Programming distributed systems with NIO

Fabienne Boyer, LIG
Université Joseph Fourier
2014-2015

Basic Socket IO

- Considering TCP
  - Lossless
  - Ordered
- Stream-oriented
  - Read one or more bytes at a time
- Blocking
  - *read()* blocks until there is some data to read
  - *write()* blocks until the data is written
  - **This generally requires one worker thread per socket (for reactivity)**
  - Not highly scalable

New Socket IO (Java NIO)

- Buffer-oriented
  - Data is read from a stream into a buffer
  - Data is written from a buffer to a stream

- **Non-blocking**
  - *read()* only reads the data that is currently available
  - *write()* only writes the data that can be currently written
  - No need for several threads dedicated to blocking read/write operations on seperate channels, a single thread (or a limited set of threads) can be used

New Socket IO

- Typical use
  - IO intensive applications
  - Mono-threaded systems
  - Communication groups

- **Event-based programming model**
  - One (or more) thread waits for events (IO events and possibly others)
  - Any event is associated with a handler (also called *callback* or *reaction*)

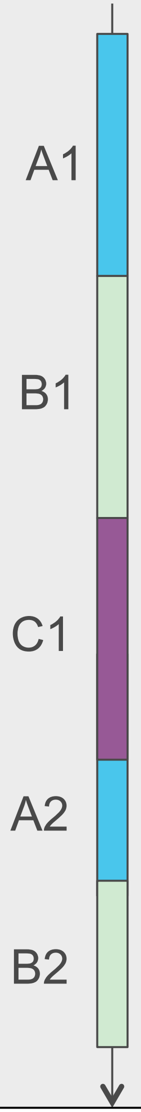
Event-based programming

Event-based

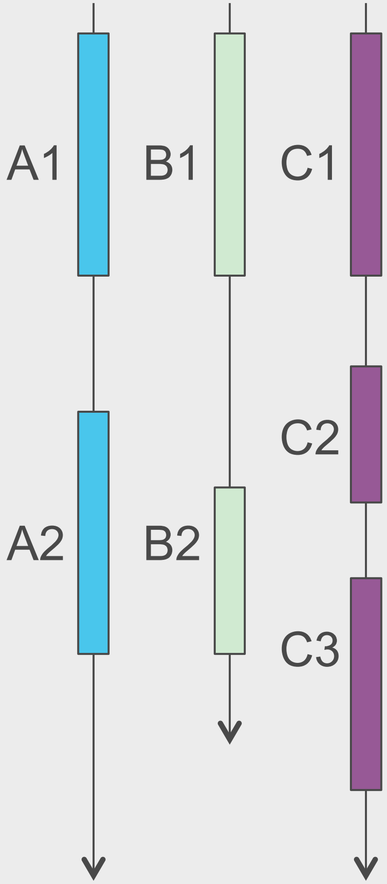
Use a single thread (scheduler thread)

The code is programmed in small chunks of code that do not contain blocking instructions

