel d'une procédure à distance sur un élément
    - ◆ Communication via une mémoire partagée
    - ◆ Diffusion d'événements
    - ◆ ...
  - ◆ Offrir des services de configuration et de gestion du système
    - ◆ Service d'annuaire pour connaître les éléments présents
    - ◆ Services de persistance, de temps, de transaction, de sécurité ...

# Notion de middleware dans un système réparti

- ◆ Protocole de communication
  - ◆ Ensemble de règles et de contraintes gérant une communication entre plusieurs entités
- ◆ But du protocole
  - ◆ Se mettre d'accord sur la façon de communiquer pour bien se comprendre
  - ◆ S'assurer que les données envoyées sont bien reçues
- ◆ Plusieurs types d'informations circulent entre les entités
  - ◆ Les données à échanger
  - ◆ Les données de contrôle et de gestion du protocole

# Notion de middleware dans un système réparti

## ❖ Middleware : Fonctions

- Masquer l'hétérogénéité (machines, systèmes, protocoles de communication)
- Fournir une API (Application Programming Interface) de haut niveau
  - Permet de masquer la complexité des échanges
  - Facilite le développement d'une application répartie
- Rendre la répartition aussi invisible (transparente) que possible
- Fournir des services répartis d'usage courant

# Notion de middleware dans un système réparti

## ❖ Services du middleware

### • Communication

- permet la communication entre machines mettant en œuvre des formats différents de données

### • Prise en charge par la FAP (Format And Protocol)

### • FAP : pilote les échanges à travers le réseau :

- synchronisation des échanges selon un protocole de communication
- mise en forme des données échangées selon un format connu de part et d'autre

### • Nommage

- permet d'identifier la machine serveur sur laquelle est localisé le service demandé afin d'en déduire le chemin d'accès.
- fait, souvent, appel aux services d'un annuaire.

### • Sécurité

- permet de garantir la confidentialité et la sécurité des données à l'aide de mécanismes d'authentification et de cryptage des informations

# Notion de middleware dans un système réparti

## ❖ Types de middleware

- Middlewares orientés objets distribués
  - Java RMI, Corba
- Middlewares orientés composants distribués
  - EJB, Corba, DCOM
- Middlewares orientés messages
  - JMS de Sun, MSMQ de Microsoft, MQSeries de IBM
- Middlewares orientés services
  - Web Services (XML-RPC, SOAP)
- Middlewares orientés SGBD
  - ODBC (Open DataBase Connectivity), JDBC de Sun



# Communication dans un système réparti

- ◆ Système distribué
  - ◆ Ensemble d'entités logicielles communiquant entre-elles
- ◆ Entités logicielles s'exécutent sur des machines reliées entre elles par un réseau
- ◆ Communication entre entités logicielles
  - ◆ Le plus basique : directement en appelant les services des couches TCP ou UDP
  - ◆ Plus haut niveau : définition de couches offrant des services plus complexes
    - ◆ Couche réalisée en s'appuyant sur les couches TCP/UDP
    - ◆ Exemple de service : appel d'une procédure chez une entité distante
    - ◆ Notion de middleware (intergiciel)



# Introduction aux sockets

## Définitions

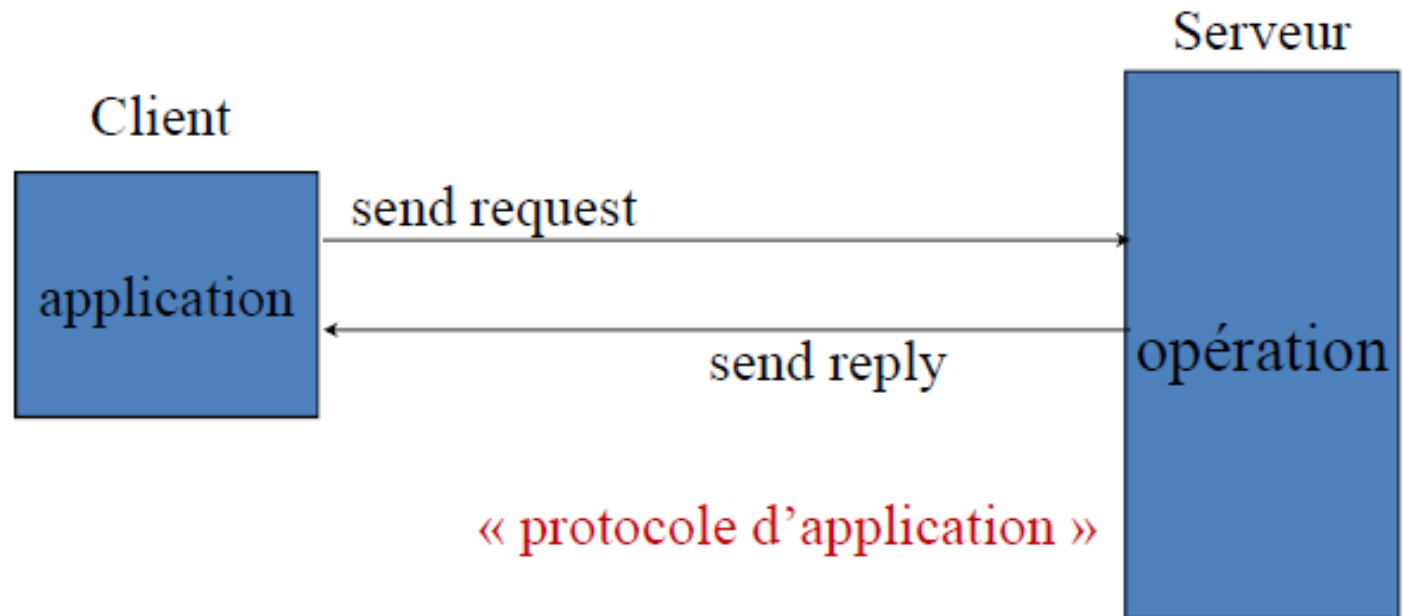
- Un socket est une interface entre les programmes d'applications et les couches dédiées à la gestion du réseau
- Le terme “ socket ” désigne à la fois une bibliothèque d'interface réseau et l'extrémité d'un canal de communication (point de communication) par lequel un processus peut émettre ou recevoir des données
- L'interface socket est un ensemble de primitives qui permettent de gérer l'échange de données entre des processus, que ces processus soient exécutés ou non sur la même machine (processus exécutés sur la même machine en local ou sur le réseau)
- La bibliothèque socket masque l'interface et les mécanismes de la couche transport : un appel socket se traduit par plusieurs requêtes transport

