COMP2221 Networks

David Head

University of Leeds

Lecture 10

Reminder of the last lectures

Two lectures ago we saw how to implement a simple Java server using `ServerSocket`:

- Called the **blocking** `accept()` method to wait for clients.
- Could only handle one client at a time.
- Examples: `DailyAdviceServer` and `KnockKnockServer`.

Last lecture we looked at **parallel computation** in general:

- Historical development of **parallel hardware**.
- The difference between:
  - **Shared** and **Distributed** memory.
  - **Parallelism** and **Concurrency**.
  - **Processes** and **Threads**.

Today's lecture

In today's lecture we will see how to implement a **multi-threaded server** in Java.

- Use Java's built-in `Thread` class.
- **Shared memory**, *i.e.* utilises multiple cores for one machine (or one node in a cluster), but **only** one machine/node.
- One thread per client.

In the next lecture we will see two alternatives to this **thread-per-client** model.

First, we will look at how to program in parallel in Java.

The Java Thread class

We have two basic options for threading in Java:

1. Subclass the `Thread` class (in `java.lang`).
2. Use the `Runnable` interface.

In both cases, you will put your code into:

```
public void run()
```

Request a new thread to execute the code in `run()` by calling:

```
public void start()
```

Other sometimes useful methods:

- `join()`, waits for the thread to finish.
- `sleep(long millisecs)`, causes the thread to sleep.

Scheduler

Threads do **not** start running the instant we create them.

- **When** a thread runs is managed by a component of the operating system called the **scheduler**.
- Allocates core time **dynamically**, depending on other applications, background tasks *etc.*

Since you cannot control when other applications or background tasks are active, a parallel program may generate **different results** each time it is executed.

- This **non-determinism** is usually undesirable.
- Can be corrected with **synchronisation** *(later this lecture)*.

Method 1: Inheritance

Code on Minerva: `HelloT.java`

Subclass `java.lang.Thread`, overriding the `run()` method:

```
public class HelloT extends Thread {

  public void run() {
    System.out.println( "Hello from a thread." );
  }

  public static void main( String args[] ) {
    HelloT t = new HelloT();
    t.start();  // Schedule the execution.
  }
}
```

When `t.run()` is actually called depends on the scheduler.

Running multiple threads

```java
public class HelloT extends Thread {
  public void run() { ... }

  public static void main( String args[] ) {
    HelloT t1 = new HelloT();
    t1.start();  // Start thread 1.

    HelloT t2 = new HelloT();
    t2.start();  // Start thread 2.

    // May have thread 1 and thread 2 running now,
    // in addition to the main thread.
  }
}
```

Note that, as with the previous example, although calling `start()` starts the thread, the **scheduler** decides when `run()` is called.

Method 2: Interface

Code on Minerva: `HelloI.java`

Implement the `java.lang.Runnable` interface:

```java
public class HelloI implements Runnable {

  public void run() {
    System.out.println( "Hello from a thread." );
  }

  public static void main( String args[] ) {
    HelloI h = new HelloI();
    Thread t = new Thread(h);  // Initialise with h.
    t.start();
  }
}
```

This time we **use** a thread, rather than **being** a thread.

Inheritance *versus* Interface

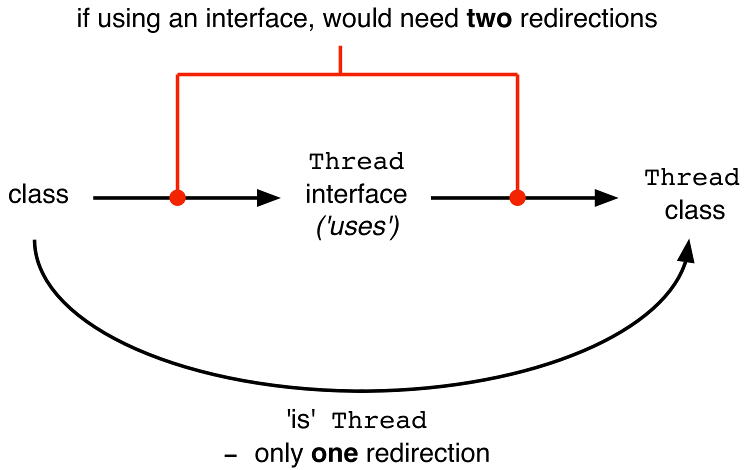
The **interface** approach is more general:

- Can inherit from a separate class.
- Can implement multiple interfaces, but cannot have multiple inheritance[1].
- Threading is a **property** of a class, not essential to its definition.

