CSE 5306: Distributed Systems
Project 1 Report

Hamza Reza Pavel (1001741797)
Manish Munikar (1001826846)

October 10, 2020

I have neither given nor received unauthorized assistance on this work.

Signed: Hamza Reza Pavel, Manish Munikar Date: October 10, 2020

Assignment 1: Single-threaded File Server

In this assignment, we implemented a simple single-threaded file server and client using socket-based message-oriented communication. The server supports 4 basic file operations: upload, download, rename and delete.

The communication between the client and the server is done using the BSD socket API. We used the TCP as the transport protocol because it provides reliability and guaranteed ordered delivery of packets, which are both vital for file transfer service. On top of TCP, we developed our own simple file transferring protocol that supports the aforementioned 4 basic file operations.

The request message consists of a set of tokens separated by a special delimiter. The first token is always the *command*. The second token is always the *filename* to perform the operation on. For the UPLOAD command, the third token is the content of the file in bytes. Similarly, for the RENAME command, the third token is the new filename. Except for the "download" operation, the server doesn't need to send any response back. The acknowledgment is handled by the TCP layer. Once the server receives the "download" request for a valid file, the server sends back the binary contents of the file back to the client.

Being a single-threaded server, it rejects additional connection requests while it is connected to a client. Therefore, it cannot support concurrent requests from the client. A benefit of this is that the server doesn't have to worry about locking and synchronization between concurrent clients.

Assignment 2: Multi-threaded File Server

The goal of assignment 2 is to implement a multi-threaded file server where the client will be the same as assignment 1, but the server has to be modified to support concurrent upload, download, rename, and delete operation.

To handle requests from multiple clients simultaneously, we used a threading based approach. We created a class named WorkerThread derived from Python's Thread class. In our server's main, we use a loop. The socket object we created to establish the connection with clients listens for incoming requests in the main loop. Whenever we get a new request from a

client, a new WorkerThread object created with the connection info is spawned from the main loop. The WorkerThread object performs the intended operation of uploading, downloading, deleting, or renaming the file.

The main challenge in implementing assignment 2 was the server part. Specifically, handling multiple client requests using thread and testing the functionality. Testing was done by increasing the buffer size, opening multiple terminals, and running multiple clients in the command windows simultaneously.

When the same file is ready by multiple clients, it does not create any issues. But issues arise when modification on a file is done by one of the files. By modification, we mean deleting and renaming the files. If one client is modifying the file name, while the other is downloading it, this may end in some unpredictable result. Another case is when one client is downloading a file from the server and another client is trying to delete it. To avoid such cases that may lead to ambiguous results, distributed locking is necessary so that no client can read or modify a file when another client is modifying it.

Assignment 3: Synchronous RPC

In this assignment, we implemented a simple synchronous RPC (remote procedure call) client-server pair. An RPC is a concept of executing the computational part of a function in a different (remote) server without the user having to explicitly implement the communication details. In an RPC system, the user (client) is provided with a function that, when called, looks like a normal function call in the user's local process, but the RPC client stub will marshal all the necessary information (such as function id, arguments, and other metadata) in a serialized data format and send it via network protocols to the RPC server. When the server receives the RPC request, the server stub unmarshals the parameters, identifies the function, and calls the actual function with the given arguments. If the function has a return value, that is marshaled and sent to the client in a similar way. All this communication is transparent to the user, so the user feels like the function was executed in his own process. In synchronous RPC, the client stub client blocks until the server sends the response.

In our implementation, the client and the server communicate with the help of TCP protocol. We implement the RPC for 4 simple functions: CalculatePi, Add, Sort, and MatrixMultiply.

Some important issues in RPC are how to serialize the parameters and how to handle reference (pointer) values like arrays. We used JSON as the serialization format, which is a popular language-agnostic data exchange format. To handle reference arguments, we made the client stub dereference the arguments (i.e., we copy the whole data structure, not just the pointer value) and pack them into the request message to the server. After getting the response from the server, the client stub adjusts the pointers in the local address space accordingly. One

Assignment 4: Asynchronous and Deferred-Synchronous RPC

The goal of assignment 4 is to improve on assignment 3 and implement asynchronous and deferred synchronous remote procedure calls.

For this implementation, we used JSON to send and receive data between server and client. Our implemented client can invoke both asynchronous and deferred synchronous RPC

using two classes implemented in the client.py file named AsyncRPC and DeferredRPC. Both classes have two common methods named invoke() and get_result(). When creating the RPC client object, we pass the function name to invoke and the parameters to that function. The functions are implemented in the server. The invoke method in the AsyncRPC and DeferredRPC creates the JSON message containing function name, parameter, and RPC type and sends it to the server. For the AsyncRPC, the server responds with an acknowledgment and a unique *token* identifying that particular RPC call. The token is later used to look up the result when get_result() method is called.

For the deferred-synchronous RPC, the client's invoke() function splits the program into two threads: one waits for server's response while the other thread does some task specified by the user. in this way, the client can have useful work done even while waiting for server's response.

The server listens for the invocation of RPC. When it receives an RPC invocation request, it parses the JSON to obtain the function name, function parameters, and the RPC type. For AsyncRPC, the server sends an acknowledgment to the client with a token. A token is a unique identifier which the client will later use to look up the result. The server then proceeds to perform the computation and stores the result in a table with the token it provided the client. For deferred synchronous RPC, the server performs the computation and sends the result to the client.