Aims of the tutorial

After completing this week's tutorial you should:

1. Understand the concepts of threading within distributed systems;
2. Be able to correctly incorporate threads into RMI applications;
3. Be able to improve the performance of RMI applications by using threads.

Introducing Multi-threading

Historically, operating systems such as Unix provided the abstraction of a process, i.e. an address space with a single flow of control. It is now realised however that this confuses two distinct concerns. Hence, modern operating systems and programming environments tend to separate out an *execution environment*, as the unit of protection, from *threads* that are the unit of concurrency. In other words an execution environment (address space) can have multiple threads at a given time.

One great advantage of threads is that they provide a *lightweight* model of concurrency when compared to processes. This is illustrated by the following table which provides typical figures for the creation and context switching of processes, kernel level threads and user level threads (kernel level threads are threads created and managed by the kernel, whereas user level threads are created and managed by a user level library):

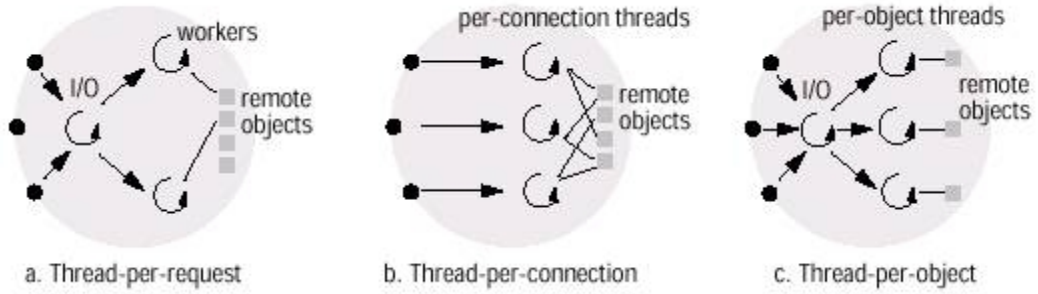
	Thread Creation	**Context Switching**
Process	O(10ms)	O(2ms)
Kernel Threads	O(1ms)	O(0.5ms)
User Threads	O(0.1ms)	O(0.05ms)

Threads, as is normal with concurrent programming, can also increase the *volume* of work carried out in a process by enabling one thread to execute while another one is blocked.

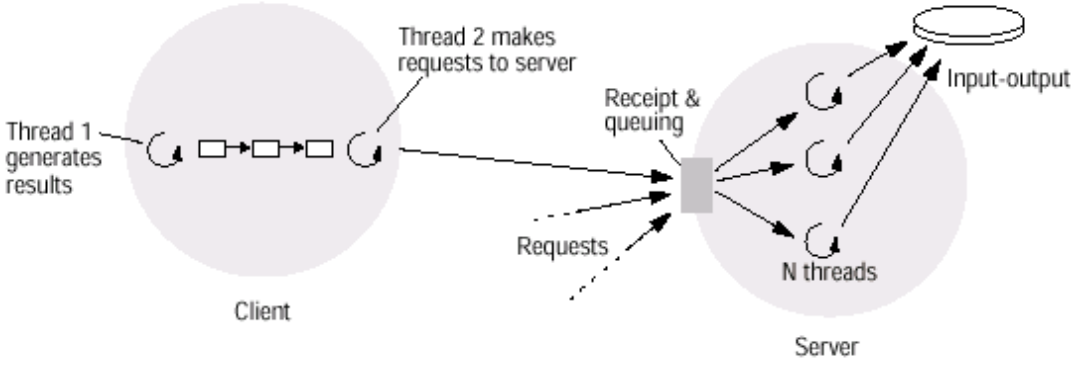
Distributed Programming with Threads

In distributed programming, threads can be used at both the client and the server end of an interaction. We consider each case in turn below.

In the case of *servers*, the main motivation is to increase the *throughput* of the server and (hence) to enhance the *scalability* of the server (a crucial concern in modern distributed systems). A number of different models are possible, in terms of the mapping of threads to incoming invocations. The following diagram (taken from the CDK textbook) illustrates a few of the possibilities:



In the case of clients, the motivation is to prevent clients from being blocked while an invocation is in progress. In particular, one thread can await the result of the RMI while another can continue to do useful work. Alternatively one thread can generate data which can then synchronise with a second thread that performs remote invocations. This latter scenario is depicted in the following diagram (which also shows a multi-threaded server):



Programming with Threads in Java

The Thread Class

The thread class in Java can be found in java.lang.thread. The normal way of using this class is to create a class as a subclass of thread and then override the run() method (the code to be executed by the thread. The following is a sketch of the code required for this purpose:

```
Class MyThread extends Thread {
        ….
        MyThread () {
                // insert constructor here
        }
        public void run () {
                // insert thread code here
        }
}
…
MyThread p = new MyThread ();
p.start ();
```

A subset of the methods defined on threads are listed below:

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Set and return the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
Enter the *READY* state and invoke the scheduler.

destroy()
Destroy the thread.

