Networks and Database Systems

Professor Tom Ellman
Lecture 4
Problem with Ordinary Java IO

• Read from InputStream may block thread.
• E.g., if data to be read from socket is not ready.
• Write to OutputStream may block thread.
• E.g., if socket is not ready to receive more data.
• Blocked thread cannot do any other work.
• Problem if we wish to serve many client connections, using only a single thread.
Implementing a Server with Non-Blocking IO

• Java provides the Selector class to encapsulate the work needed:
  – to keep track of pending client connection requests.
  – to keep track of client sockets that are ready to be read.

• Server uses special socket classes:
  – ServerSocketChannel
  – SocketChannel
The server program initializes the **Selector** by registering (Channel,Operation) pairs indicating the operations on channels it expects to handle.
The **Selector** interacts with the network to keep track of the (Channel, Operation) combinations that are ready.
The server program requests the **Selector** to provide a **Set** of **SelectionKey** objects. The **Selector** will return a **Set** including one **SelectionKey** for each channel that is ready for at least one registered operation. Each **SelectionKey** indicates the operations that are ready for one particular channel.
MultiEchoServerNIO

main()

• Create a ServerSocketChannel.
• Configure it for non-blocking behavior.
• Extract a ServerSocket from it.
• Bind the ServerSocket to an InetAddress.
• Create a Selector.
• Register the ServerSocketChannel with the Selector for connection accept operations.
• Call processConnections.
Create a socket channel that will be used to accept connection request from clients. Configure it for non-blocking IO:

```java
ServerSocketChannel serverSocketChannel;
serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.configureBlocking(false);
```

Get the server socket corresponding to the server socket channel just created:

```java
ServerSocket serverSocket;
serverSocket = serverSocketChannel.socket();
```

Bind the server socket to a port number:

```java
InetSocketAddress netAds = new InetSocketAddress(PORT);
serverSocket.bind(netAds);
```
Create a `Selector` object:

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

Register the `ServerSocketChannel` with the `Selector`, requesting it to keep track of opportunities to accept connections:

```java
serverSocketChannel.register(selector,
   SelectionKey.OP_ACCEPT);
```
procesConnections()

- Get SelectionKey set from Selector.
- For each key:
  - If key encodes an accept operation, call acceptConnection.
  - If key encodes a read operation, call acceptData.
Ask the **Selector** for the number of keys (number of channels) ready to be processed:

```java
int numKeys = selector.select();
```

If the number of keys is greater than zero, request the full set of **SelectionKey** objects:

```java
Set<SelectionKey> eventKeys = selector.selectedKeys();
```
Iterate over the set of \texttt{SelectionKey} objects. For each key, obtain an integer (bit vector) representing the operations that can be performed on the corresponding channel:

\begin{verbatim}
int keyOps = key.readyOps();
\end{verbatim}

If the key indicates the channel can accept a new connection, invoke a method for accepting and processing the connection:

\begin{verbatim}
if (keyOps.isAcceptable()) acceptConnection(key);
\end{verbatim}

If the key indicates the channel has data ready to be read, invoke a method for reading and processing the data:

\begin{verbatim}
if (keyOps.isReadable()) acceptData(key);
\end{verbatim}
Apparently the Java programmer is responsible for removing a key from the set of selected keys, once it has been processed:

```java
selector.selectedKeys().remove(key);
```
acceptConnection()

- Invoke `accept` on `ServerSocketChannel` object, obtaining a `SocketChannel`.
- Configure the `SocketChannel` for non-blocking behavior.
- Register `SocketChannel` with `Selector` to keep track of read operations that are ready.
To accept a connection, invoke the `accept` method on the `ServerSocketChannel` object, obtaining a `SocketChannel` object corresponding to the `Socket` connecting to the client:

```java
SocketChannel socketChannel;
socketChannel = serverSocketChannel.accept();
socketChannel.configureBlocking(false);
```

Register the `SocketChannel` with the `Selector`, requesting it to keep track of opportunities to read data from the corresponding client `Socket`:

```java
socketChannel.register(selector,
    SelectionKey.OP_READ);
```
acceptData()

• Get the **SocketChannel** object associated with the **SelectionKey**.

• Make a buffer to hold the data to be read.

• Invoke the **read** method on the **SocketChannel** object to read the client’s data into the buffer.
To accept data, begin by obtaining the `SocketChannel` object associated with the `SelectionKey` object.

```java
socketChannel = (SocketChannel) key.channel();
```

Allocate and clear a buffer to hold the data to be read:

```java
ByteBuffer buffer = ByteBuffer.allocate(2048);
buffer.clear();
```

Finally, read the client’s data into the buffer:

```java
int numBytes = socketChannel.read(buffer);
```
The availability of the read operation might have been triggered by the client closing the connection. This is indicated by a negative value (minus one) for the number of bytes read. In this case we should cancel the `SelectionKey` so that the `Selector` won’t keep track of this `SocketChannel` any longer. We should also close the `Socket`.

```java
key.cancel();
Socket socket = socketChannel.socket();
try {
    socket.close();
} catch (IOException ioEx) {
    //...
}
```
If the read operation actually produced data from the client, we want to process it. In the MultiEchoServerNIO example, we want to echo the data back to the client. We do this by resetting the buffer pointer to the beginning of the buffer, and then writing the buffer to the `SocketChannel` object.

```java
buffer.flip();
while (buffer.remaining() > 0)
    socketChannel.write(buffer);
```
Combining Non-Blocking IO with Threads

• In a realistic application, processing the data from a client could take a significant amount of time. (E.g., Searching a database.)
• That work should be put in a separate thread.
• The thread should die (and be garbage-collected) when the data processing is done.
• The server can continue:
  – Accepting new connections from clients.
  – Processing data received from other connected clients.
Comparison

• **MultiEchoServer**: Number of threads is equal to the number of client connections.
  – Includes connections for which the server is actively processing data. (Small number?)
  – Also includes connections on which server is waiting for data from client. (Large number?)

• **MultiEchoServerNIO**: Number of threads is equal to the number of connections for which the server is actively processing data.
Serialization

• The process of converting an object into a sequence of bytes so it can be stored in a persistent medium (e.g., a disk file) or transmitted through a byte stream (e.g., a socket). (Paraphrasing Wikipedia).

• In Java, a class may be declared to implement the `Serializable` interface.
  – Instances of the class may be written to an `ObjectOutputStream`.
  – Instances of the class may be read from an `ObjectInputStream`. 
Personnel Client/Server Example

- Server sends **Personnel** objects to Client.
- Server:
  - Obtains **OutputStream** from socket and wraps it in an **ObjectOutputStream**.
  - Uses **writeObject** method to send **Vector** of **Personnel** objects.
- Client:
  - Obtains **InputStream** from socket and wraps it in an **ObjectInputStream**.
  - Uses **readObject** method to receive **Vector** of **Personnel** objects.
- Client and Server must both have definitions of **Personnel**, defined in the same package.
In which cases will aliasing be preserved after objects are serialized (written to file(s)) and deserialized (read from file(s))? Answer: 1-Yes; 2-Yes; 3-No.

Projects:
- VectorSerialize
- VectorSerializeAliasing
- VectorSerializeAliasingAlt