Comparison of Distributed Object Architectures

Goal of the project

The goal of this project is to compare various distributed object middleware . Specifically you will compare Java RMI and CORBA (Sun's Java IDL). These two distributed computing middleware will be compared from a programmer's standpoint as well as an architectural standpoint. Thus you will need to compare the performance obtained in each case (quantitative comparison), as well as make a qualitative comparison. While discussing the quantitative results obtained, back your conclusions with data to support your conclusions. Try to answer why you feel technique A is faster than technique B etc.

In the qualitative comparison, you will address architectural issues. Examples of these include support for multiple inheritance, underlying protocol used, lookup technique used by the client to obtain a server object reference, registry service, whether distributed garbage collection is performed, whether object serialization is supported, platform independence, language independence, exception handling, locating and activating a remote object etc.

In addition, you will compare these middleware, based on your observations, in relation to the ease of understanding the technology; time required for development, difficulties encountered, stability of code etc. This list is not meant to be exhaustive. Be sure to include any pertinent conclusions based on your experiences.

Description of the Client-server interaction

You will make a connection-oriented (TCP), concurrent "database" server. Your server needs to spawn a thread to service each client request. The client must be able to READ and UPDATE data stored at the server. For the READ, since we don't care about real data, you can create a data file on the server machine with data as:

1	John Adams	123 St. Johns Blvd, Jacksonville 32224
2	Mary Adams	232 St. Johns Blvd, Jacksonville 32224

Provide a simple menu to the user, e.g.

1. READ
2. UPDATE
3. LIST
4. QUIT

If the user chooses to READ, then prompt the user for the ID number. If the user chooses to UPDATE, prompt the user to enter a string that will be sent to the server. Your code must be robust and handle any incorrect user input.

During the READ command, the client will transmit the user supplied ID number to the server as an argument. The server will perform a search of file, and if the ID number matches the ID number of a particular string, then the server returns that string to the client. You can implement a search algorithm (e.g. sequential search, or binary search) of your choice. However, you must be consistent in that the same search algorithm must be used for both projects.

For the UPDATE, the client will send a string to the server and the server will update a file. Since there can be multiple clients querying the server at the same time with each client being serviced by a different thread on the server, the server must ensure concurrency control on the data file while it is being updated. Thus only one thread must gain access to the data file at a time.

For the LIST command, the server will send the clients all the records in its data file.

Be sure to terminate each thread cleanly after each client request has been serviced.

Project 1
Implement the project as described using Java RMI.
Due: Thursday, October 7, 2010
Points: 45

Project 2
Implement the project as described using CORBA (Java IDL).
Due: Tuesday, November 16, 2010
Points: 45

In each project, be sure to measure the mean response time for the server to service the client request. Measure the response time separately for READ and LIST. Spawn multiple clients that access the server at the same time and determine the mean response time. Plot two graphs (one each for READ and LIST) that compare the mean response time with the number of clients accessing the server at the same time (determine the mean response time obtained with 1 client, 5 clients, 10 clients, 15 clients, 20 clients, and 30 clients). At the end of the two projects you should have four such graphs. Condense the two graphs for READ into one so that the reviewer can study the performance characteristics of two paradigms. Do likewise for the LIST graphs.

Note: The READ graph will be for light load and the LIST graph will for heavy load. Label the READ graph as READ – LIGHT LOAD and the LIST graph as LIST – HEAVY LOAD. Ensure that the LIST data is approximately 1 MB in size.

Next use the data from the HEAVY LOAD graph to plot the throughput for each of the middleware technologies where throughput is defined to be the number of clients serviced per second. Hence you will have one graph for the throughput, which will depict the throughput achieved by the two technologies. Your graph will have the “throughput” on the y-axis and the “offered load” on the x-axis.

Additional study: Here we will document the throughput achieved by the server in terms of # of bytes delivered/second. To graph this, you need to only have one client make requests in the order of the following data sizes: 500 KB, 1MB, 2 MB, 3MB, 4MB, 5MB. Use the LIST command to do this. Again determine the response time in each case, then plot the throughout versus offered load graph.

Note: In all there should be 4 graphs in your paper: LIGHT LOAD, HEAVY LOAD, THROUGHPUT (# of clients serviced per second), and THROUGHPUT (# of bytes delivered per second).

Your code must be well documented with adequate comments, suitable use of variable names, and follow good programming practices.

You will provide a demo at the end of each project to the instructor.