Distributed Systems based on Java/NIO

Polytech/INFO 4, 2022-2023
Fabienne Boyer, Olivier Gruber,
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

Using Sockets

- **Considering TCP**
  - Fifo, Lossless
  - Stream-oriented (read/write bytes)
  - **Blocking API**
    - read() blocks until there is some data to read
    - write() blocks until some data can be written

    → requires one worker thread per client (for reactivity)

    → limited scalability (nb simultaneous clients limited to nb threads)

    → limited performances (commutation price with high nb of threads)

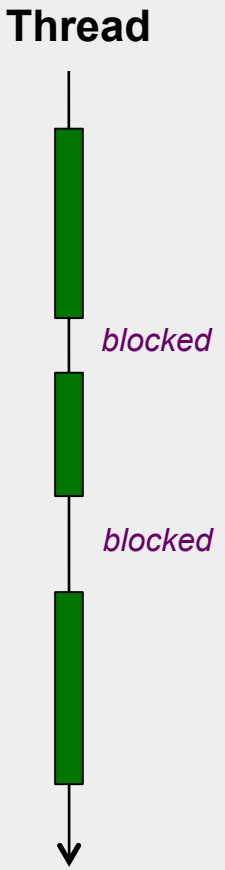
Using Sockets

Blocking API

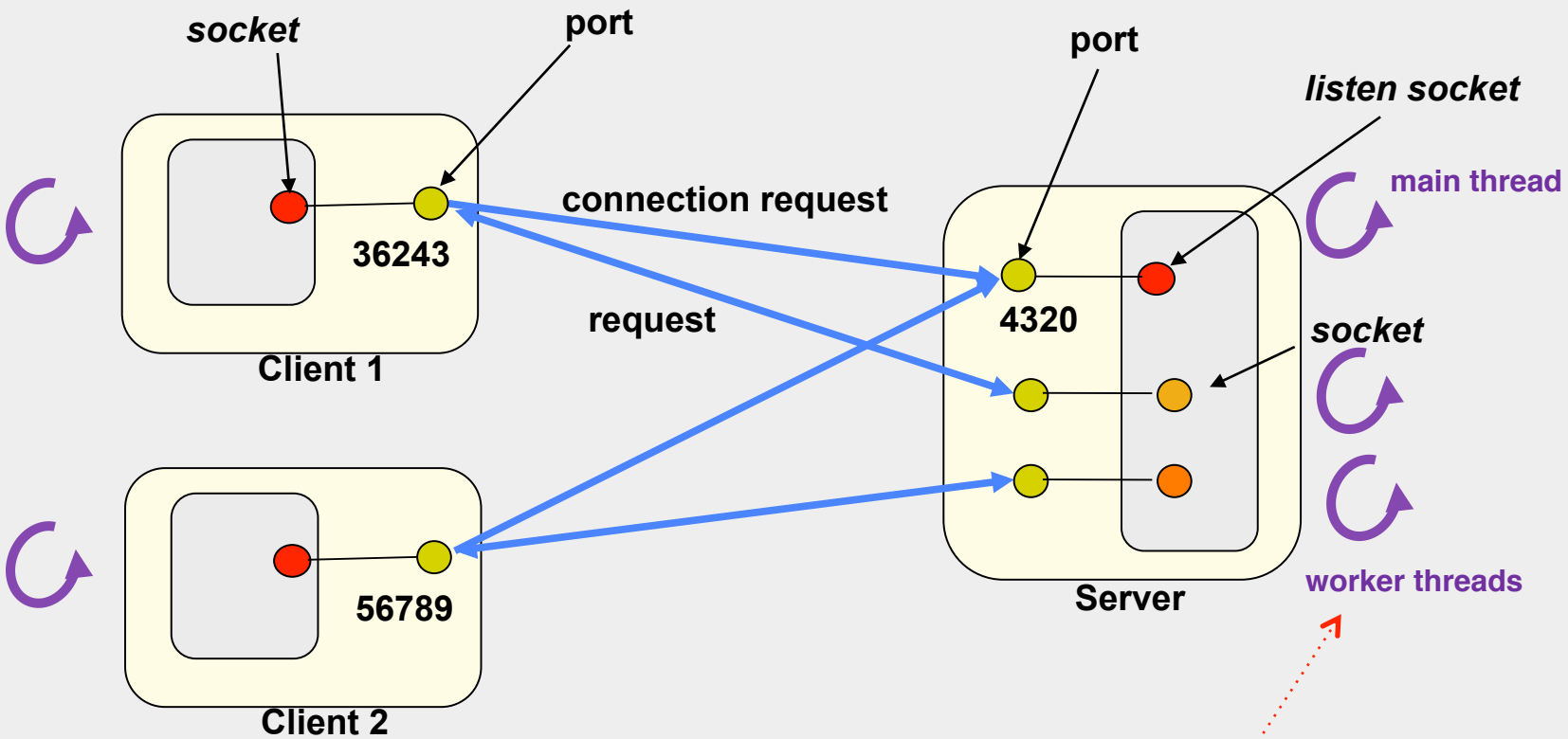
```
void send(OutputStream os, byte[] bytes, int off, int count) {
      ..
      os.write(bytes,off,count); // block if buffer is full
      ..
}
```

```
void receive(InputStream is, byte[] bytes, int off, int count) {
      ..
      is.read(bytes,off,count); // block if buffer is empty
      ..
}
```

```
void receive(InputStream is, byte[] bytes, int off, int count) {
     ..
     // even if read() was not blocking, loop until there is bytes to
     // read, which means blocking until the buffer is not empty
     while (is.read(bytes,off,count) == 0);
     ...
}
```