# Introduction aux sockets

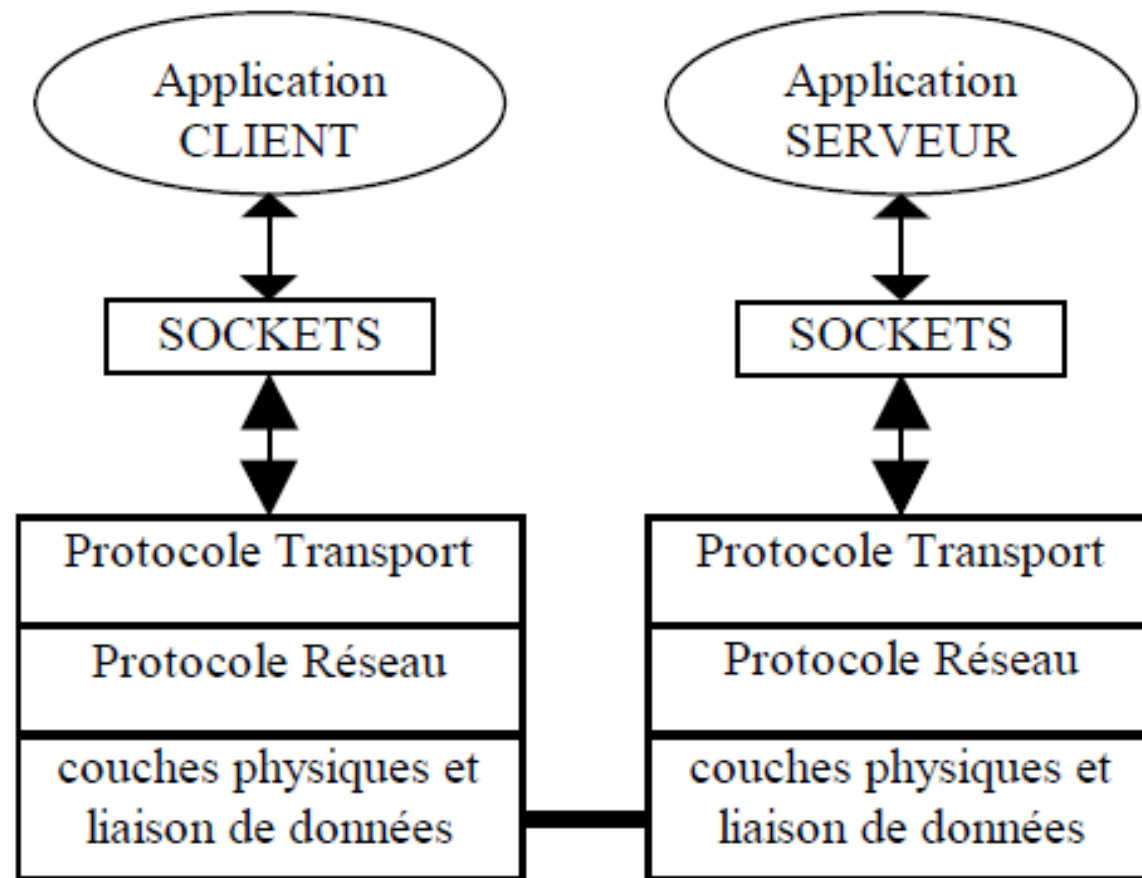
- Un socket est un port de communication ouvert au niveau des couches réseaux et permettant de faire passer des flux
- La communication est en point à point en mode client/serveur
- La communication est bidirectionnelle
- La connexion se fait en mode connecté (TCP) ou non connecté (UDP)
- Il existe deux protocoles utilisant les sockets en Java :
  - TCP : Stream Socket
    - Communication en mode connecté
    - Les applications sont averties lors de la déconnexion
  - UDP : Datagram Socket
    - Communication en mode non connecté
    - Plus rapide mais moins fiable que TCP

# Modèle client/serveur

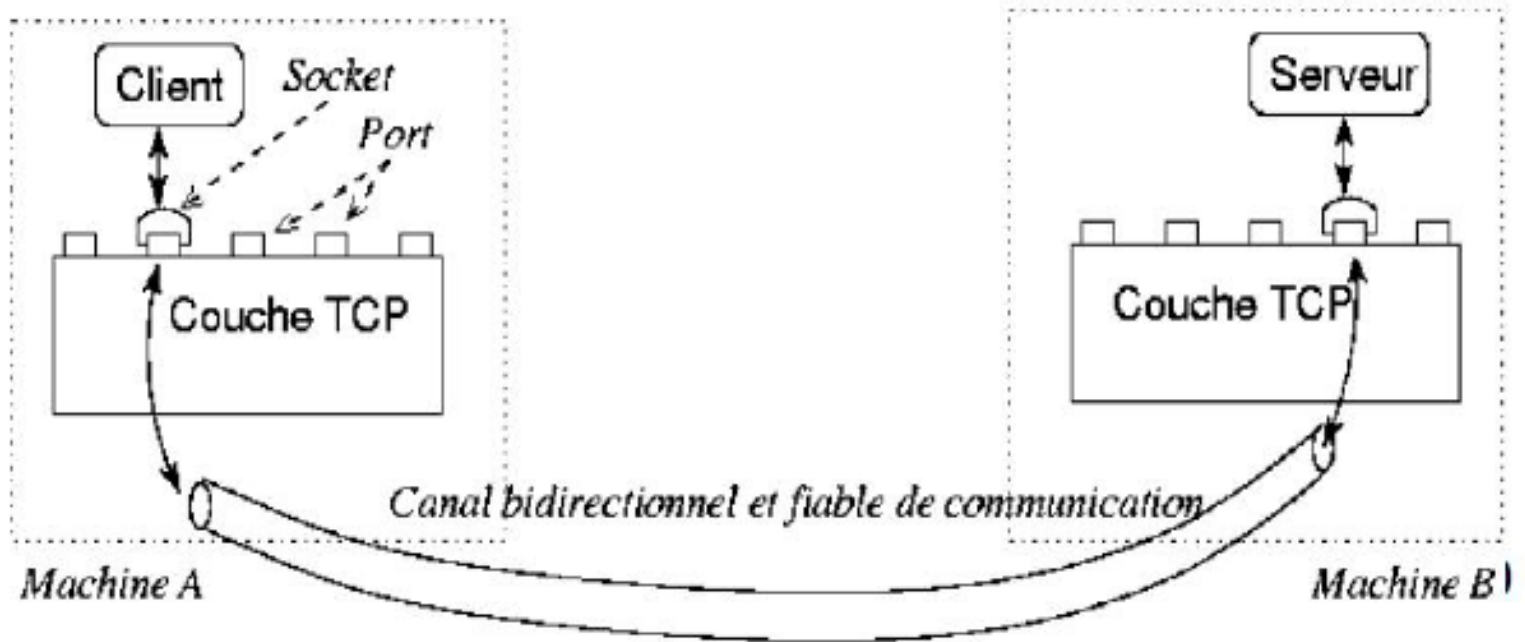
Mode de communication qu'un hôte établit avec un autre hôte qui fournit un service quelconque