No useless commutations



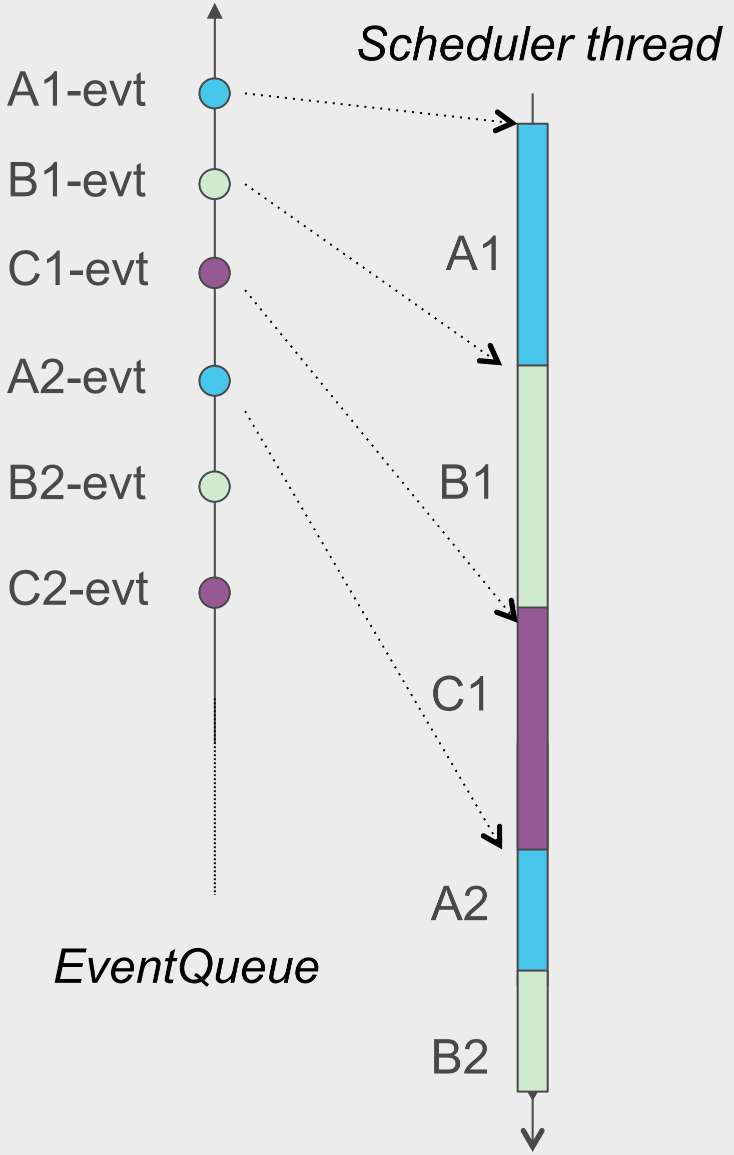
Thread-based



Event-based programming

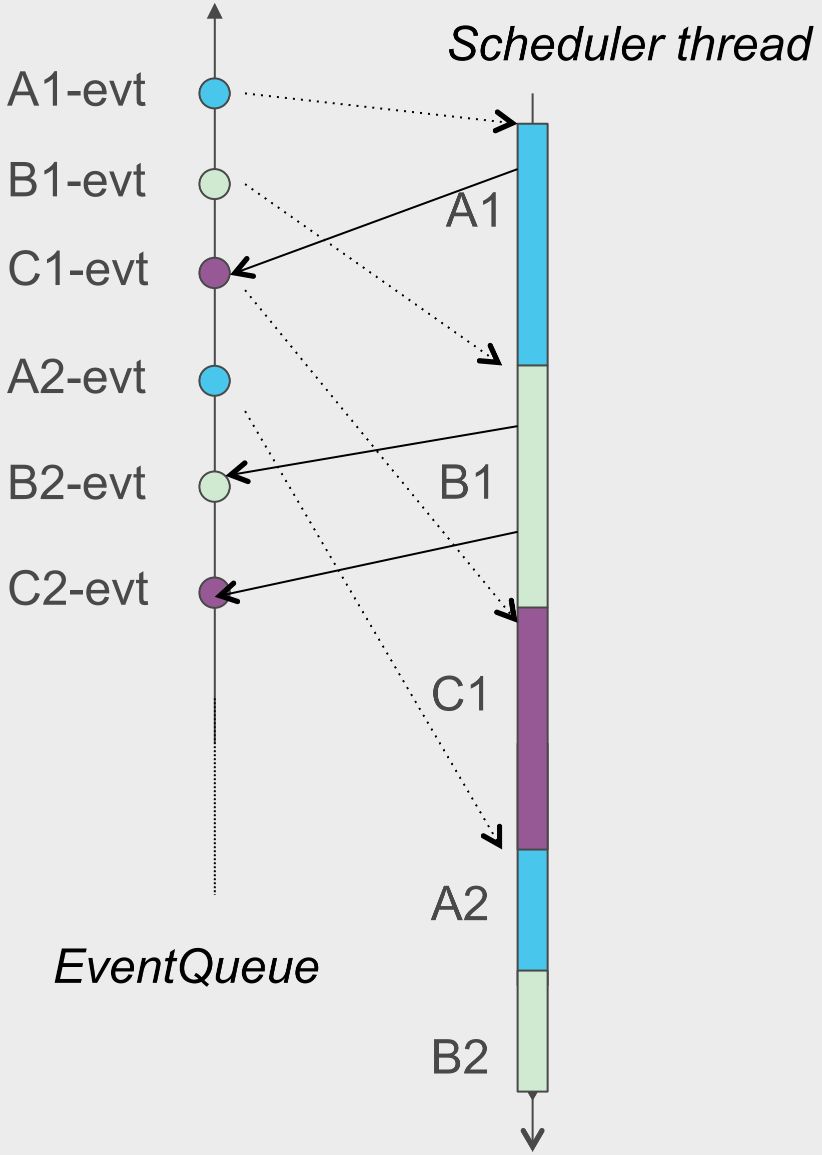
Events inform the scheduler of handlers that should be processed

