P0P Design-Test Document

Partner A: Bryce Richardson

~~Partner B: Tyler Miller~~

1. Server Design

Overview

Describe the overall server design. You must include the number of threads you create including the main thread, what each thread is for and doing, any loops you have for each thread (and do this for both event-loop and thread-based server).

- The thread-based server is composed of two threads: the main thread and the socket thread.
  - The main thread initializes the socket reads input from the keyboard. It contains a single while loop that listens for input from stdin and only breaks if it reads an EOF or a ‘q\n’ from TTY.
  - The socket thread listens for packets and responds to them accordingly. It is a daemon thread, so it will be scrapped if the main thread ends. It is composed of a single while loop that 1) accepts data from the socket, 2) ensures that data is a valid packet, and 3) responds accordingly.
- There was no event-loop server.

Justification

Justify why your design is efficient and show evidence it can handle load efficiently. Specify the number of clients, the file size for each client, and the response rate/delay each client.

- We tested our clients using Dostoyevsky.txt (3158667 bytes)
  - For a UTCS/UTCS environment (same computer), the loss rate was 19.6% for our implementation and 58.7% for the bare-bones server. When we tested it a day later, the loss rate was 83.9% for our implementation and near-zero for the bare-bones server.
  - When tested with Expect, that number became 83.0%, compared to 7.4% from the bare-bones server, so the latter result is likely more accurate.
  - However, when the number of clients was increased to two, the loss rate jumped to 98.0% compared to 96.8% from the bare-bones server. When we tested it a day later, the loss rate was 97.5% for our implementation and 78.6% for the bare-bones server.
- In general, our server seems to perform worse than the bare-bones server. However, it is also extremely fast, receiving Dostoyevsky.txt in less than 10 seconds on average. With the amount of processing our server does to ensure

correctness compared to the bare-bones server, the discrepancy in packet loss is more than accounted for.

Data structures

List any notable data structures you used and justify the use of the data structure. Also specify how you handle synchronization for the data structure if there were any need for synchronization.

- The server uses a dictionary ("sessions") to map the session id to its sequence number, IP:port address, and Timer. In addition, an instance of the aforementioned Timer class is used to close out of a session if it takes too long to respond.
- Synchronization is handled by ensuring that the dictionary is only accessed in one thread at a time. In the socket thread this is while the socket is open and the sessions can be changed, and in the main thread this is when the socket is closed and the sessions cannot be changed.

How timeouts are handled

Describe how the timeouts are handled.

- Timeouts in the server are handled with the Timer class of the threading module. Each session maintains its own Timer instance. When DATA is received from a client, that Timer is canceled. When ALIVE is sent back to the client, the Timer is reconstructed.
- If 5 seconds elapse while the Timer is active, that session will be immediately closed the client will be sent a GOODBYE command.

How shutdown is handled

Describe how you handled the shutdown gracefully. (That is when you hit 'q' or ctrl+d in the command prompt.)

- A graceful shutdown iterates through every session that is being tracked by the server and closes them, and then closes the socket.
- This is done in the main thread, so the socket thread will terminate gracefully when the main thread ends and the socket is closed.

Any libraries used

List any libraries that you used for the implementation and justify the usage.

- We only used the Python standard library, but we referenced several specific modules within it:
    - **enum:** we used enum (specifically IntEnum) to reference the different commands in a more readable format
    - **socket**: we used the socket library because the server communicates with the clients via socket

    - **sys**: we used the sys library solely to read input and to test whether it was from a tty or not
    - **threading**: we used the threading library because we utilized threading.Timer for our time-out system

Corner cases identified

Describe corner cases that you identified and how you handled them.

- There were numerous corner cases that we designed our code to recognize using EX 2. For example, it was possible for the sequence number to overflow back to zero. To solve this, we allowed our code to respond normally to Data and Goodbye commands with a sequence number of zero. However, it may have been better to recognize when a sequence number was about to overflow and preemptively reset the expected sequence number to zero.
- In many cases where the correct response was ambiguous, we chose to ignore the packet. For example, if the server receives something that isn't HELLO from a client it does not recognize, it treats it as gibberish and silently discards it.

2. Server Testing

Testing your server

It felt fairly difficult to test the server. When the server was working, it was much easier to test the clients. However, we still did our best to test it thoroughly:

- We would use Dostoyevsky.txt to make sure it compiled and ran first. Then we would test it against the bare-bones client.
- We would test concurrency for various numbers of clients, and we would also test multiple environments.
- Then we would test that time-out worked. It also turned out we needed to test 'q' and Ctrl+D as well, since we found out ours didn't work properly when the client was in its READY phase!
- After this, we would think of corner cases and ensure that the code covered them. This was mostly done manually in the code itself. Many notes from this phase of development still exist for readability's sake.
- Finally, we tested it against our very own "crazy client" which responded with unconventional commands (ALIVE instead of HELLO, or HELLO instead of GOODBYE).

