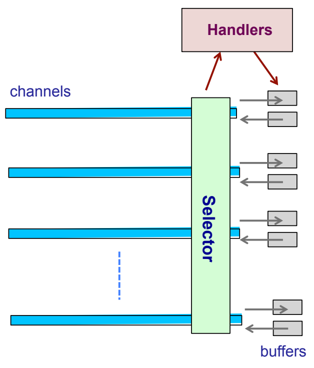
NIO Concepts

- 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 – 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
         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 – Connect Handling

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

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

       // finish establishing the connection
        socketChanne.finishConnect();

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

        ..
  }
```

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();
      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 - 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)

```
// 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 …
   }
```

NIO Buffers

- Attributes
  - Position (next index for read/write)
  - Limit (maximum number of bytes that can be read / written)
- Useful methods
  - **remaining()** *always returns the value of limit - position*
  - ByteBuffer buf = ByteBuffer.wrap (byte[] b)
    assigns b as the buffer content, set (write) position to 0 and limit to b.length
  - socketChannel.write(buf)
    send what can be send, update (write) position
  - ByteBuffer buf = ByteBuffer.allocate(128);
    set (read) position to 0 and limit to 128
  - socketChannel.read(buf)
    read what can be read, update (read) position
  - Byte[] b = buf.array()
    assigns b with the buffer content
  - buf.clear()
    set (read / write) position to 0

Advanced Reading & Writing

- For any channel, we manage a read automata
  Hashtable<SocketChannel, ReadAutomata> readAutomata = …;
- Read automata
  - Implements *handleRead(..)* to gather received bytes as they become available
  - Knows that each message is prefixed with its length on 4 bytes
  - Only deliver *complete* messages
- 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