

Applications réseaux

accès à TCP

Etienne Duris
Arnaud Carayol

Le protocole TCP

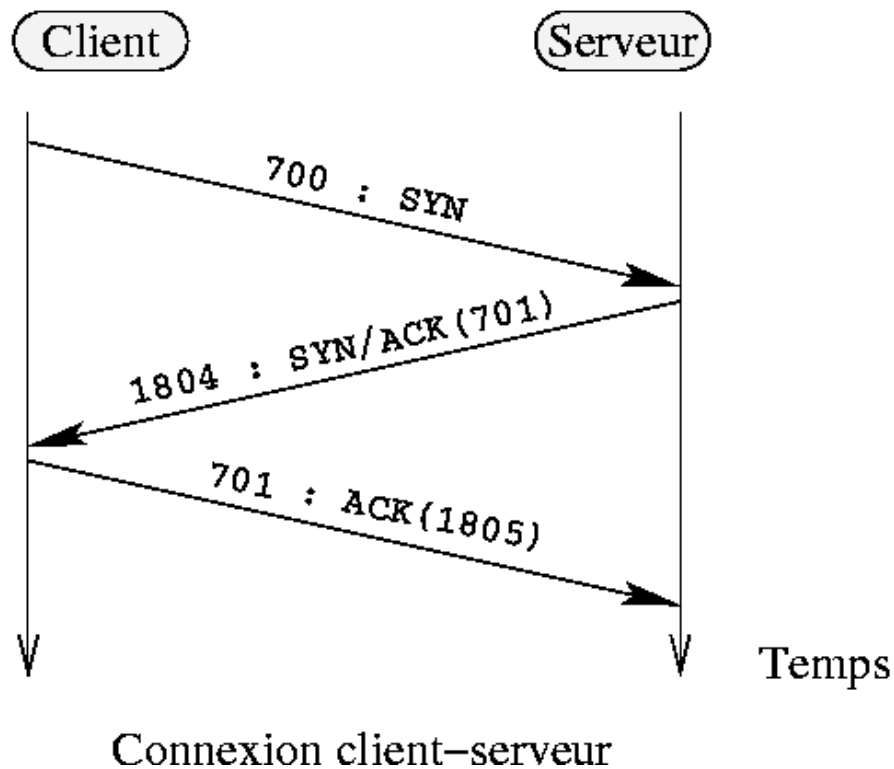
- Transmission Control Protocol, RFC 793
 - Communication
 - par flots,
 - fiable,
 - mode connecté
 - full duplex
 - Données bufferisées, encapsulées dans datagrammes IP
 - flot découpé en segments (~536 octets)
 - Mécanismes de contrôle de flot
 - ex: contrôle de congestion
 - assez lourd d'implantation (beaucoup plus qu'UDP)

Principe des segments

- La fiabilité est obtenue par un mécanisme d'acquittement des segments
 - À l'émission d'un segment, une alarme est amorcée
 - Elle n'est désamorcée que si l'acquittement correspondant est reçu
 - Si elle expire, le segment est réémis
- Chaque segment possède un numéro de séquence
 - préserver l'ordre, éviter les doublons
 - les acquittements sont identifiés par un marqueur ACK
 - transport dans un même segment des données et de l'acquittement des données précédentes: piggybacking