Using Sockets



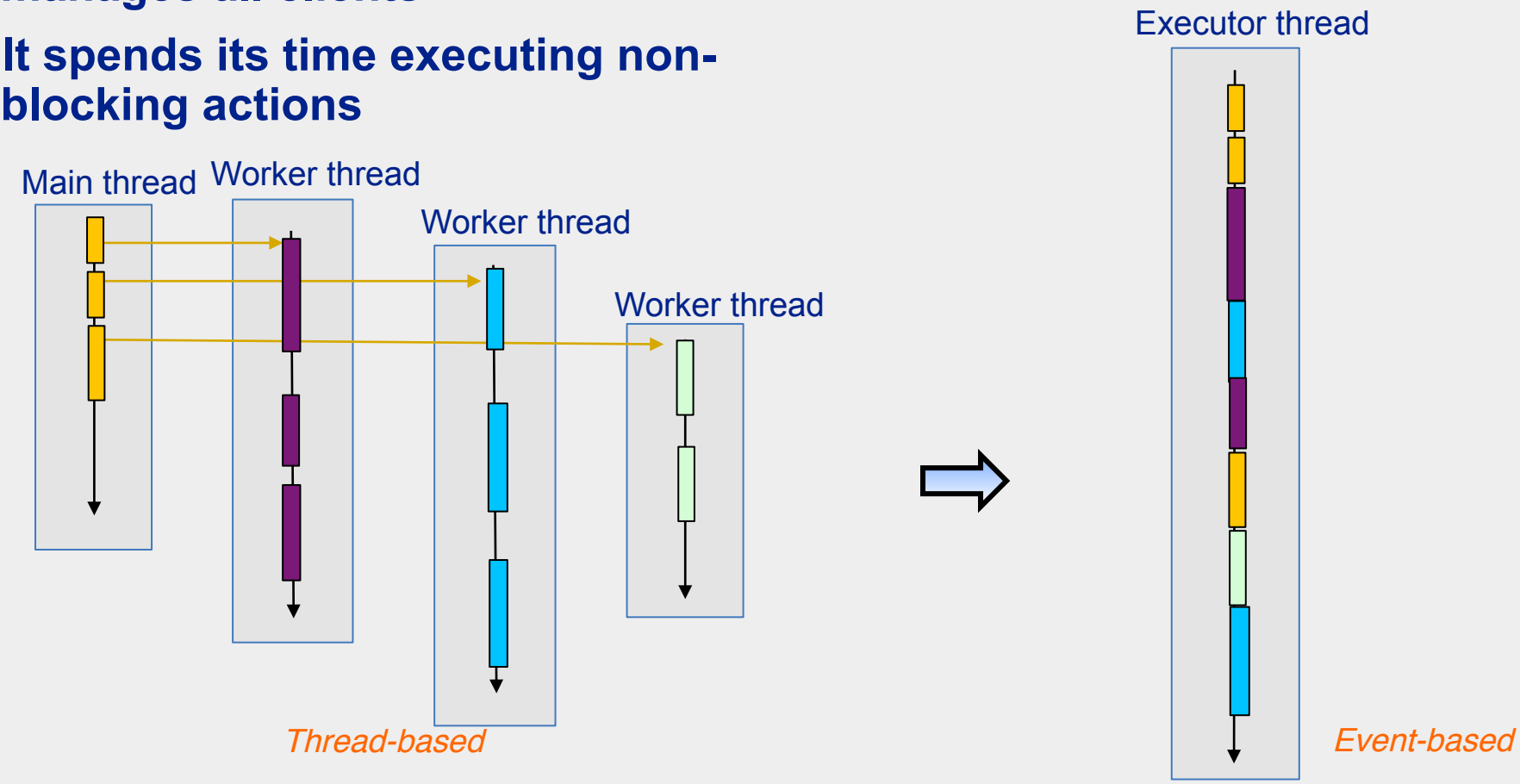
One worker processes one client request entirely, with idle periods (blocking calls)

New Socket IO (Java NIO)

- **Designed for IO intensive applications**
  - Scalability
  - Performances
- **Two main design principles**
  - Event-based model based on non-blocking methods *..as Node.js*
    - read() and write() do not block
    - read() only reads the bytes that are currently available
    - write() only writes the bytes that can be currently written
  - Optimized data transfers
    - Optimized data buffers at network level

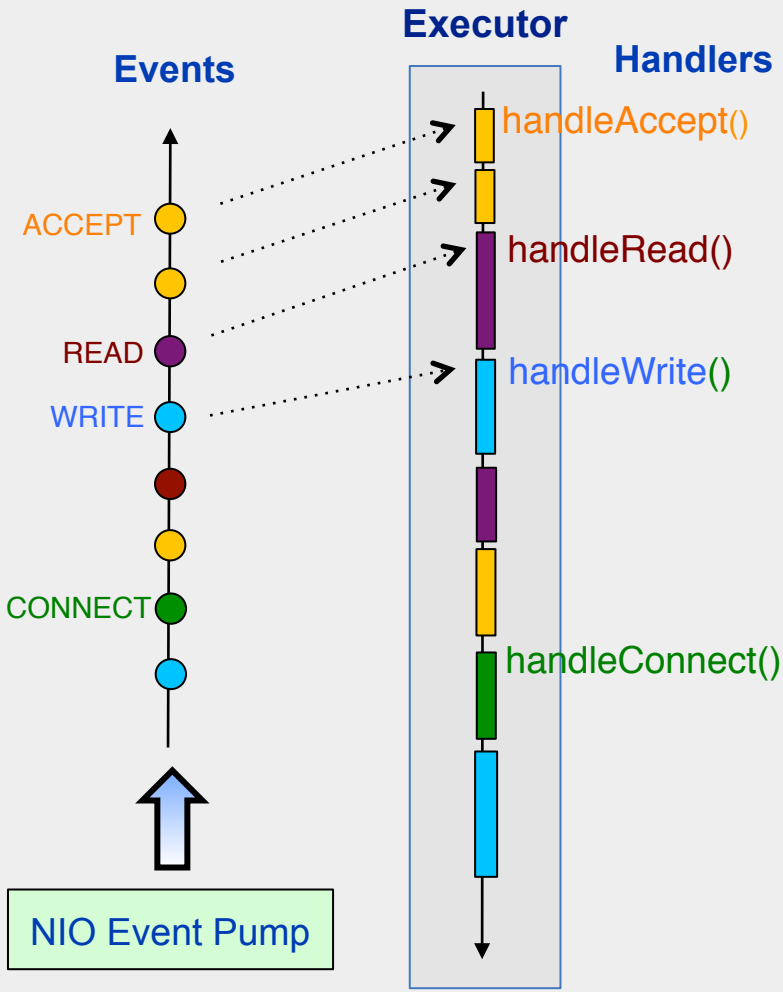
NIO event-based model

- **A single thread (called *executor*) manages all clients**
- **It spends its time executing non-blocking actions**