# Sockets: modèle client/serveur



# Sockets en mode connecté



# Sockets en mode connecté

- Client

1. Crée un socket
2. Se connecte au serveur en donnant l'adresse socket distante (adresse IP du serveur et numéro de port du service). Le système d'exploitation attribue pour cette connexion automatiquement un numéro de port local au client
3. Lit et/ou écrit (envoi et réception des messages) sur le socket
4. Ferme le socket

- Serveur (en écoute du canal)

1. Crée un socket
2. Associe une adresse socket (adresse Internet et numéro de port) au service : "binding"
3. Se met à l'écoute des connexions entrantes ;
4. Pour chaque connexion entrante:
  1. Accepte la connexion (un nouveau socket est créé avec les mêmes caractéristiques que le socket d'origine, ce qui permet de lancer un thread ou un processus fils pour gérer cette connexion).
  2. Lit et/ou écrit sur le nouveau socket
5. Ferme le socket créé

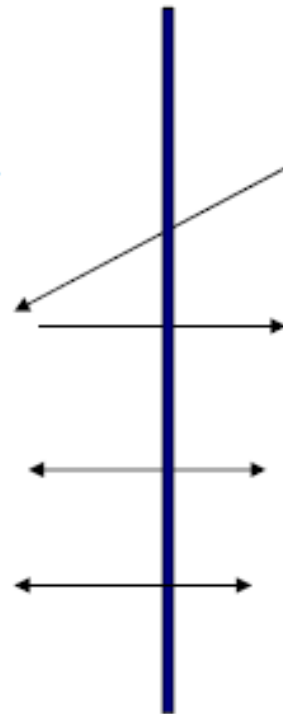
# Sockets en mode connecté

- Le serveur :

- Crée un socket
- Attend une connexion
- accepte la demande
- échange de données
- Fermer connexion

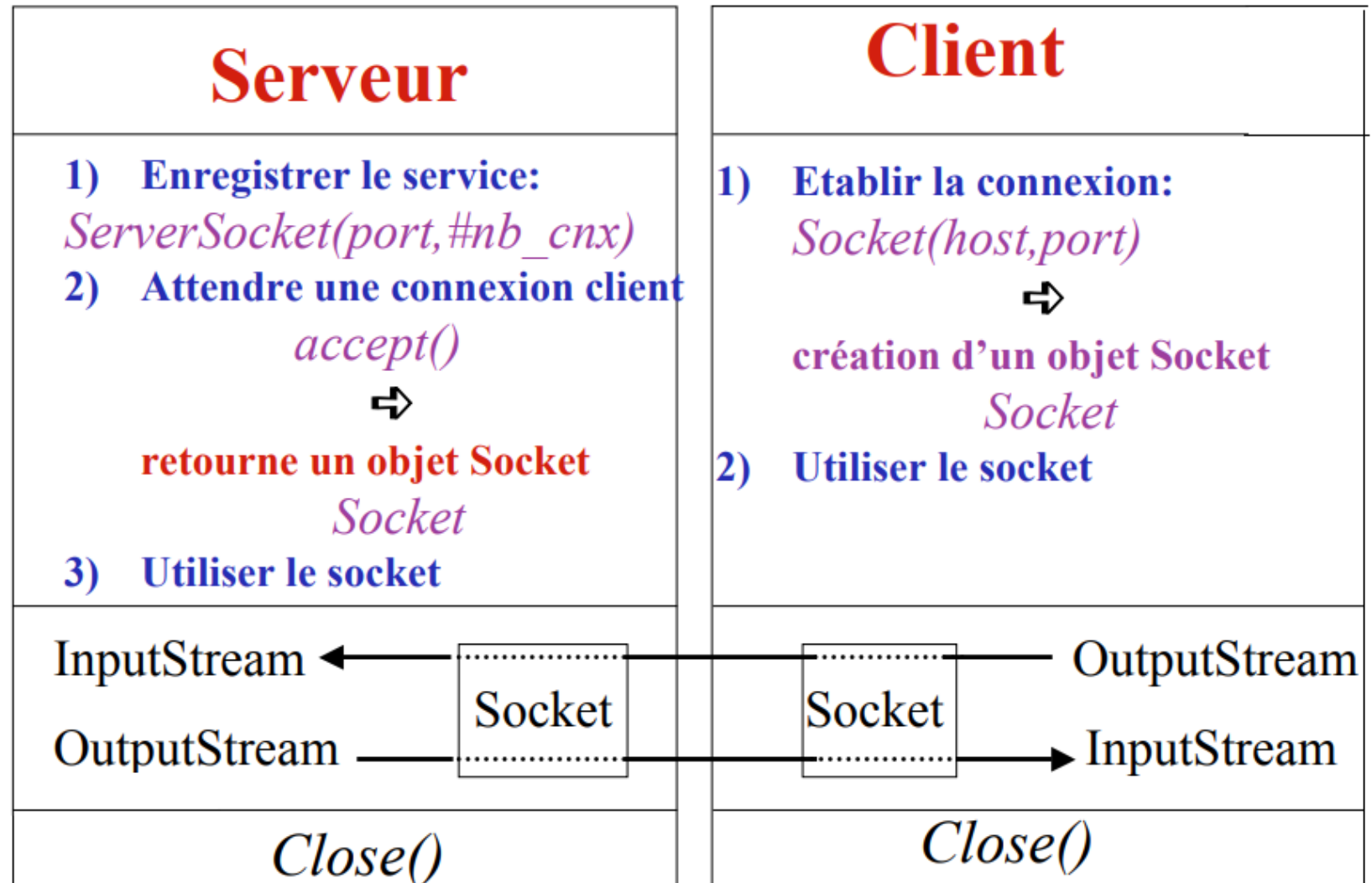
- Le client :

