The University of Western Australia
School of Electrical, Electronic & Computer Engineering
Real-time Distributed Computer Systems 423
Laboratory - 2003

Introduction to Distributed Object Technologies: CORBA, RMI & .NET
Introduction to the Design Project

PART 1: INTRODUCTION TO DISTRIBUTED OBJECT TECHNOLOGIES

Aim

The aim of the first part of this laboratory session is to familiarize yourself with the CORBA 'middleware' distributed processing environment, which will be used as the primary distributed processing platform in the main project-based part of the laboratory. It is also an opportunity to do some basic performance benchmarks on a CORBA platform and a comparison with RMI.

The capabilities of the OMG CORBA standard ORB have been discussed in lecture material. This lab uses a "mostly CORBA 2.3 compliant" ORB supplied with Sun's J2SE V1.4.

This lab is conducted in the 3.09 IT Teaching Lab. You should reboot a PC to Windows XP (if booted to Linux) and log on in your student account. If you don't have an account, or can't log in, see computer support (preferably before the lab starts).

Getting started

All J2SE documentation is on-line. There are also links to some basic CORBA tutorial introductions on the unit web page.

Manual compilation of a first CORBA client/server program

Although we are going to use *Sun ONE Studio* for later work, it is useful to explore manual compilation first (since for, example, it can be a lot faster to use if you are used to JDK, but it doesn't offer the same debugging environment, or project manager facilities). We will try both methods so you can use whichever suits you better at the time.

Directory setup & file copying

Assuming that you have a home area on the PCs on the *H:* drive, create an *H:\rdcs423\corba* directory to use for your lab programs. Bring up a *Command Prompt* console and then copy the source files for the *grid* example, i.e. all files and directories from *U:\rdcs423\corba\grid* to *H:\rdcs423\corba\grid*

IDL compilation

In the DOS window and directory *H:\rdcs423\corba\grid*, run the IDL compiler with:

> *idlj –td java_client idl\grid.idl*

If you get error here it is most likely because you are in the wrong directory or a path is not setup in the environment for you (e.g. *set PATH=%PATH%; C:\Program Files\s1studio_jdk\j2sdk1.4.1\bin*).

The IDL compiler should have generated the client side IDL interface java files. Then run:

> *idlj –fall –td java_server idl\grid.idl*

Which should generate the server side IDL interface java files without errors.

The *–td* switch specifies the target directory for the generated files, and the *–fall* switch is required to generate server and client side files). An *–fserver* switch generates server side files only.

Have a look in the *java_client/grid* and *java_server/grid* subdirectories at the generated client stub code and server skeleton code.

Client and Server compilation

We now need to compile all client and server programs. In the *H:\rdcs423\corba\grid\java_client* directory run:

> *javac –d classes Client.java grid*.java*

Change directory to *H:\rdcs423\corba\grid\java_server* and run:

> *javac –d classes Server.java grid*.java*

The *–d* switch directs class file output to the *classes* subdirectory. Now have a look at the class files in *H:\rdcs423\corba\grid*. Apart from the *Client.class* and the *Server.class* (and its implementation class *GridServant.class*) all the other class files arise from compilation of the IDL generated java files. Note that the reason for compiling client and server files in separate steps is that later on you may only want to make changes to one or the other but not both.

Manual execution of a CORBA client/server program

Run Naming Service

The CORBA naming service must be running before registering the server with it, so from a new DOS window run:

> *tnameserv*

Note that a *–ORBInitialPort* switch is optional here (and is set to port 900 by default), and that you only need to run the nameserver once and leave it running. From J2SE v1.4 an *orbd* persistent name server has also been introduced (but *–ORBInitialPort 900* needs to be specified if you use that).

Register Server

The first thing to do is register the server with the ORB and then advertise it with the naming service so that our client will be able to find it.

In a DOS window go to the *java_server* subdirectory for the grid example and run:

> *java –cp classes Server*

The grid server should now initialize the ORB and create the grid object, bind a name for grid server with the name service and then hand back control to the ORB to process incoming requests. Note that if the nameserver is run with a non-default *ORBInitialPort* switch, then the Server needs to be also be run with the same *ORBInitialPort* value.

Client execution

Open up another DOS window and go to the *java_client* sub-directory and run the grid client program with:

> *java –cp classes Client*

You should see something like this:

```
Initializing ORB
Obtaining name service object reference
Obtaining grid server object reference
Grid width is 100
Grid height is 100
Grid value at x = 2 y = 4 is 123
```

i.e. the client contacts the naming service to get an object reference to *gridServer*, then converts this string into an object reference, then you get the grid dimensions output and the execution of the set and get methods.

We could go on now to look at operation of this example across multiple processors, but lets delay that step to introduce another simple example and to look at Java/CORBA development using Sun ONE Studio. You can terminate your *Server* with a ^C before continuing, but keep the nameserver running.

Another CORBA example using Sun ONE Studio for Java

Directory setup & file copying

We can use *Sun ONE Studio* to run another CORBA client/server program, this time to look at a basic performance benchmark. Creating another *count* sub-directory in your area, copy over the directory *U:\rdcs423\corba\count* to *H:\rdcs423\corba\count*.

