COMP2221 Networks

David Head

University of Leeds

Lecture 11

Reminder of the last lecture

Last time we looked at multi-threaded programming in Java, and our first model for a multi-threaded server:

- The **thread-per-client**, where the main thread launches a **client handler** for each client on a separate thread.
- Can handle many clients simultaneously, and is fairly easy to implement.
- Can lead to an **uncontrolled increase in resources** if many clients try to connect, reducing performance.

Today we will look at two alternative models that only utilise a finite number of threads.

- This limited number of threads is known as a **thread pool**.
- **Controls** the growth of resources.
- The **thread-pool model** requires the number of threads to be estimated **in advance**.

Since thread pools are quite common in multi-threaded programming, Java now provides **Executors** to provide greater flexibility.

- Our third model is an **executor server**.

Thread-pool server

Basic approach is to have a **fixed number of threads** to handle clients.

- Known as a **thread-pool model**.
- Also called **threads-per-server**, as the number of threads is set by the server (and *not* the clients).

The threads **queue** to accept clients.

- The first `accept()` blocks all later threads (until another client makes a connection).

Behaves like a **set** of servers with the same listening port.

Implementation

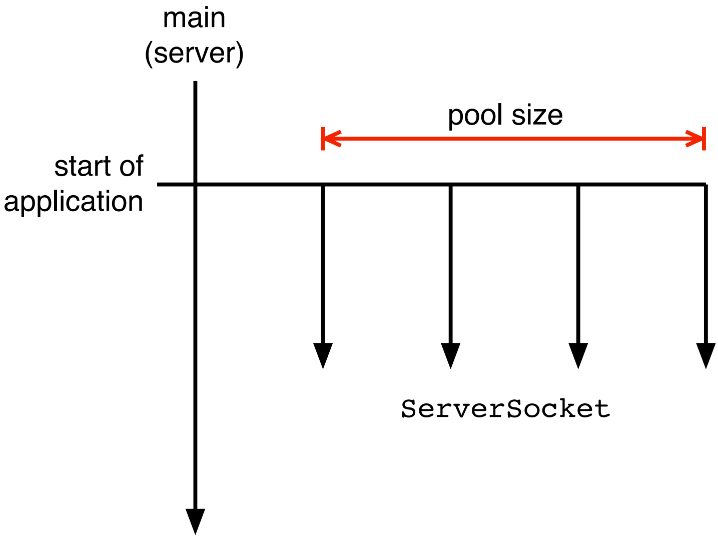
In essence, every thread runs a version of our earlier one-client-per-server (serial) model from Lecture 8:

1. Create a number of threads at the start of the application.
   - How to choose the **right number**?
2. Each thread listens to the **same listening port**.
   - Possible since these are **threads**, *not* processes.
3. Each thread calls the blocking `accept()`.
4. When this method returns, each thread will handle **one client at a time**.

Thread-pool server *(schematic)*

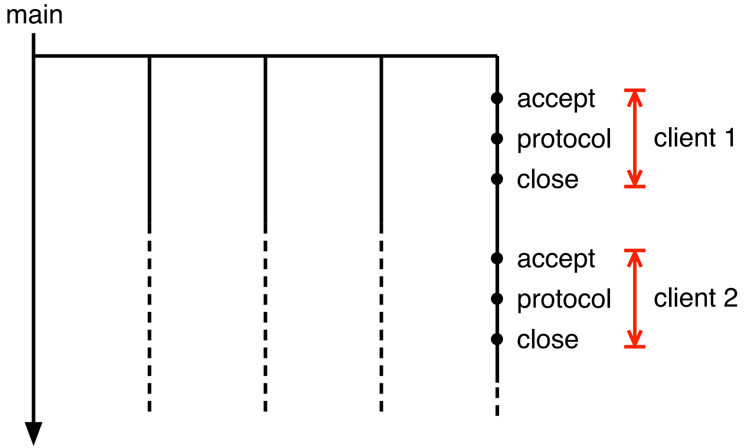
```
public class ThreadPoolServer {
  ...
  public void runServer( int poolSize ) {
    ServerSocket serverSock = new ServerSocket(2323);

    for( int i=0; i<poolSize; i++ ) {
      ServerThread s = new ServerThread( serverSock );
      s.start();
    }
  }
  ...
}
```

