UNIVERSITÀ DI PISA

Programmazione di reti
Corso B

26 Aprile 2016
Lezione 8

Correzione: Modificare un *file* usando IO tradizionale

- Classe `RandomAccessFile` - permette di aprire un *file* in modalità *read/write*.

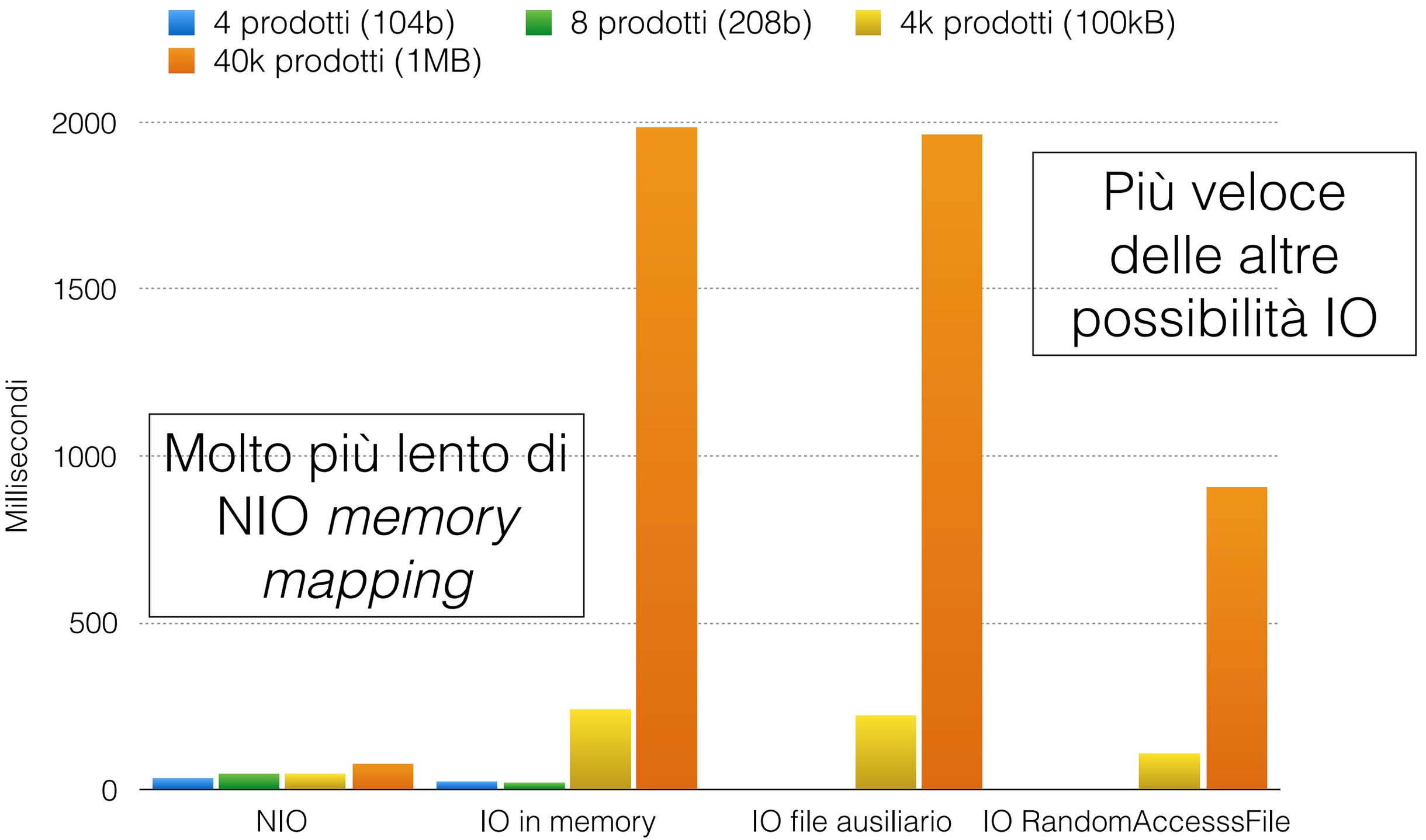
- Non fa parte della gerarchia di classi *stream*

- Possiamo sovrascrivere i dati come abbiamo visto anche per *file memory mapping*.

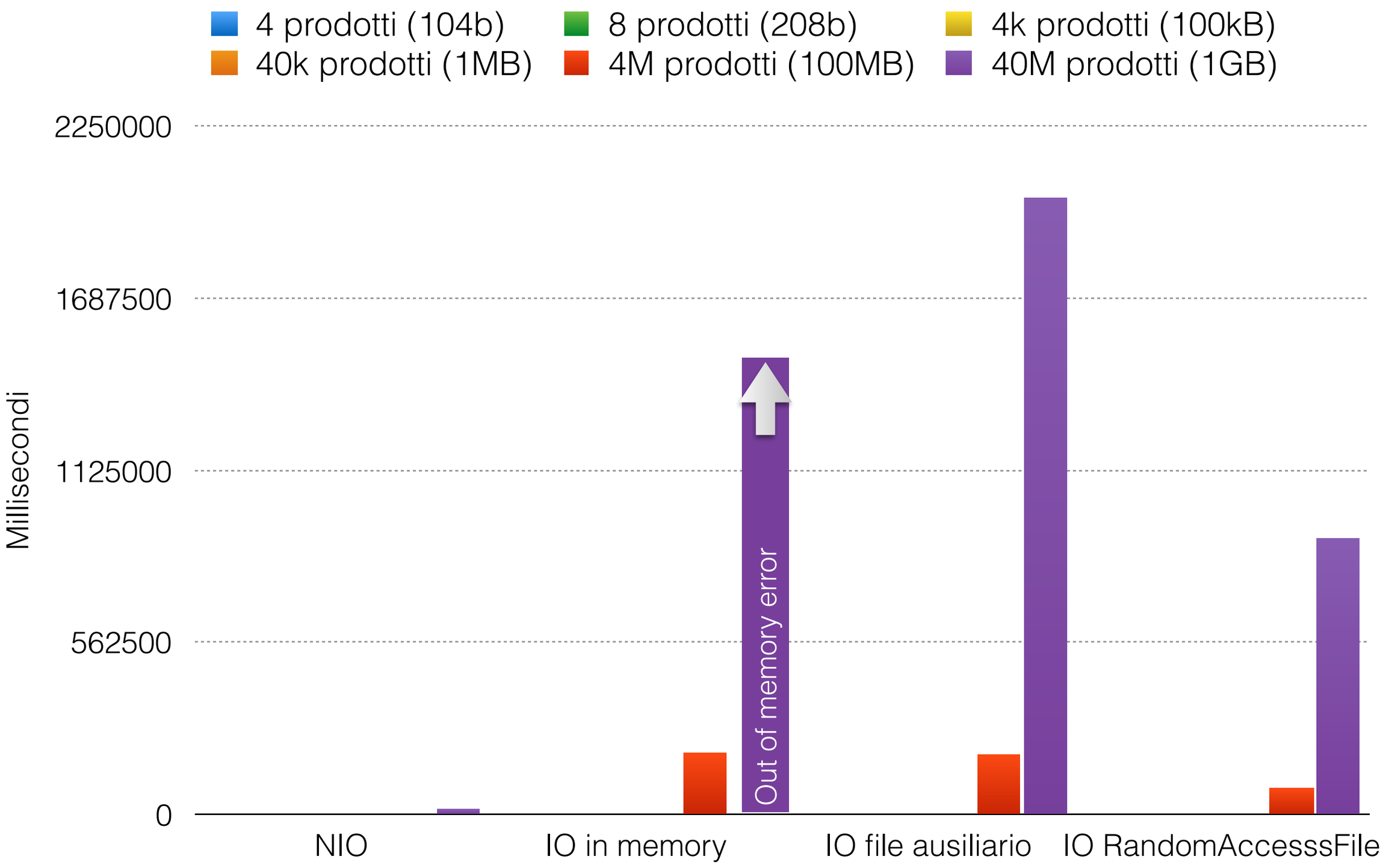
```
public class ModifyFileIORandomAccess {

   public static void main(String[] args) {
      long start= System.currentTimeMillis();
      String prodName= "chocolate bar";
      double newPrice=2.5;
      try(RandomAccessFile file = new RandomAccessFile("Products.dat","rw")){
         while(true){
            int nameLength=file.readInt();
            StringBuilder name=new StringBuilder();
            for(int i=0;i<nameLength;i++)
               name.append(file.readChar());
            if (name.toString().equals(prodName)){
               //found the product, can overwrite the price
               file.writeDouble(newPrice);
            } else {
               file.readDouble();
            }
         }
      } catch (EOFException e){ //end of file, don't do anything
      } catch (IOException e) {
         System.out.println("Something went wrong: "+e.getMessage());
      }
      System.out.format("Ran %d milliseconds.%n",System.currentTimeMillis()-start);
   }
}
```

