ARIZONA STATE UNIVERSITY

CSE 434, SLN 71101 — **Computer Networks** — Fall 2013

Instructor: Dr. Violet R. Syrotiuk

Team Socket Programming Project

Available Sunday 11/03/2013; due Sunday 11/24/2013

This version of the project contains some minor corrections and clarifications to the original project description. These are given in red to make them easy to identify.

1 The Client-Server Model

A *server* is a process that is waiting to be contacted by a *client* process so that the server can do something for the client. A typical scenario is as follows:

- The server process is started on some computer system. It initializes itself, then goes to sleep waiting for a client process to contact it requesting some service.
- A client process is started, either on the same system or on another system that is connected to the server's system with a network. Client processes are often initiated by an interactive user entering a command to a time-sharing system. The client process sends a request, possibly travelling across the network, to the server requesting service of some form.
- When the server process has finished providing its service to the client, the server goes back to sleep, waiting for the next client request to arrive.

Notice the roles of the client and server processes are asymmetric. As a result, each role is coded differently. The server is started first and typically does the following steps:

1. Open a communication channel and inform the local host of its willingness to accept client requests on some well-known address (a port number).
2. Wait for a client request to arrive at the well-known address.
3. For an iterative server, process the request and send the reply. Iterative servers are typically used when a client request can be handled with a single response from the server.
4. Go back to step 2: wait for another client request.

The client process performs a different set of actions:

1. Open a communication channel and connect to a specific well-known address on a specific host (i.e., the server providing the desired service).
2. Send service request messages to the server, and receive the responses. Continue doing this as long as necessary.
3. Close the communication channel and terminate.

2 Types of Servers

A *stateless server* does not maintain any information about client requests (e.g., the Network File System (NFS)). The client requests service by sending a message to the server. The server executes the request and sends back a response. It's possible that the request or the response might get lost. Both of these cases look the same to the client — it doesn't get an answer. If the client does not receive an answer within a timeout period, it re-issues the request. If the requests to the server are *idempotent*, meaning that the request can be executed any number of times with the same effect, then re-executing the request doesn't cause a problem.

However, not all services can be made idempotent. Some servers must remember the state of the client and are therefore called *stateful servers*. For example, in a lock server, executing a request twice can have very bad results. Suppose a client requests a lock on file F. The server grants the lock, but the response is lost. The client times out and re-issues the request. The server responds with a message saying that the file is currently locked. Now, the file is permanently blocked because the client will never send the unlock command.

An *exactly-once* service ensures that a client's request is executed at most once and is executed at least once if the client does not fail. Such a server must be stateful in order to determine whether or not a particular request has been serviced.

In this project you will implement an exactly-once (stateful) server and a client in C or C++ as described in §2.1 and §2.2 communicating through sockets using UDP. Your client and server programs must run on a Linux OS. Your client program must be able to work with a server program written by another team. Similarly, your server program must be able to work with a client program written by another team. This means that the request format, described next, must be the same for all teams.

2.1 An Exactly-Once Server

To implement exactly-once service, the server must remember the states of the clients. In particular, the server must remember the last request that the client made.

At every processor where a client for the service can reside, the processor assigns to each client a unique number. Each client numbers its requests to a server sequentially, starting at zero. Finally, each processor has an *incarnation number*, which is incremented whenever a processor crashes (and therefore it must be recorded on stable storage, i.e., a text file named `inc.txt` containing a single integer storing the incarnation number). With every request for service, a client attaches the triple of integers (I, C, R) where I is the incarnation number, C the client number, and R the request number.

When a server receives a request from processor P, numbered with the triple (I, C, R), it searches the *client table* to see if it has ever received a request from processor P, incarnation I, client C before. If not, the client is new, so the server adds an entry to the client table, tagged with the triple (P, I, C). Otherwise, the server takes one of the following three actions.

Let the last request sent by (P, I, C) be request number r:

1. If $R < r$, ignore the request. (This request must be "old" since the server has processed requests with a higher number. It must be that one request R of the client obtained a response. Perhaps this request was delayed in the network.)

2. If $R = r$, send the stored response to the client.

3. If $R > r$, perform the service, record the response in the client table, update the response number in the client table and send the response to the client.

In a "real" implementation, where the server would run continuously, the server needs to be able to garbage-collect the client table. The server should log updates to the client table so that it does not re-execute a request after a crash. However, handling this issue is complex. Proper implementation of logging and garbage collection is not necessary for this project.

2.2 The Project

This project involves writing two programs, a client and a server. The client and server should communicate through sockets using UDP (the User Datagram Protocol).

- The port number is to be read as a command line argument for the server.
- The IP address of the server (in dotted decimal), port number, and the client number (in that order) are to be read as command line arguments by the client. (Of course, the client and the server should be run using the same port.)

Be sure to log what is happening both at the client and at the server.

The Server. The server is an exactly-once server that performs the following service:

1. Maintains a five letter string, initially all blank.
2. Accepts a request from a client. The client passes in a character. The first four letters from the string are appended to the letter from the client, which becomes the new string. The new string is returned to the client.

For example, suppose that the server's string is "xyzwv". The client calls the server and passes in 'a'. The new string is "axyzw", and this is returned to the client. The idea behind this service is to be simple and non-idempotent.

The server should simulate failures of two types. In one type of failure, the server should drop the request without performing the request. In another type of failure, the server should perform the request and then drop the request, i.e., perform the service but not return a response. The server should choose at random to fail (in one of the two types) or to successfully process the request. The failure probability for each type of failure is 10%, and therefore the probability of successfully processing the request is 80%.

The Client. Before a client can make a request, it must establish a connection to the server. The client then issues twenty (20) requests to the server. On each request it randomly selects a character in the alphabet to pass to the server. Since the server may fail, the client should keep resending the request after a short time-out until it gets a response.

Failure of the client's machine is simulated by incrementing the incarnation number stored on stable storage. Since this file is used by many processes it must be locked and unlocked every time it is accessed. After processing a random number of the 20 requests that the client sends, the client machine should fail with probability 0.5.

To ensure consistency of request format among groups, use following the structure for the client request. See the sample code writing the struct to the socket and reading it from the socket.

```
struct request{
   char client_ip[16]; /* To hold client IP address in dotted decimal */
   int inc; /* Incarnation number of client */
   int client; /* Client number */
   int req; /* Request number */
   char c; /* Random character client sends to server */
};
```

3 Submission Instructions

Submit electronically, before 1:30pm on Thursday, 11/24/2013 using the submission link on Blackboard for Project #1, a zip[1] file named `Team-TeamNumber.zip` containing the following items:

[1]**Do not** use any other archiving program except `zip`.

Design and Analysis (30%): Provide a description of the methodology you followed to write your client/server. Specifically, discuss your selection of data structures for storing and manipulating the table at the server, etc. Be sure to include instructions on how to compile your program for a Linux OS.

Implementation (50%): Provide your documented C/C++ source code for both your client and server program. **You may write your code only in C or C++. You must not alter the requirements of this project in any way.**

Correctness (20%) Provide output showing a run of your client and server both running on the same machine, and running on different machines. Our TA will test your programs on a Linux OS. (We may set up demos, if that's what the class decides.)