Reti e Laboratorio III
Modulo Laboratorio III
AA. 2025-2026

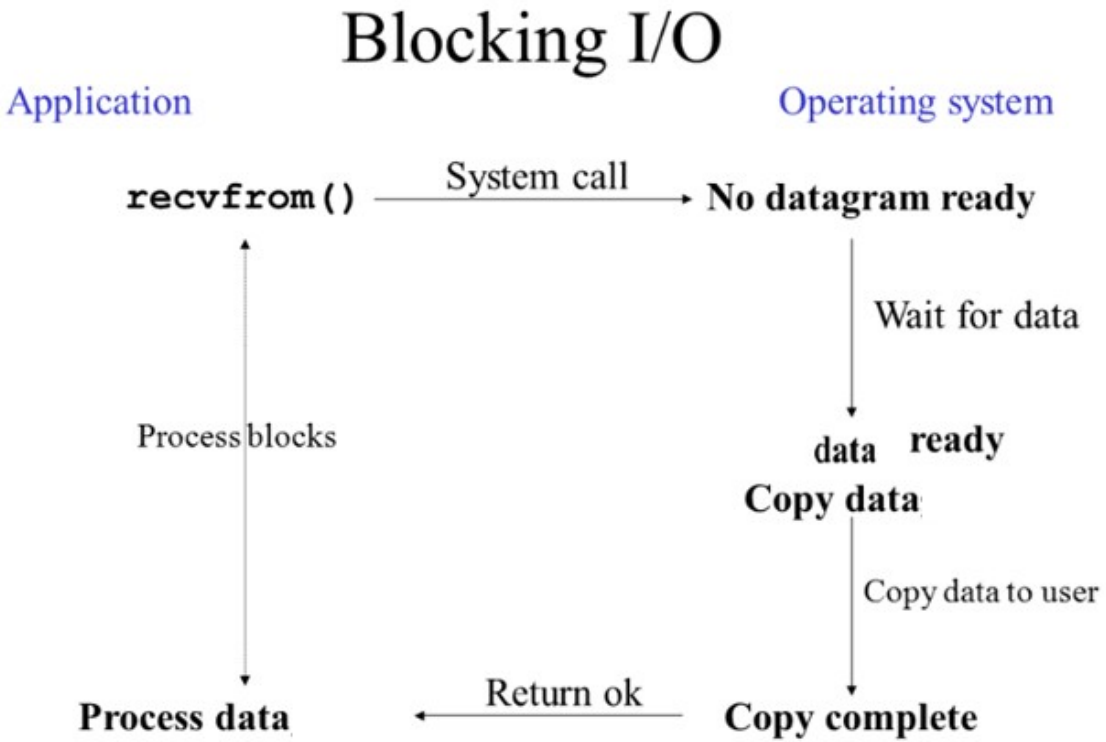
docente: Laura Ricci

laura.ricci@unipi.it

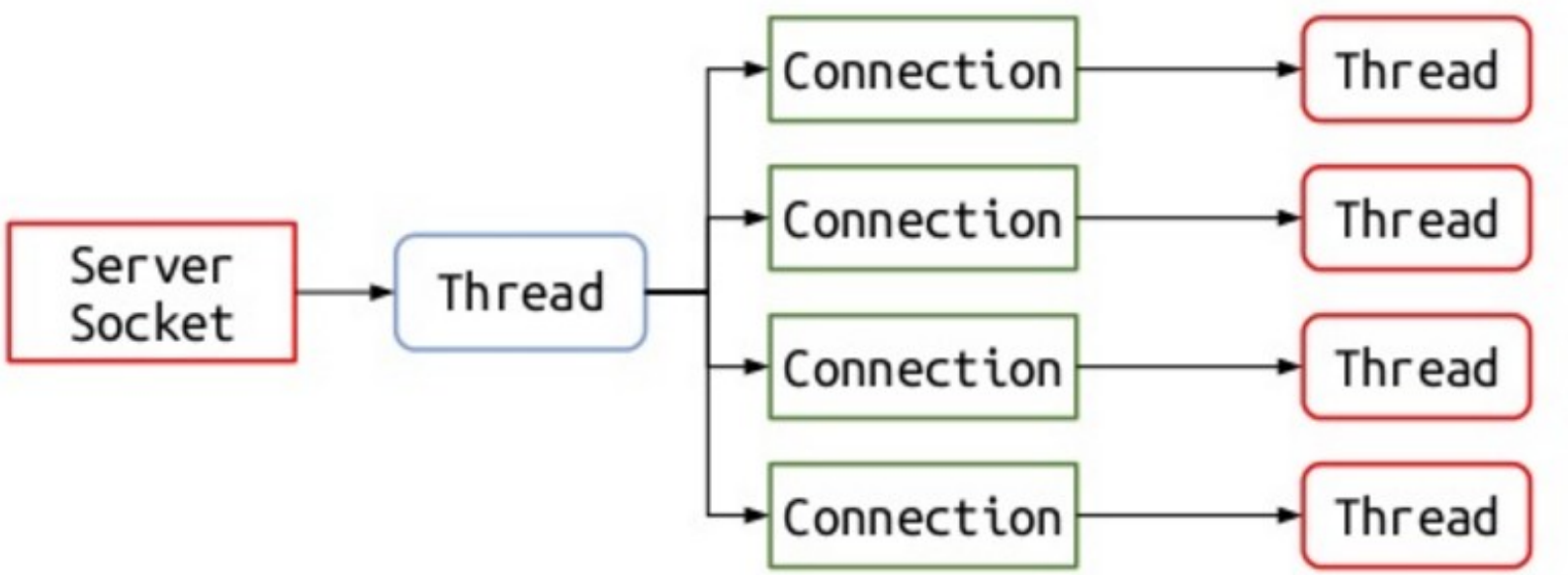
Lezione 9
JAVA NIO:
CHANNEL MULTIPLEXING

