Applications réseau
Cours 4

Slides : Damien Imbs

Corentin Travers
corentin.travers@univ-amu.fr

AMU - L3 info

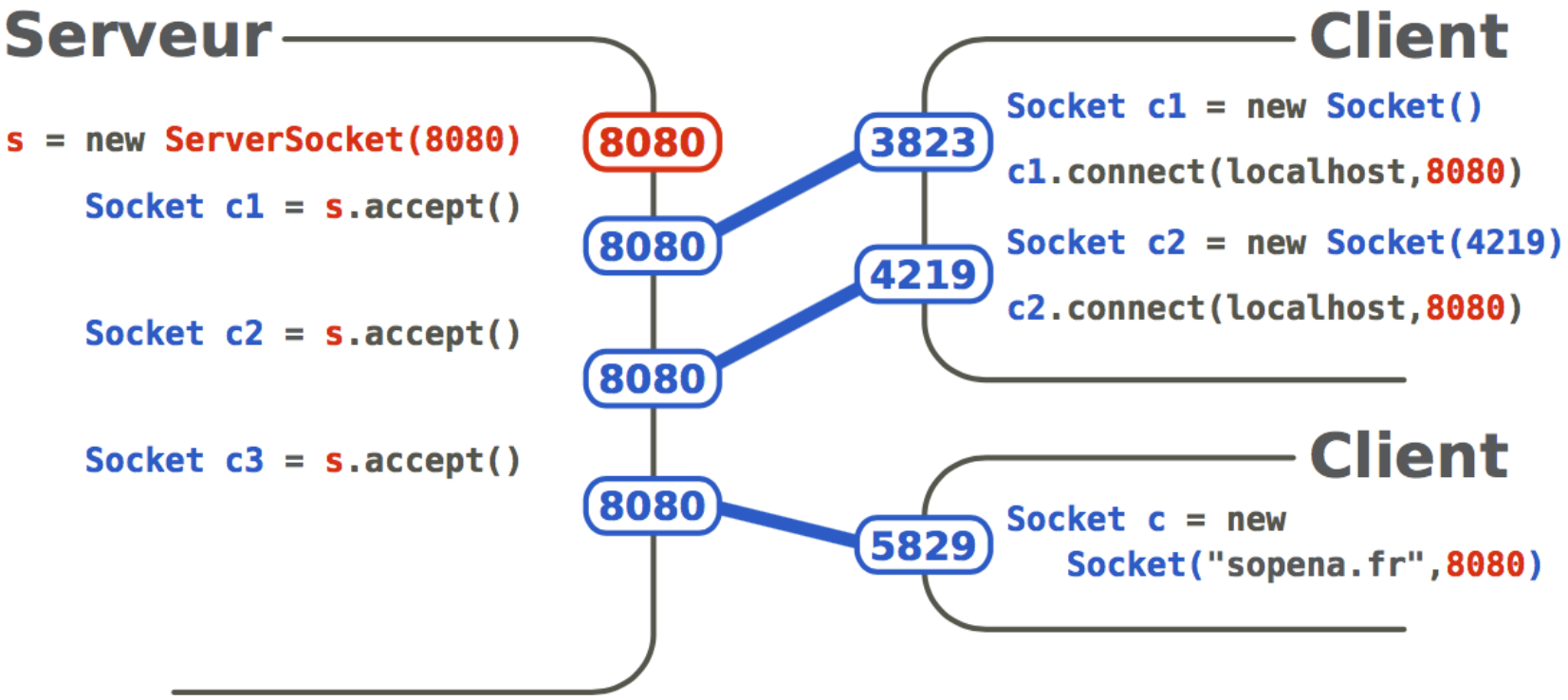
2021-2022

Plan du cours

- Gestion de clients multiples
- Entrées-sorties asynchrones (non bloquantes)

Rappel : Sockets en Java

Connexions multiples sur une socket TCP



Problématique : Communication avec Clients Multiples

Connexions multiples sur une socket TCP

Solution naïve :

```
try{
    input1 = c1.getInputStream();
    input2 = c2.getInputStream();
    input3 = c2.getInputStream();
    // Recevoir des données de clients
    y1 = input1.read();
    y2 = input2.read();
    y3 = input3.read();
    // Traitement de données...
} catch (IOException e) {
// gérer une exception
}
```

Problématique : Communication avec Clients Multiples

Connexions multiples sur une socket TCP

Solution naïve :

```
try{
    input1 = c1.getInputStream();
    input2 = c2.getInputStream();
    input3 = c2.getInputStream();
    // Recevoir des données de clients
    y1 = input1.read();
    ...
```

`read()` est bloquant

Si le client c1 n'est pas prêt, les autres clients doivent attendre !

