Programmation de la communication via les sockets

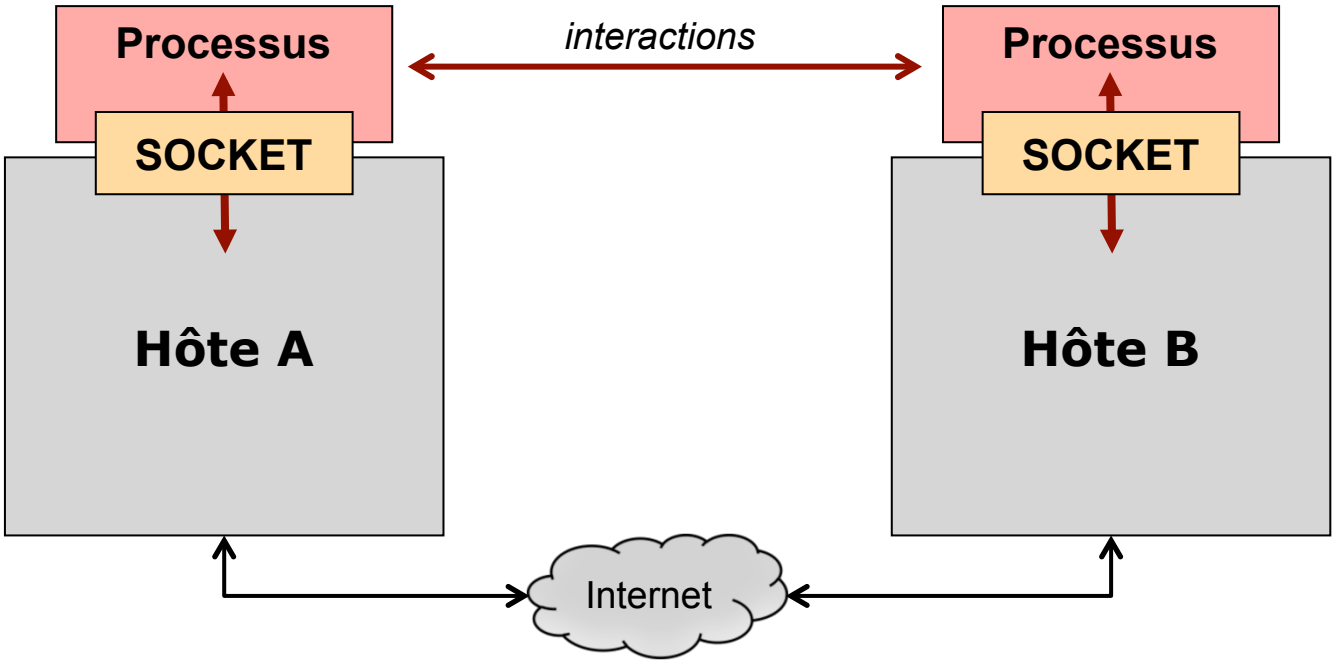
Thierry Desprats, Emmanuel Lavinal
(Prénom.Nom@irit.fr)

M1 MIAGE – UE « Réseaux »
v. 01/2017

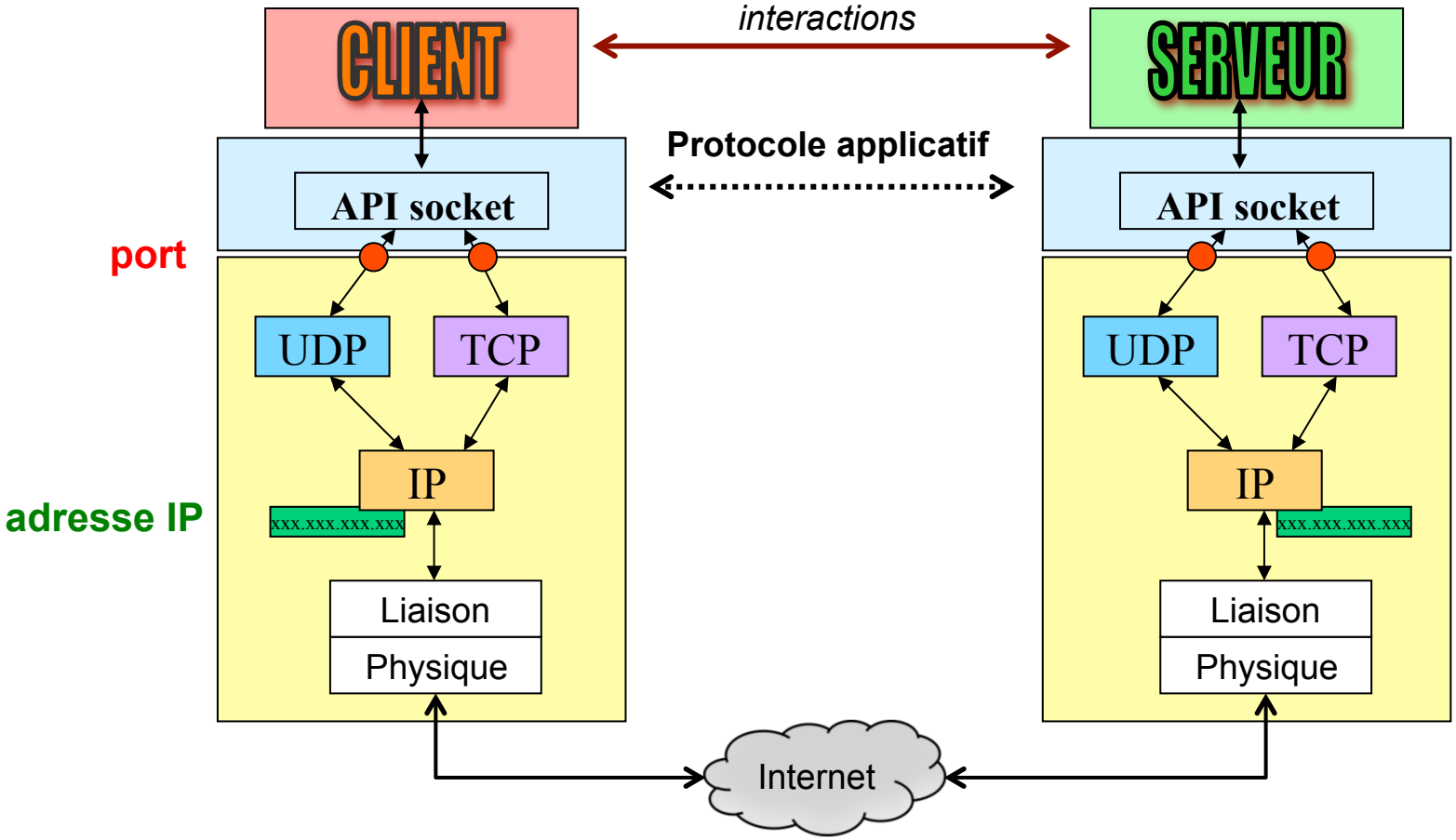
Introduction

- **Socket** :

  représente l'extrémité d'un canal de communication par lequel un processus d'application peut émettre ou recevoir des données.



Introduction



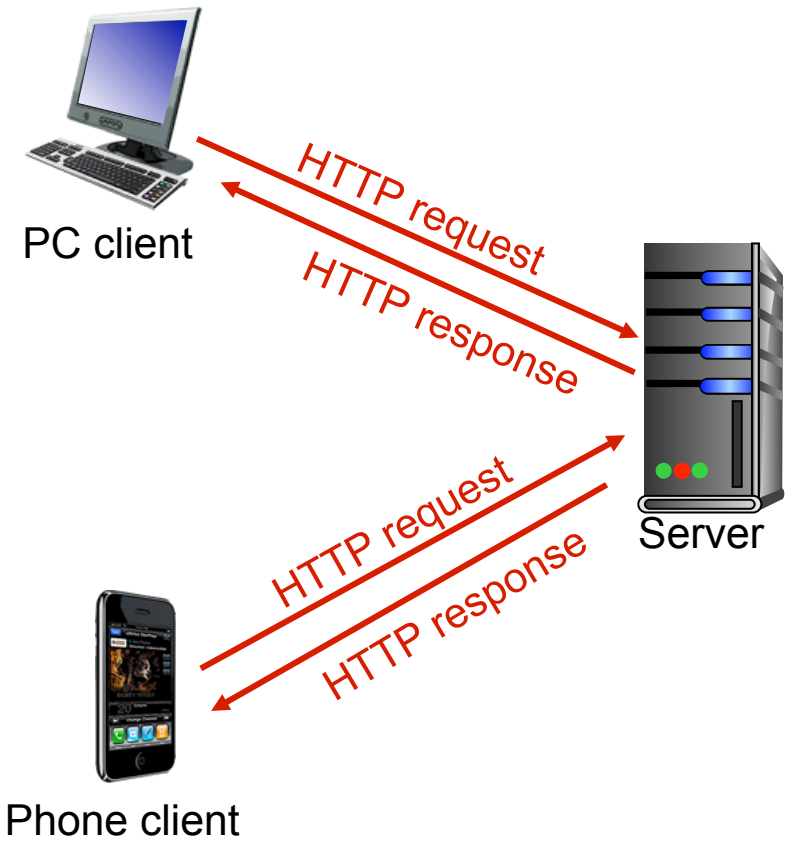
Protocoles applicatifs

- Les processus distants communiquent entre eux par l'envoi et la réception de messages dans des *sockets*.
  - Comment ces messages sont-ils structurés ? Quand sont-ils envoyés ? …

