P0P Design-Test Document

Partner A: Zain Bashir

Partner B: JB Ladera

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 server is modularized in a class so that testing is easier. The actual threads are created by a runner which uses an instance of the server.
- The state of the server is not explicitly stored, but when the state changes only certain actions are available.
- Four threads
    - The main thread handles keyboard input in a loop.
    - handle_timer checks all sessions every five seconds in a while loop and removes those that have been inactive for more than five seconds.
    - handle_socket receives packets and modifies the state of the client according to the received packet. It also prints out the packet that was sent. After finishing the current packet, this thread loops back to the blocking receive packet call. There is a lost packet loop to print out any missed packets, so this would prevent any further packets from being processed until all of these messages are written to the print buffer.
    - handle_print prints the queue of messages every two seconds.
- To close the server, each of these threads are stopped using a signal, and all clients are sent GOODBYE messages. The socket is closed because it releases the blocked thread. This design allows us to avoid daemon threads which would cause us to not release all resources before we close.

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.

- Our design is efficient because for every packet received, the handle_socket thread will take the same amount of time. Extra work such as timers and printing is moved into separate threads where they will do any time intensive work.
- Our barebones server and finished server have identical response times for processing packets and sending them back to the client. This was tested by averaging the round trip time of a client sending a DATA packet to receiving

an ALIVE packet. The response time was roughly 0.00005 seconds on a personal machine.

- When each packet loss test was performed on the same UTCS lab machine:
    - Barebones Server - One client = 16% packet loss
        - This barebones server consistently crashed during this test
    - Finished Server - One client = 19% packet loss
    - Barebones Server - Two clients = 50% packet loss
        - The server rejected one client so it had 100% packet loss and the other client came through slowly with 0% packet loss.
    - Finished Server - Two clients = 74% packet loss
- When each packet loss test was performed between two different UTCS lab machines:
    - Finished Server - One client = 58% packet loss
    - Finished Server - Two clients = 67% packet loss
    - Barebones servers were not run since they crash too often
- On a personal machine, up to 3 clients can run with less than 1% packet loss.

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.

- We have a queue to print from. This queue takes the messages in the order that they would have been sent to print, and handle_print prints them out. There is a lock on this queue so that it does not lose any messages when they are added to the queue from a different thread.
- We store all client sessions in a dictionary. This dictionary maps a session id to a ClientData class representing the state of the client. This data structure is accessed from the handle_socket and handle_timer threads, so it needs to be synchronized.

How timeouts are handled

Describe how the timeouts are handled.

- Each client session has a field that keeps track of the last time a message was received from the respective client. The handle_timer thread periodically checks every session and will close any sessions that have timestamps older than 5 seconds.

How shutdown is handled

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

1. First, the keyboard thread is the only thread that can close the server. Therefore, it tells the server to close and waits for all of the other threads to exit.
2. Second, the closing thread acquires a lock in order to prevent simultaneous execution with handle_packet. Next, another lock is acquired in order to set the _closed variable to True. All threads view this variable to decide whether they should continue looping. The next time any other thread sees this variable, it will choose to close instead of continuing to process its data. When the closing thread gets the lock, this ensures that no other threads are processing data since they require this lock to do so.
3. Then, we get the lock for the client session map which prevents the timer thread from continuing. Then, we remove all of the clients and close them as we are removing them. This involves sending a GOODBYE message to all of the clients.
4. Then, we shutdown and close the socket. This breaks the handle_socket thread if it was blocked in a receive_packet call. This will force the handle_socket thread exit.
5. Now that all of the extra threads are closed, we flush the print buffer to make sure that all of the server's output is printed to the console.

Any libraries used

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

- No libraries were used to create the server's implementation
- Pytest is used to test the server.

Corner cases identified

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

- We check the version and magic numbers for all incoming packets and discard the packet if they are incorrect
- Our print queue has a lock so that we do not lose messages through asynchrony.
- Sometimes, print is too fast for the terminal to handle and it crashes. We fix this through printing with a queue.
- Hello -> Data vs Hello -> Hello. All packets have the same sequence number. Normally, this would print out "Duplicate Packet!" However, the first will close the session because it is an invalid state transition. The second will be ignored since it is a duplicate packet. We handle this by storing the command of the previous packet to compare to the newly received packet.
- Session closed -> GOODBYE for a client should not be printed if the server receives a GOODBYE after the timer closes the session. This is handled by making sure that the client map is locked and that the client exists in the map when the server is closed.