Open project

Now run *Sun ONE Studio*. When you do this for the first time it will bring up a panel to ask you where to setup your project files, if you specify *H:\Sun ONE Studio* all should be well (i.e. it just needs to somewhere in your home area where project files can be stored). Create a new *Count* project with *Project/Project Manager/New*, then use *File/Mount Filesystem* three times to mount the three filesystems:

H:\rdcs423\corba\count\java\idl

H:\rdcs423\corba\count\java\java_client
H:\rdcs423\corba\count\java\java_server

Compilation

The IDL program can be compiled as before by bringing up a DOS prompt, then change directory to the *H:\rdcs423\corba\count* directory and generate java stub and skeleton files with:

> *idlj –td java_client idl/count.idl*
> *idlj –fserver –td java_server idl/count.idl*

Going back to Sun ONE Studio's *File Explorer* view, select the *H:\rdcs423\corba\count\java_server* directory, then right click, and select *Build All*. Hopefully you won't see errors as it compiles. Then select the *H:\rdcs423\corba\count\java_client* directory, and *Build All* again.

Execution

Still in Sun ONE Studio's *File Explorer* view, select the *Server* class in *H:\rdcs423\corba\count\java_server*, right-click, then *Execute*. Wait for the Server to indicate that it has started ok, and then restore the *Explorer view* by clicking on the *Editing* tab, select the *Client* class, and then right click and *Execute*. You can observe execution messages from both the client and server in the Output Window, by selecting the appropriate I/O tab.

Run the client a few times to get some idea of the average "ping" time, it should be around a few milliseconds or less. Recall in the course notes that we quoted about 3 msec, but that was on a P120 machine. Also note that the Execution window shows that the Server class is still running (you can right click on the *Server* class in the Execution window and *Terminate Process* if you wish to do so).

Execution of the example CORBA client/server programs on different processors

The Grid example again

You have now completed the initial examples that you can do separately with CORBA, and you now need to coordinate with a colleague using another PC. Firstly, let's go back to the simple *Grid* example. It should work in exactly the same way across different processors. The only requirement is that the PCs must share the same ORB (i.e. have the same *ORBInitialPort* value and have one nameserver running that uses the same value).

The *ORBInitialHost* switch must be specified on the client and must match the PC name on which the nameserver is running. The switch is optional for the *Server* if it is running on the same PC as the nameserver.

Assuming the nameserver is running on *pc60*, run the server on either machine with:

> *java –cp classes Server –ORBInitialHost pc60*

Run the client on the other machine with:

> *java –cp classes Client –ORBInitialHost pc60*

And that's it - simple client/server under CORBA. Let's go back to the *Count* example to make sure we can do that with *Sun ONE Studio* on different processors as well.

The Count example again

The Count example works in exactly the same way. You can run *Sun ONE Studio* now on both the designated server PC and any client PC and open up the *Count* project. The only change we need to make is the switches in Sun ONE Studio used for external execution

In the *Properties Window* for the Client, right click with the *Client* selected, select the *Execution* tab and in the blank *Arguments* list we need to at least add (assuming the designated server PC is *pc60*):

-ORBInitialHost pc60

We can specify the *ORBInitialPort* here as well if this is different to the default. Run the *Client* program on any PC and the *Server* program on the same PC as the nameserver (so the Sun ONE Studio configuration is unchanged here). Of course, note the average ping time and the variation from the time recorded for local Client/Server operation. You may find that the local ping could even be faster. That's because the observed performance can have more to do with working through two JVMs on two machines versus one JVM on each machine, than network overhead.

Callbacks under CORBA

The last technical example using CORBA is to extend the count server to perform a callback method on the client. This is particularly useful where we want the client to asynchronously receive information when it becomes available on the server (and not just when the client requests it).

Create another *count_callback* sub-directory in your area and copy over the directory *U:\rdcs423\corba\count_callback* to *H:\rdcs423\corba\count_callback*. At a Command Prompt change directory to the *H:\rdcs423\corba\count_callback* and do the *idl* compilation with:

> *idlj –fall -td java_client idl/count_callback.idl*
> *idlj –fall –td java_server idl/count_callbact.idl*

Note the use of *–fall* switch to generate both client and server stubs/skeletons in each subdirectory.

In the Sun ONE Studio IDE, create a new *CountCallback* project with *Project/Project Manager/New*, then use *File/Mount Filesystem* three times to mount the three filesystems:

H:\rdcs423\corba\count_callback\java\idl
H:\rdcs423\corba\count_callback\java\java_client
H:\rdcs423\corba\count_callback\java\java_server

In Sun ONE Studio's *File Explorer* view, select the *H:\rdcs423\corba\count_callback\java_server* directory, then right click, and select *Build All*. Hopefully you won't see errors as it compiles. Then select the *H:\rdcs423\corba\count_callback\java_client* directory, and *Build All* again.

Now execute the Server then the Client, and confirm operation of the callback of the server to the client. You should examine the code and note that two methods of passing the object reference of the callback servant (in the Client) are demonstrated; one uses a specific method ob the server to pass the callback servant's object reference, and the other simply registers the callback servant with the naming service, and the server then looks it up in the same was as the client looks the server up.

A Java Remote Method Invocation (RMI) comparison

[Note: Not operational in 3.09 this year due to Windows XP security model clashes in user mode -> still review this section – you can try it later on any PC where you have admin access]

If you have totally forgotten what RMI is about then you should consult the introduction in the Java tutorial on this topic. You can find a link from the *rdcs423* unit web page.

