TCP Socket con NIO

16 novembre 2021

JAVA NIO: OBIETTIVI

- fast buffered binary e character I/O (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
- Selector: multiplexing

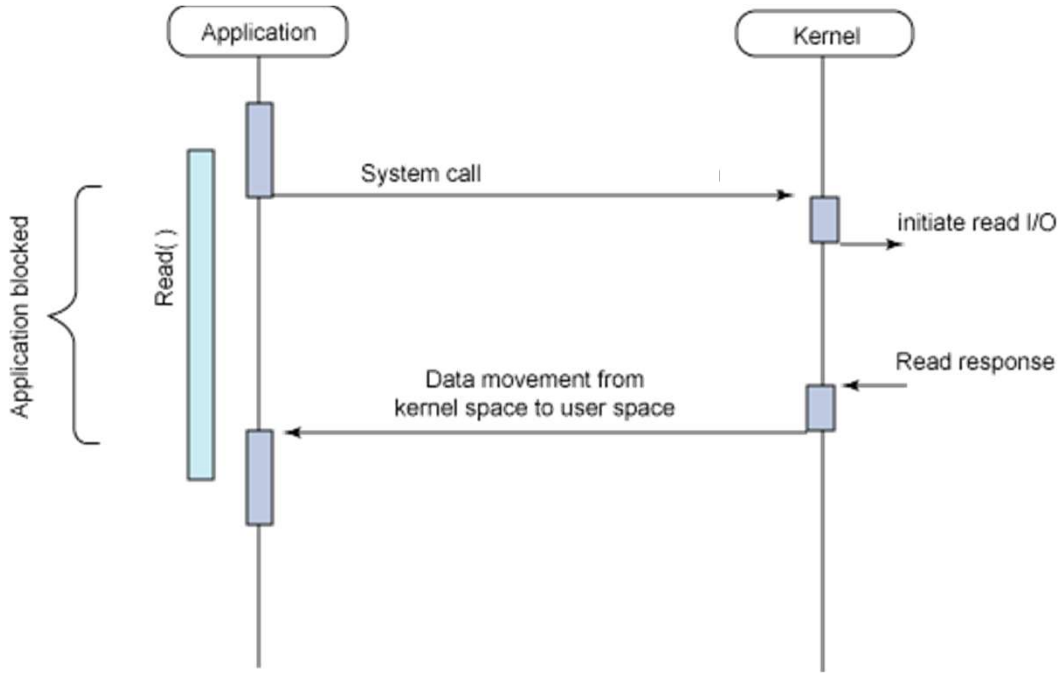
Contesto

- La rete è lenta rispetto alla CPU e alla memoria
- JAVA I/O cerca di impiegare evitare che la CPU stia in attesa della rete con meccanismi di:
  - Buffering
  - Multithreading
  - Più thread possono generare dati e i dati possono essere memorizzati finché non possono essere inviati in rete
    - Context switches
    - Memory footprint
- Principio dell'approccio NIO (modalità non bloccante + selector) per l'architettura di programmi server
  - invece di assegnare un thread per connessione, un solo thread gestisce più connessioni
  - se una connessione è pronta per inviare i dati, le passa i dati da inviare e poi si sposta alla connessione successiva
  - Supporto del sistema operativo sottostante
- NIO supporta sia la modalità bloccante che la modalità non bloccante

BLOCKING IO

Operazioni bloccanti **su stream:**

- **read():** il thread si blocca fino a quando non è stato letto un byte (un vettore di byte, un intero, ecc..)



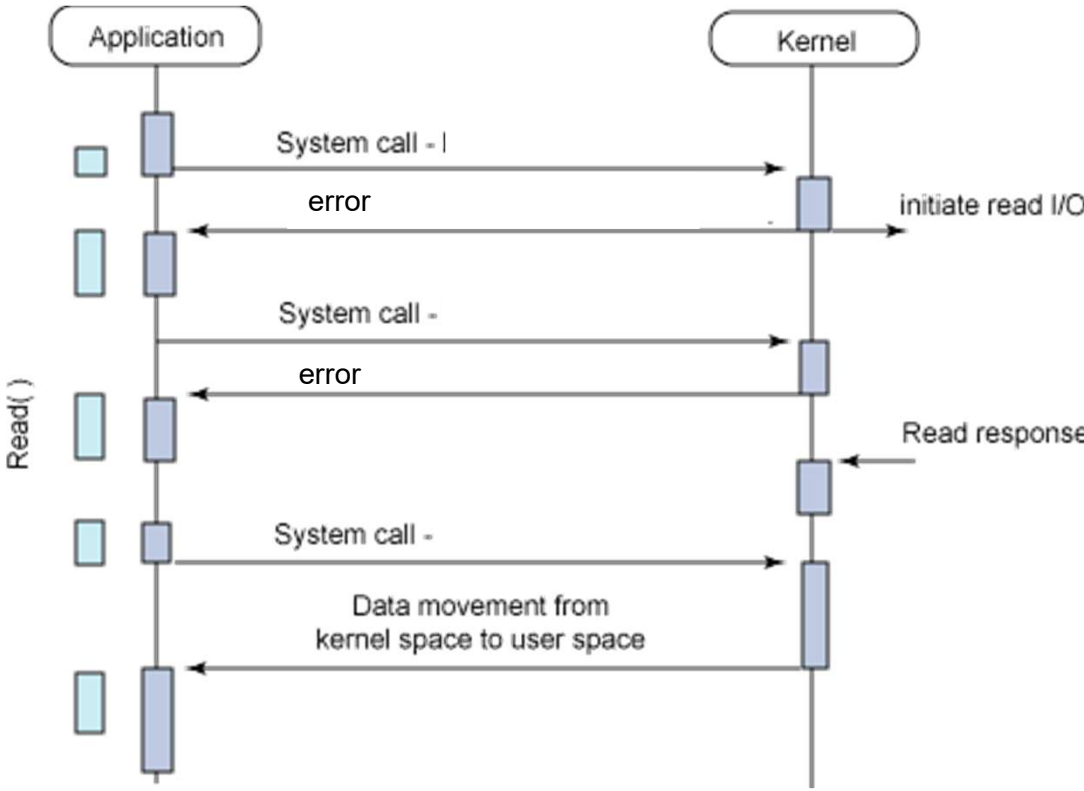
l'applicazione esegue una chiamata di sistema e si blocca fino a che tutti i dati sono ricevuti nel kernel, e copiati dal kernel space alla memoria della applicazione

- **accept():** il thread si blocca fino a che non viene stabilita una nuova connessione
- **write(byte [] buffer):** il thread si blocca fino a che tutto il contenuto del buffer viene inviato sul canale

Image from: https://developer.ibm.com/technologies/linux/articles/l-async/

NonBlocking 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 parte di essi
  - operazione I/O non possibile, restituito un codice errore o valore null
- per completare l'operazione
  - effettuare system-call ripetute
  - Finché l'operazione non può essere completata
- possibile con SocketChannels, SocketServerChannels



From: https://developer.ibm.com/technologies/linux/articles/l-async/

SocketChannel

- un channel NIO associato ad un `socket` TCP
  - 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
- ogni `socketChannel` è associato ad un oggetto `Socket` della libreria java.net, reperibile mediante il metodo `socket()` di SocketChannel

SERVER SOCKET CHANNEL

- un selectablechannel collegato a TCP welcome sockets (listening sockets)
- `ServerSocketChannel` resta in ascolto di richieste di connessione
  - crea nuove SocketChannels per la gestione della connessione
  - non trasferisce dati
- Ad ogni ServerSocketChannel è associato un oggetto ServerSocket
  - **`blocking`**: comportamento analogo a ServerSocket, ma con interfaccia buffer-based
  - **`non blocking`**: l'accept() ritorna immediatamente il controllo al chiamante e può restituire
    - null se non sono presenti richieste di connessione
    - un oggetto di tipo SocketChannel altrimenti

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

serverSocketChannel.socket().bind(new InetSocketAddress(5000));
```

SERVER SOCKET CHANNEL

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

serverSocketChannel.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... }
```

SOCKET CHANNEL

1. creazione di un SocketChannel

- implicita: creato quando una connessione viene accettata su un ServerSocketChannel.
- esplicita, lato client, quando si apre una connessione verso un server

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

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

- modalità blocking/non blocking:

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

- non blocking anche lato client:
  - Es. un'applicazione che deve gestire l'interazione con l'utente, mediante GUI e contemporaneamente, gestire uno o più socket

SOCKET CHANNEL

2. Chiusura di un socket channel

```
socketChannel.close();
```

3. Lettura da un SocketChannel

```
ByteBuffer buf= ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);
```

bytesRead dice quanti bit sono stati letti. Se viene restituito -1, è stata raggiunta la fine dello stream e la connessione viene chiusa.

in modalità non bloccante può restituire 0, può essere necessario ripetere la lettura

SOCKET CHANNEL

4. Scrittura in un socket channel

```
String newData ="Hello";
ByteBuffer buf = ByteBuffer.allocate(64);
buf.put(newData.getBytes());
buf.flip();
while (buf.hasRemaining()) {
      socketChannel.write(buf);
}
```

Il metodo write() è richiamato dentro un ciclo: in modalità non bloccante non ci sono garanzie di quanti byte sono scritti nel Channel. Si ripete quindi write() finchè nel Buffer non ci sono più byte da scrivere.