14/11/2025

BLOCKING IO: RECAP

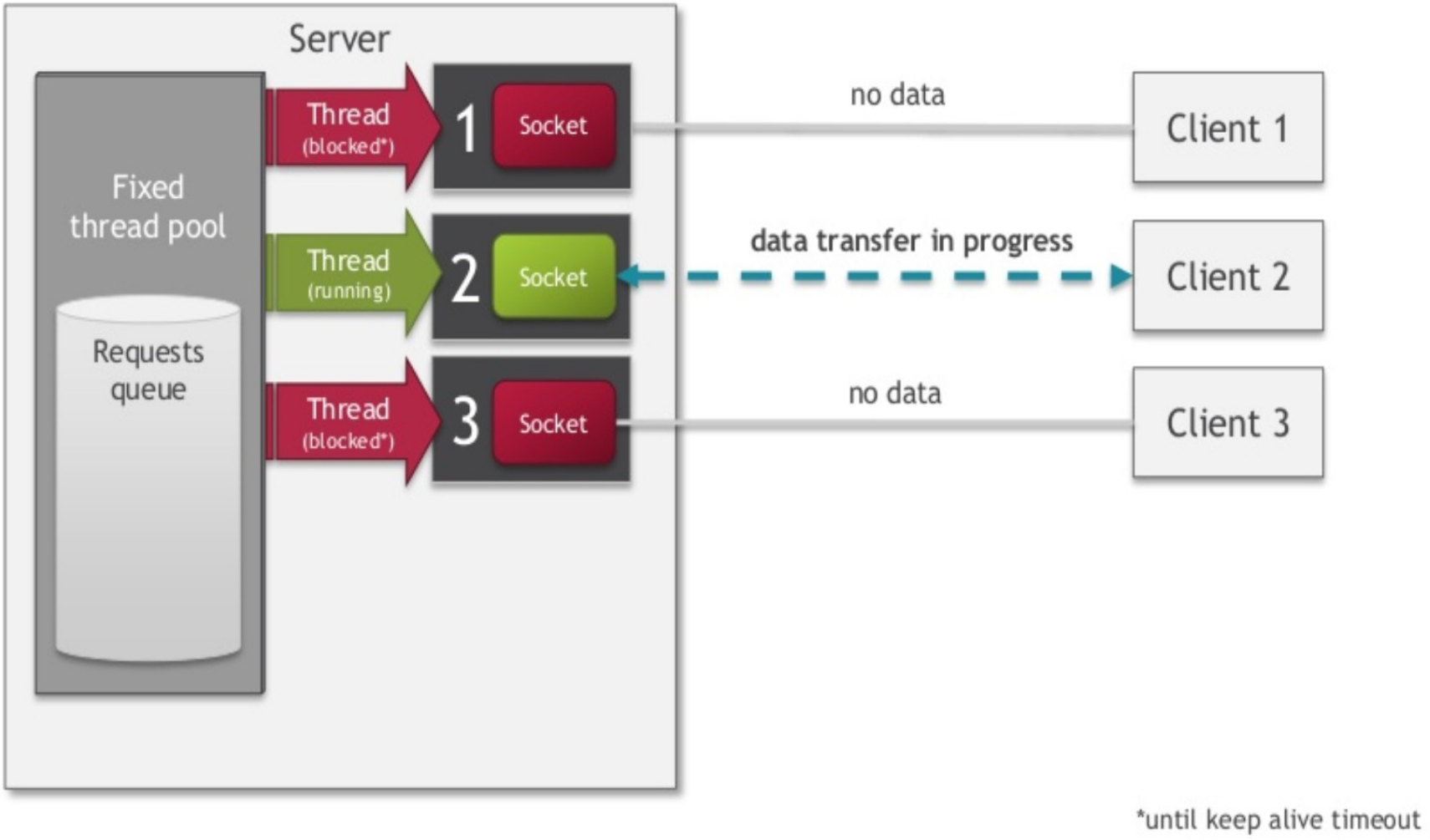


MULTITHREADED SERVER CON BLOCKING IO

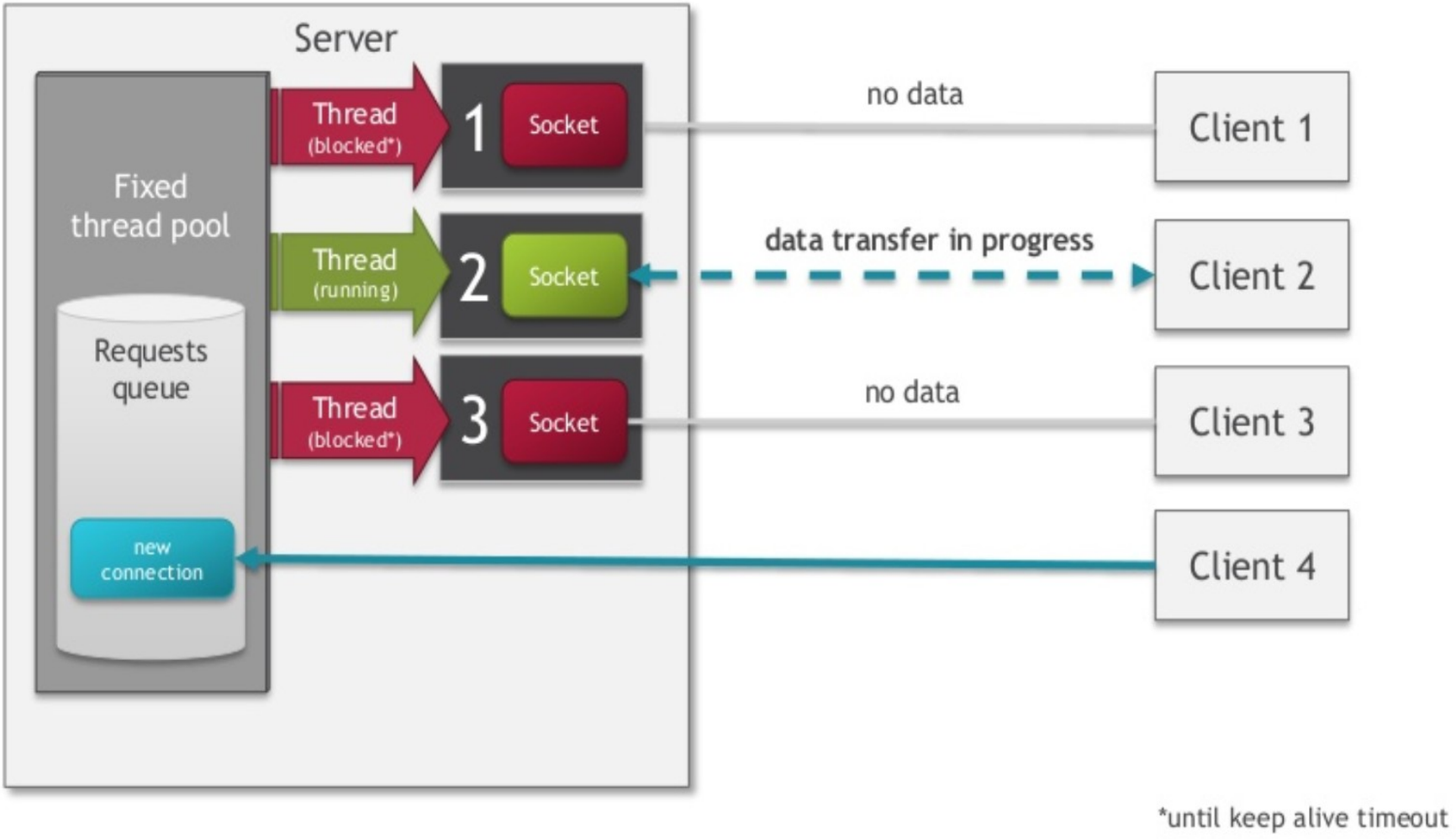


- attivazione di un thread per ogni connessione
- quando un server monitora un grande numero di comunicazioni:
  - problemi di scalabilità: il tempo per il cambio di contesto può aumentare notevolmente con il numero di thread attivi
  - maggior parte del tempo impiegata in context switching