The association between an event and an handler may be managed in several ways



Event-based programming

Events can be fired by low-level signals (e.g., incoming data over an IO channel) or by handlers



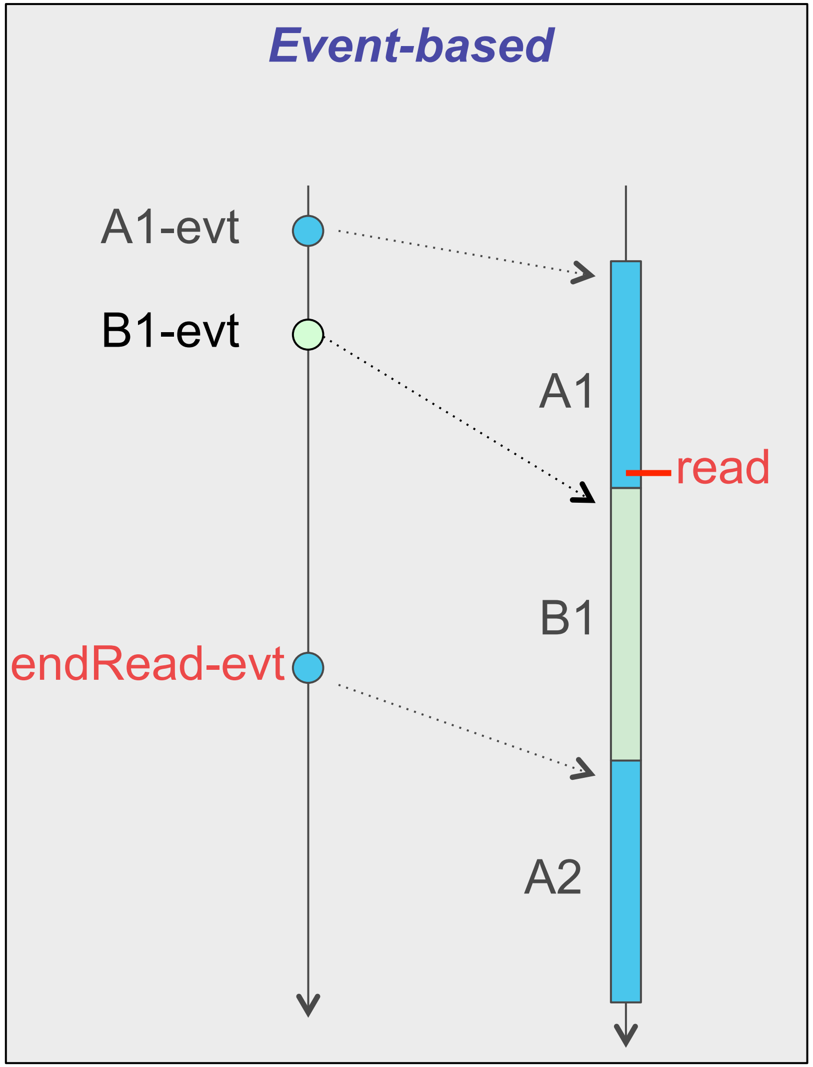
Event-based programming

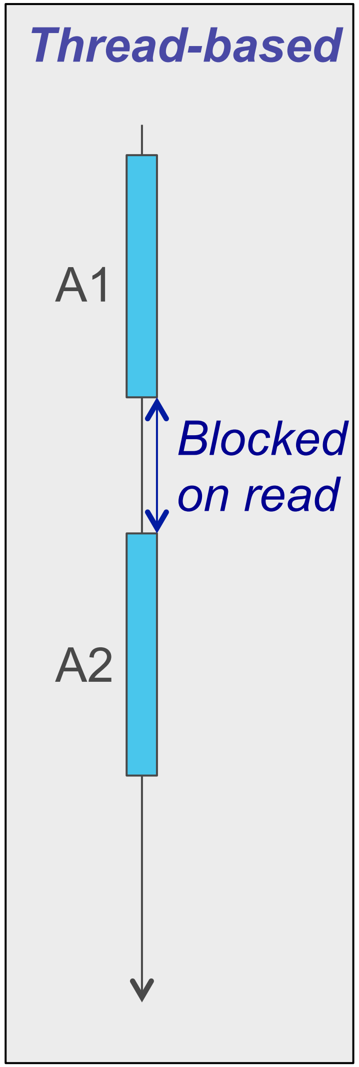
Focusing on task A

(thread-based)
Suppose that the time between A1 and A2 is due to a blocking *read* operation

