Laboratorio di Reti
Lezione 8
Channel Multiplexing

12/11/2020

Laura Ricci

JAVA NIO: OBIETTIVI

- fast buffered binary e character I/O (in una lezione precedente)

  "*provide new features and improved performance in the areas of buffer management, scalable network and file I/O, character-set support, and regular-expression matching*"

- "non blocking mode" e multiplexing (questa lezione)

  "*production-quality web and application servers that scale well to thousands of open connections and can easily take advantage of multiple processors*"

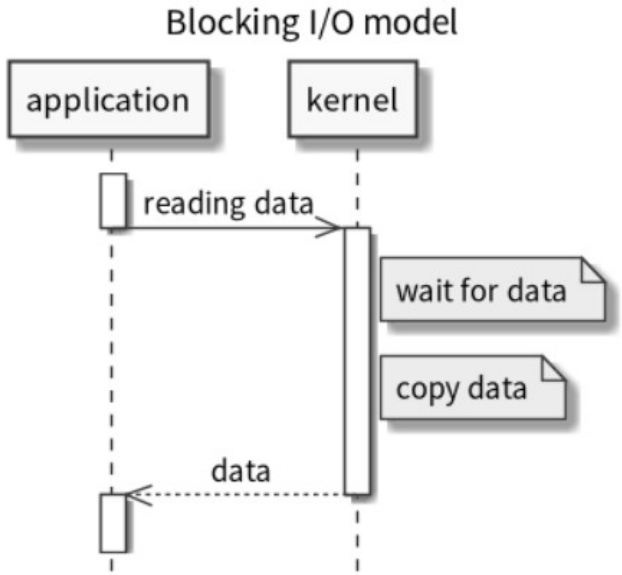
in questa lezione:

- non blocking channels associati a socket
- multiplexing: `Selector`

IL MODELLO BLOCKING IO

- operazioni bloccanti su stream: l'applicazione esegue una chiamata di sistema e si blocca fino a che tutti i dati sono ricevuti nel kernel, e copiati dal kernel space alla memoria della applicazione
- `read( )`
  - Si blocca fino a quando non è stato letto

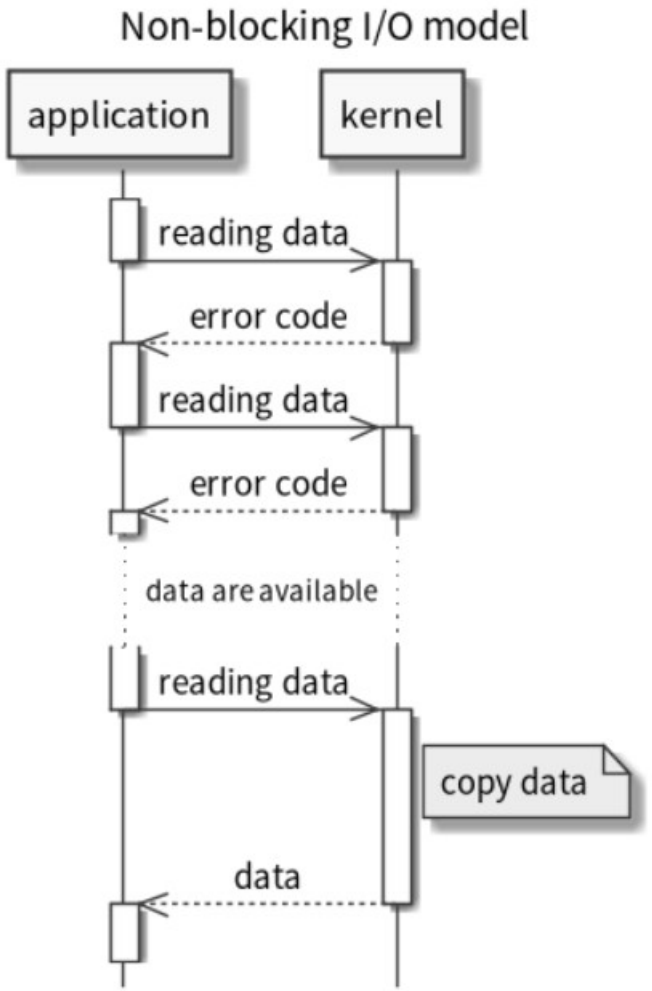
    un byte, un vettore di byte, un intero,...



- `accept( )`
  - si blocca fino a che non viene stabilita una nuova connessione
- `write(byte [ ] buffer)`
  - si blocca fino a che tutto il contenuto del buffer è stato copiato sulla periferica di I/O

IL MODELLO NON-BLOCKING IO

- la chiamata di sistema restituisce il controllo alla applicazione prima che l'operazione richiesta sia stata "pienamente soddisfatta".
- scenari possibili
  - restituiti i dati disponibili, o una parte di essi
  - operazione I/O non possibile, restituito un codice errore o valore `null`
- per completare l'operazione
  - effettuare system-call ripetute
  - fino a che l'operazione non può essere effettuata
- possibile con `SocketChannels`, `SocketServerChannels`



SOCKET CHANNEL

- non blocking I/O è possibile se si associano i channels ai socket
- un channel associato ad un socket TCP
  - "combinazione" di un socket e canale di comunicazione bidirezionale
  - scrive e legge da un socket TCP
  - estende la classe `AbstractSelectableChannel` e da questa mutua la capacità di passare dalla modalità bloccante a quella non bloccante
  - in modalità bloccante funzianamento simile a quello degli stream socket, ma con interfaccia basata su buffers
- ogni `socketChannel` è associato ad un oggetto `Socket` della libreria `java.net`
  - il socket associato può essere reperito mediante il metodo `socket()`

SERVER SOCKET CHANNEL

- un SelectableChannel collegato ad un TCP welcome socket (listening sockets)
- ad ogni `ServerSocketChannel` è associato un oggetto `ServerSocket`
  - blocking: come `ServerSocket`, ma con interfaccia buffer-based
  - non blocking: permette multiplexing di canali

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
ServerSocket socket = serverSocketChannel.socket();
socket.bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);
while(true){
    SocketChannel socketChannel = serverSocketChannel.accept();
    if(socketChannel != null){
        //do something with socketChannel...
    else //do something useful... }
```

- `InetSocketAddress`: rappresenta l'indirizzo di un socket descritto da indirizzo IP e numero di porta, altrenativo rispetto `InetAddress`

SOCKET CHANNEL

- associati ad un oggetto di tipo Socket ed utilizzati per la comunicazione dei dati tra client e server
- creazione di un `SocketChannel`
  - implicita creato se si accetta una connessione su un `ServerSocketChannel`.
  - esplicita, lato client, quando si apre una connessione verso un server, mediante una operazione di `connect()`

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("www.google.it", 80));
```

`InetSocketAddress` può essere specificato direttamente nella open, in questo caso viene effettuata implicitamente la connect

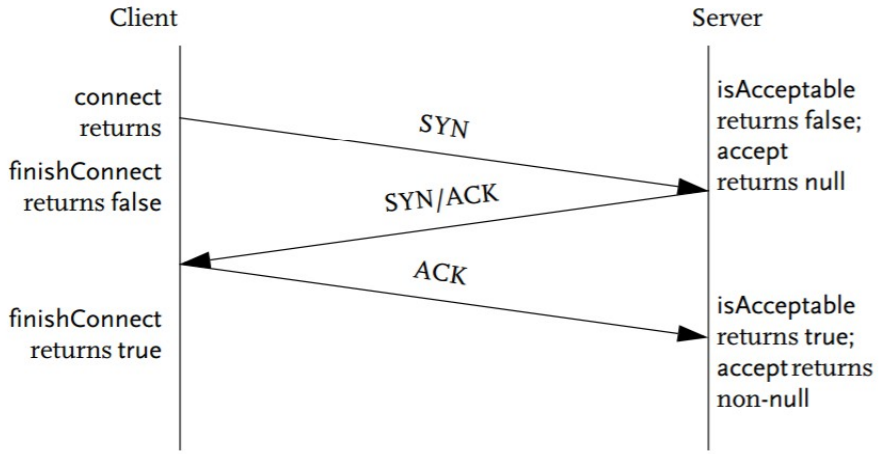
- modalità blocking/non blocking:

```
SocketChannel.configureBlocking(false);
```

- non blocking, lato client, significativa ad esempio nel caso in cui:
  - un'applicazione lato client che deve gestire l'interazione con l'utente, mediante GUI e contemporaneamente, gestire uno o più sockets

NON BLOCKING CONNECT

- può restituire il controllo al chiamante prima che venga stabilita la connessione.



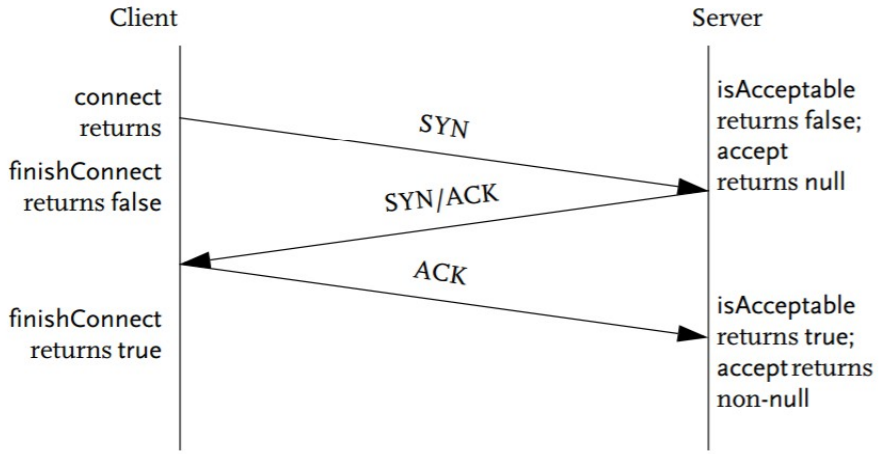
isAcceptable
restituisce vero quando la connessione può essere accettata

- `finishConnect()` per controllare la terminazione della operazione.

```
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress("www.google.it", 80));
while(! socketChannel.finishConnect() ){
        //wait, or do something else...    }
```

NON BLOCKING CONNECT

- può restituire il controllo al chiamante prima che venga stabilita la connessione.



