Non blocking socket

Introduzione

Il termine non bloccante, riferito a un socket, descrive tra l'altro la capacità dello strumento di inviare o ricevere una quantità di dati minore di quella che si vorrebbe, bizzarra caratteristica che assume un senso nella gestione della comunicazione tra più socket con un flusso di esecuzione unico. Sono altresì non bloccanti le operazioni di connessione ma è un aspetto molto meno interessante.

Chi blocca cosa

In un socket bloccante, nelle librerie standard della piattaforma Java, la lettura di dati corrisponde ad un'operazione di svuotamento del buffer associato al socket. L'operazione è bloccante in ciò che la lettura non restituisce il controllo del flusso finchè esistano dei dati nel buffer o non sia stata raggiunta la quantità di dati richiesti. In pseudo codice, a meno di eccezioni e segnali che indichino il completamento della trasmissione, l'operazione che segue:

```
socket bloccante read: buff[64 byte]
```

blocca il flusso di esecuzione, cioè impedisce l'esecuzione degli enunciati successivi a quello proposto, finchè:

a. non siano stati letti esattamente 64 byte oppure

b. sia stato letto un numero di byte pari alla quantità di byte presenti nel buffer del socket, minore dei 64 richiesti.

Per un socket non bloccante, la stessa situazione:

```
socket non bloccante read: buff[64 byte]
```

termina imprevedibilmente quando siano stati letti da zero a sessantaquattro byte. Naturalmente è una finta imprevedibiltà: la restituzione del controllo dipende da quello che è stato predisposto da chi abbia realizzato le librerie dalle quali emergono i socket non bloccanti. Non è difficile immaginare che possano esistere condizioni di restituzione del controllo indipendenti dalla quantità di byte letti, ad esempio nel caso in cui il metro di valutazione del se sia il tempo trascorso dalla richiesta di lettura: leggi quanto puoi in 50 millisecondi. In un'ottica più ampia, il termine non bloccante è riferito alla possibilità che un socket, anzichè attendere il completamento di un'operazione che comporterebbe un blocco condizionato del flusso di esecuzione, restituisca il controllo prima che l'operazione richiesta sia stata pienamente soddisfatta.

Gli ingredienti della ricetta

La comunicazione attraverso socket non bloccanti richiede più di un componente ma i tre protagonisti per genere sono i canali socket, il selettore e le chiavi di selezione. Fortuna vuole che le loro funzioni ed il loro ruolo siano facilmente comprensibili.

SocketChannel e ServerSocketChannel

Un SocketChannel è la combinazione di un socket ed un canale di trasmissione bidirezionale. Dato il nome, la cosa ha poco di sorprendente.

Un ServerSocketChannel è... la combinazione di un server socket ed un canale di trasmissione bidirezionale.

Entrambi sono figli di AbstractSelectableChannel e da questa relazione mutuano la capacità di passare dalla modalità bloccante a quella non bloccante. In modalità bloccante funzionano come i vecchi Socket e ServerSocket con la comodità e l'eventuale rapidità della scrittura e lettura a colpi di Buffer. In modalità non bloccante alcuni dei loro metodi assumono le strane caratteristiche riassunte due sezioni fa. La gestione del comportamento non bloccante, che risulta oltremodo curiosa se confrontata all'uso delle speculari proprietà bloccanti, diventa assolutamente elementare quando avvenga nel contesto offerto da un selettore. Credo che la questione fondamentale per comprendere i socket non bloccanti sia proprio questa. Da soli appaiono nulla più di un arzigogolo buono per affascinanti teorie sulla comunicazione di rete. La loro praticità emerge solo quando siano abbinati agli altri due compagni di viaggio: il selettore e le chiavi di selezione. Insieme, a mio parere, surclassano per semplicità d'uso, i predecessori bloccanti.

Il selettore

Il selettore è una specie di ruota della fortuna che accetta di quando in quando l'aggiunta e la rimozione di candidati alla vittoria. Per noi i malcapitati rotanti possono essere ServerSocketChannel o SocketChannel ma le API da per buona ogni specie di SelectableChannel. I canali sono registrati presso un selettore invocando i loro metodi register, che richiedono come argomento anche un Selector. Per poter essere registrato, un canale deve essere non bloccante, altrimenti è rilasciata un'eccezione IllegalBlockingModeException. La registrazione restituisce una chiave, SelectionKey. La chiave è ciò che rappresenta il canale nel selettore: vedremo tra poco come funzioni. Il selettore è destinato ad essere usato in un motore di selezione. Intendo per motore di selezione un semplice ciclo continuo scandito dalle operazione di selezione e scorrimento delle chiavi attive nel selettore. La condizione minima di operatività di un selettore è che questo sia aperto. Un selettore può ricevere nuove chiavi in esecuzione dunque all'avvio può essere "vuoto", cioè può non possedere alcun canale registrato.

Le chiavi di selezione

Una chiave di selezione è fatta di tre parti:

a. un'operazione;

b. un canale;

c. un allegato.

L'operazione è un segnale che la chiave usa per comunicare al selettore la natura del proprio compito. Durante lo scorrimento delle chiavi, il selettore rende disponibili quelle chiavi che manifestino un interesse per un'operazione che il selettore giudica correntemente applicabile.

Una chiave di selezione è nient'altro che un rappresentante e il canale associato alla chiave è il rappresentato. Se il selettore giudica applicabile l'operazione di lettura, per cui la chiave abbia dichiarato il proprio interesse, significa che è possibile leggere dati dal canale associato alla chiave. Se il selettore giudica applicabile l'operazione di scrittura, per cui la chiave abbia dichiarato il proprio interesse, significa che è possibile scrivere dati sul canale associato alla chiave. E così via.

L'allegato è un riferimento di tipo Object che, in Java, significa una cosa qualsiasi. L'esistenza di un allegato alla chiave o, meglio, di un allegato al canale accessibile attraverso la chiave rappresentante, ha l'obiettivo di semplificare l'accesso a informazioni inerenti lo stato pregresso delle operazioni sul canale. La natura dei socket non bloccanti richiede necessariamente che le operazioni di lettura o scrittura siano considerate non atomiche, vale a dire che non è possibile presumere che la scrittura o lettura di un certo numero di byte avvenga per effetto di una sola istruzione. Lo si è detto all'inizio: leggere 64 byte da un socket non bloccante significa ottenere per tutta risposta un numero variabile da zero a 64 byte letti. Naturalmente esitono ottime ragioni per cui uno potrebbe voler ottenere un certo numero di byte dal canale: è possibile che il protocollo di comunicazione usi tale numero per stabilire quando si possa considerare pienamente disponibile un messaggio o che, più in generale, la quantità sia l'interruttore che attiva una reazione del sistema, com'è il caso di un download manager. L'allegato consente di tenere un'agevole traccia di quanto è stato fatto affinchè due operazioni successive della stessa specie possano essere accumulate in vista di un comune risultato. Prendiamo il caso della lettura di questi 64 byte. L'allegato al canale potrebbe essere qualcosa che ci consenta di sapere quanti byte debbano essere letti prima di poter dire di aver completato la lettura e che ci offra un buffer in cui accumulare i byte restituiti da una lettura non bloccante.

```
Allegato alla lettura
  buffer del messaggio completo
  quanti byte restano per completare il messaggio
```

Avendo a disposizione questo genere di allegato, la composizione del messaggio attraverso letture non bloccanti potrebbe essere così riassunta:

```
ogni volta che è possibile leggere dalla chiave del tal canale
  leggi quel che puoi
  prendi l'allegato
    accumula nel buffer dell'allegato i byte letti in questo passaggio
    se, nell'allegato, il numero di byte che restano da leggere è zero
      allora la lettura è terminata
    altrimenti
      è necessario un altro passaggio nel selettore
```

Nonostante gli encomiabili sforzi della letteratura in materia, la cosa appare tutt'altro che complicata.

Applicazione

Creare un server che usi i canali socket non bloccanti è semplicissimo. Iniziamo dichiarando un javax.nio.channels.ServerSocketChannel.

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
```

Il passo successivo è collegare il ServerSocketChannel ad un indirizzo di ascolto.

```
InetSocketAddress boundAddress = new InetSocketAddress("localhost", 9090);
serverChannel.socket().bind(boundAddress);
```

Configuriamo il ServerSocketChannel in modalità non bloccante così che l'accettazione di una connessione da parte di un socket client risulti, appunto, non bloccante.

```
serverChannel.configureBlocking(false);
```

Ora inseriamo il ServerSocketChannel nel Selettore. Più propriamente, registriamo nel selettore l'interesse del ServerSocketChannel all'accettazione di nuove connessioni. Il mezzo di registrazione è un metodo che ServerSocketChannel possiede in quanto SelectableChannel. Tale metodo è *register(Selector, int, Object)*. Il primo argomento è il selettore che riceve la registrazione, il secondo argomento è una combinazione delle bandiere:

```
SelectionKey.OP_ACCEPT;
SelectionKey.OP_CONNECT;
SelectionKey.OP_READ;
SelectionKey.OP_WRITE.
```

Il terzo argomento è un qualsiasi oggetto che farà da allegato alla registrazione. L'operazione che interessa al nostro ServerSocketChannel è l'accettazione di nuove connessioni, OP_ACCEPT.

La registrazione di serverChannel presso selector avviene con:

```
serverChannel.register(selector, SelectionKey.OP_ACCEPT, null);
```

Cosa significa questo comando. Significa che ora il selettore è al corrente del fatto che esiste un certo canale selezionabile e che, a richiesta, dovrà essere restituita la chiave di seleziona associata a quel canale quando il canale corrispondente sia nelle condizioni di accettare una connessione. Il prosieguo del nostro discorso sta proprio in quel "a richiesta".

Finchè il selettore è attivo, possiamo chiedergli di selezionare quelle chiavi, tra tutte quelle presenti nel suo registro, quelle che siano pronte ad eseguire l'operazione o le operazioni per cui abbiano manifestato interesse. Nel caso del nostro serverChannel, la sua chiave sarà selezionata quando il serverChannel dovrà accettare una richiesta di connessione da parte di un client.

Finchè il selettore è attivo...

```
while(selector.isOpen()) {
```

...chiediamo al selettore di fare il suo lavoro...

```
  selector.selectNow();
```

Il metodo selectNow è l'omologo non bloccante di select. Il prodotto della selezione è accessibile attraverso il metodo selectedKeys di selector. Una volta selezionata, perchè pronta ad eseguire l'operazione associata, una chiave resta nell'insieme delle chiavi selezionate. Per evitare di trovarsela tra i piedi nei passaggi successivi, la chiave deve essere rimossa da quell'insieme. Vediamo come continuando il nostro esempio. Dopo aver registrato un ServerSocketChannel in un Selector, abbiamo creato un ciclo condizionato dallo stato di apertura del selettore. La prima istruzione di quel ciclo è una richiesta al selettore: vedi un po' se c'è qualche canale che può fare quello che vorrebbe. Se c'è, il selettore infila la sua chiave nell'insieme delle chiavi selezionate, sempre che non ci sia già in virtù di selezioni precedenti. Ora scorriamo l'insieme di quelle chiavi.

```
  Iterator<SelectionKey> selectedKeys = selector.selectedKeys().iterator();
```

Usiamo l'iteratore per via della sua capacità di scorrere tra gli elementi di un insieme e rimuovere un elemento da quell'insieme durante lo scorrimento.

```
  while(selectedKeys.hasNext()) {
    SelectionKey key = selectedKeys.next();
    selectedKeys.remove();
```

Ricordo, *en passant*, che il metodo *remove()* di java.util.Iterator elimina dall'insieme bersaglio dello scorrimento l'elemento corrente, cioè l'elemento restituito dall'ultima invocazione di *next()* precedente il *remove()*.

Nella prima delle nuove righe di codice, chi è "key"? Per il momento non può essere altri che la chiave di selezione associata al nostro serverChannel. Ma il selettore serve per gestire una moltitudine di canali selezionabili. Non è assolutamente anomalo che un sistema concreto richieda più di un ServerSocketChannel. Ed è certo che prima o poi ci finirà anche qualche client con il suo SocketChannel. Insomma, key può essere il facente funzioni di una varietà di canali. Il primo passo è scoprire perchè il

selettore abbia scelto di infilare questa chiave tra quelle selezionabili. Una SelectionKey ha dei metodi ad hoc per sapere quale sia l'operazione che ha causato la selezione. Ad esempio, guarda caso, se la chiave fosse stata creata dalla registrazione di un ServerSocketChannel per un'operazione OP_ACCEPT, la chiave selezionata restituirebbe true all'invocazione di *isAcceptable()*.

```
    if(key.isAcceptable()) {
```

Se la chiave è "accettabile" significa che il canale sottostante è pronto per accettare la richiesta di connessione di un client. Ottenere il canale del client è cosa rapida. Prima prendiamo il canale associato alla chiave di selezione. Supponendo che nel nostro programma i canali in grado di "accettare" siano ServerSocketChannel, cosa piuttosto comune, possiamo fare una conversione esplicita.