(event-based)
We need a non-blocking read operation

Handling *endRead-evt* on task A leads to executing A2





Event-based programming

FIRST RULE:

A handler runs to completion, it should not block

→ *Note that blocking for a very short period of time may be allowed*

→ *Note that the scheduler may be multi-threaded for performance reasons but this is an implementation choice that should not change the programming model (except that reactions should be thread-safe)*

- So event programming does not work with classical sockets / streams
  - Sockets provide a blocking API (accept, connect, read, write)
  - Input and output streams are the same (read, write)

Event-based programming

SECOND RULE

Handlers may have to be programmed as Automata (Finite State Machin)

- When do we program a handler as an Automata?
  - When events do not always require the same treatment
  - The treatment depends on the previous events that were fired
  - Being given a current state and an event, the automata determines how the event should be processed
- We may use
  - A global automata for all events
  - An automata / event type (or per set of event types)
  - An automata / client
  - ..

The GUI example

- GUI programming
  - Main (scheduler) thread goes in the Widget toolkit event pump
  - Only reacts to graphical events through callbacks
  - Callbacks are never called concurrently
  - Possible to submit events for later execution
  - Usually use an automata per widget

NIO Programming Model

- NIO Events
  - ACCEPT, CONNECT, READ, WRITE
  - Fired by the NIO layer
- NIO Handlers
  - handleAccept, handleConnect, handleRead, handleWrite
- Use automata for
  - READ handler on a given channel, as we may need several read to compose a received message
  - WRITE handler on a given channel, as a single message may be sent in several times

NIO Programming Model

- Programming handlers
  - Do not block within handlers
  - So we cannot program as follow

```
handleRead(byte[] b) {
  int nbr = 0;
  int len = b.length
   while (nbr < len){
          nbr += read(b, nbr, len-nbr)
  }
}
```

```
handleWrite(byte[] b) {
Int nbw=0;
Int len = b.length;
   while (nbw < len){
          nbw += write(b, nbw, len-nbw);
  }
}
```

- Basically
  - For read events on a given channel, use an automata that keeps *b* and *nbr*
  - For write events on a given channel, use an automata that keeps *b* and *nbw*