Problématique : Communication avec Clients Multiples

Les méthodes pour communiquer avec un client ne doivent pas bloquer la communication avec les autres clients !

Trois solutions possibles :

- Vérifier périodiquement si les données sont prêtes à être lues (attente active)
- Utiliser des I/O asynchrones (non bloquantes)
- Paralléliser les tâches (Multi-Thread)

Une solution simple : l'attente active

`soc[]` : tableau de sockets clients déjà connectés

```
While (true) {
    For (i=0; i < Soc.size(); i++)
        If (soc[i] est prêt)
            Transfert de données avec soc[i];
}
```

Principe de l'**attente active :**
Tester continuellement si une action peut être effectuée.

Une solution simple : l'attente active

Principe de l'**attente active :**
Tester continuellement si une action peut être effectuée.

En Java :

- Classe `InputStream` :
  `int available()` renvoie le nombre d'octets pouvant être lus
- Classe `Reader` :
  `boolean ready()` indique s'il est possible de lire quelque chose (au moins 1 caractère)

Une solution simple : l'attente active

Principe de l'**attente active :**
Tester continuellement si une action peut être effectuée.

Défaut : le processeur peut travailler inutilement pendant un temps indéterminé.

I/O Multiplexées

Java NIO

Autre possibilité : entrées-sorties non bloquantes

- Lancer une lecture ou écriture sans attendre sa fin
- Traiter l'opération plus tard, quand elle est finie

En Java : Java NIO (New Input-Output)

Concepts clés :

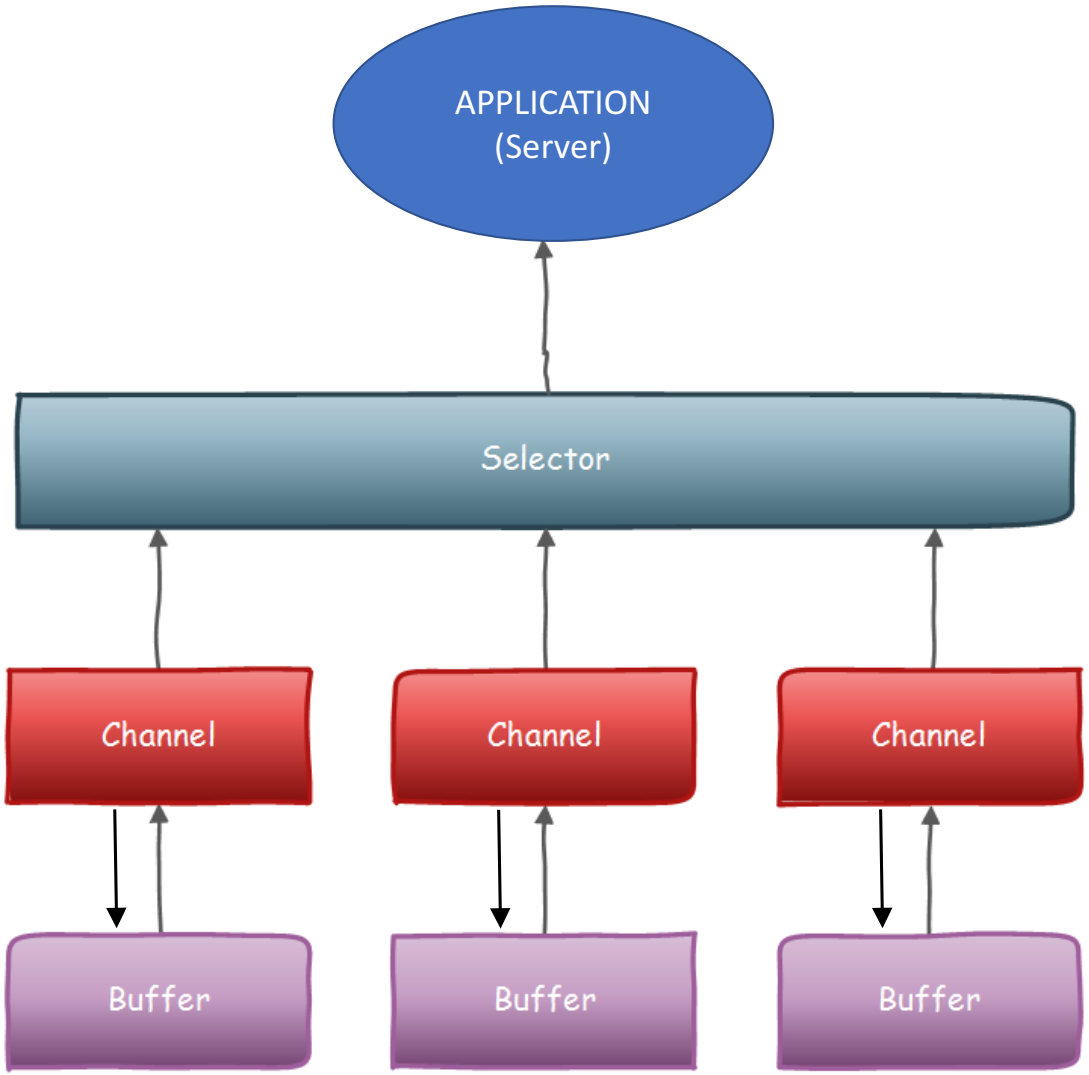
- Buffer (tampon)
- Channel (canal)
- Selector (selecteur)