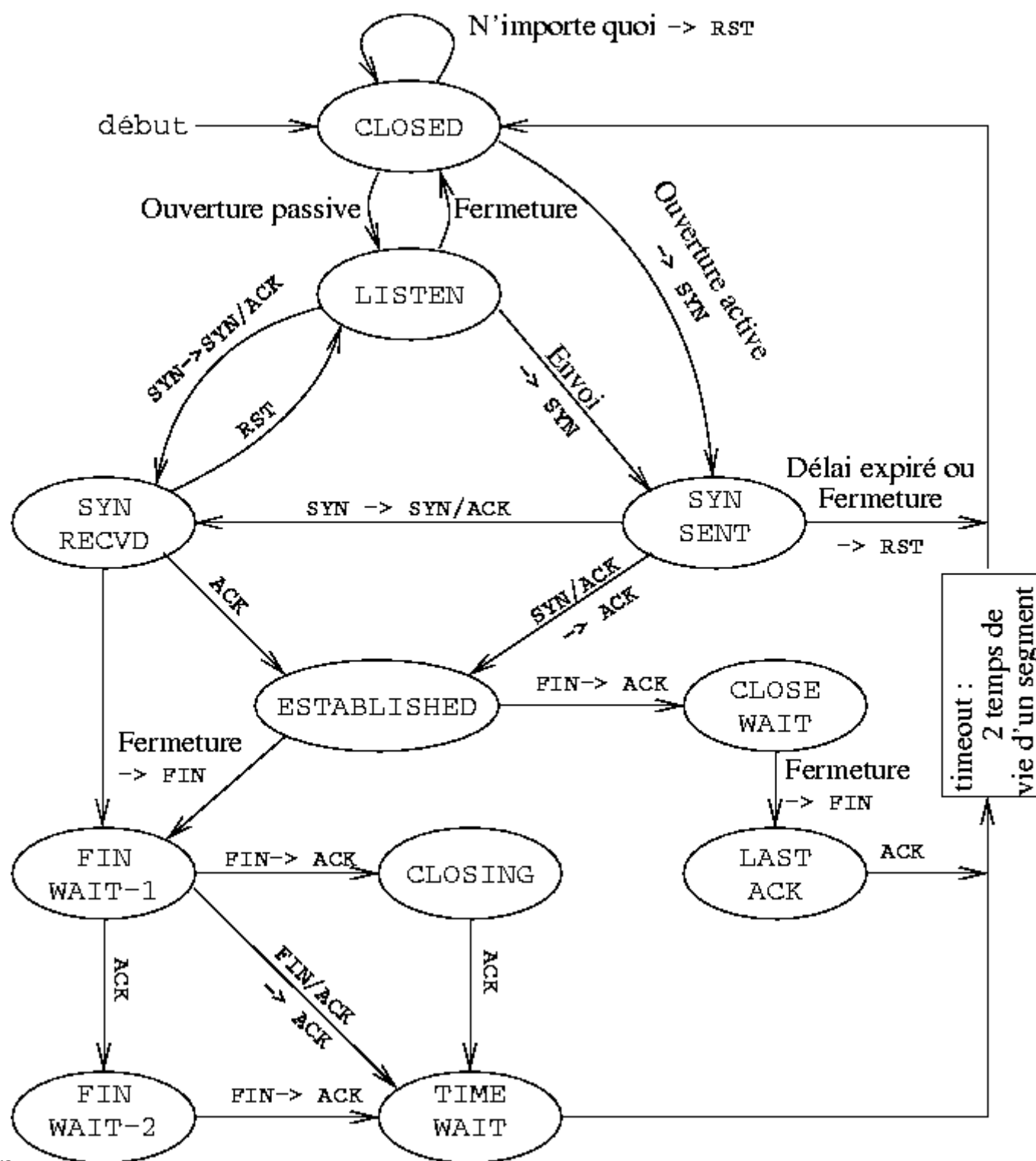
La connexion

- Three Way Handshake:
 - SYN --- SYN/ACK --- ACK

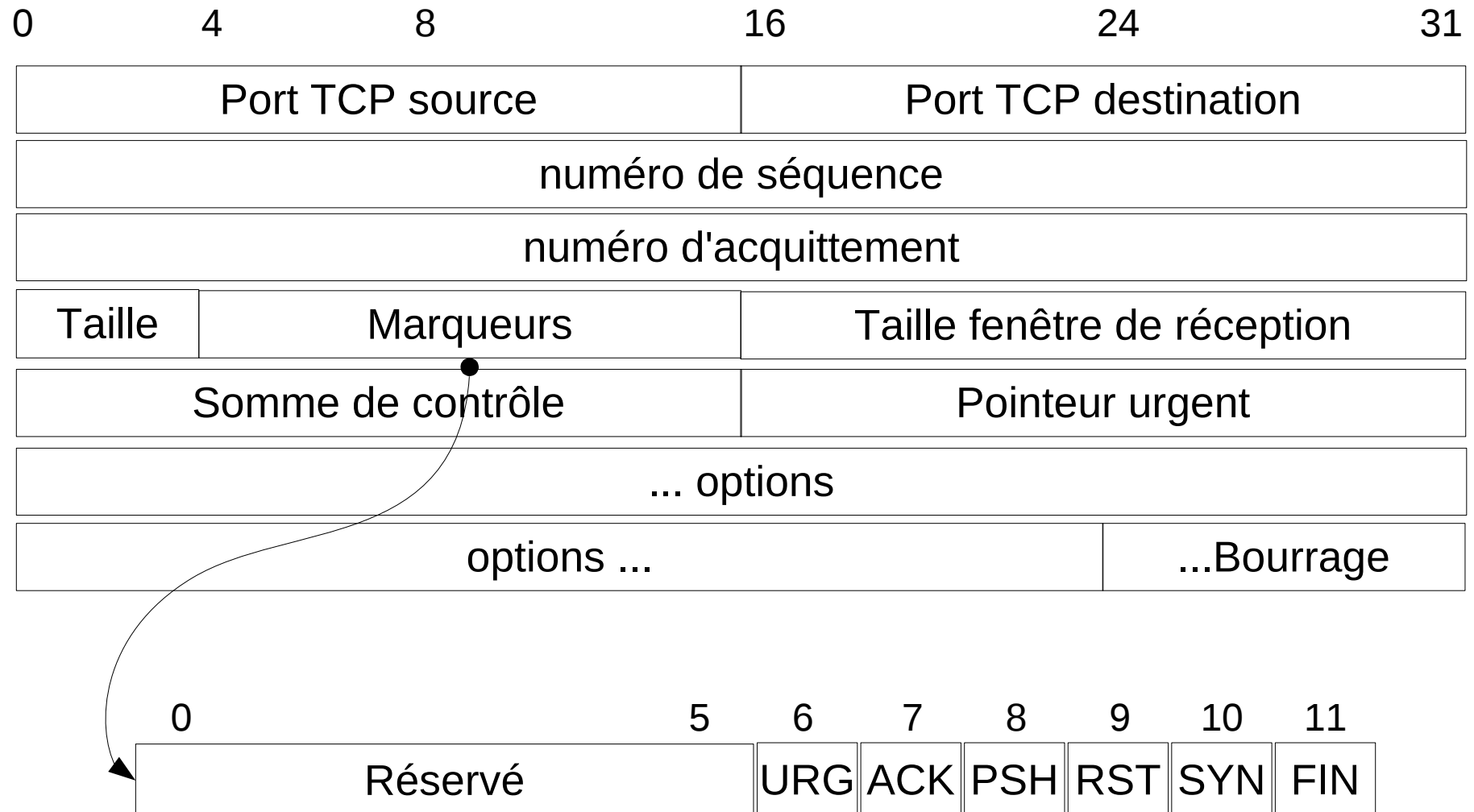


- Refus de connexion: SYN --- RST/ACK

Diagramme d'états



Format de trame TCP (en-tête)



