School of Electrical Engineering & Computer Science
University of Newcastle
SENG3400 – Network & Distributed Computing

Assignment 3

Due using the submission facility of the Blackboard Assignment facility: 11:59 p.m. 28/10/16

NOTE: The important information at the end of this assignment specification

The purpose of the exercise that forms the first part this assignment is to gain familiarity with the use of a programming language (in this case Java) and CORBA to implement distributed objects. Understanding this forms the basis for your solution to Assignment 3.

The template for your solution is provided in lectures, and with the "Hello" example provided in source code form.

Use of the sample code.

Organisational matters: For each project I set up a folder, in this case call it `ass03a`. In this folder place the `.idl` file. Off this parent folder create folders for the client code and the server code (called `Client` and `Server` respectively). These folders are the repositories for your code that implements the client and the server.

The provided sample code is organised in exactly this way, in the folder entitled `IDL Example`. You will note that the .idl file is called `Hello.idl` and that it defines a module called `HelloApp`. This module name defines, amongst other things, the name of the folder in which the idl compiler will place the files it creates.

To create the helper files for the client:

Move to the folder in which the `.idl` file resides

```
idlj –td Client –fclient Hello.idl <enter>
```

To create the helper files for the server:

Move to the folder in which the `.idl` file resides

```
idlj –td Server –fall Hello.idl <enter>
```

To compile the client:

Move to the folder in which the client programmer-produced code resides

```
javac HelloClient.java HelloApp\*.java <enter>
```

To compile the server:

Move to the folder in which the server programmer-produced code resides

```
javac HelloServer.java HelloApp\*.java <enter>
```

To start the ORB:

Move to the folder in which the `.idl` file resides

```
start orbd –ORBInitialPort 2014 <enter>
```

To start the server:

Move to the folder in which the server programmer-produced code resides

```
start java HelloServer –ORBInitialHost localhost
                       –ORBInitialPort 2014 <enter>
```

To start the client:

Move to the folder in which the client programmer-produced code resides

```
java HelloClient –ORBInitialHost localhost
                 –ORBInitialPort 2014 <enter>
```

Popup windows may appear as Windows tries to protect access to the port. You need to indicate that such access is permitted by clicking the appropriate button.

Introductory Exercise:

You will implement a situation in which a server provides clients with descriptions of products maintained in a warehouse on provision of client-provided references to product objects. In this simple example the client knows that there are two remote `Product` objects, and obtains references to these

objects from the `NameService`. The references are registered with the `NameService`, together with the objects' logical names "`first`" and "`second`", by the server application.

An instance of `Product` stores a private `String` that describes the product, and provides an external method `getDescription()` that allows callers to retrieve that `String`.

Initially you will produce an .idl file defining a `module` and the interface of the `Product` class. Then you will implement a server program that creates two instances of `Product` and makes them available to remote clients by registering their names "`first`" and "`second`". Lastly you will implement a client program that obtains references to the remote `Product` objects by looking up their names, calls their respective `getDescription()` methods, and reports the `String`s returned for each call.

YOU ARE NOT REQUIRED TO SUBMIT A SOLUTION TO THIS INTRODUCTORY EXERCISE

Assignment Question

Introduction

Prior to introducing Assignment 3, I will provide notes on the issue of forms of request synchronisation you will demonstrate in the assignment. You should read these notes carefully prior to starting work on the assignment.

Distributed Requests

Standard CORBA object requests are all synchronous, in that, after making a request of a server object, a client blocks its own execution while a server object executes the request. The client restarts its own execution either (1) when the server object completes the requested operation and returns to the clients or (2) when the middleware informs of an error.

The fact that we prefer to gain the highest possible performance from distributed applications leads us to examine the suitability of synchronous calls in some circumstances. For example:

- If server response times exceed client response-time requirements
- If there is nothing to be gained by blocking the client (e.g. when printing)
- If tasks can be executed in parallel by different servers

This leads us to the definition of alternate client-server synchronization paradigms, namely the so-called *one-way request*, *deferred synchronous* and *asynchronous* forms.

One-way Requests

These return control to the client as soon as the client's call is accepted by the middleware. From then on the client never receives any feedback on the result of the call. Such requests are suitable if the client semantics do not depend on the result of the requested operation, for example if the server result is not used by the client, or the called operation cannot violate the client's integrity.

One-way requests can be defined in IDL by declaring the called method as `oneway`. This results in the creation by the CORBA Dynamic Invocation Interface (DII) of an object of type `Request` every call. The `Request` type supports the `no-wait` parameter for the `send()` operation, and this feature can be used to achieve the required behaviour. The client can `delete()` the `Request` object without using its `get_result()` method call to learn of any returned outcome of the call.

The use of CORBA `oneway` has implications for the use of server functionality by clients. Firstly, the definition of the interaction in IDL forces every client to use the server in that way, reducing flexibility for client implementers. Additionally, the creation of a `Request` object every time a call is made is inefficient and has an adverse effect on performance. Finally, the programmer of a client may wish to use a

service in a one-way manner when the implementer of the server (and/or writer of the IDL) did not define it that way, an impossibility!