`isAcceptable`
restituisce vero quando la connessione può essere accettata

- se l'ultima fase del three way handshake non è completo quando il client effettua la read, la read restituerà 0 valori nel buffer

- se si toglie

```
while(! socketChannel.finishConnect() )
        { //wait, or do something else... }
```

viene sollevata `java.nio.channels.NotYetConnectedException`

BLOCKING E NON BLOCKING: RIASSUNTO

- Blocking accept: si blocca finchè non arriva una richiesta di connessione
- Non-Blocking accept: controlla se c'è una richiesta da accettare (se c'è accetta) e ritorna
- Blocking write: si blocca finchè la scrittura dei dati nel buffer non è completata
- Non-Blocking write: tenta di scrivere i dati nella socket, ritorna immediatamente, anche se i dati non sono stati completamente scritti
- Blocking read: si blocca in attesa di byte da leggere
- Non-blocking read: ritorna immediatamente e restituisce il numero di byte letti (anche 0)

SERVER MODELS

Criteri per la valutazione delle prestazioni di un server:

- scalability: capacità di servire un alto numero di client che inviano richieste concorrentemente
- acceptance latency: tempo tra l'accettazione di una richiesta da parte di un client e la successiva
- reply latency: tempo richiesto per elaborare una richiesta ed inviare la relativa risposta
- efficiency: utilizzo delle risorse utilizzate sul server (RAM, numero di threads, utilizzo della CPU)

UN SINGOLO THREAD

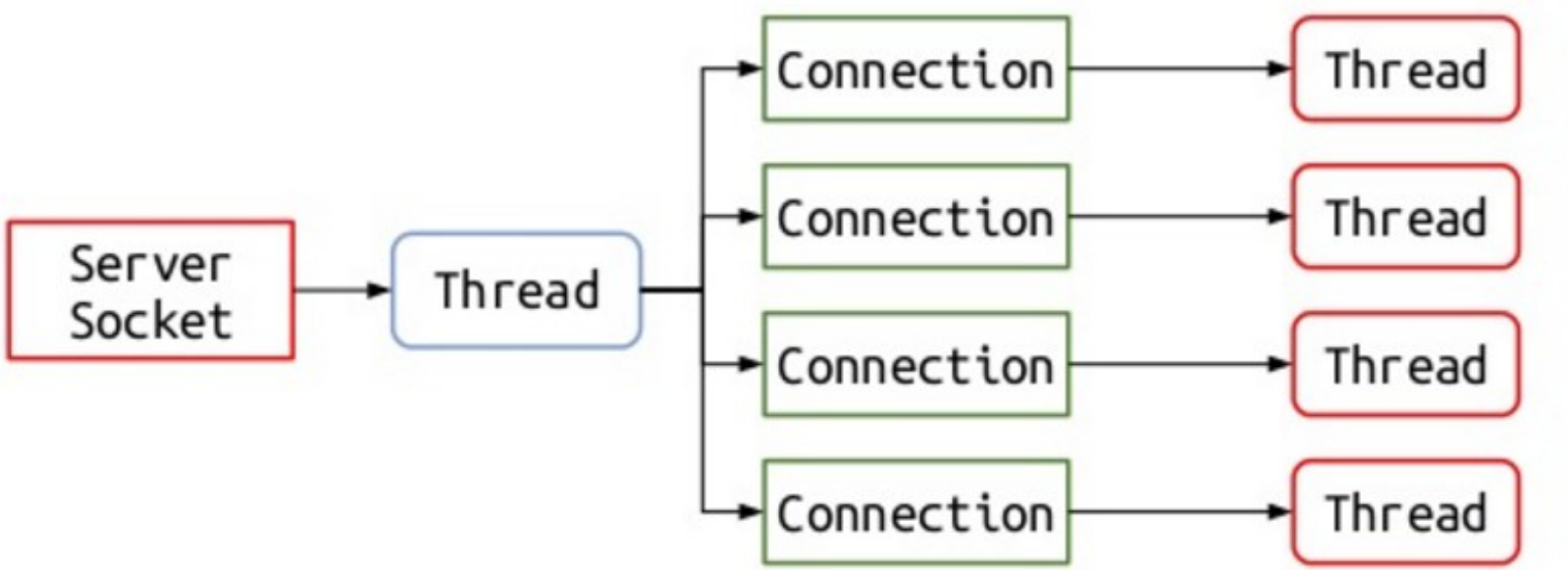
Un solo thread per tutti client:

- scalabilità: nulla, in ogni istante, solo un client viene servito
- accept latency: alta, il "prossimo" cliente deve attendere fino a che il primo cliente termina la connessione
- reply latency bassa: tutte le risorse a disposizione di un singolo client
- efficiency: buona, il server utilizza esattamente le risorse necessarie per il servizio dell'utente.
- adatto quando il tempo di servizio di un singolo utente è garantito rimanere rimanga basso

UN THREAD PER OGNI CONNESSIONE

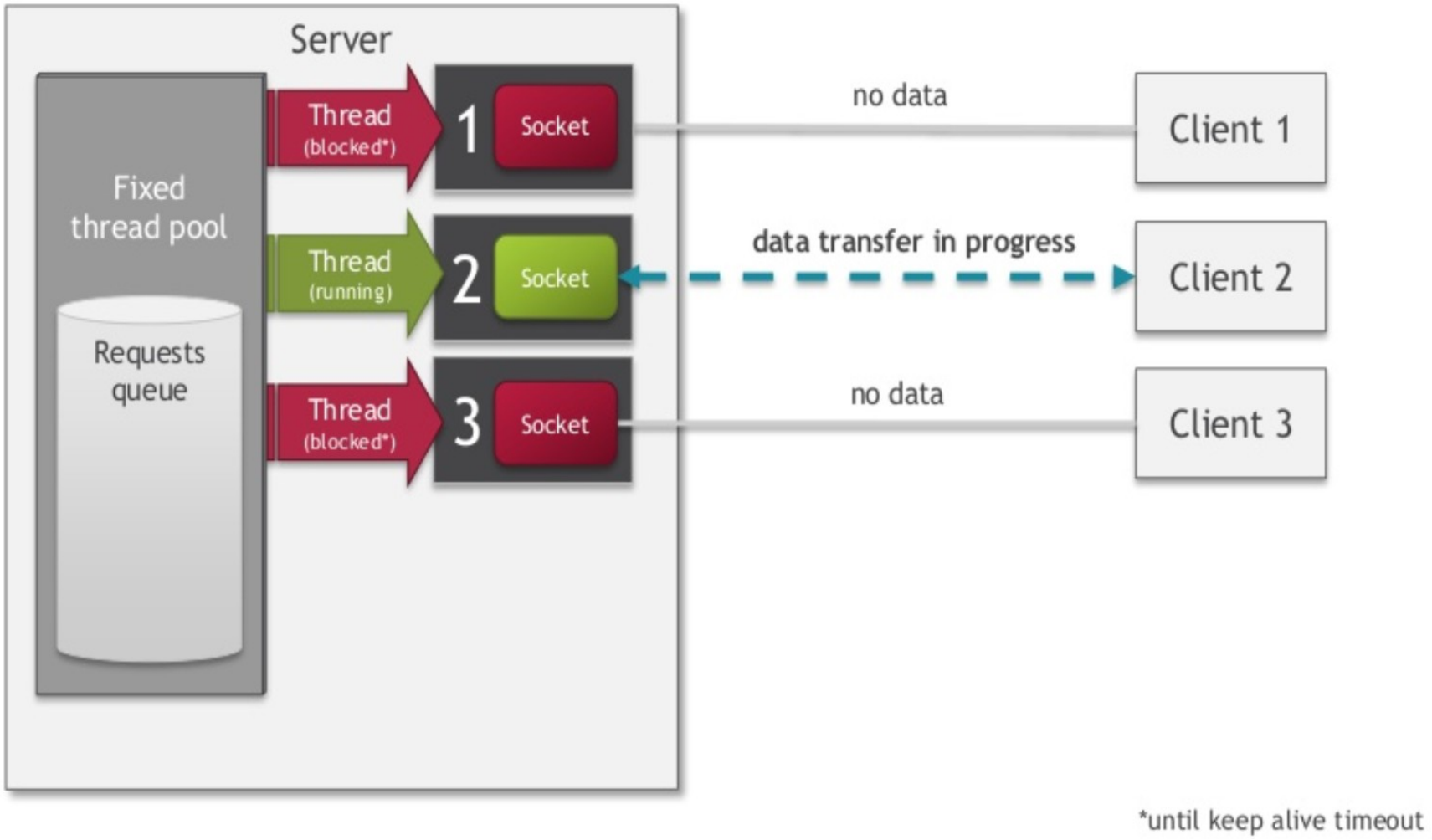
- scalabilità: possibile servire diversi clienti in maniera concorrente, fino al massimo numero di thread previsti per ogni processo
  - ogni thread alloca il proprio stack: memory pressure
  - impossibile predire il numero massimo di client: dipende da fattori esterni e può essere molto variabile
- accept latency: tempo tra l'accettazione di una connessione e la successiva è in genere basso rispetto a quello di interarrivo delle richieste
- reply latency: bassa, le risorse del server condivise tra connessioni diverse
  - ragionevole uso di CPU e RAM per centinaia di connessioni, se aumenta, il tempo di reply può non essere accettabile
- efficiency: bassa
  - ogni thread può essere bloccato in attesa di IO, ma utilizza risorse come la RAM