3. Client Design

Overview

Describe the overall client design. Specify any differences between event-loop based client design and thread-based client design. You must include the number of threads you create including the main thread, what each thread is for and doing, any loops you have for each thread (and do this for both event-loop and thread-based client).

- For the thread-based client, we used four separate threads. These were the main thread, the "handshake" thread, the keyboard thread, and the socket thread. In addition, there are multiple Timer threads.
    - The main thread initializes the client. It contains an Event signal that waits for one of the other threads to indicate that it is time to close the client.
    - The handshake thread handles the initial data exchange between the client and the server. It also creates the other two threads.
    - The keyboard thread reads input from stdin to send to the server. It contains a single while loop that listens for input and only breaks if it reads an EOF or a 'q\n' from TTY.
    - The socket thread listens for packets and responds to them accordingly. It is composed of a single while loop that 1) accepts data from the socket, 2) ensures that data is a valid packet, and 3) responds accordingly.
- For the event-loop client, we used one thread with multiple events. These events include:
    - The client event, which receives packets and then calls a function that approximates the "socket thread" in the thread-based server.
    - The clientTTY event, which listens for input from stdin and then calls a function that approximates the "keyboard thread" in the thread-based server.
    - The Timer event, which closes the session if the server takes too long to respond.
    - The goodbyeAsync event, which immediately closes the session and must be manually initiated by another event.

Justification

Justify why your design above is efficient. Also show evidence how your client can handle sending packets from a large file (each line is close to UDP packet max and also contains many lines) and receiving packets from the server at the same time. Note you do not want to cause the false TIMEOUT from server because you are too busy just sending out the packets to the server when the server actually has sent you a packet before the TIMEOUT.

- We tested our clients using Dostoyevsky.txt (3158667 bytes)
- Thread-based client:
    - For a UTCS/UTCS environment (same computer), the loss rate was 19.6% compared to a loss of 89.1% from the bare-bones client. When we tested it a day later, the loss rate was 83.9% for our implementation and 95.6% for the bare-bones client.
    - When tested with Expect, that number jumped to 83.0%, compared to 83.5% from the bare-bones client.

    - However, when the number of clients was increased to two, the loss rate jumped to 98.0% compared to 98.5% from the bare-bones client. When we tested it a day later, the loss rate was 97.9% for our implementation and 98.8% for the bare-bones client.
    - On a UTCS/UTCS environment (different computers), the loss rate was extremely low, ranging from 6% to 0%. With two clients, this remained in the same range. With four, it increased to about 50%.
- Event-loop client:
    - For a UTCS/UTCS environment (same computer) with Expect, the loss rate was 19.6% compared to a loss of 89.11% from the bare-bones client.
- In conclusion, the data suggests that our clients are noticeably more efficient than the bare-bones client at one client, and on par with additional clients. This justifies our design decisions.

Data structures

List any notable data structures you used and justify the use of the data structure. Also specify how you handle synchronization for the data structure if there were any need for synchronization.

- For the thread-based client, we used no data structures other than Timers (which is justified for the same reason as in the server).
    - However, we did use an IntEnum called FSA to keep track of what state the client was in. This made it much easier to manage edge cases as well as the timer.
    - Using FSA, we could ensure that each thread could only edit a Timer when it is in a certain state, which was helpful for synchronization.
- For the event-loop client, we used a loop object, which was useful because it automates much synchronization on its own. However, we synchronized the Timer in the same way we did in the thread-based client.

How timeouts are handled

Describe how the timeouts are handled.

- For the thread-based client,
    - Timeouts in the server are handled with the Timer class of the threading module. In general the client cancels the current Timer when it receives ALIVE, and creates a new Timer when it sends DATA. However, there are some exceptions (namely during the HELLO_WAIT phase).
    - In all cases, if 5 seconds elapse while the Timer is active, the client will enter the CLOSING phase. Here, it sends GOODBYE to the server and waits up to 5 seconds for a response.
- For the event-loop client, we used the Timer class of the pyuv module.
    - Much like the thread-based client, we cancel the Timer when ALIVE is received and restart it when DATA is received. The client closes if 5 seconds elapse while the Timer is active.

    - This was much easier to implement, because it was designed to work with the loop object.

How shutdown is handled

Describe how you handled the shutdown gracefully. (That is when you hit 'q' or ctrl+d in the command prompt.)

- For the thread-based client, the code calls a function that sends GOODBYE to the server, and then waits up to five seconds for a response. At that point, it closes the socket and ends the main function, scrapping the various daemon threads.
- For the event-loop client, the code calls a function that sends GOODBYE to the server, and then waits up to five seconds for a response. At that point, it either calls a function or starts an event that ends the loop and closes the client.

Any libraries used

List any libraries that you used and justify the usage.