- demande une connexion
- création de socket
- échange de données
- Fermer connexion





# Le modèle client /serveur java





# Principe de fonctionnement (I)

- **Serveur: enregistrer le service**
  - le serveur enregistre son service sous un numéro de port, indiquant le nombre de clients qu'il accepte de faire buffériser à un instant T (`new serverSocket(...)`)
- **Serveur : attente de connexion**
  - il se met en attente d'une connexion (méthode `accept()` de son instance de `ServerSocket`)

# Principe de fonctionnement (2)

- **Client : établir la connexion**
  - le client peut alors établir une connexion en demandant la création d'un socket (`new Socket()`) à destination du serveur pour le port sur lequel le service a été enregistré.
- **Serveur**
  - le serveur sort de son `accept()` et récupère un Socket de communication avec le client
- **Les deux : utilisation du socket**
  - le client et le serveur peuvent alors utiliser des `InputStream` et `OutputStream` pour échanger les données

# Un serveur TCP/IP (I)

- il utilise la classe `java.net.ServerSocket` pour accepter des connexions de clients
- quand un client se connecte à un port sur lequel un `ServerSocket` écoute, `ServerSocket` crée une nouvelle instance de la classe `Socket` pour supporter les communications côté serveur :

```
int port = ...;
```

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

```
Socket connection = server.accept();
```

# Un serveur TCP/IP (2)

- les constructeurs et la plupart des méthodes peuvent générer une **IOException**
- la méthode **accept()** est dite bloquante ce qui implique un type de programmation particulier : boucle infinie qui se termine seulement si une erreur grave se produit

# java.net.ServerSocket

```
final int PORT = ...;
```

```
try {  
    ServerSocket serveur = new ServerSocket(PORT,5);  
    while (true) {  
        Socket socket = serveur.accept();  
    }  
}  
catch (IOException e){  
    ....  
}
```

# Un client TCP/IP

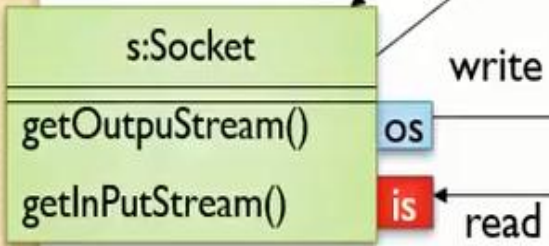
- le client se connecte au serveur en créant une instance de la classe `java.net.Socket` : connexion synchrone  
`String host = ...;`  
`int port = ...;`  
`Socket connection = new Socket (host,port);`
- le socket permet de supporter les communications côté client
- la méthode `close()` ferme (détruit) le socket
- les constructeurs et la plupart des méthodes peuvent générer une `IOException`
- le serveur doit être démarré avant le client. Dans le cas contraire, si le client se connecte à un serveur inexistant, une exception sera levée après un *time-out*

Principe de base d'une application client/serveur avec socket

Création d'un client :

```
Socket s=new Socket("192.168.1.23",1234)
```

```
InputStream is=s.getInputStream();
OutputStream os=s.getOutputStream();
os.write(23);
int rep=is.read();
System.out.println(rep);
```

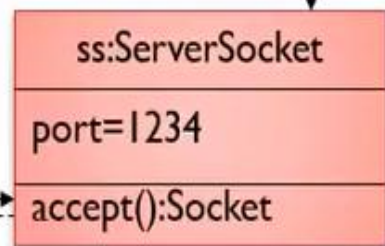


Création d'un serveur:

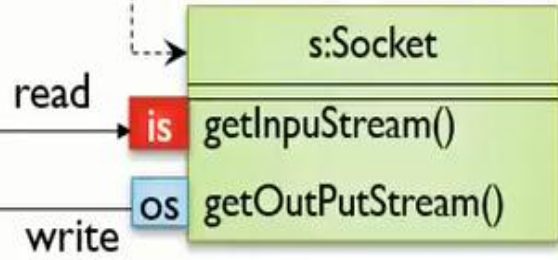
```
ServerSocket ss=new ServerSocket(1234);
```

```
Socket s=ss.accept();
```

```
InputStream is=s.getInputStream();
OutputStream os=s.getOutputStream();
int nb=is.read();
int rep=nb*2;
os.write(rep);
```



Connexion



# Flux d'échange, d'E/S :échange d'octets

- Etant donné un socket *sck*

```
InputStream is = sck.getInputStream();  
int nb = is.read();
```

```
OutputStream os= sck.getOutputStream();  
int res = nb*3;  
os.write(res);
```



# Flux d'échange, d'E/S: échange de chaînes de caractères

- Etant donné un socket sck

```
InputStream is = sck.getInputStream();  
InputStreamReader isr = new InputStreamReader(is);  
BufferedReader br = new BufferedReader(isr);  
String str=br.readLine();
```

```
OutputStream os= sck.getOutputStream();  
PrintWriter pw = new PrintWriter(os,true);  
String rep="Bonjour Mr"+str;  
pw.println(rep); //os.write(res);
```

# Flux d'échange, d'E/S: échange d'objets

- L'échange d'un objet nécessite sa sérialisation / désérialisation
  - sérialisation : transformation d'une représentation mémoire d'un objet en une suite / un tableau / d'octets à envoyer via le réseau
  - Désérialisation = processus inverse
- ➔ un objet à échanger via le réseau doit être sérialisable
  - Implémente l'interface Serializable
- Etant donné un socket *sck* et la classe sérialisable *Compte*

```
OutputStream os= sck.getOutputStream();  
ObjectOutputStream oos = new ObjectOutputStream(os);  
Compte cpt = new Compte("1200", 1000);  
oos.writeObject(cpt);
```

```
InputStream is = sck.getInputStream();  
ObjectInputStream ois = new ObjectInputStream(is);  
Compte cptRecu = (Compte) ois.readObject();
```

