Travail Pratique : Framework (cadre d'application)

Vous devez réaliser ce laboratoire avec la **version 7 de Java. Assurez-vous que votre IDE est bien configuré pour être compatible avec Java 7**.

Inscrivez vos réponses dans les classes appropriées provenant de cette archive et remplissez le document Framework.doc. Votre remise consistera d'une archive .zip des fichiers complétés. Veuillez inclure une page titre au document Framework.doc.

La remise se fera par *Moodle*, les instructions sont disponibles sur le site du cours.

Les outils NIO de Java

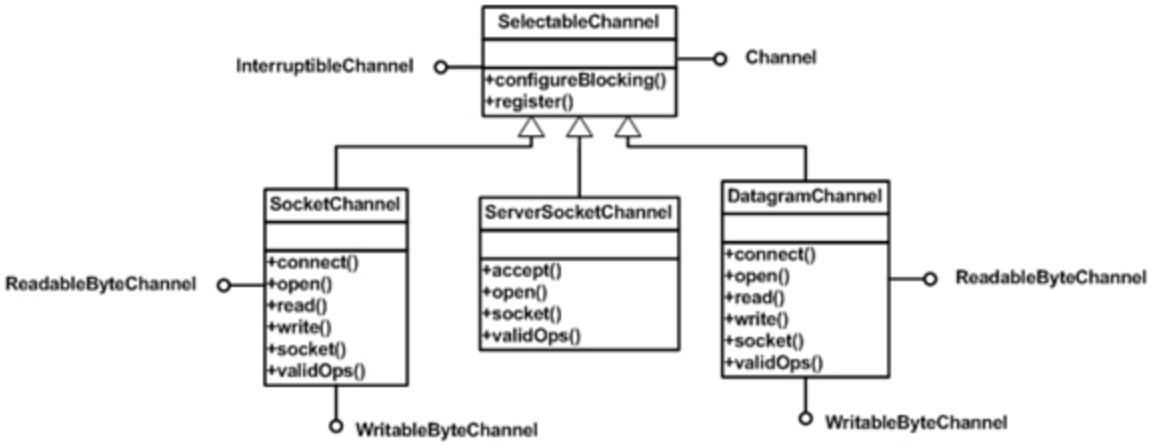
Avant de commencer avec les cadres d'applications (*framework*), commençons par une introduction aux outils NIO[1] Java. La terminologie peut être mélangeante ici, car plusieurs vont dirent que NIO est l'acronyme pour "*non-blocking IO*", ce qui n'est pas le cas. Il ne s'agit pas vraiment IO non bloquante, mais d'outils permettant de réaliser un modèle de type *plusieurs connexions par Thread* de façon efficace à l'aide de mécanismes provenant du système d'exploitation.

Cette introduction est requise, car même si les cadres d'applications offrent le choix entre des opérations synchrones et asynchrones ainsi que divers modèles pour les liens entre les *Threads* et les connexions, leurs implémentations se basent sur des mécanismes similaires à ceux de NIO. Cette introduction sert également à démontrer que le niveau de complexité et de difficulté augmente considérablement lorsque vous utilisez ce genre de mécanismes (bonne chance).

Le Canal (Channel)

Les canaux encapsulent les notions liées aux *Sockets* et à la gestion des *Input* et *Output* streams.

[1] Voici plusieurs bons liens pour NIO
- http://docs.oracle.com/javase/1.5.0/docs/guide/nio/index.html (doc officielle)
- http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf (Doug Lea)
- https://today.java.net/pub/a/today/2007/02/13/architecture-of-highly-scalable-nio-server.html (Java.net)
- http://tutorials.jenkov.com/java-nio/index.html (Très détaillé)



Tiré de http://www.developer.com/java/article.php/10922_3837206_3/An-Introduction-to-Java-NIO-and-NIO2.htm

Par exemple, pour créer un canal adapté à un **ServerSocket**, nous procédons comme suit