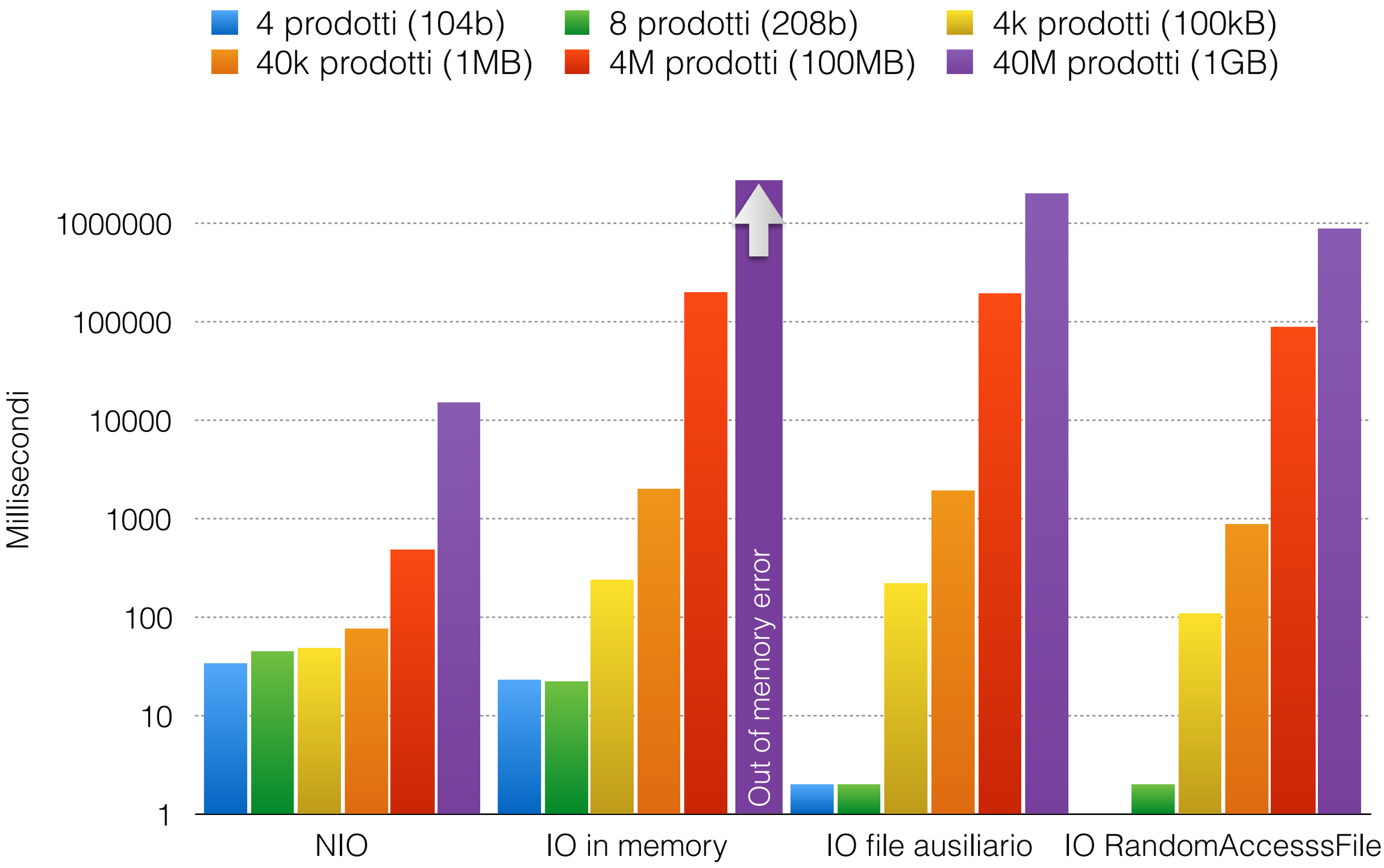
Tempi di esecuzione (scala lineare per assi verticale)



Tempi di esecuzione (scala lineare per assi verticale)



Tempi di esecuzione (scala logaritmica per assi verticale)



Contenuti

- NIO
  - *channel non-blocking*
  - selettori
  - *channel* per comunicazioni TCP e UDP

Non-blocking

- Una delle funzionalità più importanti del NIO è il **comportamento *non-blocking*** per le varie operazioni IO

- *Non-blocking*: l'operazione restituisce subito un risultato, anche se a volte l'operazione non può essere eseguita

- E.g. *Non-blocking read:* se ci sono *byte* da leggere, li legge, altrimenti restituisce 0 (non è stato possibile leggere nessun *byte*)

Non-blocking

- Usando le funzionalità *non-blocking*, il programma deve ripetere le operazioni fin che completano

```
while(buffer.remaining() && channel.read(buffer)!=-1) {}
```

- Se ci sono più operazioni da eseguire (e.g. più *channel* da leggere) uno deve iterare tra tutti i *channel*

- NON E' UNA SOLUZIONE EFFICIENTE! (*active loop*, tante operazioni IO non sono ancora disponibili)

- SOLUZIONE MIGLIORE: **SELETTORI**

Selettori

- Oggetti che facilitano il *multiplexing* dei *channel*.
- Gestiscono una lista di *channel* su cui eseguire operazioni IO.
- Possono selezionare solo i *channel* su cui le operazioni sono disponibili.
- A questo punto il programma può iterare tra i *channel* pronti e eseguire le operazioni in modo *non-blocking* (il selettore garantisce che le operazioni non saranno inutili: e.g. ci sono *byte* da leggere)
- Per essere usati con selettori, i *channel* devono essere selezionabili: estendere classe astratta `SelectableChannel`

Selector

- *Multiplexer* per *channel* selezionabili: capace di selezionare i *channel* pronti, dalla lista di *channel* registrati

`Selector open()`

crea un selettore usando l'implementazione di default del sistema.

`void close()`

chiude il selettore

`boolean isOpen()`

verifica se il selettore e aperto

Selector

- Selezionare i *channel* pronti per operazioni IO:

```
int   select()
```

Restituisce il numero di *channel* pronti. Si blocca fin che almeno un *channel* è pronto, il *thread* viene interrotto o la selezione viene fermata con il metodo `wakeup()`

```
int   select(long timeout)
```

Restituisce il numero di *channel* pronti. Si blocca fin che almeno un *channel* è pronto, il `timeout` scade, il *thread* viene interrotto o la selezione viene fermata con il metodo `wakeup()`

Selector

`int selectNow()`

Operazione *select non-blocking*. Restituisce il numero di *channel* pronti subito.

`void wakeup()`

interrompe un'operazione *select* bloccante. Se c'è un *thread* bloccato in un *select*, quel metodo restituisce subito. Se non c'è nessuna operazione *select* in corso, la prossima restituirà un risultato subito. Utile quando si cambiano le chiavi di selezione in un *thread* separato e si vuole aggiornare il selettore.

Selector

- Per accedere ai *channel* pronti:

`Set<SelectionKey> selectedKeys()`

Restituisce un set di `SelectionKey`: classe che memorizza un riferimento al *channel*, e il suo stato. Una volta gestita, ogni chiave deve essere rimossa dal set.

Metodo deve essere richiamato solo dopo aver completato un'operazione *select*.

Il set di `SelectionKey` non è *thread* safe, pero ogni oggetto `SelectionKey` è *thread* safe

`SelectableChannel`

- Classe astratta di base per *channel* che devono essere usati con selettori

- Metodi per impostare comportamento *blocking/non-blocking* del *channel*

- Metodi per la registrazione con un selettore

- Estesa da `AbstractSelectableChannel`

SelectableChannel

- Inizialmente, tutti i *channel* sono bloccanti.

`SelectableChannel  configureBlocking(boolean block)`

Imposta il comportamento *bloccante/non-bloccante*.
Un *channel* deve diventare non-bloccante prima di essere registrato con un selettore

`boolean isBlocking()`

Verifica se un *channel* è bloccante

SelectableChannel

`SelectionKey register(Selector sel, int ops)`

Aggiunge questo *channel* alla lista di *channel* gestiti dal selettore `sel`. Crea una chiave di selezione (`SelectionKey`) che include un riferimento al *channel e* le operazioni di interesse. Se il *channel* era già registrato, la chiave iniziale di selezione viene restituita, dopo essere stata aggiornata con le nuove operazioni di interesse `ops`.