# Etapes de base pour communication via des sockets TCP

## Client

- 3 Demande de connexion sur la machine d'adresse **mi** et le port **mp** avec le socket d'échange de flux **sck**
- 5 création des flux pour échanger des octets, de texte, ou d'objets via **sck**
- 6 échange via les flux créés
- 7 fermer le socket **sck**

## Serveur

- 1 Créer un socket d'écoute **se** sur la machine d'adresse IP **mi** et le port **mp**
- 2 se mettre en attente / écoute de demande de connexions sur **se**
- 4 réception d'une demande de connexion  
→ création de socket d'échange **sckServ**
- 5 création des flux pour échanger des octets, de texte, ou d'objets via **sckServ**
- 6 échange via les flux créés
- 7 fermer le socket **sckServ**
- 8 fermer le socket **se**

# Exemple 1 de communication via des sockets – Serveur mono-client

```
import java.net.*;
import java.io.*;
public class Serveur1 {
    public static void main(String[] args){
        try {
            ServerSocket ss = new ServerSocket(1234);
            System.out.println("J'attends une connexion");
            Socket s = ss.accept();
            InputStream is=s.getInputStream();
            OutputStream os=s.getOutputStream();
            System.out.println("J'attends un nombre");
            int nb=is.read();
            int res=nb*2;
            System.out.println("J'envoies la réponse");
            os.write(res);
            s.close();
        } catch (Exception e){ e.printStackTrace();
        }
    }
}
```

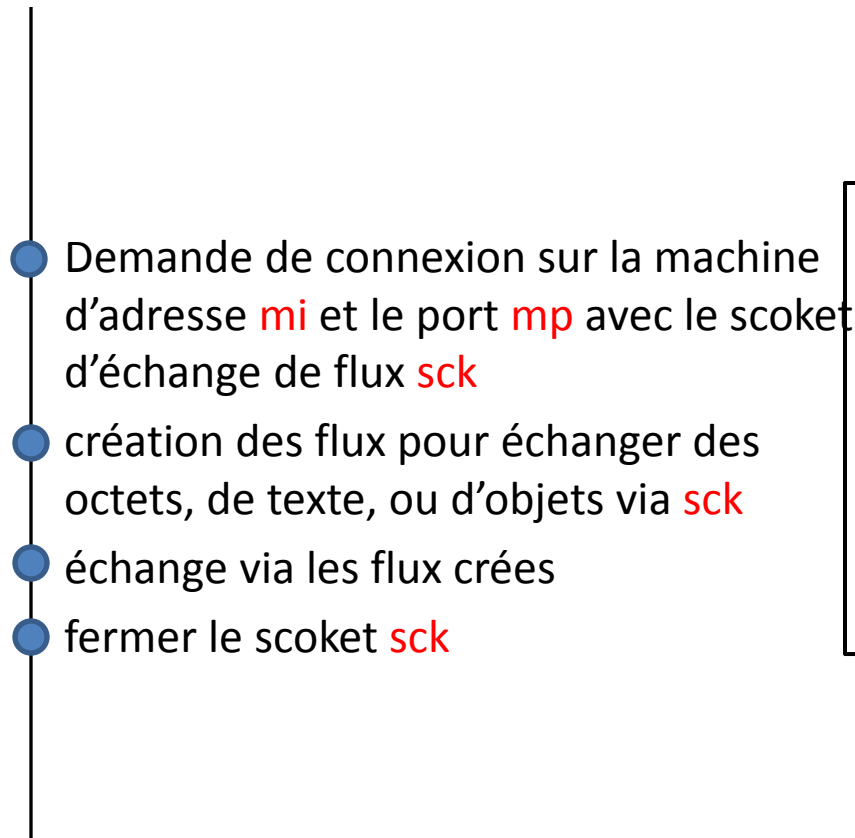
# Exemple 1 de communication via des sockets -

## Client

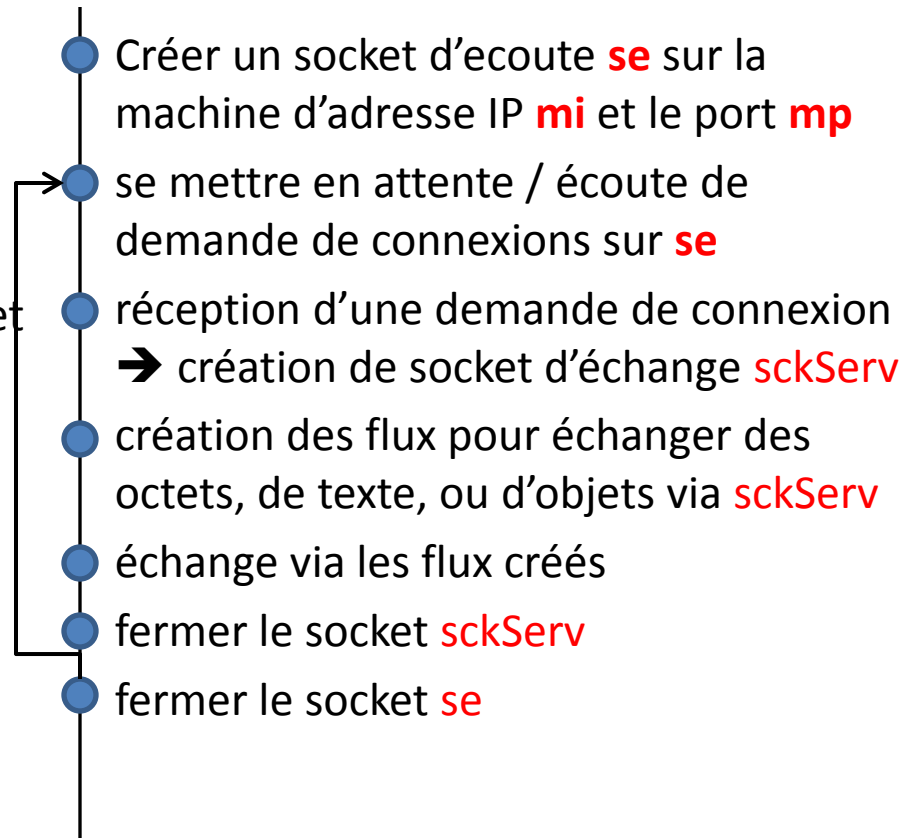
```
import java.net.*; import java.io.*;
import java.util.Scanner;
public class Client1 {
    public static void main(String[] args){
        try {
            Socket s = new Socket("localhost", 1234);
            InputStream is=s.getInputStream();
            OutputStream os=s.getOutputStream();
            Scanner clavier = new Scanner(System.in);
            System.out.println("Donner un nombre");
            int nb=clavier.nextInt();
            os.write(nb);
            int res = is.read();
            System.out.println("Res =" +res);
            os.write(res);
            s.close();
        } catch (Exception e){ e.printStackTrace(); } } }
```