2. Server Testing

Testing your server

Describe how you tested your server. Include the description of each test case and the setting (such as local only, one local one remote, how many clients, use of mock client, etc.).

We used pytest to run tests on each function in our server class. This way, each part of the server can be tested independently and in any order without any restrictions due to threads. Otherwise, we test the server and client with Dostoyevsky.txt to make sure that the basic functionality works. Some aspects that can be tested through a proper server and client are that the server's timer does not close the session before Dostoyevsky.txt is finished sending. Similarly, we can ensure that ALIVE is being sent since the client does not disconnect from the server before Dostoyevsky.txt is finished.

The first part of these test cases tests that our packet classification works properly. The second part is that the server performs the correct response to this packet's type.

test_response_small_packet checks that an incorrectly sized packet is ignored.

test_response_mismatched_magic checks a packet with the incorrect magic number is ignored

test_response_mismatched_version checks that a packet with the incorrect version number is ignored

test_response_unexpected_packet_seq checks that a new client's HELLO packet with sequence number 1 is ignored.

test_response_unexpected_packet_command tests that a first packet that is not HELLO is ignored

test_response_first_hello tests that a correct HELLO packet for a new client is accepted

test_response_sid_different_hostname tests that packets from the same client id but different port numbers are ignored

test_response_same_sid_different_address tests that packets from the same client id but different addresses are ignored

test_response_duplicate_packet_different_command tests that two packets with the same sequence number but have different commands should cause the server to close the session.

test_response_duplicate_packet_same_command tests that the extra packet is ignored and "Duplicate Packet!" is printed.

test_response_out_of_order_delivery tests that receiving a packet with an earlier sequence number causes the server to close

test_response_lost_packets tests that the right number of “Lost Packet!” messages are printed when the sequence number skips. It also makes sure that the server continues as normal and processes the received packet.

test_response_receive_another_hello tests that the session is closed when two HELLO packets are received from the same client

test_response_receive_alive tests that the server closes the session when it receives an ALIVE.

test_response_receive_goodbye tests that the server views GOODBYE as a valid message and closes the session.

test_response_receive_data tests that the server views DATA as a valid message and will process the message.

test_response_receive_unknown_command tests that the server closes the session when it receives an unknown message.

test_handle_first_hello tests that the server knows a HELLO message is a HELLO message, “Session Created” is printed, and the session is added to the server’s client map.

test_handle_duplicate_packet_different_command tests that a packet with a duplicate sequence number and different commands closes the session, removes the client from the map, and prints “Session Closed.”

test_handle_out_of_order_delivery tests the same as test_handle_duplicate_packet_different_command but with a earlier sequence number

test_handle_receive_another_hello tests that the server closes the session properly as before when receiving two HELLO messages from the same client.

test_handle_receive_alive tests that the server closes the session properly when it receives an ALIVE.

test_handle_receive_goodbye tests that the server closes the session properly when it receives a GOODBYE. It also ensures that “GOODBYE from client” is printed.

test_handle_receive_data tests that sending a DATA packet prints out the message received.

test_handle_receive_unkown_command tests that the server closes the session properly when it receives an unknown command.

test_timeout_no_remove tests that the timer does not remove any clients that have sent messages within the last 5 seconds

test_timeout_remove tests that the timer removes any clients that have not sent any messages in the last 5 seconds

test_timeout_then_goodbye tests that the server does not accept any GOODBYE packets from the client after the server has closed the session

test_server_close_clients_disconnect tests that all clients are removed when the server closes

test_server_duplicate_seq_different_data tests that two data packets with different data and the same sequence number get classified as duplicate packets

test_close_race attempts to test that there is no race condition when the server closes with handling packets. This was tested with up to 300 packets being received at once, but this number was lowered for the submission since the test takes a while on the UTCS lab machines. This test must be run several times to know the true outcome since race conditions are not deterministic.

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).

Our thread-based client has three threads: one that handles the socket on a loop, one that handles the keyboard on a loop, and one that handles the timer on a loop. The main thread blocks until one of these threads exits, so it is essentially nonexistent. Like our thread-based client, our event-based client has three separate events: the timer, keyboard, and socket. There are no loops in any of the events defined in the event-based client.

Most of the code between the event and thread-based clients is shared in the Client class that each version extends. The thread-based client overrides some of these functions to obtain locks to keep the threads synchronized.