Utilisation :

```
import java.nio.*
```

Java NIO

Vision d'ensemble



Buffers (tampons)

Buffer = zone de mémoire contiguë, permet de stocker

- une quantité fixe de données
- d'un type primitif (byte, char, short, int, long, float ou double)

Représenté en Java par des classes abstraites
L'implémentation dépend de l'OS.

Buffers (tampons)

Utilisation

Allocation d'un buffer de type primitif **type** :

Type`Buffer.allocate(int capacity)`

Création à partir d'un tableau d'éléments de type **type** :

Type`Buffer.wrap(`**type** `array[])`

Deux types d'accès :

- aléatoire :
  permet d'accéder à n'importe quel indice
  - **type** `get(int index)` renvoie l'élément à l'indice `index` du buffer
  - **type** `put(int index,` **type** `value)` écrit l'élément à l'indice `index` du buffer
- séquentiel :
  permet de consommer des données selon l'ordre d'arrivée
  - **type** `get()` renvoie le prochain élément non consommé (position courante)
  - **type** `get(`**type** `value)` ajoute un élément à la position courante

Attributs et méthodes d'un tampon

- **Capacité** : nombre d'éléments qui peuvent être contenus
  - Fixée à la création du tampon
  - Consultable par int capacity()
- **Limite** : indice du premier élément ne devant pas être atteint
  - Par défaut, égale à la capacité.
  - Fixée par `Buffer limit(int newLimit)`
  - Connue par `int limit()`
- **Position courante** : indice du prochain élément accessible
  - Consultable : `int position()`
  - Modifiable : `Buffer position(int newPosition)`

Attributs et méthodes d'un tampon

Marque (éventuelle) : position dans le tampon

- `Buffer mark()` place la marque à la position courante
- `Buffer reset()` place la position à la marque ou lève InvalidMarkException
- La marque est toujours inférieure à la position.
  - Si la position ou la limite deviennent plus petite que la marque, la marque est effacée

Invariant :
0 <= marque <= position <= limite <= capacité

`Buffer rewind()`
met la position à 0 et supprime la marque

Attributs et méthodes d'un tampon

Quand la position courante vaut la limite :

- Un appel à get() provoque `BufferUnderflowException`
- Un appel à put() provoque BufferOverflowException

Pour éviter ça :

- `int remaining()` donne le nombre d'éléments entre la position courante et la limite
- `boolean hasRemaining()` vaut vrai si la position est strictement inférieure à limite

Méthodes utilitaires sur les tampons

`compact()` :

- Place l'élément à la position courante p à la position 0, l'élément $p + 1$ à la position 1, etc. La nouvelle position courante est placée après le dernier élément décalé. La limite est mise à la capacité et la marque effacée.

