COMP3331/9331 Assignment - Discussion Forum

by Zheng LONG

System Architecture

I broke down the program into 2 sides, server and client, and each side has those components:

Server has authentication, thread management, message handling, file transfers, and UDP reliability functions:

- Authentication: checks username/password pairs against credentials.txt file and keeps track of who is logged in.
- Thread management: handles creating threads, listening and deleting the threads. And only the person who created the threads is authenticated to delete them.
- Message handling: is for posting, editing and deleting messages. And only the person who owned the messages is authenticated to edit and delete them.
- File transfers: had to use TCP connections for this part since files can be too big for UDP to handle.
- UDP reliability: had to implement sequence numbers, ACKs and retransmission timers.

Client side has command interface, network stuff and display functions.

- Command interface: parses what users type and formats it for the server.
- Network stuff: Deals with both protocols (UDP/TCP).
- Display: shows responses in a readable way.

I also wrote a autotest.py script to check whether those parts in server and client runs well:

- Sequential Test verifid basic functionality under sequential client operations. Test Coverage: User authentication (AUTH) Thread creation/listing/reading (CRT/LST/RDT) Message editing/deletion (EDT/DLT) File upload/download (UPD/DWN) Permission validation (e.g., User A cannot delete User B’s messages)
- Concurrent Test validated system stability under multi-client concurrency. Test Coverage: Let 3 clients operating simultaneously Concurrent file uploads/downloads Data consistency checks.

Data Structurs

Dictionaries are best for this assignment.

```
# For users
users = {"yoda": "wise@!man", "vader": "sithlord**"}
active_users = {"yoda": ("192.168.1.5", 45678)}

# For forum content
threads = {
  "jedi_training": {
    "creator": "yoda",
    "messages": [
      {"id": 1, "author": "yoda", "content": "Do or do not"},
      {"id": 2, "author": "luke", "content": "I'll try"}
    ],
    "files": ["lightsaber_manual.txt"]
  }
}
```

Thread safety had to use locks everywhere to make sure things didn't break when multiple users were on the forum at the same time.

Communication Protocol

Commands are formatted like: `COMMAND [THREAD] [ARGUMENTS]`

Message Type	Format	Description
Create Thread	`CRT threadtitle`	Create a new discussion thread
Post Message	`MSG threadtitle message`	Post a new message to a thread
List Threads	`LST`	Request a list of all active threads
Read Thread	`RDT threadtitle`	Request thread contents
Edit Message	`EDT threadtitle messagenumber message`	Edit an existing message
Delete Message	`DLT threadtitle messagenumber`	Delete a message
Remove Thread	`RMV threadtitle`	Remove an entire thread
Upload File	`UPD threadtitle filename`	Upload file to thread
Download File	`DWN threadtitle filename`	Download file from thread
Exit	`XIT`	Log off from the forum

The UDP reliability part added sequence #s to packets (4 bytes at the start), made the reciever send back ACKs, retransmitted after 500ms if no ACK, and doubled the timeout on repeated failures (exponential backoff)

I originally tried a sliding window approach but it was getting too complex, so I simplified to stop-and-wait which worked fine for this assignment.

For file transfers, I did:

1. Client asks to transfer via UDP.
2. Server says "connect to port XXXX".
3. Client gets TCP connection and sends JSON with file info.
4. Data gets sent in chunks.
5. Connection closes when done.

Design Decisions

Had to make some choices:

UDP vs. TCP for commands: I went with UDP with my own reliability layer for commands. Its a bit faster for small messages, and I wanted to try implementing the reliability stuff myself. For files. TCP makes more sense, but ot wasn't about to implement large file transfer over UDP.

Concurrency: I used threads with locks. Of course it is not the most scalable but it is easier than async programming (I tried using asyncio at first but got confused with all the callbacks and switched to threading).

Storage: Everything is in memory. If server restarts, all data is gone, which is not ideal but the assignment didn't require persistance.

Limitations and Improvements

My implementation has some issues:

Problems:

- Passwords are stored as plaintext.
- The program cannot resume interupted file transfers.
- Memories are lost if server crashes
- Probably won't scale larger amount of users

What I could fix with more time:

- At minimum hash the passwords
- Use SQLite or something for persistance
- Add resume capability for file transfers

Testing

I tested with the autotest.py with a few different ways:

Sequential testing: ran through all the commands in sequence to make sure they worked. Created threads, posted stuff, edited and deleted messages, etc..

Concurrent testing: got a few clients going at once to test thread safety. Found a nasty race condition where two users editing messages at the same time could cause message IDs to get messed up.