Get source files

Get a copy of all files from *U:\rdcs423\countRMI* to your area *H:\rdcs423\countRMI*.

Compilation

Bring up a DOS prompt and *cd* to the *countRMI* directory, and run (noting that its is case-sensitive):

> *rmic -d . -classpath . CountRMIImpl*

This creates the RMI stub and skeleton classes that the Client and Server will use respectively. Now compile the Client and Server classes with:

> *javac *.java*

Start the RMI registry and register the Server

Also from the DOS prompt in the *countRMI* project directory run the RMI registry with:

> *start rmiregistry*

To run the server use the provided batch file (with a parameter indicating that the server is available on the locahost):

> *Server localhost*

Note that if the server fails to run and you get exceptions at this stage, it is most likely due to an embedded path in the *Server* batch file (i.e. it assumes the codebase will be in *H:\rdcs423\countRMI* as specified above). If you examine this batch file anyway you will see that a security policy in *java.policy* is specified, and the contents of this file are:

```
grant {
        permission java.net.SocketPermission "*:1-65535", "listen,resolve";
        permission java.net.SocketPermission "*:1024-65535", "connect,accept";
        permission java.net.SocketPermission "*:80", "connect";
```

};

which allows remote connections to the TCP port number (usually > 1099 used for RMI connections).

Run the Client

Run the client with the provided batch file:

> *Client localhost*

and you should see something like this for the client:

```
Trying hostname: localhost
Setting Sum to 0
Incrementing
Avg Ping = 0.741 msecs
Sum = 1000
```

Your average ping time will differ. Recall that we quoted about 6 msec in the course notes, but that was on a P120 machine (and Java JIT compiler and JVM environments have improved since then).

Ping your server from another machine

Now terminate your running Server and rerun with the actual PC name you are on with:

> *Server pcname*

Then from another PC run:

> *Client pcname*

where *pcname* is the same (i.e. the Server PC name)

Note the average ping time. Again try a few runs for both remote and local, and you can again expect some variation. The comparison with CORBA performance is of most interest though.

A .NET comparison

The aim here is to gain some exposure to practical use of Microsoft's distributed .NET architecture on the Windows XP operating system.

Copy the files from the *U:\rdsc423\dotnet\count* area to *H:\rdcs423\dotnet\count* area and via selecting *Programs/Microsoft Visual Studio .NET/Visual Studio .NET Tools/Visual Studio .NET command Prompt* open a command window.

Change directory to the count area and compile the files with:

> *nmake*

You will notice that in addition to client and server executables and a library (.dll), there is an additional file, *HiResTimer.netmodule*. This is a timer module used to measure the performance of the .NET platform; it is added to the client application when *PingClient* is compiled.

Now run the server by typing at the prompt:

> *PingServer*

Opening another .NET command window, as previously, run the client by typing:

> *PingClient*

The remote object is created, along with a new timer, which will measure the performance of the .NET ping by timing the execution of the *Increment()* function, called on the object by the client. The result is sent back to the client (in this case, the object is an integer, incremented from 0 to 1000), and the time taken is output to the command window.

Now open the file *PingClient.exe.config* which is the configuration file that the client uses to specify the location of the remote object. It allows the user to alter the configuration of the client/server communication without having to recompile the applications. Change the server name in the URL line from *localhost* to the name of the machine on which the server is running, and save the file. Then rerun the client.

Changing Protocol:

The above examples illustrate the principle of remoting using the TCP protocol. Remoting using the HTTP protocol is virtually identical. Using the Count example, we can connect the client and server using HTTP.

Open the file *PingServer.exe.config*. Now change the channel reference from *tcp server* to *http server*. Do the same for the file *PingClient.exe.config*, as well as changing the URL reference to *http* instead of *tcp*.

Now run the client and server as before, noting the difference in performance as compared with using the TCP protocol.

CORBA and Rational Rose

Rational Rose provides some support for CORBA application development, mainly through the IDL. It is useful to introduce you to this (and it is a bit more practice with Rose). Run Rational Rose and go to *Tools/Model Properties/Edit/CORBA/Project* and select *Include Path* and set it to where you have located your *Count.idl file* (e.g. *H:\rdcs423\corba\count*), click elsewhere, then OK.

Now go to *Tools/CORBA/Reverse Engineer CORBA* and select *Count.idl*, select *Add* to add it to the lower panel and then select it. Hit *Reverse* then *Done*. This should produce a Count interface into an empty Rose model. It should also automatically show up as both a class and component in the left browser panel.

To make the interface visible in the actual class and component diagrams, click on *Logical View/Main* in the Browser and from *Query/Add Classes* select the *count* package, then select *Count* class, then >>>> the OK. Then click on the *Component View/Main* and from *Query/Add Components* select the *idl* package in the scroll box. Then select the *Count* component, select >>>> then OK. Note the component shows up in the component view as the typical lollipop stereotype (as an interface), while in the class view the interface method and attributes should be displayed.

Finally you can browse the *Count.idl* code if you (still in the Component view) right-click on Count and with the CORBA option, select *Browse code*. This just shows you how to reverse engineer a CORBA idl file into Rose - you can also expand the attributes (sum) and operations set & increment) of the Count interface to see more detail of this.