NIO Concepts

- Channels

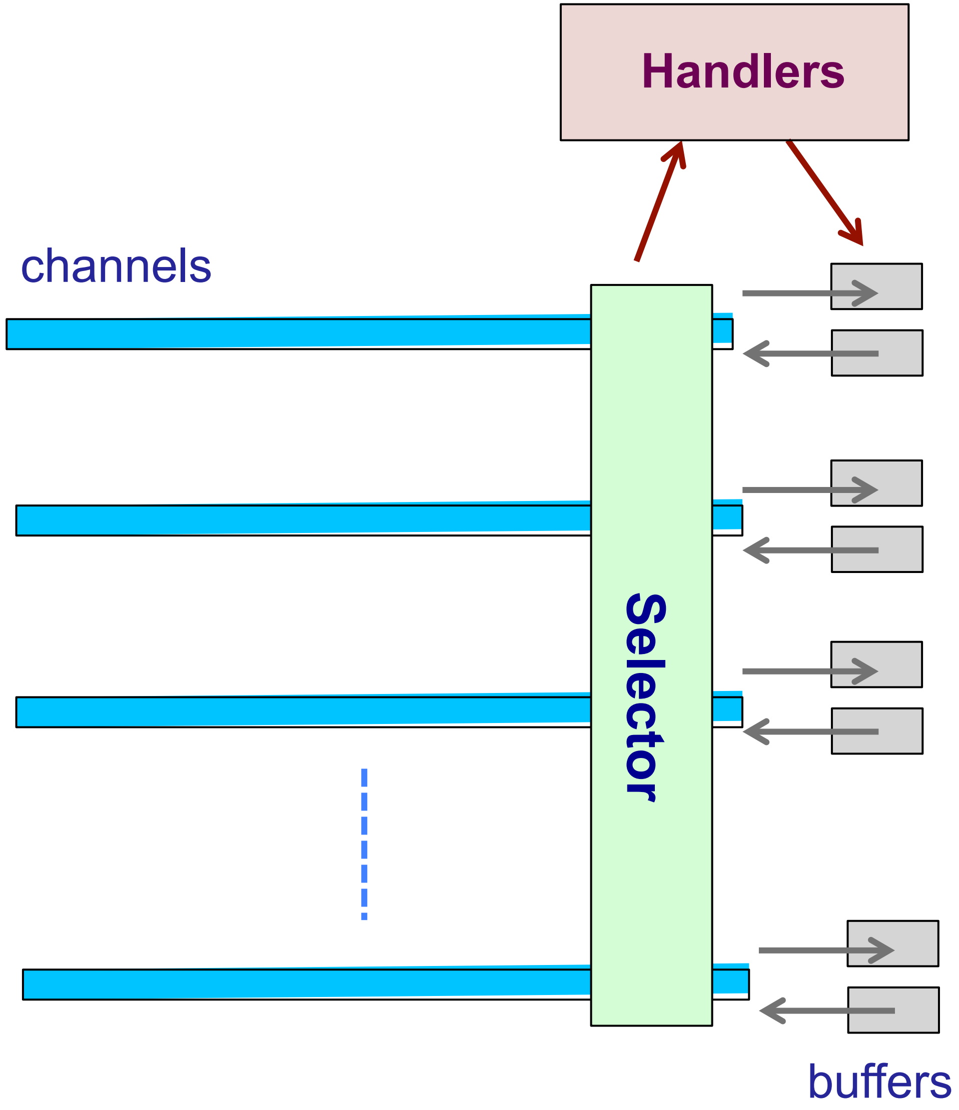
  Provide for data transfers between sockets and buffers

- Buffers

  Contiguous extent of memory for processing data

- Selector

  Allows to wait on several channels until one or more become available for data transfer



NIO Channels

- To send or receive data
  - Lossless and FIFO

- Channels are close to streams except that
  - Channels always read to (or write from) a Buffer
  - connect(), accept(), read(), write() operations on Channels are asynchronous

- Some Channel implementations
  - *SocketChannel*, to read and write data over the network via TCP
  - *ServerSocketChannel*, to listen for TCP connections
    - For each incoming connection, a *SocketChannel* is created.