Inheritance is simpler, but our class **must** be a descendent of `Thread`.

- May be a *very slight* performance improvement relative to the interface approach (less 'indirection').

[1]Some languages, *e.g.* C++, *do* allow multiple inheritance.


if using an interface, would need **two** redirections
class
Thread
interface
('uses')
Thread
class
'is' Thread
– only **one** redirection

General issues in thread programming

Two important multi-threading issues to be aware of are **data races** and **synchronisation**.

A **data race** is when two threads read and write the **same memory location**:

Thread 1	Thread 2
`globalData = 1;`	`...`
`...`	`globalData = 2;`
`a = globalData;`	`...`
`...`	

Is a 1 or 2?

- The result of the **read** depends on whether or not the other thread has changed the data.
- Since we cannot control when threads run, the result may vary each time - **non-determinism**.

Synchronisation (1)

If one thread cannot continue until another one finishes, must **synchronise**. There are two common approaches:

1. Calling `t.join()` will pause the calling thread until its subthread `t` has finished — a **blocking** call *(below)*.
2. Defining a **synchronised block** or **method** *(next slide)*.

Consider the following code called by the main thread:

```
t = new Thread();
t.start();
...  // Main thread and thread t run concurrently.
t.join();
...  // Only main thread; t.run() has completed.
```

Synchronisation (2)

Can specify **blocks** of code that only one thread can enter at a time[1].

In Java, this is achieved using `synchronised(object)`:

```
synchronized( System.out ) {
   System.out.print("Socket host:  " + so.getInetAddress());
   System.out.print(" on port:  " + so.getPort());
   System.out.println();
}
```

Without `synchronised()`, the `print()` statements for each thread could become **interleaved**.

[1]There is a single **lock** for the block with a thread must **acquire** before entering the block. It relinquishes the lock when leaving the block.

Synchronisation (3)

Java also allows you to synchronise **methods**. For example:

```
public synchronized void printStatus() {
   System.out.print("Socket host:  " + so.getInetAddress());
   System.out.print(" on port:  " + so.getPort());
   System.out.println();
}
```

This works the same way as the previous example, *i.e.* only one thread can enter the method at a time.

Note that use of `synchronized` can affect performance, sometimes severely.

Synchronisation (4)

These last two examples could be resolved by combining each `print()` statement into a single `println()`.

- `PrintStream`, `PrintWriter` internally synchronise.

`OutputStream` objects do **not** synchronise in general, so *e.g.* outputting to the same log file may require synchronisation.

For multi-threaded servers, must consider if multiple threads can write to the same global data, file *etc.*

- Data races can arise if **at least** one thread **writes**.
- Synchronisation may be required for *e.g.* outputting to a log file, accessing and updating a database, *etc.*

Client-server architecture

The software architecture we choose should be driven by the **application requirements**.

Relevant considerations are:

- How **many** clients to we expect (concurrently)?
- How **long** are clients connected for?
- What type of **protocol** are we implementing? For instance, is communication continual, or are there idle periods?

Examples:

- An `ftp` server might expect many, short connections.
- A chat client might expect fewer, longer connections.

Thread-per-client architecture

Code on Minerva: `KKMultiServer.java`, `KKClientHandler.java`, `KnockKnockProtocol.java`

Uses a **single thread per connected client**.

- Creates the thread as the connection is made.
- Destroys the thread once the connection closes.

Assumes resources are available for many threads:

- Each thread requires some **local data**.
- Creating and destroying threads also costs **CPU cycles**.

At large enough **scale** (*i.e.* large number of threads), these costs outweigh the benefits

- Adding more threads makes the system less responsive.
- Typically limited to 100's to 1000's of unique threads on a typical desktop.

Multi-threaded KnockKnockServer

KnockKnockServer is replaced with KKMultiServer:

- Still a single thread (the 'main' thread).
- For each accept(), creates a new KKClientHandler object.

KKClientHandler is derived from Thread.