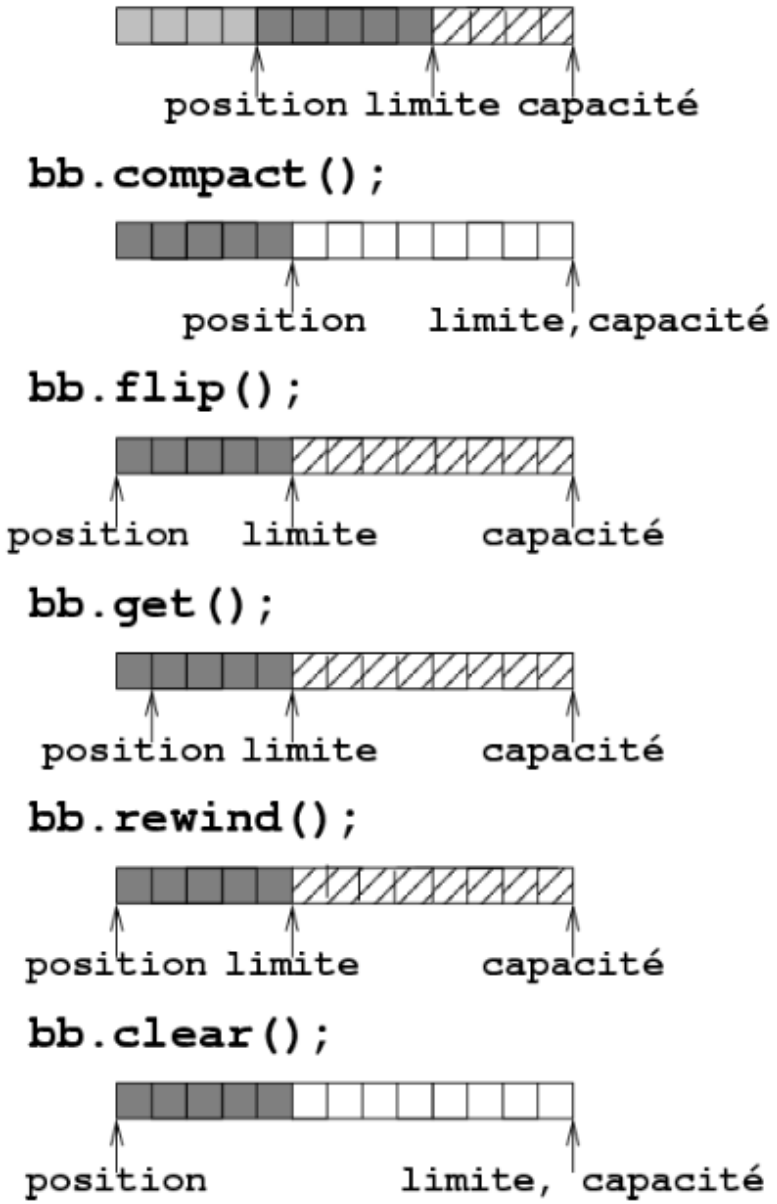
`flip()` :

- limite <- position courante
- position <- 0
- Marque indéfinie.

`rewind()`

- position <- 0
- Marque indéfinie.

`clear()` N'efface pas le contenu !



Canaux (channels)

Représentent des connexions ouvertes vers des entités capables d'effectuer des opérations d'entrées-sorties comme des fichiers, des sockets ou des tubes.

