Practical Session #10 - Java NIO, ByteBuffers

Tom Mahler
In This Recitation We’ll Cover

- Threads In C++
- Java Thread-Per-Client
- Java New-IO (NIO) Classes
Threads In C++
Similarly to Java, C++ also supports threads and synchronization.

In this course, we will use the Boost library to easily create and synchronize threads in C++.
Threads In C++

- Hopefully, you've already reviewed this in the supplementary material for ps6, but here is a quick recap:
  - There is no Runnable interface
    - Instead, create a Task class with a run function (or any other name you like)
  - Run your task classes within a boost::thread object, or using a 
    boost::threadpool object and its schedule function
  - Protect objects using boost::mutex and boost::mutex::scoped_lock

- Additional information can be found here
The Boost C++ library provides several classes that can be used to write multi-threaded programs in C++

The process of writing a multi-threaded program is similar to what you already know from Java

There are few minor differences
You need to perform the following steps:

1. There is no Runnable interface:
   - Instead, create a Task class with a run function (or any other name you like)
   - The function can receive arguments
You need to perform the following steps:

1. There is no Runnable interface:
   - Notice that if you override operator() (similar to operator= just for ()) then it will be called by default
   - On the second example, we give the function's address but since it belongs to a class it has a first hidden parameter (the "this" pointer)
You need to perform the following steps:

2. Create an instance of this class

3. Create a threadpool, and start the tasks using the method `schedule(&MyClass::MyFunction, &instance [, parameter1, parameter2 ...])` of ThreadPool

4. You can use the method `wait()` of ThreadPool to wait for all threads in the pool to complete
Examples

See code...
(See boostThreads2)
Basic Synchronization

- Basic synchronization is performed using the mutex object `boost::mutex`.

- If several threads share a resource, they should also share a lock.

- When a thread needs to access the shared resources, it should acquire the lock first by creating a `boost::mutex::scoped_lock` instance.
Basic Synchronization

- The constructor acquire the lock on the mutex for the current scope

- Upon exiting the scope, the lock is released automatically

- As in java, only one thread is allowed to acquire the lock in any point in time
Basic Synchronization

- Other threads trying to acquire the lock when it is already acquired by some thread, will wait until it is released.

- The next example corrects the synchronization problem from the previous example.
  - The shared resource is the output stream.
Examples

See code...
(See boostThreads1)
Java Thread-Per-Client
Java Thread-Per-Client

In the previous PS, we reviewed basic Java networking

As you recall reading/writing to network sockets, as well as awaiting new connections on the server side - block thread execution
A simple solution to this problem is allocating a thread for each client the server must serve.

While the drawback of this method have been discussed in class it's main advantage is that socket operation can now block threads without interrupting server operation.
The `BaseServer` abstract class contains the framework for a Thread-Per-Client server

It's `serve()` function constantly attempts to accept new connections
Each socket produced by a new connection is handed over to a BlockingConnectionHandler along with a MessagingProtocol and a MessageEncoderDecoder.

The newly created BlockingConnectionHandler is executed in its own thread.
Examples

▶ See code...
(See BaseServer)
The BlockingConnectionHandler is a runnable whose task is to handle a single connection.

In order to receive and respond to messages, it uses a MessageEncoderDecoder and a MessagingProtocol.

The first separates incoming bytes into complete messages while the second processes complete messages and produces responses.
Responses are then encoded via the MessageEncoderDecoder before writing them to the socket.

After the BlockingConnectionHandler code we'll present an example of a MessageEncoderDecoder and a MessagingProtocol for the Echo protocol.
Examples

- See code...
  (See BlockingConnectionHandler)
We define next the protocol interface

The MessagingProtocol interface operates in the following context:

- A ConnectionHandler instance wraps together: the socket connected to the client; the
- MessageEncoderDecoder which splits incoming bytes from the socket into messages.
- The next step is to pass the incoming messages from the client to the MessagingProtocol which will now execute the action requested by the client. The task of the MessagingProtocol is to look at the message and decide what should be done. This decision may depend on the state of the connection (remember the example of the "authenticated" protocol). Once the action is performed, we will need to send an answer to the client. So we expect to get an answer back from the MessagingProtocol.
Messaging Protocol

- The MessagingProtocol interface operates in the following context:
  - A ConnectionHandler instance wraps together:
    - The socket connected to the client;
    - The MessageEncoderDecoder which splits incoming bytes from the socket into messages
The MessagingProtocol interface operates in the following context:

- The next step is to pass the incoming messages from the client to the MessagingProtocol which will now execute the action requested by the client.
- The task of the MessagingProtocol is to look at the message and decide what should be done.
The MessagingProtocol interface operates in the following context:

- This decision may depend on the state of the connection
  - Remember the example of the "authenticated" protocol
- Once the action is performed, we will need to send an answer to the client
- So we expect to get an answer back from the MessagingProtocol
Examples