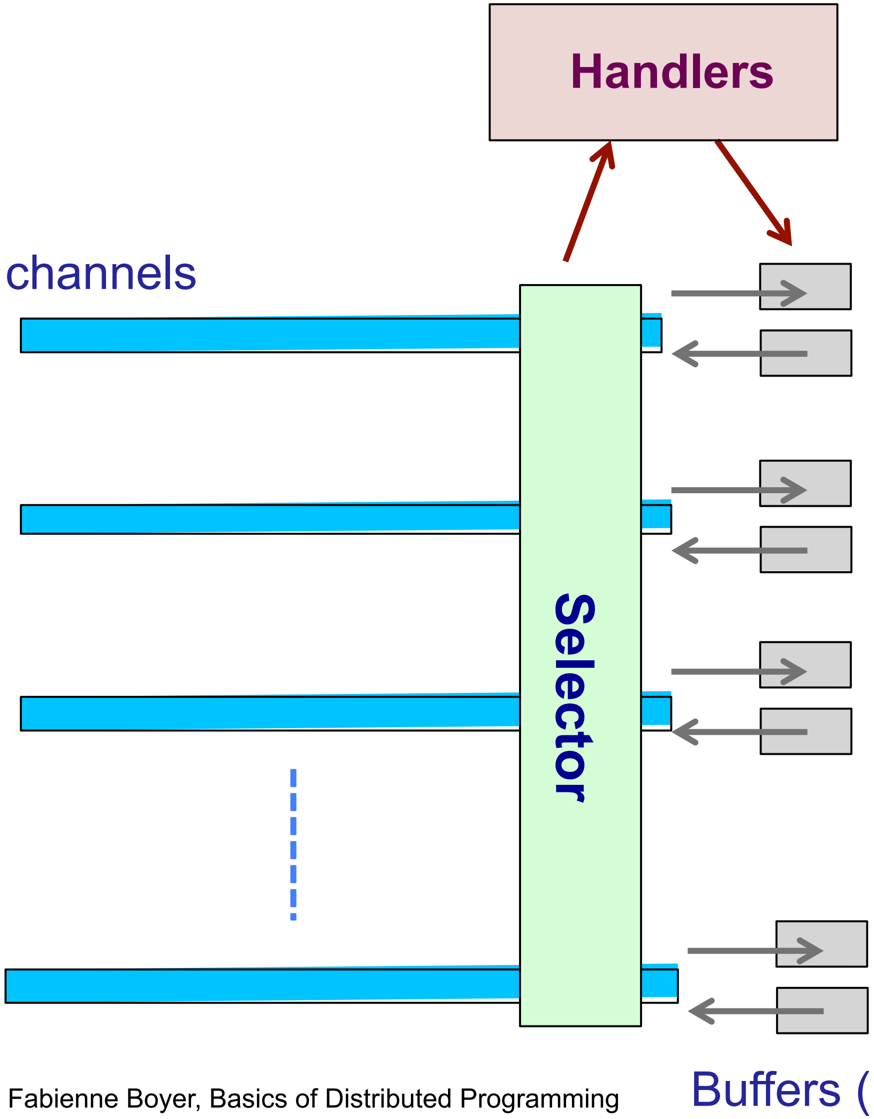
NIO Buffers

- To hold data, either received or to be sent
  - Not a stream, a random-access buffer of a given size
  - Non-blocking send (send what can be sent)
  - Non blocking receive (only receive what is currently available)
  - Keep the memory of read / write indexes
- Provide optimized operations
  - Use the physical memory of the under-laying operating system for native IO operations
  - No additional copies when transfers are processed
- Buffer attributes
  - Capacity (number of bytes)
  - Position (next index for read/write)
  - Limit (maximum number of bytes that can be read / written)

NIO Selector

- A selector allows a single thread to wait for IO 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



REMIND TO NEVER BLOCK IN HANDLER !!

Handle accept
register the created channel on the selector

Handle connect
register read/write interests

Handle read
read bytes from channel into InBuffer and process received data

Handle write
write bytes from OutBuffer into channel

NIO – Server programming

```
public class NioServer implements Runnable {
        private InetAddress address;
        private int port, length = …;
        private ServerSocketChannel serverChannel;
        private Selector selector;

      public NioServer(InetAddress hostAddress, int port) throws IOException {
          this.hostAddress = hostAddress;
          this.port = port;
          selector = SelectorProvider.provider().openSelector();

         // Create a new non-blocking server socket channel
         serverChannel = ServerSocketChannel.open();
         serverChannel.configureBlocking(false);
         serverChannel.socket().bind(new InetSocketAddress(hostAddress,port));

          // Be notified when connection requests arrive
         serverChannel.register(selector, SelectionKey.OP_ACCEPT);
         }

      public static void main(String[] args) {
          try { new Thread(new NioServer(null, 8888)).start();  } catch (IOException) ..
```

NIO – Server programming

```
public void run() {
          while (true) {
            try {
              // Wait for an event one of the registered channels
              this.selector.select();

              // Some events have been received
              Iterator selectedKeys = this.selector.selectedKeys().iterator();
              while (selectedKeys.hasNext()) {
                SelectionKey key = (SelectionKey) selectedKeys.next();
                selectedKeys.remove();
                if (!key.isValid()) { continue; }

                 // Handle the event
                 if (key.isAcceptable()){              handleAccept(key);
                  } else if (key.isConnectable()){ handleConnect(key);
                 } else if (key.isReadable()){      handleRead(key);
                 } else if (key.isWritable()) {      handleWrite(key);
          } } catch (Exception e) { … }}}
```

NIO – Client programming

```
public class NioClient implements Runnable {
        private InetAddress address;
        private int port, length = …;
        private SocketChannel clientChannel;
        private Selector selector;

     public NioClient(InetAddress hostAddress, int port) throws IOException {
          this.hostAddress = hostAddress;
          this.port = port;
          selector = SelectorProvider.provider().openSelector();
         // Create a new non-blocking socket channel
         clientChannel = SocketChannel.open();
         clientChannel.configureBlocking(false);
        // Be notified when connection is accepted
         clientChannel.register(selector, SelectionKey.OP_CONNECT);

        // Connect to the server
        clientChannel.connect(new InetSocketAddress(hostAddress, port));
         }

      public static void main(String[] args) {
          try { new Thread(new NioClient(null, 8888)).start();  } catch (IOException) ..
```

