COMP2221 Networks

David Head

University of Leeds

Lecture 12

Previous lectures

In the last few lectures we have seen how a **multi-threaded server** can serve **multiple clients simultaneously**.

- Each client has their own **handler**.
- Handlers are either derived from `Thread`, or implement the `Runnable` interface (both defined in `java.lang`).
- Convenient to use the high-level `Executor` to manage **thread-pools** of fixed or variable size (from `java.util.concurrent`).

Each thread still only handles **at most** one client at a time.

- May not be a good use of thread resources.

Today's lecture

Today we will take a high-level look at an alternative performance enhancement: **non-blocking I/O**.

- Non-blocking *versus* blocking I/O.
- `java.nio` *versus* `java.io`.
- `Buffer`, `Channel` and `Selector` classes, all defined in `java.nio`.

We will then see an example of how these can be applied to a server architecture.

- Will **not** go into any great detail in this module.

Blocking I/O with `java.io`

Methods of `java.io` such as `read()` and `write()` are **blocking**.

- The thread associated with that operation will be **idle** until a connection is made, data is received *etc.*
- Wasteful of resources, but easy to implement.

Multi-threaded servers *[Lectures 9-11]* improve use of resources:

- If one thread is idle, the scheduler will **suspend** that thread and run another thread — hopefully one of yours!

Still potential for wasted CPU time if *e.g.* each thread is handling **long-lived** clients with **infrequent** communication.

Timeout exceptions

Blocking methods would **hang** if *e.g.* the client **never** responded to the server. To check for this, can use **timeouts**.

- *e.g.* `Socket` class has the method[1]

  `setSoTimeout(int timeout)`

  where `timeout` is in milliseconds.
- Exceeding this time throws an `InterruptedIOException`, derived from `IOException`.
  - *e.g.* `SocketTimeoutException`.
- Must include this in our application logic (retry? give up?)

Still potential for wasted CPU resources.

[1] The '`So`' refers to **socket** and relates to the original coding in C.

Non-blocking I/O with `java.nio`

We can also consider the **non-blocking** I/O features in `java.nio`.

These were originally designed to support **high-performance servers**.

- **Long-lived clients** still a problem for thread-pools.
- Blocking I/O mandates the use of threads.
- Multiple threads can introduce synchronisation problems (if they interact).

`java.nio` allows multiple-client handling **in a single thread**.

- Application logic is more complex.

Primary Java classes

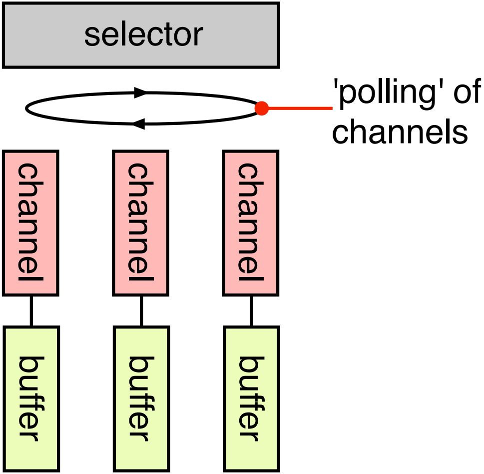
The most important classes/interfaces for non-blocking I/O are:

- **Buffers**, a container for data.
- **Channels**, a container (actually an interface) for I/O objects (*i.e.* `Socket`).
- **Selector**, manages channels.

`Buffer` is defined in `java.nio`.

`Channel` and `Selector` are defined in `java.nio.channels`.

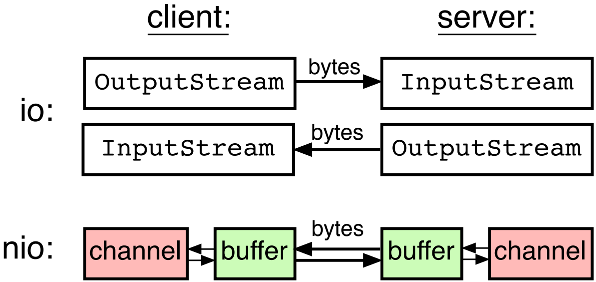
High-level architecture



Buffers

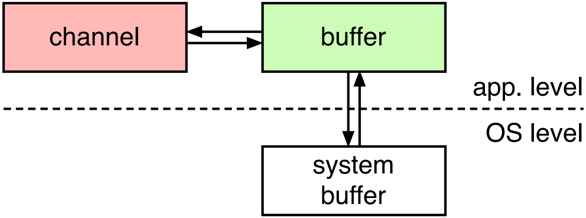
The `Buffer` class is a **non-stream based I/O container**.

- Used by `Channel` for I/O.
- Data moved as larger blocks, rather than small bytes.
- Are switchable, *i.e.* can be **bi-directional** (contrast with the I/O streams in `java.io` which are **uni-directional**).



Buffers and OS buffers

- Instances of `Buffer` are **lower level constructs** than standard I/O streams.
- Map naturally to OS system-level buffers.
- Efficient use of buffer data **in place**.



Channels

A `Channel` represents a **pollable I/O target**.

- *e.g.* `Socket`.

A `Channel` can be configured to be either blocking or non-blocking.

- We focus on non-blocking here.

A `Channel` permits two-way communication.

- Can **read or write** from its associated `Buffer`.

A `Channel` **registers** with a `Selector`.