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 the 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.

Our design is efficient since there are no loops for processing any of the data or packets. The only loops are to retrieve the next item to process. Our client is able to fully send Dostoyevsky.txt within a few seconds with no lost packets from the file on a perfect network. We confirmed this by sending the full Dostoyevsky.txt file to a server on the same machine in under a second with no packet loss.

Our client processes packets in a similar manner to the server. Since the server does not have trouble processing Dostoyevsky.txt, the client would not have trouble either. Either way, the client has to process receiving every ALIVE packet from the server for all DATA packets it sends through Dostoyevsky.txt and does so successfully. The client sends and receives packets on different threads, so it is able to accommodate this load and prevent false timeouts.

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.

Our thread-based client uses a blocking queue that lets the main thread know that either the socket or keyboard want the program to exit. This lets the main thread block until the threads are ready to close. Once the main thread gets this signal, it closes the client which causes the rest of the threads to exit. This data structure does not need to be synchronized since the main thread is the only thread reading from it while the other threads write, and the main thread only needs to see that there is data available rather than the contents of the data.

Our event-based client does not use any notable data structures.

How timeouts are handled

Describe how the timeouts are handled.

The thread-based client times-out when the message timestamp exceeds 5 seconds. This timestamp is set when HELLO is first sent, for the first DATA message before an ALIVE is received, or after GOODBYE is sent. The timestamp is removed and the timer does nothing when the first HELLO is received, ALIVE is received, or a GOODBYE is received. The event-based client starts and stops the pyuv timer at the same occurrences.

How shutdown is handled

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

Either the handle_socket or timeout threads will be closing the client. Handle_socket closes the client for a GOODBYE message, and timeout closes the client if the client has sent a GOODBYE without a response. For the thread-based client, the closing thread first signals to the main thread that it wants the client to exit. The main thread unblocks on this signal to close the client. It first marks the

closing state, then resets the timer, and finally closes the socket. The other thread's loops view that the server is in the closing state and end. These threads only exit when they are in a safe state because they choose when they end.

The event-based client just has to stop the socket and timer events. Once these events have stopped, it calls the provided callback function that stops the loop

Any libraries used

List any libraries that you used and justify the usage.

No libraries were used to implement the client.

Pytest was used to test the client.

Corner cases identified

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

Leaving any extra threads running significantly slows down python since python is single threaded where each thread runs on one core. This is why the main thread blocks instead of spinning in a while loop until a thread is no longer alive.

If we receive an ALIVE from the server without sending any DATA, then the server must have messed up so we close the client.

We also had to take into account that the client is not allowed to accept any keyboard input to send packets before the HELLO exchange is received. However, we had to make sure that the client is allowed to quit before it has received a HELLO from the server.

4. Client Testing

Testing your client

Describe how you tested your client. Include the description of each test case and the setting (such as local only, one local one remote, how many clients, use of mock server, etc.).

Our client tester functions in a similar way to the server tester. First, there are tests on each of the underlying client functions. Otherwise, the client is tested with Dostoyesvsky.txt to make sure that it can connect to the server.

The event and thread-based clients share many similar tests, however the event-based client does not have tests relating to race conditions and timeouts.

test_bad_address tests that a client getting a packet from the wrong address rejects the packet without crashing or closing

test_bad_packet_header tests that the client rejects a packet with a wrong header without crashing or closing

test_small packet tests that the client rejects a wrong-sized packet without crashing or closing

test_good tests that the client accepts a HELLO packet when it is in the hello wait state.

test_bad tests that the client does not accept an ALIVE packet when it is in the hello wait state.

test_hello_wrong_session_id tests that the client only accepts a HELLO packet with its session id. This ensures that the client closes after this wrong packet.

test_one_timeout tests that the client times out when it does not receive a HELLO from the server after sending a HELLO

test_one_timeout_good tests that the client does not timeout a second time when it receives a GOODBYE packet after timing out the first time. This test ensures that the client still closes.

test_one_timeout_bad tests that the client closes and does not accept a HELLO packet after timing out

test_one_timeout_ignore tests that the client times out but ignores any future ALIVE packets

test_one_timeout_reset_timer tests that when the client times out and enters the closing state, it does not reset the GOODBYE timer when it receives an ALIVE

test_two_timeout tests that the client times out twice when it does not receive a HELLO and a GOODBYE

test_stdin tests that the client does not send any user input until HELLO has been received from the server.