You could now proceed to add other aspects of the Client and Server java programs (which aren't specific to CORBA of course). As a model to have a look at, there is an existing partial model file for the count client and server in *Count.mdl* (you will also find this in *U:\rdcs423\corba\count* - so get it from there and load it into Rose). The *CountClient* and *CountServer* are just java classes, while *Count.idl* is a CORBA class (which is a Java interface class). You could try doing something similar for the Grid example (but leave that until after the lab for now).

That effectively completes the "technology primer" for the project part of the lab. In the next section we get to grips with the target hardware and the distributed application problem you are being asked to solve.

PART 2: PROJECT INTRODUCTION

Aim

The aim of the group project part of the lab, which extends over the next seven lab sessions, is to design, develop and test a distributed lift controller software system for a set of four adjacent lifts all servicing the same floors in the same building. The distributed software infrastructure to be used is CORBA. The programming language to be used is Java, and you are required to implement a multi-threaded solution. The software development process to be used is COMET and you should use UML as much as possible to document your software. You also need to give particular attention to how you are going to control and test your system. For example, you will need a supervisory interface on the server to initiate and monitor operation of the system (which you can envisage as operating in the plant control room of the lift system).

Since this laboratory is not about interfacing to the lift hardware, you are given an Application Programming Interface (or API) to the lift hardware that does this job for you. As a simple start, you are also given the first step in a UML model and the associated Java code. This first step just initializes the lift interface and drives the lift to floor 0 ready for normal operation. The simple use case is written from the perspective of the lift system supervisor who has to update the software and manually restart the multiple lift system individually (and the setup is identical for all lifts in the set). Your job is to provide the lift system controller software for the lift system supervisor to install. The supervisor also has to be able to monitor the status of the lift system and control it from a mobile PDA with wireless (your choice of WiFi and Bluetooth) that she uses when walking around various parts of the building. It should show a mimic panel of the status of the lift system with same manual overrides to stop and restart it. This can be a very similar interface in functionality to the lift system server.

Hardware Description

The four-floor lift models to be used in this laboratory are a scaled representation of a commercial lift. They have similar floor and lift controls to the full-sized implementation (but do not have a separate floor display on each floor – since this doesn't really add to the functionality of the lifts).

The Lift:

A. 1 × lift motor – selectable up/down, selectable two speeds fast/slow

The Floor Panel:

B. 6 × lift call request buttons (only one at the top and bottom floors and two on the two intermediate floors)
C. 6 × lift call request acknowledgment display LEDS (only one at the top and bottom floors and two on the two intermediate floors)
D. 6 × lift call answer display LEDS (only one at the top and bottom floors and two on the two intermediate floors). Note that these LEDS effectively indicate that the lift is currently responding to the lift call (i.e. it is now moving in the direction shown and will stop at this floor)
E. 7 × floor sensors (one on each floor, and one between each floor)

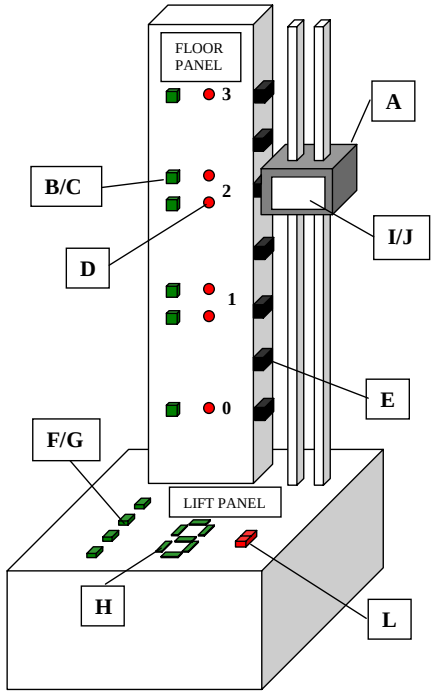
The Lift Panel:

F. 4 × lift floor request buttons

G. 4 × lift floor request acknowledgment display LEDS
H. 1 × lift floor display LED array
I. 1 × lift door open light
J. 1 × lift floor arrival bell

Additional controls and displays (not under software control):
K. 2 × limit switches at extreme ends of the lift travel to disable the motor
L. 1 × motor override switch to manually drive the motor away from the limit stops
M. 7 × diagnostic LEDS for all floor sensors on the controller PCB
N. 7 × diagnostic lift call request LEDS on the controller PCB

PC interface:
- A 48-bit digital I/O PCI bus card (using 2 x 8255s)
- Windows NT device driver in C
- Native interface to a Java Application Programming Interface (API).



Lift Hardware and API Test Program

It is just possible that a lift unit could give some problems, e.g. a lift may no longer stop at a floor, or a button press may no longer be picked up. A quick test with the *LiftTester* program