TIME NIO SERVER

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

public class ServerSocketChannelTimeServer {
 public static void main(String[] args) throws Exception {
   System.out.println("Time Server started");
   ServerSocketChannel serverSocketChannel=ServerSocketChannel.open();
   serverSocketChannel.socket().bind(new InetSocketAddress(5000));
   serverSocketChannel.configureBlocking(false);
   while (true) {
         System.out.println("Waiting for request ...");
         SocketChannel socketChannel = serverSocketChannel.accept();
         if (socketChannel != null) {
             String dateAndTimeMessage = "Date: "+ new

                                  Date(System.currentTimeMillis());
       (continua slide seguente...)
```

TIME NIO SERVER

```
ByteBuffer buf = ByteBuffer.allocate(64);
buf.put(dateAndTimeMessage.getBytes());
buf.flip();
while (buf.hasRemaining()) {
        socketChannel.write(buf);
           }
System.out.println("Sent: " + dateAndTimeMessage);
Thread.sleep(10000);
  }
else {System.out.println("nessuna connessione rilevata");
      Thread.sleep(1000); }}}
```

- esecuzione del programma permette di verificare la modalità non blocking:
- il programma stampa “nessuna connessione rilevata” fino a che non arriva una connessione, poi aspetta prima di testare di nuovo l'arrivo di connessioni
- in generale si possono eseguire altri task in attesa di una connessione

Blocking vs non Blocking - sintesi

- Blocking accept: si blocca finché non arriva una richiesta di connessione
- Non-Blocking accept: controlla se c'è una richiesta da accettare (se c'è l'accetta) e ritorna

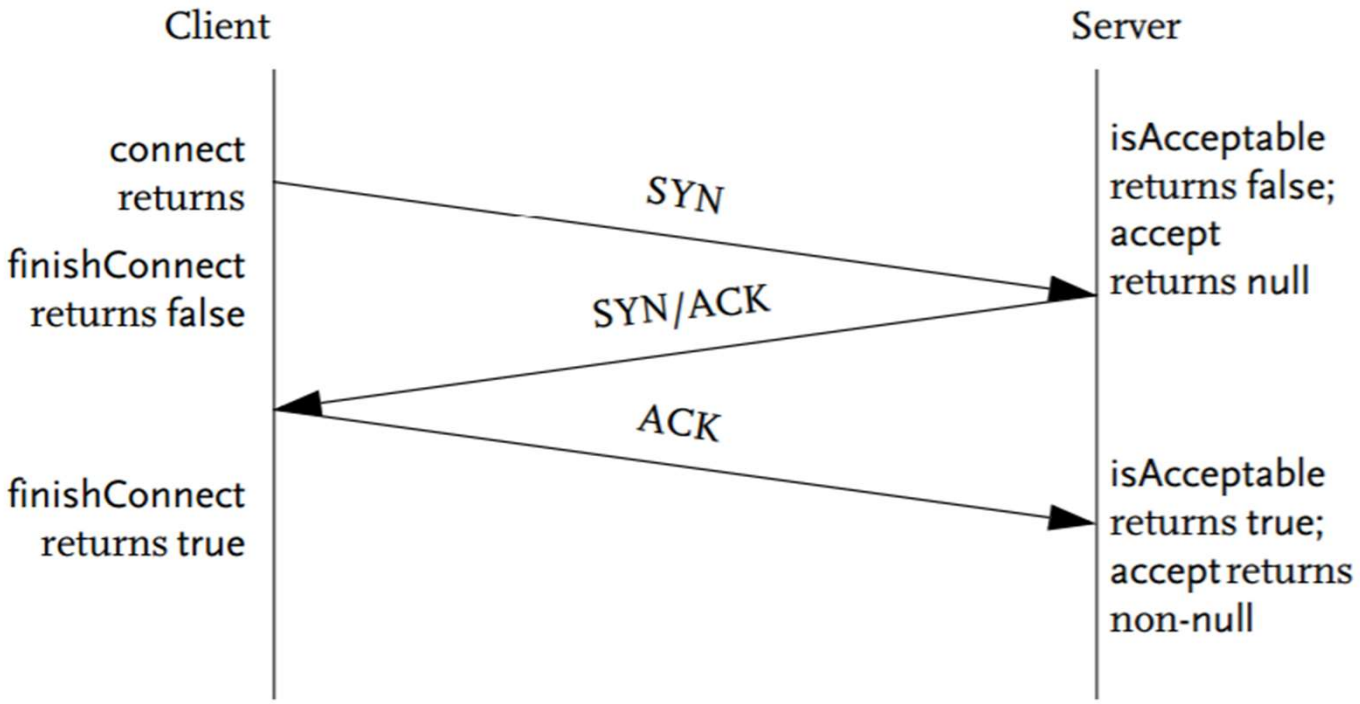
- Blocking write: si blocca finché la scrittura non è completata
- Non-Blocking write: tenta di scrivere i dati nella socket, ritorna immediatamente, anche se i dati non sono stati completamente scritti

- Blocking read: si blocca in attesa di byte da leggere
- Non-blocking read: ritorna immediatamente e restituisce il numero di byte letti (anche 0)

- Blocking vs non-blocking connect() – lato client! -> vedi slide successive

NON BLOCKING CONNECT

`connect()` se il socketChannel è in non blocking mode può restituire il controllo al chiamante prima che venga stabilita la connessione.



NON BLOCKING CONNECT

Se il canale è in blocking mode connect() si blocca fino a quando la connessione non viene completata o non può essere stabilita (restituirà true o genererà un'eccezione)

In un canale non-bloccante `connect()` può ritornare prima che la connessione sia stabilita.

`finishconnect()`: serve per controllare la terminazione della operazione (i.e. instaurazione della connessione). Il metodo non si blocca, se la connessione non è ancora stabilita restituisce false, altrimenti true

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

NON BLOCKING NIO CLIENT

