# COMP202-08B
# Computer Communications

Lecture 20

Java – Asynchronous I/O, Part Two

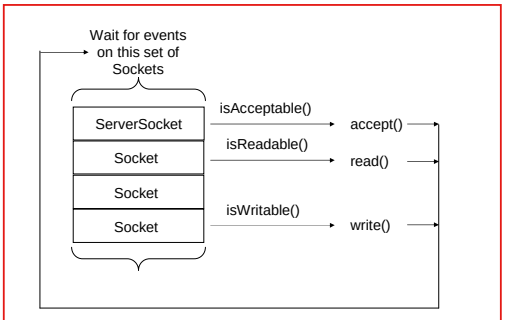


## Event-based Network Applications

- Similar to programming model used when programming GUI applications
  - wait for user to click a button, then do something in response
- Network applications:
  - wait for data to arrive, then do something
  - wait for a new connection to be established, then do something
  - wait for buffer space to come available, then write data
- Faster than spawning threads for each socket:
  - no process scheduling, synchronisation overheads



## Event-based network applications






## Putting this together : a starting point

```
Selector selector = Selector.open();
ServerSocketChannel ssc;
SocketChannel socket;
SelectionKey ssck;

ssc.socket().bind(new InetSocketAddress(1234));
ssc.configureBlocking(false);
ssck = ssc.register(selector, SelectionKey.OP_ACCEPT);

while(true) {
      selector.select();
      Set set = selector.selectedKeys();

      if(set.contains(ssck)) {
            if(ssck.isAcceptable()) {
                  socket = ssc.accept();
                  addClient(socket);
            }
            set.remove(ssck);
      }
}
```



## Putting this together : a starting point

```
while(true) {
      selector.select();

      /* code to accept new sockets removed */

      /* get all of the other events and loop through */
      Iterator it = set.iterator();
      while(it.hasNext()) {
            /* get the key for this event */
            SelectionKey key = (SelectionKey)it.next();

            /* remove the key from the selector */
            it.remove();

            /* handle event */
            if(key.isReadable())
                  handle_read();
            if(key.isWritable())
                  handle_write();
      }
```



## This lecture

Have not shown you a complete application that uses select yet, so today is about putting this all together to get a useful network application

## Problems to overcome: #1

- Consider a ChatServer, with a main room:
  - When someone says something in the room, it helps to know who said it, without requiring the person saying it to identify themselves when they speak
  - Socket connections roughly map to a person.
- When an event occurs on a SocketChannel, we need to know who it is in relation to.
  - i.e. we need to map a SocketChannel to a ChatServerSession
  - Remember which SocketChannel corresponds to which ChatServerSession
  - "keeping state"



## Problems to overcome: #2

- Consider a HelloWorldServer, where the server echoes back to the client the line they said
  - Non-blocking I/O means the Selector will tell you when there is something to read, which may not actually be a complete line yet.
  - SocketChannel::read method deals in bytes, not lines.
- For each client, we need to be prepared to read a partial line, and store it until the rest of the line arrives
  - i.e. we need to keep a buffer with each client that remembers (stores) anything partially read
  - "keeping state"



## Problems to overcome: #3

- Consider a HelloWorldServer, where the server echoes back to the client the line they said
  - Non-blocking I/O means the Selector will tell you when you are able to write something, but not how much you can actually write
  - When you perform a write, some bytes supplied are likely to be sent, but not necessarily all of them
- For each client, we need to be prepared to send only part of our reply, and store the unsent portion until we are able to send it
  - i.e. we need to keep a buffer with each client that remembers (stores) anything that has not been transmitted yet
  - "keeping state"

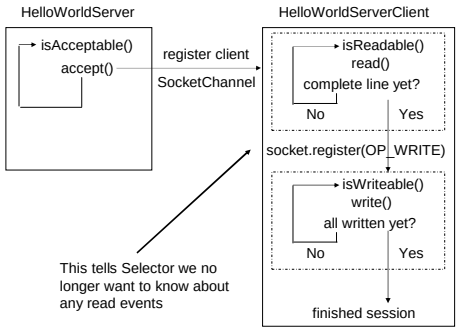


## Keeping state

- Essentially
  - remembering where you are up to
  - What are the set of valid things that can happen next?



## HelloWorldServer3: state transitions






## Solution #1: keeping state

- When a SocketChannel is created …
  - i.e. when you accept a new SocketChannel connection
- …it is natural to create a HelloWorldServerClient or ChatServerSession object to associate with the SocketChannel

## Solution #1: keeping state

```
/*
 * accept a new connection, and create a HelloWorldServerClient to
 * go with the connection.  Register the SocketChannel with the selector,
 * asking to be told when we can read from the Socket.
 */
SocketChannel channel = ssc.accept();
HelloWorldServerClient client = new HelloWorldServerClient(this);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);

/*
 * associate the session variable with the SelectionKey.
 * when the SelectionKey tells us we can read or write, we can get
 * the attachment (HelloWorldServerClient) associated with the key
 * and thus be able to keep state.
 */
key.attach(client);
```



## Solution #1: keeping state

```
/* go through the remainder of the events */
Iterator it = set.iterator();
while(it.hasNext()) {
   SelectionKey key = (SelectionKey)it.next();
   it.remove();

   /* retrieve the client attached to the SelectionKey */
   client = (HelloWorldServerClient)key.attachment();

   /* now, have the SelectionKey and the state associated with it */
```