L'operazione IO di interesse registrata viene specificata dal parametro `ops`. Ci sono 4 possibilità, da combinare usando OR (|):
`SelectionKey.OP_ACCEPT` (Per *server socket*)
`SelectionKey.OP_CONNECT` (Per *client socket*)
`SelectionKey.OP_READ` (Per tutti i *channel readable*)
`SelectionKey.OP_WRITE` (Per tutti i *channel writeable*)
Si può usare anche 0 , se si vuole la registrazione senza operazioni di interesse (che verranno aggiunte più tardi).

SelectableChannel

`SelectionKey register(Selector sel,int ops,Object att)`

La `SelectionKey` può anche contenere un oggetto definito dal programmatore (lo stato del *channel*), chiamato *attachment*. Questo metodo fa la registrazione con *attachment*. Il programmatore decide l'oggetto da mettere in *attachment* - per facilitare il protocollo di comunicazione (e.g. il *buffer*)

Se il *channel* era già registrato, la chiave iniziale di selezione viene restituita, dopo essere stata aggiornata con le nuove operazioni di interesse `ops` e il nuovo *attachment* `att`.

`SelectionKey keyFor(Selector sel)`

Restituisce la chiave di selezione associata con questo *channel* nel selettore `sel.`

`int validOps()`

Restituisce tutte le operazioni valide per un *channel*. E.g. per un `SocketChannel` aperto per leggere e scrivere il metodo restituisce (`SelectionKey.OP_READ|SelectionKey.OP_WRITE`)

SelectionKey

- Oggetto che memorizza il *channel* registrato ad un selettore e il suo stato (operazioni gestite dal selettore e *attachment*). Creato al momento della registrazione del *channel* con il selettore.

`void cancel()`

Cancella la registrazione. Una chiave di registrazione è valida fin che questo metodo viene richiamato, o il *channel* o il selettore vengono chiusi.

`SelectableChannel channel()`

Restituisce il *channel* registrato. Il risultato può essere usato per eseguire l'operazione per cui è pronto

`Object attachment()`

Restituisce l'*attachment* creato alla registrazione del *channel*.

SelectionKey

```
public final boolean isAcceptable()

public final boolean isConnectable()

public final boolean isReadable()

public final boolean isWritable()
```

Verifica se il channel è pronto per queste operazioni.

```
SelectionKey interestOps(int ops)
```

Cambia il set di operazioni monitorate dal selettore per questa chiave. In caso il selettore è gia bloccato in un *select*, le operazioni saranno aggiornate al prossimo *select*. Un nuovo *select* può essere forzato usando `wakeup`().

```
//open channel
try(SelectableChannel channel=.....){
   //open selector
   Selector selector = Selector.open();
   //register channel with selector, including attachments
   channel.register(selector,[Operation1|Operation2|…],[attachment]);
   while(true){
      selector.selectedKeys().clear();
      selector.select();
      for (SelectionKey key : selector.selectedKeys() ){
         if(key.isReadable()){
             SelectableChannel channel = (SelectableChannel) key.channel();
             //read
          }
          if(key.isWritable()){
            //write
           }
           if(key.isAcceptable()){
            //accept connection
           }
           if(key.isConnectable()){
            //connect
           }
      }
   }
}
```

Schema generale

Compattare i *buffer*

- Utile quando si legge da un *channel* e si scrive in un altro *channel*, in modo non-bloccante, usando lo stesso *buffer*

- Dopo un *drain*, i dati rimasti tra `position` e `limit` vengono spostati all'inizio del *buffer*, `position` viene spostata alla fine dei dati rimasti, e `limit=capacity`.

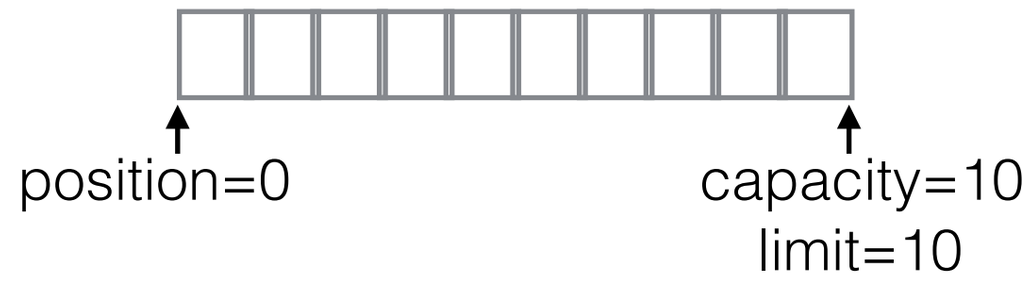
- Adesso il *buffer* è pronto per fare un nuovo *fill*

- Metodi: `***Buffer compact()` - uno per ogni `***Buffer`

- e.g. `IntBuffer compact()`

Compattare i buffer

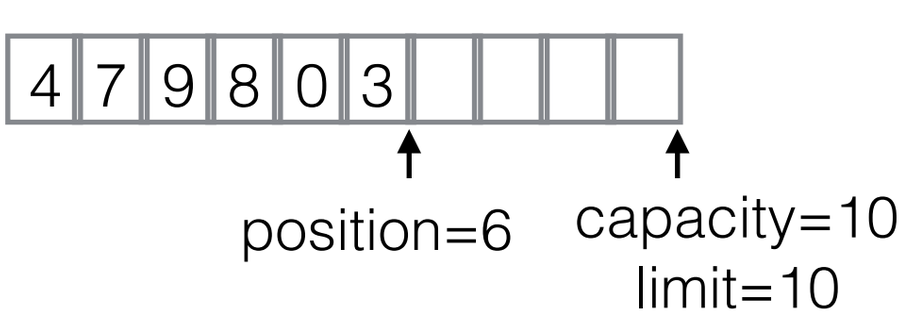
(1) `IntBuffer intBuffer= IntBuffer.allocate(10)`



