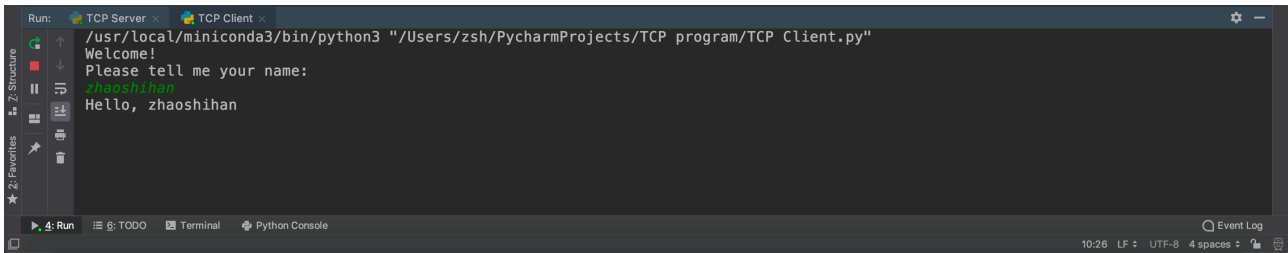


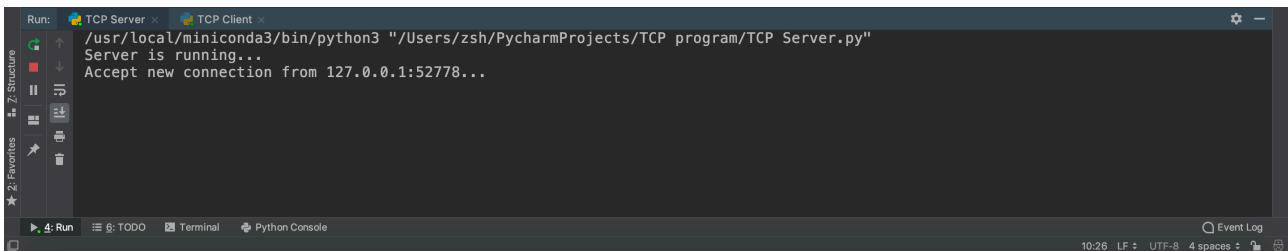
Homework 3

StudentID: 2016302580149 Name: 赵世晗

1.用 python 完成一个 tcp 应用小程序，详见同目录下TCP program文件



```
Run: TCP Server x TCP Client x
/usr/local/miniconda3/bin/python3 "/Users/zsh/PycharmProjects/TCP program/TCP Client.py"
Welcome!
Please tell me your name:
zhaoshihan
Hello, zhaoshihan
```



```
Run: TCP Server x TCP Client x
/usr/local/miniconda3/bin/python3 "/Users/zsh/PycharmProjects/TCP program/TCP Server.py"
Server is running...
Accept new connection from 127.0.0.1:52778...
```

2.第三章后作业选做5题（含一道拥塞控制的问题）

P7. In protocol rdt3.0, the ACK packets flowing from the receiver to the sender do not have sequence numbers (although they do have an ACK field that contains the sequence number of the packet they are acknowledging).

Why is it that our ACK packets do not require sequence numbers?

Answer:

To best answer this question, consider why we needed sequence numbers in the first place. We saw that the sender needs sequence numbers so that the receiver can tell if a data packet is a duplicate of an already received data packet. In the case of ACKs, the sender does not need this info (i.e., a sequence number on an ACK) to tell detect a duplicate ACK. A duplicate ACK is obvious to the rdt3.0 receiver, since when it has received the original ACK it transitioned to the next state. The duplicate ACK is not the ACK that the sender needs and hence is ignored by the rdt3.0 sender.

P8. Draw the FSM for the receiver side of protocol rdt3.0

Answer:

The sender side of protocol rdt3.0 differs from the sender side of protocol 2.2 in that timeouts have been added. We have seen that the introduction of timeouts adds the possibility of duplicate packets into the sender-to-receiver data stream. However, the receiver in protocol rdt2.2 can already handle duplicate packets. (Receiver-side duplicates in rdt 2.2 would arise if the receiver sent an ACK that was lost, and the sender then retransmitted the old data). Hence the receiver in protocol rdt2.2 will also work as the receiver in protocol rdt 3.0.

P11. Consider the rdt2.2 receiver in Figure 3.14, and the creation of a new packet in the self-transition (i.e., the transition from the state back to itself) in the Wait-for-0-from-below and the Wait-for-1-from-below states:

`sndpkt=make_pkt(ACK,1,checksum)` and `sndpkt=make_pkt(ACK,0,checksum)`.

Would the protocol work correctly if this action were removed from the self-transition in the Wait-for-1-from-below state? Justify your answer. What if this event were removed from the self-transition in the Wait-for-0-from-below state? [Hint: In this latter case, consider what would happen if the first sender-to-receiver packet were corrupted.]

Answer:

If the sending of this message were removed, the sending and receiving sides would deadlock, waiting for an event that would never occur. Here's a scenario:

- Sender sends pkt0, enter the “Wait for ACK0 state”, and waits for a packet back from the receiver
- Receiver is in the “Wait for 0 from below” state, and receives a corrupted packet from the sender. Suppose it does not send anything back, and simply re-enters the ‘wait for 0 from below’ state.

Now, the sender is awaiting an ACK of some sort from the receiver, and the receiver is waiting for a data packet from the sender – a deadlock!

P12. The sender side of rdt3.0 simply ignores (that is, takes no action on) all received packets that are either in error or have the wrong value in the acknum field of an acknowledgment packet. Suppose that in such circumstances, rdt3.0 were simply to retransmit the current data packet. Would the protocol still work? (Hint: Consider what would happen if there were only bit errors; there are no packet losses but premature timeouts can occur. Consider how many times the n th packet is sent, in the limit as n approaches infinity.)

Answer:

The protocol would still work, since a retransmission would be what would happen if the packet received with errors has actually been lost (and from the receiver standpoint, it never knows which of these events, if either, will occur).

To get at the more subtle issue behind this question, one has to allow for premature timeouts to occur. In this case, if each extra copy of the packet is ACKed and each received extra ACK causes another extra copy of the current packet to be sent, the number of times packet n is sent will increase without bound as n approaches infinity.

P14. Consider a stop-and-wait data-transfer protocol that provides error checking and retransmissions but uses only negative acknowledgments. Assume that negative acknowledgments are never corrupted. Would such a protocol work over a channel with bit errors? What about over a lossy channel with bit errors?

Answer:

In a NAK only protocol, the loss of packet x is only detected by the receiver when packet $x+1$ is received. That is, the receiver receives $x-1$ and then $x+1$, only when $x+1$ is received does the receiver realize that x was missed. If there is a long delay between the transmission of x and the transmission of $x+1$, then it will be a long time until x can be recovered, under a NAK only protocol.

On the other hand, if data is being sent often, then recovery under a NAK-only scheme could happen quickly. Moreover, if errors are infrequent, then NAKs are only occasionally sent (when needed), and ACK are never sent – a significant reduction in feedback in the NAK-only case over the ACK-only case.