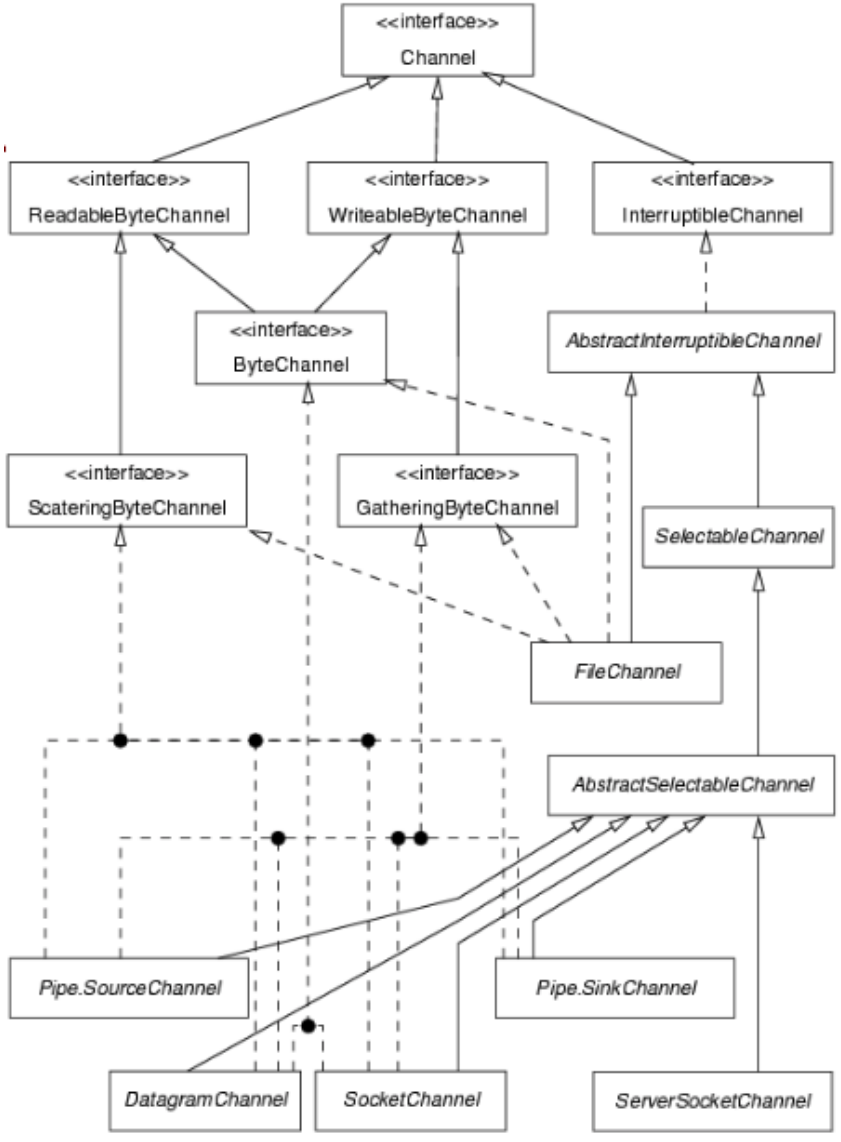
- Un canal est ouvert à sa création.
- Il ne peut plus être utilisé une fois qu'il est fermé.
- À la différence des flots, il peut être utilisé en mode bloquant ou non bloquant (on y reviendra plus tard).
  - En mode non bloquant, toutes les lectures/écritures retournent immédiatement, même si rien n'est lu ou écrit.
  - On peut alors utiliser un sélecteur pour attendre en même temps la possibilité d'effectuer des entrées-sorties sur différents canaux.

Canaux (channels)

Classes et interfaces importantes

java.nio.channels.*

- `Channel`
  - `close(), isOpen()`
- `ReadableByteChannel`
  - `int read(ByteBuffer)`
    - Tente de lire au plus `remaining()` octets.
- `WritableByteChannel`
  - `int write(ByteBuffer)`
    - Tente d'écrire au plus `remaining()` octets
- `ByteChannel`
  - Hérite des deux



Exemple : canal associé à une Socket

java.nio.SocketChannel (une connexion TCP)

- `SocketChannel sc = SocketChannel.open()` puis `sc.connect(InetSocketAddress server)`
- Ou plus directement
  `SocketChannel sc = SocketChannel.open(InetSocketAddress server)`

Mais aussi `bind()` et d'autres méthodes de Socket
Plus généralement, `sc.socket()` permet de récupérer l'objet de la classe Socket sous–jacent.

Lecture et écriture dans SocketChannel

Lecture depuis la connexion

- ByteBuffer bb = . . . int n = sc.read(bb) ;
- lit au plus bb.remaining() octets et retourne ce nombre *n*, ou 1 si le flot de lecture est fermé

Ecriture dans la connexion

- int n = sc.write(bb) ;
- Ecrit exactement bb.remaining() octets et retourne ce nombre *n*

Les deux peuvent lever *IOException*

Canal associé à une ServerSocket

java.nio.ServerSocketChannel (socket serveur d'écoute TCP)

- `ServerSocketChannel ssc = ServerSocketChannel.open()` puis `ssc.bind(SocketAddress localSocketAddress)`
- Plus généralement, `ssc.socket()` permet de récupérer l'objet de la classe `ServerSocket` sous–jacent
- Attente de connexion client : `SocketChannel sc = ssc.accept()`

UDP via les canaux

java.nio.channels.DatagramChannel

- Canal vers une socket UDP
- On peut créer une java.net.DatagramSocket à partir d'un DatagramChannel, mais pas le contraire
  - Si une socket UDP `su` a été créée à partir d'un canal, on peut récupérer ce canal par `su.getChannel()`. Sinon, cette méthode retourne null.
- `DatagramChannel.open()` crée et retourne un canal associé à une socket UDP (non attachée)