UN THREAD PER OGNI CONNESSIONE

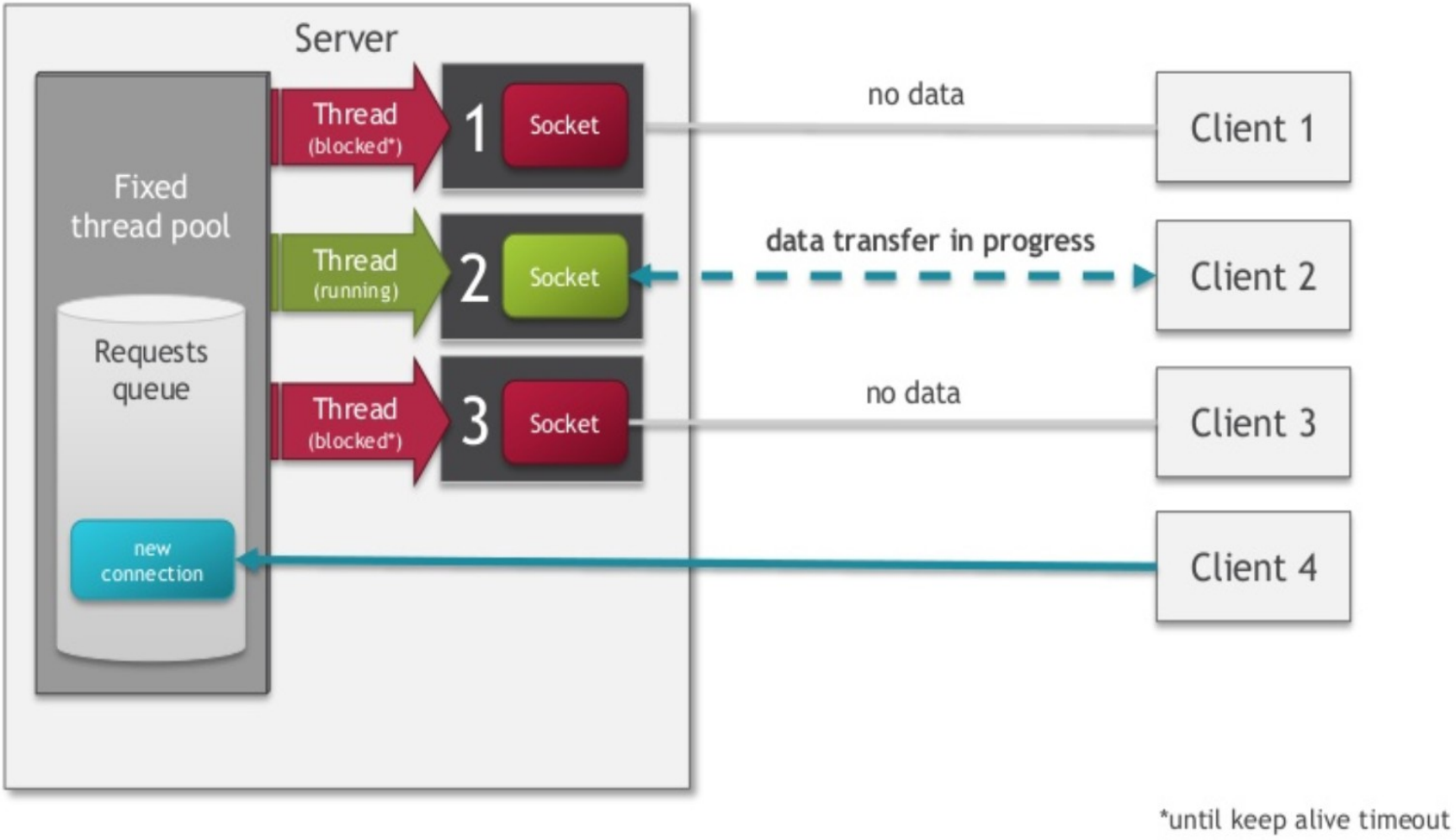


- attivazione di un thread per ogni connessione, de-attivazione a fine servizio
- quando un server monitora un grande numero di comunicazioni:
  - problemi di scalabilità: il tempo per il cambio di contesto può aumentare notevolmente con il numero di thread attivi
  - maggior parte del tempo impiegata in context switching

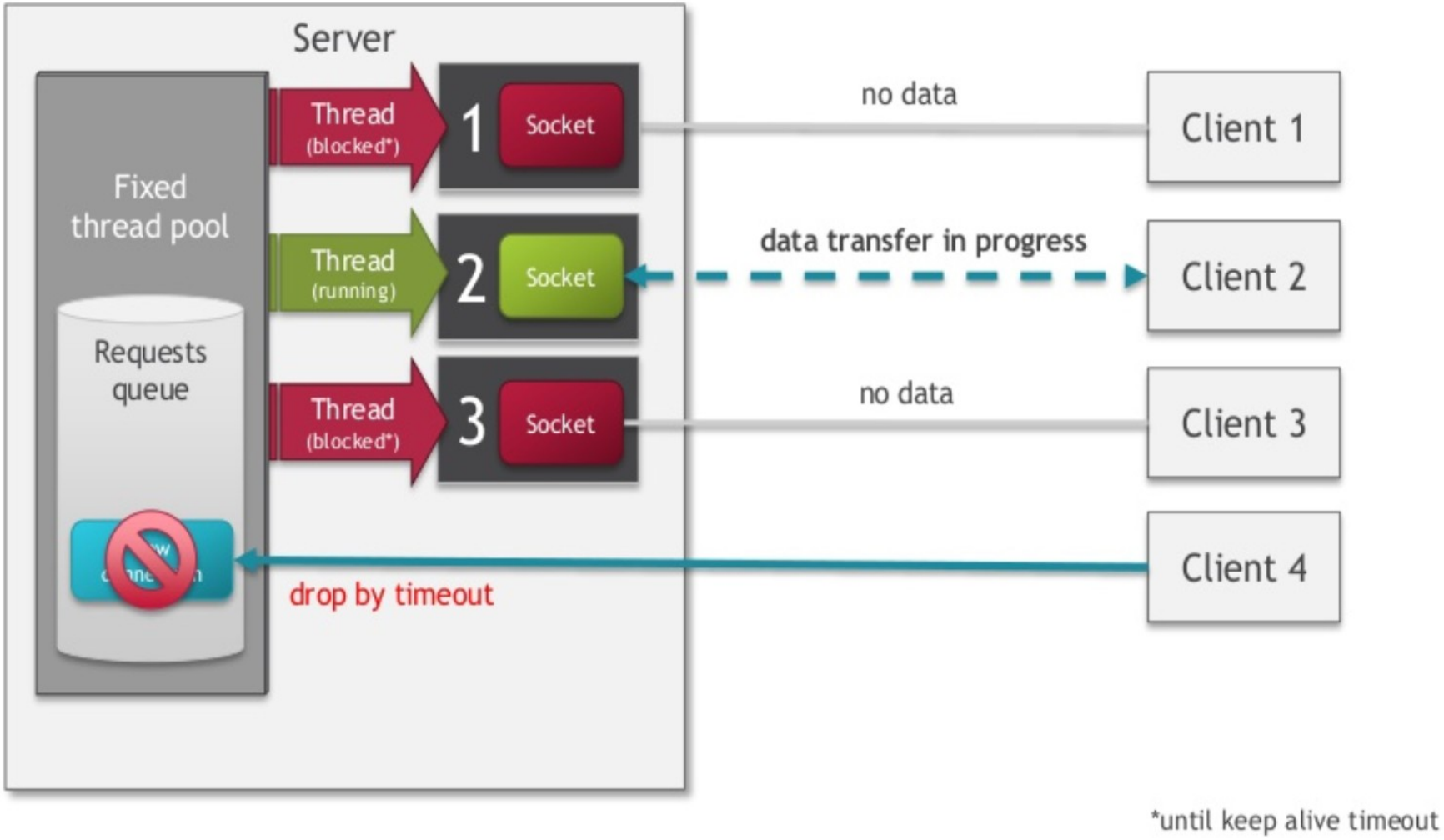
UN THREAD PER CONNESSIONE: FIXEDTHREADPOOL



UN THREAD PER CONNESSIONE: FIXEDTHREADPOOL



UN THREAD PER CONNESSIONE: FIXEDTHREADPOOL



UN NUMERO FISSO DI THREAD

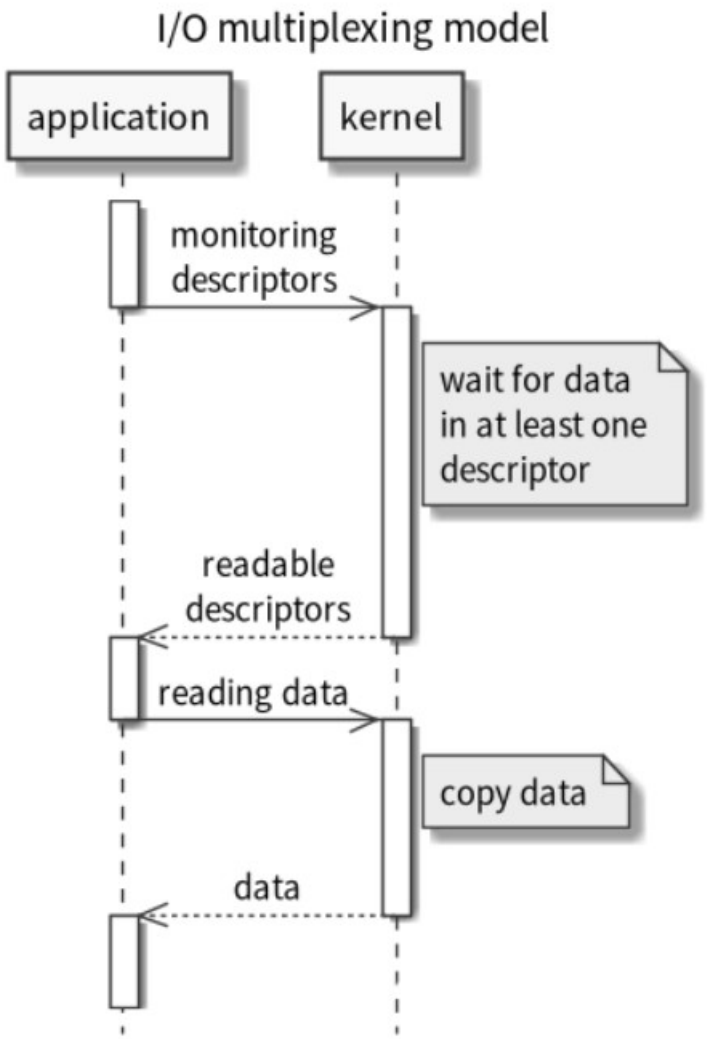
Un numero costante di thread: utilizza `Thread Pool`

- scalabilità: limitata al numero di connessioni che possono essere supportate.
- accept latency bassa fino ad un certo numero di connessioni
- reply latency: bassa fino al numero massimo di thread fissato, degrada se il numero di connessioni è maggiore
- efficiency: trade-off rispetto al modello precedente