```
import java.nio.channels.*; import java.net.*;
import java.io.*; import java.nio.*;
public class SocketChannelTimeClient {
  public static void main(String[] args) throws Exception {
      SocketAddress address = new InetSocketAddress("127.0.0.1", 5000);
      SocketChannel socketChannel;
      socketChannel = SocketChannel.open();
      socketChannel.configureBlocking(false);
      socketChannel.connect(address);
      System.out.println(socketChannel.finishConnect());
       while(!socketChannel.finishConnect()){
          //wait, or do something else...
        System.out.println("Connessione non terminata");
       }
      System.out.println("Terminata la fase di instaurazione della
connessione");
```

NON BLOCKING CLIENT: OSSERVAZIONI

- se togliamo

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

le operazioni di I/O sul canale possono sollevare

java.nio.channels.NotYetConnectedException

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)

SINGLE THREAD MODEL

Un solo thread per tutti client:

- **scalabilità:** nulla, in ogni istante, solo un client viene servito
- **acceptance 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 basso

UN THREAD PER OGNI CONNESSIONE

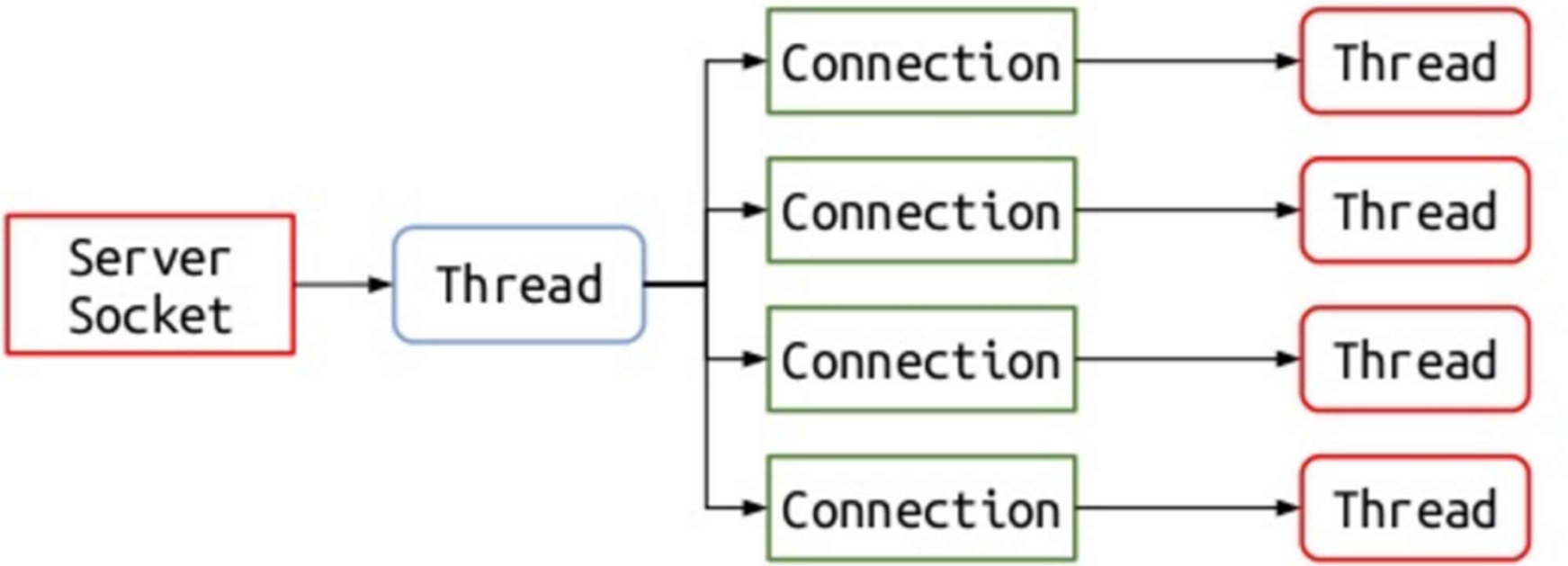
- **scalabilità:** possibilità di servire diversi clienti in maniera concorrente, fino al massimo numero di thread previsti per ogni processo
- ogni thread alloca il proprio stack: memory pressure
- impossibile predire il numero massimo di client: dipende da fattori esterni e può essere molto variabile

- **acceptance 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

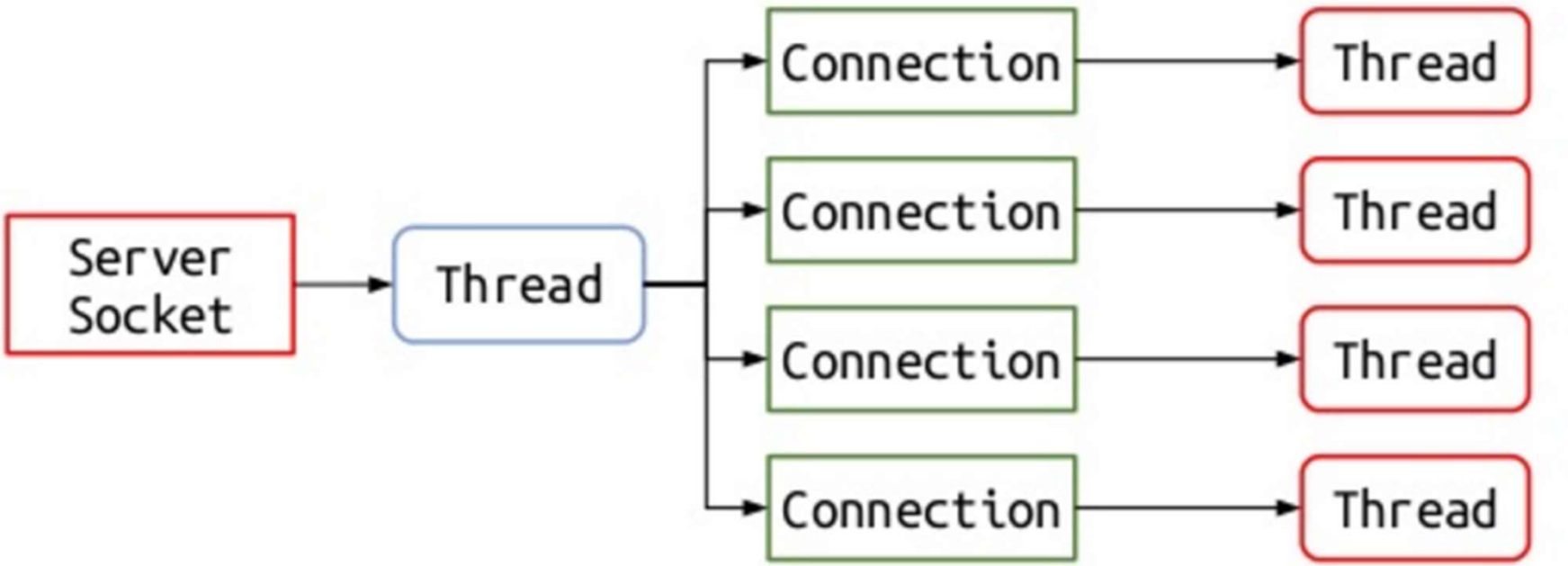