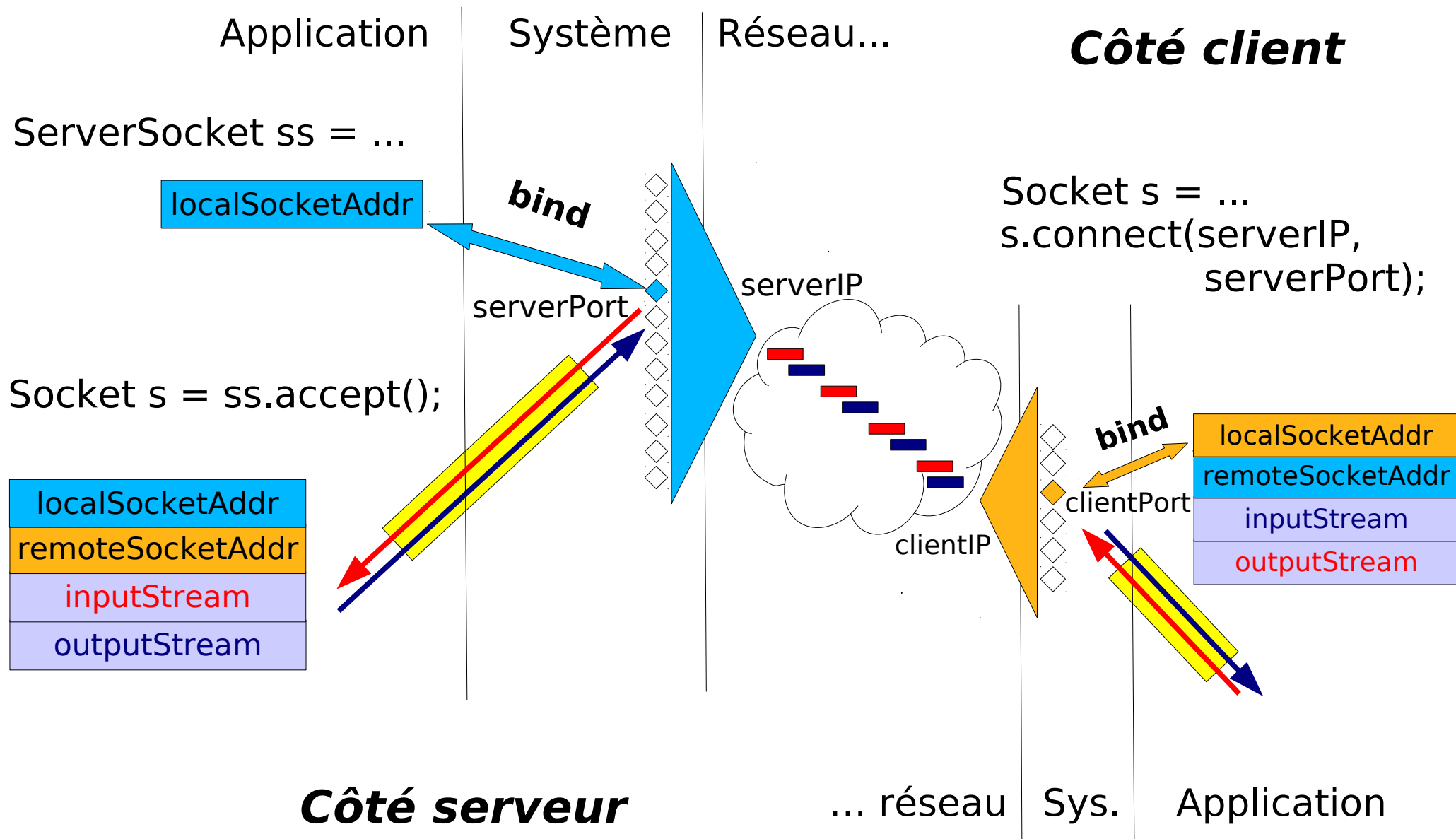
Format (suite)

- Champ Taille
 - en nombre de mots de 32 bits, la **taille de l'en-tête**
- Marqueurs
 - URG données urgentes (utilisation conjointe pointeur urgent)
 - ACK acquittement
 - PSH force l'émission immédiate (w.r.t. temporisation par défaut)
 - RST refus de connexion
 - SYN synchronisation pour la connexion
 - FIN terminaison de la connexion
- Somme de contrôle (comme IP v4 avec un pseudo en-tête)
- Options
 - Exemple: taille max. de segment, estampillage temporel

Sockets en Java

- [java.net.ServerSocket](#)
 - représente l'objet socket en attente de connexion des clients (du côté serveur)
- [java.net.Socket](#)
 - représente une connexion TCP,
 - tant du côté client (instigateur de la connexion par la création d'un objet [Socket](#))
 - que du côté serveur (l'acceptation par une [ServerSocket](#) retourne un objet [Socket](#))
- L'attachement se fait, comme en UDP, à une adresse IP et un numéro de port

Les sockets en TCP



Usage classique de Socket (côté client)

- On connaît le serveur auquel on veut se connecter
 - **Socket s = new Socket(serverIP, serverPort);**
 - De manière implicite :
 - Un port local est choisi pour attacher localement la socket
 - Le three way handshake est négocié avec le serveur
 - La socket représentant la connexion établie est retournée
 - Si problème de connexion lève une IOException
- Existe aussi avec le nom du serveur
 - **Socket s = new Socket(serverName, serverPort);**
 - Interroge le DNS pour récupérer l'adresse IP

D'autres façons de créer une socket

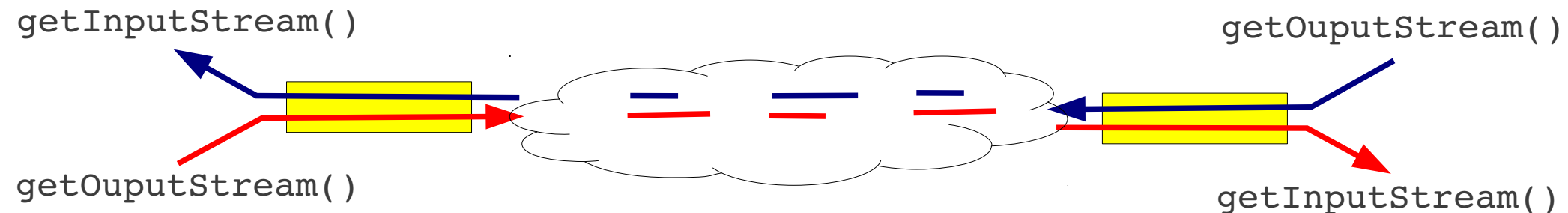
- On peut également faire étape par étape
 - En construisant un objet `Socket` inerte, puis en l'attachant localement, et enfin en demander l'établissement de la connexion avec le serveur
 - `Socket s = new Socket();`
`s.bind(SocketAddress bindpoint);`
`s.connect(SocketAddress);`
 - On peut aussi spécifier un timeout pour la connexion
 - `connect(SocketAddress bindpoint, int timeout)`
 - D'autres constructeurs précisent l'attachement local
 - `Socket(InetAddress addr, int port,`
`InetAddress localAddr, int localPort)`
 - `Socket(String host, int port,`
`InetAddress localAddr, int localPort)`

Les arguments de l'attachement

- Si `bind()` avec une `SocketAddress` qui vaut `null`
 - choisit une adresse valide et un port libre
- Possibilités d'échec de l'attachement local
 - `BindException` (port demandé déjà occupé)
 - `SocketException` (la socket est déjà attachée)
 - peut être testé avec `isBound()`
- Lors du `connect()`,
 - la socket est automatiquement attachée (si pas déjà fait)
 - initiation du *three way handshake* (`ConnectException`)
 - méthode bloquante qui retourne normalement si succès

