Qiurun Wu

Structure And Design

First, I designed a structure called *buffer_t*, which contains the data to be sent/ acknowledged/ submitted. Also, some helper verities like *Length*, *usedLength*, *lastSentTime* are also in this structure.

Each connection has three *buffer_t* linked list to manage the data:

- unsentList: Contains the data from the input, but hasn't been transferred to the peer.
- sentUnackList: Contains the data which has been sent but not been acknowledged. Maybe needed to resend in the future.
- unsubmitedList: Contains the data gotten from the sender, but hasn't been submitted to the output.

Here we should notice: Considering without selective ack option, **we will not need to visit the *buffer_t* randomly** (except on-timer check). We only pop out the buffers from the front and insert new buffers at the tail, where we greatly reduce the time complexity.

We also should know that **not every movement will lead to a buffer moved to another list**. For example, a buffer with 500Byte data can only send 200Byte at a time. So, we also need to record *usedLength* of the buffer, and *memcpy()* the sent data to *sentUnackList*, but buffers in *sentUnackList* and *unsubmitedList* are handled as a whole structure.

Process

Sending

Send() should not only be triggered by input(). For example, when it receives the data that cannot be used or the other side announces the zero window, or there has been a lot of data in the *unsentList*. So, the better way is also triggering the send() on timer and when receiving anything. But with this circumstance, send() may have nothing to send when triggered. So, I named it as *trySend()* -- if it has nothing to send, just return.

trySend has three job: sending data, sending ack and sending FIN, and they can be compatible at a time.

- Sending data: calculate how much it can send from total data in the *unsentList*, peer's rwnd, it's cwnd, and choose the minimum value and prepare data.
- Ack number: get the lastest ack number from global state's variable.
- FIN: decided by connection's status and the left data in *unsentList*. If it cannot sent all left data this time, it cannot send FIN.

The last thing to do is hanging the buffer to the *sentUnackList*.

Receiving

When it gets the data, it should validate the packet, like the checkSum, the real length and if the seqNo equal than current record. Then, it determines how much data can be acked from the ackNo, and pop out that much buffer(s) from the *sentUnackList* and hang it/them to *unsubmitedList*.

When it gets some "useless" packets, like outdated segments or 0 data size segment or unexpected seqNo, it means the sender needs the help from the receiver to synchronize, so it should immediately send a ack to the sender.

Timer

When on timer, it checks all buffers in *sentUnackList*, and decide whether to resend it, and whether to shut down the connection.

Challenges

1. Handling EOF: In the initial version, when the receiver saw the FIN, it immediately close the output channel. After analyze byte by byte of the output file and trace log. I found only the data when piggybacking with the FIN will be lost. Then, I realize I should lower the priority of handling FIN on receiver side.
2. Zero window problem: When testing with a slow outputting receiver, it could be possible the receiver announces a zero rwnd to the sender, and the sender will never send anything. And because the receiver gets nothing, it also won't update the rwnd to the sender. So, I tried to modify the code -- the sender will periodically send meaningless segment -- a segment with no data. But here is another question, the receiver won't reply anything if nothing to update to the sender. So, I forced the receiver to reply once it gets a segment.
3. Avoid too many meaningless segments in flight. Because the sender and receiver will both send empty segments, these segments may waste too much resource. So, I set a flag: only when it has ack to announce, or has data to send, or meets zero-window problem, it can send dataless segments.

Tests

- Test on ctcp_tests.py : score from 13 to 17.

  I tried to find out if a larger/ smaller timeout can get a better result, but it's clueless.

- Test on generic_tester.py ctcp_mininet.xml : score 240/240

- Test on generic_tester.py ctcp.xml : score 40/40

  Spent most of time on it. It turns out that I need to change the timeout punishment from 2 RTT to 5 RTT.

- Test on Internet Web servers : sucessful.

Bugs

Currently, it uses the config's RTT time and never update it. I think a good TCP model should measure the RTT all the time. Also, there should be a notice when the sender reads the EOF, otherwise we cannot know if it has sent over all the data and keep waiting.