- **efficiency:** bassa - ogni thread può essere bloccato in attesa di IO, ma utilizza risorse come la memoria

UN THREAD PER OGNI CONNESSIONE



considerare lo scenario di un server (o peer) che deve gestire un gran numero di connessioni

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

UN NUMERO FISSO DI THREAD

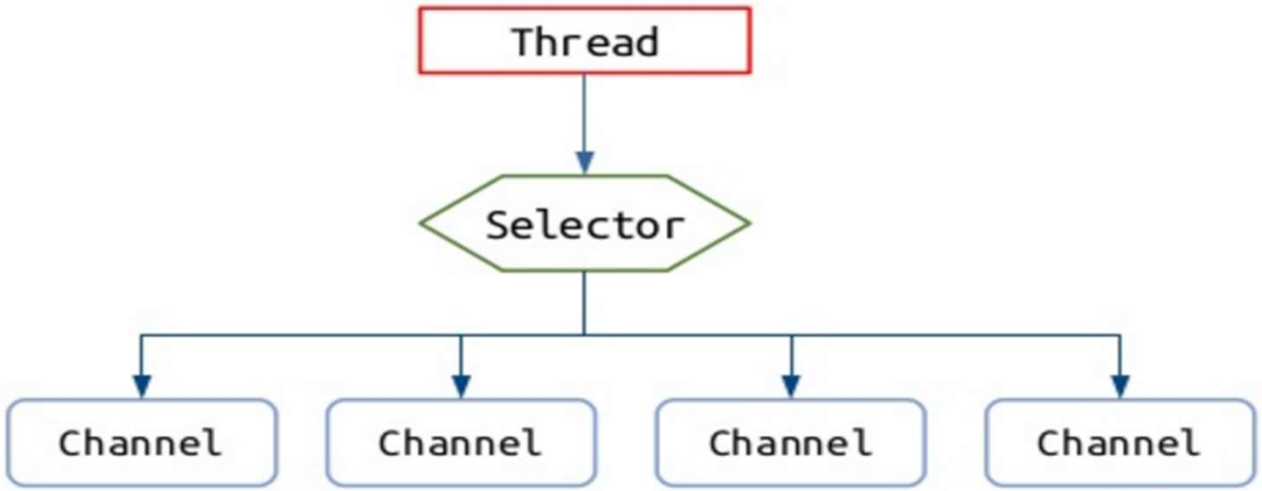
Un numero costante/massimo di thread: utilizza Thread Pool

- **scalabilità:** limitata al numero di connessioni che possono essere supportate.
- evita crash nel caso di alto numero di connessioni contemporanee
- **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

Non-blocking IO

- Una delle funzionalità più importanti del NIO è il comportamento non-blocking per le varie operazioni IO
- Non-blocking: l'operazione restituisce subito un risultato, anche se a volte l'operazione non può essere eseguita
- Es. Non-blocking read: se ci sono byte da leggere, li legge, altrimenti restituisce 0 (non è stato possibile leggere nessun byte)
- Usando le funzionalità non-blocking, il programma deve ripetere le operazioni finché queste vengono completate
  while(buffer.remaining() && channel.read(buffer)!=-1) {}
- Se ci sono più operazioni da eseguire (es. più channel da leggere) si deve iterare tra tutti i channel
- NON E' UNA SOLUZIONE EFFICIENTE! (active loop, tante operazioni IO non sono ancora disponibili)
- SOLUZIONE MIGLIORE -> Multiplexed I/O (Non-Blocking mode con SELETTORE)

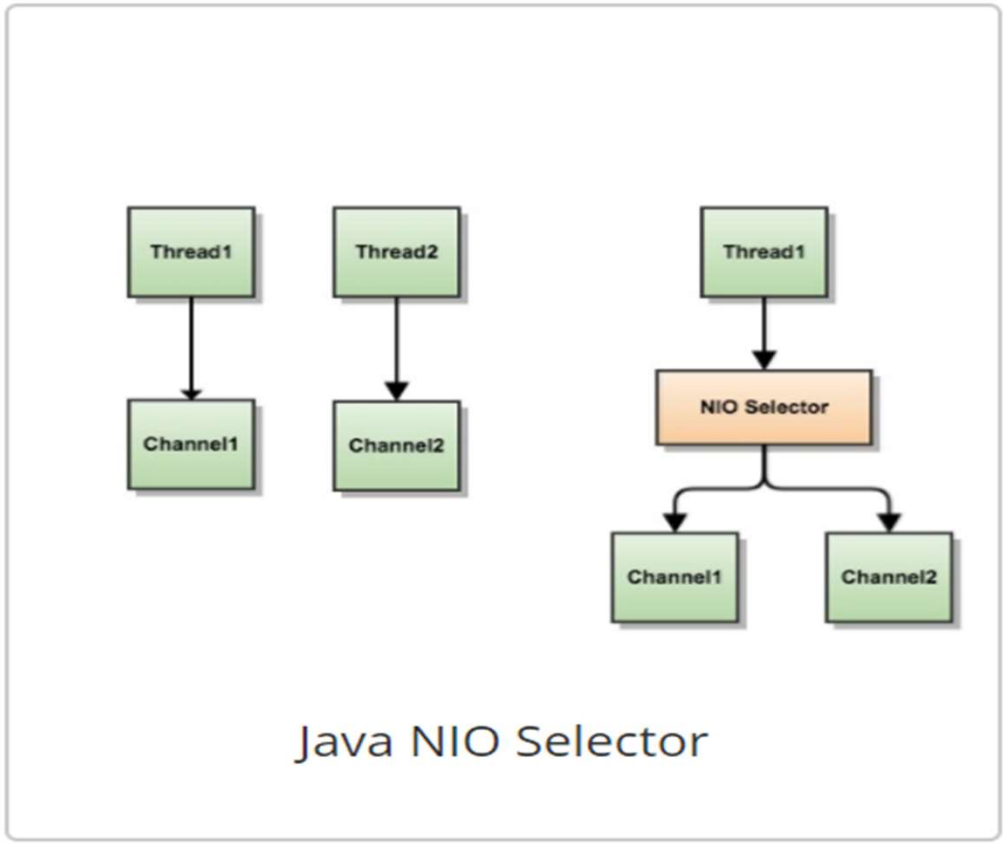
Multiplexed I/O con JAVA NIO



- 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
- miglioramento di performance e scalabilità (numero di thread basso anche con migliaia di sockets)
- SVANTAGGIO: architettura più complessa da capire e da implementare