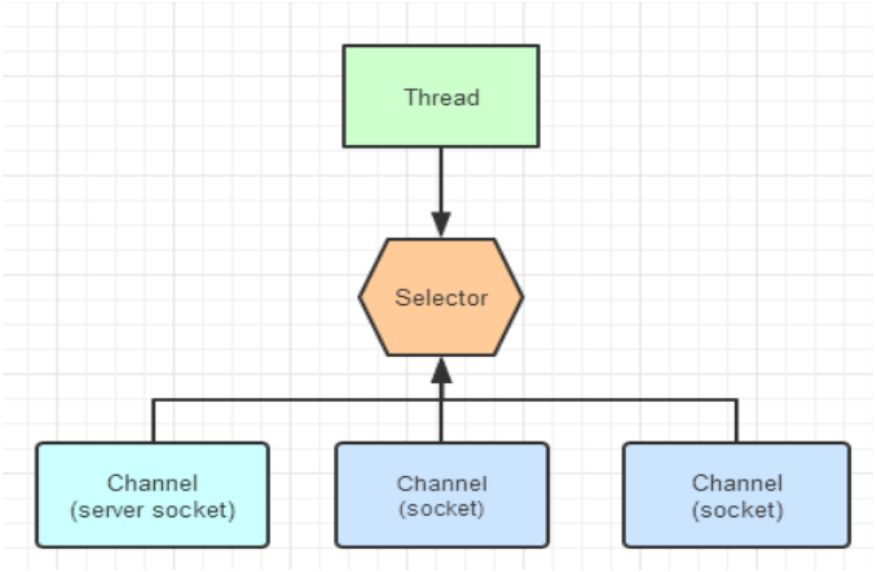
MULTIPLEXED I/O

- non blocking I/O con notifiche bloccanti
- l'applicazione registra "descrittori"

  delle operazioni di I/O a cui è interessato
- l'applicazione esegue una operazione di monitoring di canali
  - una system call bloccante
  - restituisce il controllo quando almeno un descrittore indica che una operazione di I/O è "pronta"
  - a quel punto si effettua una read non bloccante

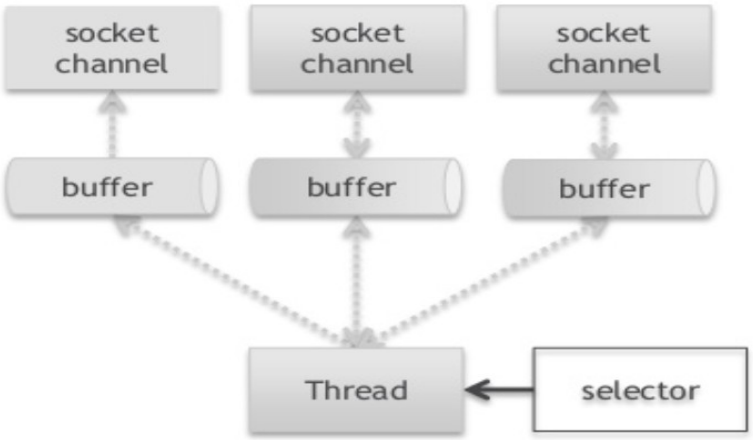


MULTIPLEXING IN JAVA



- selettore un componente che esamina uno o più `NIO Channels`, e determina quali canali sono pronti per leggere/scrivere
- più connessioni di rete gestite mediante un unico thread, consente di ridurre
  - thread switching overhead
  - uso di risorse per thread diversi
- possibile anche l'utilizzazione insieme insieme a multitherading

UN UNICO THREAD: MULTIPLEXING

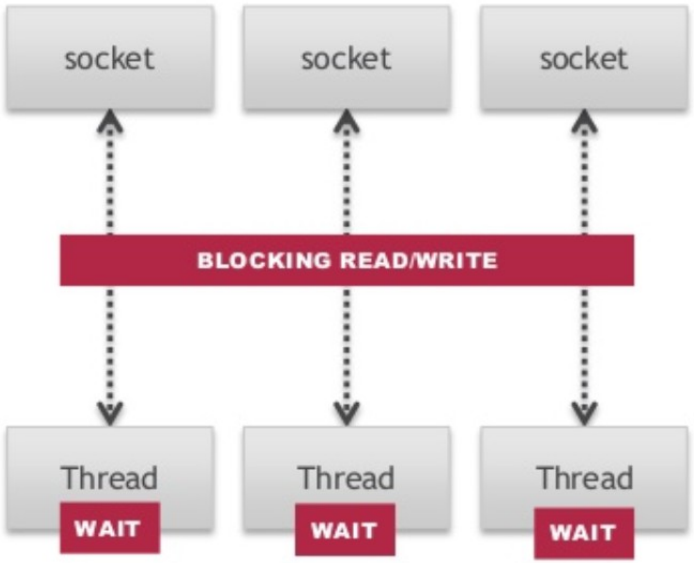


I/O multiplexing

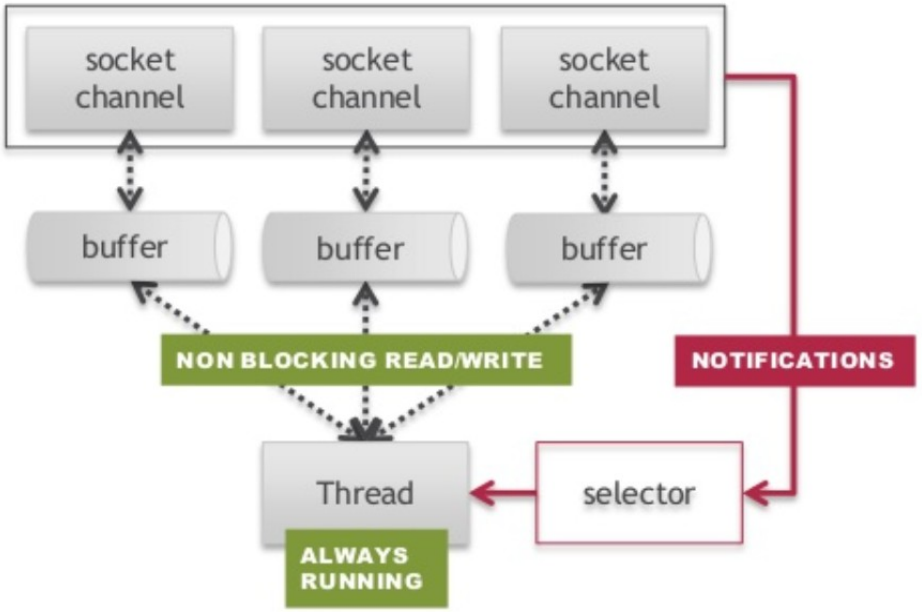
- un singolo thread che gestisce un numero arbitrario di sockets
- non un thread per connessione, ma un numero ridotto di threads
  - numero di thread basso anche con migliaia di sockets
  - anche un solo thread
- miglioramento di performance e scalabilità
- architettura più complessa da capire e da implementare

UN UNICO THREAD: MULTIPLEXING

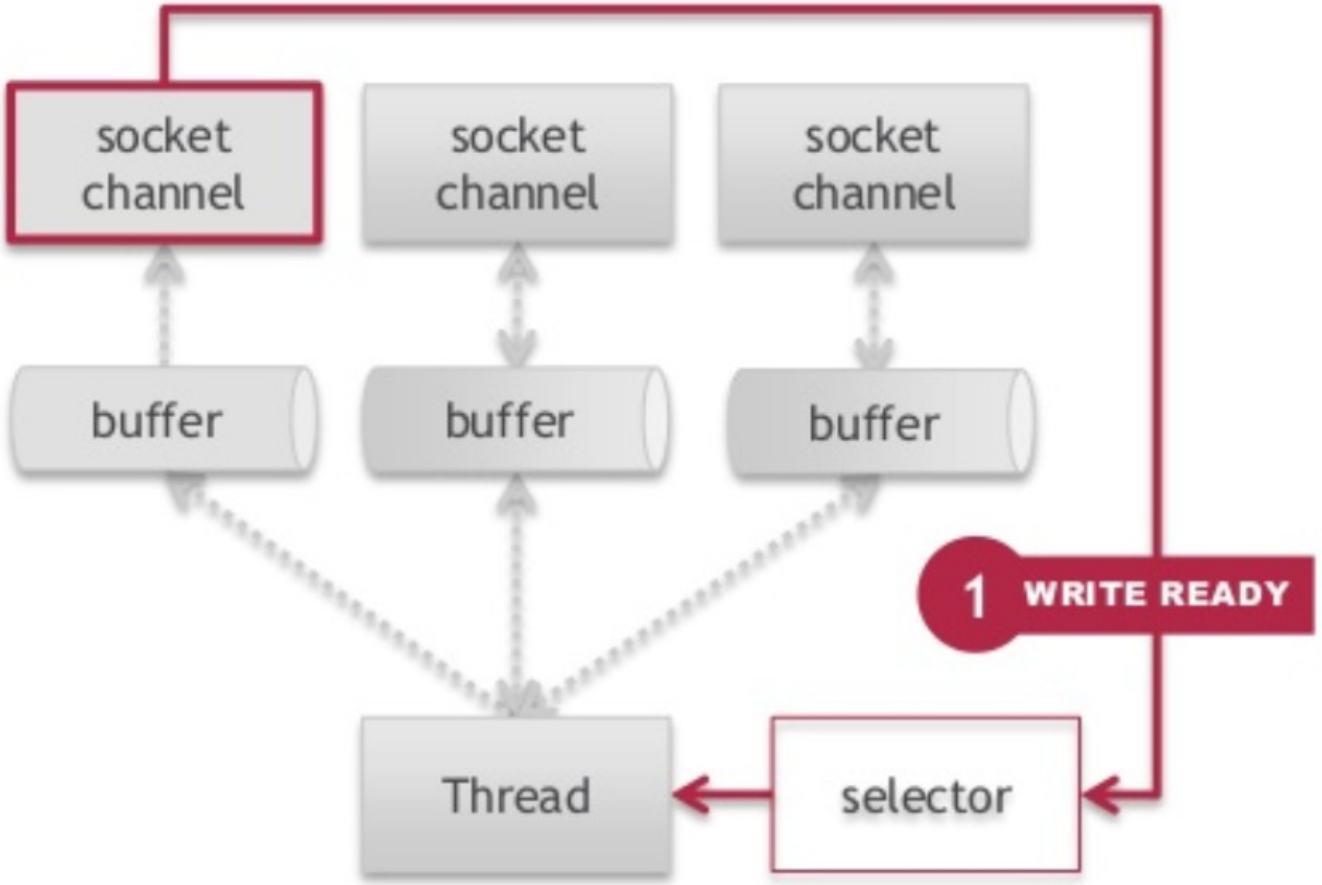
IO (blocking)