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
ServerSocket serverSocket = serverChannel.socket();
InetSocketAddress address = new InetSocketAddress(PORT);
serverSocket.bind(address);
```

ATTENTION! N'oubliez pas de réutiliser votre utilitaire (**IOUtil**) pour bien libérer les ressources.

Présentement, nous sommes encore synchrones, mais en appelant `serverChannel.configureBlocking(false)`, vous balancez en mode asynchrone.

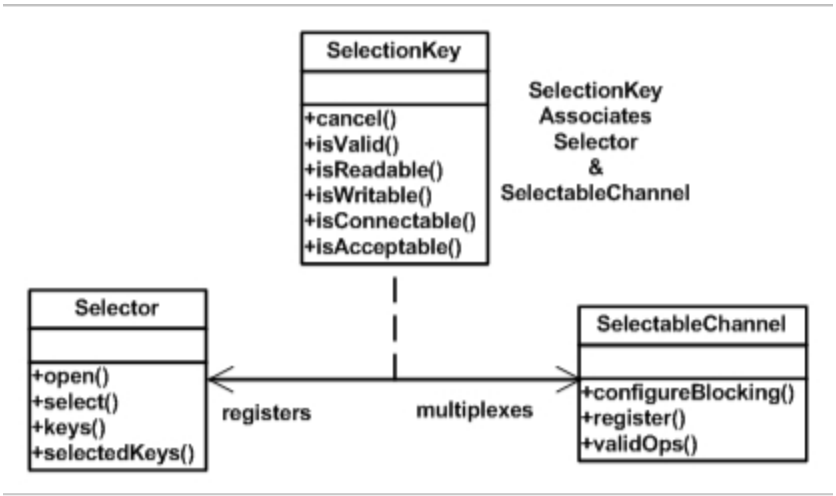
Le Selector

Jusqu'à présent, nous avons travaillé avec des *Socket*s dans un modèle de type *un Thread par connexion*. Ce modèle ne s'adapte pas très bien lorsque le nombre de connexions augmente considérablement. Par exemple, si votre serveur doit desservir 200 à 300 connexions par seconde, instancier et désinstancier 200 à 300 *Threads* par secondes demande beaucoup de ressources.

Il existe plusieurs mécanismes pour implémenter une solution du type *plusieurs connexions par Thread*. Nous allons utiliser un nouveau paradigme, les *Selectors*.

Vous serez surpris d'apprendre que les *Selectors* sont bloquants. Cependant, les *Selectors* ne bloquent pas de la même façon que les *Sockets*. Le *Selector* a la particularité de bloquer sur un événement indépendamment de la connexion. Par exemple, un seul *Selector* peut

gérer à la fois 1000 connexions.



Tiré de http://www.developer.com/java/article.php/10922_3837206_3/An-Introduction-to-Java-NIO-and-NIO2.htm

Après avoir créer votre *Selector* vous n'avez qu'à l'enregistrer au canal.

```
Selector selector = Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

Nous venons d'enregistrer notre *Selector* à l'événement *Accept* de notre canal. Ceci implique que nous serons informés de chaque connexion sur notre **ServeurSocket**. Les événements disponibles sont

- `OP_ACCEPT`— Si le *Selector* détecte que le **ServerSocket** du canal est prêt à recevoir une nouvelle connexion ou qu'il y a une erreur avec le **ServerSocket**.
- `OP_CONNECT`— Si le *Selector* détecte que le *Socket* du canal est prêt à compléter la séquence de connexion, ou qu'il y a une erreur avec le *Socket*.
- `OP_READ`— Si le *Selector* détecte que le canal est prêt pour la lecture, qu'il a été fermé ou est en erreur.
- `OP_WRITE`— Si le *Selector* détecte que le canal est prêt pour l'écriture, qu'il a été fermé ou est en erreur.

Ayant enregistré notre *Selector* sur l'événement *Accept*, la méthode `Selector.select()` va bloquer jusqu'à ce qu'une connexion soit reçue.

Il est possible que vous soyez confus, mais la prochaine passe est cruciale et va vous permettre de comprendre le fonctionnement des *Selectors*, soyez attentif. Lorsqu'une connexion est reçue, nous allons enregistrer à nouveau le *Selector* au canal, mais cette fois en lecture et/ou en écriture (il s'agit du patron Acceptor-Connector).

```
if (key.isAcceptable()) {
        ServerSocketChannel server = (ServerSocketChannel) key.channel();
        SocketChannel client = server.accept();
        client.configureBlocking(false);
        client.register(selector, SelectionKey.OP_READ, ByteBuffer.allocate(100));
}
```

Dans l'extrait précédent, la variable **key** est un objet de type **SelectionKey**, que nous obtenons du *Selector* à l'aide de la méthode `selectedKeys()`. Un **key** est lié à un événement et va nous fournir le contexte de cet événement (canal, information propre à l'événement...).

Maintenant, s'il y a du contenu à lire lié à cette connexion, le *Selector* en sera informé. Pour définir le comportement désiré dans cette situation, il suffit d'ajouter l'extrait suivant.

```
if (key.isReadable()) {
        SocketChannel client = (SocketChannel) key.channel();
        ByteBuffer output = (ByteBuffer) key.attachment();
        client.read(output);
    ...
        output.clear();
}
```

ATTENTION! Vous devez appeler la méthode output.clear(), car sinon la prochaine fois, vous allez relire le contenu une autre à nouveau. L'objet ByteBuffer est également propre à NIO, il s'agit d'un objet très utile qui encapsule un tableau d'octets.