```
      ServerSocketChannel server =
          (ServerSocketChannel)key.channel();
```

Poi prendiamo il SocketChannel che sta tentando di connettersi.

```
      SocketChannel client = server.accept();
```

Sappiamo che, essendo il nostro ServerSocketChannel non bloccante, il metodo *accept()* non si ferma ad aspettare che qualcuno si connetta: se c'è un candidato lo restituisce altrimenti passa la mano e sputa un null. Per come siamo messi, il SocketChannel restituito è sempre diverso da null. E' il selettore a dirci che qualcuno sta tentando di connettersi, c'è da sperare che non racconti frottole. Ricordate quel "la chiave corrente restituita dall'iteratore va rimossa"? Cosa succederebbe se non la rimuovessimo? Capiterebbe che la chiave, vale a dire il ServerSocketChannel, resterebbe nell'insieme delle selezionate, cioè di quelle possono fare ciò che dicono di voler fare. Il metodo di selezione non bloccante *selectNow()*, che abbiamo usato, fa proseguire il flusso di controllo a prescindere dal fatto che sia stato qualche cambiamento nei candidati alla selezione. Il nostro programma continuerebbe a girare ed il nostro iteratore restituirebbe ogni volta, tra l'altro, anche il ServerSocketChannel. La sua chiave ci direbbe che sta ricevendo una richiesta di connessione ma è un effetto della richiesta del tempo che fu, perdurante per via della mancata rimozione dall'insieme delle chiavi selezionate. Allora il metodo accept(), non bloccante, ci restituirebbe un bel null. Lo riassumo in codice, in modo che sia evidente la differenza.

```
while(selector.isOpen()) {
  selector.selectNow();
  for(SelectedKey key : selector.selectedKeys()) {
    if(key.isAcceptable()) {
      ServerSocketChannel server =
        (ServerSocketChannel)key.channel();
      SocketChannel client = server.accept();
    }
  }
}
```

Qui "client" può essere null. Lo sarà ogni volta che, dopo la prima richiesta di connessione, che fa entrare la chiave del ServerSocketChannel nell'insieme di selezione, non ci sia un'effetiva ulteriore connessione da parte di un client. Perchè? Perchè la chiave selezionata permane nell'insieme di selezione. Questo è il tipo di fenomeni prevenuto dalla rimozione della chiave a cui abbiamo prima accennato e che è qui di seguito evidenziata.

```
while(selector.isOpen()) {
  selector.selectNow();
  Iterator<SelectionKey> selectedKeys = selector.selectedKeys().iterator();
  while(selectedKeys.hasNext()) {
    SelectionKey key = selectedKeys.next();
    key.remove();
    if(key.isAcceptable()) {
      ServerSocketChannel server =
        (ServerSocketChannel)key.channel();
      SocketChannel client = server.accept();
    }
```

```
  }
}
```

Registrazione di un client

Siamo arrivato al punto in cui un server socket ha accettato la connessione di un client. Al pari di ServerSocketChannel, SocketChannel è un SelectableChannel: è materia manovrabile da un Selector. Come abbiamo fatto per serverSocket, così facciamo per il client: configuriamo il SocketChannel in modalità non bloccante e lo registriamo presso il selettore. Lettura e scrittura, OP_READ | OP_WRITE. Accettiamo temporaneamente le linee di codice che seguono. Vedremo in seguito come la lettura e la scrittura richiedano qualche accorgimento per via del comportamento non bloccante.

```
      client.configureBlocking(false);
      client.register(
        selector,
        SelectionKey.OP_READ | SelectionKey.OP_WRITE,
        null);
```

A questo punto il client è entrato a far parte dei candidati alla selezione del selettore. Quando il selettore rileverà la possibilità di leggere dal client la chiave del client sarà immessa nell'insieme di selezione e l'invocazione di *isReadable()* su quella chiave restituirà true. Quando il selettore rileverà la possibilità di scrivere sul client la chiave del client sarà immessa nell'insieme di selezione e l'invocazione di *isWritable()* su quella chiave restituirà true. Notiamo che i metodi *isReadable()*, *isWritable()* eccetera operano non in riferimento alle operazioni a cui la chiave è interessata ma basandosi sulle operazioni che la chiave è pronta ad eseguire.

La registrazione rende disponibile il canale per la lettura e scrittura e tale disponibilità può essere schematicamente sfruttata come segue.

```
while(selector.isOpen()) {
  selector.selectNow();
  Iterator<SelectionKey> selectedKeys =
    selector.selectedKeys().iterator();
  while(selectedKeys.hasNext()) {
    SelectionKey key = selectedKeys.next();
    key.remove();
    if(key.isAcceptable()) {
      ServerSocketChannel server =
        (ServerSocketChannel)key.channel();
      SocketChannel client = server.accept();
      client.register(
        selector,
        SelectionKey.OP_READ | SelectionKey.OP_WRITE,
        null);
    }
    if(key.isReadable()) {
      WritableByteChannel out = (WritableByteChannel)key.channel();
      //leggi da out
    }
    if(key.isWritable()) {
      ReadableByteChannel in = (ReadableByteChannel)key.channel();
      //scrivi su in
    }
  }
}
```

Se il canale è non bloccante, lettura e scrittura richiedono qualche accorgimento in più.

Lettura e scrittura per i canali non bloccanti.

Scrivere dei dati su un canale è elementare. Si prendono i dati, li si impacchetta in un ByteBuffer e si invia quel ByteBuffer lungo il canale tramite il metodo *write(ByteBuffer)* del canale. Per una stringa di testo, la cosa è riassumibile nel modo che segue.

```
String text = "Hello world!!!";
Charset charset = Charset.forName("UTF-8");
ByteBuffer encodedText = (ByteBuffer)charset.encode(text);
ByteBuffer message = ByteBuffer.allocate(encodedText.limit() + 4);
message.putInt(encodedText.limit());
message.put(encodedText);
message.rewind();
while(message.hasRemaining()) {
  writableByteChannel.write(message);
}
```

L'unica cosa che può apparire strana è il fatto che il messaggio sia composto dalla lunghezza del testo codificato seguito dal testo vero e proprio. E' una convenzione arbitraria di comunicazione scelta dal sottoscritto. Un protocollo, per dirla con un termine diffuso. Invio la lunghezza del messaggio per ogni messaggio, in modo tale che, dall'altra parte, chi legga possa sapere quanti byte prelevare dal canale per ricostruire il messaggio. Questa convenzione consente di scrivere messaggi di lunghezza ignota. Naturalmente è possibile adottare convenzioni diverse, ad esempio tutti i messaggi potrebbero avere la stessa lunghezza e, in questo caso, non sarebbe più necessario il numero totale di byte ad ogni comunicazione. Il punto che interessa è un altro.

```
while(message.hasRemaining()) {
  writableByteChannel.write(message);
}
```

Questo approccio alla scrittura va bene per un canale bloccante. Dice "finchè non hai consumato il buffer di byte da inviare, scrivi sul canale i byte che non hai ancora inviato". Non va nel caso in cui il canale sia non bloccante. Non perchè non funzioni ma perchè contraddice la scelta di rendere le operazioni di scrittura non bloccanti. La scelta coerente ha la forma diversa del codice successivo.

```
writableByteChannel.write(message);
```

Significa "non scrivere finchè il buffer del socket è pieno o finchè non hai esaurito i byte del messaggio ma solo quel tanto che puoi". "Quel tanto che puoi" dipende dalle librerie ma è un tanto che, idealmente, dovrebbe consentire ad una quantità di canali di scrivere ognuno un po' di ciò che devono.

Scrivere nonostante, o grazie a, questa configurazione non è affatto complicato. Tutto quello che serve è poter conservare tra una scrittura e l'altra lo stato dei tentativi precedenti. Quanti byte devono essere scritti, quanti ne sono già stati inviati, quanti ne restano da spedire. Tutte informazioni già disponibili in un solo ByteBuffer. Ci sarebbe il problema di trasportare questo ByteBuffer insieme al canale, per poter avere lo stesso riferimento tra una selezione e l'altra. Guarda caso, esiste l'allegato. Propongo un semplice esempio di scrittura tramite socket non bloccante ed allegato usando un client. Salto a piè pari la gestione delle eccezioni, cosa in sè criminale ma passatemela come licenza poetica.

Creiamo un client non bloccante che invia la stringa "Hello World". Ci atteniamo al protocollo di comunicazione su citato secondo cui i primi quattro byte di un messaggio indicano la quantità di byte successivi al quarto che occorre leggere per poter dire di avere in mano tutti i dati del testo inviato.

```
[4 byte = N][N byte = testo UTF-8]
```

Il primo passo di questo esempio è prendere il messaggio e impacchettarlo.

```
ByteBuffer message = packMessage("Hello world!");
```

Il metodo packMessage prende il testo e lo immette in un messaggio corrispodente al protocollo su riportato. I primi quattro byte formano un int Java che rappresentano la lunghezza, in byte, del testo codificato in UTF-8. Il resto del pacchetto di byte contiene il messaggio di testo.

```
private ByteBuffer packMessage(String text) {
  ByteBuffer data = (ByteBuffer)Charset.forName("UTF-8").encode(text);
  int length = data.remaining();
  ByteBuffer pack = ByteBuffer.allocate(length + 4);
  pack.putInt(length).put(data).rewind();
  return pack;
}
```

Apriamo un Selettore, e creiamo un SocketChannel non bloccante.

```
Selector selector = Selector.open();
SocketChannel client = SocketChannel.open();
```

Impostiamo il canale per operare in modo non bloccante.

```
client.configureBlocking(false);
```

Creiamo l'indirizzo di connessione.

```
InetSocketAddress address = new InetSocketAddress("localhost", 9090);
```

Connettiamo il socket all'indirizzo prescelto.

```
boolean connected = client.connect(address);
```

Passiamo quindi a registrare il canale presso il selettore.

```
if(connected) {
  client.register(selector, OP_READ | OP_WRITE, message);
} else {
  client.register(selector, OP_CONNECT, message);
}
```

Qui vale la pena fare due chiacchiere. Per un SocketChannel non bloccante, la connessione è una procedura in due tempi. Il primo passaggio si ottiene con la richiesta di connessione del metodo *connect()*. Il secondo passo è terminare la procedura di connessione, invocando il metodo *finishConnect()*. Un Selettore è in grado di dirci quando un SocketChannel che abbia iniziato una richiesta di connessione è pronto a terminare le procedure di connessione. Può dirlo nel modo in cui un selettore comunica ciò che ha da dire: selezionando la chiave del canale che manifesti interesse per quell'operazione. L'operazione di cui si parla è quella della bandiera SelectionKey.OP_CONNECT. Il selettore restituirà la chiave del nostro SocketChannel quando sia giunto il momento di invocare il metodo finishConnect(). L'invocazione di finishConnect() per un SocketChannel non bloccante ha tre possibili risultati. Uno è restituire true. Significa che il socket ha terminato con successo la connessione ed è pronto per le trasmissioni. Un altro è restituire false. Significa che il procedimento di connessione non è ancora terminato. L'ultimo è il rilascio di un'eccezione IOException. Significa che il tentativo di stabilire una connessione è fallito, ad esempio perchè il server non è attivo o l'indirizzo è sbagliato. Tra poco vedremo l'invocazione di questo *finishConnect()*. Non è detto che, nonostante sia non bloccante, la connessione non possa essere stabilita in un sol colpo. In questo caso, rappresentato nel codice dal valore true di connected, il canale non ha interesse a terminare la connessione ma direttamente alla lettura e scrittura. Notiamo che la registrazione del canale presso il selettore è stata fatta usando il ByteBuffer message come allegato. Allegato alla chiave di selezione creata dal selettore per questo SocketChannel. L'allegato accompagna la chiave di selezione ed è accessibile attraverso il metodo *attachment()* di SelectionKey. Il codice prosegue in modo familiare.

```
while(selector.isOpen()) {
```

```
selector.selectNow();
Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
while(keys.hasNext()) {
  SelectionKey key = keys.next();
  keys.remove();
```

Se il canale associato alla chiave è pronto a terminare la connessione...

```
if(key.isConnectable()) {
```

...prendiamo il canale associato, che sappiamo essere un SocketChannel, perchè è l'unica cosa che abbiamo messo nel selettore, e terminiamo la connessione. Essendo il SocketChannel non bloccante, l'operazione può non terminare subito. In questo caso il metodo *finishConnect()* restituirà false e dovremo ripetere più tardi lo stesso tentativo. Trascuriamo per semplicità il caso di connessione fallita che noteremmo intercettando un'eccezione IOException.

```
SocketChannel client = (SocketChannel)key.channel();
client.finishConnect();
key.interestOps(OP_READ |  OP_WRITE);
```

Le linee precedenti dicono questo: se il socket channel client è in grado di terminare le operazioni di connessione, allora cambia l'insieme di operazioni che comportano la selezione del canale. Non più OP_CONNECT, perchè la fase di connessione è terminata con l'invocazione di *finishConnect()*, ma OP_READ e OP_WRITE.

Dopo la connessione, passiamo alla scrittura del messaggio. Ciò che deve essere scritto sta nell'allegato alla chiave. Deve essere spedito quando il selettore ritenga il canale pronto alla trasmissione.

