Problem 1: Distributed Chat Service

Due: 5:00pm, March 3, 2003

I. Overview

The objective of the machine problem is to build a Distributed Chat Rooms Service. The system will have several Chat rooms, and Chat clients can join more than one Chat room. RPC mechanism must be used for the registration of Chat rooms with the Central Registry, and Chat clients joining the Chat rooms. Communication between Chat rooms and Chat clients will be using a socket interface.

II. Registries

Both centralized and distributed systems often rely on specialized location and/or naming services in order to access resources or information in the system. Such services are typically called *registry* services. Think of a registry as a database that maintains information about location, name, status, or other aspects of a resource in the system. The objective of this machine problem is to write a simple registry, which then can be used in future to support distributed applications.

The registry should be implemented in the form of a server (using RPC), which provides a remote invocation interface to clients that allows to register/deregister "entities" (for lack of a better word) and to query the registry about information on the registered entities. The following three remote procedures are to be provided by the registration program.

`int register(char * entity_name, struct entity_information * info)` Register an entity with the given information under the given name.

`int unregister(char * entity_name)` De-register the entity.

`int get_info(char * entity_name, struct entity_information * info_out)` Returns the information about the given entity.

You may want to include a simple management interface to the registry, preferably as described below. This allows you to write a little client program that resets or stops a registry, without having to repeatedly kill the registry server process.

`int reset()` Purge all stored information from the registry.

`int terminate()` Terminate the server program that runs the registry.

`int print_status()` Force registry to display current status, e.g, list of registered entities with their information.

In general it is a very good idea to have error codes handle erroneous use of these functions, like registering an entity twice, or requesting information about a nonexistent entity, etc.

We leave it up to you to give an XDR/IDL description of the call arguments.

III. Detailed Design Issues

As test application you will build a simple realization of a distributed chat room service. This service allows for *chat room providers* register one or more chat room(s) under some name each with the registry. *Chat room clients* then can join chat rooms by first looking up the registry to see what chatrooms are there. If they are interested in a particular chat room, they can request information about it from the registry, and then join the chat room.

Note: The given registry interface does not allow getting the identities of registered entities. So, it is difficult to inquire about what chat rooms have been registered. The challenge to you is to modify the registry interface so as to allow the retrieval of the names of all registered entities. Find a way to do this without having to ship back and forth huge messages, even in the case of millions of entities registered. (Think in terms of iterators).

A. Chat Client Program

The communication between the Chat client and Chat room service could be using UDP message passing. You will have to decide upon the message structure.

It may appear confusing that you will have to use the same console both for incoming messages from the chat room and for outgoing messages and commands from the user. This is not as hard as you may think, as you can use the `select()` call to do I/O multiplexing on both the `stdin` file descriptor and the UDP socket descriptor. To keep network and user "inputs" separate. The user command (mostly JOIN and LEAVE) can be easily identified by appropriate parsing. (If you feel like adding a fancy user interface to this with separate areas for input and output, you may want to look into the curses library.)

To keep it simple, allow clients to join multiple chat rooms using a single console. The messages from each room may be displayed with the room name as prefix, and the user should be able to choose any room and join/leave/post to that room. DO NOT USE MULTICAST.

Summarizing the above, a chat client would do the following steps, in a loop, in any order:

1) Find out the currently available chat rooms, through one or more RPC calls to the Registry Server. (Remember what we said earlier: You must keep the message sizes manageable even for millions of chat rooms around.)
2) Join a particular chat room the user wishes to. Also the client must be able to leave a group. This communication will be using UDP datagrams.
3) Send messages to chat rooms(s). All chat rooms listen on a well-known port, which the client finds out on his enquiry with the Registry.
4) Receive messages from the Chat Room and display them, while the user controlling the client program should still be able to do all the above.

B. Chat Room Provider Program

The Chat room provider can register one or more Chat Rooms with the Registry. As discussed above, a good naming convention must be thought of to identify individual Chat rooms registered with the Registry. After registering a Chat room, the server would fork/create a new process, which would then be the server for the newly created Chat Room. The functions of this server will be explained in the next section. It is up to the provider or to the chat room server to de-register any chat room that is terminated (including servers that terminate because they crash!). This program uses RPC for registering or de-registering a chat room.

C. Chat Room Server

The process implementing the Chat room server would do the following steps in a loop, in any order:

1) Listens on a well-known port to accept JOIN/LEAVE requests or chat messages from Chat clients.
2) For a JOIN request, advertise its well-known port to the chat client, and make an entry for the chat client with his (client's) port information in its local database.
3) For a LEAVE request, delete the corresponding entry from the local database.

4) For a chat message, forward the message to all the chat clients listed in its local database. DO NOT USE MULTICAST.

IV. What to Hand In

The running system will consist of the server program with the registry, the chat room server, and the chat room client. As platform, you should be using the UNIX workstations, and develop the program in C/C++. (You will see later why.)

A. Design

Before you start hacking away, plot down a design document. The result should be a system level design document, which you hand in along with the source code. Do not get carried away with it, but make sure it convinces the reader that you know how to attack the problem. List and describe the components of the system: Chat Client Program, Chat Room Program/Provider Program, Registration Program.

B. Hard Copy of Portions of Source code

Hand in a hard copy of all the code you created. Do not excessively waste trees! Try to condense the printouts, e.g. with `enscript -2r <file>` The code should be easy to read (read: well-commented!). The grader reserves the right to deduct points for code that he/she considers undecipherable.

C. Measurements

In order to compare the efficiency of your implementation, you will perform the following measurements:

I Measure the latency of a registration/de-registration cycle. This can be done by having a client repeatedly register and immediately unregister a chat room, and then measure the elapsed time. Determine whether the latency is affected by the number of chat rooms in the system.

II Similarly, measure the latency of inquiry about registered chat rooms. Again, do this with increasing number of registered chat rooms. (Note: you need to test for somewhere around 100 chat rooms.)