Exercice #1 (UTFServerNIO.java)

Réimplémentez le **UTFServer** (du TP Socket 101) à l'aide de NIO (*Channel*, *Selector*...).

Indice: N'oubliez pas de fermer toutes vos ressources avec votre utilitaire.

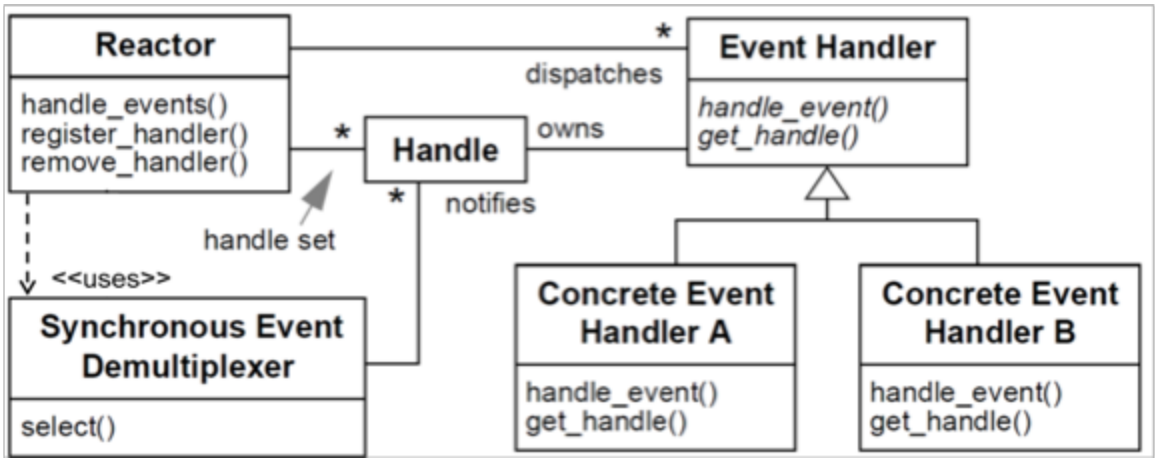
Exercice #2 (Framework.doc)

Testez le **UTFServer** et le **UTFServerNIO** avec plusieurs clients en même temps. Décrivez la différence comportementale entre **UTFServer** et **UTFServerNIO**.

Les outils NIO.2 de Java

Quoique cela peut vous paraitre nouveau, il n'y a rien de nouveau, le paradigme du *Selector* est vieux comme le monde (*UNIX System V* 1983) et NIO date de J2SE 1.4 (2002). En 2011, avec la sortie de Java SE 7, un nouveau module NIO, nommé NIO.2 est maintenant disponible dans la plateforme Java.

NIO.2 peut sembler plus complexe que NIO, mais vous êtes gagnant au change, car il élimine une grande partie du code requis pour établir la mécanique de NIO. Le gros changement est l'ajout du patron *Reactor*, qui vous offre des objets **CompletionHandler**. Ceux-ci sont appelés lorsque le *Selector* détecte un évenement.



Tiré de http://www.dre.vanderbilt.edu/~schmidt/

Par exemple, je peux me déclarer un **AcceptCompletionHandler** (conforme au patron Acceptor-Connector)

```
class AcceptCompletionHandler implements
        CompletionHandler<AsynchronousSocketChannel, ByteBuffer> {

        private final AsynchronousServerSocketChannel serverChannel;

        public AcceptCompletionHandler(AsynchronousServerSocketChannel channel) {
                this.serverChannel = channel;
        }

        @Override
        public void completed(AsynchronousSocketChannel channel,
                        ByteBuffer attachment) {
                serverChannel.accept(null, this);
```

```
                ByteBuffer buffer = ByteBuffer.allocate(100);
                channel.read(buffer, buffer, new ReadCompletionHandler(channel));
        }

    ...
}
```

La méthode `completed()` sera appelée, si le *Selector* détecte que le **ServerSocket** du canal est prêt à recevoir une nouvelle connexion ou qu'il y a une erreur avec le **ServerSocket**. Dans l'exemple précédent, je ne gère pas l'erreur. On constate qu'un **ReadCompletionHander** est enregistré à l'opération `read()` de canal.

ATTENTION! La méthode `read()` ici est asynchrone! La méthode `read()` ne bloque pas jusqu'à ce que les données soient ajoutées au buffer, elle retourne immédiatement, la méthode `complete()` de l'objet **ReadCompletionHandler** sera appelée lorsque la lecture sera terminée.