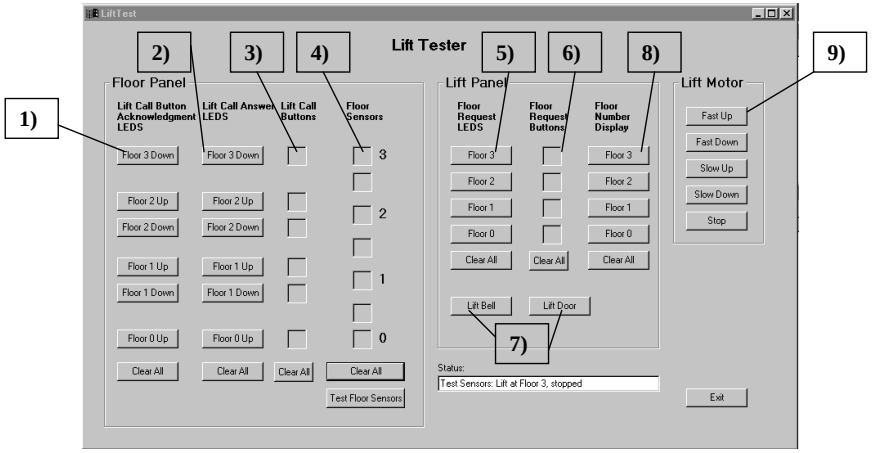
may be in order. You will find this program in *U:\rdcs423\Lift*. If you do find a problem, report it, although it should be possible to work around the problem for the remainder of the lab session.

As an exercise to test out the lift hardware systematically, just use the test program in the sequence left to right in the panel shown over the page. Note that actions are latched for display, and are then cleared with a Clear All button on the panel.

Floor Panel:
1) Check all the Lift Call Acknowledgement LEDS by pressing the appropriate buttons on the LiftTester panel
2) Check all the Lift Call Answer LEDS by pressing the buttons on the LiftTester panel.
3) Check all the Lift Call Buttons be pressing the buttons on the Lift and checking the corresponding display on Lift Tester.
4) Check all Floor Sensors by pressing Test Floor Sensors which will move the lift past all sensors in both directions.

Lift Panel:
5) Check all Floor Request LEDS by pressing the buttons on the LiftTester panel.
6) Check all Floor Request Buttons be pressing the buttons on the Lift and checking the corresponding display on Lift Tester.
7) Check the Lift Bell and Lift Door by pressing the buttons on the LiftTester panel.
8) Check the Floor Number Display by pressing the buttons on the LiftTester panel.

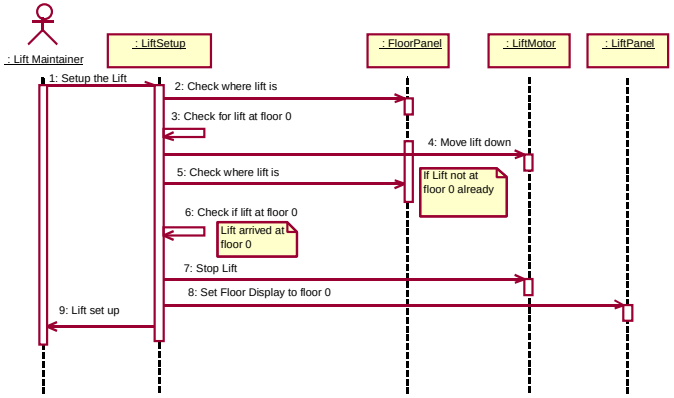
Lift Motor:
9) Check the Lift Motor operation: High/Low Speed, Up/Down and Stop



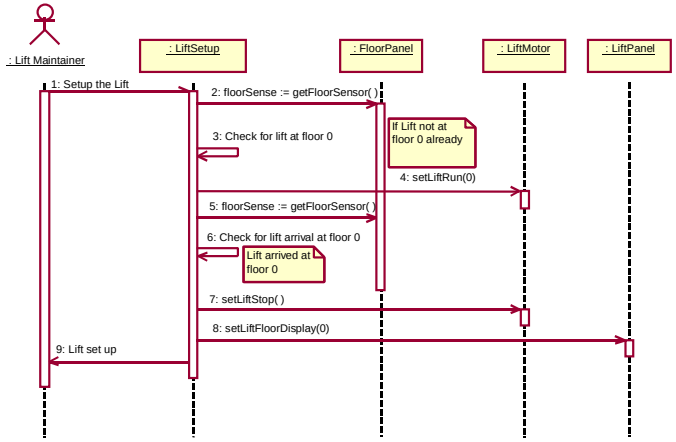
Lift Setup Model and Java Code

As far as setting up the lift for normal operation is concerned we can assume an authorized repair person (a Lift Maintainer) is able to shut the lift down, update the software, and restore the lift to normal operation. This use case captures the restoration of normal operation (i.e. before passengers are able to use the lift at all).

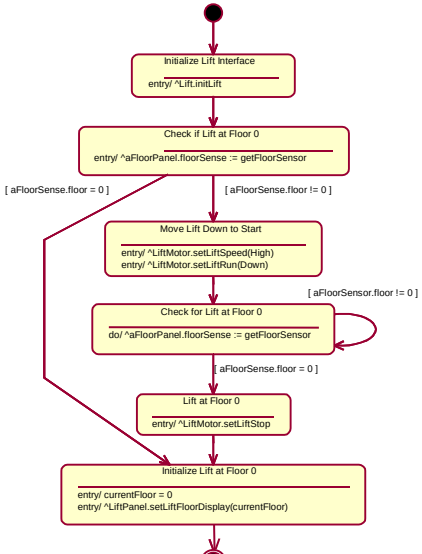
A scenario is defined in the form of a sequence diagram which details the messaging between Lift Controller API objects in this scenario:



The messages shown above are then mapped to operations on the classes (see the detailed *LiftAPI* package description below for the definition of these operations). These operations are mapped to Java methods on the *LiftAPI* package classes.