MULTIPLEXED SERVER

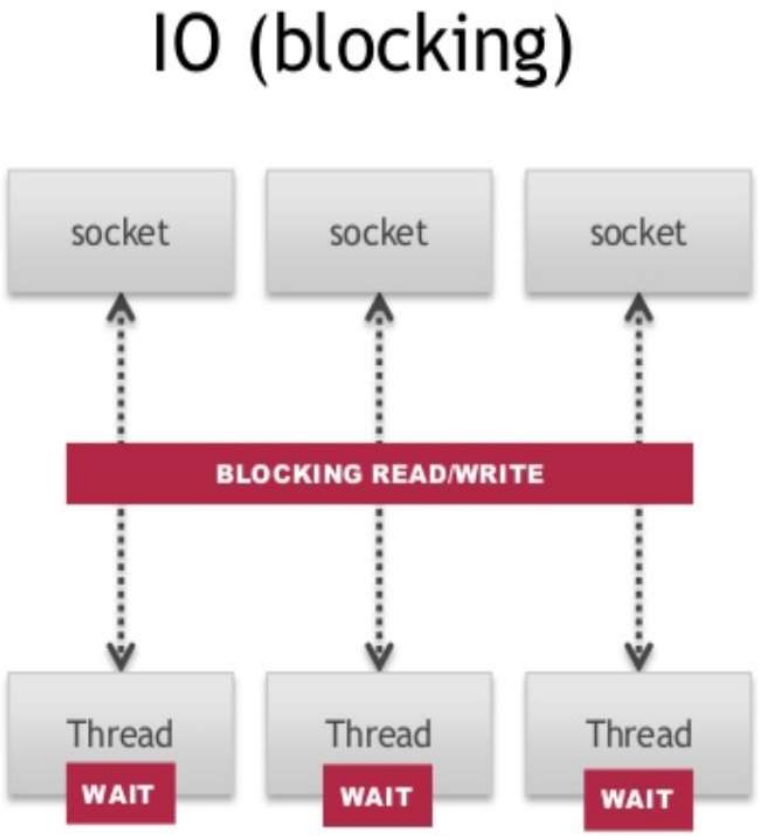
- **Selector:** Oggetto che facilita il *multiplexing* dei *channel*
- **readiness selection**
- componente JAVA che può esaminare uno o più NIO SelectableChannels e determinare se sono pronti per una operazione di rete
  - accept, write, read, connect
  - esempio: avvenuta connessione, arrivati dati, pronto per la scrittura

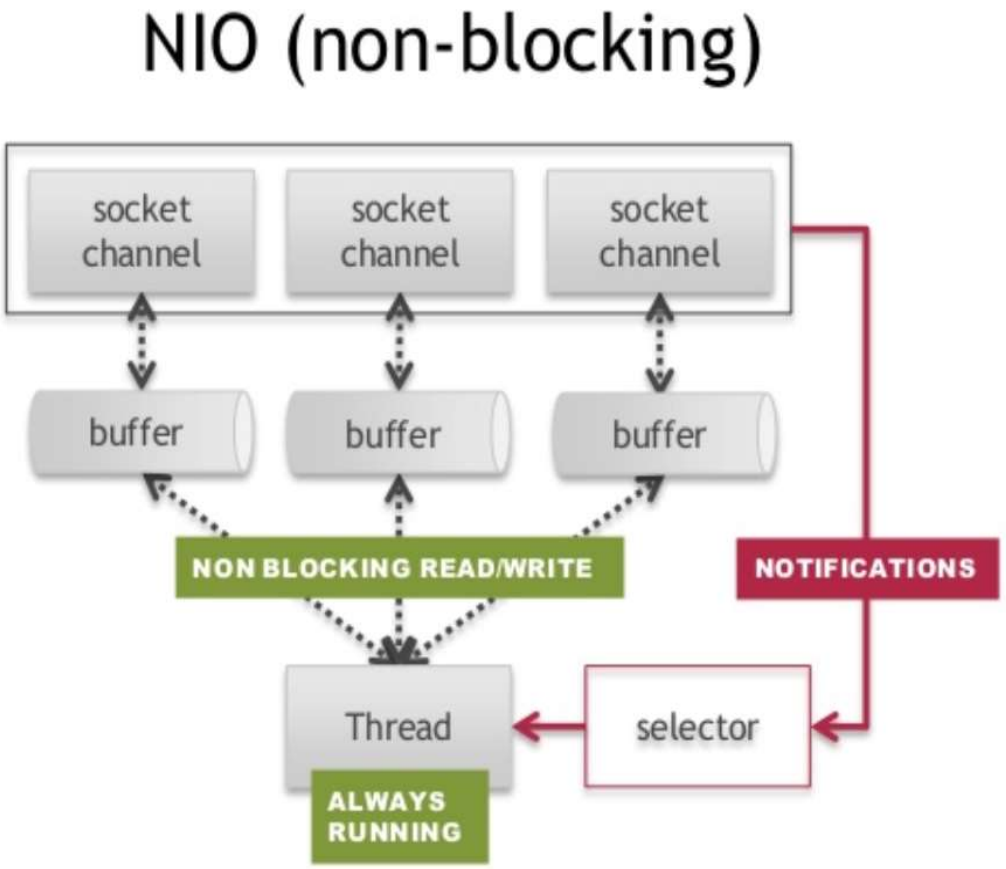


Selettore

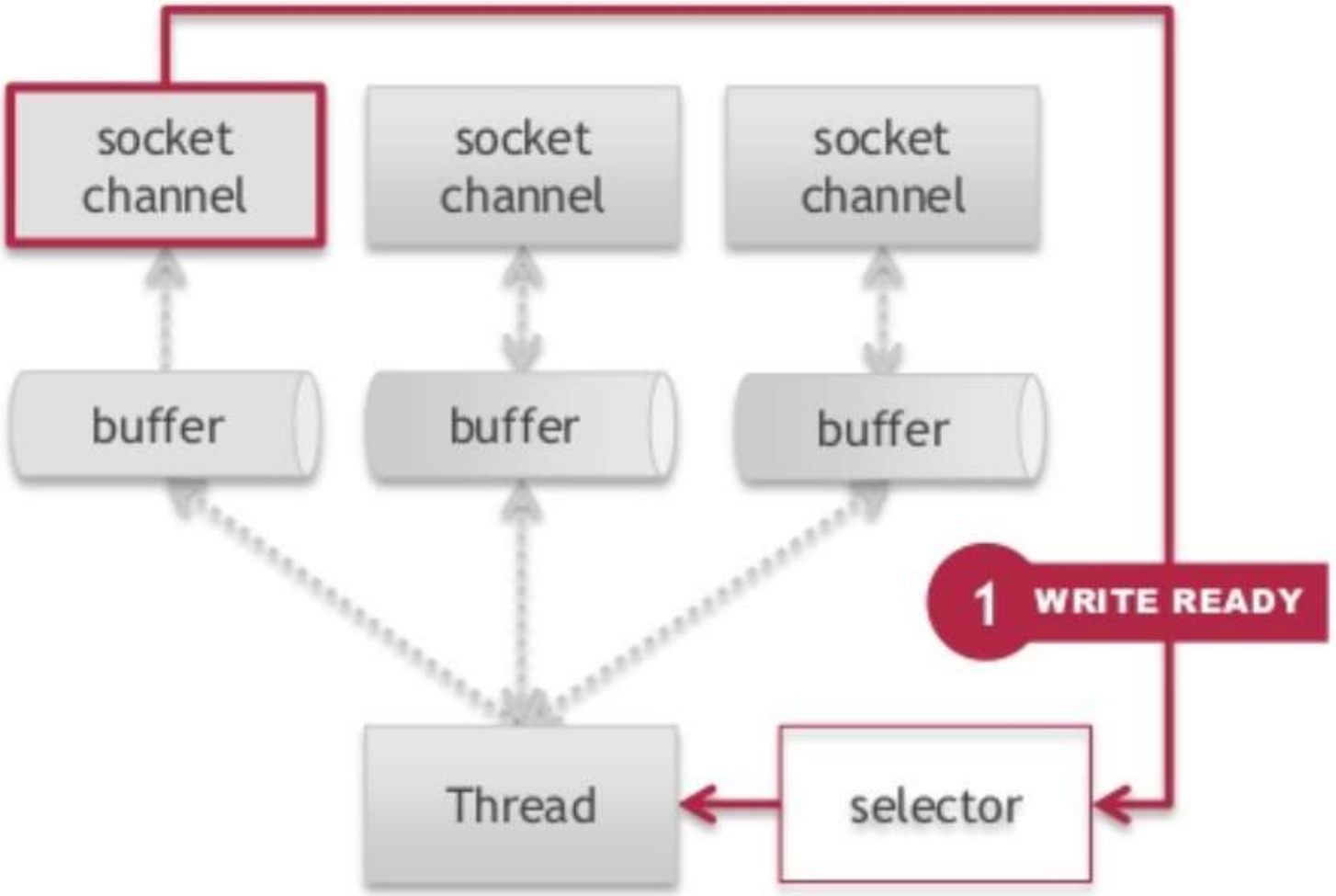
- consente di implementare **multiplexing**:
  - gestione, da parte dello stesso thread o da un numero limitato di thread, di più eventi che possono avvenire simultaneamente
- A questo punto il programma può iterare tra i *channel* pronti e eseguire le operazioni in modo *non-blocking* (il selettore garantisce che le operazioni non saranno inutili: es. ci sono *byte* da leggere)
- Per essere usati con selettori, i *channel* devono essere selezionabili: estendere classe astratta SelectableChannel
  - ServerSocketChannel
  - SocketChannel
  - DatagramChannel
  - …
  - file I/O non inclusa

NIO con Selector

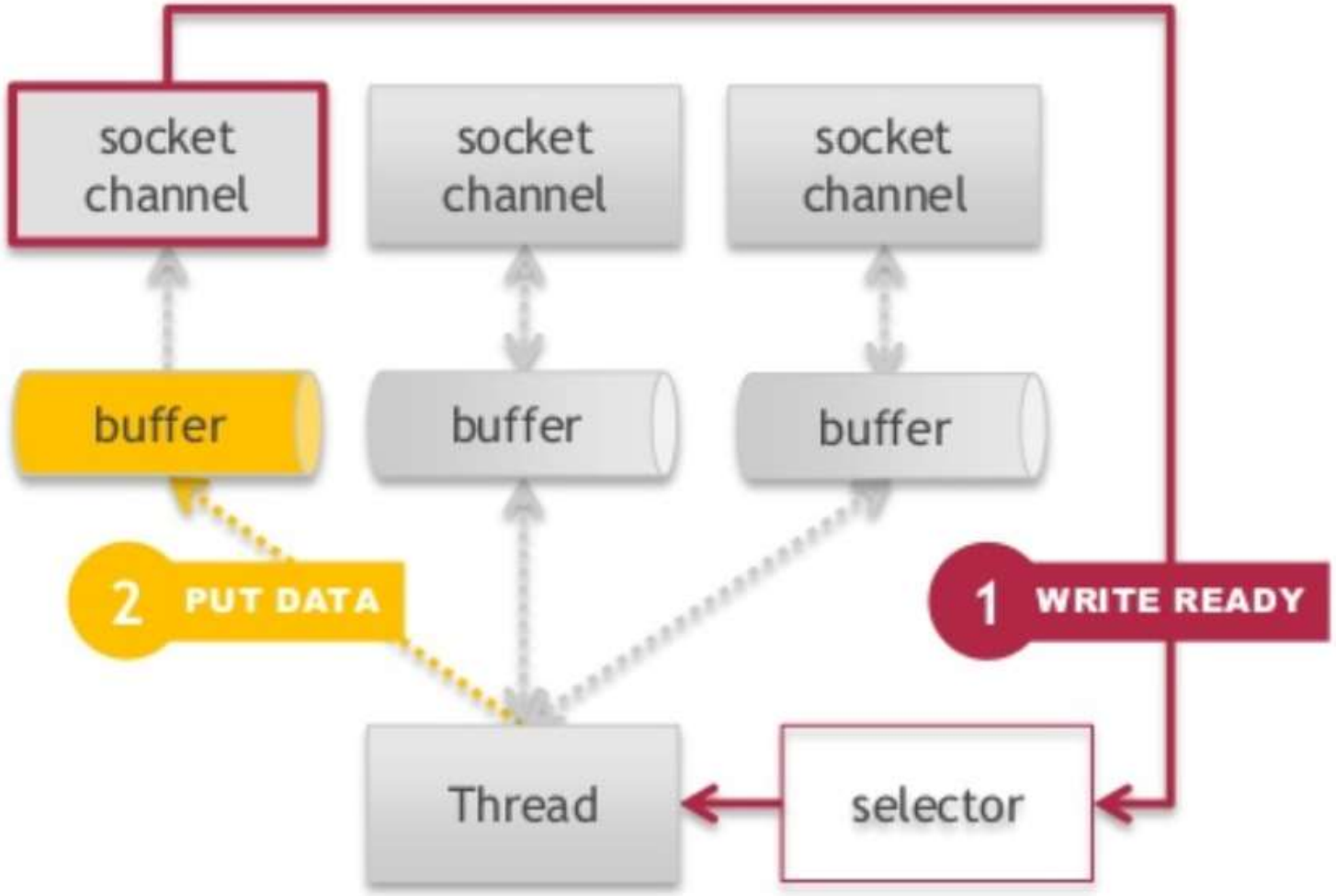




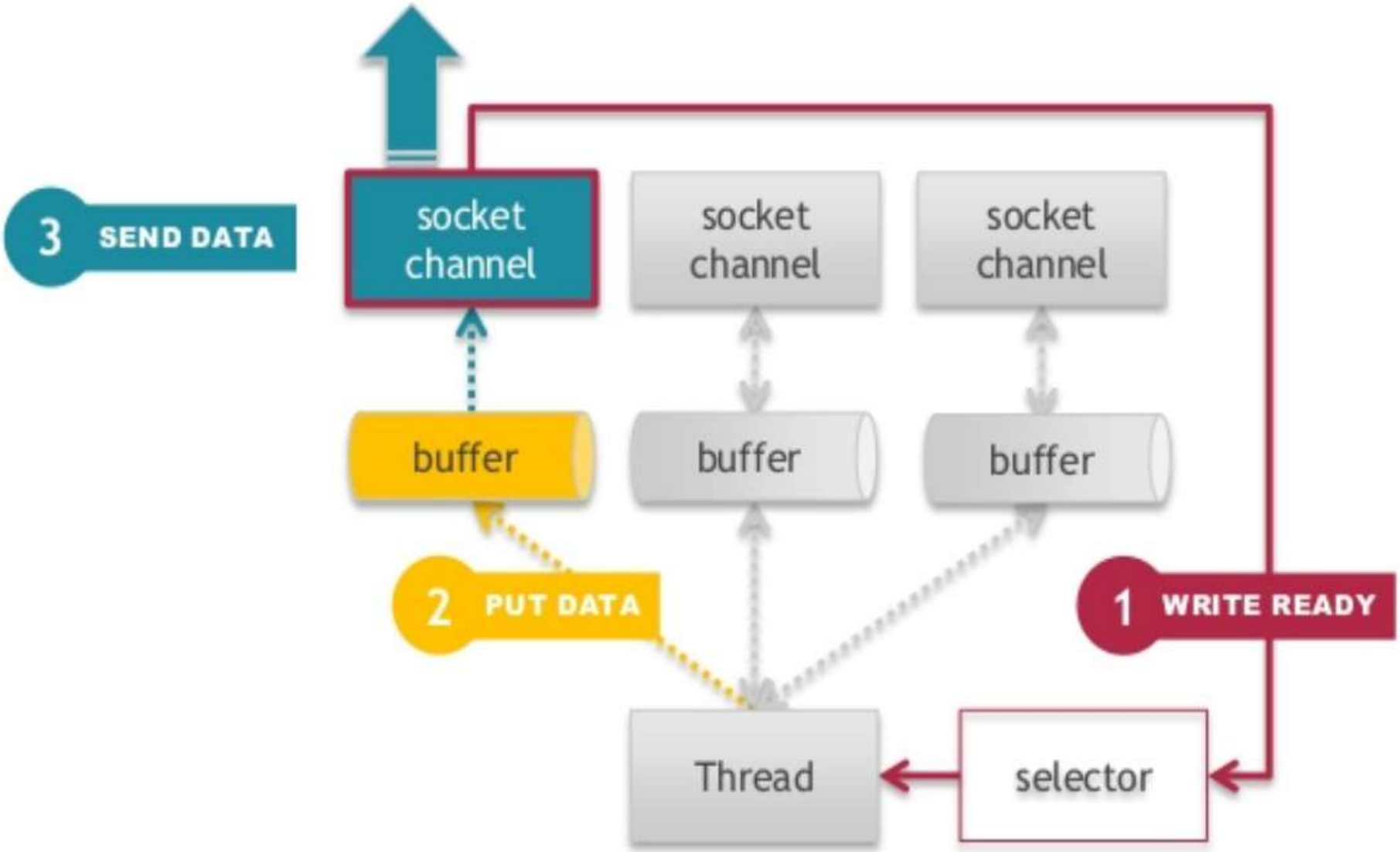
NIO con Selector



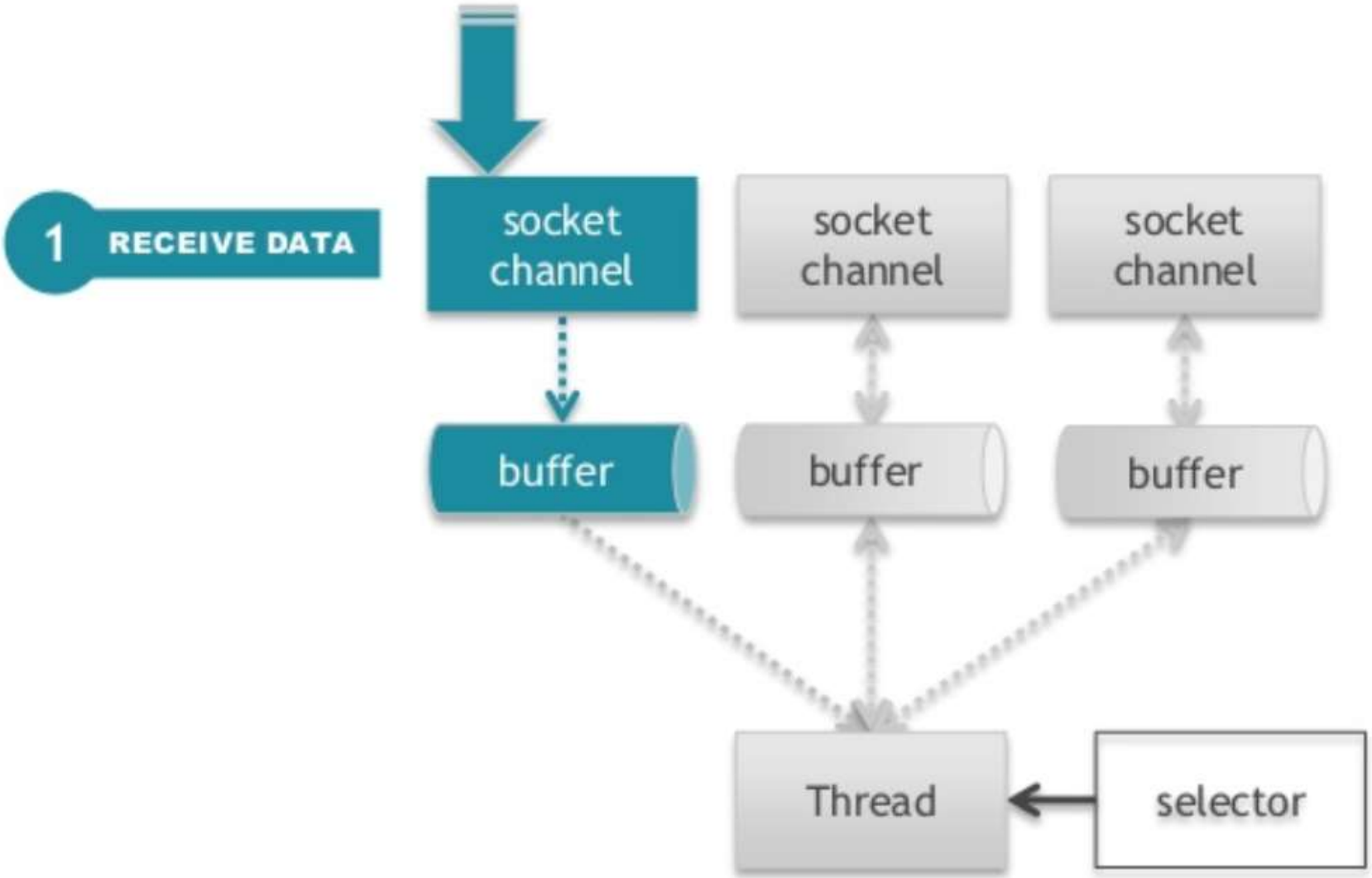
NIO con Selector



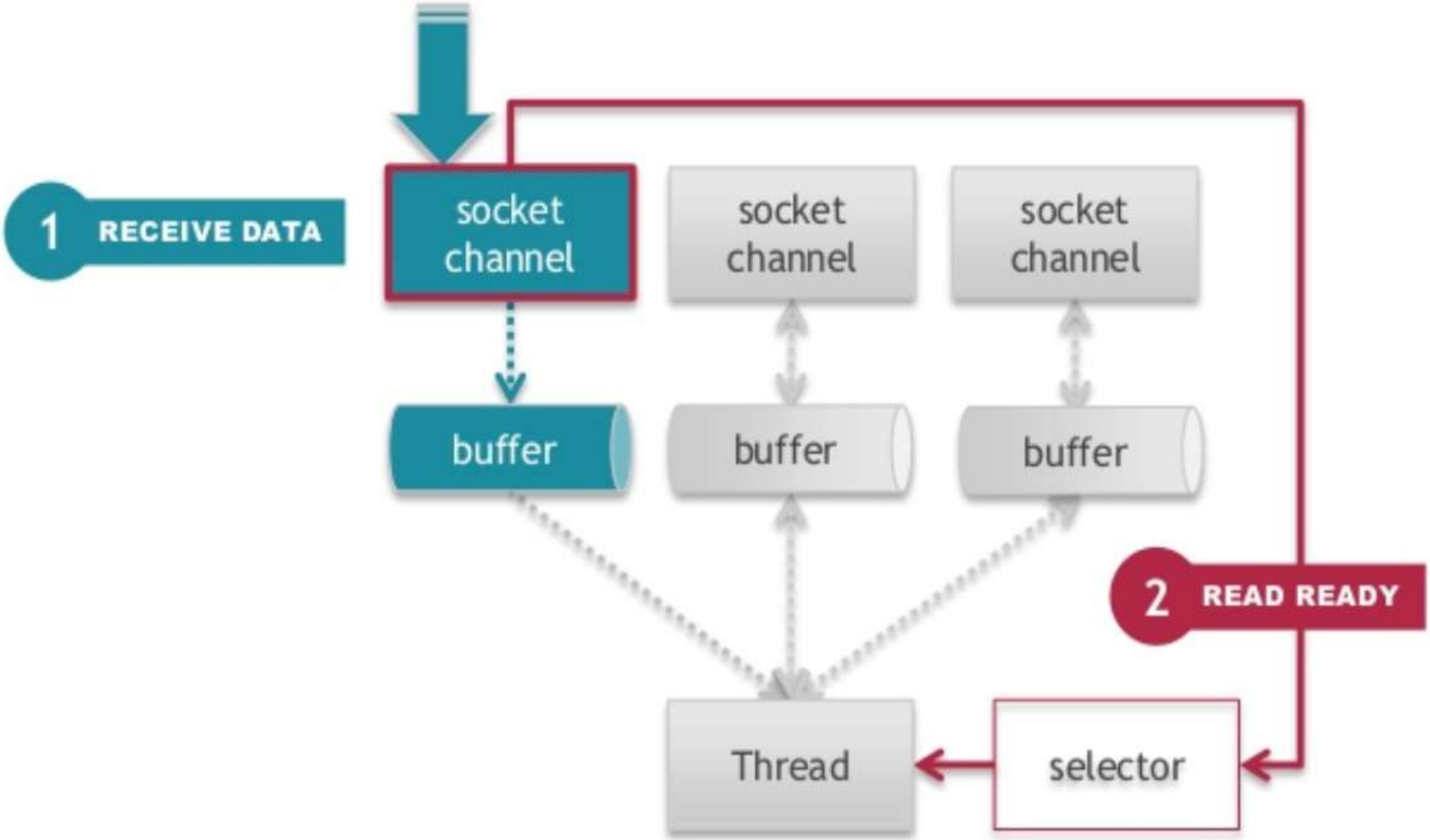
NIO con Selector



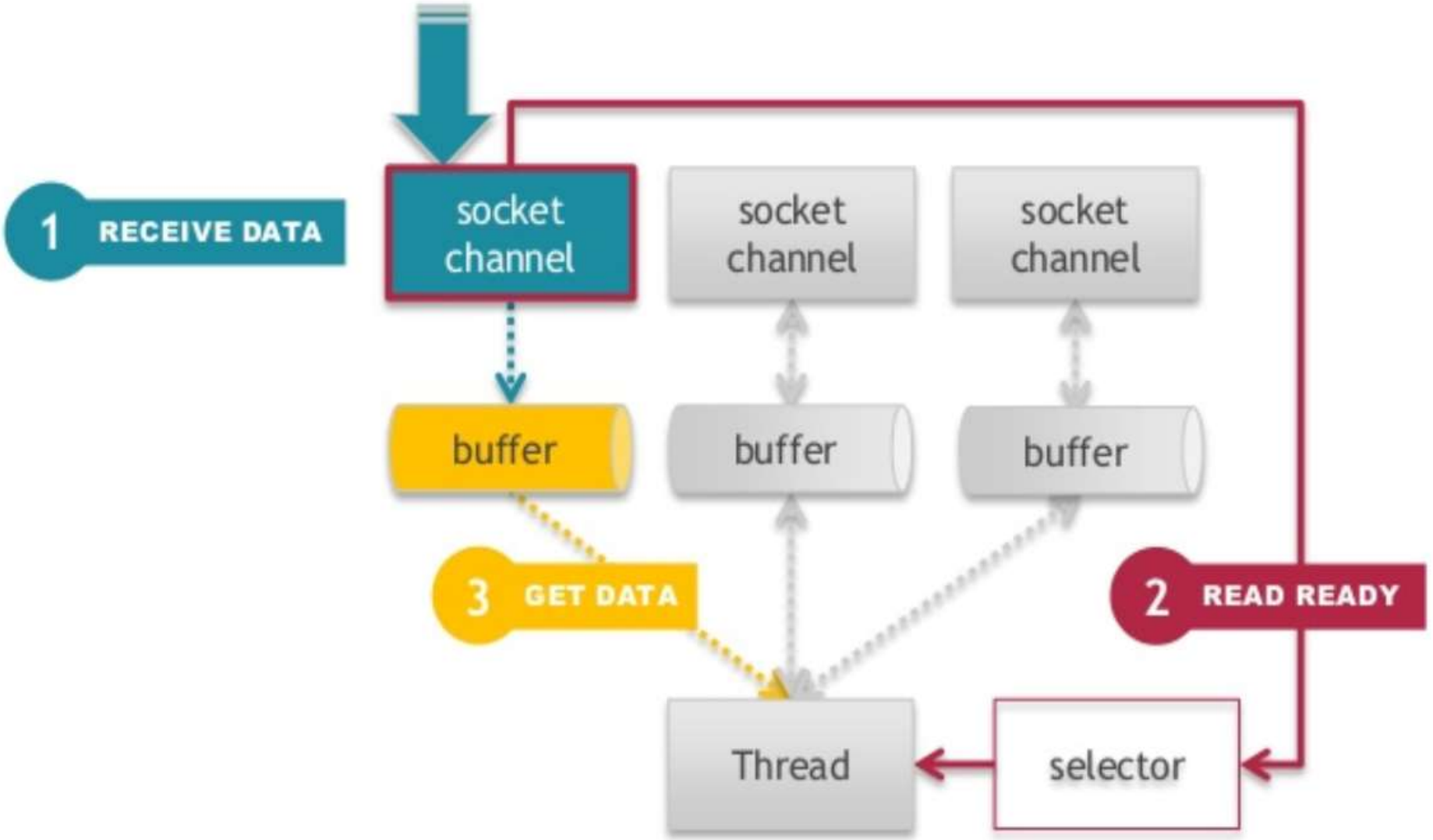
NIO con Selector



NIO con Selector



NIO con Selector



SelectableChannel

- Classe astratta di base per channel che possono essere usati con selettori
- Metodi per impostare comportamento blocking/non-blocking del channel
- Metodi per la registrazione con un selettore
- Alla creazione, tutti i *channel* sono bloccanti.