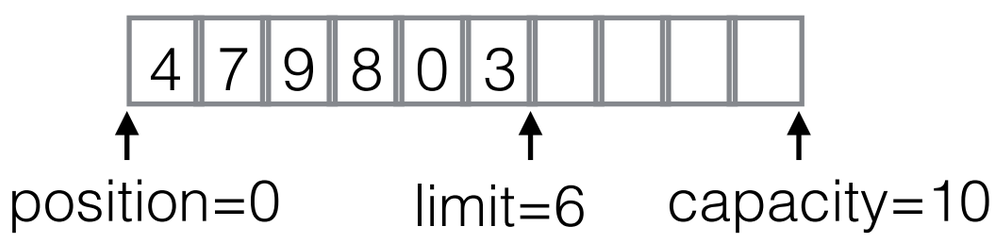
riesce a leggere 6 int

supponiamo che riesce a scrivere 3 int

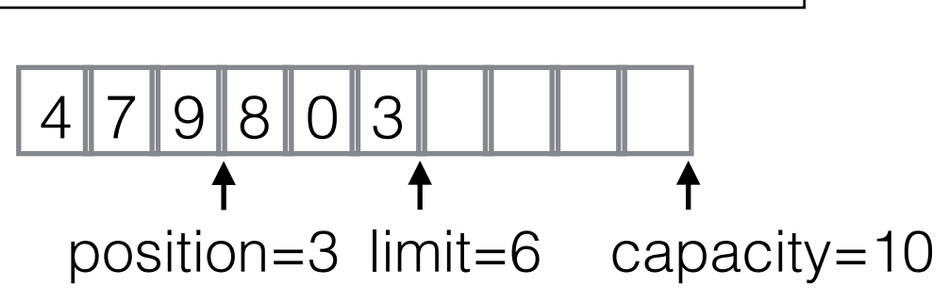
(2) `inChannel.read(intBuffer)`



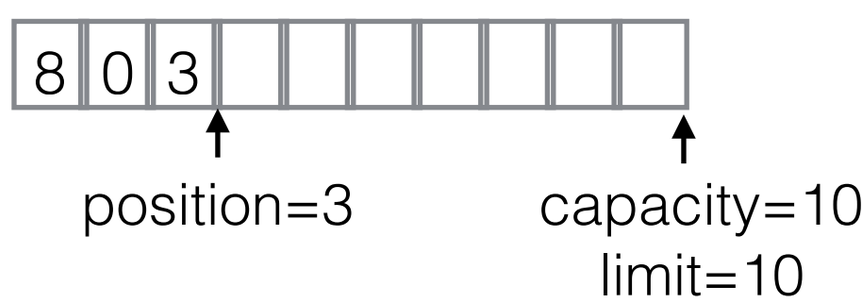
(3) `intBuffer.flip()`

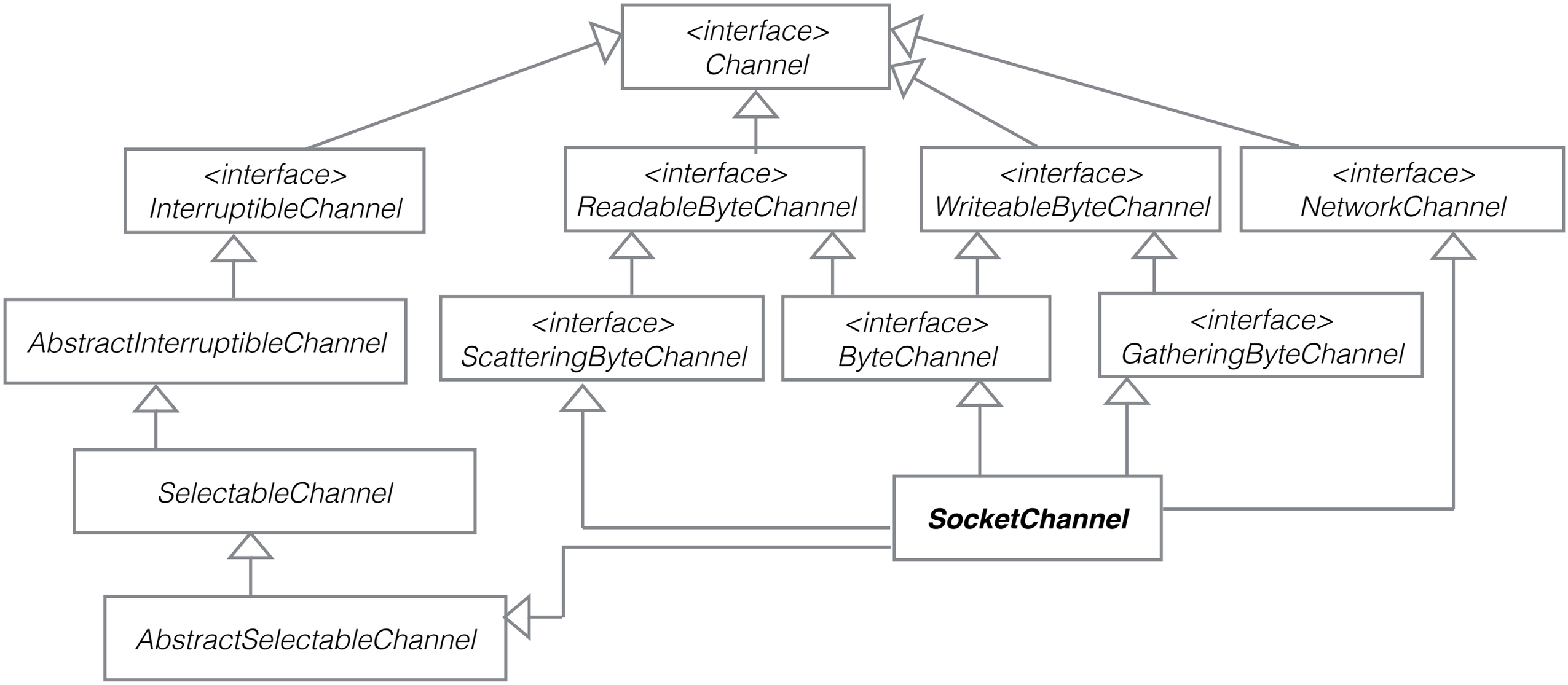


(4) `outChannel.write(intBuffer)`



(5) `intBuffer.compact()`




<interface>
Channel
<interface>
InterruptibleChannel
<interface>
ReadableByteChannel
<interface>
WriteableByteChannel
<interface>
NetworkChannel
AbstractInterruptibleChannel
<interface>
ScatteringByteChannel
<interface>
ByteChannel
<interface>
GatheringByteChannel
SelectableChannel
SocketChannel
AbstractSelectableChannel

SocketChannel

- *Channel* che trasferisce datti da/a un socket TCP. Capace di scrivere, leggere, fare *scatter* e *gather*, *blocking* e *non-blocking*, selezionabile

```
SocketChannel open(SocketAddress peer) throws
IOException
```

Apre un *socket* e lo connette al `peer`.

```
SocketChannel open() throws IOException
```

Apre un *socket* non connesso.

```
boolean connect(SocketAddress peer)
```

Apre la connessione al peer.

SocketChannel

`channel.configureBlocking(false);`

Imposta comportamento non-bloccante al *channel*. Se richiamato prima del metodo `connect()`, anche la connessione si fa in modo non-bloccante. Registrazione con un selettore si può fare usando operazione `SelectionKey.OP_CONNECT`.

`boolean finishConnect() throws IOException`

Da usare quando `connect` è stato richiamato in modo non-bloccante. Se channel è bloccante, si blocca fin che la connessione è stabilita. Se non-bloccante, metodo restituisce `false` se la connessione non è ancora usabile, `true` se la connessione è pronta.

`boolean isConnected()`

restituisce `true` se la connessione è stata completata

`boolean isConnectionPending()`

restituisce `true` se la connessione si sta ancora completando

SocketChannel

`Socket  socket()`

restituisce l'oggetto `Socket` usato dal *channel*. Questo oggetto può essere usato per impostazioni, come abbiamo visto anche prima.

NetworkChannel

- Da Java 7 in poi `SocketChannel` implementa `NetworkChannel`

`NetworkChannel bind(SocketAddress local)`

Associa il *socket* del *channel* al indirizzo locale specificato (IP e *port*). Se `local==null`, il *port* viene selezionato dal sistema. Se questo metodo non viene usato, un *bind* a un *port* automatico viene eseguito durante l'operazione *connect*.

`SocketAddress  getLocalAddress()`

Restituisce il *port* e IP associati al *socket*.

NetworkChannel

`T getOption(SocketOption<T> name)`

Restituisce il valore dell'opzione specificata. I nomi delle opzioni sono disponibili nella classe `StandardSocketOptions`

`NetworkChannel setOption(SocketOption<T> name, T value)`

Imposta un valore `value` all'opzione `name`.