- We mostly used the Python standard library, referencing several specific modules within it:
    - **enum:** we used enum to reference the different commands in a more readable format
    - **random**: we used random to generate the random session id
    - **socket**: we used the socket library because the client communicates with the server via socket
    - **sys**: we used the sys library solely to read input and to test whether it was from a tty or not
    - **threading:** we used the threading library because we used the Timer class for time-out and the Event class for broadcasting between threads (the thread-based client only)
- In addition, we used **pyuv** for the event-loop client. This was necessary to use the event-loop functions.

Corner cases identified

Describe corner cases that you identified and how you handled them.

- One major corner case we recognized during development was during the CLOSING phase. In some cases (i.e. the client times out and then receives “DATA” from server) the client would send GOODBYE twice. This was solved by setting the state to CLOSING before sending GOODBYE and adding an “if state is not CLOSING” conditional before certain blocks of code.
- The majority of the other corner cases we identified during the project are from EX 2 and are specified in the code itself (for readability).

4. Client Testing

Testing your client

The main way we tested our client was with manual inputs.

- We would use Dostoyevsky.txt to make sure it compiled and ran first. Then we would test it against the bare-bones server.
- Then we would test that time-out worked at three different stages: before the handshake, after being "ready", and while closing out.
- After this, we would think of corner cases and ensure that the code covered them. This was mostly done manually in the code itself. If there had been more time, I wonder if it would have been possible to create a program that randomly generated test input.
- Finally, we tested it against our very own "crazy server" which responded with unconventional commands (DATA in response to HELLO, or ALIVE in response to GOODBYE). This helped us find quite a few bugs with our code!
- We also used naive.exp and our own test.exp (which input the example from the specification page) to test our code using Expect.
- We followed these steps for both the thread-based and the event-loop clients. However, since we could not run Dostoyevsky.txt on the event-loop client, we used Dostoyevsky.exp instead.

5. Reflection

Reflection on the process

What was most challenging in implementing the server? What was most challenging testing the server?

- The server was actually fairly easy to implement. But it felt like the most difficult to test since all of the test cases depended on how we implemented our clients.

What was most challenging in implementing the client? What was most challenging testing the client?

- The timer was by far the most difficult part of the thread-based client to implement. I struggled for a long time, and eventually compromised on the design because of time constraints (I did not use time.sleep like I had intended). Fortunately, the design still works. Testing it was not too challenging, although I was not as thorough as I wanted to be.
- The hardest part of implementing the event-loop client was understanding how to utilize pyuv. Testing it was even more difficult because I had to use Expect. But the design itself was the easiest part of the project to implement.

What was most fun working on the project? What was most not-so-fun?

- I did not have fun working on this project in general, but I feel like it was education so I'm grateful for that. I did get an exciting rush of adrenaline when I saw the event-loop client work for the first time, however.

If you are to do this all over again how would you do it differently?

- I would get a different partner. I think it would be a lot more enjoyable if I worked with someone who was capable of splitting the workload evenly with me.
- I would also, perhaps, implement the event-loop client first. I'm curious to see how that would affect my design for the thread-based client.

Reflection on pair programming

Log of the amount of time spent driving and the amount of time spent working individually for each part (e.g., X drives 1 hour; Y drives 45 minutes; X works alone for 1 hour, etc.)

- **Part 1:**
  - Bryce drives 30 minutes; Tyler drives 30 minutes; Bryce drives 30 minutes; Tyler drives 30 minutes
  - Bryce drives 30 minutes; Tyler drives 30 minutes; Bryce drives 30 minutes; Tyler drives 30 minutes
  - Bryce works alone 30 minutes
  - Tyler works alone 1 hour
  - Bryce drives 30 minutes; Tyler drives 30 minutes
- **Part 2:**
  - Tyler works alone 1 hour
  - Bryce drives 15 minutes, Tyler drives 15 minutes
  - Tyler works alone 1 hour
  - Bryce works alone 2 hours
  - Bryce drives 20 minutes, Tyler drives 15 minutes, Bryce drives 45 minutes, Tyler drives 20 minutes
  - Tyler works alone 30 minutes
  - Bryce works alone 4t minutes
- **Part 3:**
  - Bryce drives 20 minutes; Tyler drives 20 minutes
  - Bryce works alone 2 hours
  - Tyler works alone 30 minutes
- **Part 4:**
  - Bryce works alone 18 hours

What went well/or not-so-well doing pair programming? What was your take away in this process?

- My partner was always busy and behind in this class, so he could only meet for a few hours every weekend. I was okay with this since, even though we were behind, we were still making steady progress. However, he stopped being responsive during Part 4. When I tried to reach out to him, he texted me that he had dropped the class without letting me know and that I was on my own.

- I don’t have a takeaway other than the fact that I do not like group projects after all.

Submission

Remember to export to pdf and push it to your github team repo under the project root (the same level as README.txt)