```
SelectableChannel configureBlocking(boolean block)
```

- Imposta il comportamento *bloccante/non-bloccante*.

- Un *channel* deve diventare non-bloccante prima di essere registrato con un selettore

`boolean isBlocking()` verifica se un *channel* è bloccante

SELECTOR: REGISTRAZIONE DEI CANALI

- Supportato da due classi principali
  - `Selector`: fornisce le funzionalità di multiplexing
  - `SelectionKey`: identifica i tipi di eventi pronti per essere elaborati

- Per utilizzare un selettore occorre:
  - creare il selettore
  - registrare i canali su quel selettore
  - selezionare un canale quando è disponibile (si è verificato un evento su quel canale)

`Selector open()`	crea un selettore
`void close()`	chiude il selettore
`boolean isOpen()`	verifica se il selettore è aperto

REGISTRAZIONE DEI CANALI SU SELETTORI

- i canali devono essere registrati su un selettore per operazioni specifiche
- `register()`: applicata ad un oggetto canale non bloccante, richiede:
  - il selettore
  - la specifica delle operazioni (eventi) di interesse

```
SelectionKey key=
    channel.register(selector,[Operation1|Operation2|…]);
```

  - Aggiunge questo *channel* alla lista di *channel* gestiti dal selettore `selector`.
  - Restituisce una chiave di selezione (`SelectionKey`) che include un riferimento al *channel* **(un "rappresentante " del canale nel selettore)**.
  - Se il *channel* era già registrato, la chiave iniziale di selezione viene restituita, dopo essere stata aggiornata con le nuove operazioni di interesse

- lo stesso canale può essere registrato con più selettori: la chiave identifica la particolare registrazione.

REGISTRAZIONE DEI CANALI SU SELETTORI

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

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

REGISTRAZIONE DEI CANALI SU SELETTORI

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

- La `SelectionKey` può anche contenere un oggetto definito dal programmatore, chiamato *attachment* (uno spazio di memorizzazione in cui si possono memorizzare informazioni, una sorta di stato del channel*).*
- Ad esempio per dare un custom ID al channel registrato o allegare il riferimento a un oggetto di cui vogliamo tenere traccia (es. Buffer).

Selection key

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