From the sequence diagram, it is clear that the *LiftSetup* class has internal state behaviour, which can be captured with a State Diagram. This behaviour is so simple that such a diagram is hardly warranted, but it serves here to illustrate a useful starting point to a larger state diagram which describes the behaviour of the *LiftController* class (or possibly several interacting classes with their own state diagrams depending on the design).



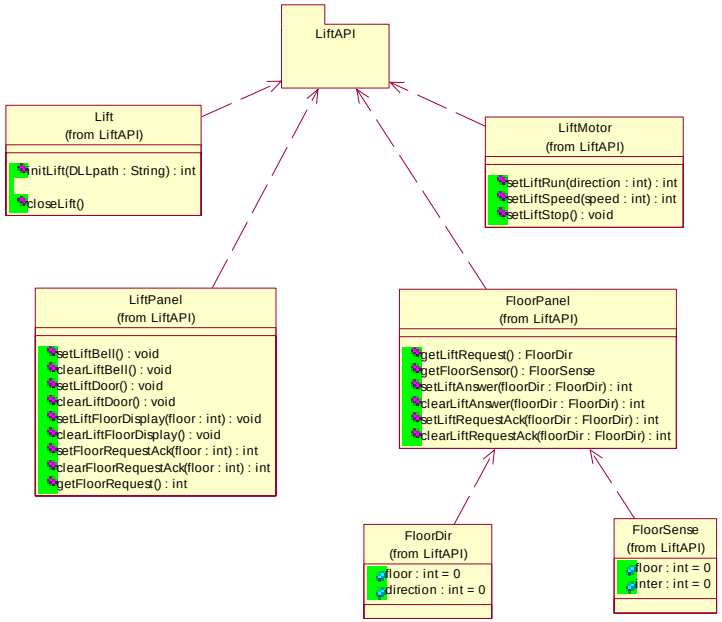
To use the *LiftAPI* package, any Java class (*LiftSetup* in this example) needs to import it (with *import LiftAPI.**). Copy the contents of the directory *U:\rdcs423\Lift\LiftSetup* to your area (say *H:\rdcs423\Lift\LiftSetup*). Run Sun ONE Studio and using *Project Manager/New* create a project called *LiftSetup*. Mount the filesystem *H:\rdcs423\Lift\LiftSetup* for this project. Then use *File/Mount Filesystem/Mount JAR File* and select *H:\rdcs423\Lift\LiftSetup\LiftAPI\LiftAPI.zip* for mounting and inclusion in the classpath. Now compile and run *LiftSetup.* When you create other new projects to implement your Lift Controller remember to include the *LiftAPI* package in the classpath (i.e. mount the JAR/ZIP filesystem).

The corresponding UML model file with all the diagrams illustrating this section can be found in *U:\rdcs423\Lift\LiftSetup\LiftSetup.mdl*. Have a look at it with Rational Rose. If you have got to this point with time to spare and want to try something else, have a go at modifying *LiftSetup* to implement a simple cyclic mode of lift operation (say initiated by and stopped by a button press)

Java Lift API Overview

The Java API is described by the following UML class model. It is contained in a Java package *LiftAPI*, and is made up of six classes which separate the lift API classes into the three main functional areas of *LiftMotor*, *FloorPanel* and *LiftPanel*, the *Lift* itself and two utility classes *FloorDir* and *FloorSense*. The detailed LiftAPI description is contained in the Appendix. The classes make available a total of 20 methods and 4 attributes as indicated. Each method is defined below. To use it, it is only necessary to call the *initLift()* method which does some initialization of the hardware and lower-level interface software. Strictly, when finished with the

Lift, the *closeLift()* method should be called. Of note, all these methods are implemented as static methods, since in this application each local controller interacts with only one physical lift object so the object and its interface package (and classes) are 1-1.



Lift API Operation

Although you don't need to be concerned about the detail of the low-level hardware interface to the lift you do need to know that the lift hardware does not store events (e.g. button presses or lift arrivals) so you need to ensure your software does not spend unduly long not processing lift related operations, or lift related events could be missed.

iPAQ PDA information

You don't actually need to know a great deal about the iPAQ's operation although an on-line user manual will be provided. What you do need to know about is the embedded Visual Basic (eVB) programming environment (an example application describing the development process is provided in a document available in the labs, see below). In addition, a small example Java server program is provided that interfaces to the iPAQ's via TCP sockets – this provides you with a way of interfacing to the iPAQ using the Java language on the PC's (client/server) and eVB on the iPAQ clients. All the application software and documentation for iPAQ development can be found under *U:\rdcs423\iPAQ*.

Project Teams

The lab is run on a project team basis, i.e. all members of each group (3 or 4 per group) work on the same project. How you divide up the problem is up to your group, but initially at least, it makes sense to divide up the problem into some obvious areas: analysis of the provided information and the requirements, design of the solution, prototyping in some areas (e.g. GUI, CORBA functionality, etc). Once the design is complete, task decomposition is centered on distribution of coding functions, integration and testing, and finally documentation completion.

Assessment

The laboratory assessment in this unit is weighted at 30% of the total RTDS408 unit mark. Assessment is based on a group lab report that is contributed to by all members of the group and a group demonstration of the working system. All members of the group should be present at this demonstration. The report will have a standard blue front cover sheet specifying the percentage contributions of all group members and it also must be signed by all group members or any not signing the assignment will get zero mark for it. There should be a paragraph each near the front of the report written by each team member describing their individual contributions to the project. The format of the report is up to your group but it should contain:

1. A software requirements section, and a software design section. You should seriously consider using Rational Rose to provide the required diagrams (and then import them into a Word document).