```
if(key.isWritable()) {
```

Qui si tratta semplicemente di prendere il ByteBuffer contenente il messaggio e scriverlo sul canale.

```
ByteBuffer data = (ByteBuffer)key.attachment();
SocketChannel out = (SocketChannel)key.channel();
if(data.hasRemaining() && out.finishConnect()) {
  out.write(data);
}
```

Il metodo *write(ByteBuffer)* di un channel preleva i byte dal buffer. Il prelievo aggiorna la posizione del cursore del buffer. La posizione del cursore nel buffer rispetto alla quantità di dati in esso contenuti determina se il buffer abbia o meno dei dati restanti. E' come se "write" svuotasse progressivamente il buffer scritto. Sappiamo che, essendo il nostro SocketChannel non bloccante, non è detto che il buffer sia svuotato del tutto. Le istruzioni che abbiamo scritto dicono che ogni volta che il selettore rilevi la possibilità di scrivere dati lungo il canale, quel canale deve scrivere dei byte prelevandoli dal ByteBuffer allegato alla sua chiave. Ogni scrittura riduce il numero di byte non ancora scritti. Quando tutti i byte sono stati inviati, cioè *hasRemaining()* restituisce false, il messaggio è stato interamente inviato. Il tutto è colorato dalla condizione *finishConnect()*. Per un SocketChannel non bloccante anche l'operazione che conclude la connessione è non bloccante. L'invocazione di *finishConnect()* contenuta nella fase di gestione dell'operazione OP_CONNECT non è detto che abbia reso il canale effettivamente connesso. Ciò che ha fatto è stato dichiarare conclusa quella fase. Avremmo potuto omettere, nel momento in cui tentiamo di scrivere i dati sul canale, il controllo dell'effettiva connessione? Si da punto di vista delle funzioni, no da quello della coerenza. Se si prova a scrivere su un canale, bloccante o non bloccante, dopo l'invocazione di finishConnect ma prima che il canale sia effettivamente pronto per la trasmissione, la scrittura causa un blocco del flusso di controllo fintantochè le operazioni di connessione non siano terminate. L'invocazione semplice di *finishConnect()*, invece, è non bloccante e, dal nostro punto di vista, restituisce true se sia possibile scrivere senza interruzioni o false se, leggendo o scrivendo, saremmo costretti ad attendere.

L'invio del messaggio conclude lo scopo del nostro codice. Lasciamo eccezioni, disconessione,

liberazione delle risorse di basso livello nelle mani degli dèi e ricapitoliamo il codice.

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;
import static java.nio.channels.SelectionKey.*;

public class Client {
  public static void main(String[] args) throws Throwable {
    new Client().start();
  }

  public void start() throws Throwable {
    /* Impacchetta il messaggio */
    ByteBuffer message = packMessage("Hello World!!!");

    /* Apre un selettore */
    Selector selector = Selector.open();

    /* Apre un canale */
    SocketChannel channel = SocketChannel.open();

    /* Imposta il canale in modalità non bloccante */
    channel.configureBlocking(false);

    /* Crea un indirizzo di connessione */
    InetSocketAddress address = new InetSocketAddress("localhost", 9090);

    /* Connette il canale e secondo il risultato stabilisce
    le operazioni di interesse per il canale */
    int ops = channel.connect(address) ?
      OP_READ | OP_WRITE : OP_CONNECT;

    /* Registra il canale presso il selettore */
    channel.register(  selector, ops,   message);

    /* Finchè il selettore è aperto */
    while(selector.isOpen()) {

      /* Richiede una selezione al selettore */
      selector.selectNow();

      /* Ottiene un iteratore sulle chiavi
      selezionate */
      Iterator<SelectionKey> keys =
        selector.selectedKeys().iterator();

      /* Scorre le chiavi selezionate */
      while(keys.hasNext()) {

        /* Prende una chiave... */
        SelectionKey key = keys.next();

        /* ...e la rimuove dall'insieme di selezione */
        keys.remove();

        /* Il canale associato alla chiave è pronto
        a terminare le operazioni di connessione? */
        if(key.isConnectable()) {

          /* Ottiene il canale associato alla chiave */
          SocketChannel client = (SocketChannel)key.channel();

          /* Conclude le operazioni di connessione */
          client.finishConnect();
```

```
            /* Cambia le operazioni a cui è interessato
            il canale rappresentato dalla chiave key */
            key.interestOps(OP_READ | OP_WRITE);
          }

          /* Il canale associato alla chiave è pronto
          alla scrittura? */
          if(key.isWritable()) {

            /* Ciò che deve essere ancora scritto è contenuto
            nell'allegato alla chiave, in forma di ByteBuffer */
            ByteBuffer data = (ByteBuffer)key.attachment();

            /* Ottiene il canale associato alla chiave */
            SocketChannel out = (SocketChannel)key.channel();

            /* Se restano dei dati da scrivere e il canale può
            scrivere senza attendere che la connessione sia
            effettivamente stabilita... */
            if(data.hasRemaining() && out.finishConnect()) {

              /* ...scrive un po' dei byte del messaggio */
              out.write(message);
            }
          }
        }

        /* Il metodo selectNow() di Selector è non bloccante.
        Il Thread corrente esegue il ciclo del selettore a rotta
        di collo il che generalmente tende a deprimere la CPU.
        Basta una piccola pausa per renderla più felice. */
        Thread.sleep(10);
      }
    }

    /* Restituisce un ByteBuffer che contiene il numero di byte
    che compongono il messaggio, codificato in formato UTF-8. I primi
    quattro byte del buffer si riferiscono ad un intero che rappresenta
    la lunghezza del testo codificato, in byte */
    private ByteBuffer packMessage(String text) {
      ByteBuffer data = (ByteBuffer)
        Charset.forName("UTF-8").encode(text);
      int length = data.remaining();
      ByteBuffer pack = ByteBuffer.allocate(length + 4);
      pack.putInt(length).put(data).rewind();
      return pack;
    }
}
```

A rischio di essere pedante, riassumo quello che succede in quel codice e perchè deva succedere.

Impacchettiamo in buffer di byte un messaggio di testo. Il buffer di byte è ciò che sarà inviato attraverso il socket channel. Il buffer di byte sarà allegato alla chiave di selezione del canale. La scrittura avviene prendendo l'allegato, cioè il buffer di byte, e usandolo come parametro del metodo write del socket. Il socket scriverà nessuno, alcuni o tutti i byte contenuti del buffer. Non sappiarmo a priori quanti byte siano inviati da un write non bloccante. Ciò che sappiamo è che per effetto del metodo write e secondo le proprietà di un ByteBuffer, l'invio causa un esaurimento progressivo della quantità di byte rimanenti nel buffer – a patto che il buffer non sia riavvolto ma non è il nostro caso. Riteniamo concluso l'invio, cioè evitiamo di invocare altre volte il metodo write, quando il buffer di byte allegato alla chiave di selezione è esaurito.

Apriamo un selettore. Del selettore ci interessa la capacità di operare ciclicamente, la predisposizione a gestire capacità diverse di un solo canale e la possibilità di controllare, attraverso le chiavi, il passaggio dal momento in cui leggiamo i dati dal canale al momento in cui scriviamo i dati su quel canale.

Apriamo un SocketChannel e lo impostiamo in modalità non bloccante.

Connettiamo il socket ad un indirizzo. Se il nostro socket fosse stato bloccante allora il flusso di controllo non avrebbe passato la "barriera" dell'invocazione di connect finchè il SocketChannel non fosse effettivamente connesso o non fosse stata sollevata una qualche eccezione. Per un socket non bloccante l'operazione di connessione è a sua volta non bloccante: il flusso di controllo passa subito all'enunciato seguente. Il risultato del metodo *connect*, non bloccante, ci informa se la connessione sia immediatamente stabilita o se sia necessario un ulteriore passaggio per il metodo *finishConnect()*. Nel primo caso il socket è pronto per la lettura e la scrittura e possiamo immetterlo nel selettore come interessato a OP_READ | OP_WRITE, cioè lettura e scrittura. Nel secondo caso dobbiamo terminare le operazioni di connessione prima di iniziare a leggere o scrivere e dobbiamo immetterlo nel selettore associato all'operazione OP_CONNECT, che forse avrebbe dovuto chiamarsi OP_FINISH_CONNECT.

Dopo la registrazione, iniziamo il ciclo sul selettore. Finchè il selettore è aperto. Un selettore può essere disattivato attraverso l'invocazione del suo metodo *close()*. Un'eventuale invocazione di *close()* su selector determinerebbe quindi l'uscita del flusso di controllo dal ciclo while.

Il ciclo inizia con una richiesta al selettore: *selectNow()*, cioè "vedi un po' se c'è qualche canale pronto per fare almeno una delle operazioni per cui si è candidato". L'invocazione di *selectNow()* è non bloccante, a differenza di *select()* o *select(long millis)*. Dove sta la praticità di una selezione non bloccante? La praticità sta in ciò che la restituzione immediata del controllo ci consente di immettere nel ciclo del selettore e, quindi, nelle mani dell'unico Thread che gestisce il nostro server, operazioni che devono essere compiute a prescindere dall'operatività di un canale. Non è difficile immaginare quali siano queste ulteriori operazioni. Il selettore è Thread-safe, cioè i suoi metodi possono essere invocati da più Thread. Le sue chiavi, ed in particolare le sue chiavi attive, non lo sono. Alla fine della fiera, l'unico Thread autorizzato a maneggiare le chiavi di selezione, a meno di accorgimenti vari che peraltro renderebbero discutibile la scelta di gestire la comunicazione con un solo Thread, è quello che fa girare il selettore. Ora, da qualche parte i messaggi spediti ai client devono arrivare. In una chat l'utente scrive e quando preme invio il messaggio parte. Non è il Thread del selettore che gestisce la digitazione sull'area di testo o sulla console: ha già il suo bel da fare manca solo che si debba occupare anche della produzione dei messaggi. Un altro Thread crea i messaggi da inviare o gestisce i messaggi giunti al server. Per comunicare quei messaggi e quelle risposte rende disponibile al Thread del selettore i dati necessari affinchè questo possa capire quale delle chiavi del selettore debba essere attivata o quale canale tra quelli associati ad una chiave già disponibile debba trasmettere i dati del messaggio.

Evidenziamo anche la necessità di imporre una piccola pausa al Thread che gestisce il selettore nel caso in cui si usi un *selectNow()*. Essendo composto di istruzioni non bloccanti, l'esecuzione di quel ciclo avviene senza interruzioni di sorta. Il Thread gira al massimo delle possibilità offerte dal sistema il che causa alla cpu il fiatone, a meno che non si tratti di un'archiettura multi-thread *a la* Sparc. Dieci millisecondi di pausa consentono alla cpu di andare in idle o di eseguire istruzioni di altri processi, secondo le politiche delle schedulatore di sistema.

Dopo la selezione c'è lo scorrimento con rimozione delle chiavi selezionate. Abbiamo già detto perchè si rimuova: per evitare che permanga nella selezione una chiave che un attimo prima poteva fare una certa operazioni ma non è detto che possa farlo ancora la volta successiva.

Se la chiave *isConnectable()*. Quando *isConnectable()*? La chiave *isConnectable()* quando il SocketChannel non bloccante è stato connesso, cioè ha subito l'invocazione *connect(SocketAddress)* e quell'invocazione abbia restituito false non producendo eccezioni. In questo caso, prima o poi, il canale segnalerà che o la connessione è pronta o si è verificata una qualche eccezione. In entrambi i casi, il canale diventerà connectable in quel momento il selettore selezioneraà la chiave associata, interessata ad un OP_CONNECT. La gestione di questa chiave deve terminare la procedura di connessione invocando *finishConnect()* sul canale associato. Nel caso in cui il canale abbia terminato il primo passo della connessione perchè qualcosa è andato storto e si è verificata un'eccezione, allora l'invocazione di *finishConnect()* restituirà quell'eccezione. Altrimenti l'invocazione restituisce true o false. Se il canale è bloccante, il metodo blocca il flusso e restituisce true quando la connessione sia stata completata o spara

una IOException se qualcosa vada storto in questa seconda fase. Se il canale è non bloccante, il metodo restituisce true, false o sputa un'eccezione. In ogni caso, l'invocazione di finishConnect fa passare il canale dallo stato *isConnectionPending() = true* a *isConnectionPending() = false* cioè l'operazione OP_CONNECT è da considerarsi terminata. Ecco perchè, a prescindere dal risultato, cambiamo l'interesse della chiave da OP_CONNECT alla combinazione di OP_READ più OP_WRITE.