    ➔ rôle du **protocole de la couche application**
- Un **protocole applicatif (*application-layer protocol*)** définit :
  - Les types de messages échangés
  - La syntaxe des différents types de messages (~ les « champs » du msg)
  - La sémantique des champs du message (i.e. leur signification)
  - Les règles pour déterminer quand un processus envoie ou reçoit tel type de message
- Exemples de protocoles applicatifs standards (IETF) :
  - HTTP, SMTP, FTP, SIP, RTP…

Exemple : aperçu de HTTP

- HTTP (HyperText Transfer Protocol)
  - Protocole applicatif utilisé par le web…
  - Modèle Client / Serveur
  - S'appuie sur un service de transport fiable TCP (port serveur « well-known » : 80)
  - Sans état (*stateless*)



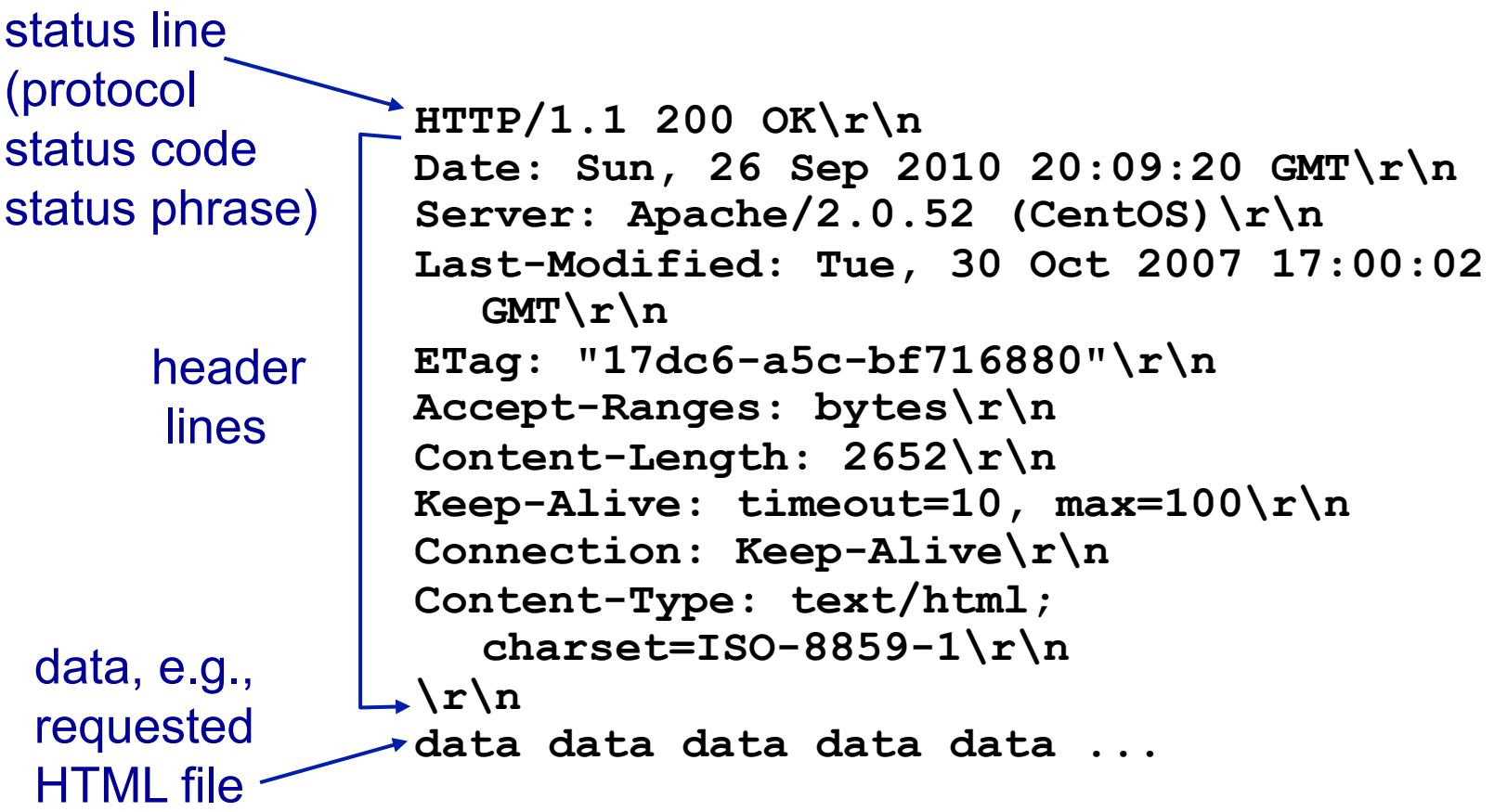
  - **Requête :** opérations `GET, HEAD, POST, PUT,…`
  - **Réponse :** code de retour, corps du message éventuel (données)
  - **Encodage :** texte ASCII + ressources dans structures MIME (`text/html, image/gif, image/jpeg,…`)

Exemple : aperçu de HTTP

- Format des messages HTTP
  - **Exemple Requête :**

Exemple : aperçu de HTTP

- Format des messages HTTP
  - **Exemple Réponse :**



Plan

- **Caractéristiques générales**
  - Définitions et historique
  - Types de sockets
  - Abstraction pour le programmeur
- **Sockets en mode datagramme**
  - Principe
  - Création et caractérisation
  - Transfert de données
  - Récapitulatif
- **Sockets en mode connecté**
  - Principe
  - Ouverture passive et ouverture active
  - Acceptation
  - Transfert de données
  - Fermeture
  - Récapitulatif
- **Gestion des options**

Caractéristiques générales (1/4)

- **Socket** :
  représente l'extrémité d'un canal de communication par lequel un processus d'application peut émettre ou recevoir des données.

  = = > *Abstraction des services de communication*

- Interface de Programmation :
  ensemble de primitives pour :
  - attribuer aux processus un rôle (par ex. client ou serveur)
  - manipuler des informations de communication
    permettre les mises en relation
  - échanger des messages protocolaires applicatifs en rendant transparents les services de com. de plus bas niveau (transport, réseau, liaison…)
    supporter l'interaction applicative
  - contrôler et paramétrer les sockets

Caractéristiques générales (2/4)

- Historique
  - Initialement
    - UNIX Berkeley
    - Langage C
    - IPC et Pile TCP/UDP-IP
  - Autres protocoles : AppleTalk, Novell IPX,…
  - Autres systèmes d'exploitation : MS-Windows,…
  - Autres langages : Java, Python, PhP, …

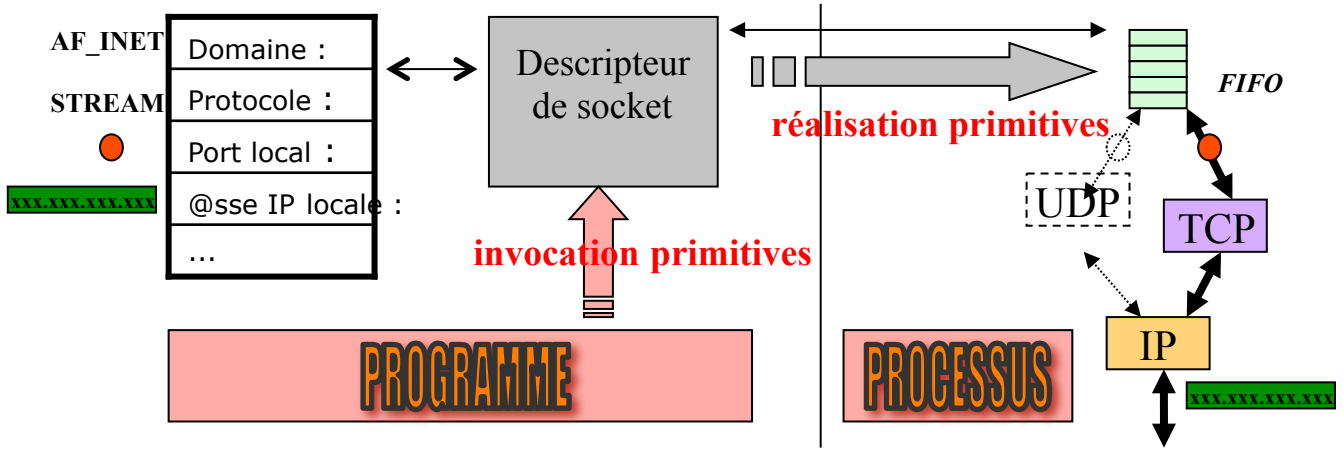
==> concept universel

Caractéristiques générales (3/4)

- Différents types de sockets :
  (variables selon les technologies)
  - Communications locales :
    - IPC UNIX (domaine AF_UNIX)
  - Communications distantes
    - point à point
      - *Mode stream* (ou connecté) :
        - établir et utiliser une connexion TCP pour un transfert fiable
        - Exemples de services : ftp, http, smtp, telnet, BGP…
      - *Mode datagramme* (ou non connecté) :
        - Datagrammes indépendants
        - Non fiable, plus rapide
        - Exemples de services : snmp, tftp, dhcp, dns, RIP…
      - *Mode raw* (interface de bas niveau)
        - Accès aux couches basses (ex : ICMP)
    - multipoint
      - *Socket Multicast*

Caractéristiques générales (4/4)

- Abstraction pour le programmeur :
  Assimiler un flux de données échangé avec un processus distant via le réseau à un flux d'E/S comme un autre !
  - Vision réseau : association d'application = Quintuplet :
    - Adresse IP et port locaux
    - Adresse IP et port distants
    - Protocole de Transport (UDP ou TCP)
  - Vision Système :
    - Équivalent à un fichier (FIFO) avec descripteur