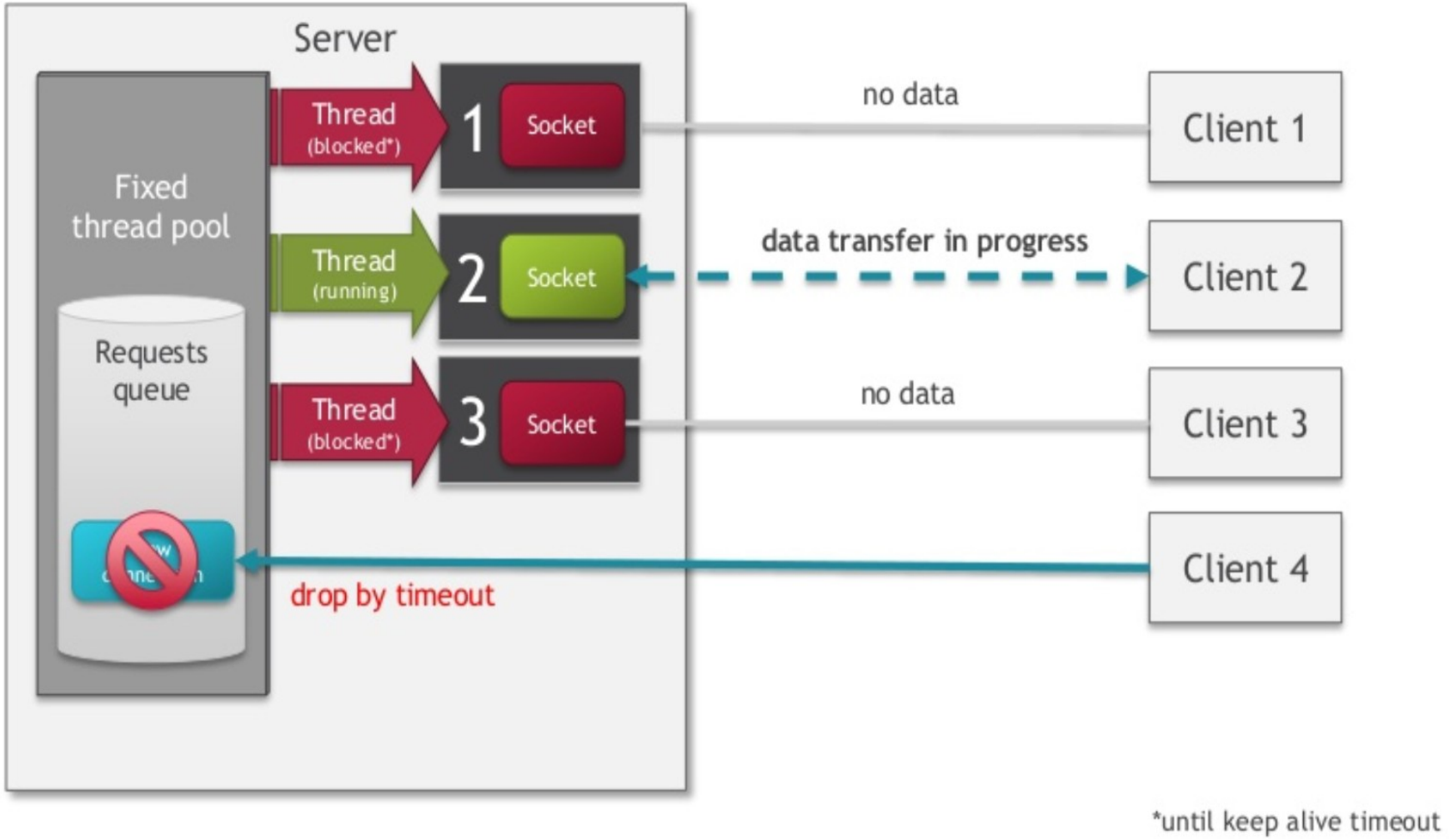
MULTITHREADED SERVER CON BLOCKING IO



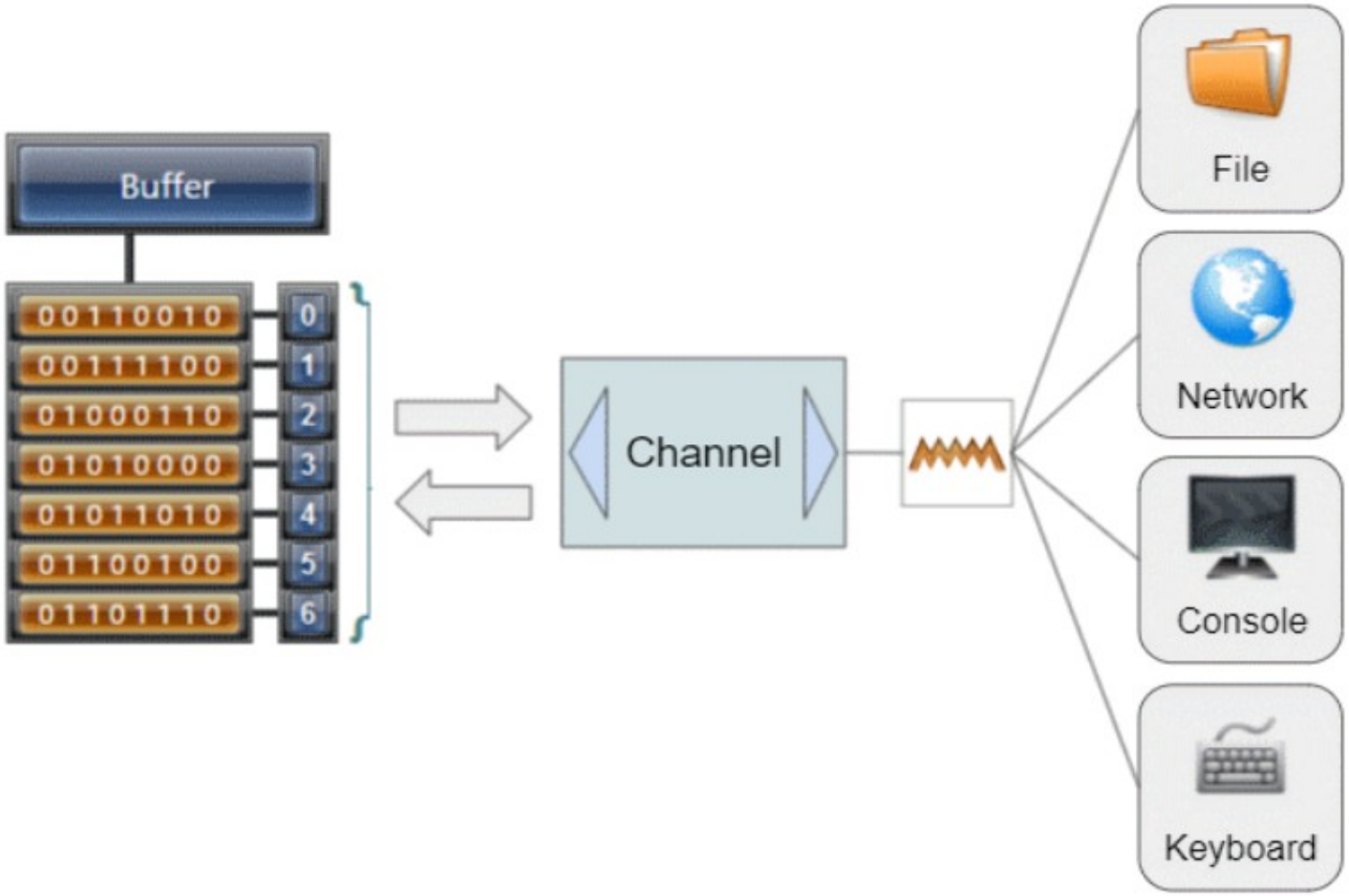
MULTITHREADED SERVER CON BLOCKING IO



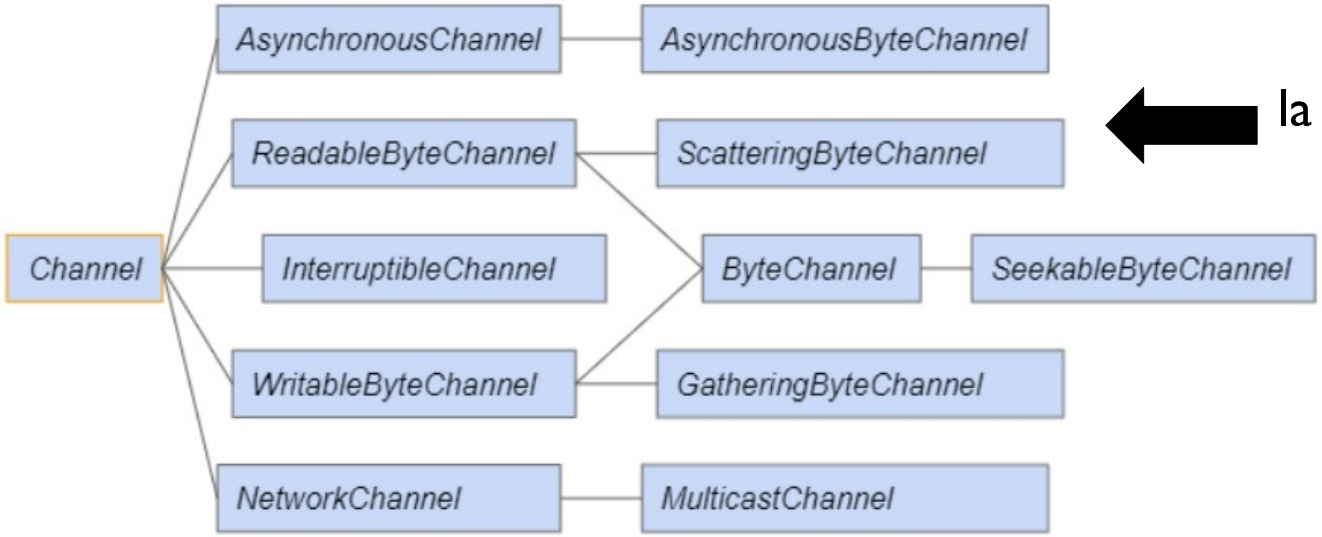
MULTITHREADED SERVER CON BLOCKING IO



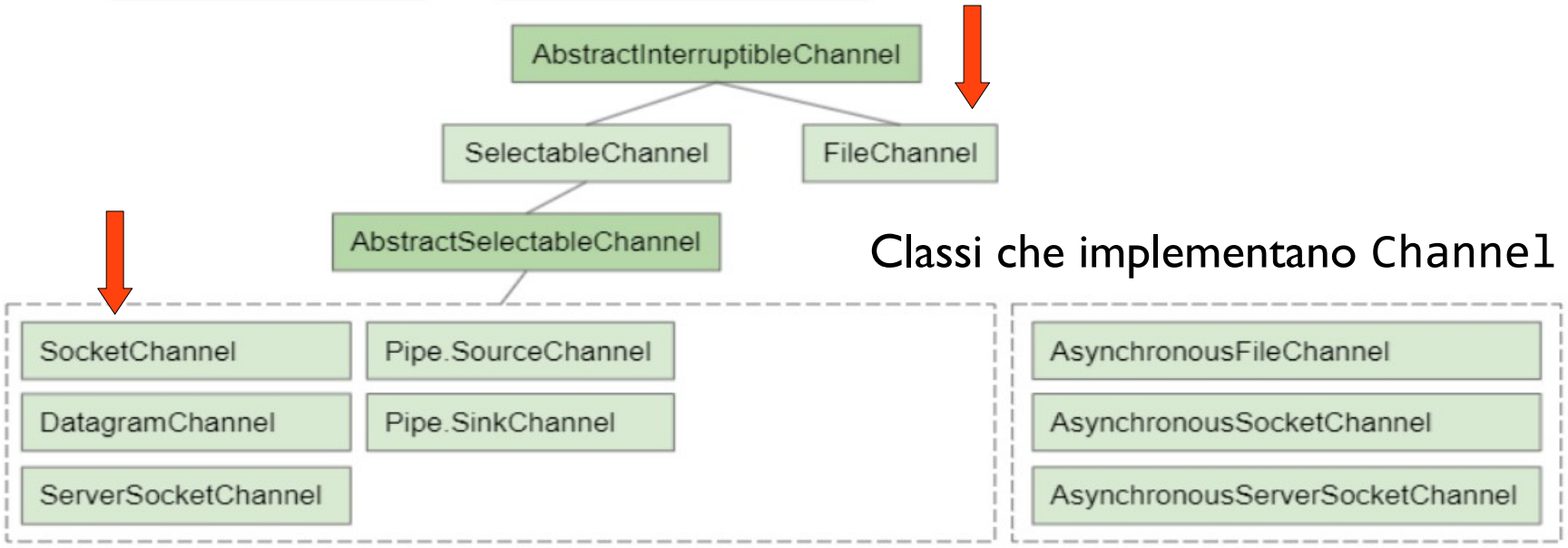
CHANNEL RECAP



JAVA NIO.CHANNEL



la gerarchia delle interfacce



Classi che implementano Channel

SOCKET CHANNELS BLOCCANTI

```
import java.net.InetSocketAddress;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.ByteBuffer;
import java.io.*;

public class NonBlockingServerApplication {
    public static void main(String[] args) throws IOException {
        ServerSocketChannel serverSocket = ServerSocketChannel.open();
        serverSocket.bind(new InetSocketAddress(8080));
        while (true) {
            SocketChannel socket = serverSocket.accept();
            handleRequest(socket);
            socket.close();
        }
    }
```

SOCKET CHANNELS BLOCCANTI

```
private static void handleRequest(SocketChannel socket) {
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(80);
        try {
            while (socket.read(byteBuffer) != -1) {
                byteBuffer.flip();
                toUpperCase(byteBuffer);
                Thread.sleep(3000);
                while (byteBuffer.hasRemaining()) {
                    socket.write(byteBuffer);
                }
                byteBuffer.clear();
            }
        }
          catch(Exception e) { e.printStackTrace();}
    }
```

SOCKET CHANNELS BLOCCANTI

```
private static void toUpperCase(final ByteBuffer byteBuffer) {