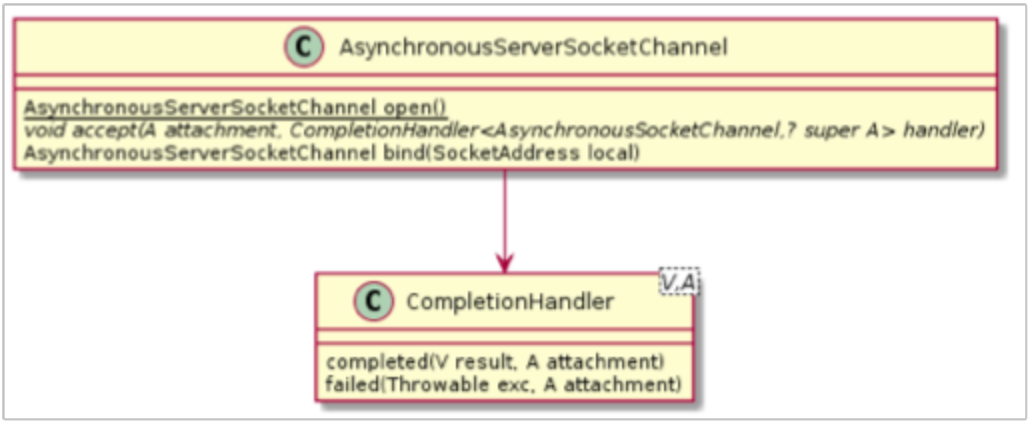
Voici une partie de la classe **ReadCompletionHandler**

```
class ReadCompletionHandler implements
        CompletionHandler<Integer, ByteBuffer> {

    private final AsynchronousSocketChannel channel;

    public ReadCompletionHandler(AsynchronousSocketChannel channel) {
        this.channel = channel;
    }

    @Override
    public void completed(Integer result, ByteBuffer buffer) {
        if (result != -1) {
                ...
                channel.read(buffer, buffer, this);
         }
    }
}
```

Rappelez-vous que la méthode `completed()` est appelée, car l'appel précédent de la méthode `read()` a terminé, on vérifie que la lecture s'est bien déroulée et on effectue le traitement. Par la suite, nous invoquons à nouveau la méthode `read()` et le même objet est utilisé pour gérer le résultat de la prochaine lecture.

Tout ce qui reste est d'assembler les morceaux, le canal NIO.2 est de type **AsynchronousServerSocketChannel** on effectue le *bind* de la même façon que pour NIO, soit

```
AsynchronousServerSocketChannel serverChannel = AsynchronousServerSocketChannel.open();
InetSocketAddress address = new InetSocketAddress(PORT);
serverChannel.bind(address);
serverChannel.accept(null, new AcceptCompletionHandler(serverChannel));
```



ATTENTION! La méthode `accept()` ici est asynchrone! La méthode `accept()` ne bloque pas jusqu'à la réception d'une connexion, elle retourne immédiatement. La méthode `complete()` de l'objet **AcceptCompletionHandler** sera appelée lorsqu'une connexion arrivera.

Vous allez devoir bloquer votre *Thread* principal, car sinon l'application va arrêter avant qu'une connexion arrive. Un **CountdownLatch** peut vous être utile dans ce genre de situation.

Exercice #3 (UTFServerNIO2.java)

Réimplémentez le **UTFServer** à l'aide de NIO.2 (**AsynchronousServerSocketChannel** , **AcceptCompletionHandler**, **ReadCompletionHandler**...).

Indice: N'oubliez pas de fermer toutes vos ressources avec votre utilitaire.

Le cadre d'application Netty

Netty est un cadre d'application qui simplifie grandement le développement d'applications communicantes, tout en permettant des performances époustouflantes et une configurabilité hors du commun. Il est possible de configurer Netty pour être optimal dans des conditions d'un très grand nombre de petites requêtes, ou bien de le configurer pour être optimal pour un grand nombre de grosses requêtes. Il est possible de le configurer pour consommer peu de ressource, ou dans un contexte ou on veut garantir un temps de réponse fixe. Bref, si vous comprenez les cadres d'applications vous n'aurez plus jamais à utiliser de Sockets et vos applications seront plus performantes (mais aussi plus complexes).