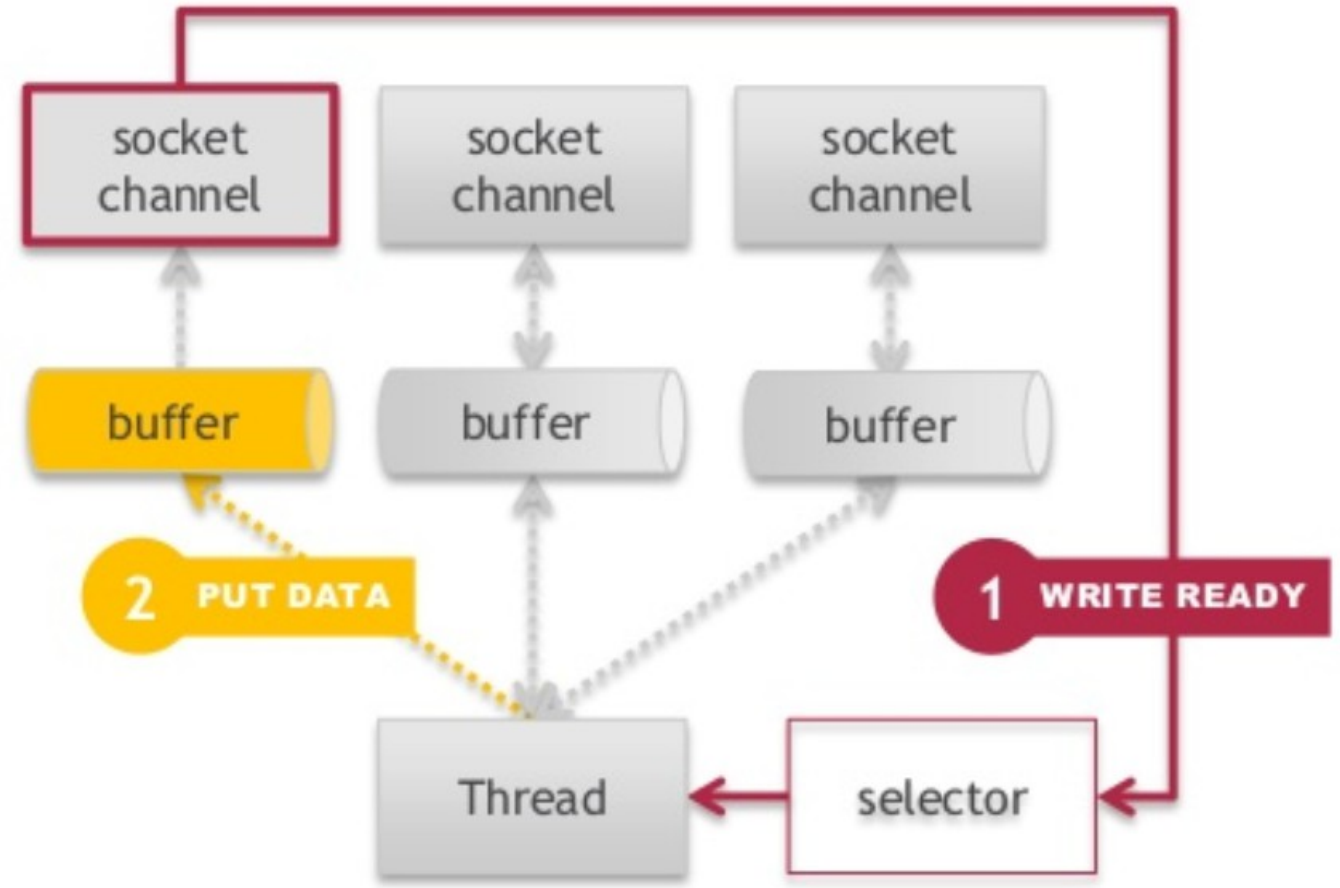
NIO (non-blocking)