A complete list can be found in the Java documentation here.

Associated Classes

As with any concurrent programming, it is necessary to provide a level of concurrency control. Java provides a number of primitives for this purpose. Crucially, Java supports the **`synchronized`** keyword that enables the creation of monitors. A given class can have a mixture of synchronized and non-synchronized methods, with the former preserving monitor semantics on concurrent access, i.e. mutual exclusion. Java also offers condition variables, with methods `wait(), notify()` and `notifyAll()`. Finally, the thread class itself has methods for synchronisation purposes such as `join()` and `interrupt()`.

A summary of the key methods is given below:

thread.join(int millisecs)
Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()
Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)
Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), object.notifyAll()
Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Java also supports the thread group abstraction representing a set of threads (in java.lang.ThreadGroup). Every thread belongs to one group, as allocated at creation time. In addition, thread groups are organised into a tree whereby every thread group apart from the original has a parent.

ThreadGroups provide a grouping mechanism in case separate computations share the same JVM. They also support the management of priorities of threads for scheduling.Full details of the ThreadGroup class can be found in the Java documentation here.

Server-side threads in Java RMI

One method of improving a server's performance is to create a thread for each remote method invocation, allowing multiple

calls to be processed concurrently. Therefore, several clients are not blocked in a queue waiting for their call to be executed. Java RMI automatically provides you with this level of server-side threading; this policy is described as follows in the RMI specification:

> *"A method dispatched by the RMI runtime to a remote object implementation (a server) may or may not execute in a separate thread. Calls originating from different clients Virtual Machines will execute in different threads. From the same client machine it is not guaranteed that each method will run in a separate thread"*

Therefore, if you make remote calls from **separate** clients (executing in different JVMs) each call will run in a separate thread. However, if you make concurrent calls from the same client (this can be acheived by using client-side threading, see later) then it is possible these calls will execute on the same server thread.

Task 7.1 Illustration of Server Side Threading in Java RMI

To demonstrate how the threading policy provided by Java RMI works, you will create a simple RMI application that uses two clients and a single server. The concurrent execution of two threads on the server will be displayed to you. To do this, the remote implementation provides two methods; each one displays a set of messages stating that it is running (e.g. "Method X currently executing"). When the separate clients call both methods concurrently the server will illustrate that these two are threaded according to the RMI policy.

To carry out this task, work through the following steps:

First create the server and the remote object implementation that provides the two methods. The code for this is provided for you.

1. Copy the interface file from threads.java and read through the code so you understand what is being provided.
2. Copy the implementation file from threadsimpl.java
3. Compile the two files and create the stub files.
4. Compile the server: threadsserver.java and compile the client: threadsclient.java

N.b. There are no videos this week.

To run the application perform the following:

5. Launch the RMI registry and the threadsserver in separate command prompts.

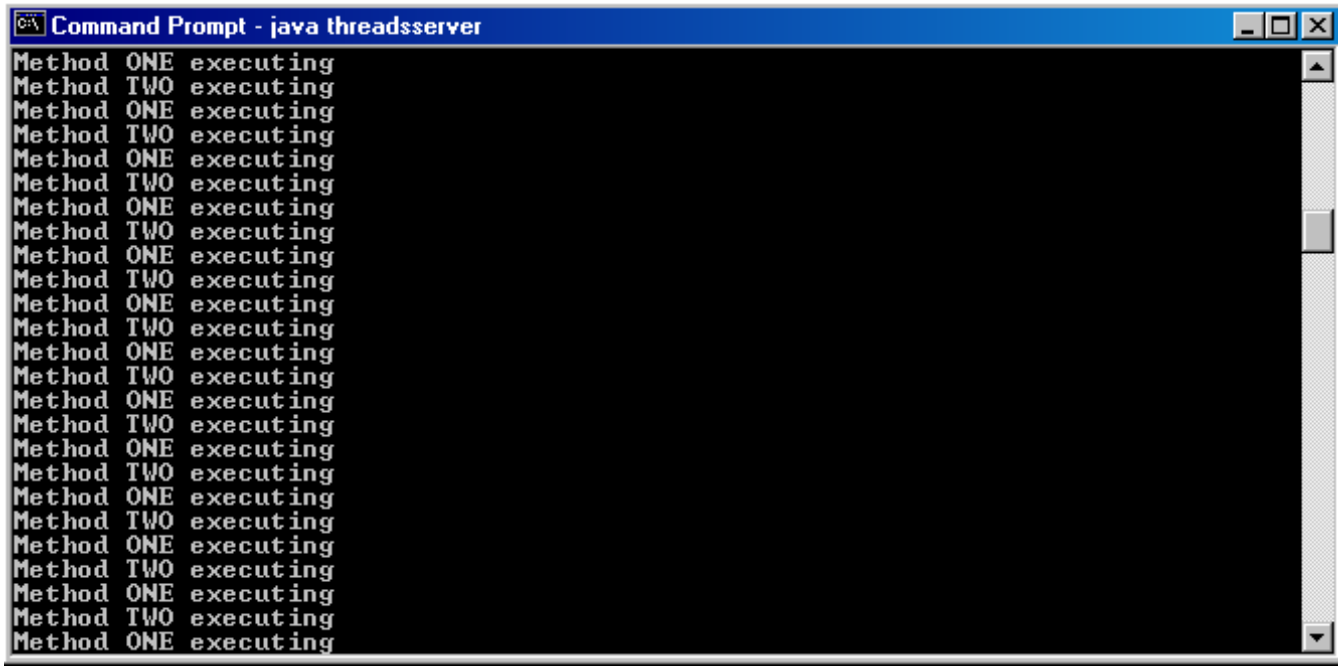
6. Create two new commands prompts and run the client in each of these, using the following commands:

>java threadsclient one

>java threadsclient two

Execute both within a reasonable timeframe, so that they run concurrently.

The following picture of the server illustrates the results you should obtain and demonstrate:



Consequences of the Java RMI threading policy

The Java RMI policy of providing threading automatically within the server is beneficial (it saves you from writing the threading code). However, the ability to allow multiple methods to run concurrently on a server is not thread safe. With such concurrent execution, it is possible for client requests to conflict. For example, two clients could concurrently attempt to write to the same field in a database (or any shared data structure on the server). This may result in one of the writes being lost. Therefore, you must ensure that access to a shared data structure is controlled, so that concurrent client calls (running in separate threads) do not conflict.

Task 7.2 Managing conflicting calls

The second task is to create an RMI application that is safe for multiple thread access to a shared data structure of the remote service. To do this you are required to write a simple "stocks&shares" system. This can have the following features (but you can expand on these if you wish):

- Your remote service (on the server) must contain a data structure that stores one or more shares with the following details:
    1. **Unique share code (e.g. a three letter code)**
    2. **Description of share**
    3. **Current share price**
    4. **Share starting price**
    5. **Number of shares available to buy**

- You must provide a set of methods in the remote service that:
    1. **Returns the current price of a share.**
    2. **Updates the current price of a share.**
    3. **Adds a new share to the service.**
    4. **Returns the share's description field.**
    5. **Returns the number of shares left to buy for a particular share type.**
    6. **Buy shares of a particular type.**
    7. **Sell shares a particular type.**

- You must write a client that calls the remote methods and illustrates the required thread-safe behaviour, see below (i.e. you must implement this behaviour within the sharesimpl code).

    1. Only one thread can write to the data structure at a time.
    2. Threads cannot read while a write is taking place.

An incomplete remote implementation is provided as a starting point for you: shares.java & sharesimpl.java. However, you may implement your own version if you wish.

You must demonstrate the following:

1. The client correctly reads and writes share prices, and buys and sells shares.
2. The system is safe against conflicting thread requests (Use following scenarios as a basis).
    - Two conflicting writes - The implementation in the sharesimpl template illustrates that a concurrent update of a share's price and its description will lose one of the updates. Try this by calling WriteSharePrice in one client first and then BuyShares in a second. Demonstrate that your code is thread-safe by showing that neither update is lost in the above scenario.
    - Read during write - make one client perform a write operation and then another perform a concurrent read. The example code sleeps on the write method before changing the value, therefore a non thread-safe read will not return the most up-to-date value (even though it was invoked after the write operation). You must

demonstrate that the write operation has finished before the read begins (e.g. WriteSharePrice followed by ReadSharePrice, or WriteSharePrice followed by SellShares.

- NB. Feel free to demonstrate other situations that show your implementation is safe from conflicting calls.

Hint: If you need to know more about thread control in Java read the tutorial at the following link: Java Programming: Threads of Control

Client-side threading in Java RMI

When a client makes a call in RMI, it blocks waiting for the result of the method invocation to be returned (This is illustrated well in task 7.1 – the two clients block while the loops run on the server). This is inefficient; if the client does not need the result urgently or there is something else that needs to be processed then it can be performed while the call is blocking. To implement this in RMI, you add threads to your client. That is, one thread is created for the call (it calls the method and awaits the result) and other threads are created on the client to perform alternative tasks (e.g. another remote call).

Task 7.3 Providing client-side threading

Extend the Share Service to include a new method that predicts the closing price of a given share. The code to do this can be found here: forecast.java. Therefore, add this code to the remote object implementation and extend the service interface.

When you call this method from the client you will notice that the request takes time to return a share forecast (between 5 and 30 seconds). This is not efficient, as the client could be reading share values while this is taking place. Therefore, to improve this scenario you must do the following:

- Create a client-side thread that invokes the remote forecast method, awaits its result and displays the information to the screen or stores it in a file.

You should demonstrate that after the forecast method has been invoked your client does not block, but is free to perform other operations (e.g. buy and sell)

Your Tasks (re-stated)

7.1 Create an RMI application that illustrates the server-side threading policy that is provided.

7.2 Write a simple stocks & shares application that makes use of a shared data structure on the server and is safe from multiple client calls.

7.3 Write a multi-threaded client for the Share application.