DatagramChannel n'est pas une abstraction complète des sockets UDP : pour les opérations précises (binding, etc. . .) on récupère l'objet DatagramSocket sous–jacent

DatagramChannel

- Par défaut, un canal `dc` récupéré par `DatagramChannel.open()` est bloquant. Il peut être configuré non bloquant.
  - `dc.socket()` récupère alors la `DatagramSocket` correspondante
  - Elle n'est pas attachée. On peut faire `bind(SocketAddress)` sur cette socket.
- Les méthodes `send()` et `receive()` sont accessibles depuis le canal
  - Elles manipulent des `ByteBuffer` et `receive()` retourne un objet `SocketAddress` identifiant l'émetteur des données reçues
- On doit faire une pseudo–connexion pour pouvoir utiliser les méthodes `read()` et `write()` avec des `ByteBuffer`, plus classiques sur les canaux (interlocuteur implicite pour ces méthodes)

Envoi sur un DatagramChannel

`int send(ByteBuffer src, SocketAddress target)`

- Provoque l'envoi des données restantes du tampon `src` vers `target`
- Semblable à un `write()` du point de vue du canal
- Si **canal bloquant**, la méthode retourne lorsque tous les octets ont été émis (leur nombre est retourné)
- Si **canal non bloquant**, la méthode émet tous les octets ou aucun
- Si une autre écriture est en cours sur la socket par une autre thread, l'invocation de cette méthode bloque jusqu'à ce que la première opération soit terminée

Réception depuis un DatagramChannel

`SocketAddress receive(ByteBuffer dst)`

- Par défaut, méthode bloquante tant que rien n'est reçu par la socket
- Au plus `dst.remaining()` octets peuvent être reçus. Le reste est tronqué.
- Retourne l'adresse de socket (IP+port) de l'émetteur des données
- Si canal non bloquant, soit tout est reçu, soit le tampon n'est pas modifié et la méthode retourne `null`

Exemple de client UDP avec canaux : envoi

```
// Récupération de l'adresse IP et du port
InetAddress server = InetAddress.getByName(args[0]);
int port = Integer.parseInt(args[1]);
InetSocketAddress isa = new InetSocketAddress(server, port);

// Création d'un objet canal UDP non attaché
DatagramChannel dc = DatagramChannel.open() ;

// Les données à envoyer doivent être dans un ByteBuffer
ByteBuffer bb = ByteBuffer.wrap("Hello".getBytes("ASCII"));

// L'attachement de la socket UDP sous-jacente est implicite
dc.send(bb,isa) ;

// si j'attends une réponse, je ne ferme pas le canal...
```

Exemple de client UDP avec canaux : réception

```
// pour me préparer à recevoir la réponse...
// j'alloue une zone de données de taille suffisante
ByteBuffer bb = ByteBuffer.wrap(new byte[512]);

// la réception place les données dans la zone de données et
// retourne la socket représentant l'émetteur
SocketAddress sender = dc.receive(bb) ;

bb.flip();
System.out.println(bb.remaining()+" octets ont été reçus de "
                   +sender+":");
System.out.println(new String(bb.array(), 0, bb.remaining(),
                   "ASCII"));

// L'échange est fini: je ferme (définitivement) le canal...
dc.close();
```

Canaux à mode non bloquant

Lecture et écriture ne bloquent jamais
Le nombre d'octets transférés peut être inférieur à l'indication

- Eventuellement nul, en lecture comme en écriture

Tous les canaux standards peuvent être non bloquants SAUF :

- Ceux obtenus par la classe utilitaire Channels
- Les canaux sur les fichiers

A leur création, les canaux sont en mode bloquant

- Changement de mode par `configureBlocking(false)`
- Consultation du mode actuel par `isBlocking()`

Ils sont les seuls à pouvoir être multiplexés avec un sélecteur de la classe java.nio.channels.Selector

ServerSocketChannel

Appeler `accept()` sur le canal pour accepter des connexions (la server socket sous–jacente doit être attachée)
Si le canal est bloquant, l'appel bloque

- Retourne un SocketChannel quand la connexion est acceptée ou
- Lève une IOException si une erreur d'entrée–sortie arrive

Si le canal est non bloquant, retourne immédiatement

- `null` s'il n'y a pas de connexion pendante

Quel que soit le mode (bloquant / non bloquant) du `ServerSocketChannel` le `SocketChannel` retourné est initialement en mode bloquant