test_stdin_q tests that the client accepts quitting from stdin before the client has received a HELLO by sending a GOODBYE.

test_pre_ignore tests that ALIVE packets do not change the state of the client

test_pre_goodbye tests that the GOODBYE packet closes the client

test_pre_bad tests that the client closes when it receives a DATA packet

test_send_data tests that the client sends data without crashing and increases the sequence number accordingly

test_send_data_receive_alive tests that the client operates properly when it receives an ALIVE after sending DATA

test_send_data_receive_goodbye tests that the client closes if it receives a GOODBYE after sending a DATA

test_send_data_receive_random tests that the client closes if it receives a packet with a command that it does not recognize

test_send_data_timeout tests that the client times out and sends a GOODBYE if it sends DATA and does not get a response

test_data_bad_session_id tests that the client closes if it receives an ALIVE with a different session id than its own

test_send_data_twice_timestamp tests that the client does not reset the timestamp to timeout when it sends more DATA before receiving an ALIVE. The timestamp is only set for the first DATA sent.

test_receive_alive tests that ALIVE packets received in the closing state are ignored

test_receive_goodbye tests that the client closes when it receives a GOODBYE packet in the closing state

test_receive_random tests that the client closes when it receives a packet with an unknown command in the closing state

test_timeout tests that the client times out if it does not receive a GOODBYE in the closing state

test_receive_goodbye_bad_session_id tests that receiving a GOODBYE packet in the closing state with a wrong session ID causes the client to close

test_timeout tests that the timer does nothing in the closed state

test_send_data tests that the client does nothing when it sends DATA in the closed state

test_receive_goodbye tests that the client does nothing when it receives a GOODBYE packet in the closed state

test_receive_alive tests that the client does nothing when it receives an ALIVE packet in the closed state

5. Reflection

Reflection on the process

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

The most challenging part in implementing the server was managing race conditions due to multiple variables being used by multiple threads. Testing for race conditions was also challenging as not every single run may have experienced an error due to a race condition. As a result, we had to do multiple runs.

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

The most challenging part in implementing the client was designing how the client should be structured because we needed to support both event and thread based clients where one dealt with locks and one did not. The most challenging part for testing the client testing for race conditions and avoiding deadlock with the thread-based client as it utilized locks.

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

The most fun part working on the project was trying to come up with a good design because we went through several ideas before finalizing on one. The most not-so-fun part was testing for race conditions. Utilizing multithreading can be quite complicated as not recognizing race conditions can take a lot of time to debug. It is also difficult to test our implementations on multiple machines since finding other devices on the network can be cumbersome.

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

We would definitely try to think of the test cases prior to coding because we sometimes missed some scenarios in our design and had to rewrite some logic in our classes. We also could have planned out the shared code better. Most of our code is contained in different files than they originally began in, and most of the code was able to be consolidated towards the end, especially between the thread and event-based clients.

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.)

- For Part 1
    - 2/12
        - Zain drives 45 minutes
        - JB drives 45 minutes
        - Zain drives 40 minutes
- For Part 2
    - 2/16
        - JB drives 45 minutes
        - Zain drives 45 minutes
        - JB drives 45 minutes
        - Zain drives 45 minutes
    - 2/17
        - JB drives 45 minutes
        - Zain drives 45 minutes

        - JB drives 40 minutes
    - 2/19
        - Zain drives 45 minutes
        - JB drives 45 minutes
- For Part 3
    - 2/23
        - Zain drives 45 minutes
        - JB drives 45 minutes
        - Zain drives 45 minutes
        - JB drives 45 minutes
        - Zain drives 45 minutes
    - 2/26
        - JB drives 45 minutes
        - Zain drives 45 minutes
        - JB drives 45 minutes
        - Zain drives 45 minutes
        - JB drives 45 minutes
        - Zain drives 45 minutes
- For Part 4
    - 3/5
        - JB drives 45 minutes
        - Zain drives 45 minutes

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

During pair programming, we were able to recognize each other's mistakes quite well while one was driving and one was observing. As a result, we were able to lessen the amount of time spent debugging and use that to think more about our design. We both have different coding styles and strengths and were able to learn from each other and adapt our preferences. Overall, the takeaway from this experience is that communication is very important to successfully create a working and efficient design and helps in coding faster.

Submission

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