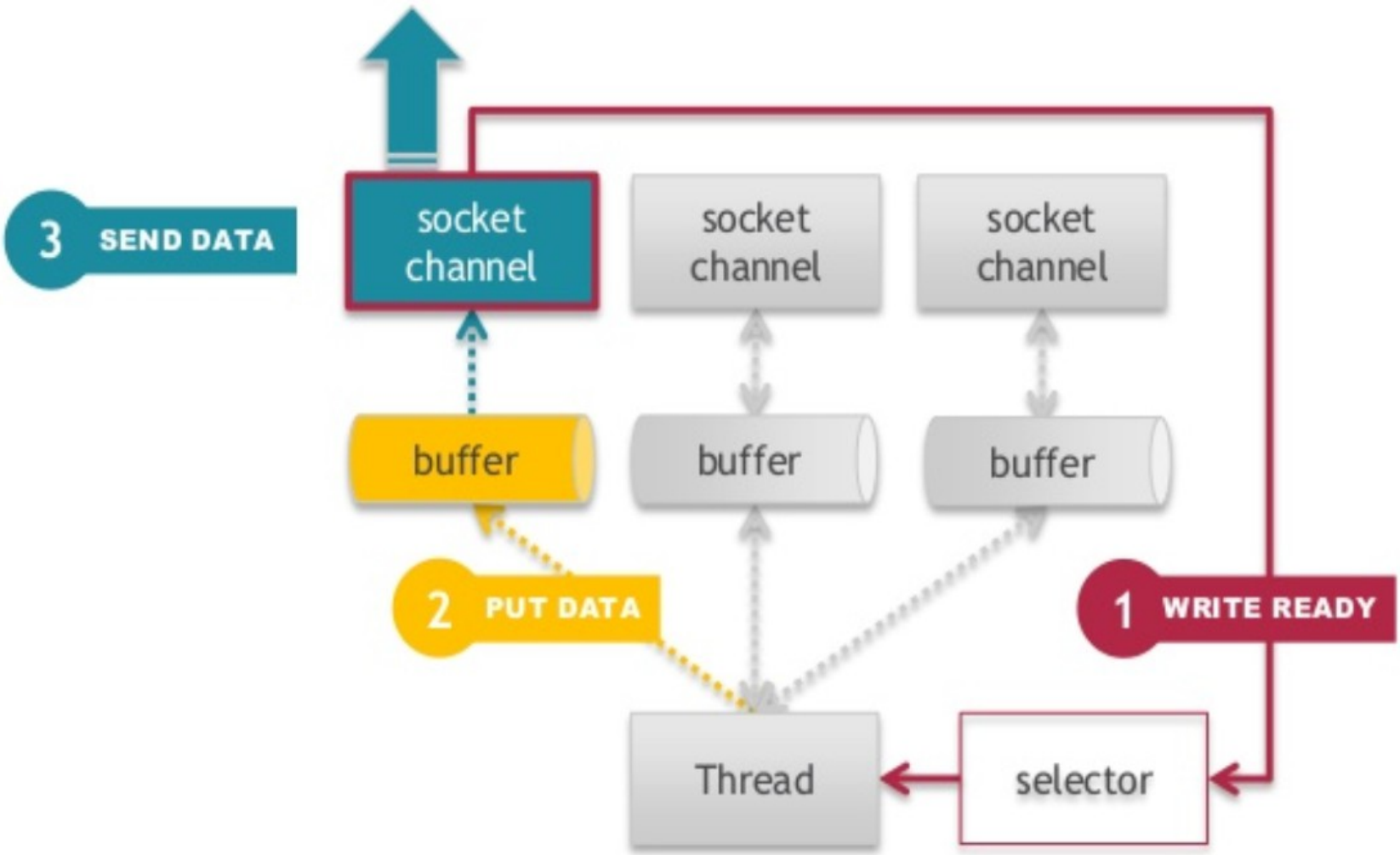
MULTIPLEXING: INVIO DEI DATI



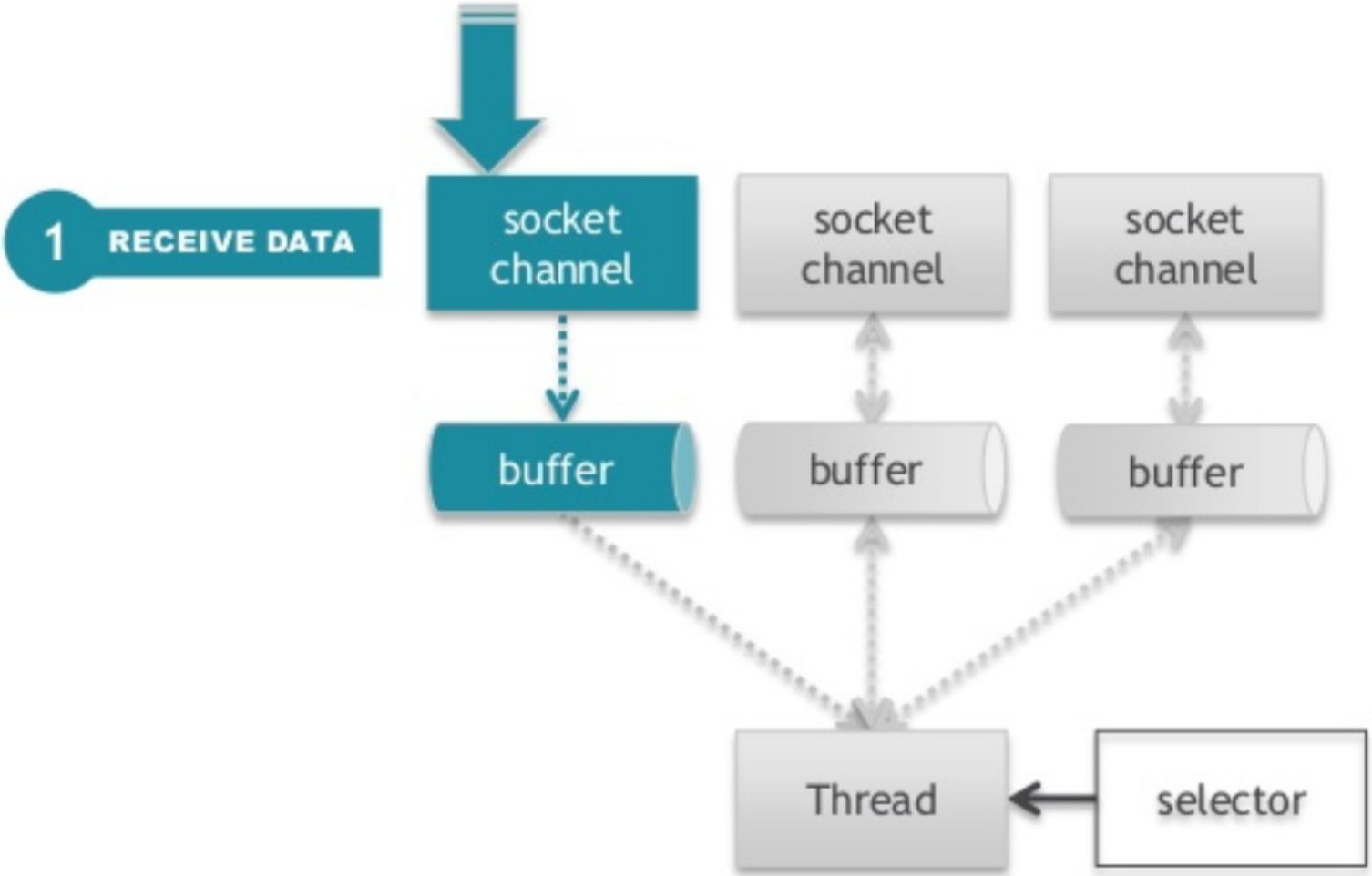
MULTIPLEXING: INVIO DEI DATI



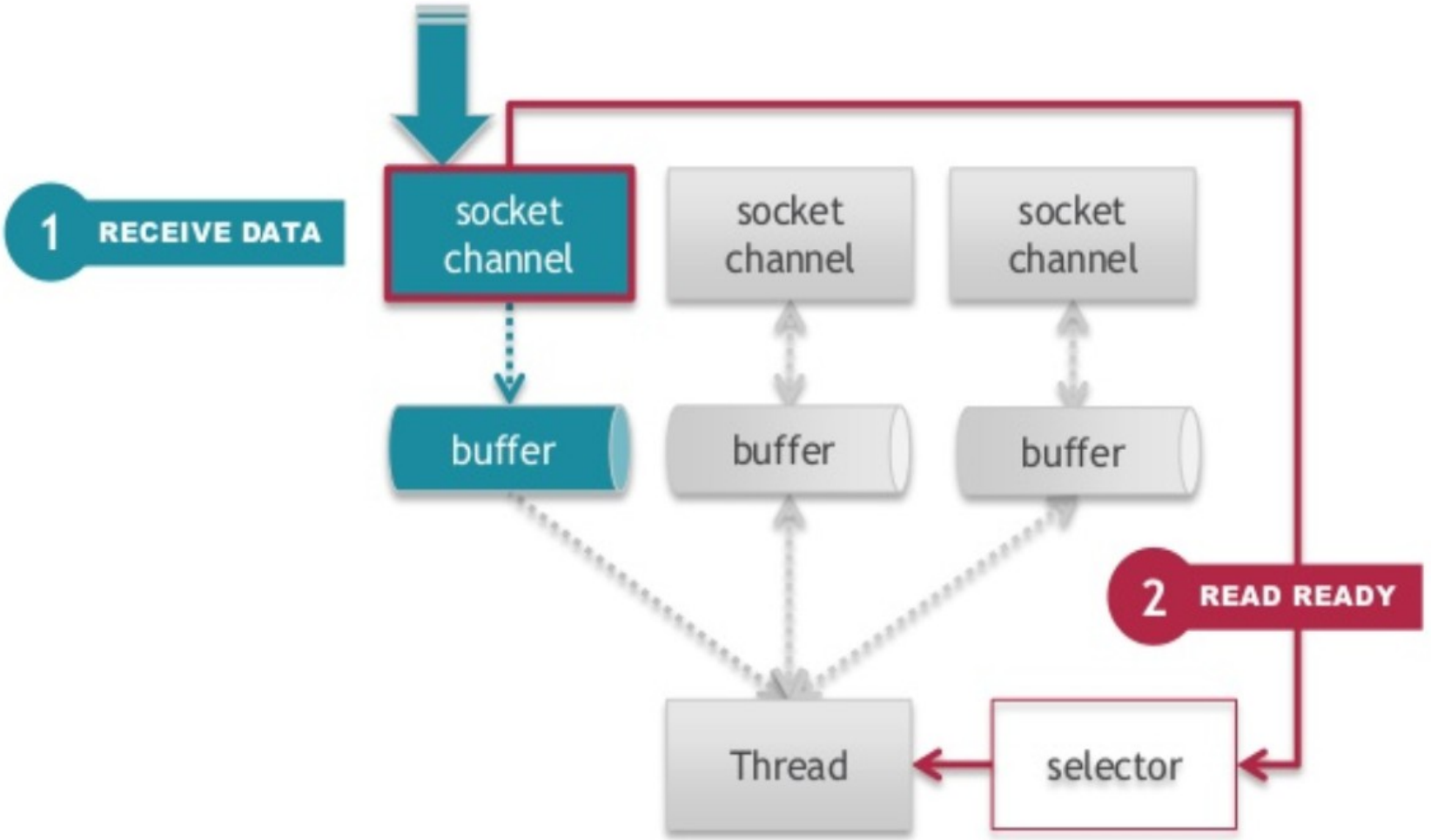
MULTIPLEXING: INVIO DEI DATI



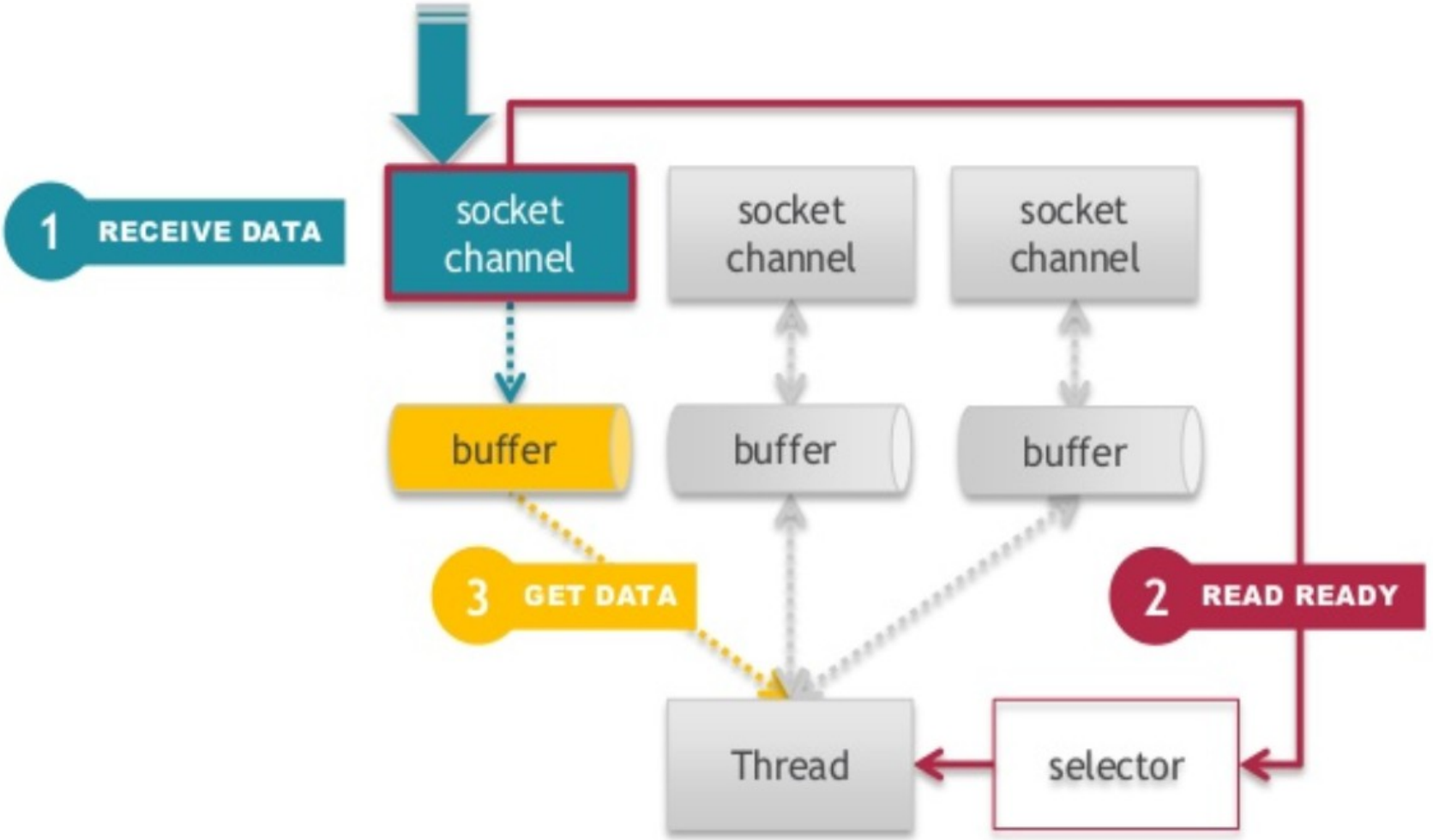
MULTIPLEXING: RICEZIONE DEI DEI DATI



MULTIPLEXING: RICEZIONE DEI DEI DATI



MULTIPLEXING: RICEZIONE DEI DEI DATI



L'OGGETTO SELECTOR

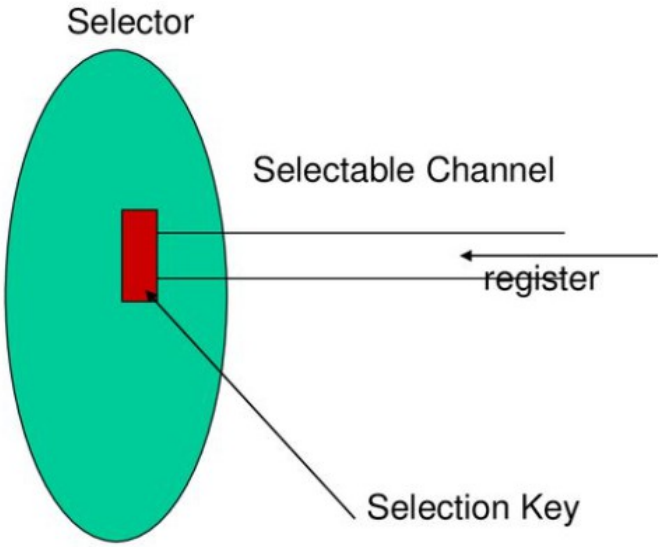
- componente base per permettere il multiplexing
  - readiness selection
- permette di selezionare un `SelectableChannel` che è pronto per operazioni di rete
  - accept, write, read, connect
  - stesso thread che gestisce più eventi che possono avvenire simultaneamente
- selectable channels
  - `ServerSocketChannel`
  - `SocketChannel`
  - `DatagramChannel`
  - `Pipe.SinkChannel`
  - `Pipe.SourceChannels`
  - file I/O non inclusa

