Remote Procedure Call

CSE 5306

Distributed Systems

Submitted by :

LeelaMadhav Somu

Part-1 :

I used a VM warehouse and installed Ubuntu for this Assignment. Since this is the first time installing a virtual machine, it took some time. This taught me how to install any virtual machine going forward.

Part-2 :

We implemented a multi-threaded file server and client message-oriented communication. The server supports 4 basic file operations: upload, download, rename and delete. And used different folders to hold the files downloaded to the client or that can be uploaded to the server.

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

The main challenge in implementing this was the server part. Since multiple clients try to access the same server, testing is done by increasing the buffer size, by opening multiple terminals. When we try to download the same file from different clients it did not cause any issues and issues arise when the same file is being modified

Part-3 :

The file transfer between the client and the server can be made transparent to users and automatically handled by a helper thread. We created a Dropbox-like synchronized storage service. In general, Dropbox means once a file is added to a Dropbox, it's synced to our secure online servers. That means if we did any changes in one server that might be the server for suppose if we upload anything the server that needs to be added in the client is the same as it is if we change anything in the client that must be changed in the server. Same as creating the new file, updating the file, or deleting the helper thread will establish a connection with the server and automatically send the corresponding operation to the server to update the folder on the server side.

To handle requests from multiple clients simultaneously, we used a threading-based approach.

The socket object we created to establish the connection with clients listens for incoming requests Whenever we get a new request from a client, a new Worker Thread

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

The main challenge in implementing this 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.

The file should be synchronized if we did anything on the client side. To simplify the design, we do not need to consider incremental updates. Thus, if the content of a file is updated, the entire file should be sent to the server to overwrite the original copy at the server.

Thus, if the content of a file is updated, the entire file should be sent to the server to overwrite the original copy at the server.

On the server side, the functions DeleteFileOperation, RenameFileOperation, and PutFileOperations are declared so that the functions are operated accordingly. Whenever the file gets modified, it is printed on the console stating the file is modified and all the other operations are shown similarly. On the client side,are used to track the changes in the files **Part-4 :**

In this, we tried to implement asynchronous and deferred synchronous remote procedure calls. 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. We implemented a simple synchronous RPC (remote procedure call) client-server pair. We tried to send and receive data between the server and the client. Our implemented client can invoke both asynchronous and deferred synchronous RPC using two classes implemented in the file. 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. When creating the RPC client object, we pass the function name to invoke the parameters to that function. The functions are implemented in the server

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

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 the get_result() method is called.

The server listens for the invocation of RPC. When it receives an RPC invocation request, it parses the 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 that 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.