Il segmento successivo nella gestione delle chiavi è relativo alle operazioni di scrittura. Passano di qui le chiavi candidata ad una OP_WRITE il cui canale sia pronto alla scrittura. Una curiosità. L'if che "scatena" la scrittura non è vincolato ad un "else" sulla connessione di cui abbia discusso appena sopra. Nel blocco che termina le procedure di connessione assegnamo alla chiave le operazioni OP_READ e OP_WRITE. C'è caso che dopo aver passato il controllo del primo if la stessa chiave passi anche il controllo di questo secondo if? Vale a dire, avendo noi impostato per la chiave "connettibile" le nuove operazioni OP_READ e OP_WRITE ed essendo questo secondo if dipendente dalla scrittura sul canale, da OP_WRITE, la chiave di prima è leggibile? La risposta è no. Le chiavi non sono selezionate in base alle operazioni che vogliono fare ma in base alle operazioni che il loro canale poteva fare al momento della selezione. La chiave key, dopo essere entrata nel primo if perchè connettibile, passa nelle mani del secondo if avendo come operazioni di interesse OP_READ e OP_WRITE ma come operazione di selezione OP_CONNECT: era l'unica operazione di interesse della chiave al momento della selezione. Insomma, è importante tener presente la differenza tra quello che una chiave vuole fare e quello che il selettore autorizza una chiave a fare, scelta fondata su ciò che la chiave voleva fare al momento della selezione e che permane tra una selezione e l'altra.

Se è possibile scrivere sul canale associato alla chiave, candidata alla scrittura, recuperiamo innanzitutto il messaggio che deve essere scritto sul canale. Il messaggio è stato allegato alla chiave di selezione quando abbiamo registrato il canale presso il selettore, all'inizio del codice. Recuperiamo poi il canale associato alla chiave che sappiamo essere un SocketChannel non bloccante. Dopodichè dobbiamo scrivere concretamente il messaggio. Quante invocazioni di un metodo write non bloccante devono essere fatte per poter spedire, diciamo 100 byte? Possono essere benissimo una, due, un miliardo: non lo sappiamo. Speriamo che sino meno di un milardo. Ogni volta che invochiamo write dobbiamo anche trovare il modo di stabilire quanti byte siano stati scritti così che la prossima volta quelli non li ripeteremo. Dobbiamo anche sapere se, per effetto di un numero imprecisato di write, siano stati inviati per intero i byte che volevamo. Tutto questo insieme di arzigogoli sta nell'invocazione *write(message)* ed è il naturale effetto dell'operazione di lettura dei dati da un ByteBuffer. Il metodo write legge i byte del ByteBuffer e li copia nel canale. Ogni volta che si legge un byte da un ByteBuffer il ByteBuffer restituisce il byte successivo all'ultimo letto e tiene traccia di questa successione attraverso un valore detto cursore. Un ByteBuffer è in grado di dirci quando il cursore abbia raggiunto il limite del buffer, cioè quando sia stato letto anche l'ultimo dei byte validi del buffer. Vista *a contrario* un ByteBuffer può dirci, tramite il metodo *hasRemaining()* se ci siano ancora dei byte che non sono stati letti. Poichè ogni write "legge" dal buffer, al massimo tanti byte quanti ce ne siano ancora da leggere, prima o poi la condizione *message.hasRemaining()* restituirà false. Restituirà false quando tutti i byte siano stati inviati lungo il canale per effetto di applicazioni successive del metodo write su message.

L'altra condizione che determina l'invio del dati è che il canale non bloccante abbia effettivante terminato il secondo stadio della procedura di connessione. L'abbiamo detto e vale la pena di ripeterlo. Per un SocketChannel non bloccante *finishConnect()* è un'educata richiesta: se e quando ti garba vedi un po' di concludere le operazioni di connessione. Il nostro *finishConnect()* che condiziona la scrittura verifica appunto quello: che al socket "abbia garbato" connettersi. Data la mancanza di qualsivoglia gestione delle eccezioni, il codice scarta in radice un'altra possibilità di cui invece dovrebbe preoccuparsi. Per mille ragioni il socket potrebbe fallire nel suo tentativo di terminare la procedura di connessione. Questo fatto sarebbe segnalato da una IOException e la gestione di questa eccezione dovrebbe occuparsi di rimuovere il socket dal selettore o di tentare una connessione ad un indirizzo diverso o di fare qualsiasi altra cosa tranne fregarsene. Io ho scelto la strada rapida e facile che finisce dritta giù per il burrone per semplicità espositiva.

Cosa succede all'atto dell'esecuzione. E creato un canale, c'è una prima connessione che solitamente non connette direttamente il socket, dopo qualche istante la chiave di selezione interessata alla

OP_CONNECT per il canale viene selezionata, il socket termina la connessione. In una delle selezioni successive, la stessa chiave, ora interessata ad OP_READ e OP_WRITE, viene selezionata per l'esecuzione delle operazioni OP_WRITE. I dati sono scritti lungo il canale, probabilmente in un colpo solo trattandosi di pochi byte. Comunque sia, ad un certo punto il buffer message non avrà più dati da restituire perchè il suo cursore avrà raggiunto il limite dei byte validi. Da notare che un canale "è sempre OP_WRITE" finchè resta aperto. Dunque i passaggi successivi del selettore selezioneranno la chiave associata. Nulla sarà scritto perchè *data.hasRemaining()* restituirà false. Alla domanda "ma allora potremmo far sì che una volta inviati tutti i dati la chiave perda interessa per l'operazione OP_WRITE" la risposta è "sì e per farlo basterebbe che, una volta ottenuto false come risposta a data.hasRemaining(), si invocasse *key.interesetOps(OP_READ)*", nel caso in cui si voglia tenere aperto il canale per leggere un'eventuale risposta, oppure potremmo chiudere il canale e chiudere il selettore per completare il programma.

La lettura non bloccante.

Qui ci occupiamo di come funzioni la lettura non bloccante di un messaggio. Abbiamo sempre per le mani un SocketChannel. Supponiamo di usare la nostra convezione: quando un socket riceve un messaggio i primi quattro byte che riceve sono i costituenti di un intero che rappresenta il numero ulteriore di byte che devono essere letti affinchè si possa dire di avere un messaggio completo. La prima operazione da fare è accumulare quattro byte. Con questi quattro byte creiamo un buffer che ospiterà il messaggio e lo riempiamo con le successive invocazioni read. Quando questo buffer sarà pieno, il messaggio potrà dirsi completo e qualcuno potrà manipolarlo.

Sfruttiamo ancora un allegato ma un po' più articolato del ByteBuffer che abbiamo usato per l'invio.

Partiamo dal risultato. Quello che vogliamo è che l'allegato possa accumulare i byte letti tramite un read non bloccante, gestendo autonomamente la porzione di intestazione, cioè i quattro byte che indicano la lunghezza del messaggio, e la parte del contenuto. In più dobbiamo sapere quando il messaggio sia stato completamente letto. Serve dunque un metodo che potrebbe chiamarsi:

```
public void accumulate(ByteBuffer data) { ... }
```

il cui scopo sia accumulare i dati letti dal SocketChannel, intepretando la faccenda dei primi quattro byte come intero. Poi un metodo:

```
public boolean messageComplete() { ... }
```

che invocheremo dopo accumulate per sapere se il messaggio possa dirsi completato. Infine, un metodo:

```
public String decodeMessage() { ... }
```

che useremo dopo che messageComplete abbia restituito true, per ottenere il testo comunicato.

Il codice che segue è un esempio, debitamente commentato, di un oggetto che abbia le proprietà su esemplificate e possa quindi essere usato come buffer per un messaggio in transizione nel modo da noi previsto.

```
import java.nio.*;
import java.nio.charset.*;

/** Allegato in grado di ricostruire un messaggio accumulando
i dati ottenuti da una successione di read non bloccanti. I primi
quattro byte sono interpretati come un intero che determina il numero
di byte che devono essere successivamente letti prima che il
messaggio possa dirsi completo */
public class BufferedMessage {
```

```
/* Buffer che contiene i primi quattro byte della comunicazione,
i quali indicano la lunghezza del messaggio codificato */
private ByteBuffer sizeBuffer = ByteBuffer.allocate(4);

/* Interruttore che controlla se la lunghezza del messaggio
sia stata già letta (true) o no (false) */
private boolean sizeRead = false;

/* Buffer che contiene i byte del testo, codificato */
private ByteBuffer messageBuffer;

/** Ricostruisce il messaggio accumulando i byte in
argomento */
public void accumulate(ByteBuffer data) {

  /* Se la dimensione del messaggio è ancora ignota
  e finchè ci sono byte da leggere in data (che
  potrebbe anche averne meno di 4, per quanto ne
  sappiamo... */
  while(!sizeRead && data.hasRemaining()) {

    /* trasferisce un byte da data a sizeBuffer */
    sizeBuffer.put(data.get());

    /* se sizeBuffer è pieno, cioè i famosi
    quattro byte sono stati letti... */
    if(sizeBuffer.hasRemaining() == false) {

      /* riavvolge il buffer per poterlo rileggere */
      sizeBuffer.rewind();

      /* la lunghezza del messaggio è l'intero che
      si ottiene dai quattro byte di sizeBuffer */
      int  messageSize = sizeBuffer.getInt();

      /* e quella lunghezza è usata per stabilire
      le dimensioni del buffer che dovrà contenere i byte
      del testo. Quanto messageBuffer sarà pieno allora
      il testo sarà stato letto */
      messageBuffer = ByteBuffer.allocate(messageSize);

      /* Segnala che la dimensione è stata già letta così
      che i passaggi precedenti non siano ripetuti */
      sizeRead = true;
    }
  }

  /* C'è ancora qualcosa in data? Se sì allora messageBuffer,
  per le istruzioni precedenti, è logicamente diverso da null */
  if(data.hasRemaining()) {
    messageBuffer.put(data);
  }
}

/** Azzera lo stato di questo BufferedMessage. Da qui in poi
accumulate opera come se nulla fosse mai stato letto. */
public void reset() {

  /* Scarta il buffer del testo */
  messageBuffer = null;

  /* Riavvolge il buffer della lunghezza */
  sizeBuffer.rewind();

  /* Cambia lo stato dell'interruttore che controlla
  se i byte in arrivo devano essere usati per ricostruire
  la lunghezza del messaggio */
  sizeRead = false;
}
```

```
  /** True se questo BufferedMessage contenga tutti i dati necessari
  a restituire il messaggio accumulato */
  public boolean messageComplete() {

    /* Sì se il buffer del messaggio non ha più spazio per altri byte.
    Poichè il metodo può essere invocato in ogni momento controlliamo
    anche che un buffer del messaggio esista */
    return messageBuffer != null && messageBuffer.hasRemaining() == false;
  }

  /** Trasforma il contenuto del buffer del messaggio in una stringa.
  Azzera lo stato di questo BufferedMessage invocando reset(). */
  public String decodeMessageAndReset() {

    /* Imposta la posizione corrente del cursore nel buffer del messaggio
    come limite di lettura e riavvolge. Usa flip e non rewind perchè nulla
    vieta che qualcuno possa cercare di recuperare un messaggio parzialmente
    ricevuto (magari perchè la connessione è andata a ramengo prima che
    messageComplete() restituisse true) */
    messageBuffer.flip();

    /* Converte il contenuto disponibile in messageBuffer in una stringa
    usando UTF-8 come formato di testo */
    String message = Charset.forName("utf-8").decode(messageBuffer).toString();

    /* Azzera lo stato di questo BufferedMessage, rendendolo disponibile
    per la lettura di un altro messaggio */
    reset();

    /* Restituisce il testo del messaggio */
    return message;
  }
}
```

Con BufferedMessage proviamo a leggere lato server. Dobbiamo attivare un ServerSocketChannel con un Selector, metterlo in ascolto per connessioni su una certa porta e, quando è rilevata una connessione registriamo il canale del client appena entrato in contatto allegando alla sua chiave un BufferedMessage. Quando il selettore rileva che il canale del client sta ricevendo, leggiamo i dati ricevuti in un buffer temporaneo di dimensione fissa, diciamo 64 byte, e li travasiamo nell'allegato. Infine, quando l'allegato ci dirà che è "pieno", stamperemo il messaggio.

Mantenedo la gestione assassina delle eccezioni, vediamo passo dopo passo come fare. Apriamo un Selector, che servirà per gestire il ServerSocketChannel e il SocketChannel client, quando sarà connesso.

```
Selector selector = Selector.open();
```

Creiamo un ServerSocketChannel, che riceverà le connessione, lo configuriamo in modalità non bloccante, e colleghiamo il server socket che gli sta dietro ad una porta.

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
serverChannel.socket().bind(new InetSocketAddress("localhost", 9090));
```

Registriamo il ServerSocketChannel presso il selettore, dichiarando il suo interesse per l'accettazione di nuove connessioni.

```
serverChannel.register(selector, OP_ACCEPT, null);
```

A questo punto siamo pronti a far girare la ruota. Finchè il selettore è aperto, gli facciamo selezionare le chiavi "papabili" ed esaminiamo, una per volta, quelle chiavi, sempre ricordandoci si eliminare la chiave in esame.

```
while(selector.isOpen()) {
  selector.selectNow();
  Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
  while(keys.hasNext()) {
    SelectionKey key = keys.next();
    keys.remove();
```

L'ordine non conta ma iniziamo con la chiave "isAcceptable() == true", cioè la chiave del ServerSocketChannel, registrata per la selezione quando, appunto, il ServerSocketChannel stia ricevendo una richiesta di connessione.