- Still only one `ServerSocket` which is shared by all threads.
- Therefore still only one **listening port**.
- `poolSize` could be a static instance variable, set by command line arguments in `main()`, *etc.*


main
(server)
pool size
start of
application
ServerSocket

Server thread *(schematic)*

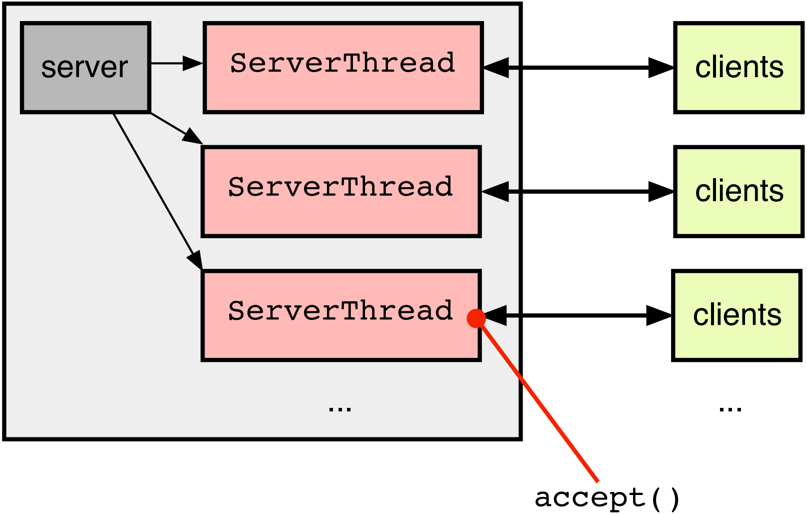
```java
public class ServerThread extends Thread {
  private ServerSocket serverSock = null;
  ...
  // Constructor stores the ServerSocket
  public ServerThread( ServerSocket s ) {
    serverSock = s;
  }
  ...
  public void run() {
    while( true ) {
      Socket client = serverSock.accept();
      Protocol p = new Protocol( client );
      p.handleClient();
      // e.g. KnockKnockProtocol
    }
  }
  ...
}
```



(each server thread handles multiple clients consecutively)

Thread-pool behaviour

- The pool size is a **fixed parameter** of our model.
- Each thread is a **sequential** server running `accept()`.
  - Essentially the same that in Lecture 8.
- The threads are **queued** at the listening port.
- Which thread returns from `accept()` depends on the JVM and the thread scheduler.
  - May vary from run to run — **non-determinism**.
- When a client connection is ended, the corresponding thread returns to the start of its `while`-loop and calls `accept()`.
  - It has 'returned to the pool.'


server
ServerThread
ServerThread
ServerThread
...
clients
clients
clients
...
accept()

Thread safety

Note that multiple threads call `accept()` on the **same** `ServerSocket` object.

Does `accept()` perform properly in this situation?

- *i.e.*, will it only return the `Socket` object to one thread, or might it return the same client to two or more threads?

In other words, is `accept()` **thread safe**?

The official documentation does not make it clear if `accept()` is thread safe.

- Some implementations of JVM may be, some may not.

As a general rule, if a class of not explicitly declared thread safe, it is best to assume it is **not**.

Thread-safe accept()

If we were worried about this, we could use `synchronized(...)` from last lecture:

```java
public void run() {
  while( true ) {
    synchronized( serverSock ) {
      Socket client = serverSock.accept();
    }
    Protocol p = new Protocol( client );
    p.handleClient();
    // e.g. KnockKnockProtocol
  }
}
```

This way, only one thread can call `accept()` at a time.

- Thread-safety of `accept()` no longer required.
- Which thread enters the synchronized block first is still unpredictable.