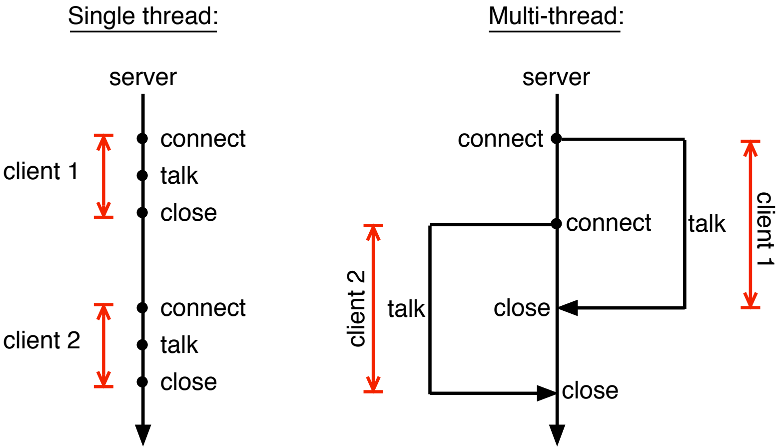
- Passed the client Socket in its constructor.
- Creates one instance of the protocol **per client**.
  - *i.e.* KnockKnockProtocol, unchanged from the earlier version.

The client KnockKnockClient is unchanged.

- Will not consider multi–threaded clients, although they are sometimes useful, *e.g.* loading all images from a web page concurrently.

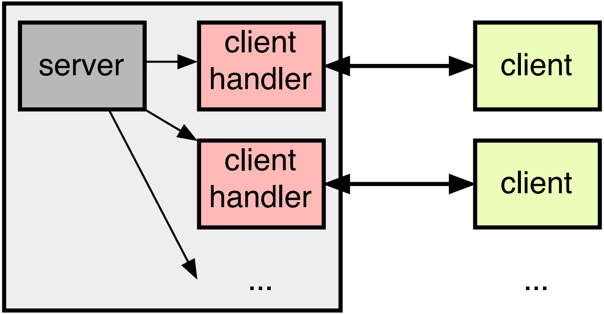
KKMultiServer

```java
public class KKMultiServer {
  public static void main(String[] args) throws IOException
  {
    boolean listening = true; // Always true here.

    try {
      serverSocket = new ServerSocket(2323);
    } catch (IOException e) {
      System.err.println("Could not listen on port: 2323.");
      System.exit(-1);
    }

    // Each accept() creates a new thread.
    while( listening )
      new KKClientHandler(serverSocket.accept()).start();

    serverSocket.close();
  }
}
```


Single thread:
server
connect
talk
close
client 1
connect
talk
close
client 2
Multi-thread:
server
connect
connect
talk
client 1
close
client 2
talk
close

KKClientHandler

```java
public class KKClientHandler extends Thread {
  public void run() {
    try {
      PrintWriter out = new PrintWriter(
                              socket.getOutputStream(),true);
      BufferedReader in = new BufferedReader(
                             new InputStreamReader(
                              socket.getInputStream()));
      String inputLine, outputLine;
      KnockKnockProtocol kkp = new KnockKnockProtocol();
      outputLine = kkp.processInput(null);
      out.println(outputLine);

      while ((inputLine = in.readLine()) != null) {
        // As per the basic version
      }
      out.close();
      in.close();
      socket.close();
    } catch (IOException e) { ... }
}
```


server
client handler
client handler
...
client
client
...

Thread-per-client pros and cons

Pros:

- Simple to understand and implement.
- Improvement over the non-threaded version we saw earlier.

Cons:

- Resources can potentially grow **without limit**.
  - Could try to estimate a maximum number of clients.
- A thread is created and destroyed for **each client**.
  - Although a much smaller overhead than for spawning processes[1], it can be significant.
- Better to create threads **once** and re-use for multiple clients.

[1] Multiple processes **were** dynamically created/destroyed in early servers; see Harold, *Java Network Programming*, 4$^{\text{th}}$ ed., chapter 3.

Today we have looked at **multi-threading** in Java, and our first implementation of a **multi-threaded server**:

- The **thread-per-client** model where each client has a **client handler** on its own thread.
- Risks unlimited growth of resources.

Next time we will see two more models using **thread pools** that remove the risk of unlimited growth of resources.

Code for the thread-per-client model is on Minerva.