- See code...
  (See MessagingProtocol)
Messaging Protocol

- Note that we allow the protocol to use message any type of message (the type argument T)

- This means that the operation of Serialization and Deserialization (encode/decode complex parameters to/from Strings) will be performed by the MessageEncoderDecoder - which yield a good separation of concerns
Echo Protocol

- This interface is now implemented in the form of an "echo protocol" - which will
  - Print received messages on the server side screen,
  - Then reply with the original message while repeating the last two chars a couple of times

- That is, if a client send to the server the line "hello" it will be responded with the line "hello .. lo .. lo .."
Examples

➤ See code...
  (See EchoProtocol)
Examples

- See code...
  (See MessageEncoderDecoder)
Examples

See code...
(See LineMessageEncoderDecoder)
Echo Client

Now that we have a full Thread-Per-Client framework, we can test it with a corresponding echo client
Examples

- See code...
  (See EchoClient)
Java New-IO (NIO) Classes
Goal

In this practical session we will:

1. Devise a server that can handle more than one client at a time
2. Learn a bit about new IO capabilities of Java, and rewrite the EchoClient with this API
Non-blocking IO

- If we examine our client, once we reach the following code:

  ```java
  System.out.println(in.readLine());
  ```

- The program is blocked until the user press enter
Non-blocking IO

- No data can be received through the network buffer
- We want a non-blocking I/O handling so that the network buffer will keep working, even when we get to the above code
- Also, we would like to do so without using more than one thread
Non-blocking IO

Examples for such program:

A chat application:
- The client is expected to keep receiving messages, even when the user is typing a new message.

A client that need to get updates from several servers:
- It should try reading from all of them without blocking.
Java NIO API

- Java's java.nio package is a new, efficient IO package
  - It also supports Non-blocking IO
- We will see how to do non-blocking IO in the next practical session
In this session, we will concentrate on some of the concepts of the NIO API, and will go over the EchoClient written using the nio API.

The key players we need to know are Channels, Buffers, and Selector.

This week we will learn about Channels and Buffers.
Channels

- A Channel is something you can read from and write to
- Channels can be either blocking (by default) or non-blocking
- We are interested in SocketChannel and ServerSocketChannel
  - These are like Socket and ServerSocket we already know, but their read(), write() and accept() methods can be non-blocking
Channels

- The ServerSocketChannel's accept() method returns a SocketChannel

- The other side of the connection can of course be either blocking or non-blocking (it doesn't matter)
Setting up a non-blocking ServerSocketChannel listening on a specific port is done as follows:

```java
int port = 9999;
ServerSocketChannel ssChannel = ServerSocketChannel.open();
ssChannel.configureBlocking(false);
ssChannel.socket().bind(new InetSocketAddress(port));
```
Setting up a non-blocking SocketChannel and connecting to a server is done as follow:

```java
SocketChannel sChannel = SocketChannel.open();
sChannel.connect(new InetSocketAddress("localhost", 1234));
sChannel.configureBlocking(false);
```
Buffers

- All the IO in NIO is going through Buffers

- Channels know how to read and write into Buffers, and buffers can read and write into other buffers

- We'll be using ByteBuffer
  - These are buffers that hold bytes
Buffers

- Creating a new ByteBuffer:

```java
final int NUM_OF_BYTES = 1024;
ByteBuffer buf = ByteBuffer.allocate(NUM_OF_BYTES);
```

- Creating a ByteBuffer from a an array of bytes:

```java
byte[] bytes = new byte[10];
ByteBuffer buf = ByteBuffer.wrap(bytes);
```
Flipping

- Each buffer has a
  - Capacity
  - Limit
  - Position marker

- A read operation reads a specified number of bytes from the current position, and updates the position marker to point to the yet unread bytes
Flipping

- Similarly, a write operation writes some bytes from the current position, and then advances the position

- You can't read or write more than the limit of the buffer

- You can't increase the limit over the capacity
Flipping

- It can be described as

\[0 \leq \text{position} \leq \text{limit} \leq \text{capacity}\]

- This means that if someone wrote some bytes into a buffer, and then you want to read them, you need to set:
  - The position marker to the start of the buffer set
  - The limit of the buffer to the former position
Flipping

- This is what the Buffer's `flip()` method does

- It sets the limit of the buffer to the current position marker, and the position marker to the start of the Buffer
Flipping

- It is equivalent to:

Buffer b = ...;
int position = b.position();
b.limit(position);
b.position(0);
Reading from a channel to a buffer:

```java
numBytesRead = _socketChannel.read(buf);
```
From Channel To Buffer & Back

- Writing from a buffer to a channel:

  ```java
  numBytesWritten = _socketChannel.write(buf);
  ```

- If read or write returns -1, it means that the channel is closed

- Read and write operations on Buffers update the position marker accordingly