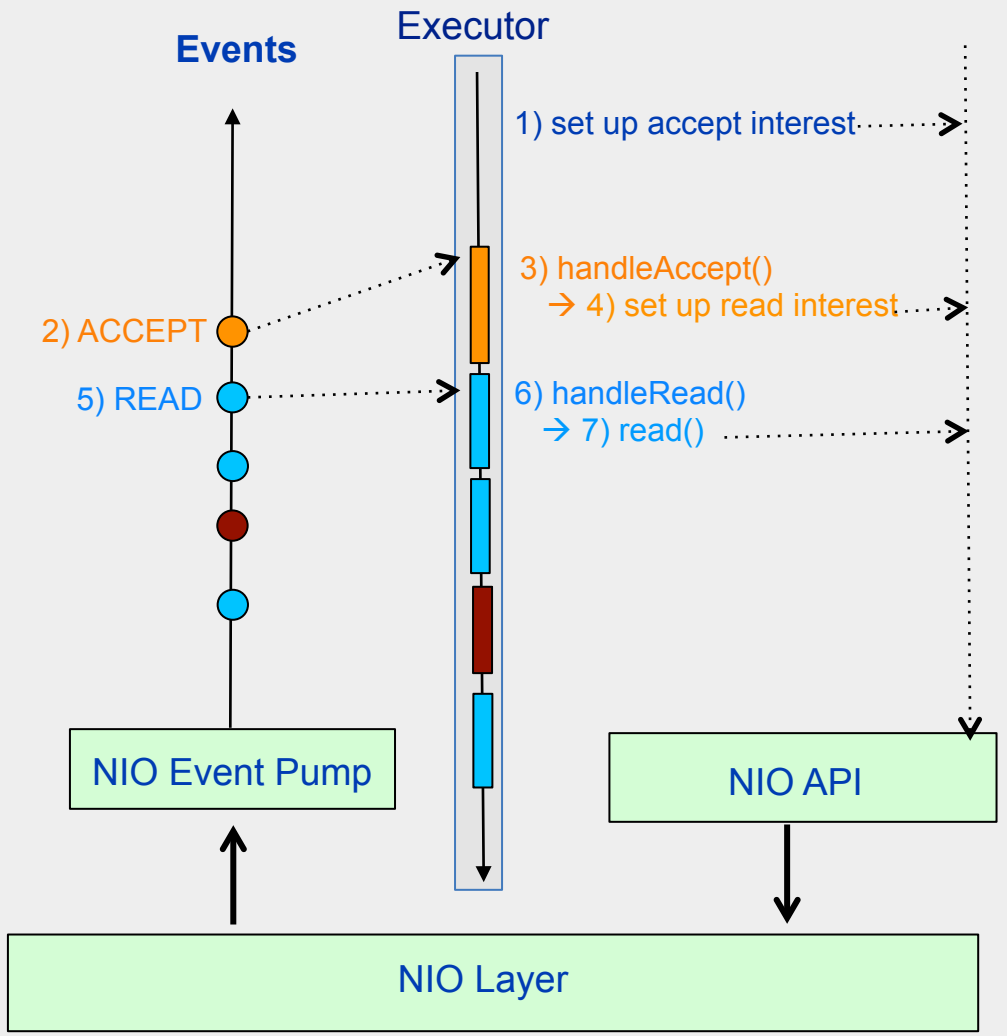
NIO event-based model

- **NIO triggers events**
  when things happen at the NW level
- **The executor listens to events**
  ACCEPT, CONNECT, READ, WRITE
- **For each event, the executor executes a dedicated handler (applicative code)**
  HandleAccept, HandleConnect, HandleRead, HandleWrite



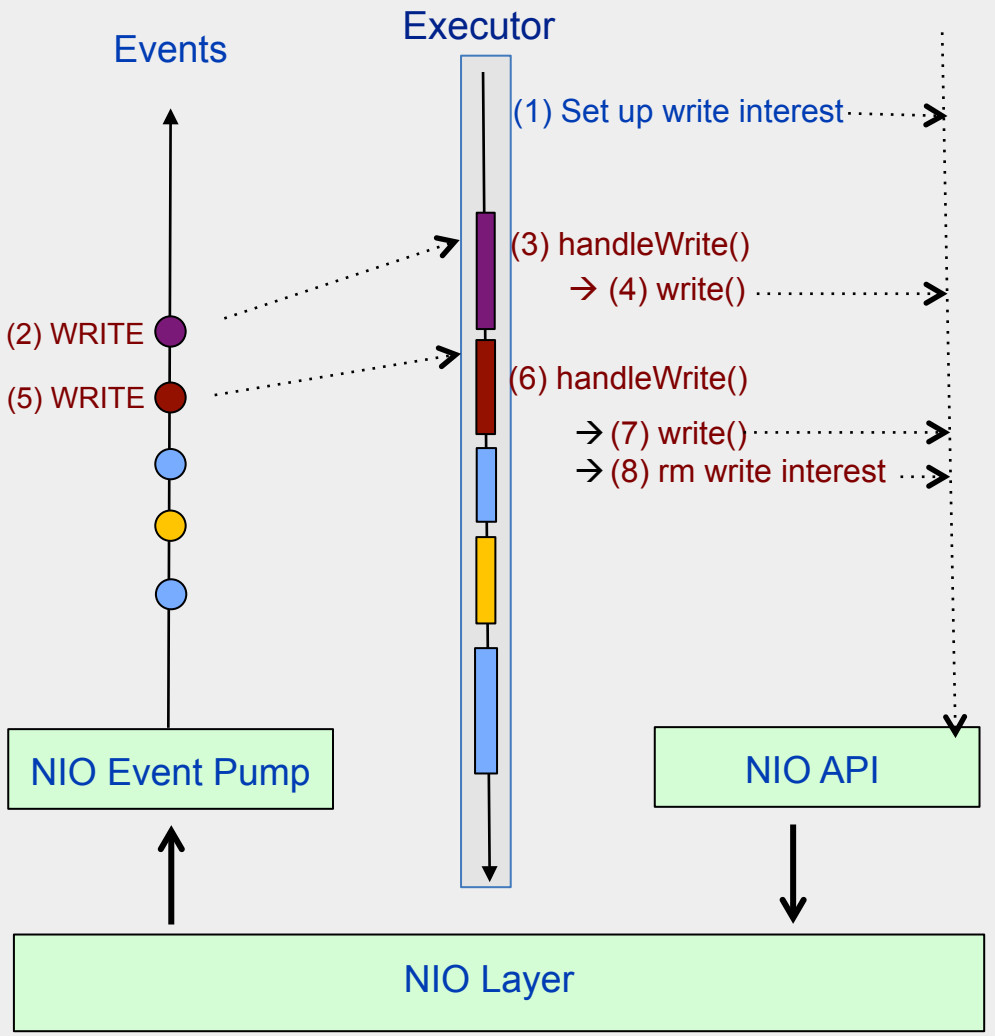
NIO event-based model

- **What can handlers do?**
  - Execute any sequence of instructions that does not block
  - Call the **NIO API**
- **NIO API (non blocking)**
  - to connect, accept, read, write
  - to setup events of interest
  - to request to connect (async)



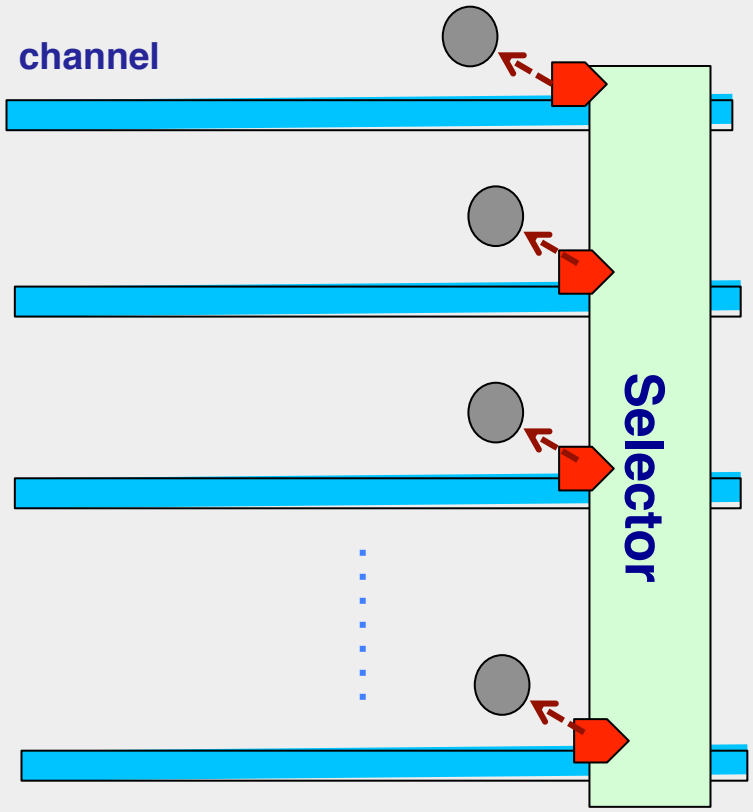
NIO event-based model

- **Consider writing a message**
  - Sets up write interest
  - When handling WRITE event
    - Write the bytes that can be written
    - If the full message has been writen, remove the write interest