REGISTRAZIONE DEI CANALI: SELECTION KEYS

- i canali devono essere registrati su un selettore per operazioni specifiche

```
channel.configureBlocking(false);
Selectionkey key =
    channel.register (selector, ops, attach);
```

- ogni registrazione di un canale su un selettore
  - restituisce una chiave, un "token" che la rappresenta
    - un oggetto di tipo `SelectionKey`.
    - valida fino a che non viene cancellata esplicitamente
- lo stesso canale può essere registrato con più selettori
  - una chiave diversa per ogni registrazione.



L'OGGETTO SELECTION KEY

È il risultato della registrazione di un canale su un selettore e memorizza

- il canale a cui si riferisce
- il selettore a cui si riferisce
- l'interest set
  - definisce le operazioni, del canale associato, su cui si deve fare il controllo di "readiness", la prossima volta che il metodo `select` verrà invocato per monitorare i canali del selettore
- il ready set
  - dopo la invocazione della select, contiene gli eventi che sono pronti su quel canale
    - si può leggere dal canale, si può scrivere, c'è una richiesta di connessione,...
- un allegato, attachment
  - uno spazio di memorizzazione associato a quel canale

L'INTEREST SET COME BITMASK

- rappresentato da una bitmask che codifica le operazioni per cui si registra un interesse su quel canale
- attualmente sono supportati 4 tipi di operazioni
  - connect
  - accept
  - read
  - write
  - non tutte le operazioni valide per tutti i `SelectableChannel`, ad esempio `SocketChannel` non supporta `accept()`
- 4 costanti predefinite nella classe `SelectionKey`, rappresentano le operazioni, ognuna corrisponde ad una bitmask

```
1  SelectionKey.OP_CONNECT
2  SelectionKey.OP_ACCEPT
3  SelectionKey.OP_READ
4  SelectionKey.OP_WRITE
```

L'INTEREST SET COME BITMASK

- Interest Set è manipolato con gli operatori JAVA |, &,^, ~ che eseguono operazioni bit a bit su operandi interi o booleani
- in fase di registrazione del canale con il `Selector` si imposta il valore iniziale dell'Interest Set

```
Selector selector = Selector.open();
channel.register(selector,SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

- l'interest set è reperibile e può essere manipolato tramite gli operatori

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept  =
  (interestSet & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT
```

oppure `selectionkey.interestOps(SelectionKey.OP_READ);`

IL READY SET

- aggiornato quando si esegue una operazione di monitoring dei canali, mediante una `select` (vedi slide successive)
- identifica le chiavi per cui il canale è "pronto", per l'esecuzione
  - sottoinsieme dell'interest set
  - interest set={read, write} ready set={read}
- inizializzato a 0 quando la chiave viene creata
- non può essere modificato direttamente
- restituito dal metodo `readyOps( )` invocato su una `SelectionKey`

```
if ((key.readyOps( ) & SelectionKey.OP_READ) != 0)
     { myBuffer.clear( );
       key.channel( ).read (myBuffer);
       doSomethingWithBuffer (myBuffer.flip( ));}
```

- shortcuts
  - `key.isReadable()` equivale a `key.readyOps( )& SelectionKey.OP_READ) != 0`
  - analogo per le altre operazioni

REGISTRAZIONE CANALI: PATTERN GENERALE

```
// Crea il socket channel e configuralo come non bloccante

ServerSocketChannel server = ServerSocketChannel.open();
server.configureBlocking(false);
server.socket().bind(new java.net.InetSocketAddress(host,8000));
System.out.println("Server attivo porta 2001");

// Crea il selettore e registra il server al Selector

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

L'eventuale allegato

Tipo di registrazione	Significato: il Selector riporta che ...
OP_ACCEPT	Il client richiede una connessione al server
OP_CONNECT	Il server ha accettato la richiesta di connessione
OP_READ	Il channel contiene dati da leggere
OP_WRITE	Il channel contiene dati da scrivere

LA CLASSE SELECTION KEY

```
import java.nio.channels.*;
    public abstract class SelectionKey
       {public static final int OP_READ; public static final int OP_WRITE;
        public static final int OP_CONNECT;Public static final int OP_ACCEPT;
        public abstract SelectableChannel channel( );
        public abstract Selector selector( );
        public abstract void cancel( );
        public abstract boolean isValid( );
        public abstract int interestOps( );
        public abstract void interestOps (int ops);
        public abstract int readyOps( );
        public final boolean isReadable( ) {};
        public final boolean isWritable( ) {};
        public final boolean isConnectable( ) {};
        public final boolean isAcceptable( ) {};
        public final Object attach (Object ob) {};
        public final Object attachment( ) {};}
```

MULTIPLEXING DEI CANALI: LA SELECT

- `int selector.select( );`
  - bloccante, seleziona, tra i canali registrati sul selettore `selector`, quelli pronti per almeno una delle operazioni di I/O dell'interest set.
  - si blocca finchè una delle seguenti condizioni è vera
    - almeno un canale è pronto
    - il thread che esegue la selezione viene interrotto
    - il selettore viene sbloccato mediante il metodo `wakeup()`
  - restituisce il numero di canali pronti
    - che hanno generato un evento dopo l'ultima invocazione della `select()`
    - e costruisce un insieme contenente le chiavi dei canali pronti
- `int select(long timeout)`
  - si blocca fino a che non è trascorso il timeout, oppure valgono le condizioni precedenti
- `int selectNow()`
  - `non bloccante`, nel caso nessun canale sia pronto restituisce il valore 0

ANALISI PROCESSO DI SELEZIONE

ogni oggetto selettore mantiene i seguenti insiemi di chiavi:

- `Key Set`: contiene le `SelectionKeys` dei canali registrati con quel selettore.
  - restituite dal metodo `keys()`
- `Selected Key Set:` dopo che una `select()` consente di accedere ai canali pronti per l'esecuzione di qualche operazione
  - restituiti dal metodo `selectedKeys()`, invocato sul selettore
  - Selected Key Set è l'insieme di chiavi precedentemente registrate e per cui una delle operazioni nell'interest set è anche nel ready set della chiave
- `Cancelled Key Set` contiene la chiavi invalidate, quelle su cui è stato invocato il metodo `cancel()`, ma non ancora de-registrate

COSA FA LA SELECT?

- "delayed cancellation"
  - cancella ogni chiave appartenente al `Cancelled Key Set` dagli altri due insiemi. Cancella così la registrazione del canale
- interagisce con il sistema operativo per verificare lo stato di "readiness" di ogni canale registrato, per ogni operazione specificata nel suo interest set.
- per ogni canale con almeno una operazione "ready"
  - se il canale già esiste nel `Selected Key Set`
    - aggiorna il ready set della chiave corrispondente al canale pronto: calcolo dell'or bit a bit tra il valore precedente del ready set e la nuova maschera
    - i bit ad 1 si "accumulano" con le operazioni pronte.
  - altrimenti
    - resetta il ready set viene ed lo imposta con la chiave della operazione pronta
    - aggiunge il canale al `Selected Key Set`

ANALISI PROCESSO DI SELEZIONE

"comportamento cumulativo" della selezione

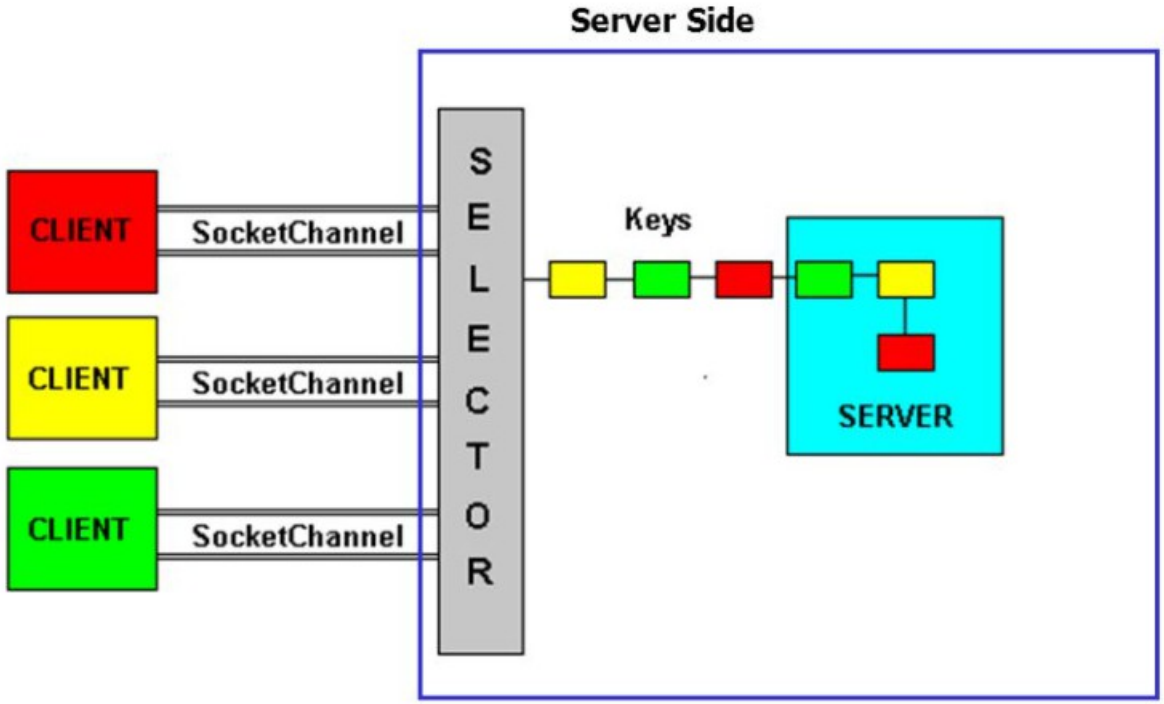
- una chiave aggiunta al `selected key set`, può essere rimossa solo con una operazione di rimozione esplicita
- il `ready set` di una chiave inserita nel `selected key set`, non viene mai resettato, ma viene aggiornato incrementalmente
- scelta di progetto: assegnare al programmatore la responsabilità di aggiornare esplicitamente le chiavi
- per resettare il `ready set`
  - rimuovere la chiave dall'insieme delle chiavi selezionate

SELEZIONE: PATTERN GENERALE

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator <SelectionKey> keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = (SelectionKey) keyIterator.next();
    keyIterator.remove();
    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.
         }
    else if (key.isConnectable()) {
        // a connection was established with a remote server.
        }
        else if (key.isReadable()) {
               // a channel is ready for reading
        }
        else if (key.isWritable()) {
        // a channel is ready for writing }
```