Sockets mode datagramme

- Principe
  - **Datagrammes individuels transférés**
    - de façon non fiable, du « mieux » possible
    - pas de connexion entre émetteur(s) et récepteur
    - contrôle parfois possible de couplage

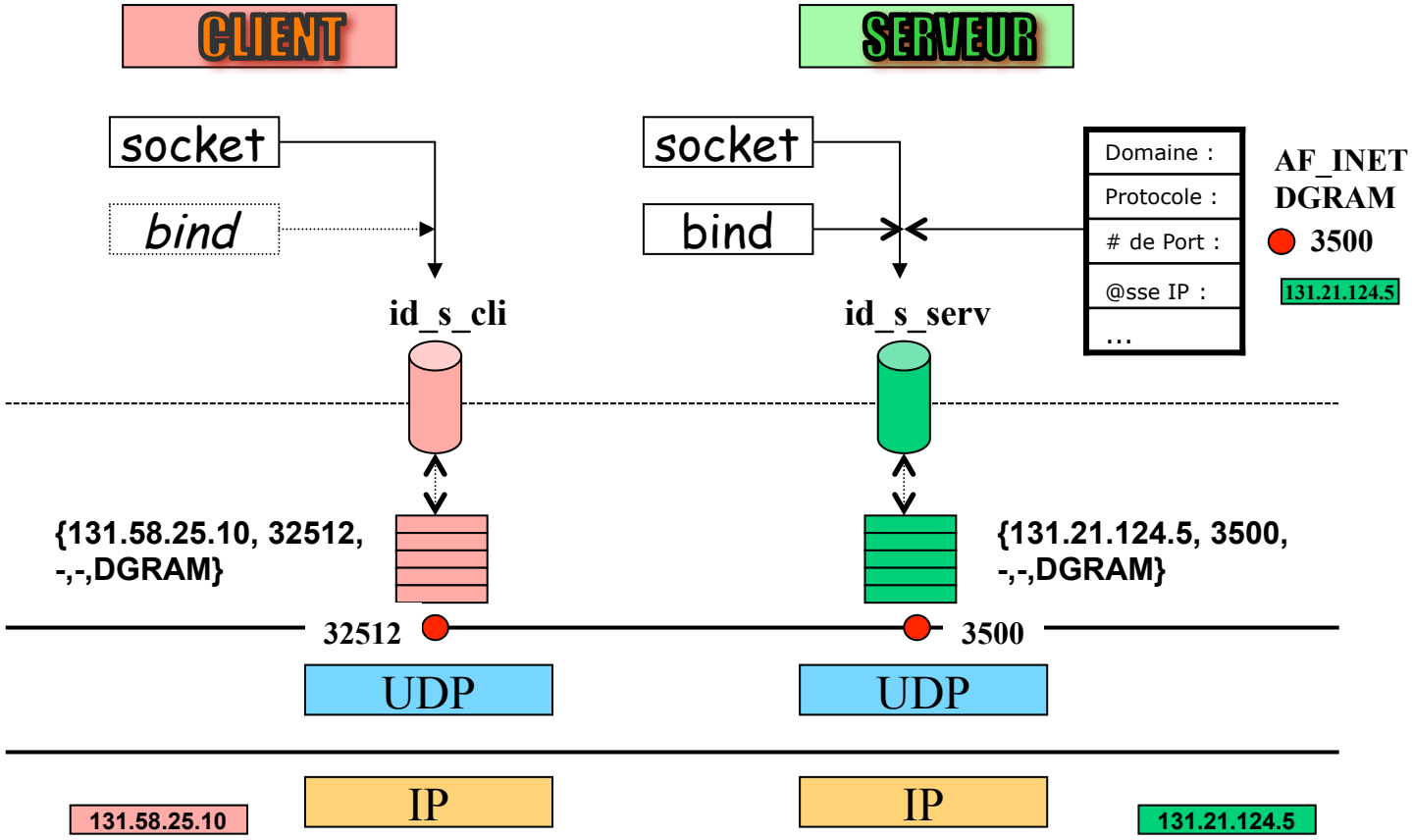
    ==> service UDP utilisé
  - Une seule phase à programmer :
    - transfert de données
      - envoi et réception possibles (**requêtes et réponses selon le protocole applicatif**)
      - détermination de l'expéditeur par le récepteur;
        L'expéditeur peut ensuite devenir destinataire.
      - éventuellement contrôle de la fiabilité et du dialogue
  - Au préalable, création et caractérisation des sockets
  - Libération de port après usage

Sockets mode datagramme

- **Création et caractérisation** :
  - Conception :
    - Créer une socket localement
    - La caractériser si réception de données prévue
      - Numéro de port prédéfini côté « serveur »
  - Programmation en C/Unix :
    - Primitive **socket** *// création en mode DGRAM*
    - *[Primitive **bind**]* *// assignation port et @sse IP locaux*
  - Programmation en JAVA
    - Classe **DatagramSocket**
    - Trois constructeurs possibles
      - sans paramètre : port anonyme local
      - avec no_port local *// épier des messages entrants sur ce port*
      - et @sse IP_locale

Sockets mode datagramme

- Illustration : création et caractérisation



Sockets mode datagramme

- **Transfert de données (des deux côtés) :**
  - Conception :
    - EMISSION : Envoyer X octets vers une socket distante dont on précise les caractéristiques
    - RECEPTION : Recevoir X octets provenant d'une socket distante dont on récupère les caractéristiques pour en déterminer l'origine
      = = > fonction bloquante
  - **Programmation en C/Unix**
    - Emission : primitives **sendto**
    - Réception : primitives **recvfrom**
  - **Programmation en JAVA**
    - Classe **DatagramSocket** *// socket pour recevoir ou émettre des datagram packets*
      - Emission : méthode **send**
      - Réception : méthode **receive**
    - Classe **DatagramPacket** *// paquet UDP émis ou reçu*
      - Constructeurs pour l'envoi
      - Constructeurs pour la réception

Compléments mode datagramme en Java

- Classe **`DatagramPacket`** pour manipuler des paquets UDP
  - Principal constructeur pour émission de paquets :
    ```
    DatagramPacket(byte[] buf, int length, InetAddress address, int port)
    ```
  - Principal constructeur pour réception de paquets :
    ```
    DatagramPacket(byte[] buf, int length)
    ```
- Méthodes utiles : `getData(), getLength() getAddress(),…`
- Exemple
  - Code client :

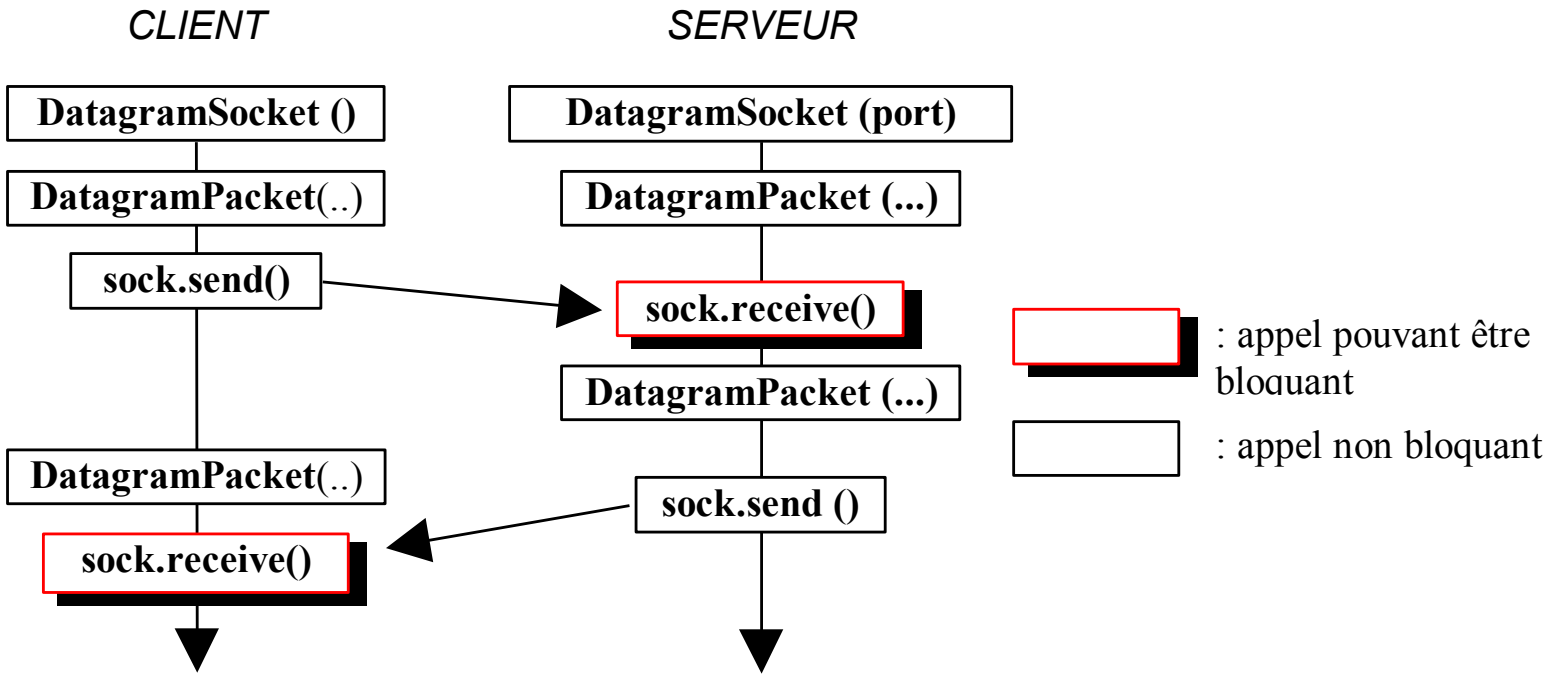
```
byte[] req = myStringRequest.getBytes();
DatagramPacket packetOut =
    new DatagramPacket(req, req.length,
            InetAddress.getByName(remoteServer), remotePort);
```