# Etapes de base pour communication via des sockets TCP – multi-clients itératif

## Client



## Serveur



## Exemple 2 de communication via des sockets – serveur multi clients itératifs

```
import java.io.*;

public class Serveur3 {
    public static void main(String[] args){
        try { ServerSocket ss = new ServerSocket(123);
            while(true){
                System.out.println("J'attends une connexion");
                Socket s = ss.accept();
                InputStream is=s.getInputStream();
                InputStreamReader isr = new InputStreamReader(is);
                BufferedReader br = new BufferedReader(isr);
                OutputStream os=s.getOutputStream();
                PrintWriter pw = new PrintWriter(os,true);
                System.out.println("J'attends une chaine");
                String str=br.readLine();
                String rep="Bonjour Mr"+str;
                System.out.println("J'envoies la réponse");
                pw.println(rep);
                s.close();
            } } catch (Exception e){ e.printStackTrace(); } } }
```

# Serveur Multi-Threads

- Pour qu'un serveur puisse communiquer avec plusieurs client en même temps, il faut que:
  - Le serveur puisse attendre une connexion à tout moment.
  - Pour chaque connexion, il faut créer un nouveau thread associé à la socket du client connecté, puis attendre à nouveau une nouvelle connexion
  - Le thread créé doit s'occuper des opérations d'entrées-sorties (read/write) pour communiquer avec le client indépendamment des autres activités du serveur.

