Assignment 1: Client-Server File Store

Distributed Systems Class

In order to build a "peer-to-peer" Facebook, one of the most important building blocks is to provide a way to invoke routines on a remote machine. To this end, students in groups of three will build an RPC protocol and set of procedures that implement a very simple file system, e.g., to store friend groups and message board postings.

1 The Storage System

For this assignment, you are to set up a two node system, one the client and the other the server. The client can send requests to the server to store, get, or modify files. For simplicity, the file system can be flat (i.e. no directories). You also do not have to handle large files (i.e. files are smaller than the maximum length of a packet minus header lengths), although you should still check that this assumption is true for debugging purposes. The onCommand function takes user input or commands from a file, and your implementation of the onCommand function should be able to parse and execute the following commands:

create *server filename* This command creates an empty file called *filename* on the node with virtual address *server*. If the file already exists, this should fail in the manner described below.

get *server filename* Get a file from a server. This returns the file in its entirety. The client should then print this file to the console. If the file does not exist, this should fail in the manner described below.

put *server filename contents* Write the contents to a file on a server. This should only work on an already created file, and should replace the existing contents of that file.

append *server filename contents* Append contents to a file on a server. This should only work on an already created file, and should be appended to the end of the existing file.

delete *server filename* This command deletes the file called *filename* on the node with virtual address *server*. If the file does not exist, this should fail in the manner described below.

If the command fails for any reason except timeout, no files should change and an error message should print out. In the case of a timeout or any kind of failure, the system should provide at-most-once semantics. Error messages should be of the form:
"Node n: Error: *command* on server *server* and file *filename* returned error code x"
where x is:

10 File does not exist
11 File already exists
20 Timeout

30 File too large

You should keep in mind that these commands will be used by future assignments, so you should design them to be modular, and so that it is easy to call a create(server, filename) internally in addition to calling it from a command file/user input.

2 Reliable, In-order Message Layer

We have provided an example Node subclass and related functions that implement a reliable, in-order message layer on top of the framework. When a node sends a message with this layer, the layer uses timeouts to resend packets until they are acknowledged by the receiver. The receiver then uses a sequence number to hold on to out-of-order packets before delivery. This is somewhat similar to TCP's support for reliability and in-order delivery, except that a theoretically infinite number of packets can be in-flight without acknowledgment.

It is important to note, however, that this layer is not complete. Specifically, it is not correct in the presence of node failures (you should think about why this is true). Since reliable, in-order delivery is useful for this project, before implementing the file storage protocol, you should correct this problem.

3 Facebook-Style RPC Commands

Using the RPC-based file system interface, implement the following operations:

- Create a user
- Login/logout as user
- Request a friend
- Accept a friend
- Post a message to all friends
- Read all messages posted

You should feel free to specify how the information is stored on the server and you can define the user interface to be anything you would like (text command line is fine).

4 Deliverables

4.1 Code

Students will need to provide the source code that implements the above requirements. The source code should be well-documented in order to ease grading and prevent misunderstandings about functionality. Compile/execution scripts must be included as well, and you can assume that they will run on support managed machines in the lab from the directory just above the jars/ and proj/ directories. The groups will meet with the TA to demonstrate the code and explain how it works.

4.2 Write-up

In addition to the source code, a major component of your grade will be based on a project write-up, in which your group should explain the protocol and any relevant implementation details. You should include, at a minimum, the following:

- a detailed description of your group's implementation
- any assumptions you made
- how to use your implementation
- any outstanding issues
- anything else that you feel is important to discuss (e.g. quirks, interesting behavior, performance characteristics, etc)

5 Framework Issues

For this first project, it is safe to concentrate on the simulator mode (-s option) and ignore the Emulator. In subsequent assignments, we will add the ability to run on multiple machines in the lab in order to add realism and help students write code that works in all cases.