Compléments mode datagramme en Java

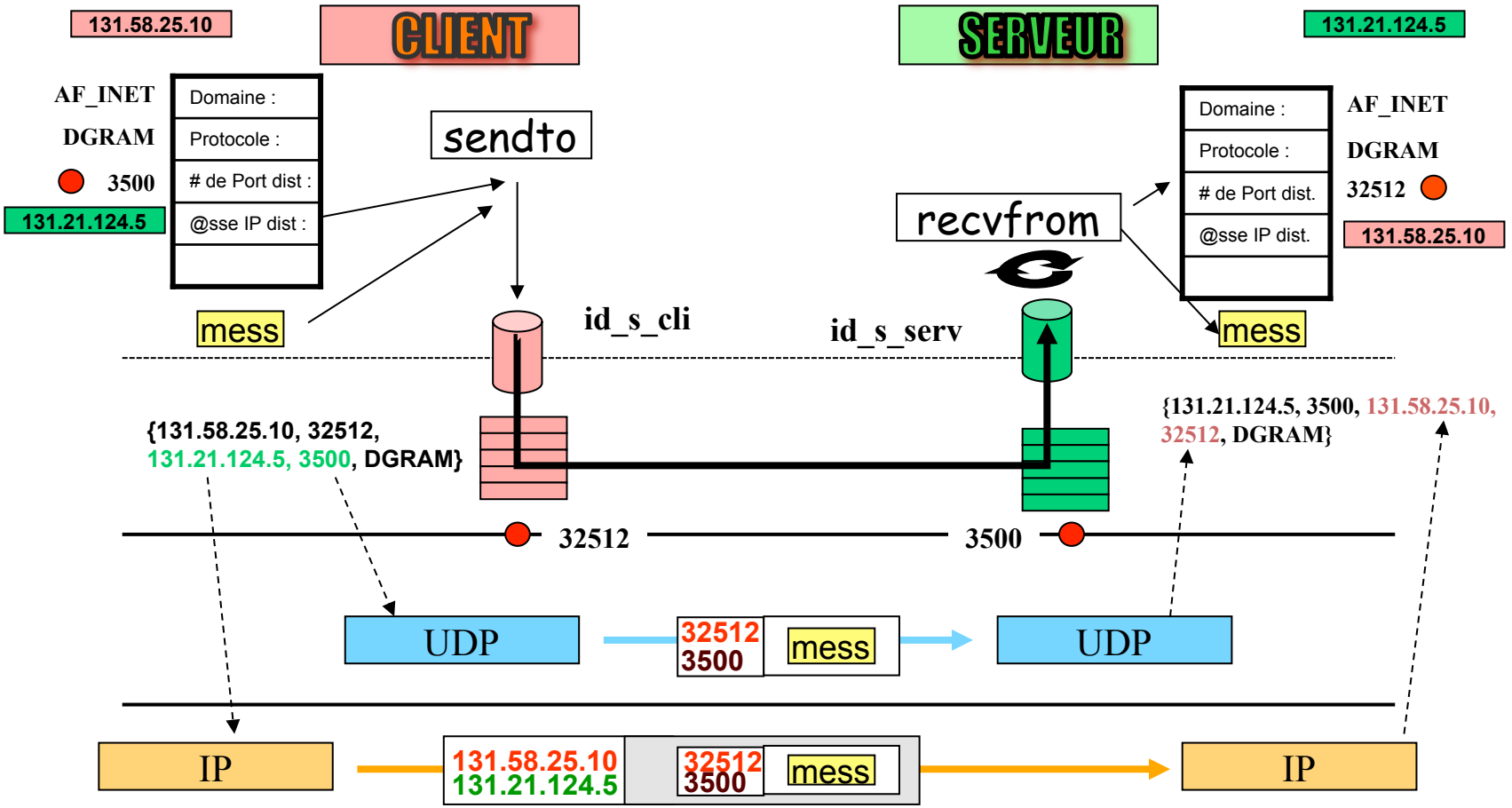
- Classe **`DatagramSocket`** : représente une socket pour émettre et recevoir des `DatagramPacket`
  - Constructeur associant la socket locale sur un port quelconque disponible
    `DatagramSocket()`
  - Constructeur associant la socket locale sur un port spécifique
    `DatagramSocket(int port)`
  - Constructeur associant la socket locale sur port et adresse spécifiques
    `DatagramSocket(int port, InetAddress laddr)`
- Méthodes utiles : `send(), receive(), close(),…`
- Exemple
  - Code client

```
DatagramSocket dsock = new DatagramSocket();
dsock.send(packetOut);
```

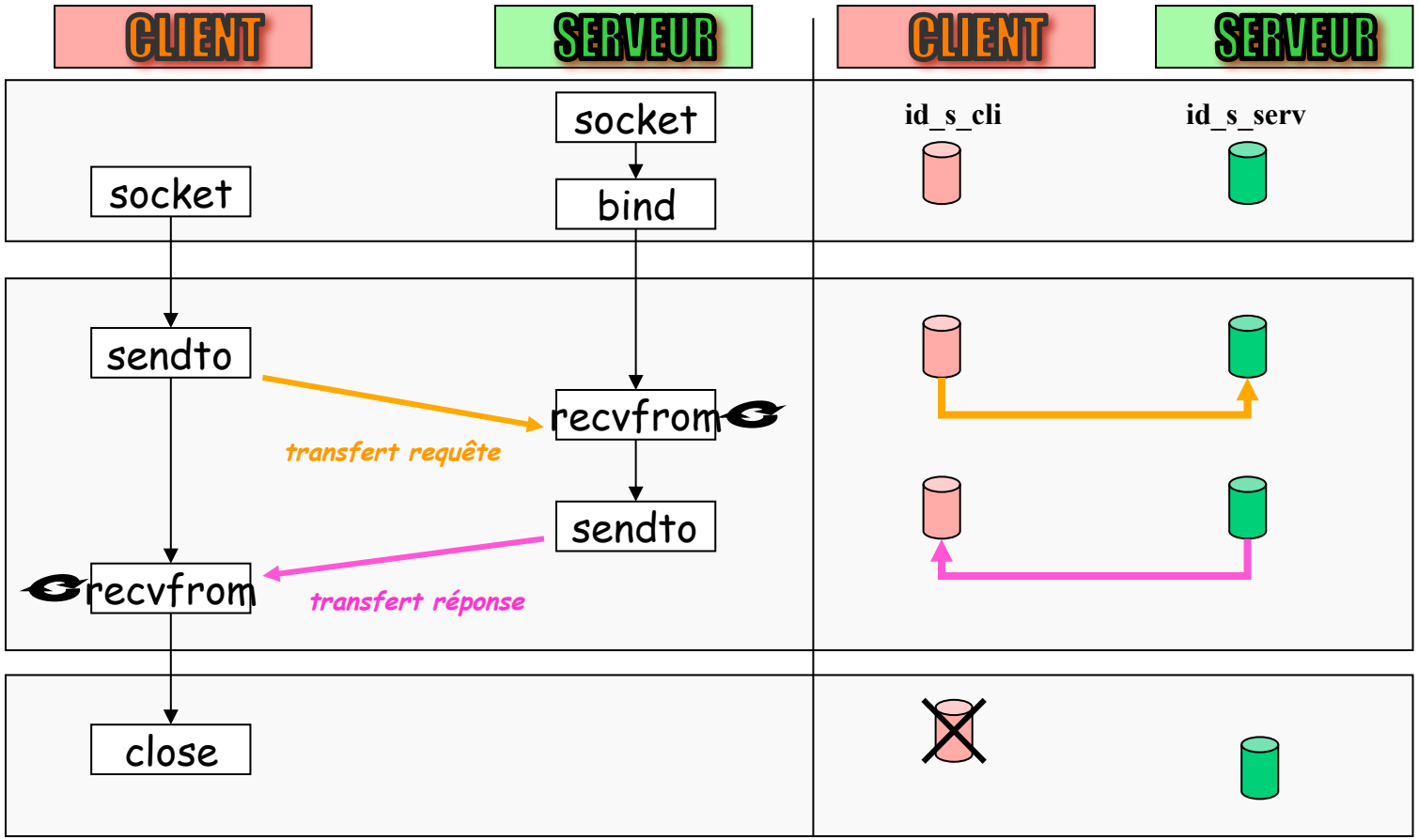
Mode datagramme – récapitulatif vision Java