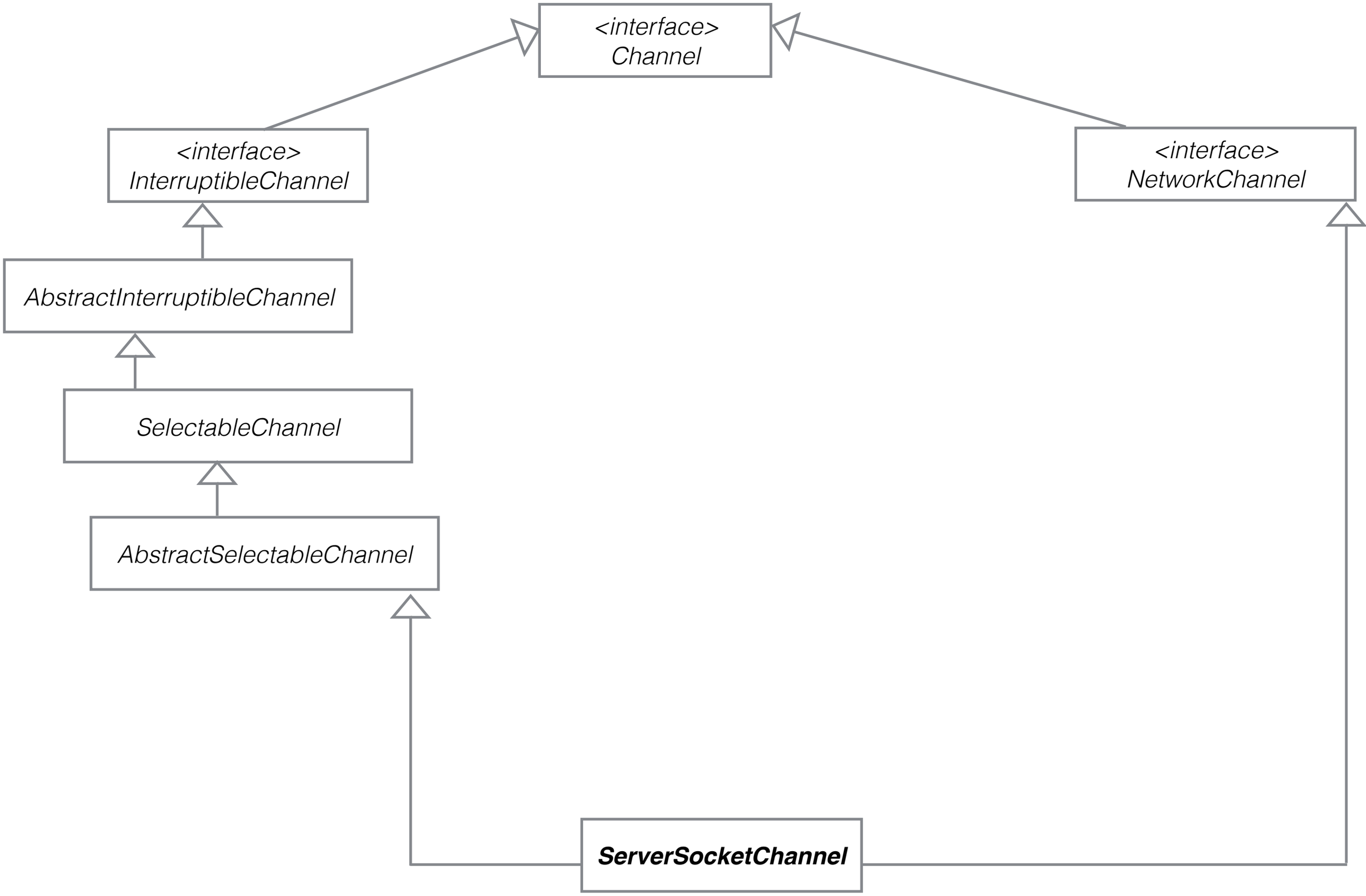
`Set<SocketOption<?>> supportedOptions()`

Restituisce una lista di opzioni validi per questo *channel*.

SocketOption<T>

- Per `SocketChannel`, le opzioni valide sono gli stessi della classe `Socket`:

```
SocketOption<Boolean> StandardSocketOptions.SO_KEEPALIVE
SocketOption<Integer> StandardSocketOptions.SO_LINGER
SocketOption<Integer> StandardSocketOptions.SO_RCVBUF
SocketOption<Boolean> StandardSocketOptions.SO_REUSEADDR
SocketOption<Integer> StandardSocketOptions.SO_SNDBUF
SocketOption<Boolean> StandardSocketOptions.TCP_NODELAY
```


<interface>
Channel
<interface>
InterruptibleChannel
<interface>
NetworkChannel
AbstractInterruptibleChannel
SelectableChannel
AbstractSelectableChannel
ServerSocketChannel

ServerSocketChannel

- Classe *channel* che accetta nuove connessioni su un *socket* TCP, *blocking e non-blocking*, non sa scrivere ne leggere, è selezionabile

```
static ServerSocketChannel open()
```

Crea un oggetto di tipo `ServerSocketChannel` non associato con un *port*. l'associazione si fa usando metodo **`bind`** dell'interfaccia `NetworkChannel`. Implementazione usa un `ServerSocket` alla base.

```
ServerSocket  socket()
```

Restituisce il `ServerSocket` usato dal *channel*. Questo oggetto può essere usato per impostazioni e per fare *bind*, come abbiamo visto per i socket TCP.

ServerSocketChannel

`SocketChannel accept()`

Metodo che e accetta nuove connessioni. In modo non-bloccante, questo metodo restituisce subito `null` se non c'è nessuna richiesta di connessione.

Registrazione con un selettore si deve fare con `SelectionKey.OP_ACCEPT`

Esempio: MiniFTP usando NIO

Cliente (bloccante)

Si connette al *server*

Invia il nome di un *file*, ricevuto dall'utente

Aspetta la risposta dal *server*, salvando i *byte* ricevuti in un *file*.

Server (non-bloccante)

Apre un `ServerSocketChannel` e lo registra ad un selettore per operazioni `OP_ACCEPT`

Accetta connessione e registra il `SocketChannel` `client` con il selettore per operazioni `OP_READ`.

Legge nome del file. Apre il file usando un `FileChannel file`. Crea un `buffer` per la risposta. Registra `client` con il selettore per operazioni `OP_WRITE`, usando `buffer` e `file` come attachment.

Scrive dati dal channel `file` nel *channel* `client`, *buffer* dopo *buffer,* fin che il *file* finisce, quando la connessione viene chiusa.

```
public class FtpClient {

   public static void main(String[] args) {
      try(SocketChannel client= SocketChannel.open(
            new InetSocketAddress(InetAddress.getLocalHost(),
                  FtpServerBuffer.PORT));
            ReadableByteChannel in = Channels.newChannel(System.in);) {
ByteBuffer buffer= ByteBuffer.allocate(FtpServerBuffer.BLOCK_SIZE);
in.read(buffer);
String fileName="downloaded_"+
      new String(buffer.array(),0,buffer.position()).trim();
buffer.flip();
client.write(buffer);
buffer.clear();
```

Cliente invia nome del *file* usando un unico *buffer* per leggere da `System.in` e scrivere nel `SocketChannel`

```
ByteBuffer[] bufferArray= new ByteBuffer[2];
bufferArray[0]=ByteBuffer.allocate(Integer.BYTES);
bufferArray[1]=buffer;
FileChannel out= null;
int responseCode=-1;
while(client.read(bufferArray)!=-1){
   if(responseCode==-1){
      if (!bufferArray[0].hasRemaining()){
         //set the response code
         bufferArray[0].flip();
         responseCode=bufferArray[0].getInt();
      }
   }
   if (responseCode==0){//file was found
      if (out == null) //file not created yet
         out= FileChannel.open(Paths.get(fileName),
               StandardOpenOption.CREATE,StandardOpenOption.WRITE);
      bufferArray[1].flip();
      out.write(bufferArray[1]);
      bufferArray[1].compact();
   }
   if (responseCode==1){
      throw new NoSuchFileException("Server could not find file");
   }
}
out.close();
System.out.println("Finished transferring file");
```

Cliente legge il codice di risposta e i contenuti del *file* usando 2 *buffer* diversi.

Se non ancora letto, cliente legge il codice di risposta.

Cliente riceve nel secondo buffer il contenuto del *file*.

Cliente ha ricevuto risposta negativa: *file* non esiste.

```
} catch (UnknownHostException e) {
   System.out.println("Error connecting to server: "+e.getMessage());
} catch (NoSuchFileException e) {
   System.out.println("Error in locating file: "+e.getMessage());
}catch (IOException e) {
         System.out.println("General error: "+e.getMessage());
      }
   }

}
```

Server usando NIO *non-blocking* e *buffer*

Server

Crea ServerSocketChannel

Selettore, *channel* non-bloccante, registrazione *accept*

Ciclo infinito, selezione *channel* pronti, iterazione tra *channel* pronti

Gestione operazioni *accept*: registrazione nuovo *channel* col selettore

```
public class FtpServerBuffer {
   public final static int PORT=2000;
   public final static int BLOCK_SIZE=1024;

   public static void main(String[] args) {
      try(ServerSocketChannel server= ServerSocketChannel.open() ){
         server.bind(new InetSocketAddress(InetAddress.getLocalHost(),
             FtpServerBuffer.PORT));
         Selector selector = Selector.open();
         server.configureBlocking(false);
         server.register(selector, SelectionKey.OP_ACCEPT);
         while (true){
            selector.selectedKeys().clear();
            selector.select();
            for( SelectionKey key : selector.selectedKeys()){
               if(key.isAcceptable()){
                  try{
                     SocketChannel client=
                         ((ServerSocketChannel)key.channel()).accept();
                     client.configureBlocking(false);
                     client.register(selector, SelectionKey.OP_READ);
                     System.out.println("New client accepted");
                  }catch (IOException e){
                     System.out.println("Error accepting client: "+e.getMessage());
                  }
               }
```

Gestione operazioni *read*

Lettura nome del *file*

File esiste: metto codice 0 nel *buffer.* Registro *channel* per scrittura, metto *buffer* e *file* in *attachment*