An alternative is available to clients whose language and execution environment supports threads.

Use of Threads in Alternate Synchronisation Paradigms

One-way requests can be implemented for threaded client languages by having the client create a thread whose purpose is to make the server call, and then block on it. While the thread is blocked, the client can, of course, continue executing and the calling thread can be allowed to 'die' in due course without the main client program ever conferring with it again. Similar techniques can be used to implement deferred synchronous and asynchronous calls.

Deferred Synchronous Requests

Deferred synchronous requests are suitable if the client needs the result of the call to the server object, but it can do something else pending the return. Thus the client can perform other computations, or can issue other concurrent requests while server(s) are at work servicing other requests by the client.

Deferred synchronous requests can be implemented using CORBA `Request` objects if that is preferred by the implementer of a service, or if the client programming language does not support multiple threads. The client simply makes the service call, then, when ready for the result, requests it from the `Request` object using `get_result()`. If the server has not yet returned to the `Request` object, the client can either block (usually the appropriate action given that it needs the result to continue) or go into a loop in which it continually calls the `Request` object. This client-initiated request for the result of a call is termed *polling*.

Asynchronous Requests

Asynchronous requests differ from deferred synchronous in that the client does not poll for a return from the remote call. Rather the `Request` object calls the client with the result. This requires that the `Request` object has a reference to the client to effect the call providing the client's result, and that the client implements a method suitable for the return call.

Message Queues

The previous discussion of non-synchronous requests assumes that some entity waits on behalf of the client while the server object executes. This has the advantage that the client can be doing something else while it 'waits'.

In some situations caused by, for example, high network latency (e.g. the Internet) or different time zones, the requests may be implemented as synchronous, when in fact there is a large time gap between client request and service provision. An example of this situation is a Sydney stock broker, who may make requests on the London stock exchange

that need to wait until that exchange is trading (noting that Stock Exchanges around the world operate in different time zones).

This situation can be managed by middleware provision of a *message queue* service. The clients, and the server, maintain their own queues onto which messages can be added and from which they can be taken. These are organised as FIFO lists, so that messages are guaranteed to be delivered in order of receipt.

Assignment Task

Your task in this assignment is to implement one or more programs that demonstrate *synchronous*, *deferred synchronous* and *asynchronous* interaction between a client and server. You will use Java IDL to produce your solution to this assignment. You can either implement the synchronisation in your client program code or through the use of the Java IDL `Request` objects that exist for this purpose. This also simulates the situation in which a client you are writing requires more than synchronous interaction with an already-implemented server that offers only synchronous (i.e. blocking) calls.

Your IDL file will be called `demo.idl`, and you will also create classes called `DemoServer.java` and `DemoClient.java`. It will be necessary for you to produce other classes for which you may choose your own meaningful names.

Your **server** will simulate real life propagation and load effects by introducing random-sized and discernable (by the user) delays between receipt of requests and responses to those requests. The server's function is to provide a 'random character' service that, on request, returns a randomly selected alphabetic (i.e. 'a' ... 'z', 'A' ...'Z') character. Your client will execute a loop, each iteration of which causes a counter, and the content of a `private` (to the client) variable to be displayed.

With regards your **client**, the value of the `private` variable starts off at '*', and after 5 iterations of the above-mentioned loop, a call is made to the server – the value returned by that call will be used to update the value of the variable.

- In the case of *synchronous* interaction between client and server, the loop pauses until a return is received from the server call. Synchronisation will be demonstrated by a change in the value of the `private` variable content shown at iterations 6 to 10, after which the loop terminates.
- In the case of *deferred synchronous* interaction between client and server, the loop continues for a further 5 iterations after the server call before synchronising with the server (this may cause a pause in operation of the client). Synchronisation will be demonstrated by a change in the value of the `private` variable content shown at iterations 11 to 15, after which the loop terminates.
- In the case of *asynchronous* interaction between client and server, the loop continues for an indeterminate number of iterations after the call to the server, and for 5 iterations after the `private` variable changes as a result of a return from the server call. The asynchronous nature of the interaction will be demonstrated by unpredictability of the iteration counter value at the time of change of the `private` data value.

An example client screen display is as follows:

```
Demonstrating synchronous interaction
 1.  Value is *
 2.  Value is *
 3.  Value is *
 4.  Value is *
 5.  Value is *
Call to server
 6.  Value is D
 7.  Value is D
```

```
 8.  Value is D
 9.  Value is D
10.  Value is D
```

You are only required to provide support for a single client-server interaction, i.e. your server will not be interacting with multiple concurrent clients.

Your submission should be made using the Assignments section of the course Blackboard site. **Incorrectly submitted assignments will not be marked. Assignments that do not use the specified class names will not be further marked.** You should provide all your .java files. Also provide a readme.txt file containing any instructions for the marker. In particular you should describe to the marker how your program's output demonstrates compliance with the assignment specification. Each program file should have a proper header including your name, course and student number, and your code should be properly documented. Finally, a completed Assignment Cover Sheet should accompany your submission.

This assignment is worth 15 marks of your final result for this course.

Dr. Mark Wallis