Mode datagramme – récapitulatif vision UNIX / C

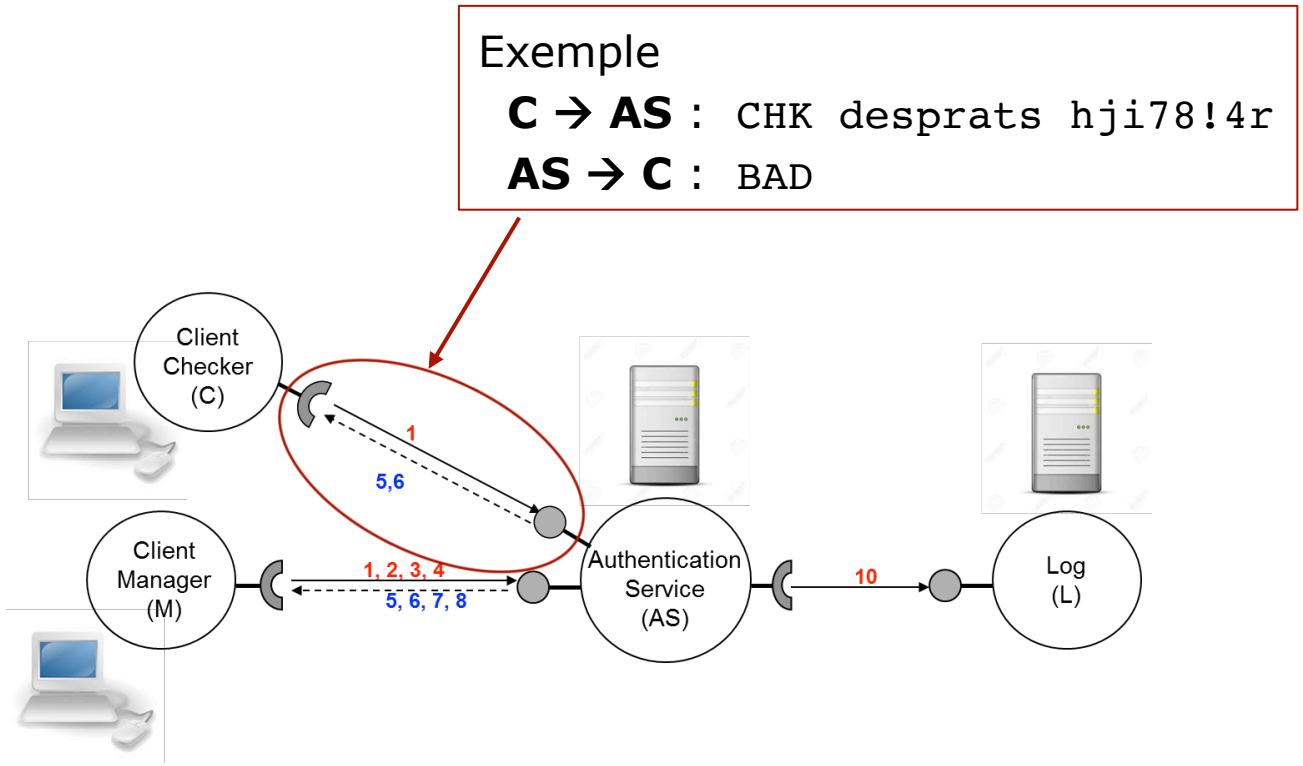


Mode datagramme – récapitulatif vision UNIX / C



Travaux pratiques !

- Développement d'un service (simplifié) d'authentification entre un client « Checker » et un serveur « Authentication Service », sur un transport non connecté UDP



Travaux pratiques !

- Développement d'un service (simplifié) d'authentification entre un client « Checker » et un serveur « Authentication Service », sur un transport non connecté UDP
- Deux programmes :
  - Client : `CheckUDP.java`
  - Serveur : `AuthServerUDP.java`
    (ce dernier s'appuiera sur la classe `ListeAuth.java` fournie)

Sockets mode connecté (principe)

- **Principe**
  - Flot bidirectionnel d'octets transférés
    - de façon fiable (ni perte, ni duplication) et ordonnée
    - entre deux locuteurs préalablement « connectés »

    ==> service TCP utilisé
  - Trois phases nécessaires à programmer :
    - établissement de connexion
      - Un initiateur (e.g. l'entité « client » : ouverture active)
      - Un répondant (e.g. l'entité « serveur » : ouverture passive)
    - transfert de données
      - Requêtes, [réponses] (selon le protocole applicatif)
    - libération de connexion
      - Bi partie
  - Au préalable, création et caractérisation des sockets

Sockets mode connecté (établissement connexion)

- **Ouverture passive (côté « répondant »)** :
  - Conception :
    - Dédier une socket destinée à intercepter des demandes d'établissement de connexion
      = = > socket générique d'écoute
    - Caractériser cette socket en terme de communication :
      - au moins un numéro de port (associé au service)
      - éventuellement une adresse IP (interface cible)
    - Lui permettre de scruter et stocker les demandes de connexions entrantes avant acceptation
      - paramétrer la taille de la FIFO des demandes de connexion
  - Programmation en C/Unix :
    - Primitive **socket** *// création en mode STREAM*
    - Primitive **bind** *// assignation numéro de port et adresse IP*
    - Primitive **listen** *// taille maximale FIFO*
  - Programmation en JAVA :
    - Classe **ServerSocket**
    - Trois constructeurs possibles
      - numéro_port (obligatoire)
      - Et taille maximale FIFO
      - Et Adresse IP

Sockets mode connecté (établissement connexion)

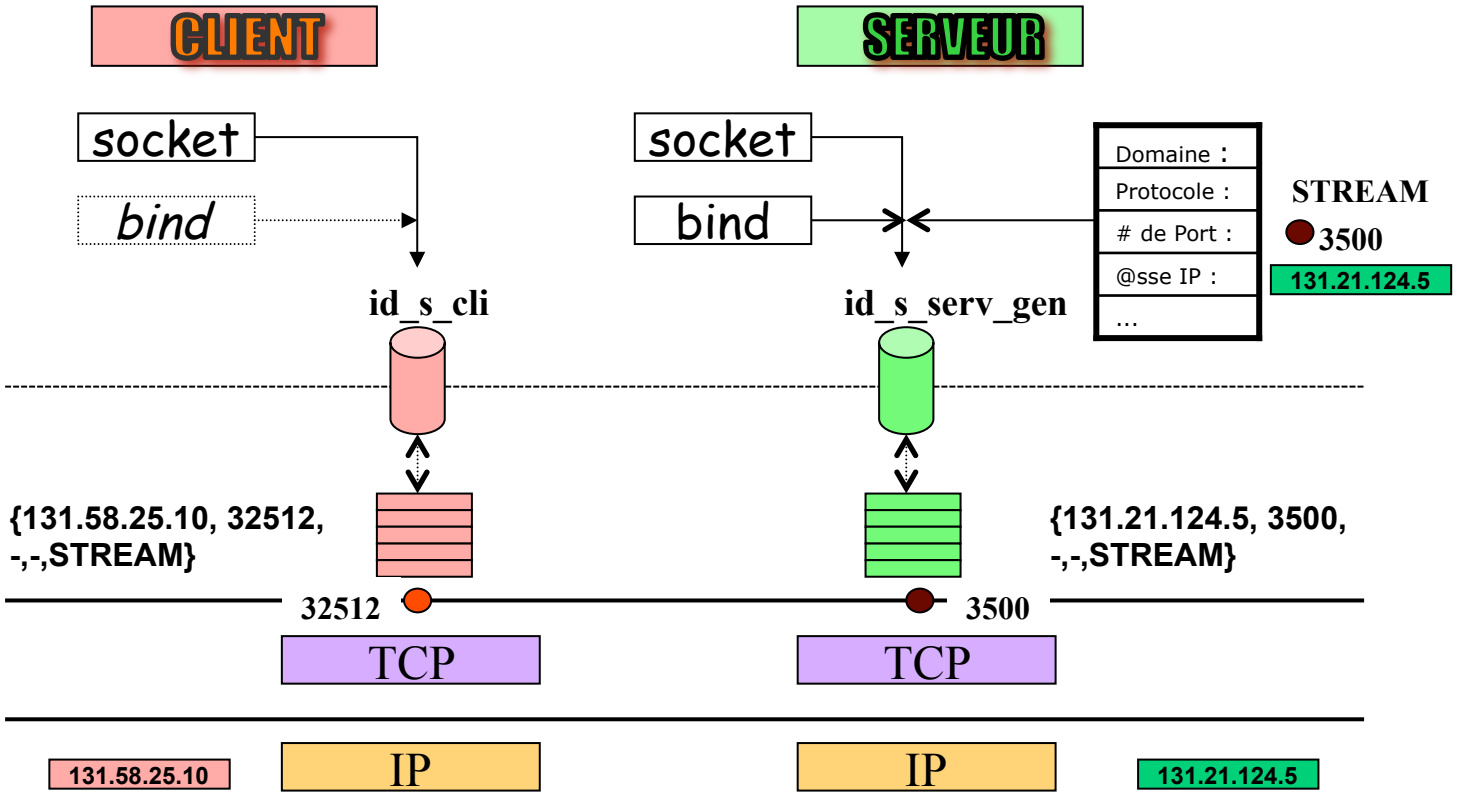
- **Ouverture active (côté « initiateur »)** :
  - Conception :
    - Créer une socket localement (et éventuellement la caractériser)
    - L'utiliser pour demander l'établissement d'une connexion TCP avec un processus distant « répondant »
    - Spécifier ce dernier grâce aux caractéristiques de communication de sa socket générique d'écoute :
      - numéro de port
      - Adresse IP
  - Programmation en C/Unix :
    - Primitive **socket** *// création en mode STREAM*
    - *[Primitive **bind**]* *// assignation numéro de port et @sse IP locaux*
    - Primitive **connect** *// désignation de la socket cible par num de port et @sse IP distants*
  - Programmation en JAVA
    - Classe **Socket**
    - Quatre constructeurs possibles
      - @sse IP **ou** nom_host distant **ET** numéro_port distant (obligatoires)
      - Et @sse IP **ou** nom_host local **ET** numéro_port local