File non esiste: metto codice 1 nel *buffer.* Registro *channel* per scrittura, metto solo *buffer* in *attachment*

```
if (key.isReadable()){
    try{
        ByteBuffer buffer= ByteBuffer.allocate(FtpServerBuffer.BLOCK_SIZE);
        SocketChannel client=(SocketChannel)key.channel();
        client.read(buffer);
        String fileName= new String(buffer.array(),
                0, buffer.position()).trim();
        System.out.println("Requested file: "+fileName);
        buffer.clear();
        ArrayList<Object> attachment= new ArrayList<>();
        attachment.add(buffer);
        try{
            FileChannel file= FileChannel.open(
                Paths.get(fileName), StandardOpenOption.READ);
            buffer.putInt(0);
            attachment.add(file);
            client.register(selector, SelectionKey.OP_WRITE, attachment);
            System.out.println("File exists. Sending file.");
        } catch (NoSuchFileException e){
            buffer.putInt(1);
            client.register(selector, SelectionKey.OP_WRITE,attachment);
        }
    } catch (IOException e){
        System.out.println("Error reading from client: "+e.getMessage());
        key.cancel();
    }
```

Gestione operazioni *write*

```
if (key.isWritable()){
   try{
      SocketChannel client=(SocketChannel)key.channel();
      ArrayList<Object> attachment=
               (ArrayList<Object>)key.attachment();
      ByteBuffer buffer = (ByteBuffer) attachment.get(0);
      buffer.flip();
      client.write(buffer);
      if (attachment.size()>1){ //file exists
         FileChannel file= (FileChannel) (attachment).get(1);
         buffer.compact();
         if(file.read(buffer)==-1 && buffer.position()==0){
            file.close();
            client.close();
         }
      } else {
         //file doesn't exist, close channel
         if(!buffer.hasRemaining())
            client.close();
      }
   } catch (IOException e){
      System.out.println("Error writing to client: "+e.getMessage());
      key.cancel();
   }
}
```

Scrivo contenuto del buffer (trovato in attachment)

Metto più contenuti nel buffer, leggendoli dal file. Questi saranno scritti all'iterazione successiva.

File finito: chiudo connessione.

File non esiste nel attachment: chiudo connessione.

```
            }
        }
    } catch (IOException e){
        System.out.println("General error: "+e.getMessage());
    }
        }
    }
```

Server usando NIO *non-blocking* e *memory mapping*

Gestione operazioni *read*

Lettura nome del *file*

File esiste: metto codice 0 nel *buffer.* Registro *channel* per scrittura, metto *buffer* e *file* mappato in *attachment*

File non esiste: metto codice 1 nel *buffer.* Registro *channel* per scrittura, metto solo *buffer* in *attachment*

```
if (key.isReadable()){
    try{
        ByteBuffer buffer= ByteBuffer.allocate(FtpServerMap.BLOCK_SIZE);
        SocketChannel client=(SocketChannel)key.channel();
        client.read(buffer);
        String fileName=
              new String(buffer.array(), 0, buffer.position()).trim();
        System.out.println("Requested file: "+fileName);
        buffer.clear();
        try{
            FileChannel file= FileChannel.open(
                 Paths.get(fileName), StandardOpenOption.READ);
            buffer.putInt(0);
            buffer.flip();
            ByteBuffer[] attachment= new ByteBuffer[2];
            attachment[0]=buffer;
            attachment[1]=file.map(MapMode.READ_ONLY, 0, file.size());
            client.register(selector, SelectionKey.OP_WRITE, attachment);
            System.out.println("File exists. Sending file.");
        } catch (NoSuchFileException e){
            buffer.putInt(1);
            buffer.flip();
            ByteBuffer[] attachment= new ByteBuffer[1];
            attachment[0]=buffer;
            client.register(selector, SelectionKey.OP_WRITE,attachment);
        }
    } catch (IOException e){
        System.out.println("Error reading from client: "+e.getMessage());
        key.cancel();
    }}
```

Scrivo contenuto dei due buffer (trovati in attachment) fin che tutti e due sono finiti.

```
if (key.isWritable()){
    try{
        SocketChannel client=(SocketChannel)key.channel();
        ByteBuffer[] attachment=(ByteBuffer[]) key.attachment();
        client.write(attachment);
        if (attachment.length==1 || !attachment[1].hasRemaining()){
            //file doesn't exist or wrote everything, close channel
            client.close();
        }
    } catch (IOException e){
            System.out.println("Error writing to client: "+e.getMessage());
            key.cancel();
        }
    }
```

Server usando NIO *non-blocking* e *transfer*

Gestione operazioni *read*

```
if (key.isReadable()){
    try{
        ByteBuffer buffer= ByteBuffer.allocate(FtpServerBuffer.BLOCK_SIZE);
        SocketChannel client=(SocketChannel)key.channel();
        client.read(buffer);
        String fileName=
            new String(buffer.array(), 0, buffer.position()).trim();
        System.out.println("Requested file: "+fileName);
        buffer.clear();
        ArrayList<Object> attachment= new ArrayList<>();
        attachment.add(buffer);
        try{
            FileChannel file= FileChannel.open(
                Paths.get(fileName), StandardOpenOption.READ);
            buffer.putInt(0);
            attachment.add(file);
            client.register(selector, SelectionKey.OP_WRITE, attachment);
            System.out.println("File exists. Sending file.");
        } catch (NoSuchFileException e){
            buffer.putInt(1);
            client.register(selector, SelectionKey.OP_WRITE,attachment);
        }
        buffer.flip();
    } catch (IOException e){
        System.out.println("Error reading from client: "+e.getMessage());
        key.cancel();
    }
}
```

Lettura nome del *file*

File esiste: metto codice 0 nel *buffer.* Registro *channel* per scrittura, metto *buffer* e *file* in *attachment*

File non esiste: metto codice 1 nel *buffer.* Registro *channel* per scrittura, metto solo *buffer* in *attachment*

Faccio flip per poter fare direttamente write più tardi.

Gestione operazioni *write*

```
if (key.isWritable()){
    try{
        SocketChannel client=(SocketChannel)key.channel();
        ArrayList<Object> attachment=(ArrayList<Object>)key.attachment();
        ByteBuffer buffer = (ByteBuffer) attachment.get(0);
        client.write(buffer);
        if (! buffer.hasRemaining()){ //wrote header
            if (attachment.size()>1){ // has file
                FileChannel file= (FileChannel) (attachment).get(1);
                long transfered=file.transferTo(file.position(),
                        file.size()-file.position(), client);
                file.position(file.position()+transfered);
                if(file.position()==file.size()){
                    file.close();
                    client.close();
                }
            }else {
                //file doesn't exist, close channel
                client.close();
            }
        }
    } catch (IOException e){
        System.out.println("Error writing to client: "+e.getMessage());
        key.cancel();
    }
}
```

Scrivo contenuto del buffer (trovato in attachment)

Trasferisco parte del file nel SocketChannel. Cambio la posizione nel FileChannel per memorizzare quanto si è trasferito.

File finito: chiudo connessione.

File non esiste nel attachment: chiudo connessione.

Server *multithreaded* usando NIO *non-blocking* e *transfer*

```
public class FtpServerTransferMultithreaded {
    public final static int PORT=2000;
    public final static int BLOCK_SIZE=1024;
    public final static int THREADS=10;

    public static void main(String[] args) {
        try(ServerSocketChannel server= ServerSocketChannel.open() ){
            server.bind(new InetSocketAddress(InetAddress.getLocalHost(), FtpServerBuffer.PORT));
            Selector selector = Selector.open();
            server.configureBlocking(false);
            server.register(selector, SelectionKey.OP_ACCEPT);
            ExecutorService executor=
                    Executors.newFixedThreadPool(FtpServerTransferMultithreaded.THREADS);
            while (true){
                selector.selectedKeys().clear();
                selector.select();
                for( SelectionKey key : selector.selectedKeys()){
                    if(key.isAcceptable()){
                        try{
                            SocketChannel client=( (ServerSocketChannel)key.channel()).accept();
                            client.configureBlocking(false);
                            client.register(selector, SelectionKey.OP_READ);
                            System.out.println("New client accepted");
                        }catch (IOException e){
                            System.out.println("Error accepting client: "+e.getMessage());
                        }
```

Creo un executor.

Delego gestione *read* e *write* ai *thread*.

