COMP9331 Assignment

Peer-to-Peer File Sharing System: “BitTrickle”

2024 T3

Changyue Tan

z5325184

BitTrickle is a simple Peer-To-Peer file-sharing system written in Python that allows users to:

- Share files with other active users
- Download files shared by other users.

The system consists of two programs, “client.py” and “server.py”.

The flow of execution “client.py” is as follows:

- Create a UDP socket for communication with the server, the server’s port number is given as a command line argument, and the client’s port number is allocated by OS arbitrarily.
- Prompt user to enter credentials, and send it to server for authentication.
    - Application lay protocol: “*AUTH {username} {password}*“.
        - If it receives “*ERR*”, prompt the user to enter credentials again.
        - If it receives “*OK*”, prompt the user to enter commands.
- Start a thread that opens a welcoming socket that listens for peer connection requests.
    - The welcoming thread with run with other threads in parallel
    - Upon receiving peer connection requests
        - Check “stop_welcome” flag. If it is True, close the welcoming socket
    - Start a thread that opens a TCP connection socket
        - The connection socket will be used to upload files requested by a peer
            - Remote host: *“{filename}”*
            - Localhost: *“{data of the requested file}”*
        - Add the upload thread to a list of upload threads
        - The upload thread will run with other threads in parallel
        - The connection socket will be closed after the upload finishes
- Start a thread that starts sending heartbeats to the server, every 2 seconds.
    - Client: *“HBT {username} {welcoming port number}”*
- User can enter the following 1 of 7 commands:
    - get
        - Client: “*GET {username} {filename}”*
        - Server: “*OK {username of an active peer with the requested file} {port number of that peer’s welcoming socket}”*
        - Create a thread that opens a TCP connection with the peer
            - Add this thread to a list of download threads

        - Localhost: *"{filename}",* to tell the peer which file it wants
        - Remote host: *"{data of the requested file}"*
        - The download thread will run parallel with other threads
        - The connection socket will be closed after the download finishes
    - lap
        - Client: "*LAP {username}"*
        - Server: "*OK {# of active users} {active username 1} {active username 2} …"*
    - lpf
        - Client: "*LPF {username}"*
        - Server: "*OK {# of published files} {published filename 1} {published filename 2} …"*
    - pub
        - Client: "*PUB {username} {filename}"*
        - Server: "*OK"* or "*ERR*" depending on publication success or failure
    - sch
        - Client: "*SCH {username} {substring}"*
        - Server: "*OK {# of files found} {filename 1 containing substring} {filename 2 containing substring}…"*
    - unp
        - Client: "*UNP {username} {filename}"*
        - Server: "*OK"* or "*ERR*" depending on unpublication success or failure
    - xit
        - Set a "stop_welcome" flag to be True
        - Send a connection request to the client welcoming socket
        - Since the "stop_welcome" is true, instead of creating a connection socket for this request, the welcoming socket will be closed.

When closing the client, it will wait for all threads to terminate. This includes the welcoming thread, the heartbeat thread, and every thread in the downloading and uploading thread list to finish. Then the client program gracefully exits.

The flow of execution of "server.py" is as follows:

- Opens a UPD socket to receive messages from clients, the port number of the listening socket is given as a command line argument
- Create containers to store information about clients:
    - credentials: a dictionary of credentials loaded from "*credentials.txt*"
        - <username: password>
    - active_clients: a set of unique usernames

        - {a set of users that are active}
    - heartbears_record: last heartbeat time for each user, initialised to be 0
        - <username: last heartbeat time>
    - published_files: a dictionary from files to a set of users with that file published
        - <username: {a set of clients with this file published}>
    - file_publishing_users: a dictionary from users to a set of files published by that user
        - <username: {a set of files published by this user}>
    - contact_book: the last known address of every user
        - <username(could be offline): (last known) welcoming port number>
- Every time before a new message is received:
    - Check if any user in the "active_clients" set is no longer active by
        - comparing its last know time of heartbeat in "heartbears_record" to the current time
        - If not, remove it from "active_clients".
    - Receive the message, identify its type, and call respective handling functions
        - Client: *"{request type} {username} {other possible message specific to each request type}*"
    - Request type:
        - AUTH
            - If the credentials are incorrect or the user is already active, send "*OK*"
            - Otherwise send "*ERR*"
        - HBT
            - Update the heartbeat record
            - Update the contact book according to the heartbeat's source port
        - LAP
            - "*OK {#of active clients} {active user 1} {active user 2} …"*
        - LPF
            - "*OK {#of published files} {published file 1} {published file 2} …"*
        - PUB
            - "*OK*". Update the published files dictionary
        - UNP
            - "*OK*" or "*ERR*" depending on if the intended file to be unpublished was already in published files
        - SCH
            - "*OK {# of files found} {filename 1 with substring 1} {filename 2 with substring} …"*
        - GET
            - "*OK {user name} {address}*" if any active client has the request file
            - "*ERR*" if not

The Server will run indefinitely until killed by the shell terminal.