Sockets mode connecté (établissement connexion)

- **Acceptation d'ouverture (côté « répondant »)** :
  - Conception :
    - Le processus doit être bloqué tant qu'aucune demande de connexion ne lui parvient sur la socket générique d'écoute :
      = = > fonction bloquante
    - L'acceptation consiste à considérer positivement une demande reçue et à **créer localement une socket spécifique dédiée à supporter l'échange de données sur cette connexion particulière**
      = = > Cette socket représente l'extrémité « serveur » de la connexion TCP initiée par un « client » distant.
  - Programmation en C/Unix :
    - Primitive **accept** *// retourne l'identificateur de la nouvelle socket créée*
      // ainsi que les caractéristiques du client distant
  - Programmation en JAVA
    - Méthode **accept** sur classe ServerSocket *// retourne un objet Socket*

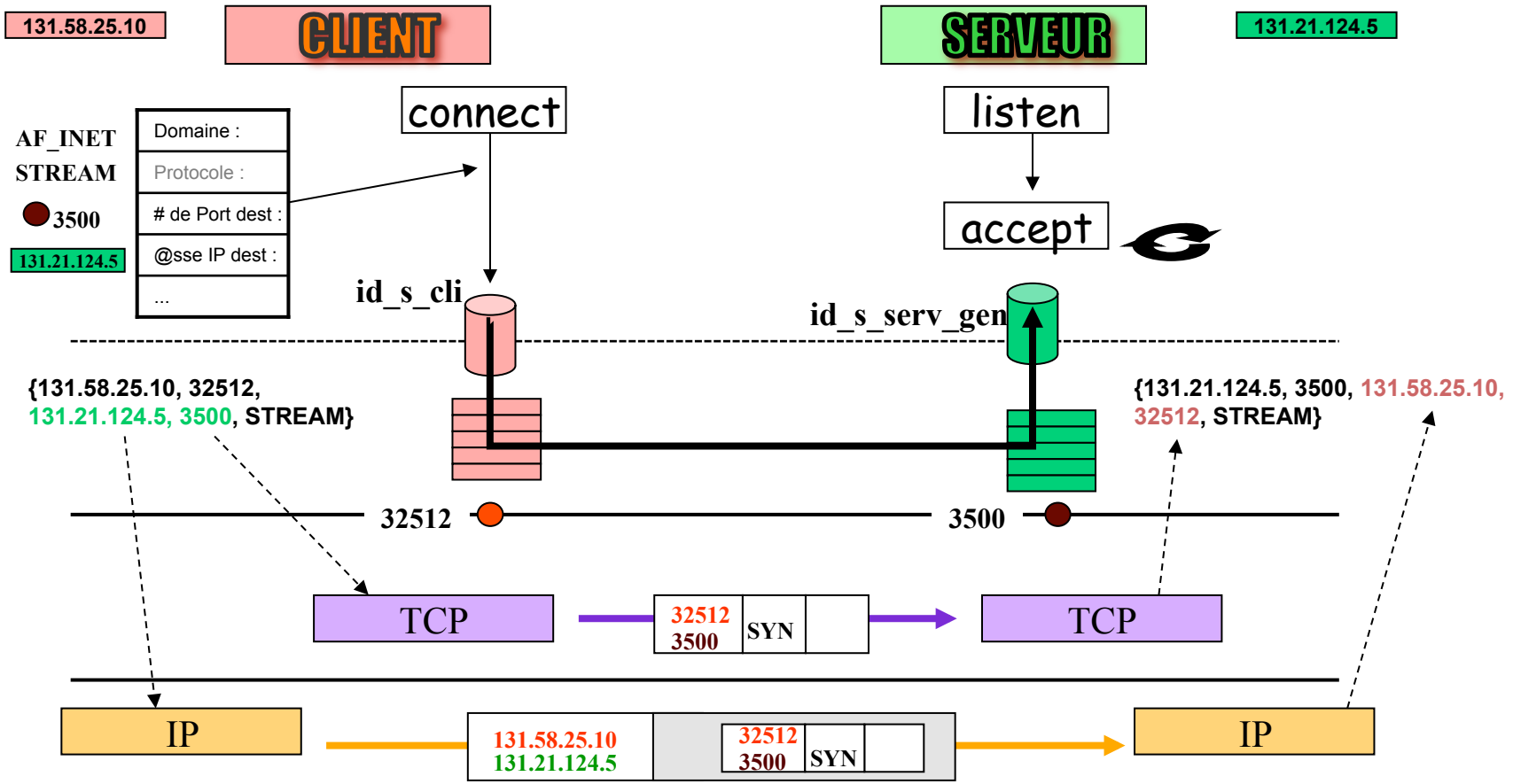
Sockets mode connecté (établissement connexion)

- Illustration : création et caractérisation



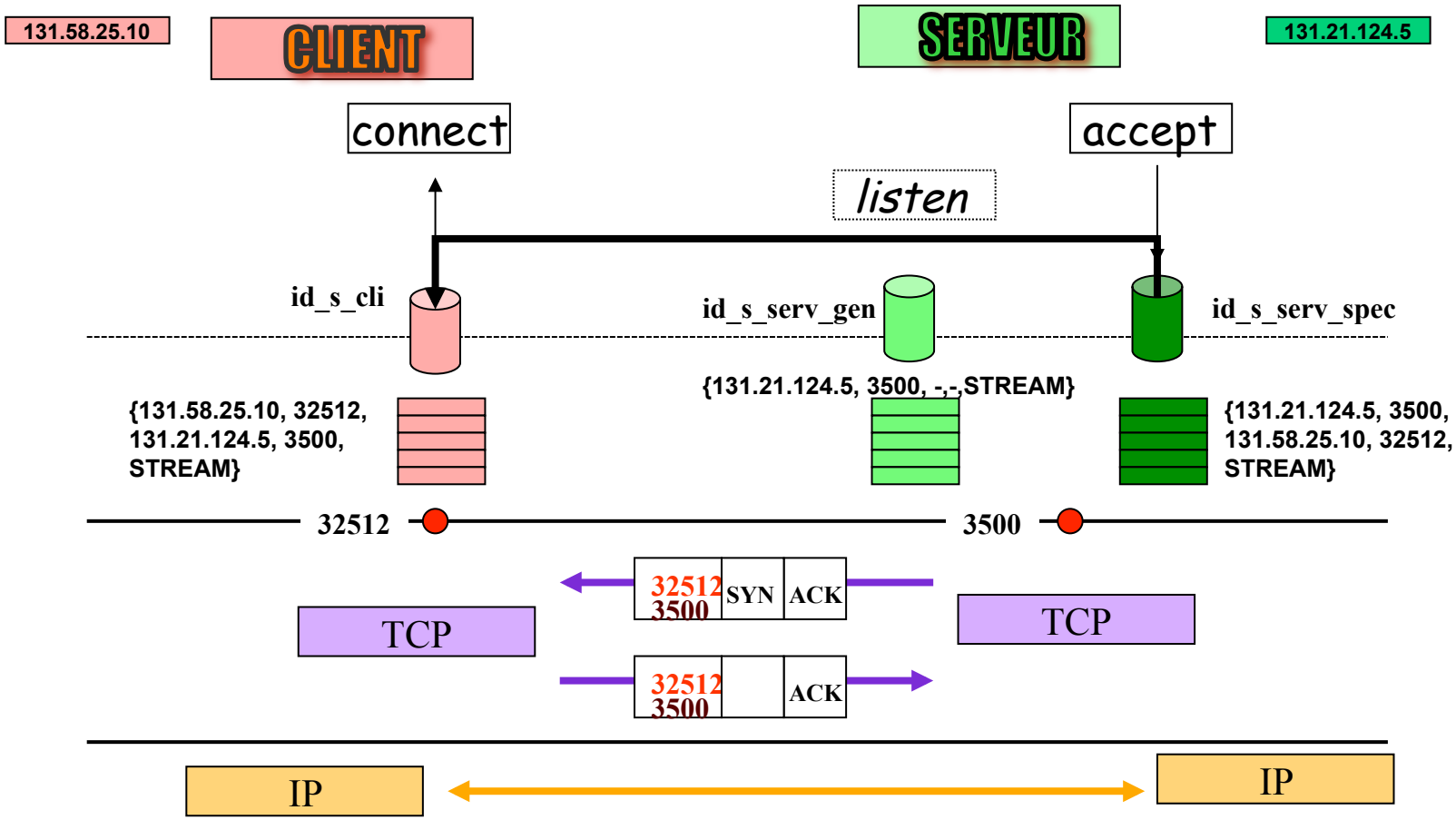
Sockets mode connecté (établissement connexion)

- Illustration : établissement de connexion



Sockets mode connecté (établissement connexion)