Fini le discours de vente, programmons. Voici la version Netty de la classe **ReadCompletionHandler** que nous avions rédigée dans le contexte de NIO.2.

```
class ReadHandler extends SimpleChannelInboundHandler<String> {
	@Override
	protected void channelRead0(ChannelHandlerContext ctx, String msg)
			throws Exception {
		...
	}
}
```

La classe **SimpleChannelInboundHandler** fait tout le travail de gérer le *Selector* et l'enregistrement pour les événements (patron *Reactor* et Acceptor-Connector). En plus, on peut lui dire qu'on désire un message de type *String* avec les *Generics*. Cela dit, la configuration de Netty est un peu plus ardue.

Tout d'abord, il faut décider d'une configuration du serveur Netty, nous allons utiliser un *Threadpool* de *Boss* et un *Threadpool* de *Workers* (les *Boss* reçoivent les requêtes et les *Workers* les exécutent).

Nous allons nous binder au port de façon bloquante et nous allons également bloquer jusqu'à la fermeture de l'application. Par la suite, nous allons nettoyer les ressources utilisées.

```
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();

try {
	ServerBootstrap b = new ServerBootstrap();
	b.group(bossGroup, workerGroup)
	.channel(NioServerSocketChannel.class)
		.childHandler(new ReadInitializer());
```

```
        ChannelFuture f = b.bind(PORT).sync();
        f.channel().closeFuture().sync();

} catch (InterruptedException e) {
        e.printStackTrace();
} finally {
        workerGroup.shutdownGracefully();
        bossGroup.shutdownGracefully();
}
```

Où se trouve notre **ReadHandler**? Il se trouve dans notre **ReadInitializer**. La classe **ReadInitializer** permet de créer un *Pipeline* pour la réception et l'envoi de messages ce qui offre une très grande flexibilité. Dans notre cas, nous devons créer un **StringDecoder** avant d'utiliser notre **ReadHandler**.

Exercice #4 (UTFServerNetty.java)

Réimplémentez le **UTFServer** à l'aide de Netty (ServerBootstrap...).

Indice: N'oubliez pas de fermer toutes vos ressources.

Netty vous offre une panoplie de Handler pour toutes sortes de formats, et s'adapte à vos besoin. Testons cette affirmation

Exercice #5 (TimeClient.java, TimeServer.java, NTP.java)

Le **UTFServer** utilise TCP et un encodage en chaîne de caractères. Nous allons donc développer un système UDP avec un encodage en octets. Vous devez implémenter un serveur qui retourne l'heure (**TimeServer**). Le client envoie une requête au serveur qui lui retourne l'heure.

Notes :

- Vous devez utiliser Netty sur le client et le serveur.
- Le **NioDatagramChannel** de Netty encapsule une connexion UDP
    - Pour envoyer un **DatagramPacket**, commencez par *binder* votre canal sur le port (0).
- N'utilisez **pas** de *BROADCAST*
- N'oubliez pas d'appeler la méthode `flush()` sur votre canal.
- Pour stocker de l'information dans le contexte d'une connexion, utilisez un objet de type AttributeKey et la méthode `attr().set()` et `attr().get()`.

- Netty offre la méthode `channelReadComplete()` qui est appelée après la méthode `channelRead()`.
- Pour répondre à une requête UDP, vous devez créer un DatagramPacket avec la réponse et l'adresser à l'émetteur de la requête
    - Vous pouvez stocker l'adresse de l'émetteur dans le contexte, si vous avez de la difficulté à l'obtenir lorsque vous créez la réponse.
- N'utilisez pas l'objet **ByteBuffer** de NIO, mais **ByteBuf** de Netty. Pour créer votre objet **ByteBuf**, utilisez la méthode statique `Unpooled.buffer()`.
- Lorsque vous manipulez votre **ByteBuf**, préférez les méthodes `writeLong()` et `readLong()` par rapport aux méthodes `setLong()` et `getLong()`.

Indice: N'oubliez pas de fermer toutes vos ressources.

ATTENTION! L'heure décodée par le client sera en retard. Votre client doit avoir l'heure exacte, pour se faire, consultez l'algorithme de synchronisation du protocole NTP pour plus d'information.

Pour vous permettre de tester sur une même machine et dans un environnement contrôlé, vous devez utiliser la méthode `getClientTime()` sur le client.

Pour vérifier votre résultat, utilisez la méthode `report()`. Cette méthode prend en paramètre la mauvaise heure, l'estimation de la bonne heure et l'offset mesuré par le client. Il est possible d'arriver à un estimé précis à la milliseconde.

Objectif du TP: Une estimation à l'intérieur de 10 millisecondes du vrai temps.

Pour des exemples d'utilisation de Netty, consultez la documentation officielle (version 4.0).

- L'exemple Quote est en UDP, mais avec des Broadcast.