```
  if(key.isAcceptable()) {
```

Chiave accettabile = il nostro ServerSocketChannel ha qualcuno che vuole connettersi. Abbiamo già visto gli elementi dell'accettazione. Si prende il ServerSocketChannel associato alla chiave e gli si fa accettare il client in connessione. Poichè accept è non bloccante, il metodo non inchioda tutto in attesa che qualcuno si connetta ma restituisce subito il controllo e un bel null nel caso in cui non ci siano connessioni in attesa. Noi sappiamo tuttavia che una connessione c'è: è stato il selettore a scegliere la chiave proprio perchè ha rilevato una connessione in coda. Dunque se la chiave "è accettabile"...

```
    if(key.isAcceptable()) {
```

...prendiamo il ServerSocketChannel associato...

```
      ServerSocketChannel server = (ServerSocketChannel)key.channel();
```

...e da questo otteniamo il client in connessione.

```
      SocketChannel client = server.accept();
```

Il SocketChannel connesso è bloccante per definizione. Noi lo passiamo al lato oscuro della comunicazione.

```
      client.configureBlocking(false);
```

A questo punto lo registriamo presso il selettore. Dichiariamo il suo interesse alle operazioni di lettura e scrittura. OP_READ più OP_WRITE. In aggiunta, alleghiamo alla sua chiave di selezione un BufferedMessage. Questo sarà lo stesso BufferedMessage che ci troveremo per le mani quando la chiave del canale client sarà selezionata al momento della lettura e che noi useremo per travasare i dati in arrivo. Questa operazione termina anche la procedura richiesta per stabilire una connessione tra il lato server dell'applicazione e un'indefinita moltitudine di client.

```
      client.register(selector, OP_READ | OP_WRITE, new BufferedMessage());
    }
```

Questo era il segmento dedicato alla connessione. Ora passiamo alla lettura. Quando sarà pronto per farlo, quel client che abbiamo appena registrato o, meglio, i client che abbiamo appena registrato diventeranno "leggibili", cioè qualche byte starà passando per il canale, in direzione del server. Il selettore sceglierà la sua chiave e la renderà "leggibile". Noi controlliamo se la chiave sia leggibile per stabilire se ci siano dei byte da trasferire.

```
    if(key.isReadable()) {
```

Per trasferire prendiamo il canale associato e il suo allegato, che abbiamo stabilito essere un BufferedMessage. Creiamo un buffer di 64 byte per trasferire i dati. Questo buffer potrebbe essere preinizializzato, fuori dal ciclo del selettore. Potrebbe essere diretto e sarebbe anche facilmente tra letture diverse: tutto il meccanismo usa un solo Thread, non ci sono problemi di concorrenza nell'accesso al buffer. Noi ne creiamo uno ad ogni passaggio per maggiore evidenza.

```
    SocketChannel in = (SocketChannel)key.channel();
    BufferedMessage message = (BufferedMessage)key.attachment();
    ByteBuffer data = ByteBuffer.allocate(64);
```

Una lettura non bloccante da client riempie buffer di qualche byte, da zero a 64. E qui un'eccezione la gestiamo, tanto per abituarci gradualmente. Gestiamo l'eccezione prodotta dal tentativo di leggere da un socket disconnesso.

```
    try {
      in.read(data);
    } catch(IOException ex) {
      key.cancel();
    }
```

Il tentativo di leggere da un SocketChannel disconnesso spara una IOException. La gestione propone al selettore la cancellazione della chiave di selezione del socket: il client è sparito dall'orizzonte, la sua chiave deve essere rimossa affinchè il selettore non sia appesantito da chiavi che non hanno più ragion d'essere. La rimozione effettiva avverà alla prossima invocazione del metodo *select()* e analoghi per il selettore ma la chiave non è più valida dal momento in cui il metodo *cancel()* restituisce il controllo.

Trasferiamo i byte letti a condizione che la chiave sia ancora valida. La chiave perde validità come effetto primo e diretto dell'invocazione del metodo *cancel()*. Per noi è il segnale che non è più necessario continuare a leggere. Il destinatario del trasferimento è il BufferedMessage allegato.

```
    if(key.isValid()) {
      data.flip();
      buffer.accumulate(data);
```

Dopo aver accumulato quei byte, il messaggio è concluso? Lo chiediamo al nostro BufferedMessage. Se il messaggio è concluso, lo stampiamo a video.

```
      if(buffer.messageComplete()) {
        System.out.println(buffer.decodeMessageAndReset());
      }
```

Decodificando il messaggio, come sottolineato dal nome del metodo, azzeriamo il buffer, in modo che sia pronto a catturare un altra comunicazione dallo stesso client.

Quello che segue è il codice completo del server.

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;
import static java.nio.channels.SelectionKey.*;

public class Server {
  public static void main(String[] args) throws Throwable {
    new Server().start();
  }

  public void start() throws Exception {

    /* Apre un selettore che gestirà i vari canali */
    Selector selector = Selector.open();

    /* Crea un indirizzo a cui collegare il server */
    InetSocketAddress serverAddress = new InetSocketAddress("localhost", 9090);

    /* Crea un ServerSocketChannel, usato per
    accettare le connessioni dei client */
```

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();

/* Configura il socket in modalità non bloccante.
L'operazione non bloccante che ci interessa è
l'accettazione delle connessioni. */
serverChannel.configureBlocking(false);

/* Colleghiamo il server alla porta di ascolto */
serverChannel.socket().bind(serverAddress);

/* Registriamo il server presso il selettore dichiarando
un interesse per l'accettazione di nuove connessioni,
senza allegato */
serverChannel.register(selector, OP_ACCEPT, null);

/* Iniziamo il ciclo di selezione, a selettore aperto */
while(selector.isOpen()) {

  /* Chiediamo al selettore di selezionare le chiavi
  i cui canali possono compiere qualcuna delle operazioni
  a cui dichiarano di essere interessati... */
  selector.selectNow();

  /* e otteniamo un iteratore su quelle chiavi */
  Iterator<SelectionKey> keys = selector.selectedKeys().iterator();

  /* Per ogni chiave... */
  while(keys.hasNext()) {

    /* Prendiamo la chiave corrente... */
    SelectionKey key = keys.next();

    /* ...e la rimuoviamo dall'insieme di
    selezione. */
    keys.remove();

    /* La chiave è accettabile? Vale a dire, è il
    ServerSocketChannel che ci sta dicendo "hey, qualcuno
    bussa alla porta!" */
    if(key.isAcceptable()) {

      /* Prendiamo il ServerSocketChannel associato alla chiave */
      ServerSocketChannel server = (ServerSocketChannel)key.channel();

      /* E otteniamo il client che vuole connettersi */
      SocketChannel client = server.accept();

      /* Lo impostiamo come non bloccante */
      client.configureBlocking(false);

      /* E registriamo quel client come interessato
      alla lettura e alla scrittura. Nella sua chiave
      di registrazione mettiamo un nuovo BufferedMessage,
      che useremo per ottenere un messaggio completo dall'accumulo
      dei byte ottenuti da diverse letture non bloccanti */
      client.register(selector,  OP_READ | OP_WRITE, new BufferedMessage());

      /* Sulla console stampiano un messaggio che avvisa
      dell'avvenuta connessione */
      System.out.println("Client connected and registered...");
    }

    /* La chiave è leggibile, cioè lungo il canale associato
    alla chiave stanno arrivando dei dati? */
    if(key.isReadable()) {

      /* Prendiamo l'allegato-accumulatore dalla chiave */
      BufferedMessage buffer = (BufferedMessage)key.attachment();
```

```
            /* Prendiamo il canale associato alla chiave */
            SocketChannel in = (SocketChannel)key.channel();

            /* Creiamo un buffer per la lettura non bloccante
            dal canale */
            ByteBuffer data = ByteBuffer.allocate(64);

            /* Tentiamo di trasferire un po' di byte dal canale
            al buffer data */
            try {
              in.read(data);
            } catch(IOException ex) {

              /* Se il canale è chiuso arriviamo qui e a quel punto
              significa che il canale non è più leggibile. Per noi
              è come se fosse morto e sepolto. Per tutta risposta
              cancelliamo la sua chiave dal selettore. La cancellazione
              invalida la chiave che sarà effettivamente rimossa solo
              alla prossima invocazione di select() o selectNow() o
              select(long) sul selettore */
              key.cancel();
            }

            /* A meno che la chiave non sia stata invalidata, perchè
            l'abbiamo detto noi quando il canale è diventato illegibile */
            if(key.isValid()) {

              /* Impostiamo come limite di lettura il punto in cui si
              trova l'ultimo byte trasferito nel buffer data (write
              può passare da 0 a 64 byte a data) e riavvolgiamo il buffer
              per poter leggere quello che c'è stato scritto */
              data.flip();

              /* Accumuliamo il contenuto di data nel BufferedMessage */
              buffer.accumulate(data);

              /* Dopo l'ultima infornata di byte, il messaggio è stato
              completato? */
              if(buffer.messageComplete()) {

                /* Sì, benissimo, stampa il contenuto decodificato e
                appronta il buffer per un'altra eventuale lettura */
                System.out.println(buffer.decodeMessageAndReset());
              }
            } else { //Qui ci arriviamo se "se key.isValid() == false

              /* Stampa un messaggio che informa dell'avvenuta
              morte del client */
              System.out.println("Connessione terminata");
            }
          }
        }

        /* Povera bestia di una cpu, tira un attimo il fiato... */
        Thread.sleep(10);
      }
    }
}
```

Chat

Concludiamo il nostro breve viaggio incollando le varie parti a formare un macchinario funzionante. L'applicazione che vediamo scambia messaggi di testo tra più client: una lontana cugina di una chat.

Il codice sorgente è articolato in nove classi.

BidiBuffer.java

ChatClient.java

ChatLogger.java

ChatServer.java

ChatUtils.java

KeyProcessor.java

Message.java

MessageDispatcher.java

MessageManager.java

Si avvia ChatServer, poi si lanciano uno più ChatClient. Ciò che è scritto nella console del client arriva al server, il server propaga i messaggi a tutti i client connessi al momento della ricezione, incluso chi abbia inviato il messaggio. Esaminiamo i sorgenti uno alla volta, partendo dai meno significativi.

ChatLogger.java

ChatLogger è un'interfaccia che dichiara un metodo. Il suo scopo è separare idealmente la comunicazione dei messaggi di funzionamento dell'applicazione, lato client e lato server, rispetto alla presentazione dei messaggi derivanti dalla comunicazione via chat. Idealmente perchè nell'applicazione reale tutto andrà a finire nel calderone della console dei comandi.

```
/** Metodi di un logger per i componenti
dell'applicazione chat */
public interface ChatLogger {

  /** Preme un messaggio nella coda del logger.
  @param message il testo del messaggio
  @param ex un'eventuale eccezione allegata al messaggio */
  void pushLog(String message, Exception ex);
}
```

L'interfaccia è concretizzata da ChatServer e ChatClient ed è usata da ChatServer, ChatClient e KeyProcessor. Tanto ChatServer quanto ChatClient danno la stessa definizione di pushLog, che riporto qui per comodità.

```
/** @inheritDoc */
public void pushLog(String message, Exception ex) {
  System.out.printf("Date: %s%nMessage: %s%nException: %s%n%n",
    new Date(), message, ex == null ? "none" : ex);
}
```

ChatUtils.java

ChatUtils è una biechissima collezione di funzioni il cui scopo principale è rendere meno affastellato di parentesi il codice sorgente delle altre classi.

```
import java.nio.ByteBuffer;
import java.nio.charset.Charset;
```

```
/** Insieme di funzioni utili */
public class ChatUtils {

  /** Restituisce come stringa il contenuto del messaggio
  in argomento. L'estrazione si fonda sulla convezione
  secondo cui i primi quattro byte del buffer del messaggio
  rappresentano la lunghezza del testo UTF-8 seguente */
  public static String decode(Message m) {
    ByteBuffer data = m.getBuffer().duplicate();
    data.position(4); //scarta i primi 4 byte
    return Charset.forName("UTF-8").decode(data).toString();
  }

  /** Restituisce e rimuove l'elemento successivo dell'iteratore
  in argomento */
  public static <T> T getAndRemove(java.util.Iterator<T> iterator) {
    T value = iterator.next();
    iterator.remove();
    return value;
  }

  /** Invoca il metodo close sull'oggetto.
  Restituisce sempre null */
  public static <T> T close(Object o) {
    if(o != null) {
      try {
        o.getClass().getMethod("close").invoke(o);
      } catch(Exception ex) {
        System.err.println("ChatUtils, close threw an unexpected exception!");
        ex.printStackTrace();
      }
    }
    return null;
  }

  /** Mette in pausa il Thread currente per millis
  millisecondi. Restituisce true se durante la pausa
  si sia verificata un'interruzione. Restituisce false
  se durante la pausa non si sia verificata
  un'interruzione */
  public static boolean pause(final long millis) {
    try {
      Thread.sleep(millis);
    } catch(InterruptedException ex) {
      return true;
    }
    return false;
  }

  /** Invoca il metodo selectNow del selettore in argomento.
  Restituisce l'eccezione prodotta dall'invocazione o null
  se non si sia verificata alcuna eccezione */
  public static Exception selectNow(java.nio.channels.Selector selector) {
    try {
      selector.selectNow();
    } catch(java.io.IOException ex) {
      return ex;
    }
    return null;
  }
}
```

I metodi di ChatUtils sono invocati in ChatClient, ChatServer e KeyProcessor.

Message.java

La classe Message.java è un contenitore per i byte del testo di un messaggio.