- Illustration : acceptation de connexion

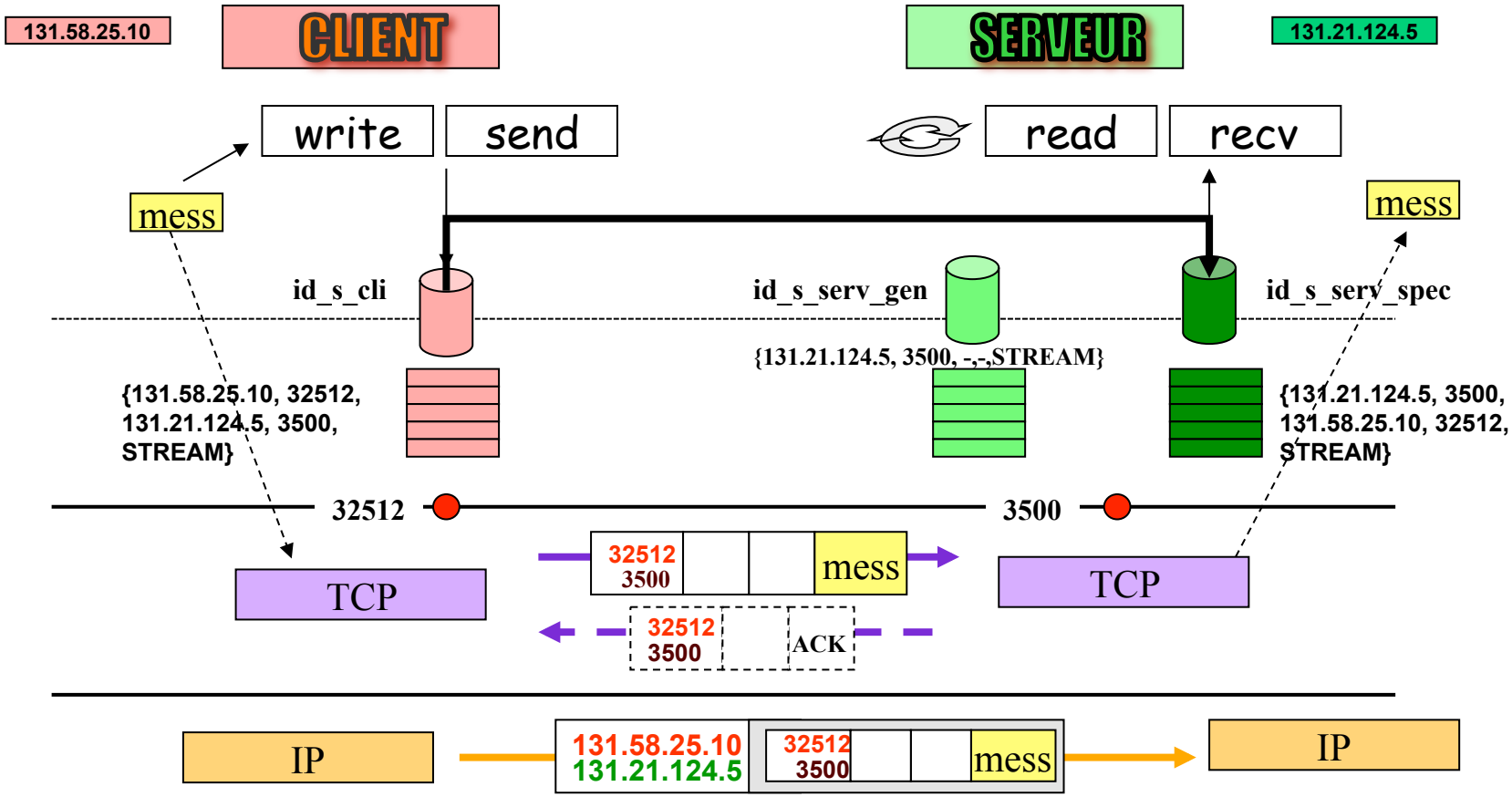


Sockets mode connecté (transfert de données)

- **Transfert de données** (des deux côtés) :
  - Conception :
    - EMISSION : Revient à Ecrire/Envoyer un message (x octets) sur la connexion
    - RECEPTION : Revient à Lire/Recevoir un message (x octets) sur la connexion
      = = > fonction bloquante
  - Programmation en C/Unix :
    - Emission : primitives **send** ou **write**
    - Réception : primitives **recv** ou **read**
  - Programmation en JAVA
    - Classe Socket
      - Emission : méthode **getOutputStream**
      - Réception : méthode **getInputStream**

Sockets mode connecté (transfert de données)

- Illustration : transfert de données

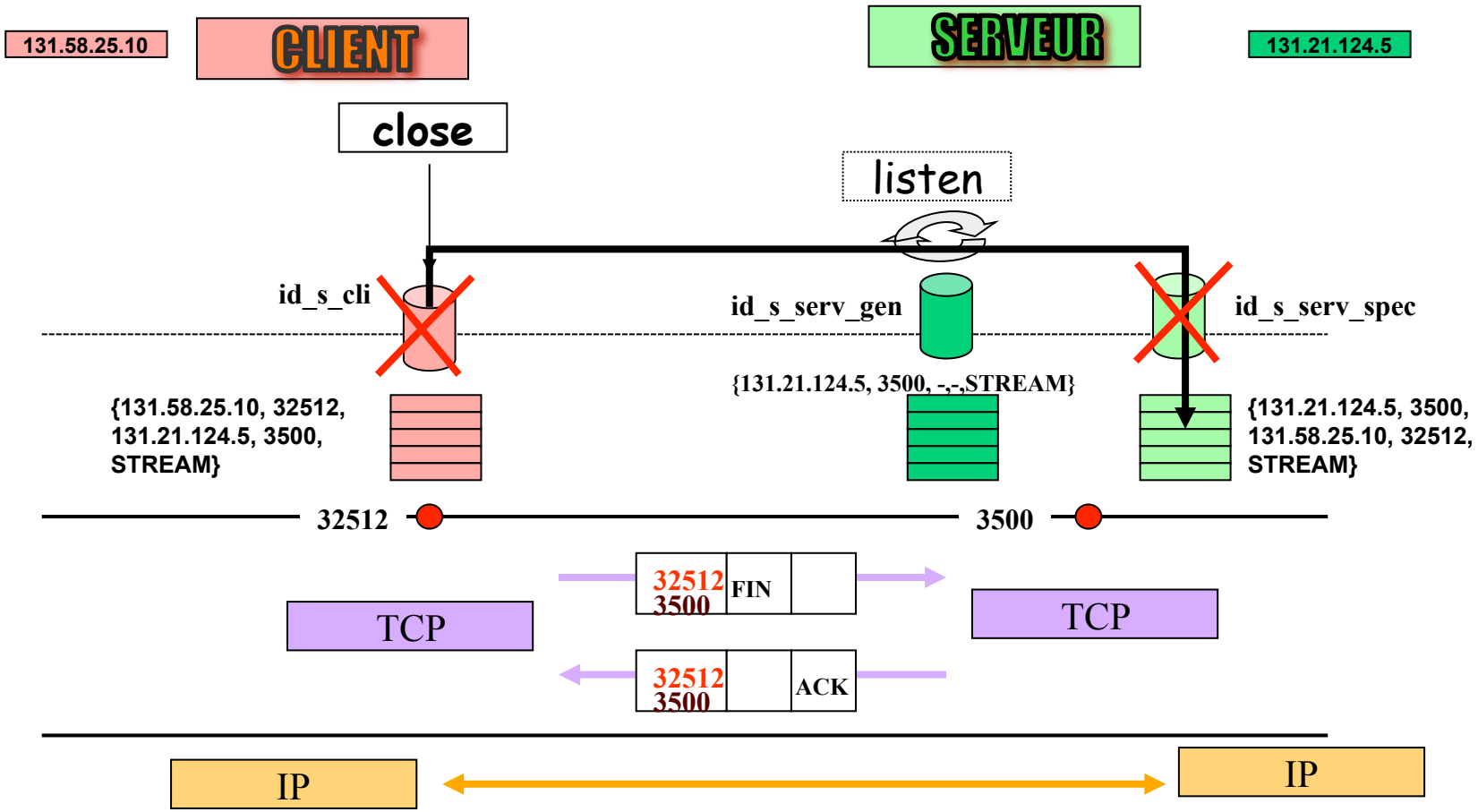


Sockets mode connecté (fermeture)

- **Libération de connexion** (des deux côtés) :
  - Conception :
    - Fermeture totale :
      - revient à fermer l'extrémité d'une connexion :
        = = > émission et réception deviennent impossibles
    - Fermeture partielle :
      - possible selon les technologies
        = = > émission ou réception devient impossible
  - Programmation en C/Unix :
    - primitive **close**
  - Programmation en JAVA
    - Classe Socket
      - méthode **close**
      - méthodes **shudownInput** et **shutdownOutput**

Sockets mode connecté (fermeture)