```
public final boolean isAcceptable()
public final boolean isConnectable()
public final boolean isReadable()
public final boolean isWritable()
```

Metodi per verificare se il channel è pronto per queste operazioni

```
SelectableChannel channel()
```

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

Tipi di Operazioni e canali "pronti"

Operation	Meaning
OP_ACCEPT (ServerSocketChannel)	ServerSocketChannel.accept would not return null: either an incoming connection exists or an exception is pending.
OP_CONNECT (SocketChannel with connection pending)	SocketChannel.finishConnect would not return false: either the connection is complete apart from the finishConnect step or an exception is pending, typically a ConnectException.
OP_READ (connected SocketChannel)	read would not return zero: either data is present in the socket receive-buffer, end-of-stream has been reached, or an exception is pending. End-of-stream occurs if the remote end has closed the connection or shut it down for output, or if the local end has shut it down for input.
OP_WRITE (connected SocketChannel)	write would not return zero: either space exists in the socket send-buffer, the connection has been closed or shutdown for input at the remote end, or an exception is pending.

NB Il metodo validOps() invocato su una istanza di SelectableChannel restituisce tutte le operazioni valide per quel *channel*. E.g. per un SocketChannel aperto per leggere e scrivere il metodo restituisce (SelectionKey.OP_READ| SelectionKey.OP_WRITE)

Selection key

`Object attachment()`

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

`void cancel()`

- Cancella la registrazione. Una chiave di registrazione è valida finché questo metodo viene richiamato, o il *channel* o il selettore vengono chiusi.
- `Selector selector = key.selector();`

- La key memorizza canale, operazione sul canale, allegato ed inoltre due bit mask, codificate come interi
  - **interest set**: indica per quali operazioni del canale si è registrato un interesse
  - **ready set:** quali operazioni sono pronte sul canale

INTEREST SET

- operazioni per cui si è registrato un interesse
- definito in fase di registrazione del canale con il Selector

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

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

- modificabile, successivamente, invocando interestOps

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

- reperibile tramite

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

READY SET

- insieme di operazioni sul canale associato alla chiave che sono “pronte”, ovvero che il canale è pronto ad eseguire
  - sottoinsieme dell'interest set
- inizializzato a 0 quando la chiave viene creata e successivamente aggiornato quando si esegue una `select()` (vedi slide successive)
- restituito dal metodo readyOps( ) invocato su una `SelectionKey`