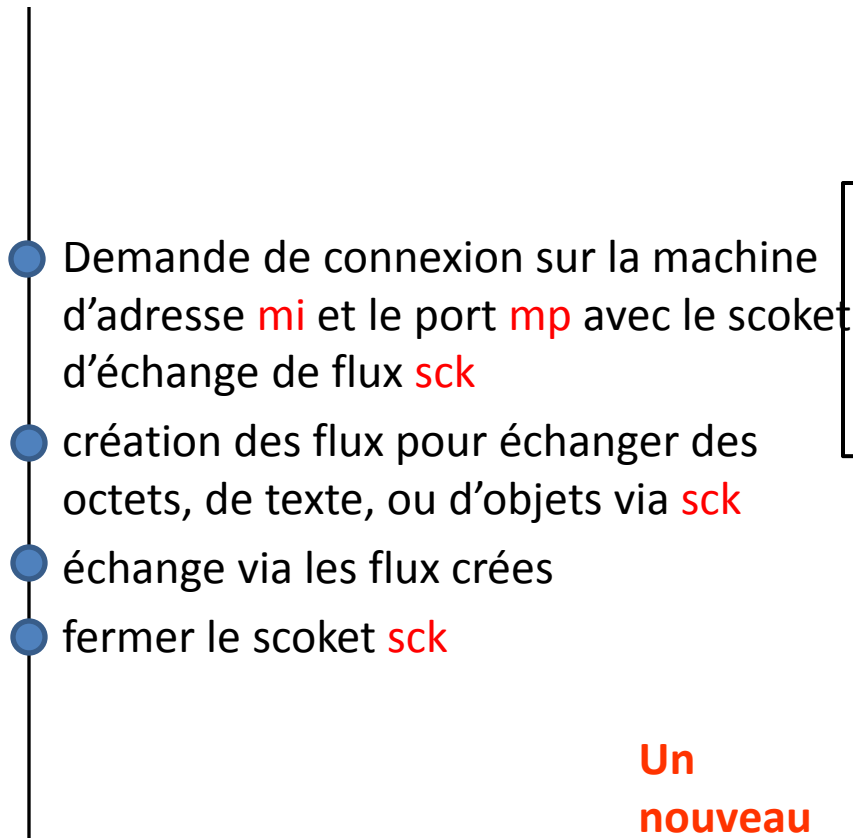


# Architecture du serveur concurrent :

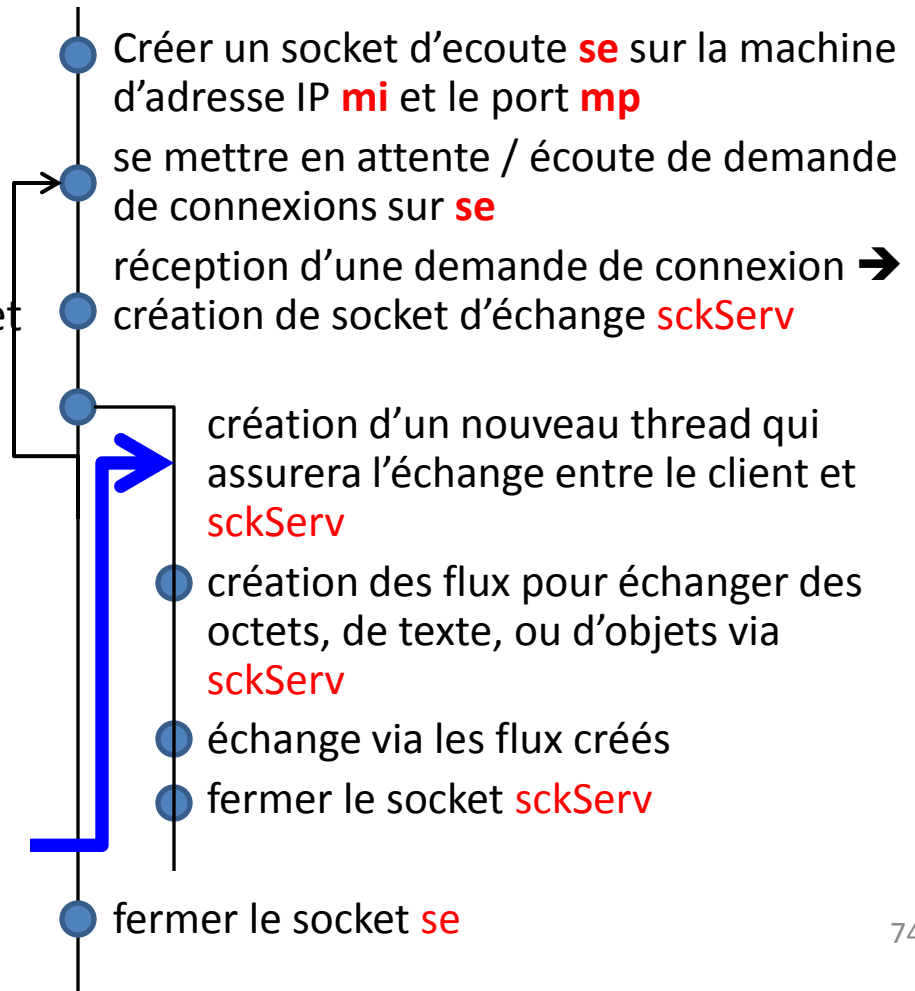
- Il y a un flot d'exécution principal (appelé veilleur) qui attend sur `accept()`.
- Lorsqu'il reçoit une demande de connexion, le veilleur crée un nouveau flot (appelé exécutant) qui va interagir avec le nouveau client.
- Après la création de l'exécutant, le veilleur revient se mettre en attente sur `accept()`.
- Plusieurs exécutants peuvent co-exister simultanément.

# Etapes de base pour communication via des sockets TCP – multi-clients concurrents

## Client



## Serveur – thread initial



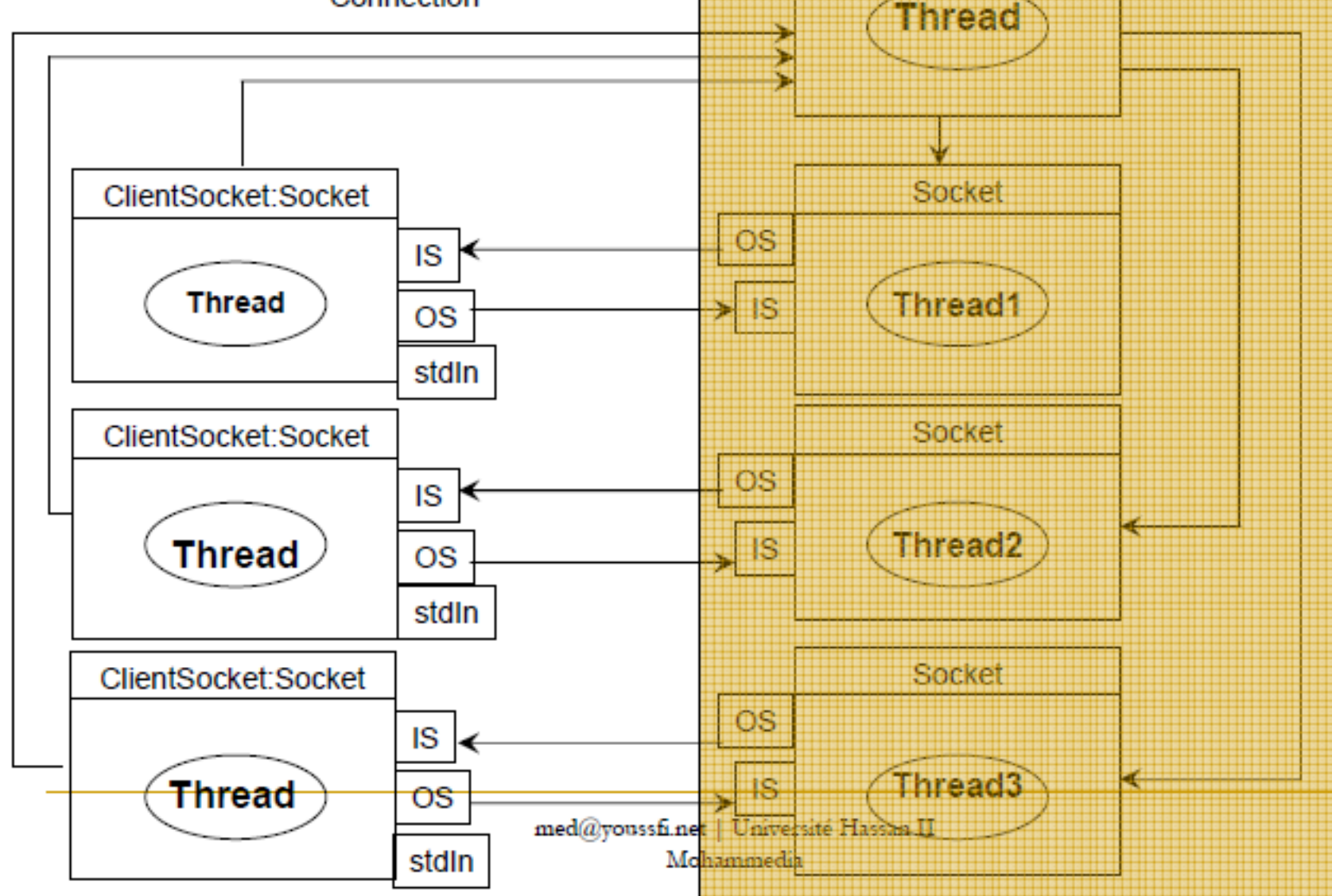
Un  
nouveau  
thread  
par client

# Connections Multithread

Serveur

Client

Connection



# L'adressage IP en Java

- L'adresse IP est fournie par le biais d'objets de la classe **InetAddress** dont on retiendra les méthodes suivantes :
  - **getAddress()** : retourne l'adresse IP sous forme de tableau de "byte".
  - **getLocalhost()** : retourne un objet **InetAddress** contenant l'adresse IP de l'hôte (pas 127.0.0.1).
  - **getHostName()** : retourne une chaîne de caractère : l'adresse symbolique de l'objet.
  - **InetAddress.getByName(chaîne)** : permet de récupérer l'adresse IP sous forme d'objet **InetAddress** correspondant au nom symbolique donné en paramètre.
  - La méthode **.toString** permet d'afficher un objet **InetAddress** sous la forme nom symbolique/adresse IP.

