DESIGN DOCUMENT

CS550 - Advanced OS

Fall 2020

Amit Nikam

A20470263

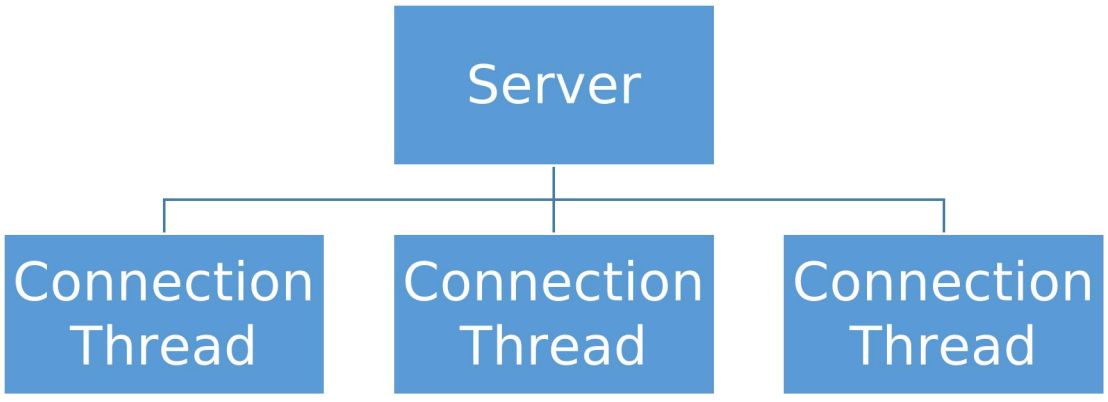
anikam@hawk.iit.edu

Implementation

Overview: For my implemented I have programmed client - server where the server hosts a collection of files which the client can view, select from and download. The server and client are both multithreaded and can upload and download files in parallel respectively. Server and client make use of sockets along with a set of predefined protocol to establish connection and communicate. Sockets are software structures within a network node of a computer network that serve as an endpoint for sending and receiving data across the network. This program is implemented using python 3.8.3 and it's default modules.

Server: Being a socket server, upon initialization the server binds itself to an endpoint (by default self host and port 9000). An endpoint is a port on the host machine and can be set manually by a user by passing it as an argument to the server while initializing. The server can be configured to set the directory from which the files need to be served, to set port number and to set host. The server is programmed in a way that it implements everything even without passed arguments and start listening for connections.

Clients can connect to the server using the endpoint address, which is server's IP address and port number. Client needs to be aware of the server's endpoint to connect to it. Every time a client connects to the server, the server creates a new thread that specifically handles that particular connection. Once the connection is closed, the corresponding thread also terminates. No limit has been set for the creation of threads to experiment with the performance.



The server actively keeps listening for any new connections and messages. The Server in particular doesn't initiate any communication but waits for the clients to communicate. Upon receiving a message from the client, the server takes required actions. It follows a request-reply form of communication. If a client request for a list of files, the server creates the list of current files on demand and returns the list as a reply. If the client request a download, the server creates a md5 hash for the file and sends the hash followed by the file to the client.

Client: The client implementation happens to be what is known as a fat-client. While initializing the client, the user can pass arguments to declare where to download the files to, set server IP address or set server port for connection. If the download directory is not created, the client creates one.

Upon initializing, the client establishes a connection to the server i.e. a client is connected with a connection thread at the server end. After successfully establishing the connection, the client sends the server a message to fetch the list of files in the host directory. The server sends the list of the files generated on demand as a reply. The client then requests the user to provide user input for selecting the files they want to download. All the checks for validity are conducted on client end and only the existing files from the selected list are selected for downloading.

After getting the list of files to download, the client again requests the user for input to selected the mode of download i.e. if they want to download files serially one after another or parallely at the same time.

For a serial download, the client sends the download message to the server with the name of the file and keeps listening for a response. The server upon receiving a download message, opens the file binary and creates an md5 hash and sends it to the client as a message followed by the file binary. The client then saves the received md5 hash and the file, performs an integrity check by comparing the md5 hash received from the server to the md5 hash performed on the file received. If the integrity check is successful, the client writes the file to the download folder with the same name as the file has on the server. This process continues until all the files have been downloaded.

For a parallel download, the client establishes a new threaded connection with the server for each file it wants to download. In this way each connection only focuses on downloading one file through their connection. The receiver which is otherwise blocking while receiving is now being used parallely with other file receivers. Since the connection happens to be a thread on server side too, one to one connection threads are mapped between the client and the server. As soon as the client thread receives the md5 hash and file binary, it sends the server a disconnect message and closes the connection. In case it fails the integrity check, the thread informs the main client thread so that it can handle the fails. For one client downloading N number of files from the server, N+1 threads are active at the server, out of which N connections download the file and disconnect, and so finally only one thread is left, which was the original client thread.

The main client thread handles the issues like timeouts and integrity check failures irrespective of the method of downloading the file. The client again attempts to download the files which failed to download using the same method as user specified before (serial or parallel). Client attempts thrice to download these files, after which if the files still fail to download, the client informs the user and closes the connection and program.

The user can start the client again and check if the files they were trying to download have been deleted or not.

Protocol: The server and client follow certain communication protocol. For every message, a header of 64 Bytes is padded at the beginning of the message. This header contains the length of the offload / body. The header is first decoded and validated before accepting the rest of the message. Messages have a packet size of 2048 Bytes, so any messages larger then that are sent across in multiple packets. A default timeout of 5 seconds has been set for messages so that any packet loss shall terminate the message. All the messages are encoded and decoded using python's pickle module, which happens to be a serializer. Serializing the messages enabled to send python lists and objects through the message body.

Possible Improvements

One of the major area of improvement in my opinion happens to be in the way client downloads. A hybrid of parallel and serial mode can improve the download speeds significantly. By creating a limited number of parallel connection as threads and serially downloading files through these parallel connections, better performance can be achieved. For example, creating 16 parallel connections for 16 file downloads will slow down the client and degrade performance. But creating 4 parallel connection threads and serially download 4 files in each of those connections can significantly improve the performance.

Additionally, a smart client, which can optimize download speeds by fetching file size from server along with the file list and properly assigning a threaded connection and sequence in the hybrid mode of downloading can add to better performance.

Looking into the server side, limiting the maximum thread limit can improve the server response time and stop the server from being bombed with connections. Server is not really useful if too many connection are getting too little time to use the actual resource.

Lastly, timing out the inactive connections on server can be really useful to terminate any blocking connections. This way the resources are freed and new connections can be established.