        for (int x = 0; x < byteBuffer.limit(); x++) {
            byteBuffer.put(x, (byte) toUpperCase(byteBuffer.get(x)));
        }
    }

    private static int toUpperCase(int data) {
        return
          Character.isLetter(data) ? Character.toUpperCase(data) : data;
    }
}
```

SOCKET CHANNEL BLOCCANTI

- un channel associato ad un socket TCP "combina" un socket con un 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 funzionamento simile a quello degli stream socket, ma con interfaccia basata su buffers
- classi SocketChannel, ServerSocketChannel
- ognuno di essi associato ad un oggetto Socket della libreria java.net
  - il socket può essere reperito mediante il metodo socket(), applicato al channel

SOCKET CHANNELS BLOCCANTI

- canali associati ad un oggetto di tipo `Socket`
- 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`
- in `handleRequest`, quando il client chiude la connessione TCP, restituisce il valore `-1`
  - `-1` `end of stream`
  - il canale non è chiuso localmente, ma la connessione è chiusa

SOCKET CHANNELS BLOCCANTI

- il programma precedente accetta una sola connessione per volta e tutte le operazioni avvengono esclusivamente nel thread principale
- la differenza con un server che gestisce stream è che utilizza un `ByteBuffer` e il `ServerSocketChannel` di `NIO` per ricevere le connessioni.
- vediamo cosa accade se utilizziamo connessioni non bloccanti

JAVA NIO: OBIETTIVI

- fast buffered binary e character `I/O`

  "*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*"