```
import java.nio.*;
import java.nio.charset.*;

/** Involucro del testo di un messaggio. */
public class Message {
  private ByteBuffer data;

  /** Crea un messaggio usando i byte del buffer in
  argomento come rappresentazione del testo. Si suppone
  che il testo codificato sia in formato UTF-8. Il buffer
  deve essere pronto per la lettura */
  public Message(ByteBuffer textData) {
    data = ByteBuffer.allocate(textData.limit() + 4);
    data.putInt(textData.limit());
    data.put(textData);
    data.rewind();
  }

  /** Crea un messaggio codificando la stringa in
  argomento in formato UTF-8 */
  public Message(String text) {
    ByteBuffer encoded = Charset.forName("utf-8").encode(text);
    data = ByteBuffer.allocate(encoded.limit() + 4);
    data.putInt(encoded.limit());
    data.put(encoded);
    data.rewind();
  }

  /** Restituisce un duplicato del buffer che contiene
  il messaggio. I primi quattro byte del buffer sono quelli
  di un int Java e rappresentano il numero di byte oltre il
  quarto che devono essere letti per avere tutti i byte
  del testo, in formato UTF-8 */
  public ByteBuffer getBuffer() {
    return data.duplicate();
  }
}
```

L'involucro Message incapsula quella convenzione di cui abbiamo già parlato, usata anche in quest'applicazione, secondo cui i primi quattro byte di inviati o ricevuti sono i costituenti di un int Java che rappresenta la "lunghezza" del testo che sta arrivando, in formato UTF-8. Message può incapsulare una stringa di testo o un buffer di byte. La prima capacità è utile al client, che deve trasformare la stringa digitata sulla console in un buffer prima di inviarlo. La seconda capacità è utile al server che non ha la capacità/necessità di comprendere il testo inviato ma deve semplicemente smistarlo tra i vari client una volta che l'abbia ricevuto.

BidiBuffer.java

BidiBuffer è la somma di due cose già viste. Il buffer che abbiamo usato per spedire un messaggio tramite una sequenza di scritture non bloccanti e il buffer che abbiamo usato per leggere un messaggio, i cui primi quattro byte eccetera eccetera, tramite una sequenza di letture non bloccanti.

```
import java.io.IOException;
import java.nio.channels.WritableByteChannel;
import java.nio.ByteBuffer;

/** Allegato alle chiavi dei canali. Tiene traccia
dei byte inviati e ricevuti tra una lettura/scrittura
```

```
non bloccante e l'altra */
public class BidiBuffer {
  /* Buffer della dimensione del testo in lettura */
  private ByteBuffer inSize = ByteBuffer.allocate(4);

  /* Buffer dei byte del testo in lettura */
  private ByteBuffer inBuffer;

  /* Interruttore che segnale se la dimensione sia
  stata già letta (true) o non sia ancora stata letta
  (false) */
  private boolean inSizeRead;

  /* Buffer del messaggio in scrittura */
  private ByteBuffer outBuffer;

  /* ID del Client a cui appartiene questo allegato */
  private long id;

  /** Crea un BidiBuffer appartenete al client con
  l'id in argomento */
  public BidiBuffer(long id) {
    this.id = id;
  }

  /** Restituisce l'id del client a cui appartiene
  questo BidiBuffer */
  public long getId() {
    return id;
  }

  /** true se il buffer del messaggio in uscita
  non sia vuoto */
  public boolean hasOut() {
    return outBuffer != null;
  }

  /** Usa il buffer in argomento come buffer del
  messaggio in uscita */
  public void setOut(ByteBuffer outBuffer) {
    this.outBuffer = outBuffer;
  }

  /** Scrive i byte del messaggio in uscita sul canale out.
  Se la scrittura svuota il buffer, dopo l'invocazione di
  writeOut hasOut restituirà false */
  public void writeOut(WritableByteChannel out) throws IOException {
    out.write(outBuffer);
    if(outBuffer.hasRemaining() == false) {
      outBuffer = null;
    }
  }

  /** Accumula nel buffer del messaggio in arrivo i byte
  del buffer in argomento */
  public void accumulateIn(ByteBuffer data) {
    while(!inSizeRead && data.hasRemaining()) {
      inSize.put(data.get());
      if(inSize.hasRemaining() == false) {
        inSize.rewind();
        inBuffer = ByteBuffer.allocate(inSize.getInt());
        inSizeRead = true;
      }
    }
    if(data.hasRemaining()) {
      inBuffer.put(data);
    }
  }
```

```
  /** Restituisce il buffer del messaggio in arrivo o
  null se la lettura del messaggio non sia stata completata */
  public ByteBuffer getInData() {
    return
      inBuffer != null && inBuffer.hasRemaining() == false ?
      inBuffer :
      null;
  }

  /** Azzera lo stato della lettura del messaggio in arrivo.
  L'azzeramento rende questo BidiBuffer disponibile alla
  lettura di un nuovo messaggio */
  public void resetIn() {
    inBuffer = null;
    inSize.rewind();
    inSizeRead = false;
  }
}
```

BidiBuffer è usato da ChatClient e da KeyProcessor. Ogni SocketChannel che partecipa al gioco ha come allegato alla propria chiave di registrazione un BidiBuffer. Quando il canale è leggibile è usata la parte di BidiBuffer dedicata alla lettura. Quando il canale è scrivibile, è usata la parte di BidiBuffer dedicata alla scrittura.

MessageDispatcher.java e MessageManager.java

Un ambo (sulla ruota della tristezza). Il problema che risolvono è: come fa un Server a smistare i messaggi, che arrivano da tutte le parti come cazzotti a un pugile suonato in partenza, tenendo conto che a dargliele c'è chi va e chi viene? Così. Qualcuno, MessageManager, sa quali e quanti siano i client connessi in un certo istante. Quando arriva un messaggio, il server lo passa a MessageManager. MessageManager prende il messaggio, prende l'elenco di client connessi e li accoppia in un involucro, MessageDispatcher. Quando un client è pronto a ricevere un messaggio, il server prende l'id del client e lo passa a MessageManager. MessageManager verifica quale sia il primo messaggio in cosa che non sia stato ancora dato a quel client. Per saperlo, scorre all'indietro una coda di MessageDispatcher. Ad ogni MessageDispatcher chiede: hai già dato il tuo messaggio a ID? MessageDispatcher controlla se ci sia già una bella X sul nome di quel client nell'elenco dei client con cui è stato costruito. Se c'è dice che non ha messaggi per quel client. Se non c'è mette la croce e gli passa il Message che contiene. Quando un MessageDispatcher risulta vuoto, perchè tutti quelli che potevano chiedere il messaggio l'hanno chiesto, il MessageDispatcher diventa "vuoto". Quando MessageManager rileva che un MessageDispatcher è vuoto, lo elimina dalla sua coda. Quando il server rileva che un client connesso è andato per margherite, chiede a MessageManager di eliminare quel client. MessageManager lo elimina e poi segnala a tutti i MessageDispatcher di considerare quel client come se avesse già ricevuto il messaggio. MessageManager controlla se questo faccia diventare vuoto il MessageDispatcher e, in caso affermativo, lo elimina.

Insieme MessageDispatcher e MessageManager sono molto più brevi della lunga spiegazione che ne ho dato. Quello che segue è il codice sorgente di MessageDispatcher.

```
import java.util.*;

/** Contiene un Message e un elenco di client, per id.
A richiesta, MessageDispatcher  restituisce il suo messaggio
e segnala al suo interno che un certo client, tra quelli
compresi nella sua lista, ha già richiesto il messaggio. Sulla
base di questa traccia può rispondere ad una richiesta di un
certo client dicendo "te l'ho già dato".*/
public class MessageDispatcher {

  /* Elenco degli id dei client a cui questo MessageDispatcher
  deve fornire il suo messaggio */
  private LinkedList<Long> clients;
```

```
  /* Messaggio fornito da questo MessageDispatcher */
  private Message message;

  /** Crea un MessageDispatcher che può fornire il messaggio in argomento
  ai client i cui id sono contenuti nell'insieme clientIDs */
  public MessageDispatcher(Message message, Collection<Long> clientIDs) {
    clients = new LinkedList<Long>(clientIDs);
    this.message = message;
  }

  /** true se tutti i client a cui questo MessageDispatcher
  può fornire il suo messaggio lo abbiano già richiesto */
  public boolean isEmpty() {
    return clients.isEmpty();
  }

  /** Restituisce il messaggio per il client di id in
  argomento o null se risulti che quel client lo
  abbia già richiesto */
  public Message requestMessage(long clientId) {
    int index = clients.indexOf(clientId);
    if(index < 0) {
      return null;
    } else {
      /*Per sapere chi ha ricevuto e chi no, questo
      pazzo semplicemente cancella l'id del client
      dalla sua lista. Quando è richiesto il messaggio
      se l'id non sia presente nella lista, il MessageDispatcher
      considera il client come "già fornito" */
      clients.remove(index);
      return message;
    }
  }
}
```

MessageManager non è molto più lungo. Eccolo.

```
import java.util.*;

public class MessageManager {

  /* intero che rappresenta l'id assegnato all'ultimo client */
  private Long lastClientId = 0L;

  /* elenco dei client attivi, per id */
  private ArrayList<Long> clients = new ArrayList<Long>();

  /* "coda" dei messaggi da propagare */
  private LinkedList<MessageDispatcher> messages =
    new LinkedList<MessageDispatcher>();

  /** Restituisce un id diverso da quello di ogni
  altro client finora attivato e registra quell'id
  come client attivo. L'id restituito è lo stesso ottenuto
  dall'invocaziond immediatamente precedente di getNewClientId()*/
  public synchronized long createNewClient() {
    lastClientId++;
    clients.add(lastClientId);
    return lastClientId;
  }

  /** Restituisce un id diverso
  da quello di ogni altro client finora attivato. L'id
  restituito è lo stesso restituito dall'invocazione
  immediatamente successiva di createNewClient() */
  public synchronized long getNewClientId() {
    return lastClientId + 1;
  }
```

```
  /** Elimina il client dal registro dei client attivi.
  Aggiorna lo stato dei messaggi da propagare. */
  public synchronized void deleteClient(long id) {
    clients.remove(id);
    Iterator<MessageDispatcher> iterator = messages.iterator();
    while(iterator.hasNext()) {
      MessageDispatcher m = iterator.next();
      m.requestMessage(id);
      if(m.isEmpty()) {
        iterator.remove();
      }
    }
  }

  /** Restituisce il primo messaggio in coda per il client
  di id in argomento o null se non ci siano messaggi in coda
  per quel client */
  public synchronized Message getNextMessage(long clientId) {
    Message message = null;
    /* Scorre i messaggi dall'ultimo al primo (i più vecchi stanno in fondo
    per via dell'offer usato in enqueueMessage)*/
    for(int i = messages.size() - 1; message == null && i >= 0; i--) {
      MessageDispatcher disp = messages.get(i);
      message = disp.requestMessage(clientId);

      /*Se è vuoto, coglie al volo l'occasione per eliminarlo */
      if(disp.isEmpty()) {
        messages.remove(i);
      }
    }
    return message;
  }