NIO – Client programming

```
public void run() {
          while (true) {
            try {
              // Wait for an event one of the registered channels
              this.selector.select();

              // Some events have been received
              Iterator selectedKeys = this.selector.selectedKeys().iterator();
              while (selectedKeys.hasNext()) {
                SelectionKey key = (SelectionKey) selectedKeys.next();
                selectedKeys.remove();
                if (!key.isValid()) { continue; }

                 // Handle the event
                 if (key.isAcceptable()){             handleAccept(key);
                  } else if (key.isConnectable()){ handleConnect(key);
                 } else if (key.isReadable()){      handleRead(key);
                 } else if (key.isWritable()) {      handleWrite(key);
          } } catch (Exception e) { … }}}
```

NIO - Connecting principles

- Connecting
  - Create a SocketChannel
  - Configure it to non-blocking
  - Configure it to TCP no delay
  - Register that channel on a selector, getting a key
  - Set the CONNECT interest
  - Start the connection operation (*SocketChannel.connect(InetSocketAddr)*)
- Later Connect Handling
  - When the key will be in the connectable state, the selector will unblock
  - The connection sequence must be finished (*SocketChannel.finishConnect())*
  - Set the READ interest on the key to receive data

NIO – Connect Handling

```
private void handleConnect(SelectionKey key) throws IOException {

   SocketChannel socketChannel = (ServerSocketChannel) key.channel();

        // finish the connection
      socketChanne.finishConnect();

     // register the read interest on the selector
      socketChannel.register(this.selector, SelectionKey.OP_READ);

      ..
 }
```

NIO - Accepting principles

- Accepting
  - Create a ServerSocketChannel
  - Configure it to non-blocking
  - Bind it to a port
  - Register it to a selector, getting a key
  - Set at least the ACCEPT interest
- Later Accept Handling
  - Upon accepting a connection, the selector will unblock
  - The key will be in the acceptable state
  - Get the socket channel from the key
  - Set it to non blocking and noTCP delay
  - Set the READ interest on the key to receive data

NIO – Accept Handling

```
private void handleAccept(SelectionKey key) throws IOException {


   ServerSocketChannel serverSocketChannel = (ServerSocketChannel) key.channel();

        // Accept the connection and make it non-blocking
        SocketChannel socketChannel = serverSocketChannel.accept();
        Socket socket = socketChannel.socket();
        socketChannel.configureBlocking(false);

        // Register the new SocketChannel with our Selector, indicating
        // we'd like to be notified when there's data waiting to be read
        socketChannel.register(this.selector, SelectionKey.OP_READ);
      }
```

NIO – Reading (basic version)

- Reading
  - Set the READ interest on the channel

- Later Read Handling
  - Read available bytes and deliver them immediately

NIO - Read Handling (basic version)

```
private void handleRead(SelectionKey key) throws IOException {
   SocketChannel socketChannel = (SocketChannel) key.channel();
   ByteBuffer inBuffer = ByteBuffer.allocate(length);   // Read up to length bytes

   int nbread = 0;
   try {
         nbread = socketChannel.read(inBuffer);
   } catch (IOException e) {
     // the connection as been closed unexpectedly, cancel the selection and close the channel
     key.cancel();
     socketChannel.close();
     return;
   }
   if (nbread == -1) {
     // the socket has been shutdown remotely cleanly
     key.channel().close();
     key.cancel();
     return;
   }
   // process the received data, being aware that it may be incomplete
  deliver(this, socketChannel, inBuffer.array(), nbread);
 }
```

NIO – Writing (basic version)

- Writing
  - Manage a write buffer per channel
  - Put any message to send in this buffer (possibly loosing part of the previous message if not entirely sent)
  - Set the WRITE interest on the channel
- Later writeHandler
  - Write as much as we can
  - Remove the WRITE interest if we wrote the entire buffer

NIO – Writing (basic version)

```
// outBuffers contains the data to write per channel
Hashtable<SocketChannel, ByteBuffer> outBuffers = …;

private void write(SocketChannel socketChannel, byte[] data) throws IOException {

    // we suppose that previous data in outBuffer have already been sent
    // or we do not mind loosing them
     outBuffers.put(socketChannel, ByteBuffer.wrap(data));

     // indicate we want to select OP-WRITE from now
     SelectionKey key = socketChannel.keyFor(this.selector);
     key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE }
```