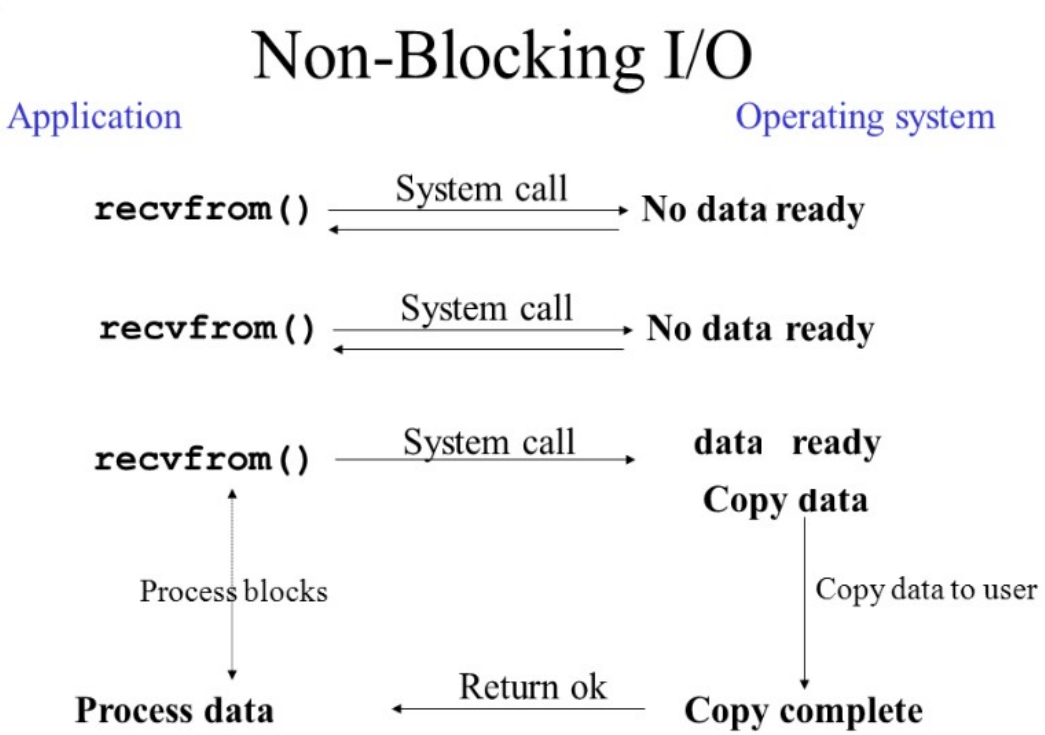
- "non blocking mode" e multiplexing

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

di seguito:

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

NON BLOCKING SOCKET CHANNELS



- modalità non blocking:

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

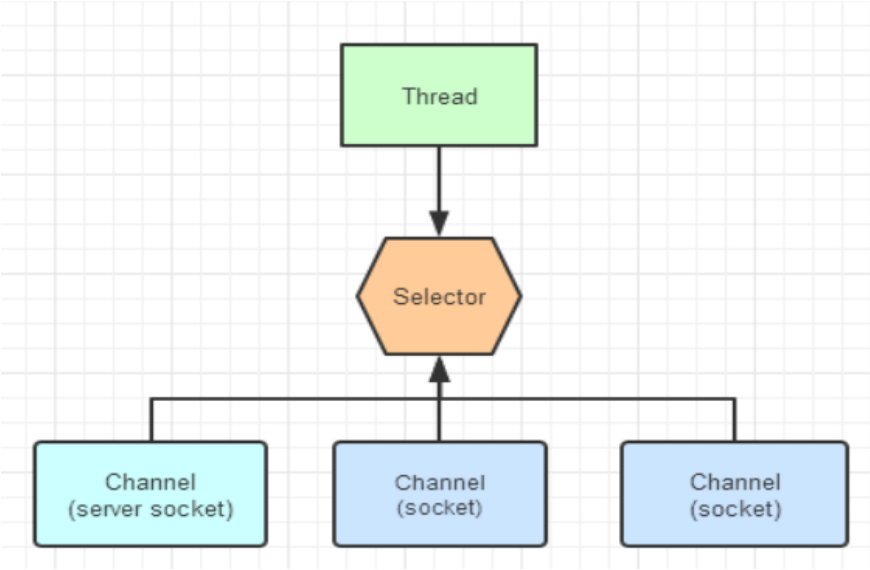
- utilizzabile sia lato client che lato server

MODALITA' NON BLOCKING: POLLING SERVER

```
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 {}
    }
```

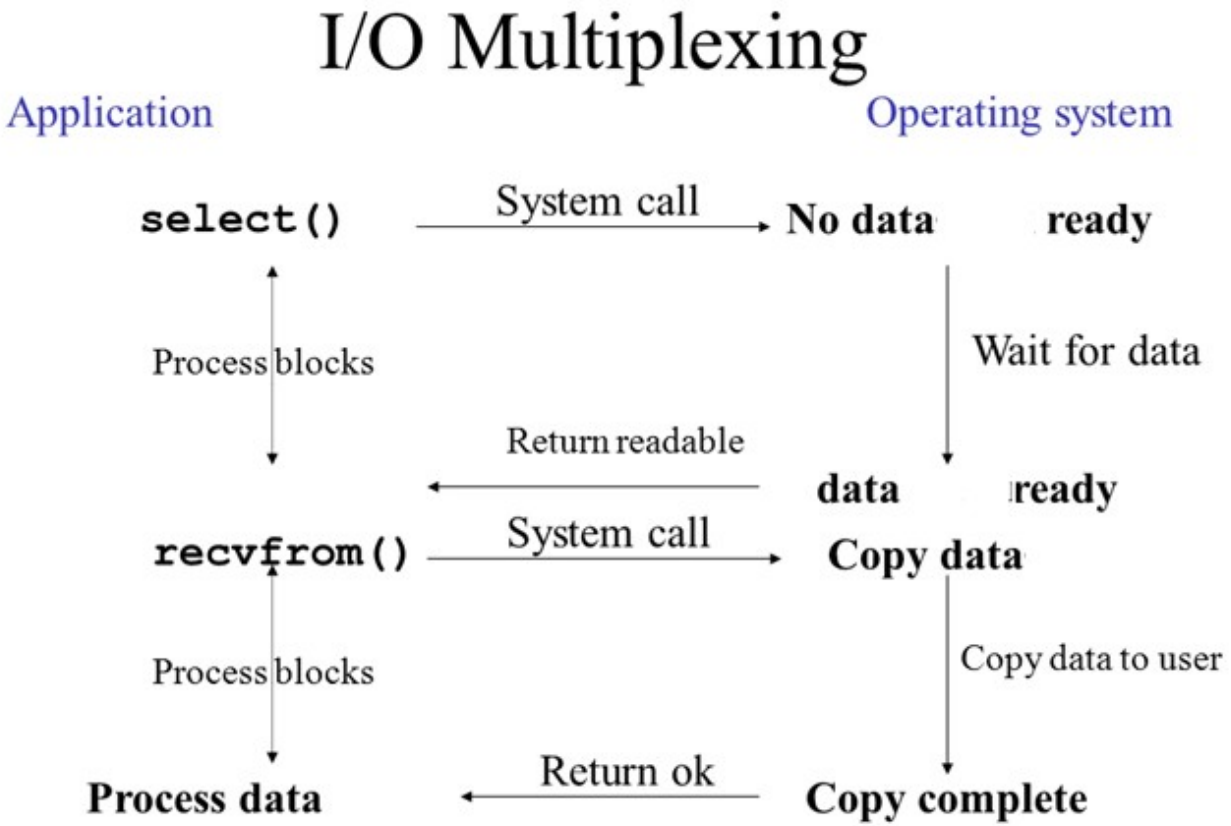
- permette semplice multiplexing dei canali configurati in modalità non bloccante, mediante polling
- inutile consumo di CPU, continue system call di I/O verso il kernel per verificare continuamente cosa sia accaduto su un determinato socket
- se lo schema è applicato a tutti i channel che collegano il server agli n client il costo cresce linearmente