NIO concepts & architecture

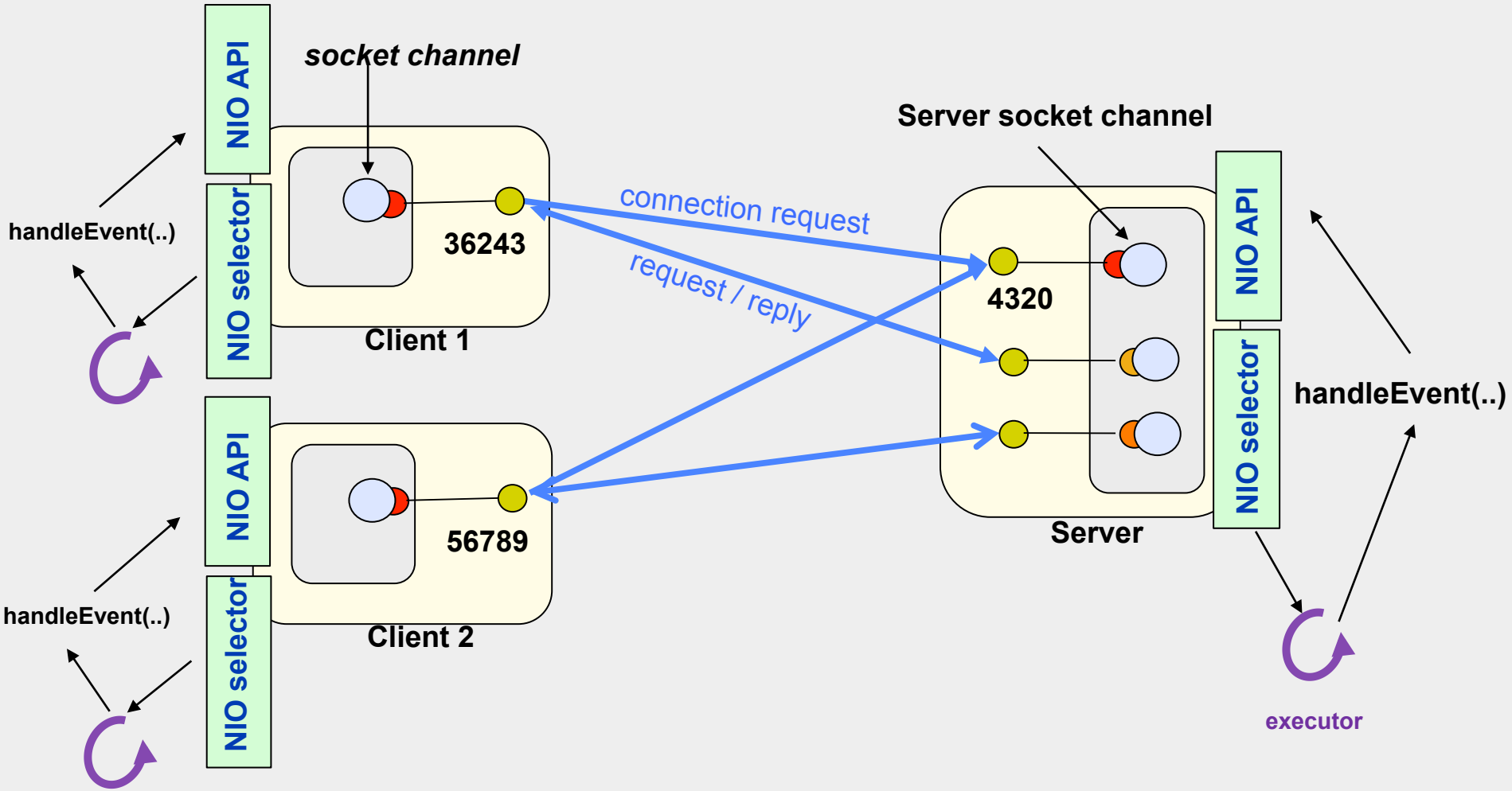
- **Socket channels**
  - For data transfers
  - Based on TCP channels
- **Sockets**
  - Based on TCP sockets
- **Selector**
  - **Event pump**, waits for events of interest on channels
- **Key**
  - A token representing the registration of a channel on the selector
  - Allows to attach a **data object**



NIO concepts & architecture

- **Selector use**
  - A selector allows a single thread to wait for events on multiple channels
    - select(), select(long timeout), selectNow()
  - Channels of interest should be registered on the selector
    - Register one or more channels
    - For each registered channel, get a selection key
    - Set interest on that key, any mix of ACCEPT, CONNECT, READ, WRITE
    - Interests on a key can be changed at any time
  - Typical use
    - Always have READ to be notified
    - Only have WRITE when there is something to send
    - Only have CONNECT when you just connected a socket
    - Only have ACCEPT when you have created server sockets

NIO concepts & architecture



NIO programming principles

Client side

```
public static void main() {
   // create an event pump
   sel = <create selector>

   // create client channel
   sc = < create client socket channel >

   // register the channel on the selector
   sel.register(cc);

   // set up interest for connect event
  <set connect interest on sc>

  // request to connect to server
  sc.connect(addr, port);

  // become the executor thread
  eventLoop();
}
```

Server side

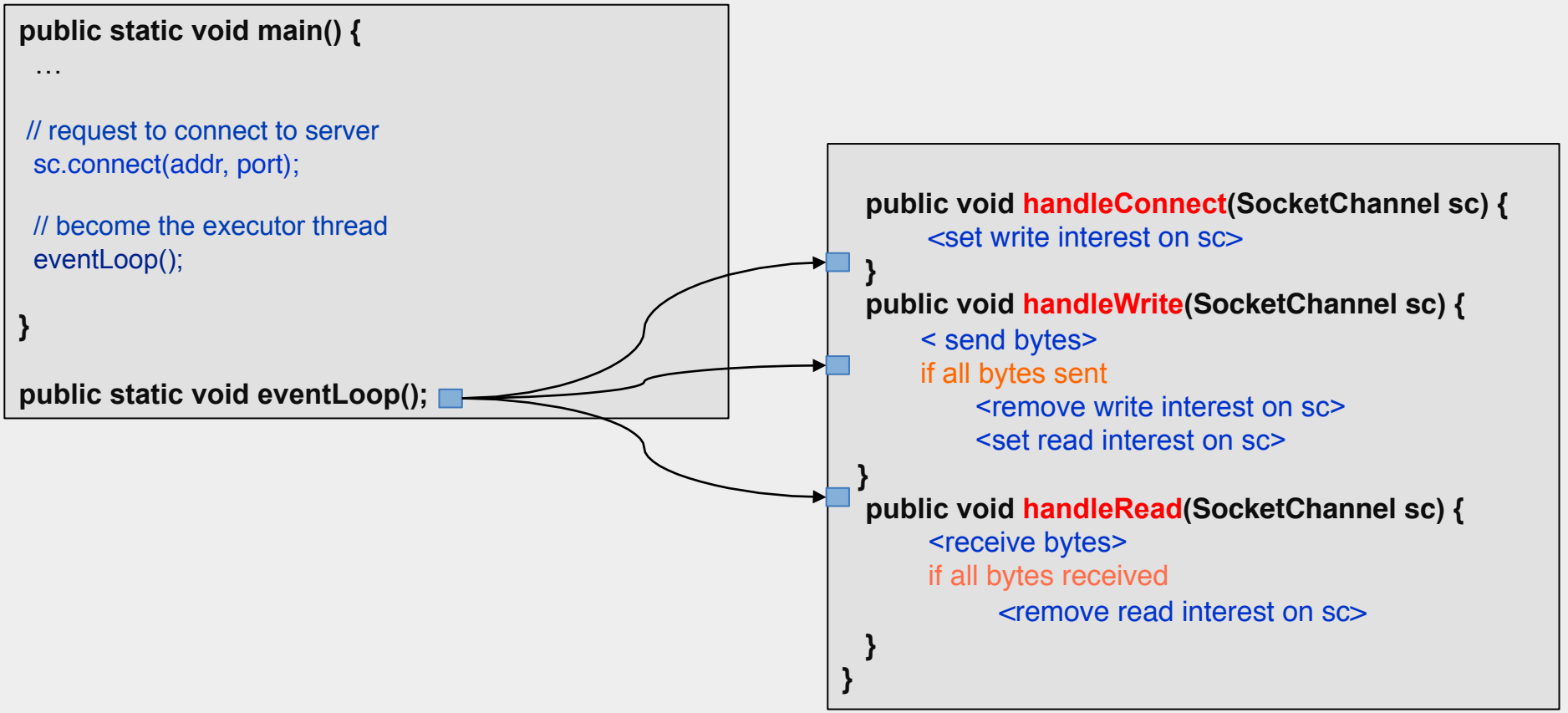
```
public static void main() {
   // create an event pump
  sel = <create selector>

   // create server channel
   ssc = <create server socket channel >

   // register the channel on the selector
   sel.register(ssc);

   // set up interest for accept events
   <set accept interest on ssc>

  // become the executor thread
  eventLoop();
}
```