Thread-pool model pros and cons

Pros:

- Can handle multiple clients connecting to the same port.
- The required resource is finite and controllable.
- Threads are not created or destroyed in between client connections.

Cons:

- Must **tune** the pool size to the expected load.
  - Too small, and clients will have to wait.
  - Too large, and the overhead will increase and all threads will slow down.

Ideally we would **dynamically** alter the pool size to match the load.

Executor server

Ideally we would like to **re-use** existing threads if they are 'idle' *(i.e. doing nothing / waiting for a client to connect)*, and only create a new thread if all current ones are in use.

This would take some time to develop using `java.lang.Thread`.

However, because it is useful in many contexts, Java already provides the high-level **Executor** service that does most of the work for us.

- Found in `java.util.concurrent`.

java.util.concurrent

`java.lang.Thread` is actually quite a low-level[1] detail of concurrency.

Java 5 introduced a **higher-level** programming model:

- The `java.util.concurrent` package
- Decouples **scheduling** and **execution**.
- Allows us to manage the use of threads, and leave the details of creation/destruction to elsewhere.

Assume the JVM handles this sensibly, so the performance is 'good enough.'

[1] Although some other languages allow even lower level control, *i.e.* C++11.

newCachedThreadPool

The `newCachedThreadPool` creates a **flexible** pool that:

- **Creates new threads** if one of the current ones are in use.
- **Re-uses** existing threads if they have completed their task.

It therefore **automatically adjusts** to the load (*i.e.* the number of clients).

There is also a `newFixedThreadPool(...)`, which sets the pool size.

- Similar to our thread-pool model, with the same 'con's.
- Only requires one line of code to change between `newCachedThreadPool` and `newFixedThreadPool`.

Using `newCachedThreadPool`

`newCachedThreadPool` is a **factory method** that returns an instance of `ExecutorService`.

Usage is straightforward:

```
ExecutorService service = Executors.newCachedThreadPool()
```

- `Executors` can return different objects, including cached and fixed-size thread pools.
- `ExecutorService` is generic (polymorphism).

```
service.submit( new KKClientHandler(client) )
```

- **Schedules** the execution of the task.
- `submit`'s argument implements the `Runnable` interface, and/or extends `Thread`.

KKExecutorServer

Code on Minerva: KKExecutorServer.java

```java
public class KKExecutorServer {
  public static void main(String[] args) throws IOException
  {
    ServerSocket server = null;
    ExecutorService service = null;

    try {
        server = new ServerSocket(2323);
    } catch (IOException e) { ... }

    service = Executors.newCachedThreadPool();

    while( true )
    {
      Socket client = server.accept();
      service.submit( new KKClientHandler(client) );
    }
  }
}
```

Executor server pros and cons

Pros:

- Only creates new threads when existing resources are exhausted.
- Can switch to a fixed pool by changing one line.

Cons:

- The pool is still **unbounded**, so may exhaust machine resources (go out of memory, or run very slow).
- In particular, **long-lived clients** keep their resources.

Finer control, including a max. no. of threads, is possible using `ThreadPoolExecutor`, but we will not cover that here.

Choosing an architecture

What determines which architecture we should use?

Application requirements

- Protocol (*e.g.* storage required, compute-intensive *etc.*)
- Expected no. of connections at a time.

User behaviour

- Length of connection.
- Typical amount of communication per connection.

Should be careful to separate **design** (*i.e.* client-server architecture) from **communication** (*i.e.* client-server protocol).

- Can re-use **same** architecture for **different** protocols.

Overview and next time

Today we have looked at two types of **thread pool** for servers:

- The **thread-pool model**, where each thread effectively runs its own server, albeit with the same `ServerSocket`.
- The **Executor** service, a high-level system for running multi-threaded applications.

Combined with the **thread-per-client** model from last lecture, these are the 3 architectures we consider in this course.

Next time we will look at a different way to improve server performance - **non-blocking I/O**.