MODALITA' NON BLOCCATE: MULTIPLEXING

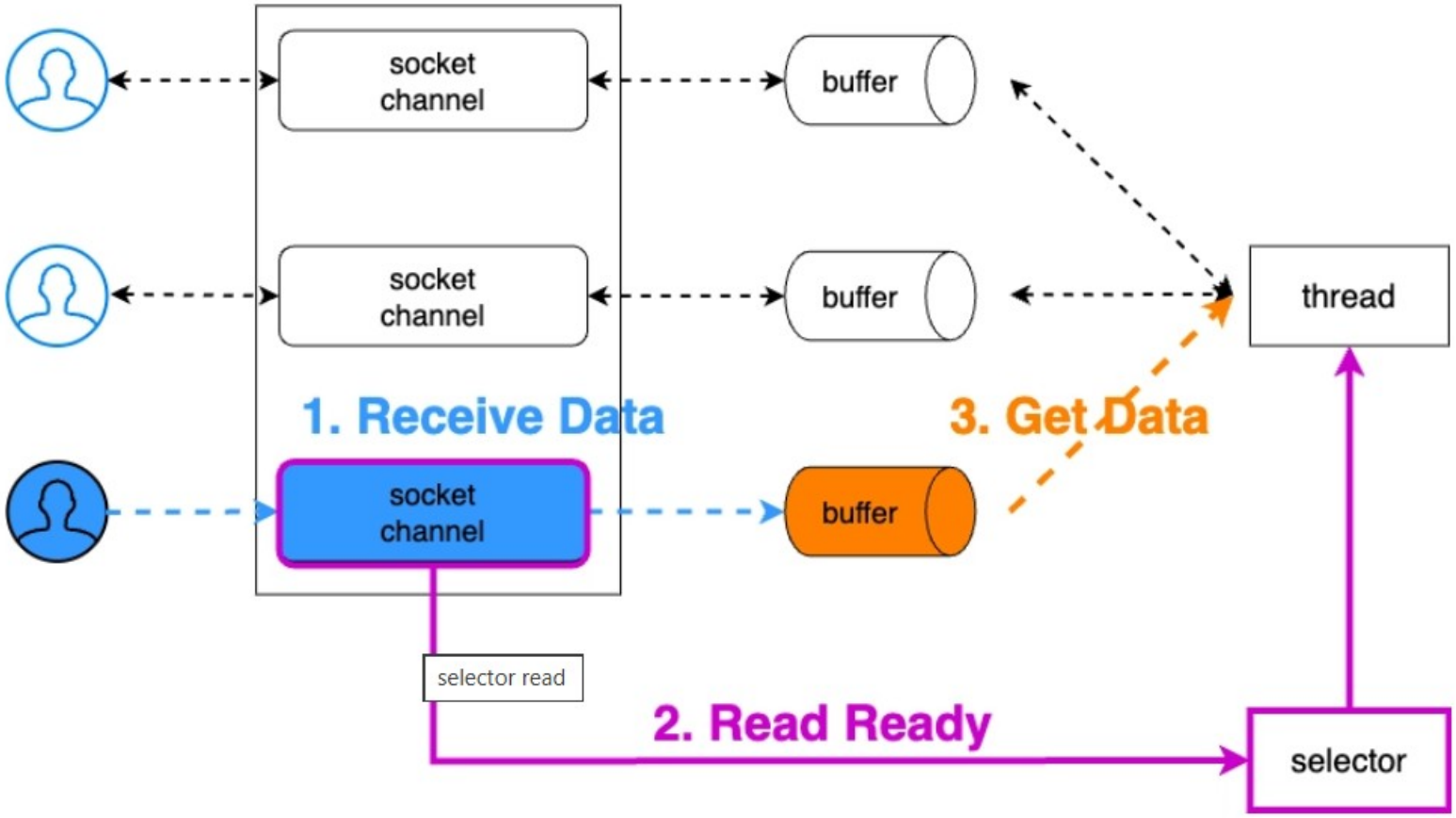


- selettore un componente che esamina uno o più `NIO Channels`, definiti in modalità non bloccante, e determina quali sono pronti per leggere/scrivere
- più connessioni di rete gestite mediante un unico thread, consente di ridurre
  - thread switching overhead
  - uso di risorse per thread diversi
-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