- Illustration : fermeture



Compléments mode connecté en Java

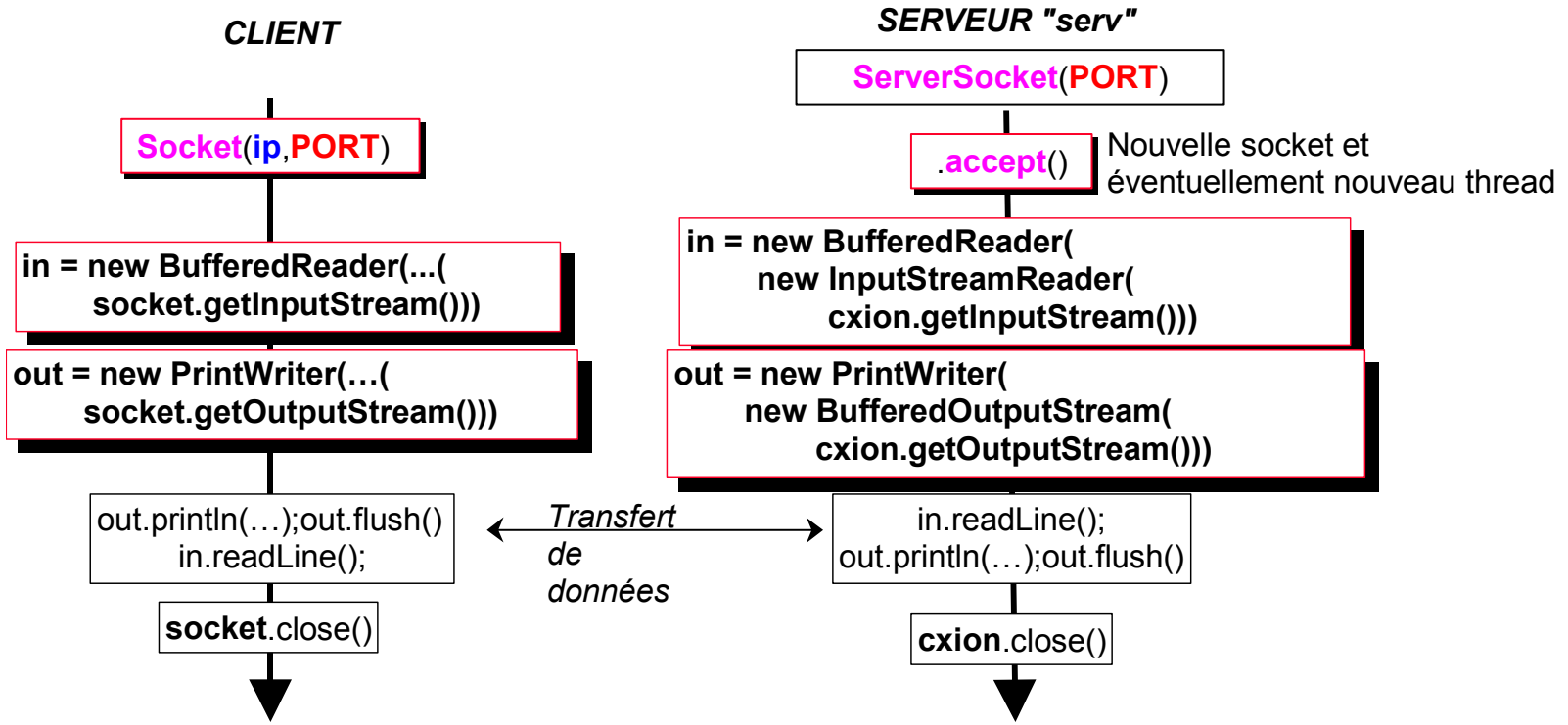
- Classe **`ServerSocket`** : représente une socket « serveur »
  - Principal constructeur :
    `ServerSocket(int port)`
    Creates a server socket, bound to the specified port.
  - Constructeur avec taille max file de demandes de connexion :
    `ServerSocket(int port, int backlog)`
    Creates a server socket and binds it to the specified local port number, with the specified backlog. […] The maximum queue length for incoming connection indications (a request to connect) is set to the backlog parameter. If a connection indication arrives when the queue is full, the connection is refused.
- Méthode pour attendre et accepter connexion : **`accept()`**
  (retourne un objet **`Socket`**)
- Exemple code serveur :
  - ```
    ServerSocket listenSock = new ServerSocket(2244, 10);
    Socket connectionSock = listenSock.accept();
    ```
    ```

Compléments mode connecté en Java

- Classe **`Socket`** : représente une socket en mode connecté
  - Principal constructeur (utilisé par un « client ») :
    `Socket(InetAddress address, int port)`
    *Creates a stream socket and **connects it** to the specified port number at the specified IP address.*
  - Un objet **`Socket`** est également renvoyé par la méthode `accept()` de la classe `ServerSocket` lors de l'acception d'une connexion

- Emission / réception via flux d'entrée-sortie :
  - Méthodes **`getInputStream()`** et **`getOutputStream()`**
    Ces flux d'octets (`InputStream/OutputStream`) peuvent être ensuite concaténés à d'autres flux offrant davantage de fonctionnalités :
    - Exemples : flux orientés « caractères » (`Reader/Writer`) , flux « bufferisés » : `BufferedXXX`, flux `Print` (méthodes print, println…)

Compléments mode connecté en Java

- Exemple code client
  - ```
    // Create client socket and connect it to server
    Socket sock = new Socket("141.115.64.43", 8888);

    // Buffered character stream for input
    BufferedReader br = new BufferedReader(
                 new InputStreamReader(sock.getInputStream()));
    // Line-oriented character stream for output
    PrintWriter pw = new PrintWriter(sock.getOutputStream(), true);

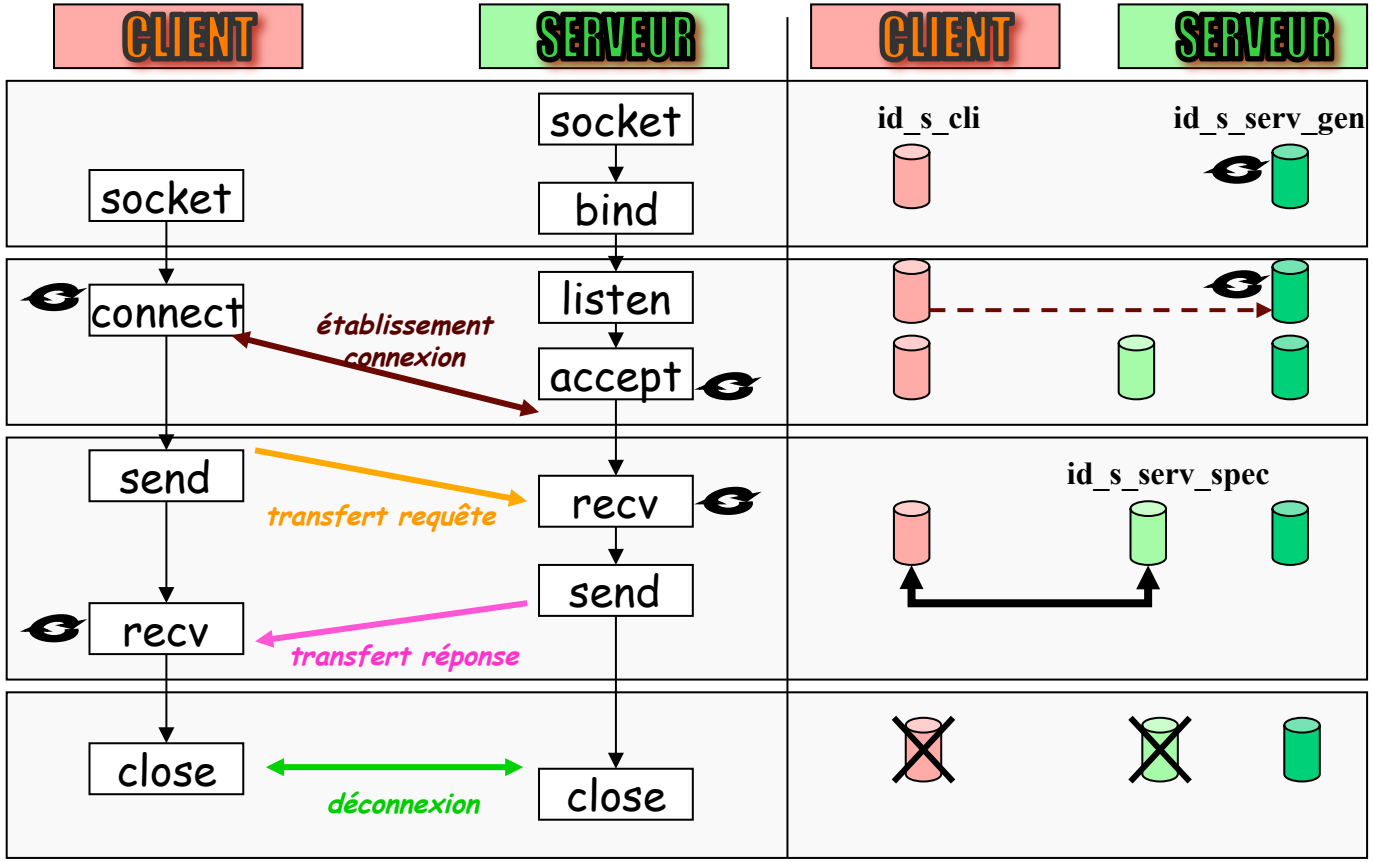
    // Send request
    pw.println(myStringRequest);
    // Wait for response
    String response = br.readLine();
    System.out.println("*** Response --> " + response);

    // Close socket
    sock.close();
    ```
    ```

Sockets mode connecté – récapitulatif vision Java



Sockets mode connecté – récapitulatif vision UNIX/C



Gestion des options

- Lire ou Ajuster des valeurs de paramètres
  - Temporels
    - Envoi immédiat (TCP_NODELAY)
    - Valeur maximale d'attente d'arrivée d'octets lors d'une lecture ou d'une acceptation de connexion (SO_TIMEOUT)
  - Fonctionnels
    - Gestion des paquets restants (non expédiés) lors d'une fermeture (SO_LINGER)
    - Taille des buffers de réception (SO_RCVBUF) et de transmission (SO_SNDBUF)
    - Maintien d'une connexion ouverte si silencieuse depuis une longue période de temps (SO_KEEPALIVE)
- Programmation en C/Unix :
  - primitives **getsockopt** et **setsockopt**
- Programmation en JAVA
  - Classes **SocketDatagram, Socket, ServerSocket**
    - Accesseurs **SetXXX** et **GetXXX**

Retour sur les TP…

- Communication client « Checker » et serveur « Authentication Service » sur un **transport TCP**