```
if ((key.readyOps( ) & SelectionKey.OP_READ) != 0) {
      myBuffer.clear( ); //preparo il buffer per la write
      key.channel().read(myBuffer); //leggo dal channel
      doSomethingWithBuffer
       myBuffer.flip();
       }
```

```
key.isReadable() equivale a
            if ((key.readyOps() & SelectionKey.OP_READ) != 0)
```

in modo analogo 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( ) {};}
```

MULTIPLEXING DEI CANALI

- `int n = selector.select( );`
  - bloccante, si blocca finché almeno un channel è pronto
  - seleziona i canali pronti per almeno una delle operazioni di I/O tra quelli registrati con quel selettore
  - restituisce il numero di canali pronti (diverso da 0)
  - costruisce un insieme contenente le chiavi dei canali pronti
  - il thread che esegue la selezione può essere interrotto e il selettore viene sbloccato mediante il metodo `wakeup()` invocato da un altro thread

- `int select(long timeout)`
  - si blocca fino a che non è trascorso il timeout, oppure come sopra

- `int selectNow()`
  - non si blocca e, 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`: SelectionKeys dei canali registrati con quel selettore.
  - restituite dal metodo **keys()**

- `Selected Key Set:` SelectionKeys dei canali identificati come pronti, nell'ultima operazione di selezione, per eseguire almeno una operazione contenuta nell'interest set della chiave,
  - restituite dal metodo **selectedKeys()**
  - diverso dal ready set: selectedKeys è un insieme di chiavi e per ogni chiave esiste il ready set
  - Per ogni chiave dobbiamo determinare quale evento sia accaduto

- `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()?

Processo di selezione, invocato da una select(), comprende due fasi:

- gestione di operazioni di `cancel()`: rimozione di ogni chiave appartenente al cancelled key set dagli altri due insiemi e rimozione della registrazione del canale con il selettore
- interazione con il sistema operativo per verificare lo stato di "readiness" di ogni canale, per ogni operazione specificata nel suo interest set.

Cosa fa la select()?

- per ogni canale con almeno una operazione “ready” dell'interest set:
  - se la chiave corrispondente non appartiene già al selected key set
    - la chiave viene inserita nel selected key set
    - il ready set di quella chiave viene resettato ed impostato con le chiavi corrispondenti alle operazioni pronte
  - altrimenti: il ready set viene aggiornato calcolando l'or bit a bit con il valore precedente del ready set.
    - un bit settato non viene mai resettato, ma i bit ad 1 si “accumulano” man mano che le operazioni diventano pronte.

SELECTOR: DEFINIZIONE

- A questo punto si itera sull'insieme di chiavi che individuano i “canali pronti”
- per ogni chiave, si individua il tipo di operazione per cui il canale è pronto
  - si effettua l’operazione (es. lettura, scrittura,…)
- il programma ha preso in carico gli eventi, li cancello dal selectedKey set.
  - Si invoca `keyIterator.remove()`, poiché il Selector non rimuove le istanze delle SelectionKey dall'insieme delle chiavi selezionate

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 incrementalmente aggiornato
  - scelta di progetto: assegnare al programmatore la responsabilità di aggiornare esplicitamente le chiavi
  - per resettare il `ready set` si deve rimuovere la chiave dall'insieme delle chiavi selezionate

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.
    }
    if (key.isConnectable()) {
        // a connection was established with a remote server.
    }
    if (key.isReadable()) {
            // a channel is ready for reading
    }
    if (key.isWritable()) {
        // a channel is ready for writing }
```

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 uno sequenza di interi infinita
- 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) {}
}}}}}
```

NOTA BENE:
write in un canale non bloccante

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(); } }  }
```

Ultime considerazioni

- 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 perché le operazioni di lettura o scrittura sono non bloccanti (nessuna assunzione sul numero di bytes letti/scritti)

- NIO MULTIPLEXED AND MULTITHREAD SERVER
  - combinare multiplexing e multithreading
  - Un thread per il selettore ed un threadpool per servire i canali pronti (es. processare la richiesta e preparare la risposta)

Esercizio di preparazione all'assignment

- Scrivere un programma JAVA che implementa un server TCP 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 multiplexed NIO (canali non bloccanti e il selettore, e ovviamente i buffer di tipo ByteBuffer).
- Per il client potete usare un client telnet.
- Due opzioni possibili (per l'esercizio) -> vedi slide successiva

Esercizio

1. Opzione più semplice:

- come primo esercizio potete sviluppare un programma in cui quando la serverSocketChannel ha connessioni da accettare (key.isAcceptable() è vera) il server scrive subito sulla socketChannel restituita dall'operazione di accept() e chiude la connessione.

2. Opzione più completa (vedi esempio IntGenServer sulle slide)

- Se key.isAcceptable() è verificata, 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 9: 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.