## Solution #1: keeping per-session state

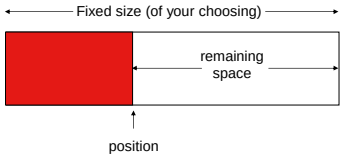
```
class HelloWorldServerClient
{
        HelloWorldServer3       server;
        private String          read;
        private byte[]          write;

        /* other state variables */
}
```

Put various bits of information that need to be kept per-session in the per-session structure



## Problems to overcome: #4

- Programming Java Selector and SocketChannels is more painful than it needs to be
  - ByteBuffer




To transmit, you write into the buffer, then pass it to write()
To receive, you pass a ByteBuffer with space in it to read(), then parse it



## ByteBuffer: read

```
buffer = ByteBuffer.allocate(512);          <-- Allocate a ByteBuffer

if(key.isReadable()) {
        buffer.rewind();                    <-- shift position to zero

        int len = key.channel().read(buffer);   <-- do the read
        if(len == -1) {                     <--
                removeClient(client);
                continue;
        }                                   The ByteBuffer position has
                                            been shifted with the read

        byte[] array = new byte[len];       <-- allocate a new array
        buffer.rewind();                    <-- position back to zero
        for(int j=0; j<len; j++) {
           array[j] = buffer.get();         <-- copy the data in
        }
        client.handle_read(array, len);     <-- pass data to client
  }
```



## Solution #2: remembering read() data

- For each client, we need to be prepared to read a partial line, and store it until the rest of the line arrives
  - i.e. we need to keep a buffer with each client that remembers (stores) anything partially read

**Note: will not go into details of readLine, please see code on Moodle**

## Solution #2: remembering read() data

In class HelloWorldServerClient

```
/* class variable we use to keep partially read strings in */
private String read;

public void handle_read(byte[] array, int len)
{
  /* create a string from the byte array */
  String str = new String(array, 0, len);

  /* concat with any existing string portion */
  this.read = read.concat(str);

  /* internal method to extract a line */
  str = this.readLine();
  if(str != null) {
        /* got a line, do something with it */
  }
}
```



## Solution #2 : deal with read() data

In class HelloWorldServerClient

```
if(str != null)
{
        /* got a line, buffer what we need to send back to the client */
        this.write("You said: " + str + "\r\n");

        /*
         * tell the Selector to let us know when we can write back
         * note: will no longer get read events, but could do so with
         * (OP_WRITE | OP_READ)
         */
        key.interestOps(SelectionKey.OP_WRITE);
}
```



## Solution #3: buffering write() data

- For each client, we need to be prepared to send only part of our reply, and store the unsent portion until we are able to send it
  - i.e. we need to keep a buffer with each client that remembers (stores) anything that has not been transmitted yet



## Solution #3: buffering write() data

In class HelloWorldServerClient

```
private byte[]          write;
private ByteBuffer      writebuf;

private void write(String str)
{
        byte[] strb = str.getBytes(); /* convert the string to an array of bytes */

        /* if there is already data buffered, extend that array and add to it */
        if(write != null) {
           byte[] newb = new byte[write.length + strb.length];
           System.arraycopy(write, 0, newb, 0, write.length);
           System.arraycopy(strb, write.length, newb, 0, strb.length);
           write = newb;
        } else {
           /* else, no data buffered, just use the byte array generated */
           write = strb;
        }
}
```



## Solution #3: buffering write() data

- HelloWorldServerClient::handle_write()
- Receives notification when a write may be used

- Implemented in three parts
  - First part checks the **byte[] write** variable to see if there is anything new to be sent via the ByteBuffer. If there is, add it to the ByteBuffer
  - Second part writes the actual data
  - Final part checks to see if there is anything left to write in either the **write[]** or **writebuf** variables. If nothing left, close the session



## Solution #3: buffering write() data: Part 1

```
        /* if there is new buffered data, at it to the ByteBuffer */
        if(write != null) {
           while(i<write.length && writebuf.hasRemaining()) {
                writebuf.put(write[i]);
                i++;
           }
           /*
            * if we managed to put all the data in the byte buffer
            * then we don't need write any more.  Otherwise remember
            * the part we could not put in the ByteBuffer
            */
           if(i == write.length) {
                write = null;
           } else {
                byte[] newb = new byte[write.length-i];
                System.arraycopy(write, i, newb, 0, write.length);
                write = newb;
           }
```

## Solution #3: buffering write() data: Part 2

```
/* pass the ByteBuffer writebuf to the channel to send */
channel.write(writebuf);

/* remove the part that has been sent */
writebuf.compact();
```



## Solution #3: buffering write() data: Part 2

```
/* if there is nothing to be sent at all, we're done */
if(writebuf.position() == 0 && write == null) {
  server.removeClient(this);
}
```



## If your head hurts, that's ok

- You'll be getting practice with this in Lab 5, so all these things I've talked about you'll put into practice and it will sink in.



## Summary

- When an event occurs on a SocketChannel, we need to know who it is in relation to.
  - Solution: keep state
- For each client, we need to be prepared to read a partial line, and store it until the rest of the line arrives.
  - Solution: keep state
- For each client, we need to be prepared to send only part of our reply, and store the unsent portion until we are able to send it
  - Solution: keep state