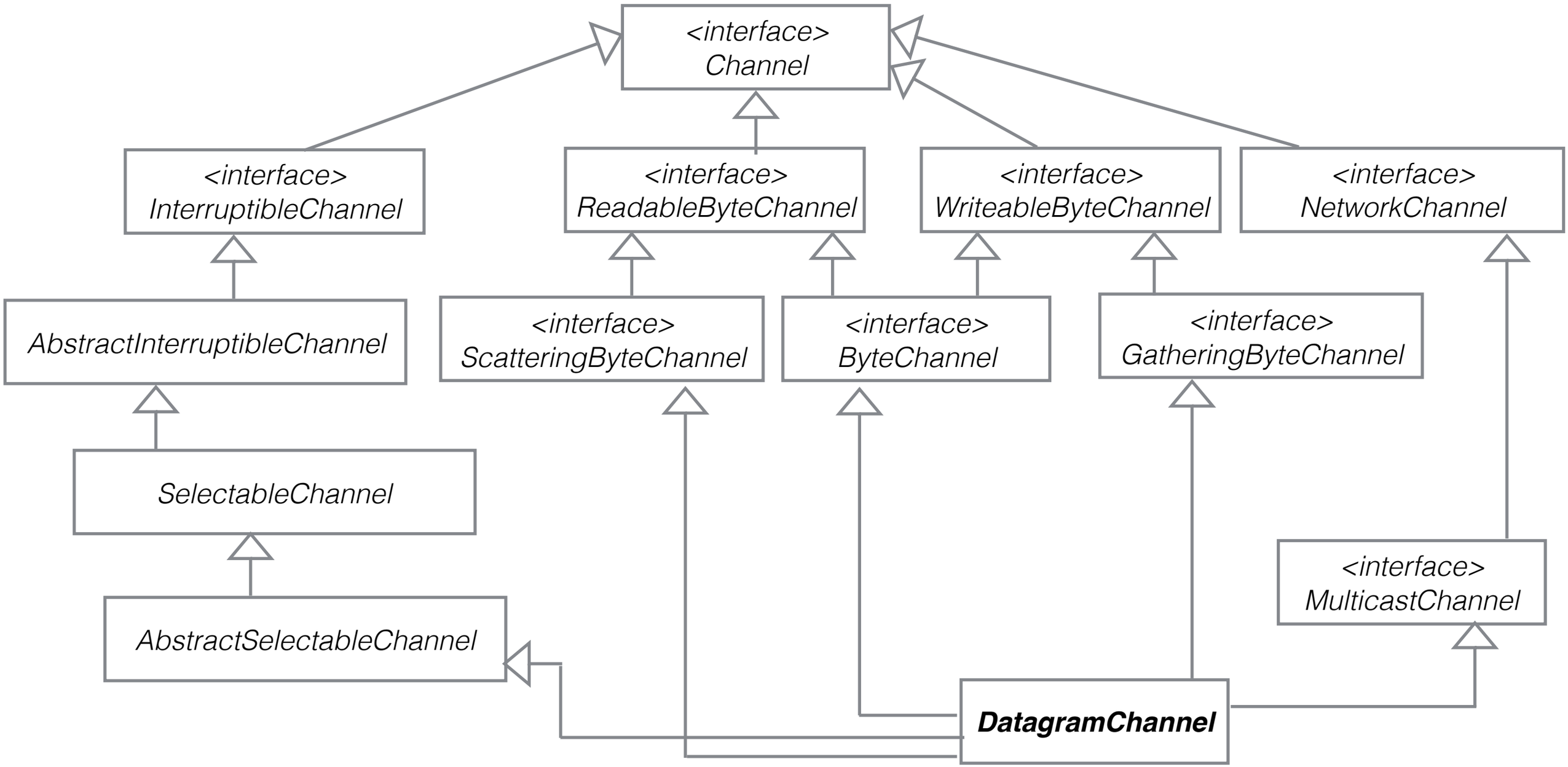
```
                if (key.isReadable()){
                    ReadingTask task = new ReadingTask(key, selector);
                    executor.submit(task);
                }
                if (key.isWritable()){
                    WritingTask task = new WritingTask(key);
                    executor.submit(task);
                }
            }
        }
    } catch (IOException e){
            System.out.println("General error: "+e.getMessage());
        }
    }
}
```

```java
public class ReadingTask implements Runnable {
    SelectionKey key;
    Selector selector;
    public ReadingTask(SelectionKey key, Selector selector){
        this.key=key;
        this.selector=selector;
    }
    public void run() {
        try{
            ByteBuffer buffer= ByteBuffer.allocate(FtpServerBuffer.BLOCK_SIZE);
            SocketChannel client=(SocketChannel)this.key.channel();
            client.read(buffer);
            String fileName= new String(buffer.array(), 0, buffer.position()).trim();
            System.out.println(Thread.currentThread()+"Requested file: "+fileName);
            buffer.clear();
            ArrayList<Object> attachment= new ArrayList<>();
        attachment.add(buffer);
        try{
            FileChannel file= FileChannel.open(Paths.get(fileName), StandardOpenOption.READ);
            buffer.putInt(0);
            attachment.add(file);
            client.register(this.selector, SelectionKey.OP_WRITE, attachment);
            System.out.println(Thread.currentThread()+"File exists. Sending file.");
        } catch (NoSuchFileException e){
            buffer.putInt(1);
            client.register(this.selector, SelectionKey.OP_WRITE,attachment);
        }
        buffer.flip();
    } catch (IOException e){
        System.out.println(Thread.currentThread()+"Error reading: "+e.getMessage());
        this.key.cancel();
    }}}
```

Gestione operazioni *read* funziona esattamente come prima, ma avviene in un *thread* separato.

```
public class WritingTask implements Runnable{
    SelectionKey key;
    public WritingTask(SelectionKey key){
        this.key=key;
    }
    public void run() {
        try{
            SocketChannel client=(SocketChannel)this.key.channel();
            ArrayList<Object> attachment=(ArrayList<Object>)this.key.attachment();
            ByteBuffer buffer = (ByteBuffer) ((ArrayList<Object>)this.key.attachment()).get(0);
            client.write(buffer);
            if (! buffer.hasRemaining()){ //wrote header
                if (attachment.size()>1){ // has file
                    FileChannel file= (FileChannel) (attachment).get(1);
                    long transfered=file.transferTo(file.position(),
                                file.size()-file.position(), client);
                    file.position(file.position()+transfered);
                    if(file.position()==file.size()){
                        file.close();
                        client.close();
                    }
                }else {
                    //file doesn't exist, close channel
                    client.close();
                }}
    } catch (IOException e){
        System.out.println("Error writing to client: "+e.getMessage());
        this.key.cancel();
    }}}
```

Gestione operazioni *write* funziona esattamente come prima, ma avviene in un *thread* separato.


<interface>
Channel
<interface>
InterruptibleChannel
<interface>
ReadableByteChannel
<interface>
WriteableByteChannel
<interface>
NetworkChannel
AbstractInterruptibleChannel
<interface>
ScatteringByteChannel
<interface>
ByteChannel
<interface>
GatheringByteChannel
SelectableChannel
<interface>
MulticastChannel
AbstractSelectableChannel
DatagramChannel

DatagramChannel

- Implementa interfaccia `MulticastChannel` - offre metodi per aderire a un gruppo multicast

`MembershipKey join(InetAddress group, NetworkInterface interface)`

Aderisce ad un gruppo *multicast* sulla interfaccia `interface`.

`MembershipKey join(InetAddress group, NetworkInterface interface, InetAddress source)`

Aderisce ad un gruppo *multicast* sulla interfaccia `interface`, per ricevere solo messaggi dall'indirizzo `source`. Solo se la piattaforma supporta source filtering. Il `MembershipKey` restituito incapsula tutte le informazioni sul gruppo *multicast* aderito. Per lasciare il gruppo si usa il metodo `void drop()` di questo oggetto.

`DatagramChannel`

- In più implementa `ByteChannel`, `ScatteringByteChannel` e `GatheringByteChannel` : può leggere, scrivere, fare *scatter* e *gather*.

- Implementa `NetworkChannel`: metodi per fare *bind* e settare opzioni.

- Estende la classe `AbstractSelectableChannel`: può fare operazioni non-bloccanti e può essere usato con selettori.

DatagramChannel

- Metodi addizionali:

`void connect()`

Connette il *channel* a una destinazione (nel senso UDP: non si stabilisce una connessione, però solo una verifica della destinazione o mittente). Un *channel* deve essere connesso per usare i metodi **`read`** e **`write`** del *channel* (altrimenti la destinazione/mittente non sono conosciuti)

`DatagramChannel disconnect()`

Disconnette il *channel*.