Both sides

```
public static void eventLoop() {
  while (true) {
     Event e = <get next event from sel>
     switch(< type of Event e>){
                  case <CONNECT>:  handleConnect(..);
                  case <ACCEPT>:    handleAccept(..);
                  case <READ>:        handleRead(..);
                  case  <WRITE>:      handleWrite(..);
          default: ..
  }
```

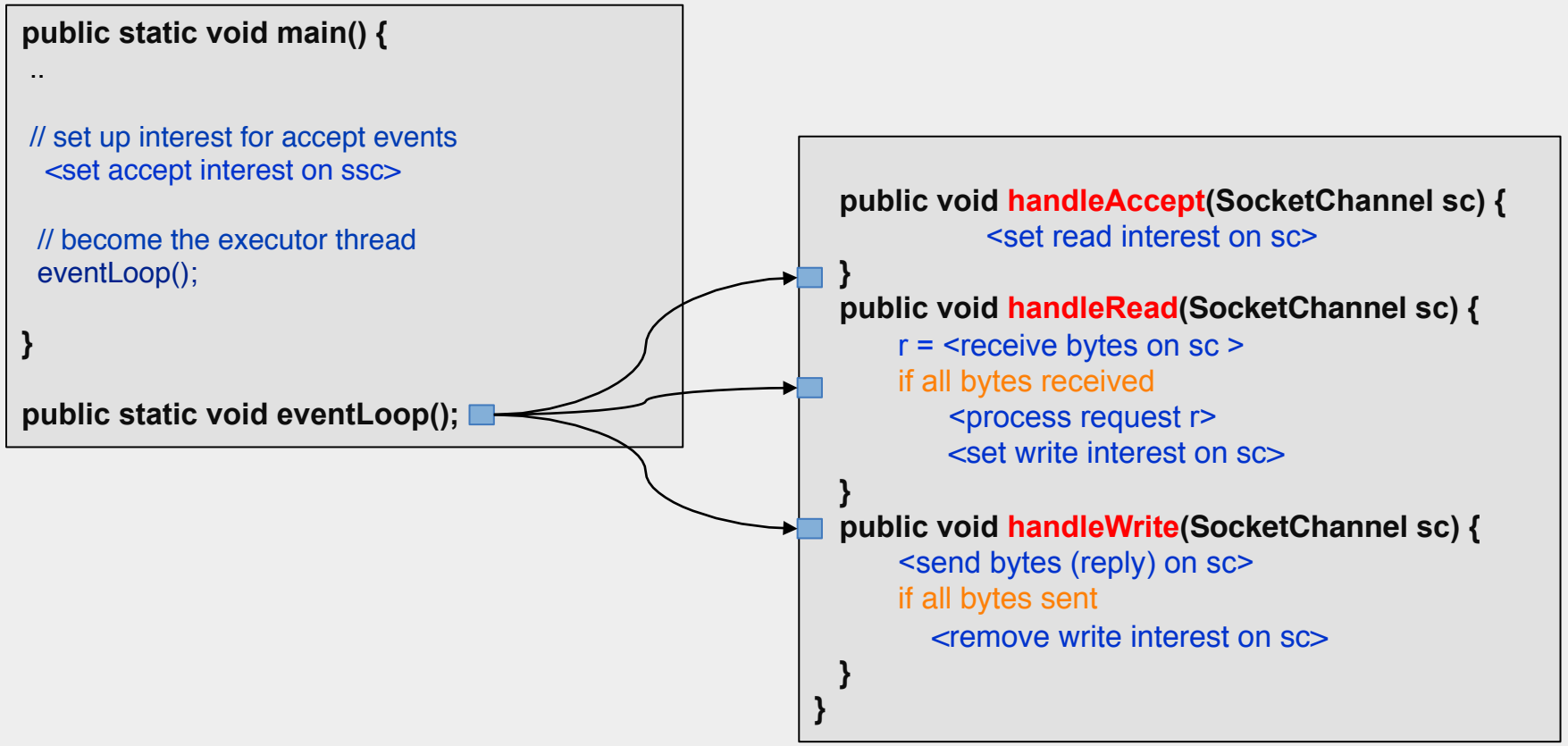
NIO programming principles

Typical Client-Server Interaction (client side)



NIO programming principles

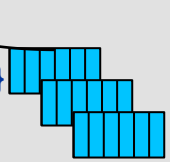
Typical client-Server Interaction (server side)



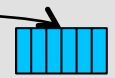
NIO programming principles

- **Program read/write handlers as finite state machines (automata)**
  - Writer automata
    - sendMsg: memorize the messages to sent in a list
    - handleWrite: send remaining bytes, as much as possible
  - Reader automata
    - handleRead: memorize received bytes, when a full message is received, process it