  /** Aggiunge il messaggio in argomento alla coda di propagazione.
  Il messaggio è registrato come destinato a tutti i client
  attivi al momento dell'invocazione */
  public synchronized void enqueueMessage(Message m) {
    MessageDispatcher disp = new MessageDispatcher(m, clients);
    messages.offer(disp);
  }
}
```

Ora abbiamo modo di sapere a chi devano essere spediti i messaggi che il server riceve. Abbiamo anche lo strumento per spedirli e riceverli con una serie di scritture e letture non bloccanti. Insomma, siamo a cavallo.

KeyProcessor.java

La classe KeyProcessor definisce il comportamento di un oggetto che gestisce le connessioni e la lettura e scrittura di Message lungo i canali attraverso dei BidiBuffer. Va bene sia per il server che per i client. C'è il truccone sotto. KeyProcessor usa un MessageManager come fonte e destinazione dei Message. Per il server, usa un MessageManager normale, mantenuto internamente al server. Il client, che produce i messagge attraverso la tastiera e li proietta sulla tastiera, rifila al suo KeyProcessor un MessageManager che anzichè accodare i messaggi che letti da KeyProcessor li spara sulla console e quando rileva del testo da spedire lo accoda come messaggio. Il tutto senza che KeyProcessor abbia modo di accorgersene. Ciò che cambia, in modo insensibile al KeyProcessor, è che quando quest'ultimo è usato da un Server opera su una moltitudine di canali mentre quando è usato da un Client opera su un solo canale, quello del client. Che è come aver detto tutto e niente.

La classe è lunga. Sono un po' più di 120 linee di codice. Commentando l'incommentabile sono arrivato a 200 e passa. Essendo il sistema circolatorio della chat, sia lato client che lato server, ho ritenuto

importante non tralasciare i dettagli. E' idealmente divista in due parti. Dapprima c'è un generico metodo processKey che gestisce una chiave di selezione. Sarà poi il motore che fa ruotare il selettore a passare ad un KeyProcessor una chiave attiva alla volta. Il metodo processKey smista quella chiave ad altri quattro metodi, uno per l'accettazione di nuove connessioni, uno che termina il procedimento necessario alla connessione di un client, uno per la lettura ed uno per la scrittura. Tutte cose già viste, solo che qui son tutte insieme. Ecco il codice.

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import static java.nio.channels.SelectionKey.*;

public class KeyProcessor {

  /* Buffer condiviso usato per la lettura dei messaggi dai canali.
  L'intensità d'uso suggerisce la possibilità che il maggior onere
  di creazione e distruzione sia positivamente compensato dalla
  maggiore rapidità di trasferimento dei dati*/
  private final ByteBuffer READ_BUFFER = ByteBuffer.allocateDirect(512);

  /* Coda dei "messaggi che sanno a chi devono andare" */
  private final MessageManager MESSAGE_MANAGER;

  /* Destinazione dei messaggi di funzionamento */
  private final ChatLogger LOGGER;

  /** Crea un KeyProcessor. Il logger e il gestore dei messaggi
  non devono essere null*/
  public KeyProcessor(ChatLogger logger, MessageManager manager) {

    /* Gestisco questi due null come eccezione IllegalArgumentException
    perchè altrimenti, se i due parametri fosser null "liberi" allora
    sarebbe generata un'eccezione NullPointerException in una parte
    del codice che potrebbe rendere difficile raccapezzarsi circa
    il perchè le cose siano andate a schifìo. */
    if(logger == null) {
      throw new IllegalArgumentException(
        "Chat Logger cannot be null in KeyProcessor constructor");
    }
    if(manager == null) {
      throw new IllegalArgumentException(
        "Message manager cannot be null in KeyProcessor constructor");
    }
    LOGGER = logger;
    MESSAGE_MANAGER = manager;
  }

  /** Questo metodo gestice le operazioni di accettazione,
  connessione, lettura e scrittura del canale associato a key
  secondo l'operatività di quella chiave. Restituisce true
  se tutto vada bene. Restituisce false se è rilevata
  un'eccezione che questo KeyProcessor considera
  "invalidate il canale", cioè capita qualcosa, tipo
  una illeggibilità dovuta a chiusura, che il server
  dovrebbe gestire rimuovendo la chiave key. Lascia
  tuttavia questa decisione al motore che fa girare
  il selettore, altrove definito */
  public boolean processKey(SelectionKey key) {

    /* Questo valore può cambiare in questo metodo
    ed è il segnale restituito in fine */
    boolean cancel = false;

    /* Questo try sta qui perchè se una chiave diventa
    invalida allora è prodotta una eccezione Runtime. Questa
```

```
    eccezione può saltar fuori ogni volta che si tenta un'invocazione
    key.isQualcosa. Raggruppo la gestione in un solo try-catch
    perchè una volta che una chiave è invalida per un "is" è
    invalida per tutti gli altri "is". */
    try {

      /* Qualche SocketChannel deve terminare le procedure di
      connessione? (tipicamente sarà il lato Client dell'applicazione
      a infilare nel Selettore un SocketChannel OP_CONNECT) */
      if(!cancel && key.isConnectable()) {
        cancel = connect(key);
      }

      /* Qualcuno vuole connettersi al ServerSocketChannel
      (tipicamente sarà il lato Server ad avere un ServerSocketChannel) */
      if(!cancel && key.isAcceptable()) {
        cancel = accept(key);
      }

      /* Il canale sottostante è pronto a leggere? */
      if(!cancel && key.isReadable()) {
        cancel = read(key);
      }

      /* Il canale sottostante è pronto a scrivere? */
      if(!cancel && key.isWritable()) {
        cancel = write(key);
      }
    } catch(CancelledKeyException ex) {
      /* L'eccezione è gestita come uno di quei casi che
      fa ritenere al KeyProcessor che la chiave debba sparire */
      cancel = true;
    }

    return cancel;
  }

  /* Chiave "connettibile": gestiamo il secondo passo della
  procedura di connessione di un SocketChannel ad un indirizzo. */
  private boolean connect(SelectionKey key) {
    SocketChannel client = (SocketChannel)key.channel();
    try {
      client.finishConnect();
      key.interestOps(OP_READ | OP_WRITE);
    } catch(IOException ex) {
      LOGGER.pushLog("Cannot connect client", ex);

      /* Se il client non può terminare la connessione
      allora la sua chiave non ci serve più. Restituiamo
      "true", cioè "hey, motore di selezione, questa la
      puoi anche buttare!" */
      return true;
    }
    return false;
  }

  /* Chiave leggibile. */
  private boolean read(SelectionKey key) {
    SocketChannel client = (SocketChannel)key.channel();
    BidiBuffer bidiBuffer = (BidiBuffer)key.attachment();

    /* Azzera il buffer diretto condiviso. */
    READ_BUFFER.clear();
    try {

      /* legge fino a READ_BUFFER.limit() byte */
      client.read(READ_BUFFER);
    } catch(IOException ex) {
```

```
      /* Anche un'eccezione in lettura è catastrofica. */
      LOGGER.pushLog("Client unreachable", ex);

      /* E comporta eliminazione del client: se non possiamo
      leggere significa che il client non può spedire e
      se non può spedire, che chat è? */
      MESSAGE_MANAGER.deleteClient(bidiBuffer.getId());
      return true;
    }

    /* Appronta il buffer per il "travaso" dei suoi dati
    nella parte di bidiBuffer che accumula il messaggio
    in arrivo */
    READ_BUFFER.flip();
    bidiBuffer.accumulateIn(READ_BUFFER);
    ByteBuffer bufferData = bidiBuffer.getInData();
    if(bufferData != null) {//hai finito, bidiBuffer?
      bufferData.flip();

      /* I dati letti finiscono nel MESSAGE_MANAGER. Nel caso
      del Server, quel messaggio sarà pronto per la propagazione
      agli altri client. Nel caso del Client quel messaggio
      finirà nella console perchè, vedremo, il Client usa un
      MessageManager un po' particolare per il suo KeyProcessor */
      MESSAGE_MANAGER.enqueueMessage(new Message(bufferData));

      /* Appronta il bidiBuffer per la lettura del prossimo
      messaggio */
      bidiBuffer.resetIn();
    }

    /* Se siamo arrivati qui, non ci sono stati problemi */
    return false;
  }

  /* Chiave scrivibile */
  private boolean write(SelectionKey key) {
    SocketChannel client = (SocketChannel)key.channel();
    BidiBuffer bidiBuffer = (BidiBuffer)key.attachment();

    /* Se non c'è niente che attende di essere scritto
    nel BidiBuffer allegato... */
    if(bidiBuffer.hasOut() == false) {

      /* Allora chiede al MessageManager: hey, c'è qualche
      messaggio in coda per me? */
      Message nextMessage = MESSAGE_MANAGER.getNextMessage(bidiBuffer.getId());

      /* Se c'è... */
      if(nextMessage != null) {

        /* Usa i dati di quel messaggio per riempire il buffer
        in "uscita" del BidiBuffer allegato */
        bidiBuffer.setOut(nextMessage.getBuffer());
      }
    }

    /* Se prima era vuoto, adesso potrebbe essere pieno */
    if(bidiBuffer.hasOut()) {
      try {
        bidiBuffer.writeOut(client);
      } catch(IOException ex) {
        LOGGER.pushLog("Client unreachable", ex);
        MESSAGE_MANAGER.deleteClient(bidiBuffer.getId());

        /* Se non posso scriverti, allora che chat-client sei? */
        return true;
      }
    }
```

```
    /* Se siamo qui significa che non siamo stati quattro linee
    sopra... tutto è andato bene*/
    return false;
  }

  /* Chiave accettabile: qualcuno bussa alla porta del
  ServerSocketChannel. */
  private boolean accept(SelectionKey key) {
    ServerSocketChannel server = (ServerSocketChannel)key.channel();
    SocketChannel client = null;

    /* Proviamo a pigliare 'sto client... */
    try {
      client = server.accept();
    } catch(IOException ex) {
      LOGGER.pushLog("Server cannot accept new client", ex);

      /* Se il server non riesce ad accettare la connessione
      non è detto che sia per questo da buttare... Per questo
      qui non restituiamo "true" come per gli altri metodi */
    }
    if(client != null) {

      /* Prende il nuovo ID del client ma non lo registra perchè
      la configurazione e la registrazione possono ancora andare
      a male */
      long id = MESSAGE_MANAGER.getNewClientId();
      BidiBuffer bidiBuffer = new BidiBuffer(id);

      /* Tenta di configurare il client in modalità non bloccante
      e tenta di registrare il client presso il selettore, allegandogli
      un BidiBuffer */
      try {
        client.configureBlocking(false);
        client.register(key.selector(), OP_READ | OP_WRITE, bidiBuffer);

        /* Se siamo qui allora è andato tutto bene. Invoca createNewClient.
        Questo metodo registra l'id restituita dalla precedente invocazione
        di getNewClientId() come id di un nuovo arrivato */
        MESSAGE_MANAGER.createNewClient();

      } catch(IOException ex) {
        LOGGER.pushLog("Cannot register client", ex);
        /* come sopra */
      }
    }

    /* In pratica per il KeyProcessor il ServerSocketChannel non
    va mai a male. */
    return false;
  }
}
```

Nulla di preoccupante, ripeto. Tutto già visto ma in una botta sola.

ChatServer.java

Anche ChatServer è un *deja-vu*. Quello che ha in più è la gestione delle eccezioni. Tutta la sarabanda di cose che possono andare male e di cui occorre tener conto affinchè, a prescindere dal tipo di evento catastrofico, il selettore sia sempre chiuso prima di mollare armi e bagagli. Non c'è molto da aggiungere. I metodi sono autoesplicativi, nel senso che hanno tanti commenti che leggendoli potrebbero spiegarsi tra di loro.

```
import java.io.*;
```

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import static java.nio.channels.SelectionKey.*;

/** Server della Chat a colpi di socket non bloccanti */
public class ChatServer implements ChatLogger, Runnable {

  /* Entry-Point! */
  public static void main(String[] args) {
    /* Crea un ChatServer e lo affida ad un nuovo Thread */
    ChatServer server = new ChatServer();
    new Thread(server, "Chat Server").start();

    /* Il server gira in parallelo al Thread Main e il Thread
    Main   può sedersi ad aspettare che l'utente digiti "quit" */
    System.out.println("Type quit [enter] to stop server");
    Scanner in = new Scanner(System.in);
    while(in.hasNextLine()) {
      String line = in.nextLine();
      if(line.equals("quit")) {
        server.stop();
        break;
      }
    }
  }

  /* Per l'InetSocketAddress del ServerSocket */
  private final String HOST = "localhost";
  private final int PORT = 9090;

  /* Questo è nel caso in cui si volesse passare ad un
  ChatLogger esterno */
  private final ChatLogger CHAT_LOGGER = this;

  /* Interruttore che controlla, insieme allo stato
  "open" del selettore, il ciclo di selezione. E'
  volatile perchè il suo valore può essere cambiato da
  un altro Thread. I Thread qui sono due: uno è quello
  che fa girare il selettore, creato nel main, l'altro è
  IL thread main, che è usato per gestire l'input dell'utente
  (attesa per la stringa "quit" che spegne tutto) */
  private volatile boolean loopEngine = true;

  /* Ferma tutto! */
  public void stop() {
    CHAT_LOGGER.pushLog("Server shutdow request", null);

    /* Questo ferma la faccenda "all'istante" perchè il selettore
    cicla con selectNow, che è non bloccante. In verità
    deve prima terminare la scrittura dei dati in partenza
    e la lettura dei dati in arrivo ma essendo entrambe non
    bloccanti si tratta al massimo di aspettare che ogni socket
    abbia ricevuto 512 byte (dimensione del buffer diretto in
    lettura) e che abbia scritto i pochi o molti byte del
    messaggio attualmente in partenza */
    loopEngine = false;
  }

  /* Questo è il run che sarà affidato al Thread del
  selettore */
  public void run() {
    loopEngine = true;
    Selector selector = null;
    ServerSocketChannel server = null;

    /* Tenta di creare entrambi */
    try {
```

```
      selector = Selector.open();
      server = ServerSocketChannel.open();
      server.socket().bind(new InetSocketAddress(HOST, PORT));
      server.configureBlocking(false);
      server.register(selector, OP_ACCEPT, null);
    } catch(IOException ex) {