SocketChannel

- `boolean connect(SocketAddress remote)` sur le canal
- Si `SocketChannel` en mode bloquant, l'appel à `connect()` bloque et retourne `true` quand la connexion est établie, ou lève `IOException`
- Si `SocketChannel` en mode non bloquant, l'appel à connect()
  - Peut retourner `true` immédiatement (connexion locale, par exemple)
  - Retourne `false` le plus souvent : il faudra plus tard appeler `finishConnect()`
- Tant que le canal est non connecté, les opérations d'entrée/sortie lèvent `NotYetConnectedException` (peut être testé par `isConnected()`)
- Un `SocketChannel` reste connecté jusqu'à ce qu'il soit fermé

Etablissement de connexion

Méthode `finishConnect()`

- Si connexion a échoué, lève `IOException`
- Si pas de connexion initiée, lève `NoConnectionPendingException`
- Si connexion déjà établie, retourne immédiatement true
- Si la connexion n'est pas encore établie
  - Si mode non bloquant, retourne false
  - Si mode bloquant, l'appel bloque jusqu'au succès ou à l'échec de la connexion (retourne true ou lève une exception)
- Si cette méthode est invoquée lorsque des opérations de lecture ou d'écriture sur ce canal sont appelés
  - Ces derniers sont bloqués jusqu'à ce que cette méthode retourne
- Si cette méthode lève une exception (la connexion échoue)
  - Alors le canal est fermé

Communication sur canal de socket TCP

Une fois la connexion établie, le canal de socket se comporte comme un canal en lecture et écriture
Méthodes `read()` et `write()`

- En mode bloquant, `write()` assure que tous les octets seront écrits mais `read()` n'assure pas que le tampon sera rempli (au moins un octet lu ou détection de fin de connexion : retourne 1)
- En mode non bloquant, lecture comme écriture peuvent ne rien faire et retourner 0.

La fermeture de socket par `close()` entraine la fermeture du canal.

Les sélecteurs

Objets utilisés avec les canaux configurés en mode **non bloquant**

- Ces canaux peuvent être enregistrés auprès d'un sélecteur après avoir été configurés non bloquant (`configureBlocking(false)`)
- `java.nio.channels.Selector` : les instances sont créées par appel à la méthode statique `open()` et fermés par `close()`

Les sélecteurs

Enregistrement d'un canal auprès d'un sélecteur

- Se fait par un appel, sur le canal à enregistrer, de :
- SelectionKey register(Selector sel, int ops) ou
- SelectionKey register(Selector sel, int ops, Object att)
  - ops représente les opérations "intéressantes" pour ce sélecteur
  - SelectionKey retourné représente la **clé de sélection** de ce canal, qui permet de connaître, outre le sélecteur :
    - channel() renvoie le canal lui même
    - interestOps() renvoie les opérations "intéressantes"
    - attachment() renvoie l'objet att éventuellement attaché

Autour des sélecteurs

- keys() appelé sur un sélecteur retourne toutes ses clés
  - SelectionKey keyFor(Selector sel) sur un canal donne la clé de sélection du sélecteur pour ce canal
- On peut enregistrer plusieurs fois un même canal auprès d'un sélecteur (il met la clé à jour)
  - Il est plus élégant de modifier sa clé de sélection
    - En argument de interestOps() ou de attach() sur cette clé de sélection
- Les opérations intéressantes sont exprimées par les bits d'un entier (faire des OU binaires (|))
  - SelectionKey.OP_READ, SelectionKey.OP_WRITE, SelectionKey.OP_CONNECT, SelectionKey.OP_ACCEPT
  - Etant donné un canal, validOps() renvoit ses opérations " valides "

Utilisation des sélecteurs

Appel à la méthode `select()`

- Entraîne l'attente passive d'événements intéressants pour les canaux enregistrés
- Dès qu'une de ces opérations peut être effectuée, la méthode retourne le nombre de canaux sélectionnés
- Elle ajoute également les clés de sélection de ces canaux à l'ensemble retourné par `selectedKeys()` sur le selecteur
  - Il suffit de le parcourir avec un itérateur

C'est à l'utilisateur de retirer les clés de sélection correspondant aux canaux sélectionnés qu'il a "utilisé"

- méthode `remove()` de l'ensemble ou de l'itérateur ou
- méthode `clear()` de l'ensemble qui les retire toutes