SELEZIONE: PATTERN GENERALE

- iterazione sull'insieme di chiavi che individuano i "canali pronti"
- dalla chiave si può ottenere un riferimento al canale su cui si è verificato l'evento
- `keyIterator.remove()` deve essere invocata, poiché il `Selector` non rimuove le chiavi



SELECTION KEY: L'ATTACHMENT

- attachment: riferimento ad un generico `Object`
- utile quando si vuole accedere ad informazioni relative al canale (associato ad una chiave) che riguardano il suo stato pregresso
- necessario perchè le operazioni di lettura o scrittura non bloccanti non possono essere considerate atomiche: nessuna assunzione sul numero di bytes letti
- consente di tenere traccia di quanto è stato fatto in una operazione precedente. Ad esempio
  - l'attachment può essere utilizzato per accumulare i byte restituiti da una sequenza di letture non bloccanti
  - memorizzare il numero di bytes che si devono leggere in totale.

NIO MULTIPLEXED INTEGER GENERATION SERVICE

- sviluppare un servizio di generazione di una sequenza di interi il cui scopo è testare l'affidabilità della rete, mediante generazione di numeri binari
- quando il server è contattato dal client, esso invia al client una sequenza di interi rappresentati su 4 bytes

  0, 1, 2, ...

- il server genera una sequenza infinita di interi
- il client interrompe la comunicazione quando ha ricevuto sufficiente informazioni

NIO MULTIPLEXED INTEGER GENERATION SERVICE

```
import java.nio.*; import java.nio.channels.*;
import java.net.*; import java.util.*; import java.io.IOException;
public class IntGenServer {
  public static int DEFAULT_PORT = 1919;
  public static void main(String[] args) {
        int port;
        try {
             port = Integer.parseInt(args[0]);
             } catch (RuntimeException ex) {port = DEFAULT_PORT; }

    System.out.println("Listening for connections on port " +
                                                        port);
```

NIO MULTIPLEXED INTEGER GENERATION SERVICE

```
ServerSocketChannel serverChannel;
Selector selector;
   try {
       serverChannel = ServerSocketChannel.open();
       ServerSocket ss = serverChannel.socket();
       InetSocketAddress address = new InetSocketAddress(port);
       ss.bind(address);
       serverChannel.configureBlocking(false);
       selector = Selector.open();
       serverChannel.register(selector, SelectionKey.OP_ACCEPT);
      } catch (IOException ex) {
                ex.printStackTrace();
                return;
                                }
```

NIO MULTIPLEXED INTEGER GENERATION SERVICE

```
while (true) {
    try {
         selector.select();
        } catch (IOException ex) {
                ex.printStackTrace();
                 break;
                                  }
Set <SelectionKey> readyKeys = selector.selectedKeys();
Iterator <SelectionKey> iterator = readyKeys.iterator();
```

NIO MULTIPLEXED INTEGER GENERATION SERVICE

```
while (iterator.hasNext()) {
   SelectionKey key = iterator.next();
   iterator.remove();
   // rimuove la chiave dal Selected Set, ma non dal Registered Set
   try {if (key.isAcceptable()) {
     ServerSocketChannel server = (ServerSocketChannel) key.channel();
     SocketChannel client = server.accept();
     System.out.println("Accepted connection from " + client);
     client.configureBlocking(false);
     SelectionKey key2 = client.register(selector,
                                         SelectionKey.OP_WRITE);
     ByteBuffer output = ByteBuffer.allocate(4);
     output.putInt(0);
     output.flip();
     key2.attach(output); }
```

NIO MULTIPLEXED INTEGER GENERATION SERVICE

```
else if (key.isWritable())
        { SocketChannel client = (SocketChannel) key.channel();
          ByteBuffer output = (ByteBuffer) key.attachment();
          if (! output.hasRemaining())
             {
                 output.rewind();
                 int value = output.getInt();
                 output.clear();
                 output.putInt(value + 1);
                 output.flip();
              }
         client.write(output);}
 } catch (IOException ex) { key.cancel();
                            try { key.channel().close(); }
                            catch (IOException cex) {}      }}}}}
```

NIO INTEGER GENERATION CLIENT

```
import java.nio.*; import java.nio.channels.*;
import java.net.*; import java.io.IOException;
public class IntGenClient {
  public static int DEFAULT_PORT = 1919;
     public static void main(String[] args) {
         if (args.length == 0) {
           System.out.println("Usage: java IntgenClient host [port]");
           return;   }
          int port;
          try {
               port = Integer.parseInt(args[1]);
              } catch (RuntimeException ex) {
                      port = DEFAULT_PORT;
                                          }
```

NIO INTEGER GENERATION CLIENT

```
try { SocketAddress address = new InetSocketAddress(args[0], port);
      SocketChannel client  = SocketChannel.open(address);
      ByteBuffer    buffer  = ByteBuffer.allocate(4);
      IntBuffer     view    = buffer.asIntBuffer();
      for (int expected = 0;  ; expected++) {
        client.read(buffer);
        int actual = view.get();
        buffer.clear();
        view.rewind();
        if (actual != expected) {
          System.err.println("Expected " + expected + "; was " + actual);
          break;
        }
        System.out.println(actual);
      }
    } catch(IOException ex) { ex.printStackTrace(); } }  }
```

ESERCIZIO DI PREPARAZIONE ALL'ASSIGMENT

- scrivere un programma JAVA che implementi un server che apre una listening socket su una porta e resta in attesa di richieste di connessione.
- quando arriva una richiesta di connessione, il server accetta la connessione, trasferisce al client un messaggio ("HelloClient") e poi chiude la connessione.
- usare canali non bloccanti e il selettore (e i buffer di tipo `ByteBuffer`).
- lato client è possibile utilizzare telnet

Opzione più semplice

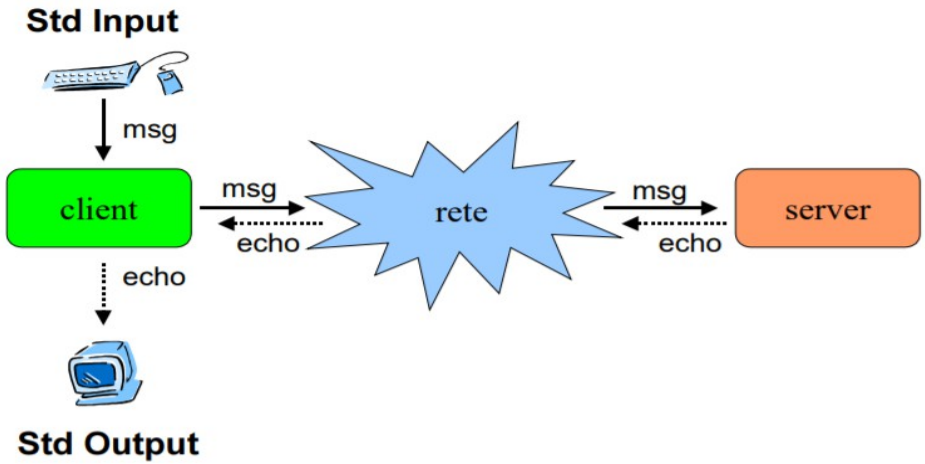
- come primo esercizio potete sviluppare un programma in cui dopo il controllo `key.isAcceptable` il server scrive subito sulla socketChannel restituita dall'operazione di accept e chiude la connessione

Opzione più completa (ma un po' più complicata - vedi esempio IntGenServer sulle slide)

- dopo il controllo key.isAcceptable la socketChannel restituita dall'operazione di accept viene registrata sul selettore (con interesse all'operazione di WRITE) e il messaggio viene inviato quando il canale è pronto per la scrittura (`key.isWritable` è true)

ASSIGNMENT 7: NIO ECHO SERVER

- scrivere un programma echo server usando la libreria `java NIO` e, in particolare, il `Selector` e canali in modalità non bloccante, e un programma echo client, usando `NIO` (va bene anche con modalità bloccante).
- Il server accetta richieste di connessioni dai client, riceve messaggi inviati dai client e li rispedisce (eventualmente aggiungendo "echoed by server" al messaggio ricevuto).
- Il client legge il messaggio da inviare da console, lo invia al server e visualizza quanto ricevuto dal server.



QUANDO NIO E' UTILE?

- esperienza di tirocinio: un client light weight per l'analisi della topologia della rete P2P di Bitcoin
- altissimo numero di connessioni gestite al passare del tempo, uno snapshot:

```
Nuove Connessioni: 68
Connessioni cadute: 3
Nuove Connessioni: 594
Connessioni cadute: 35
Nuove Connessioni: 566
Connessioni cadute: 118
Nuove Connessioni: 675
Connessioni cadute: 230
Nuove Connessioni: 648
Connessioni cadute: 250
Nuove Connessioni: 502
Connessioni cadute: 252
Nuove Connessioni: 418
Connessioni cadute: 320 …...
```