      /* la cosa può andare male anche dopo che le istanze siano
      state create e pure dopo che siano state aperte. Ci
      assicuriamo che sia invocato il metodo close */
      selector = ChatUtils.<Selector>close(server);
      server = ChatUtils.<ServerSocketChannel>close(selector);
    }

    /* Se non c'è stata alcuna eccezione allora il selettore
    e il server saranno entrambi diversi da null. Controlliamo
    tutti e due perchè ci servono tutti e due, sebbene, il server
    sia usato indirettamente attraverso la sua chiave che risiede
    nel selettore */
    if(selector != null && server != null) {
      try {
        /* Fai partire il ciclo del selettore */
        startServerEngine(selector);
      } finally {

        /* e comunque vada a finire, assicurati che l'ultimo
        passo prima dell'oblio sia la chiusuare del selettore
        e del server, sempre che siano rimasti aperti */
        selector = ChatUtils.<Selector>close(selector);
        server = ChatUtils.<ServerSocketChannel>close(server);

        /* Sulla console compare il messaggio "libera tutti" */
        CHAT_LOGGER.pushLog("Server offline", null);
      }
    }
  }

  /* Questo è il metodo che contiene avvio e rotazione
  del selettore */
  private void startServerEngine(Selector selector) {

    /* Crea un processore delle chiavi che usa il CHAT_LOGGER
    (per noi l'istanza di questa classe ma lui non lo sa) e un
    nuovo MessageManager) */
    KeyProcessor keyProcessor = new KeyProcessor(CHAT_LOGGER, new MessageManager());

    /* Se deve girare e se può girare... */
    while(loopEngine && selector.isOpen()) {

      /* Tenta una selezione con eccezione catturata alla moda di parigi */
      Exception ex = ChatUtils.selectNow(selector);

      /* Tutto ok con la selezione? */
      if(ex == null) {

        /* Scorri le chiavi selezionate */
        Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
        while(keys.hasNext()) {

          /* Con rimoselezione volante */
          SelectionKey key = ChatUtils.getAndRemove(keys);

          /* e gestione delocalizzata che potrebbe restituire "false" se
          la chiave, a detta del processore, risulti non più utile */
          if(keyProcessor.processKey(key)) {

            /* caso in cui va cancellata */
            key.cancel();
          }
```

```
        }
      } else { //La selezione non è andata bene. Evento epocale.
        CHAT_LOGGER.pushLog("Selection error", ex);

        /* che causa lo stop del server */
        loopEngine = false;
      }

      /* Una pausa per la cpu, che anche questa potrebbe andar male... */
      if(ChatUtils.pause(50)) {

        /* ed parimenti catastrofica: qualcuno ha detto a questo Thread che è
        ora di fare i bagagli. */
        loopEngine = false;
        return; //e li facciamo più in fretta possibile (il finally chiuderà tutto)
      }
    }
  }

  /** E' triste, lo so :D */
  public void pushLog(String message, Exception ex) {
    System.out.printf("Date: %s%nMessage: %s%nException: %s%n%n",
      new Date(), message, ex == null ? "none" : ex);
  }
}
```

Non resta che il gran finale.

ChatClient.java

Colgo l'occasione per annoiarvi con qualche elucubrazione. Che gioia. ChatClient è stato scritto prima di ChatServer. A dimostrazione che non è mai la stessa acqua che passa due volte sotto lo stesso ponte, è un po' diverso da ChatServer. Parte di questa diversità è necessaria per una differenza di obiettivi ma un'altra generosa porzione è in verità diversa perchè è il frutto di una visione più primitiva, per quanto possa esserlo un ragionamento che precede di qualche ora quello più voluto. Che sia una diversità qualititativamente diversa, passatemi il gioco di parole, si evince da ciò che nonostante essa sia manifesta comunque le due classi, ChatServer e ChatClient, presentano un'affinità struttura. Quel genere di affinità nella differenza che, agli occhi di un programmatore, fa venir voglia di vedere se non si possa risolvere con un colpo d'astrazione. Nella fattispecie è il motore che cicla sul selettore ad essere affine tra le due classi. Comunque sia, sappiata che dovrebbe saltarvi in testa, vedendo ChatClient dopo ChatServer, che dai due sia ricavabile un terzo comune.

Bene, dopo avervi intontiti in modo tale che possano sfuggirvi le stupidate che ho scritto in ChatClient, passo a darvi il codice.

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import static java.nio.channels.SelectionKey.*;

/** Client della chat con socket non bloccanti */
public class ChatClient extends MessageManager implements ChatLogger, Runnable {

  /** Entry-Point! */
  public static void main(String[] args) {

    /* Avvia lo strato di comunicazione in un Thread a parte */
    ChatClient client = new ChatClient();
    new Thread(client).start();

    /* Il Thread Main prosegue con la parte di interazione */
```

```
    System.out.println("ChatClient Sample. Type quit [enter] to shutdown. Type and
press enter to send message.");
    Scanner in = new Scanner(System.in);
    while(in.hasNextLine()) {
      String line = in.nextLine();
      if(line.equals("quit")) {
        client.stop();
        ChatUtils.close(in);
        return;
      } else {

        /* Se non è quit allora è un messaggio. La stringa viene
        impacchettata in un messaggio e il messaggio viene affidato al
        MessageManager-ChatClient client. L'invocazione di superEnqueueMessage
        chiama la definizione originale di enqueueMessage che causa la
        disponibilità del messaggio per l'invio. La super definizione è
        necessaria perchè la ri-definizione che avviene in questo
        ChatClient-MessageManager fa tutt'altro */
        client.superEnqueueMessage(new Message(line));
      }
    }
  }

  private final ChatLogger CHAT_LOGGER = this;

  /* Pensa che me lo sono chiesto anch'io: ma c'era proprio bisogno di
  'sto final CHAT_LOGGER? */
  private KeyProcessor keyProcessor = new KeyProcessor(CHAT_LOGGER, this);

  /* Qui selettore e socket sono campi perchè è il metodo stop
  ad occuparsi della loro chiusura anzichè, come in ChatServer,
  delegare la chiusura alle operazioni di pulizia del motore
  di selezione. Il selettore è anche usato come interruttore di
  controllo del motore di selezione: cicla finchè il selettore
  è "open". */
  private Selector selector;
  private SocketChannel client;

  /* Ferma il motore del client */
  public void stop() {
    /* Chiudendo il selettore termina anche il ciclo
    di selezione */
    ChatUtils.close(selector);

    /* Logica vuole che "liberando" qui il selettore
    si liberi anche il client */
    ChatUtils.close(client);
  }

  /** Tre righe da mal di testa. Invoca la super definizione
  del metodo enqueueMessage. Tale invocazione causa l'iniezione
  del messaggio m nella coda di propagazione dei messagi di questo
  MessageManager (ChatClient è un MessageManager). La super
  invocazione in un metodo diverso consente a questo ChatClient di usare il suo
  metodo enqueueMessage, invocato dal KeyProcessor, per stampare
  il messaggio ricevuto tramite il KeyProcessor */
  public void superEnqueueMessage(Message m) {
    super.enqueueMessage(m);
  }

  /* Questo è il metodo che invoca KeyProcessor quando gli
  arriva un nuovo messaggio (che non può che arrivare dal Server). Il
  messaggio è riprodotto sulla console con la preposizione "CHAT: " */
  public void enqueueMessage(Message m) {
    System.out.println("CHAT: " + ChatUtils.decode(m));
  }

  /** Questo è l'archetipo della tristezza di quell'altro */
  public void pushLog(String message, Exception ex) {
```

```
  System.out.printf("Date: %s%nMessage: %s%nException: %s%n%n",
    new Date(), message, ex == null ? "none" : ex);
}

/** Compito del Thread che esegue lo strato "socket" di ChatClient */
public void run() {
  SocketChannel client = null;
  Selector selector = null;

  /* Tenta la creazione... */
  try {
    client = SocketChannel.open();
    client.configureBlocking(false);
    selector = Selector.open();
    InetSocketAddress address = new InetSocketAddress("localhost", 9090);

    /* Qui vale quanto detto in quella parte del tutorial riferita ai
    SocketChannel client: OP_CONNECT se la connessione non bloccante
    ci dica, tramite un "false", che l'operazione deve essere compiuta
    in due tempi. Altrimenti siamo pronti a leggere e scrivere */
    int ops = client.connect(address) ? OP_READ | OP_WRITE : OP_CONNECT;

    /* Registrazione e associazione ad un BidiBuffer. L'id del client,
    l'unico di questo MessageManager,  è prodotto da questo ChatClient
    (appunto in quanto MessageManager). */
    client.register(selector, ops, new BidiBuffer(createNewClient()));
  } catch(IOException ex) {

    /* Errori vari in partenza. Chiudi tutto quello che è
    stato eventualmente aperto. Informa l'utente con un log */
    CHAT_LOGGER.pushLog("Cannot create client", ex);
    client = ChatUtils.<SocketChannel>close(client);
    selector = ChatUtils.<Selector>close(selector);
  }

  /* Qui si vede forse anche di più quella somiglianza
  strutturale citata nei miei vaneggiamenti. */
  if(client != null && selector != null) {
    this.selector = selector;
    this.client = client;
    try {
      loopSelector(selector);
    } finally {

      /* Non è detto che sia il metodo "stop" a causare
      la restituzione del controllo da parte del metodo
      che contiene il ciclo del selettore. Noi dobbiamo
      assicurarsi che sia tutto chiuso a prescindere. */
      ChatUtils.close(this.selector);
      ChatUtils.close(this.client);
    }
  }
}

/* Metodo che contiene il ciclo del selettore */
private void loopSelector(Selector selector) {

  /* Finchè il selettore è aperto... */
  while(selector.isOpen()) {

    /* selectNow con eccezione */
    Exception ex = ChatUtils.selectNow(selector);

    if(ex != null) {

      /* Ecco un caso di uscita senza stop: la selezione
      ha prodotto un'eccezione: usciamo e il finally
      pulirà le connessioni */
      CHAT_LOGGER.pushLog("Selection failed", ex);
```

```
        break;
      } else {

        /* Il ciclo sulle chiavi è affidato ad un altro metodo */
        loopKeys(selector.selectedKeys().iterator());
      }

      if(ChatUtils.pause(50)) {
        /* vale quanto detto per il Server. L'interrupt va
        sempre preso sul serio */
        break;
      }
    }

    /* Qui diciamo "type quit" perchè c'è caso che il ciclo termini
    ma sia ancora attivo quello del Thread Main, il quale aspetta
    che l'utente scriva quit per finire. E' un po' sgraziato ma
    non il punto non è la gestione dell'interazione utente. Licenza poetica, via */
    CHAT_LOGGER.pushLog("Client shutdown, type quit [enter] to exit", null);
  }

  /* Ciclo sulle chiavi */
  private void loopKeys(Iterator<SelectionKey> keys) {
    while(keys.hasNext()) {
      SelectionKey key = ChatUtils.getAndRemove(keys);
      boolean cancel = keyProcessor.processKey(key);
      if(cancel) {
        /* Qui c'è una sola chiave, quella del SocketChannel. Se diventa
        invalida questa, non ha più senso che l'applicazione giri. */
        stop();
      }
    }
  }
}
```

ChatClient conclude l'insieme delle classi dell'applicazione.

Fine

C'è qualche altra ragione, oltre alla semplicità d'uso che spero sia emersa con chiarezza, per cui l'idea di usare i socket non bloccanti possa risultare allettante? Bisogna vedere qual'è l'alternativa. L'altra metà della mela è il multi threading. Teoricamente il server che accetta le connessioni crea per ogni client ricevuto un Thread che gestisce la comunicazione con quel client. Consideriamo irrilevante il peso in più dato dalla presenza di questi molti Thread. La gestione della comunicazione di rete, indotta da applicazioni che presuppongono una produzione discreta di messaggi, mi pare soggetta a due categorie di interruzioni, due categorie di eventi che generano un'attesa improduttiva del Thread che porta avanti la comunicazione. La prima categoria è quella degli interrupt, prodotti dalla necessità di acquisire una risorsa hardware. Comunica, invia i dati alla scheda di rete, fai qualcos'altro, la scheda dice che ha finito, riprendi l'esecuzione delle istruzioni del Thread, comunica... eccetera. La seconda categoria, decisamente più rilevante, è quella delle interruzioni da interazione. Supponendo che alla console della chat non ci sia un pazzo che digita messaggi alla velocità della luce, senza soste, il Thread che gestisce un solo client ha delle prolungate pause spezzate dall'effettivo invio di un messaggio da parte dell'utente. L'idea del Selettore, delle operazioni non bloccanti e del Thread "uno per tutti" riduce certamente l'improduttività derivante da questa seconda categoria di pause:. Se un client non comunica il Thread maneggia il traffico generato da un altro client. Entra in pausa se e solo se non ci sia alcuna comunicazione in corso, da parte di nessuno. Per prudenza non mi pronuncio sulla minimizzazione delle pause da interrupt. Può darsi che esista anche questo beneficio ma ritengo che dipenderebbe dai dettagli di realizzazione delle librerie.