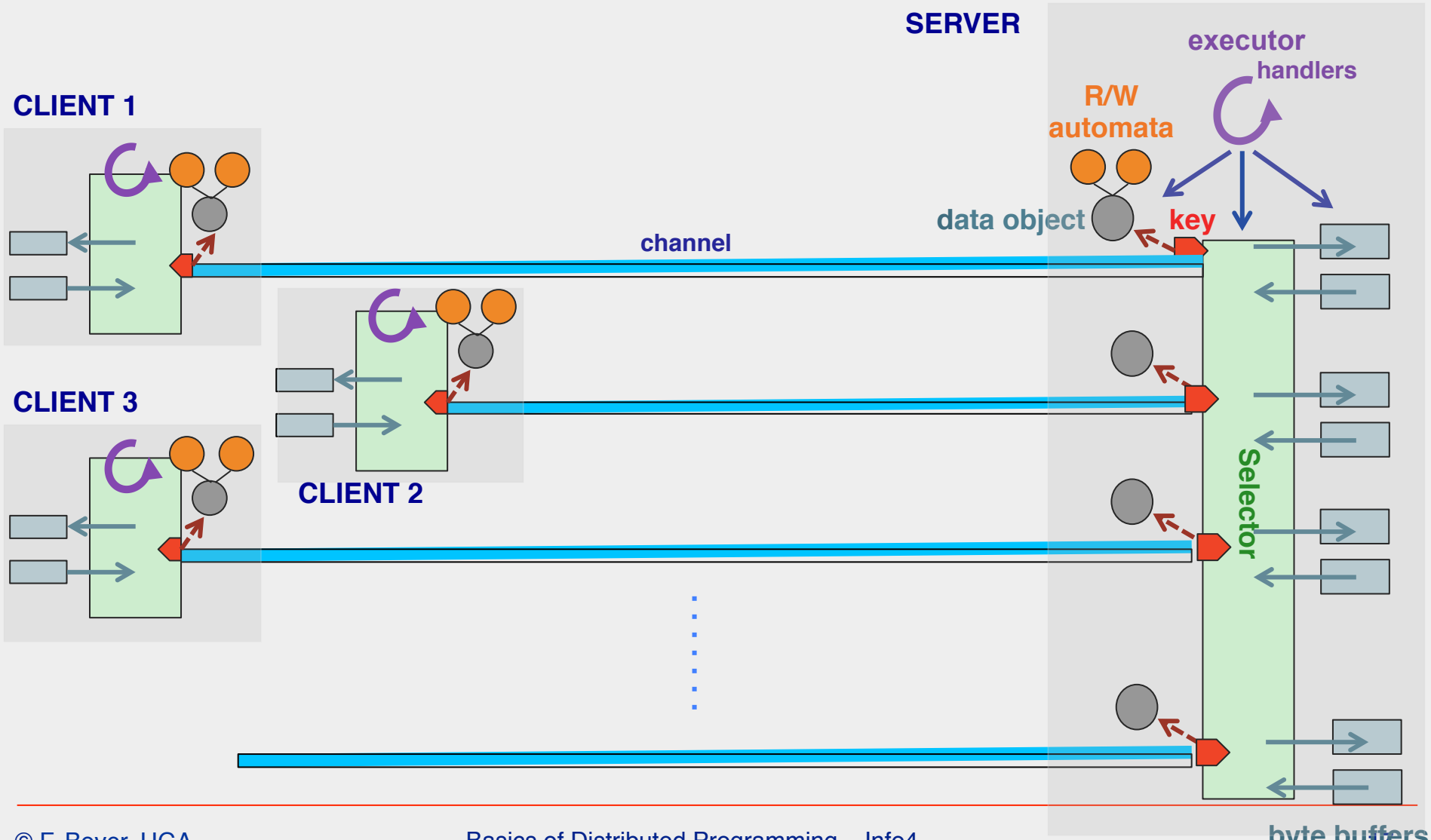
```
public void handleWrite(SocketChannel sc) {
      Writer wa = <get write-automata for sc>
       wa.handleWrite(..)
}

class WriterAutomata {
  …
      public void handleWrite(..) {..}
      public void sendMsg(byte[] msg) {..}
}
```



```
public void handleRead(SocketChannel sc) {
      Reader ra = <get read-automata for sc>
       ra.handleRead(..)
}

class ReaderAutomata {
  …
      public void handleRead(..) {..}
      public void processMsg(byte[]);
}
```

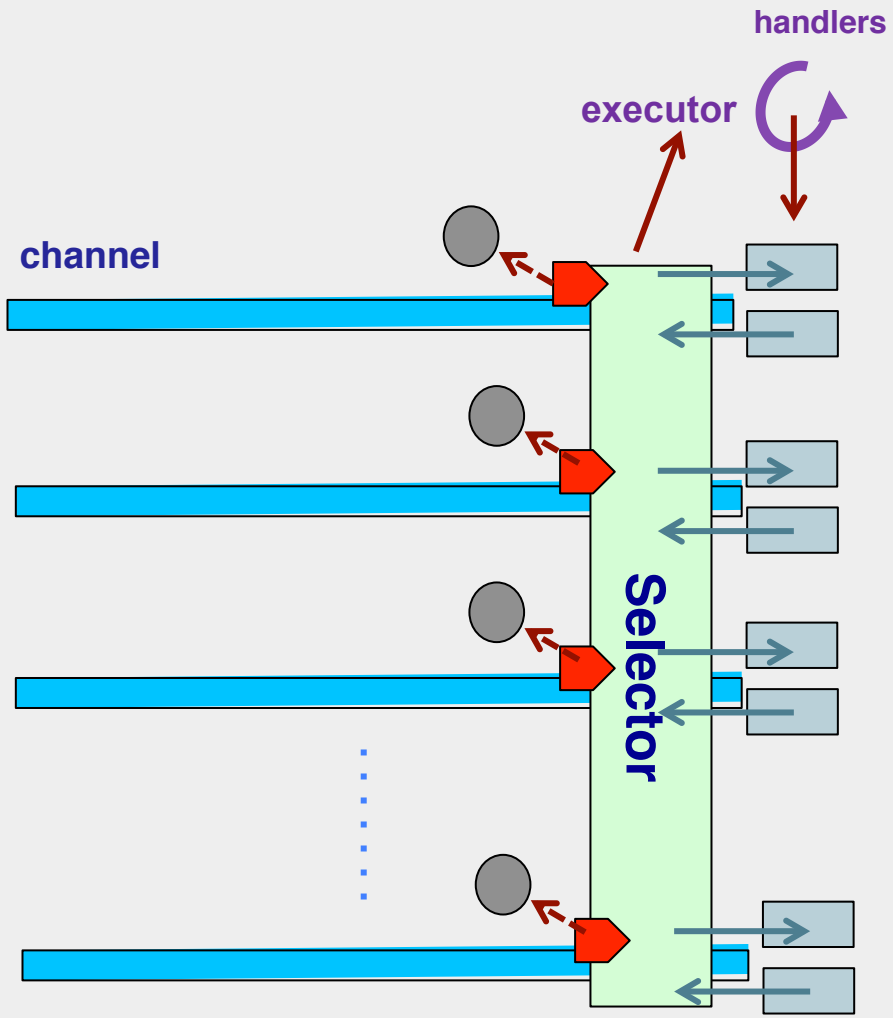


NIO – The global picture



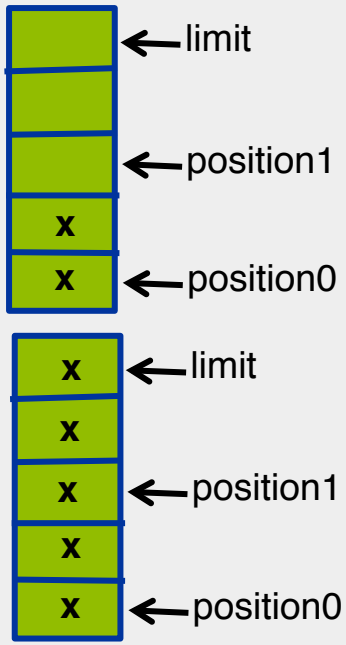
NIO Buffers

- **Channels**
  - For data transfers
- **Selector**
  - Wait for events of interest on several channels
- **Key**
  - A token representing the registration of a channel on a selector
  - Allows to attach a data object
- **ByteBuffer** 
  - Continuous extent of memory to read/write data