Si une clé est intéressée par plusieurs opérations

- `readyOps()` donne celles qui sont prêtes
- raccourcis `isAcceptable()`, `isConnectable()`, `isReadable()` et `isWritable()`

Sélection bloquante ou non bloquante

- Les méthodes `select()` ou `select(long timeout)` sont bloquantes
  - Elles ne retournent qu'après que
    - un canal soit sélectionné ou
    - la méthode `wakeup()` soit appelée ou
    - la thread courante soit interrompue ou
    - le timeout ait expiré
- La méthode `selectNow()` est non bloquante
  - Retourne 0 si aucun canal n'est sélectionné
- L'annulation de l'enregistrement d'un canal auprès d'un sélecteur peut se faire par cancel() sur la clé de sélection. Elle est aussi réalisée implicitement à la fermeture du canal.

Code type d'un serveur non bloquant

```
public void launch() throws IOException {
// registers the server socket for 'accept' operations
serverSocket.register(selector,SelectionKey.OP_ACCEPT);
// retrieves the selectedKeys set reference
Set<SelectionKey> selectedKeys = selector.selectedKeys();
while(true) {
  // blocking select operation, until there is something to do
  selector.select();
    for(SelectionKey key : selectedKeys) {
      if(key.isAcceptable()) {
        doAccept(key);
      }
      if(key.isValid() && key.isWritable()) {
        doWrite(key);
      }
      if(key.isValid() && key.isReadable()) {
        doRead(key);
      }
    }
  }
}
```

Programmation Multi-thread

Autre possibilité : utiliser plusieurs threads

Threads et Processus

- Un thread est processus "léger" à l'intérieur d'un processus
- Chaque thread contient son propre espace d'adressage
- Les threads issus d'un même processus partagent la même zone mémoire (et les descripteurs fichier/socket)
- Possible d'augmenter la productivité par l'exécution concurrente de threads.

Pourquoi utiliser des threads pour implementer un serveur ?

Les threads Java

Exemple simple :

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Bonjour!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Les threads Java

Interface Runnable

```
public class Handler implements Runnable {
    public void run() {
        System.out.println("Bonjour!");
    }
    public static void main(String args[]) {
        (new Handler()).run();
    }
}
```

Communication avec Clients Multiples

Serveur Multi-Thread

Code Serveur :

```
ServerSocket ssocket; // socket d'écoute
ssocket = new ServerSocket(port);
```

Créer un thread par connexion client :

```
 /* attendre indéfiniment les connexions */
while (true) {
    (new Handler(ssocket.accept())).run();
}
```

Programmation Multi-thread

- Le processus serveur peut créer plusieurs Threads (Un serveur web reçoit des millions de requêtes clients...)
- Que passe-t-il quand un Thread a fini son exécution ? (Un thread terminé reste dans la mémoire)

- Ce n'est pas efficace de créer trop de threads en parallèle.
- Il y a une limite sur le nombre de threads (imposée par le système)
- Il faut réutiliser les threads terminés

Programmation Multi-thread

Java Executer

Comment gérer les Threads dans un Processus ?

```
import java.util.concurrent.*;
Executor e =
        Executors.newSingleThreadExecutor();
e.execute(new Handler(ssocket.accept()));
```

Executor est une interface Java avec la méthode `execute()`

Executer Service

Java.util.concurrent.ExecuterService

Pour définir la police d'exécution des threads

- Choisir un thread pour chaque tâche
- Choisir l'ordre d'exécution de tâches
- Combien de threads peuvent s'exécuter simultanément
- Combien de tâches peuvent être en attente
- Quelles tâche à rejeter si le système est surchargé
- Actions a poursuivre avant / après l'exécution d'une tâche

Executers pour Threads

Java.util.concurrent.Executors

Exécuteurs prédéfinis en Java

- Executors.newFixedThreadPool()
  // Nombre de Threads fixée
- Executors.newCachedThreadPool()
  // Nombre de Threads variable
- Executors.newSingleThreadExecutor()
  // Un seule Thread a la fois
- Executors.newScheduledThreadPool()
  // activer les threads à une heure précise

Thread Pool : exemple

Un serveur qui utilise 10 Threads en parallèle :

```
private final Executor executor;
/* création du pool de 10 threads */
executor = Executors.newFixedThreadPool(10);
try {
    ssocket = new ServerSocket(port);
    while (true) {
      /* attendre les connexions entrantes */
      executor.execute(new Handler(ssocket.accept()
    }
} catch (IOException ex) { ... }
```