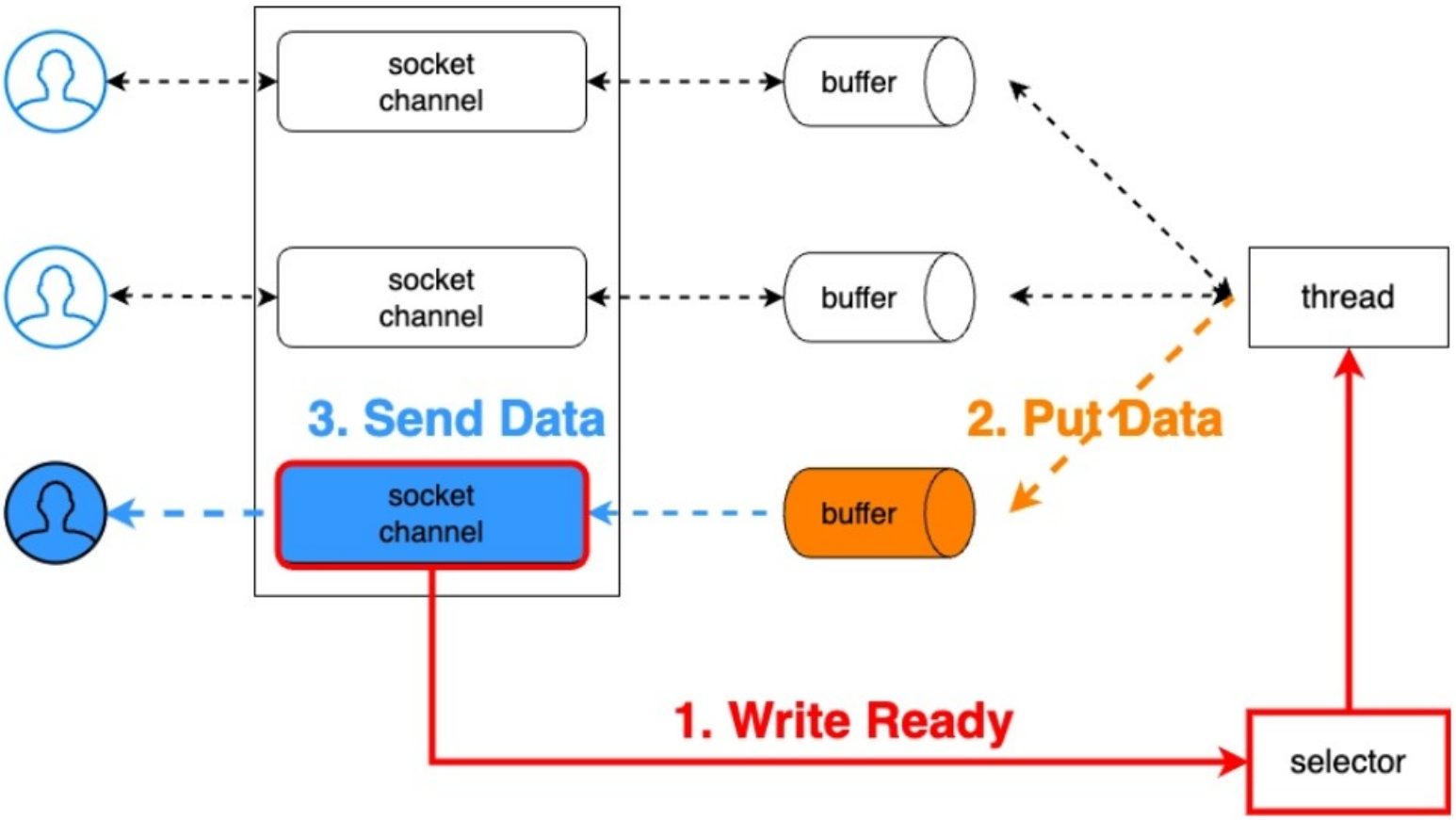
MODALITA' NON BLOCCATE: MULTIPLEXING



IO MULTIPLEXING CON SELECTOR



IO MULTIPLEXING CON SELECTOR



L'OGGETTO SELECTOR

- componente base per il multiplexing

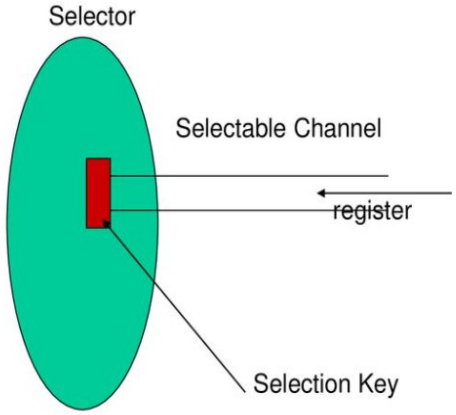
```
Selector selector = Selector.open();
```

- permette di selezionare un `SelectableChannel` che è pronto per operazioni di rete
  - `connect`, `write`, `read`, `accept`
  - stesso thread che gestisce più eventi che possono avvenire simultaneamente
- selectable channels
  - `ServerSocketChannel`
  - `SocketChannel`
  - `DatagramChannel`
  - `Pipe.SinkChannel`
  - `Pipe.SourceChannels`
  - file non inclusi

REGISTRAZIONE DEI CANALI: SELECTION KEYS

- registrazione di un canale su un selettore

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



  - il canale deve essere in modalità non bloccante
  - non si possono usare `Filechannels` con i `Selector`

- secondo parametro della `register (ops)` è l' "`interest set`"
  - indica quali eventi si è interessati a monitorare su quel canale

```
SelectionKey key = channel.register(selector,SelectionKey.OP_READ);
```

- terzo parametro della `register (attach)` è un `Object` associato al canale, che rappresenta uno spazio di buffferizzazione

L'INTEREST SET COME BITMASK

- `bitmask di 8 bit` (un intero) codifica le operazioni di interesse su quel canale
- attualmente sono supportati 4 tipi di operazioni, ad ogni operazione corrisponde una bitmask
  - `read`
  - `write`
  - `accept`

OP_READ - 1	0 0 0 0 0 0 0 1
OP_WRITE - 4	0 0 0 0 0 1 0 0
OP_ACCEPT - 16	0 0 0 1 0 0 0 0
OP_READ | OP_WRITE - 5	0 0 0 0 0 1 0 1

- è manipolato con gli operatori JAVA `|, &,^, ~,` che eseguono operazioni bit a bit su operandi interi o booleani
- non tutte le operazioni valide per tutti i `SelectableChannel`, ad esempio `SocketChannel` non supporta `accept()`

L'INTEREST SET COME BITMASK

- nella classe `SelectionKey`, 4 costanti predefinite che corrispondono alle bitmask predefinite

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

- utilizzabili in fase di registrazione del canale con il `Selector` per impostare il valore iniziale dell'`Interest Set`

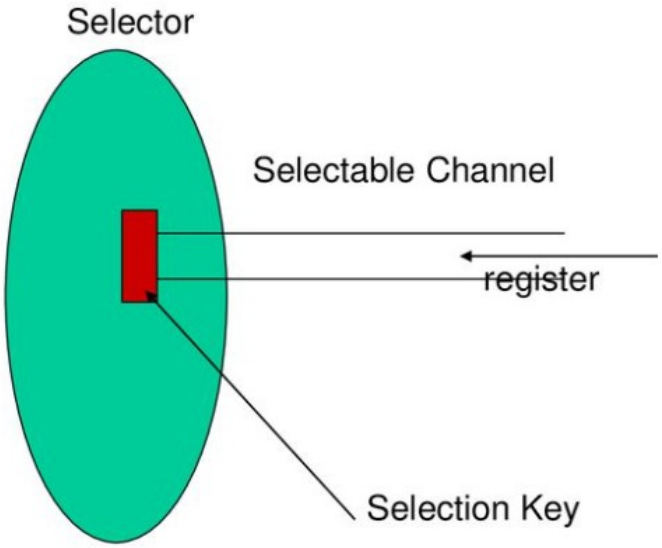
```
Selector selector = Selector.open();
channel.register(selector,SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

- per reperire l'interest set

```
int interestSet = selectionKey.interestOps();
```

REGISTRAZIONE DEI CANALI: SELECTION KEYS

- 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.



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 8000);

// 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

MULTIPLEXING DEI CANALI: LA SELECT

- `int selector.select( );`
  - bloccante, seleziona, tra i canali registrati sul selettore `selector`, quelli pronti per almeno una delle operazioni dell'`interest set`.
  - si blocca fino a che 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()`
    - costruisce un insieme contenente le `SelectionKeys` dei canali pronti
- `int select(long timeout)`
  - si blocca fino a che non è trascorso il timeout, oppure vale una delle condizioni precedenti
- `int selectNow()`
  - non `bloccante`, nel caso nessun canale sia pronto restituisce il valore 0

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
  - utilizzato quando il metodo `select` viene invocato per monitorare i canali del selettore
  - definisce le operazioni su cui si deve fare il controllo di "readiness",
- il ready set
  - dopo la invocazione della `select`, contiene gli eventi che sono pronti su quel canale
- un allegato, attachment
  - uno spazio di memorizzazione associato a quel canale per quel selettore

IL READY SET

- aggiornato quando si esegue una operazione di monitoring dei canali, mediante una `select`
- 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
- operazioni su bitmask per verificare se si è verificato un evento

$\forall 1 \leq i \leq 8 \; e_i \in \{0,1\}$ e

e_8	e_7	e_6	e_5	e_4	e_3	e_2	e_1
				&			
0	0	0	0	0	0	0	1

OP_READ

Different than 0 iff $e_1 = 1$

0	0	0	0	0	0	0	e_1

IL READY SET

- restituito dal metodo `readyOps( )` invocato su una `SelectionKey`
- supponiamo `key` sia una `SelectionKey`, per testare se ci sono dati pronti per essere letti

```
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`
  - analoghi shortcuts per le altre operazioni