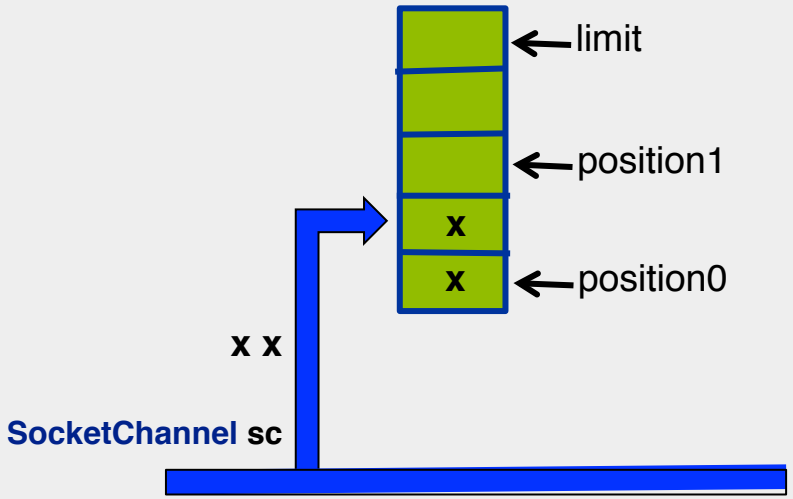


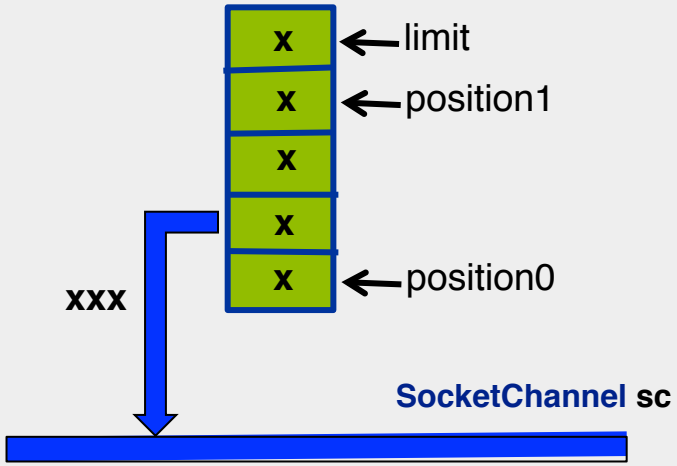
NIO ByteBuffers

- **Java NIO uses byte buffers (class ByteBuffer) rather than byte arrays**
  - Fixed capacity (ex: ByteBuffer b = ByteBuffer.allocate(5);)
  - Keeps offsets such as
    - position: current offset to read from / write to
    - limit: last offset to read / write bytes
    - remaining() {return limit – position;}
- **Non-blocking read and write operations**
  - Int Channel.read(ByteBuffer b)
    - Read bytes from NW and insert them at the current position
    - Increment the current position by the number of bytes read
    - Never exceeds the limit
  - int Channel.write(ByteBuffer b)
    - Write bytes to NW from the current position
    - Increment the current position by the number of bytes written
    - Never exceeds the limit



NIO ByteBuffers

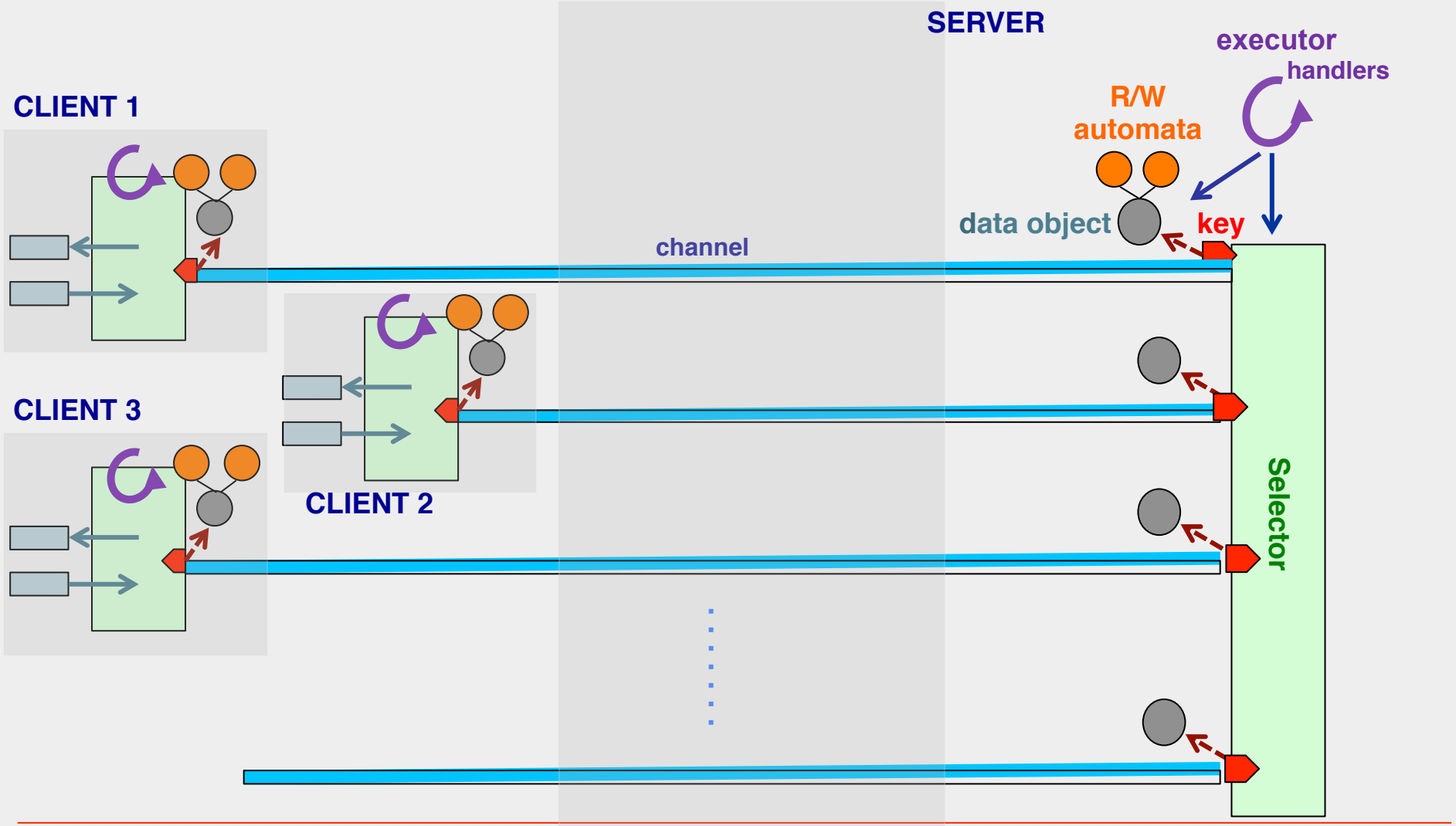





```
SocketChannel sc;
..
ByteBuffer buffer = ByteBuffer.allocate(5);
sc.read(buffer);
if (sc.remaining() == 0) {
  // all expected bytes have been received
  byte[] data = new byte[buffer.position()];
  buffer.rewind();
  buffer.get(data);
  buffer.rewind(); // if reuse buffer
  ...
```

```
SocketChannel sc;
  ..
 ByteBuffer buffer = ByteBuffer.wrap(
"hello".getBytes(Charset.forName("UTF-8")));
 sc.write(buffer);
 if (buffer.remaining() == 0)
     // all bytes have been sent
     ...
```