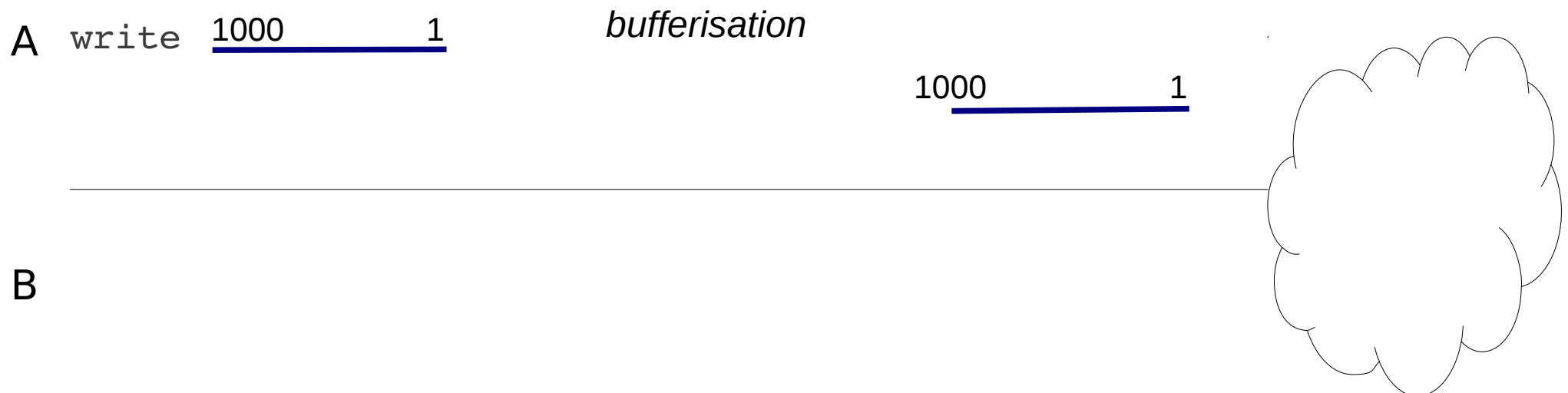
Utiliser la connexion

- Une fois l'objet socket construit, attaché et connecté, la connexion est établie
 - `InputStream getInputStream()` donne un flot de lecture des données arrivant sur la connexion
 - `OutputStream getOutputStream()` donne un flot d'écriture sur la connexion
 - les méthodes de lecture (`read()`) sont bloquantes
 - On peut borner l'attente en lecture par `setSoTimeout(int milliseconds)`
 - au delà de cette durée, `read()` lève une `SocketTimeoutException`
 - moins utile en TCP qu'en UDP (TCP est fiable!)



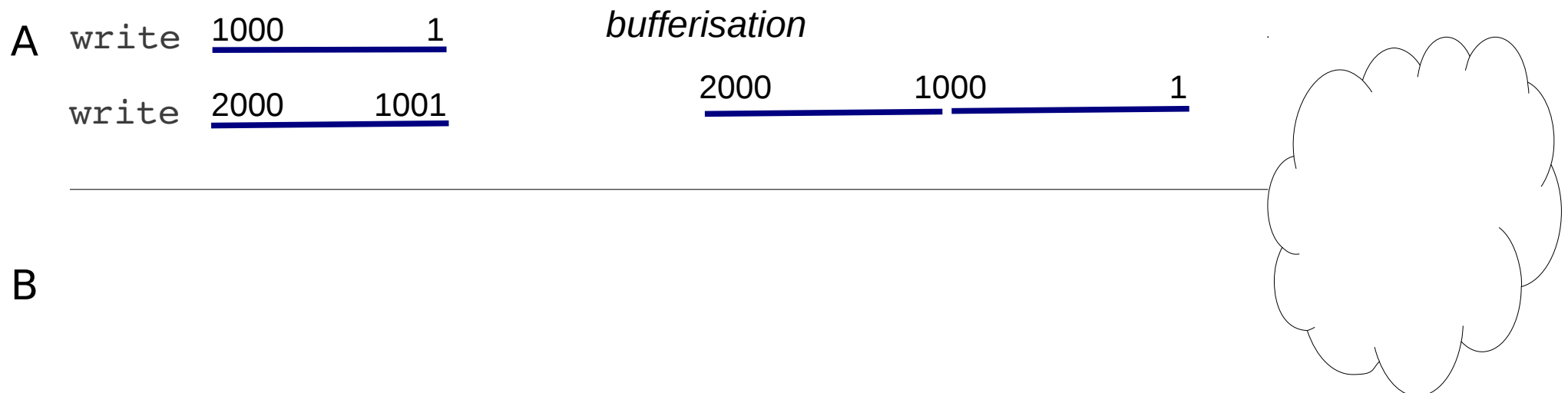
La segmentation TCP

- Contrairement à UDP, TCP ne préserve pas les limites
 - Si A écrit vers B deux fois 1000 octets...
 - Il est possible que TCP découpe ces données en 4, 5 segments ou plus, éventuellement de taille variable
 - Ces segments sont transportés dans des paquets IP indépendants
 - Il est donc possible que B lise tout d'un coup, ou en plusieurs fois...