NIO – Basic Write Handling

```
// outBuffers contains the data to write per channel
Hashtable<SocketChannel, ByteBuffer> outBuffers = …;

private void handleWrite(SelectionKey key) throws IOException {
    SocketChannel socketChannel = (SocketChannel) key.channel();
    ByteBuffer outBuffer = outBuffers.get(socketChannel);
     if (outBuffer.remaining() > 0) {
         try {
                socketChannel.write(outBuffer);
          } catch (IOException e) {

          // The channel has been closed
          key.cancel();
          socketChannel.close();
          return;
       }
     } else …
  }
```

Advanced Reading

- For any channel, we manage a read automata

  Hashtable<SocketChannel, ReadAutomata> readAutomata = …;

- Read automata
  - Implements the *handleRead(..)* method to gather received bytes as they become available
  - Knows that each message is prefixed with its length on 4 bytes
  - Only deliver *complete* messages

Read Automata

```
Class ReadAutomata {
  SocketChannel sock;
  ByteBuffer lenBuf = ByteBuffer.allocate(4);  // for reading the length of a message
  ByteBuffer msgBuf = null; // for reading a message
  static final int READING_LENGTH = 1;
  static final int READING_MSG = 2;
  int currentState = READING_LENGTH;  // initial state
  …

  private void handleRead() throws IOException {
   if (state == READING_LENGTH){
        int nb =  sock.read(lenBuf);
        …
        if (lenBuf.remaining == 0) {
            length = byteArrayToInt(lenBuf.array());
            msgBuf = ByteBuffer.allocate(byteArrayToInt(lengthBuf.array());
            lenBuf.position(0);
            state = READING_MSG;
        }
   }
   if (state == READING_MSG) {
         …
         if (msgBuf.remaining == 0){  // the message has been fully received
                  deliver(msgBuf.array());  // deliver it
                  msgBuf = null;
         } else …
```

Advanced Writing

- For any channel, we manage a write automata

  Hashtable<SocketChannel, WriteAutomata> writeAutomata =

- Write automata
  - Manage a FIFO queue of messages to send
  - Prefix each message with its length when sending it
  - So each buffer to send contains the length of the message on 4 bytes

Write Automata

```
Class WriteAutomata {
 SocketChannel sock;
 ArrayList<ByteBuffer> messages = new ArrayList<ByteBuffer>() ;  // messages to send
 ByteBuffer lenBuf = ByteBuffer.allocate(4);  // for writing the length of a message
 ByteBuffer msgBuf = null; // for writing a message
 static final int WRITING_LENGTH = 1;
 static final int WRITING_MSG = 2;
 int currentState = WRITING_LENGTH;  // initial state
 …
 private void handleWrite() throws IOException {
  if (state == WRITING_LENGTH){
      …
      if (lenBuf.remainig == 0) {
       state = WRITING_MSG;
       }
    } else if (state == WRITING_MSG) {
        if (msgBuf.remaining > 0){
              nb = sock.write(msgBuf);
        }
        if (msgBuf.remaining == 0){  // the message has been fully sent
                 if (! messages.isEmpty()){
                     msgBuf = messages.remove(0);
                    lenBuf.position(0);  lenBuf.putInt(0, buf.remaining());
                     state = WRITING_LENGTH;
                 } else STATE = WRITING_DONE;
        …
```

Write Automata

```
Class WriteAutomata {
  SocketChannel sock;
  ArrayList<ByteBuffer> messages = new ArrayList<ByteBuffer>() ;  // messages to send
  ByteBuffer lenBuf = ByteBuffer.allocate(4);  // for writing the length of a message
  ByteBuffer msgBuf = null; // for writing a message
  static final int WRITING_LENGTH = 1;
  static final int WRITING_MSG = 2;
  int currentState = WRITING_LENGTH;  // initial state
  …
  private void handleWrite() throws IOException {
         <see previous slide>
 }

 private void write(ByteBuffer msg) {
   messages.add(msg);
   if (state == WRITING_DONE)
         key.interestOps(SelectionKey.OP_READ I SelectionKey.OP_WRITE);
   }
 }
```

NIO – Work to do

- PingPong NIO server
  - A server waits for
    - Connection requests from clients
    - Messages sent by connected clients
    - The server returns to each client the message it sent
  - Once connected, a client spends its time sending textual messages and receiving the server’s response

  - First version: without automata management
  - Second version: with automata management