2. Any design decisions made in producing a software solution should be documented to demonstrate application of the design approach.

3. A test specification for the software should also be included since the project demonstration will be to this test specification (and the degree of compliance/non-compliance shown).

4. The performance of your solution will be assessed. However, this should not be seen as the overriding requirement (correct functional operation and reliability are also important attributes). You can use the timing statements from the count example to time how long it takes the lift starting at floor 0 to fulfill a passenger request on floor 3 when there are just prior outstanding floor requests to all floors and lift calls from all lower floors (this is clearly a stress test and should be part of your test specification anyway).

5. Each software component (task or module) should have a brief unit specification.

6. A listing of all operating code and any generated documentation should be included in the report and stored to your allocated group area on the PCs. This area may be examined as part of the assessment. Use individual accounts for prototyping, etc.

Duration

The project part of the lab is intended to take a total of four three-hour sessions for group meetings so scheduled access to the lab for the group is provided for these periods. Apart from the initial introductory session, there is no requirement for the group to use these allocated session times. The due date for the final demonstration of your software is by 5.00 pm Friday the 6th June 2003. The due date submission of the lab report is by 5.00pm on Monday the 9th June 2003. Late submission of the report will incur a penalty of 10% per day after this deadline.

Conclusion

Believe it or not, this lab is meant to contain a slight element of fun in it (totally unintentional, of course). The reason for the somewhat ponderous specifications detailed above, is so that you know exactly what outcome is required and what will be assessed. This is not intended to limit your creativity in considering alternative solutions, and even for this fairly simple system, there are many alternatives (some of which might work).

APPENDIX

Detailed Java Lift API

This package groups all of the Top-level API classes and is mostly automatically generated by the Rational Rose UML model file for the API (*LiftAPI.mdl*).

Lift

This class provides for the initialisation of the Top Level Lift API

Public Operations:

initLift (DLLpath : String) : int

Purpose:
Method to initialise the lift interface

Inputs:
String: Full directory path to the DLL interface to the DII device driver, if no path is specified then the operation searches the user directory and all set classpaths for the dll

Outputs:
int status
= -1 -> Couldn't find any devices (e.g. no DII cards)
= -2 -> Couldn't find PnPdevice0 (e.g. not the right DII card)
= -3 -> Couldn't setup 8255s (e.g. card problem)
= -99 -> DLL not be found (e.g. path not set correctly or not found)

closeLift() :

Purpose:
Method to close the lift interface (not normally needed since the lift controller runs continuously)

Inputs:
none

Outputs:
None

LiftPanel

This class provides the Lift Panel part of the API, and allows the floor request display LEDs, lift floor display LEDs, internal lift light, lift bell to be controlled and the floor request buttons to be sensed.

Public Operations:

setLiftBell () :

Purpose:
Method to set the lift bell to ring

Inputs:
none

Outputs:
none

clearLiftBell () :

Purpose:
Method to clear the lift bell ring

Inputs:
none

Outputs:
none

setLiftDoor (:) :

Purpose:
Method to set the lift door to open (represented by the lift internal light being switched on)
Inputs:
none
Outputs:
none

clearLiftDoor () :

Purpose:
Method to clear the lift door from open -> close (represented by the lift internal light being switched off)
Inputs:
none
Outputs:
none

setLiftFloorDisplay (floor : int) :

Purpose:
Method to set the lift floor display (7-segment LED)
Inputs:
int floor
= 0 -> set floor 0
= 1 -> set floor 1
= 2 -> set floor 2
= 3 -> set floor 3
Outputs:
none

clearLiftFloorDisplay () :

Purpose:
Method to clear (blank) the lift floor display (7-segment LED)
Inputs:
none
Outputs:
none

setFloorRequestAck (floor : int) :

Purpose:
Method to set the floor request acknowledgment LEDs within the lift panel floor request buttons
Inputs:
int floor
= 0 -> set floor 0 LED
= 1 -> set floor 1 LED
= 2 -> set floor 2 LED
= 3 -> set floor 3 LED
Outputs:
none

clearFloorRequestAck (floor : int) :

Purpose:
Method to clear the floor request acknowledgment LEDs within the lift panel floor request buttons
Inputs:
int floor
= 0 -> clear floor 0 LED
= 1 -> clear floor 1 LED
= 2 -> clear floor 2 LED
= 3 -> clear floor 3 LED
Outputs:
none

getFloorRequest () : int

Purpose:
Method to get the latest floor request from the lift panel
Inputs:
none
Outputs (only returns one event per request)::
int floor
= 0 -> floor 0
= 1 -> floor 1
= 2 -> floor 2
= 3 -> floor 3
= -1 -> no floor request

FloorPanel

This class provides the Floor Panel part of the API, and allows the lift call request display LEDs and lift call answer display LEDs to be controlled, and the lift call request buttons and floor sensors to be sensed.