NIO – The global picture

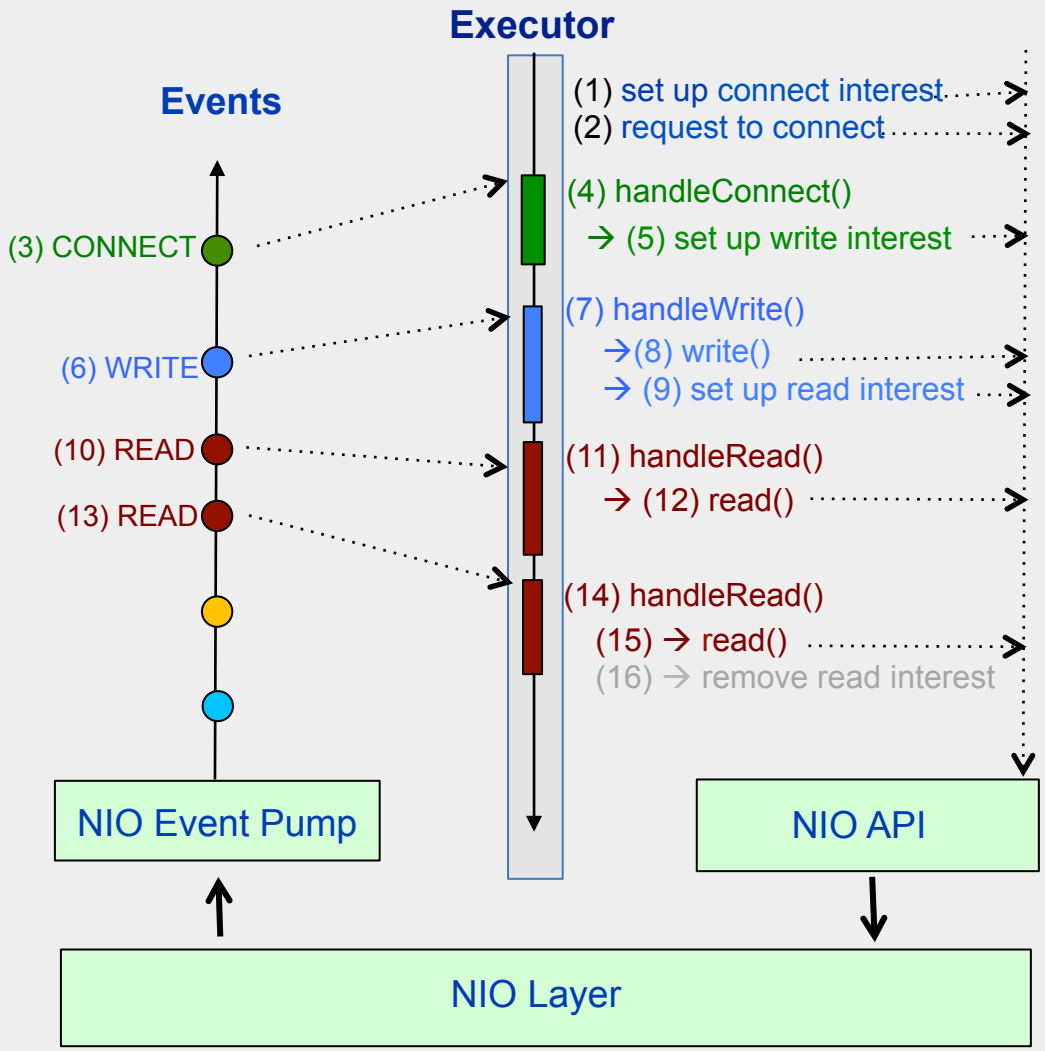


NIO – Work to do

- **Baby steps**
  - Step 1: observe
  - Step 2: program Reader and Writer automata
  - Step 3: support multiple clients at server side
- **NIO middleware**
  - Step 1: understand interfaces (read and observe the code)
  - Step 2: implement interfaces
  - Step 3: test your implementation with given applications

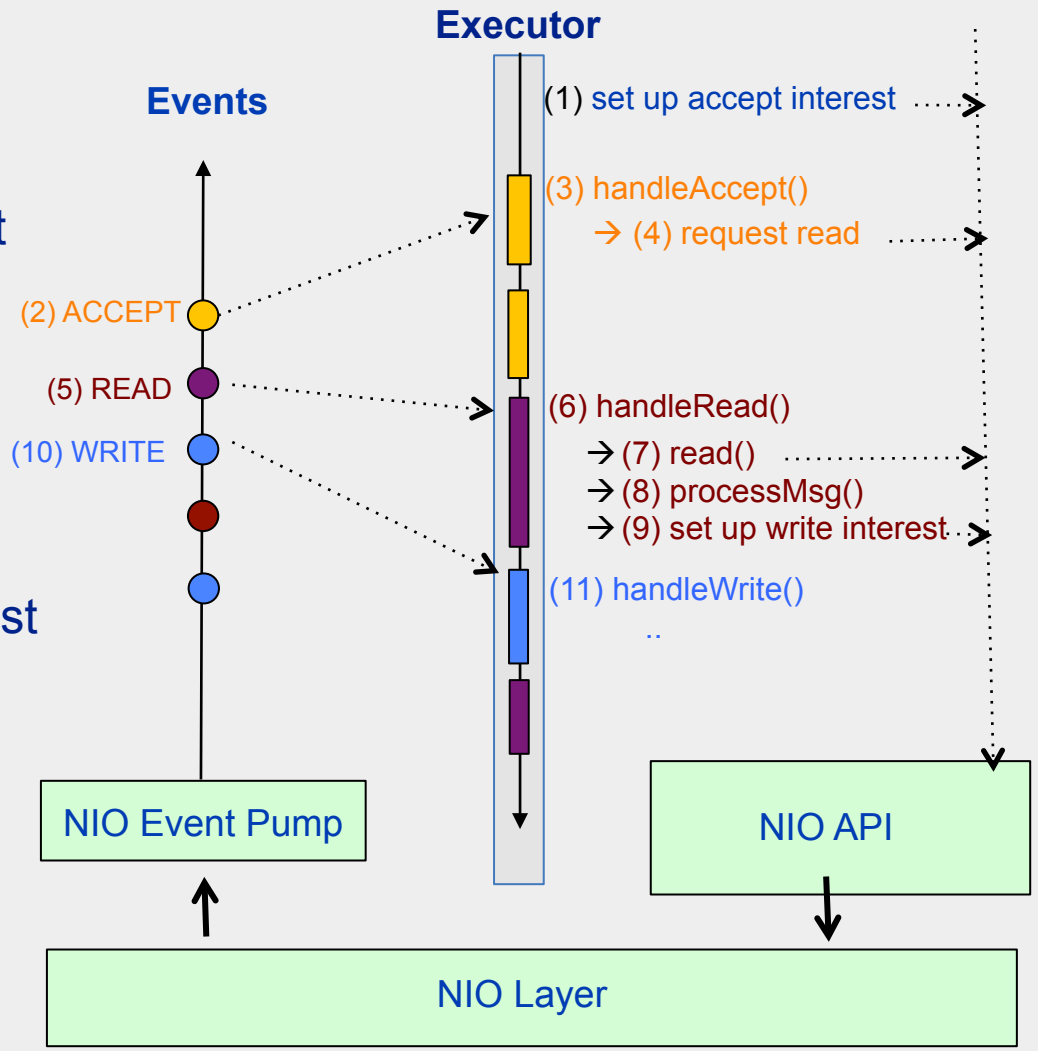
NIO event-based model

- **Client-side scenario**
  - Sets up connect interest & request to connect
  - When handling CONNECT, request to write
  - When handling WRITE, write bytes
  - If the full message has been sent, request to read
  - When handling READ, read the received bytes
  - If the full message has been read, ..



NIO event-based model

- **Server-side scenario**
  - Sets up accept interest
  - When handling ACCEPT, request to read
  - When handling READ, read the available bytes
  - If the full message has been received, process it and request to write
  - When handling WRITE, write bytes
  - …



NIO event-based model

- **Writing a message**
  - Sets up write interest & memorize the message to write
  - When handling WRITE event
    - Consider pending messages
    - Write as most bytes as possible
    - If all pending messages have been writen, remove write interest