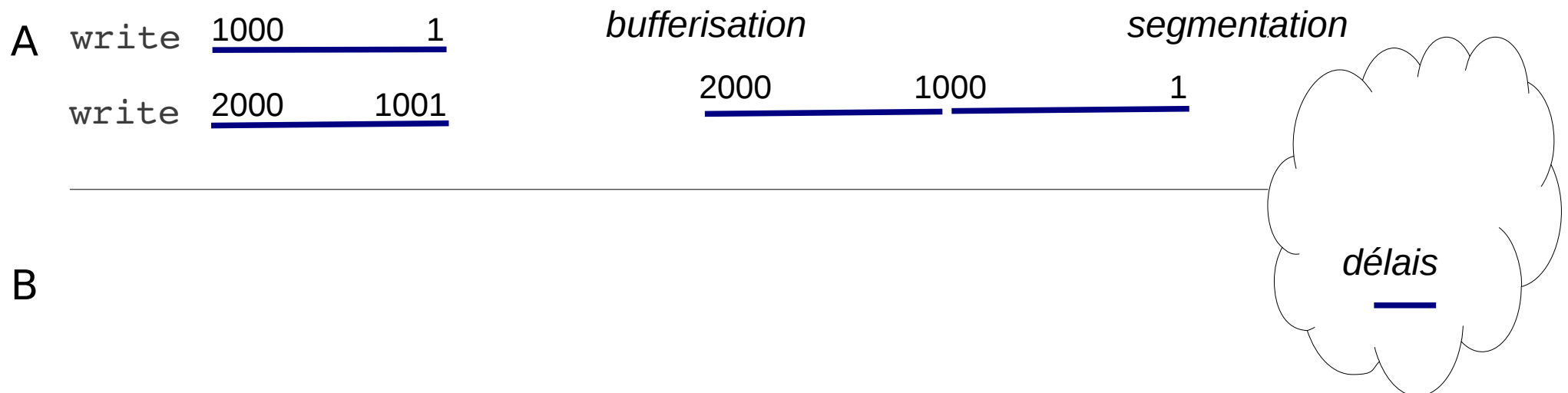
La segmentation TCP

- Contrairement à UDP, TCP ne préserve pas les limites
 - Si A écrit vers B deux fois 1000 octets...
 - Il est possible que TCP découpe ces données en 4, 5 segments ou plus, éventuellement de taille variable
 - Ces segments sont transportés dans des paquets IP indépendants
 - Il est donc possible que B lise tout d'un coup, ou en plusieurs fois...



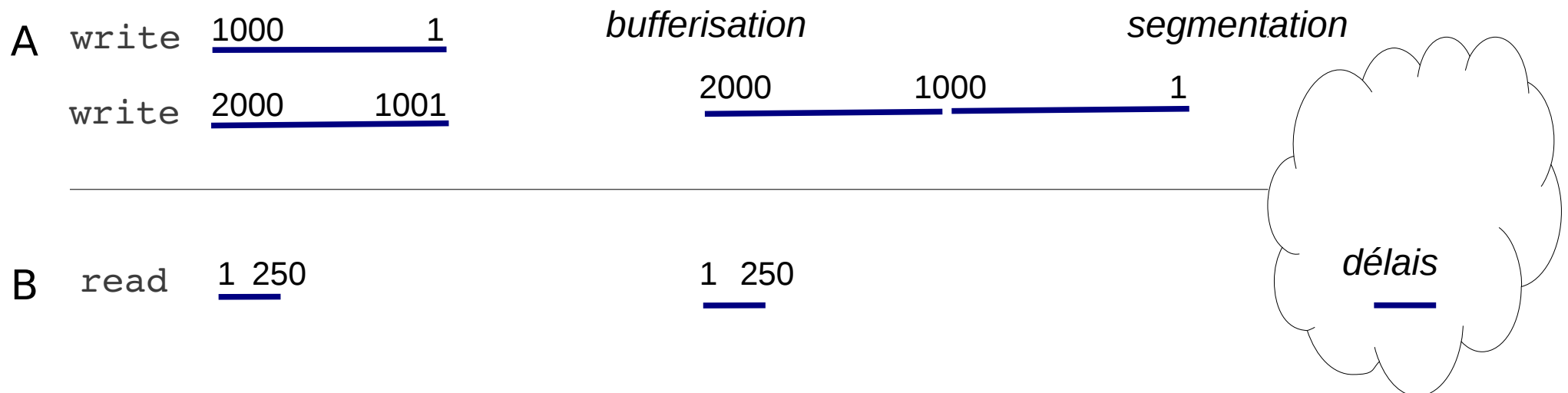
La segmentation TCP

- Contrairement à UDP, TCP ne préserve pas les limites
 - Si A écrit vers B deux fois 1000 octets...
 - Il est possible que TCP découpe ces données en 4, 5 segments ou plus, éventuellement de taille variable
 - Ces segments sont transportés dans des paquets IP indépendants
 - Il est donc possible que B lise tout d'un coup, ou en plusieurs fois...



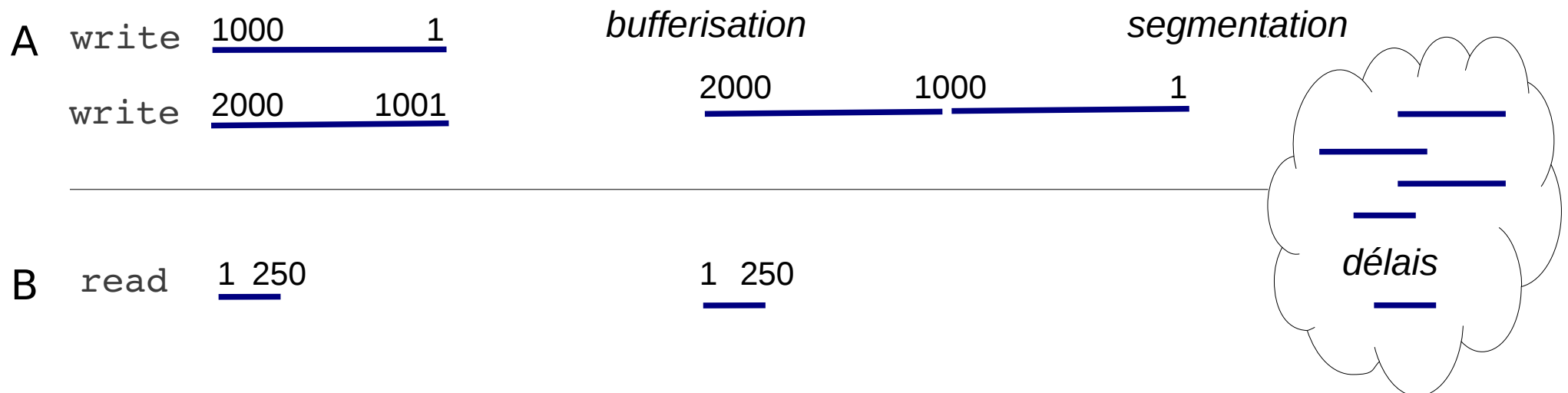
La segmentation TCP

- Contrairement à UDP, TCP ne préserve pas les limites
 - Si A écrit vers B deux fois 1000 octets...
 - Il est possible que TCP découpe ces données en 4, 5 segments ou plus, éventuellement de taille variable
 - Ces segments sont transportés dans des paquets IP indépendants
 - Il est donc possible que B lise tout d'un coup, ou en plusieurs fois...