`SocketAddress getRemoteAddress()`

Restituisce l'indirizzo a cui il *channel* è connesso.

DatagramChannel

`SocketAddress receive(ByteBuffer data)`

Riceve un pacchetto su questo *channel*. Il `ByteBuffer` memorizza i dati ricevuti. Se i dati sono più dello spazio rimasto nel `ByteBuffer`, quelli rimanenti vengono scartati. Il metodo restituisce l'indirizzo del mittente.

`int  send(ByteBuffer data, SocketAddress target)`

Invia i datti del *buffer* in un pacchetto UDP verso l'indirizzo `target`. Se non c'è abbastanza spazio nei *buffer* di rete, in modo non-bloccante, nessun pacchetto viene inviato. Il metodo restituisce il numero di *byte* inviati (o tutti o 0).

Per questi due metodi non è obbligatorio che il *channel* sia connesso.

DatagramChannel

- Metodi statici per creare i *channel*:

`DatagramChannel open()`

Apre un *channel* specializzato in inviare pacchetti UDP.

`DatagramChannel open(ProtocolFamily family)`

Da usare per creare *channel multicast*. `family` può essere `StandardProtocolFamily.INET` o `StandardProtocolFamily.INET6` . Deve corrispondere all tipo di indirizzo del gruppo *multicast*.

Esempio: TimeServer con NIO

Cliente: bloccante

```
public class TimeClient {

    public static void main(String[] args) {
        ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);
        buffer.put((byte)0);
        buffer.flip();
        try(DatagramChannel channel= DatagramChannel.open();){
            channel.connect(new InetSocketAddress(
                InetAddress.getByName(TimeServer.HOST), TimeServer.PORT));
            channel.write(buffer);
            System.out.println("Sent request");
            buffer.clear();
            channel.read(buffer);
            buffer.flip();
            System.out.println("Received response: " + new Date(buffer.getLong()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

Server: non-bloccante

```
public class TimeServer {
    public static int PORT=2000;
    public static String HOST="localhost";

    public static void main(String[] args) {
        //setup buffer
        ByteBuffer receiveBuffer= ByteBuffer.allocate(1);
        ByteBuffer sendBuffer= ByteBuffer.allocate(Long.BYTES);
        //open channel
        try(DatagramChannel server= DatagramChannel.open()){
            server.bind(new InetSocketAddress(TimeServer.PORT));
            server.configureBlocking(false);

            //open selector
            Selector selector = Selector.open();
            //register channel with selector, including attachments
            LinkedList<SocketAddress> destinations=new LinkedList<>();
            server.register(selector,SelectionKey.OP_READ,destinations);

            while(true){
                selector.selectedKeys().clear();
                selector.select();
                for (SelectionKey key : selector.selectedKeys() ){
```

Inizialmente, *channel* viene registrato con il selettore solo per leggere.

Gestione lettura

```
if(key.isReadable()){
        //read
        System.out.println("received a datagram");
        DatagramChannel channel = (DatagramChannel) key.channel();

        SocketAddress client=channel.receive(receiveBuffer);
```

Leggo i dati e memorizzo l'indirizzo del cliente

```
        ((LinkedList<SocketAddress>)key.attachment()).add(client);
        key.interestOps(SelectionKey.OP_READ|SelectionKey.OP_WRITE);
        receiveBuffer.clear();
}
```

Aggiungo l'indirizzo del cliente nella lista del attachment, poi aggiungo l'operazione *write* nella lista di operazioni di interesse per il channel.

Gestione scrittura

```
if(key.isWritable()){
    //write
    System.out.println("sending a datagram");
    DatagramChannel channel = (DatagramChannel) key.channel();
    LinkedList<SocketAddress> dests=
            (LinkedList<SocketAddress>)key.attachment();
    if (dests.size()>0){
        sendBuffer.putLong(System.currentTimeMillis());
        sendBuffer.flip();
        channel.send(sendBuffer, dests.poll());
        sendBuffer.clear();
    }
                    }
                    if (dests.isEmpty()){
                        key.interestOps(SelectionKey.OP_READ);
                    }
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

Invio timestamp al primo indirizzo nella lista

Se non ci sono più clienti nella lista, rimuovo l'operazione write dalla lista di operazioni di interesse per il selettore (altrimenti il channel sarebbe sempre selezionato anche se io non ho niente da inviare)

Esempio: Multicast TimeServer con NIO

Cliente: operazioni bloccanti

```
public class MulticastTimeClient {

   public static void main(String[] args) {
      ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);
      try(DatagramChannel channel=
            DatagramChannel.open(StandardProtocolFamily.INET);){

         channel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
         channel.bind(new InetSocketAddress( MulticastTimeServer.PORT));
         channel.join(InetAddress.getByName(MulticastTimeServer.MC_GROUP),
               NetworkInterface.getByInetAddress(InetAddress.getLocalHost()));

         while(true){
            channel.receive(buffer);
            buffer.flip();
            System.out.println("Received time: " + new Date(buffer.getLong()));
            buffer.clear();
         }

      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Server: operazioni bloccanti

```
public class MulticastTimeServer {

    public static String MC_GROUP="239.255.1.1";
    public static int PORT=3000;

    public static void main(String[] args) {
        try(DatagramChannel server= DatagramChannel.open()){
            ByteBuffer sendBuffer= ByteBuffer.allocate(Long.BYTES);
            SocketAddress group= new InetSocketAddress(
                    InetAddress.getByName(MulticastTimeServer.MC_GROUP),
                     MulticastTimeServer.PORT);
            server.bind(null);
            server.connect(group);
            while(true){
                sendBuffer.putLong(System.currentTimeMillis());
                sendBuffer.flip();
                server.write(sendBuffer);
                sendBuffer.clear();
                System.out.println("Sent the time ");
                Thread.sleep(10000);
             }
        } catch (IOException e) {
             e.printStackTrace();
        } catch (InterruptedException e) {
             e.printStackTrace();
        }
    }
}
```

Conclusioni NIO

- NIO introduce nuove funzionalità per IO
  - Operazioni IO interrompibili
  - Ottimizzazione nel lavoro con *file* di grandi dimensioni
  - Funzionalità *non-blocking* per i *socket*, con modalità di selezione dei *channel* pronti
  - NOTA: l'effetto del comportamento *non-blocking* per UDP e molto meno visibile, siccome UDP è già più asincrono del TCP.

Esercizio - MiniChatRoom

- Gli utenti entrano senza registrarsi, ogni utente può inviare un messaggio, tutti gli utenti ricevono tutti i messaggi, senza sapere da chi vengono.
- *Server*:
    - Lancia un servizio *multicast* per trasmettere i messaggi ricevuti.
    - Aspetta connessioni TCP da clienti
    - Quando una connessione viene stabilita, il *server* aspetta dei messaggi dal cliente su quella connessione, fin che la connessione viene chiusa dal cliente
    - Ogni volta che un messaggio arriva, il *server* lo invia a tutti gli clienti usando *multicast*. Se il messaggio ricevuto e più lungo di 512 byte, viene troncato.
    - Tutti i *channel* devono essere multiplexati usando `Selector` in un solo *thread (NIO, channel* non-bloccanti).
    - Tip: il *channel multicast* può essere inizialmente registrato con il selettore usando `0` come `interestOps`, poi aggiungere l'operazione `SelectionKey.OP_WRITE` quando si riceve un messaggio da un cliente.

Esercizio - MiniChatRoom

- *Cliente*:
  - Aderisce al gruppo *multicast* per ricevere messaggi dal *server*.
  - Si connette al *server* e gli invia dei messaggi scritti dall'utente alla riga di comando, fin che l'utente scrive "exit".
  - Per ogni messaggio il cliente invia il testo del messaggio più il carattere '#' per segnalare la fine del messaggio.
  - La lettura dal *channel multicast* e la scrittura nel *channel* TCP si fa in un solo *thread*, usando `Selector`.
  - L'input dalla tastiera si gestisce in un altro *thread* (*thread* di lettura), usando una lista di messaggi per condividere l'input dell'utente con il *thread main*.
  - La possibile sovrapposizione dell'*input* del utente alla riga di comando con i messaggi arrivati dal server non deve essere trattata.
  - Tip: il *channel* TCP può essere inizialmente registrato con il selettore usando `0` come `interestOps`, poi aggiungere l'operazione `SelectionKey.OP_WRITE` nel *thread* di lettura quando si inserisce un messaggio nella lista di messaggi. Quando la lista di messaggi ridiventa vuota, gli `interestOps` devono diventare di nuovo uguali a `0`.