- `Selector` manages the channel and monitors its state, *i.e.* if it is readable, writable *etc.*

Server channel code

Open a `ServerSocketChannel` (implements the `Channel` interface; defined in `java.nio.channels`):

```
ServerSocketChannel serverChannel = ServerSocketChannel.open();
```

Configure to be non-blocking (as it defaults to blocking):

```
serverChannel.configureBlocking(false);
```

Bind to the **listening** port:

```
serverChannel.socket().bind( new InetSocketAddress(portNum) );
```

Selector

The `Selector` class manages access to a set of `Channels`.

- Of any type; server or client connections.

The `select()` method queries all channels **registered** with this selector.

- Returns any **ready** channel.
- Can be ready for **reading**, **writing**, or **accepting** (a new client).

Registering a channel with a selector

Use the `Selector.open()` **factory method**, which returns the system's default selector provider:

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

Register using the `register()` method **of the channel**:

```
serverChannel.register( selector, SelectionKey.OP_ACCEPT );
```

Registered channels are known as **keys**.

Common options are:

- `SelectionKey.OP_ACCEPT` for a listening `ServerSocket`.
- `SelectionKey.OP_READ` to read data.
- `SelectionKey.OP_WRITE` to write data.

Channel state

A `Channel` can be idle, or ready for one of several operations.

For a `ServerSocketChannel`, there is only really one:

- Client connection requested (`SelectionKey.OP_ACCEPT`).

For each of the client `SocketChannels`:

- Reading from a buffer (`SelectionKey.OP_READ`).
- Writing to the buffer (`SelectionKey.OP_WRITE`).

We use the `Selector` to return **ready channels** and a **key** to their state.

- Selectors can manage a collection of channels with various states.

```java
while (true) {
  selector.select();
  Iterator keys = selector.selectedKeys().iterator();
  while( keys.hasNext() ) {
    SelectionKey key = (SelectionKey) keys.next();
    keys.remove();

    if( !key.isValid() ) continue;

    if( key.isAcceptable() ) {
      accept(key);
    }
    else if( key.isReadable() ) {
      read(key);
    }
    else if( key.isWritable() ) {
      write(key);
    }
  }
}
```

Code explanation

`selector.select()` finds all **ready channels**, possibly more than one.

`Iterator keys = ...` returns an iterator over the keys.

`key = (SelectionKey) keys.next()` takes the next item — also calls `remove()` so it is not duplicated.

We then see what state this key is in — `isAcceptable()`, `isReadable()`, `isWritable()` — and call the corresponding method.

Note this is an **event-driven** model, **on a single thread**.

Client connections

Our server channel can accept new client connections.

- Calls accept(SelectionKey key) in our example.

```java
// Recover the server channel
ServerSocketChannel serverChannel =
    (ServerSocketChannel) key.channel();

// Create a non-blocking channel for this client
SocketChannel channel = serverChannel.accept();
channel.configureBlocking(false);

// Register with the same selector; expect to read
    next.
channel.register(selector,SelectionKey.OP_READ);
```

Reading from the client

Reads in the client data from the channel's **buffer**. If there is none, assume the connection has been closed:

```java
ByteBuffer buffer = ByteBuffer.allocate(8192);
int numRead = -1;
try {
    numRead = channel.read(buffer);
}
catch( IOException e ) { ... }

if( numRead==-1 ) { ... } // Close connection.
```

Else store the message in a `HashMap<SocketChannel,...>`, and register the channel for **writing**:

```java
key.interestOps( SelectionKey.OP_WRITE );
```

Writing to the client

Get the client message from the HashMap<SocketChannel,...>, and send back to the client.

```
byte[] message = ...;  // Get message from hash map.
channel.write( ByteBuffer.wrap(message) );
  // 'wraps' a byte array into a buffer.
```

Now register our interest in reading (*i.e.* another message) from this channel:

```
key.interestOps( SelectionKey.OP_READ );
```

Other details

Code on Minerva: `Client.java`, `EchoServer.java`

There are many details for non-blocking I/O that we do not have time to cover here.

See the code example on Minerva for a server that just re-sends what the client sent to it, *i.e.* `EchoServer.java`.

- The `HashMap` stores *multiple* byte array messages from the client, in case the client sends multiple messages.
- `Client.java` just uses blocking I/O for simplicity and only sends a single message.

See also the relevant chapter of the Harold book *Java Network Programming* (chapter 11 in the 4^{th} ed.).

Pros and cons of non-blocking I/O

The single-threaded non-blocking approach is **conceptually** simpler than the multi-threaded approach, but in reality **harder to implement**.

- Requires care.
- Switching between read and write can be complex.

When first released in the 1990's, non-blocking architectures dramatically out-performed blocking ones.

- Java was slow to release `nio`.
- By the time it came out, the advantages over multi–threaded architectures were already eroding.

Non-blocking I/O today

By the 2000's, machine and operating system improvements have reduced the advantage to the point that:

- Commodity servers can support $\sim$10,000 threads.
- Multi-threaded, stream-based I/O servers can out-perform non-blocking architectures by as much as 30%.

May still outperform multi-threaded servers in situations with:

- A large number of threads that are long-lived but low activity.
- Blocking I/O would waste resources in this case.

Overview and next lecture

Today we have looked at **non-blocking I/O**, which is implemented in Java in the `java.nio` package:

- Requires greater coding complexity that multi-threaded servers.
- Use `Buffer`, `Selector` and `Channels` to maintain multiple connections on one thread.
- Example code on Minerva: `Client.java`, `EchoServer.java`

Next time we will briefly look at security and **secure sockets**.