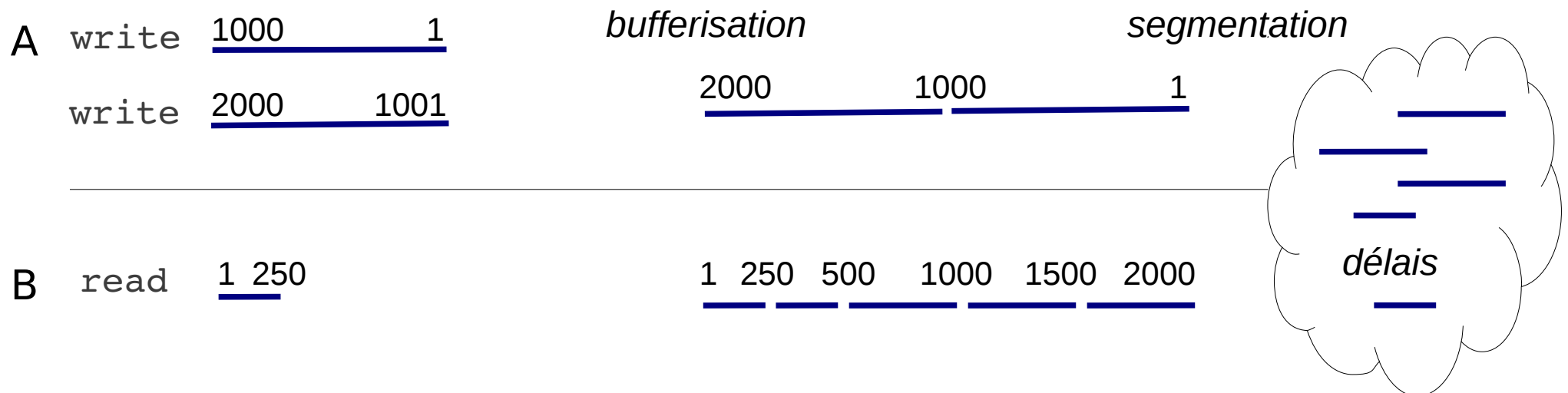
La segmentation TCP

- Contrairement à UDP, TCP ne préserve pas les limites
 - Si A écrit vers B deux fois 1000 octets...
 - Il est possible que TCP découpe ces données en 4, 5 segments ou plus, éventuellement de taille variable
 - Ces segments sont transportés dans des paquets IP indépendants
 - Il est donc possible que B lise tout d'un coup, ou en plusieurs fois...



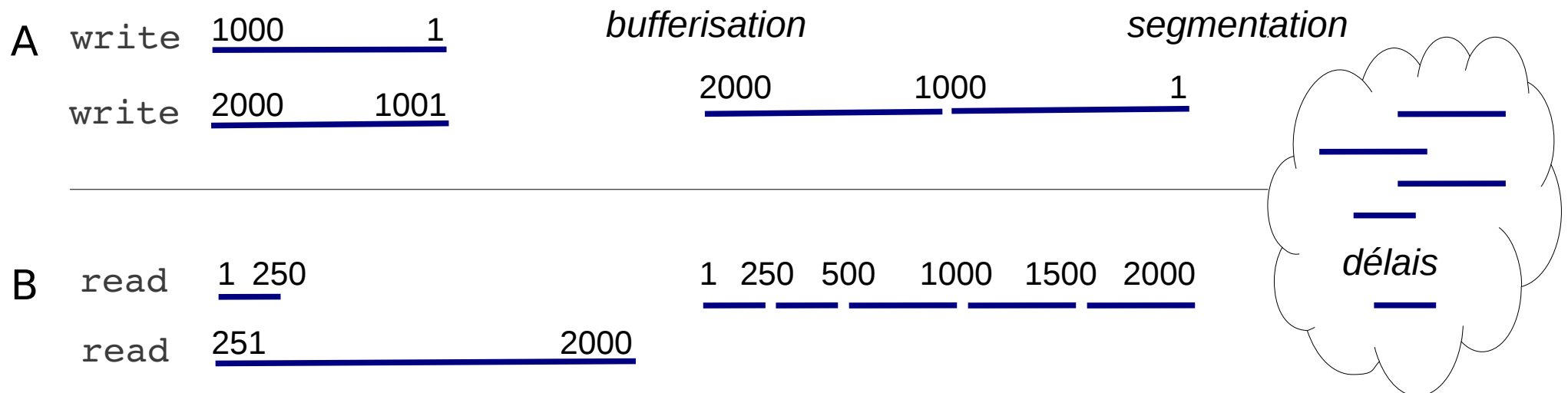
La segmentation TCP

- Contrairement à UDP, TCP ne préserve pas les limites
 - Si A écrit vers B deux fois 1000 octets...
 - Il est possible que TCP découpe ces données en 4, 5 segments ou plus, éventuellement de taille variable
 - Ces segments sont transportés dans des paquets IP indépendants
 - Il est donc possible que B lise tout d'un coup, ou en plusieurs fois...



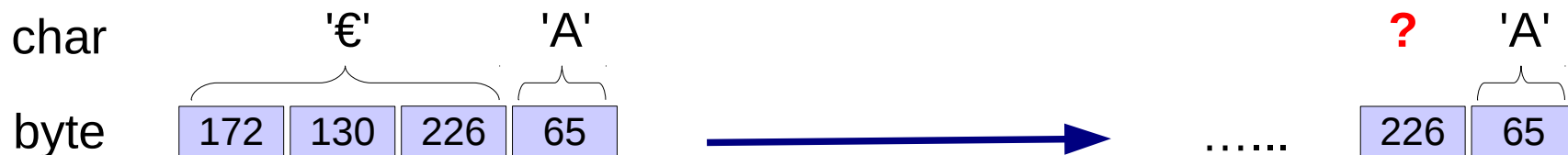
La segmentation TCP

- Contrairement à UDP, TCP ne préserve pas les limites
 - Si A écrit vers B deux fois 1000 octets...
 - Il est possible que TCP découpe ces données en 4, 5 segments ou plus, éventuellement de taille variable
 - Ces segments sont transportés dans des paquets IP indépendants
 - Il est donc possible que B lise tout d'un coup, ou en plusieurs fois...