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( ) {};}
```

ANALISI PROCESSO DI SELEZIONE

ogni oggetto selettore mantiene, al suo interno, i seguenti insiemi di chiavi:

- **`Key Set`**:
  - `SelectionKeys` dei canali registrati con quel selettore.
  - restituito dal metodo **`keys()`**
- **`Selected Key Set`**
  - restituito dal metodo **`selectedKeys()`**, invocato sul selettore
  - insieme di chiavi precedentemente registrate tali per cui una delle operazioni nell'`interest set` è pronta per l'esecuzione
  - dopo una `select()` consente di accedere ai canali pronti per l'esecuzione di qualche operazione
- **`Cancelled Key Set`**
  - chiavi che sono state cancellate (quelle su cui è stato invocato il metodo **`cancel()`**, ma il cui canale è ancora registrato sul selettore)

COSA FA LA SELECT?

1. "delayed cancellation": cancella ogni chiave appartenente al `Cancelled Key Set` dagli altri due insiemi.
2. interagisce con il sistema operativo per verificare lo stato di "readiness" di ogni canale registrato, per ogni operazione specificata nel suo interest set.
3. per ogni canale con almeno una operazione "ready"
   - se il canale già esiste nel `Selected Key Set`
     - aggiorna il ready set della chiave corrispondente: calcola l'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 e lo imposta con la chiave della operazione pronta
     - aggiunge la chiave al `Selected Key Set`

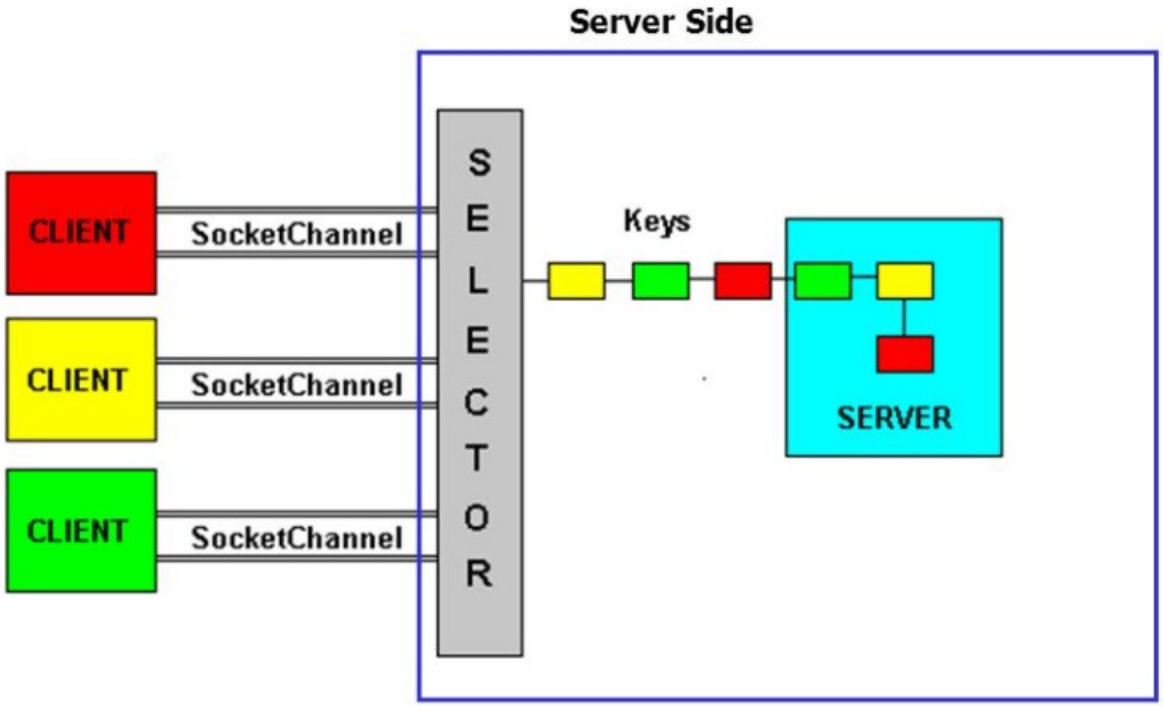
ANALISI PROCESSO DI SELEZIONE

"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

- 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



SELEZIONE: PATTERN GENERALE

```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
while(true) {
    int readyChannels = selector.selectNow();
    if(readyChannels == 0) continue;
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
    while(keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();
       if(key.isAcceptable()) { // a connection was accepted by a ServerSocketChannel.
      } else if (key.isConnectable()) { // a connection was established with a remote
                                        Server (client side)
            } else if (key.isReadable()) { // a channel is ready for reading
       } else if (key.isWritable()) { // a channel is ready for writing }
     keyIterator.remove();
      } }
```

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.
  - 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 sufficienti 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

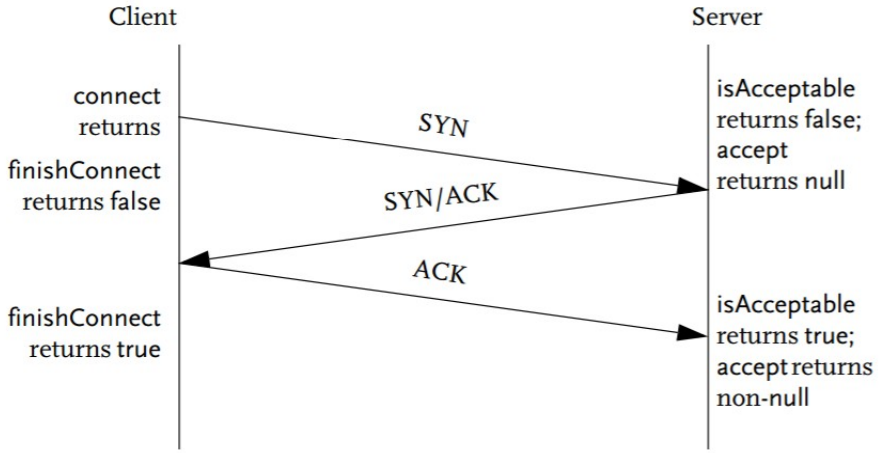
```
while (iterator.hasNext()) {
   SelectionKey key = iterator.next();
   iterator.remove();
   // rimuove la chiave dal Selected Set, ma non dal Registered Set
   try {if (key.isAcceptable()) {
     ServerSocketChannel server = (ServerSocketChannel) key.channel();
     SocketChannel client = server.accept();
     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) {}      }}}}}
```

ALTRI METODI PER NIO

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



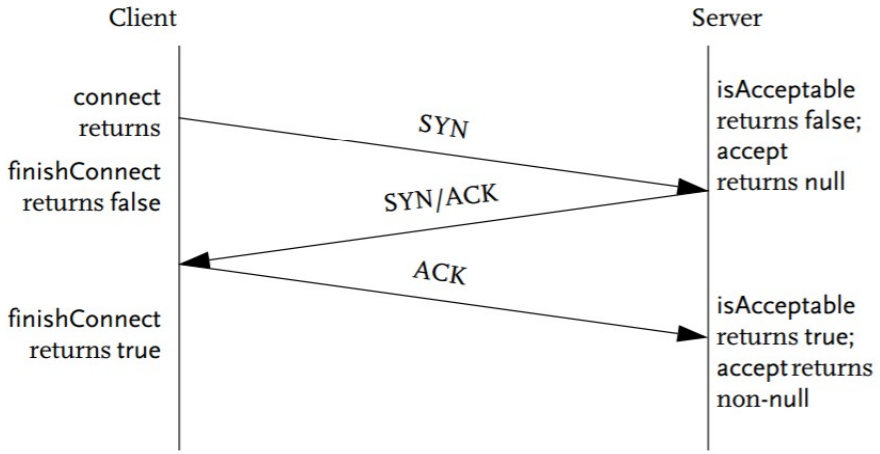
`isAcceptable`
restituisce vero quando la connessione può essere accettata

- `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...    }
```

ALTRI METODI PER NIO

- 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`

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.