# Sockets UDP

- Le protocole UDP offre un service non connecté et non fiable.
- Les données peuvent être perdues, dupliquées ou délivrées dans un ordre différent de leur ordre d'émission.
- Le fonctionnement est ici symétrique. Chaque processus crée un socket et l'utilise pour envoyer ou attendre et recevoir des données.
- En Java on utilise la classe DatagramPacket pour représenter les datagrammes UDP qui sont échangés entre les machines.

# Etapes de base pour communication via des sockets UDP

## processus 1 sur machine d'@

IP **aip**

- Création du DatagramPacket **dgPRecu** dans lequel on recevra le Datagram envoyé
- Création d'un DatagramSocket **dgSck** écoutant des requêtes sur le port numéro **np**
- 4 Réception du DatagramPacket envoyé dans dgPRecu

## processus 2

- 1 Créer un DatagramPacket **dgP** qui contient : Les données à envoyer, l'adresse IP **aip** de la machine cible, et le numéro du port **np** de la socket vers laquelle ces données seront envoyées.
- 2 Création d'un DatagramSocket **dgS**
- 3 Envoie de **dgP** via **dgS**

# Sockets UDP

- **Quelles classes utiliser ?**
- Il faut utiliser les classes **DatagramPacket** et **DatagramSocket**
- Ces objets sont initialisés différemment selon qu'ils sont utilisés pour *envoyer* ou *recevoir* des paquets

# Sockets UDP

- **Envoi d'un Datagramme**
- **1.** créer un `DatagramPacket` en spécifiant :
  - les données à envoyer
  - leur longueur
  - la machine réceptrice et le port
- **2.** utiliser la méthode `send(DatagramPacket)` de `DatagramSocket`
  - pas d'arguments pour le constructeur car toutes les informations se trouvent dans le paquet envoyé



# Sockets UDP

- **Envoi d'un Datagramme**

//Machine destinataire

```
InetAddress address = InetAddress.getByName(" rainbow.essi.fr ");
```

```
static final int PORT = 4562;
```

//Création du message à envoyer

```
String s = new String (" Message à envoyer");
```

```
int longueur = s.length();
```

```
byte[] message = new byte[longueur];
```

```
s.getBytes(0,longueur,message,0);
```

//Initialisation du paquet avec toutes les informations

```
DatagramPacket paquet = new DatagramPacket(message,longueur,  
address,PORT);
```

//Création du socket et envoi du paquet

```
DatagramSocket socket = new DatagramSocket();
```

```
socket.send(paquet);....
```

# Sockets UDP

- **Réception d'un Datagramme**

1. créer un `DatagramSocket` qui écoute sur le port de la machine du destinataire

2. créer un `DatagramPacket` pour recevoir les paquets envoyés par le serveur

- dimensionner le buffer assez grand

3. utiliser la méthode `receive()` de `DatagramPacket`

- cette méthode est bloquante

# Sockets UDP

- **Réception d'un Datagramme**

//Définir un buffer de réception

```
byte[] buffer = new byte[1024];
```

//On associe un paquet à un buffer vide pour la réception

```
DatagramPacket paquet = new DatagramPacket(buffer,buffer.length());
```

//On crée un socket pour écouter sur le port

```
DatagramSocket socket = new DatagramSocket(PORT);
```

```
while (true) {
```

//attente de réception

```
socket.receive(paquet);
```

//affichage du paquet reçu

```
String s = new String(buffer,0,0,paquet.getLength());
```

```
System.out.println("Paquet reçu : + s);
```

```
}
```

# Exemple

## Créer un DatagramPacket

```
String message = "Bonjour le monde!" ;
```

```
byte[] tampon = message.getBytes();
```

```
InetAddress adresse = null;
```

```
// recupère l'adresse IP de la machine
```

```
adresse = InetAddress.getByName("toto.isimm.rnu.tn");
```

```
// crée l'objet qui stockera les données du datagramme à envoyer
```

```
DatagramPacket dgp= new
```

```
    DatagramPacket(tampon,tampon.length,adresse,50000);
```

# Création d'un DatagramSocket et envoie de dgp

```
// crée un socket UDP (le port est choisi par le système)
DatagramSocket    socket=new DatagramSocket();
// envoie le datagramme UDP
socket.send(dgp);
```

## Création du DatagramPacket dgPReçu

```
byte[] tampon = new byte[1000];
// crée un objet pour stocker les données du datagramme attendu
DatagramPacket dgPReçu = new DatagramPacket(tampon, tampon.length);
// attends puis récupère les données du datagramme
```

# Réception du DatagramPacket envoyé dans dgPRECU

```
// crée un socket UDP qui attends des datagrammes sur le port 50000
```

```
DatagramSocket dgSck = new DatagramSocket(50000);
```

```
// attends puis récupère les données du datagramme
```

```
dgSck.receive(dgpRecu);
```

```
// récupère la chaîne de caractère reçue
```

```
// Note: dgpRecu.getLength() contient le nombre d'octets reçus
```

```
String texte=new String(tampon, 0, dgpRecu.getLength());
```

```
System.out.println("Reception de la machine "+  
dgpRecu.getAddress().getHostName()+ " sur le port "  
+dgpRecu.getPort()+ " :\n"+ texte );
```

# Critique sockets UDP

- Avantages
  - Simple à programmer (et à appréhender)
- Inconvénients
  - Pas fiable
  - Ne permet d'envoyer que des tableaux de byte
  - Format des données à transmettre
    - Très limité a priori : tableaux de byte
    - Et attention à la taille réservée : si le récepteur réserve un tableau trop petit par rapport à celui envoyé, une partie des données est perdue

# Implémentation de sockets en Java

- Paquetage java.net