Segmentation, lecture et écriture

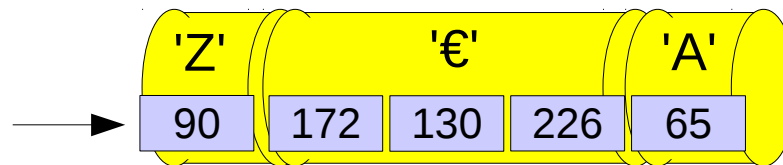
- En java.io, les lectures et écritures sont « bloquantes »
 - Un « write » de 10 octets ne retourne qu'après écriture des 10 octets
 - `byte[] array = new byte[10];`
`write(array);`
`// on est garanti que les 10 octets sont écrits`
 - Un « read » de 10 octets peut retourner dès qu'un seul octet est lu, ou aucun si le flot est fermé (retourne -1)
 - `int nb = read(array); // bloque tant que rien à lire`
`// soit nb vaut -1 (fin de flot)`
`// soit nb <= 10 octets ont été lus`
 - Pour un caractère encodé sur plusieurs octets, seule une partie de ces octets peut être disponible à la lecture à un instant donné



Nécessité de prévoir pour la lecture

- Ces contraintes nécessitent de bufferiser :
 - Soit on peut le faire « à la main », soi-même
 - Soit on utilise des classes des API qui le font ; par exemple
 - DataInputStream : permet de faire des readLong(), readDouble()...
 - Reader (resp. Writer) savent lire (resp. écrire) des char étant donné un encodage.

```
OutputStream os...  
os.write(65);  
os.write(226);  
os.write(130);  
os.write(172);  
os.write(90);
```



```
InputStreamReader isr...  
char[] buf = new char[10]  
int nb = isr.read(buf);  
// nb vaut 3  
// buffer[0] vaut 'A'  
// buffer[1] vaut '€'  
// buffer[2] vaut 'Z'
```

Et de bufferiser les lectures et écritures

- Soit pour des raisons de commodité
 - Par exemple, `BufferedReader` offre une méthode `readLine()` qui :
 - À partir des octets qu'elle est capable de lire dans un flot
 - Les interprète comme étant des char (étant donné un encodage)
 - Les bufferise pour ne retourner une chaîne que lorsqu'elle aura trouvé un caractère de fin de ligne
- Soit pour des raisons d'efficacité
 - Aller lire (ou écrire) un octet sur le réseau est coûteux
 - En accès système / réseau, en recopie de zones de données...
 - Mieux vaut y aller « pour la peine » et en transporter tout un buffer de 1024 ou 4096 d'un coup !
 - Attention : nécessite de faire des `flush()` pour purger ces buffers en écriture

Observations sur les sockets clientes

- Adresse IP et port d'attachement local
 - `getLocalAddress()`, `getLocalPort()`, `getLocalSocketAddress()`
- Adresse IP et port auxquels la socket est connectée
 - `getInetAddress()`, `getPort()`, `getRemoteSocketAddress()`
- Taille des zones tampons
 - `[set/get]SendBufferSize()`, `[set/get]ReceiveBufferSize()`
- Fermeture de la connexion: `close()`, `isClosed()`
 - la fermeture de l'un des flots ferme les deux sens de communication et la socket

Fermeture de socket

- Possibilité de ne fermer qu'un seul sens (*half-closed*)
 - `socket.shutdownOutput()` `isOutputShutdown()`
 - initie le *three way handshake* de déconnexion après l'émission de toutes les données présente dans le buffer d'émission
 - lève une `IOException` à chaque tentative d'écriture
 - `socket.shutdownInput()` `isInputShutdown()`
 - effet local: indication de fin de flot à chaque tentative de lecture
 - effacement après acquittement de toute donnée reçue
 - `close()` non bloquante,
 - mais socket reste ouverte tant qu'il y a des données
 - `setSoLinger(boolean on, int linger_sec)`
 - peut rendre la méthode `close()` bloquante, avec un timeout

Exemple socket cliente

```
// Create a socket and establish a connection with server
Socket s = new Socket(serverName, port);
System.out.println("Connexion established between " +
    s.getLocalSocketAddress() + " and " +
    s.getRemoteSocketAddress());
// Get output stream of the connection (for chars)
BufferedWriter bw = new BufferedWriter(
    new OutputStreamWriter(s.getOutputStream(), "ASCII"));
bw.write("Hello!"); // write the string
bw.newLine();       // write an end of line
bw.flush();         // flush it to the system
// Get input stream of the connection (for chars)
BufferedReader br = new BufferedReader(
    new InputStreamReader(s.getInputStream(), "ASCII"));
// Read a line from server
String line = br.readLine();
System.out.println("Received data : " + line);
// Close the socket, the connection and the streams
s.close();
```

Configurations des sockets clientes

- Gestion données urgentes
 - `sendUrgentData(int octet)` côté émetteur
 - `setOOBInline(boolean on)` Si `true`, replace les données urgentes dans le flot normal côté récepteur (éliminées par défaut, `false`)
- Emission forcée
 - `setTcpNoDelay(boolean on)` pour ne pas temporiser l'émission (débraye l'algorithme de Nagle qui remplit les segments au max.)
- Classe de trafic (*Type Of Service* ou *Flow Label*)
 - `[set/get]TrafficClass()` permet de manipuler:
bits 2 (coût monétaire faible), 3 (haute fiabilité), 4 (haut débit)
et 5 (faible délai). Ex: `s.setTrafficClass(0x02 | 0x10);`
- Etat de la connexion
 - `[set/get]KeepAlive()`: (détection de coupure) `false` par défaut

Socket du côté serveur

- `java.net.ServerSocket`
 - permet d'**attendre** les connexions des clients qui, lorsqu'elles sont établies, sont manipulées par un objet de la classe `Socket`
- Différents constructeurs
 - `ServerSocket()`, puis `bind(SocketAddress sockAddr)` ou `bind(SocketAddress sockAddr, int nbPendantes)`
 - Attachement local de la socket TCP permettant éventuellement de fournir le nombre de connexions pendantes (début du *three way handshake*, mais pas terminé, ou bien connexion **pas encore prise en compte par l'application**)