Public Operations:

getLiftRequest () : FloorDir

Purpose:
Method to get a lift call request from the floor panel
Inputs:
none
Ouputs (only returns one event per request):
FloorDir.floor
= 0 -> floor 0
= 1 -> floor 1
= 2 -> floor 2
= 3 -> floor 3
= -1 -> no lift request
FloorDir.direction
= 0 -> down
= 1 -> up
= -1 -> no lift request

getFloorSensor () : FloorSense

Purpose:
Method to get the floor sensor outputs (primary and intermediate).
Inputs:
none
Outputs (remains set while the lift is at that floor):
FloorSense.floor
= 0 -> floor 0
= 1 -> floor 1
= 2 -> floor 2
= 3 -> floor 3
= -1 -> no floor sensor
FloorSense.inter
= 1 -> between floor 0 & 1
= 2 -> between floor 1 & 2
= 3 -> between floor 2 & 3
= -1 -> no intermediate floor sensor

setLiftAnswer (floorDir : FloorDir) :

Purpose:
Method to set the lift call answer LEDs

Inputs:

FloorDir.floor	FloorDir.direction	
0	1	-> set floor 0 Up answer LED
1	0	-> set floor 1 Down answer LED
1	1	-> set floor 1 Up answer LED
2	0	-> set floor 2 Down answer LED
2	1	-> set floor 2 Up answer LED
3	0	-> set floor 3 Down answer LED

Outputs:
none

clearLiftAnswer (floorDir : FloorDir) :

Purpose:
Method to clear the lift call answer LEDs

Inputs:

FloorDir.floor	FloorDir.direction	
0	1	-> clear floor 0 Up answer LED
1	0	-> clear floor 1 Down answer LED
1	1	-> clear floor 1 Up answer LED
2	0	-> clear floor 2 Down answer LED
2	1	-> clear floor 2 Up answer LED
3	0	-> clear floor 3 Down answer LED

Outputs:
none

setLiftRequestAck (floorDir : FloorDir) :

Purpose:
Method to set the lift call request acknowledgement LEDs

Inputs:

FloorDir.floor	FloorDir.direction	
0	1	-> set floor 0 Up ack LED
1	0	-> set floor 1 Down ack LED
1	1	-> set floor 1 Up ack LED
2	0	-> set floor 2 Down ack LED
2	1	-> set floor 2 Up ack LED
3	0	-> set floor 3 Down ack LED

Outputs:
none

clearLiftRequestAck (floorDir : FloorDir) :

Purpose:
Method to clear the lift call request acknowledgement LEDs

Inputs:

FloorDir.floor	FloorDir.direction	
0	1	-> clear floor 0 Up ack LED
1	0	-> clear floor 1 Down ack LED
1	1	-> clear floor 1 Up ack LED
2	0	-> clear floor 2 Down ack LED
2	1	-> clear floor 2 Up ack LED
3	0	-> clear floor 3 Down ack LED

Outputs:
none

LiftMotor

This class represents the Lift Motor part of the API. and allows the lift motor direction and speed to be controlled.

Public Operations:

setLiftRun (direction : int) :

Purpose:
Method to set the lift to run in a particular direction

Inputs:
int direction
= 0 -> down
= 1 -> up

Outputs:
none

setLiftStop () :

Purpose:
Method to set the lift to stop

Inputs:
none

Outputs:
none

setLiftSpeed (speed : int) :

Purpose:
Method to set the lift speed

Inputs:
int speed
= 0 -> low speed
= 1 -> high speed

Outputs:
none

FloorSense

This class just contains two integer attributes for floor and intermediate floor sensors values

Public Attributes:
floor : int
inter : int

FloorDir

This class just contains two integer attributes for floor and direction

Public Attributes:
floor : int
direction : int

LiftSetup source

```
//Source file: LiftSetup.java
import LiftAPI.*;
/*
This class is run by the lift maintainer - it sets up the lift for normal operation
*/
public class LiftSetup
{
	public static void main(String args[])
	{
		// Setup some useful constants
		final int HIGH = 1; // Lift Motor High speed
		final int LOW = 0; // Lift Motor Low speed
		final int UP = 1; // Lift Up direction
		final int DOWN = 0; // Lift Down direction

		// Define a variable to maintain the current floor the lift is on
		int currentFloor;

		// Create instances of LiftAPI variables
		FloorDir aFloorDir = new FloorDir();
		FloorSense aFloorSense = new FloorSense();

		// Initialize Lift interface
		int status = Lift.initLift("");  // Don't specify a DLL path, let the method find it
		if (status < 0) { // If error return on setup, no point continuing
			System.out.println("Error: initLift failed = "+status);
			return;
		} else {
			System.out.println("Lift API initialised OK");
		}

		// Check if Lift at Floor 0
		aFloorSense = FloorPanel.getFloorSensor();
		if (aFloorSense.floor != 0) {
			//Move lift down to start
			System.out.println("Looking for floor 0, moving down ...");
			LiftMotor.setLiftSpeed(HIGH); // Set lift to high speed
			LiftMotor.setLiftRun(DOWN); // Drive lift down

			// Check for Lift at Floor 0
			do {
				aFloorSense = FloorPanel.getFloorSensor();
			} while (aFloorSense.floor != 0);

			// Lift at Floor 0
			LiftMotor.setLiftStop();
		}
		// Initialize Lift at Floor 0
		System.out.println("Lift at Floor 0, stop ...");
		currentFloor = 0;
		LiftPanel.setLiftFloorDisplay(currentFloor);
	}
}
```