D'autres constructeurs

- `ServerSocket(int port)`,
`ServerSocket(int port, int nbPen)` ou
`ServerSocket(int port, int nbPen, InetAddress addr)`
- Si port 0, attachement à un port libre
 - nécessité de le divulguer pour que les clients puissent établir des connexions
- Observations sur l'attachement local:
 - `getInetAddress()`, `getLocalPort()` et `getLocalSocketAddress()`

Acceptation de connexion

- `Socket s = serverSock.accept();`
 - méthode bloquante, sauf si `setSoTimeout()` a été spécifié avec une valeur en millisecondes non nulle
 - L'objet `Socket` retourné est dit « **socket de service** » et représente la connexion établie (comme du côté client)
 - On peut récupérer les adresses et ports des deux côtés de la connexion grâce à cet objet socket de service
 - Si plusieurs sockets sont retournées par la méthode `accept()` du même objet `ServerSocket`, ils sont attachés au même port et à la même adresse IP locale
=> démultiplexage sur *(ipCli, portCli, ipSer, portSer)*

Exemple de serveur (pas très utile)

```
// Create and bind a ServerSocket
ServerSocket ss = new ServerSocket(serverPort);
while (true) {
    Socket s = ss.accept(); // wait for incoming connection
    try {
        // Get input and output streams of the client connection
        BufferedReader br = new BufferedReader(
            new InputStreamReader(s.getInputStream(), "ASCII"));
        BufferedWriter bw = new BufferedWriter(
            new OutputStreamWriter(s.getOutputStream(), "ASCII"));
        System.out.println("Received: "+br.readLine());
        // Print the received "Hello" and always send the message
        bw.write("You're welcome!");
        bw.newLine();
        bw.flush();
    } finally {
        try { // Best effort to close the client socket
            s.close();
        } catch (Exception e) {
            // do nothing
        }
    }
}
```

Paramétrage de socket serveur

- Paramètres destinés à configurer les socket clientes acceptées
 - `[set/get]ReceiveBufferSize()`,
`[set/get]ReuseAddress()`
- Modifier l'implantation des sockets
 - `setSocketFactory(SocketImplFactory fac)`

Serveur « itératif »

- « Bloque » tant qu'un client (connexion entrante) n'est pas là
 - Est occupé à 100 % par le traitement de ce client
 - Lorsque ce client est servi (la connexion fermée), peut retourner en attente d'une nouvelle connexion (nouveau client)

```
public void launch() throws IOException {  
    while(!Thread.interrupted()) {  
        // if accept() throws an exception, propagate it: server's down  
        Socket client = serverSocket.accept();  
        System.out.println("New client accepted : "  
                           + client.getRemoteSocketAddress());  
        // serve this client  
        serve(client);  
    }  
}
```

- Si un autre client arrive avant la fin du précédent, il est pris en charge par le système (accepté, mais pas traité)
 - dans certaines limites (**backlog**)

Serveur itératif

- « Bloque » tant qu'un client (connexion entrante) n'est pas là
 - Est occupé à 100 % par le traitement de ce client
 - Lorsque ce client est servi (la connexion fermée), peut retourner en attente d'une nouvelle connexion (nouveau client)

```
public void launch() throws IOException {  
    while(!Thread.interrupted()) {  
        // if accept() throws an exception, propagate it: server's down  
        Socket client = serverSocket.accept();  
        System.out.println("New client accepted : "  
                           + client.getRemoteSocketAddress());  
        // serve this client  
        serve(client);  
    }  
}
```

- Si un autre client arrive avant la fin du précédent, il est pris en charge par le système (accepté, mais pas traité : **en attente**)
 - dans certaines limites (**backlog**)

Serveur concurrent

- Permet de traiter plusieurs client « simultanément »
- Nouvelle thread dédiée à chaque client

```
public void launch() throws IOException {
    while(!Thread.interrupted()) {
        // if accept() throws an exception, propagate it: server's down
        final Socket client = serverSocket.accept();
        System.out.println("New client accepted : "
                           + client.getRemoteSocketAddress());
        // now serve this client in a new dedicated thread
        Thread clientThread = new Thread(new Runnable() {
            @Override
            public void run() {
                serve(client);
                // and let's die this thread
            }
        });
        clientThread.start();
    }
}
```

- Pbme : si beaucoup de clients simultanés ?

Serveur concurrent limité à N clients

- Éviter de créer des threads jusqu'à épuisement des ressources
 - Compter le nombre nb de clients servis dans une thread
 - Si $nb < N$ accepter un client et créer une nouvelle thread
 - Si $nb \geq N$ ne pas accepter de client (mise en attente : `wait()`)
 - Protéger l'accès à nb (concurrency)
 - Décrémenter nb dès que le service du client est fini
 - fin du `run()`, dans un `finally`, attention à la concurrence
 - Si plus de N clients veulent être servis, ceux qui dépassent N sont acceptés et en mis attente par le système
 - Dans la limite du backlog
- Pbme : créer et tuer une thread à chaque nouveau client
 - Coûteux => on aimerait « réutiliser » les threads

Serveur concurrent avec nombre de threads limité

- On peut essayer d'utiliser les Executor, par exemple un `Executors.newFixedThreadPool()`
 - Avantage : les « worker threads » sont réutilisées
 - Inconvénient : la file d'attente associée à l'Executor
 - Elle conduit à accepter systématiquement tout nouveau client et à le mettre en attente dans la queue de l'Executor (explose...)
 - Cela « inhibe » le mécanisme de sécurité du système avec son backlog maximal
 - Solution possible : un Executor ad hoc avec une queue adaptée
- Solution simple : pré-crée le nombre de thread souhaité
 - Toutes en attente d'un client (concurrence d'accès `ServerSocket`)
 - Si N clients sont en cours de traitement, personne de fait `accept()`
 - Les clients surnuméraires sont pris en charge par le système (backlog)

Serveur concurrent à nombre fixe de threads pré-démarrées

```
public void launch() {
    for(int i=0; i<N; i++) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(!Thread.interrupted()) {
                    Socket client = null;
                    synchronized(serverSocket) {
                        try {
                            client = serverSocket.accept();
                        } catch (IOException e) {
                            // server socket is down :-( nothing else to do...
                            e.printStackTrace(System.err);
                            return;
                        }
                    }
                    // now serve this client
                    serve(client);
                }
            }
        }).start();
    }
}
```