bTCP

Cas Haaijman, s4372662 Koen Sauren, s1024202

May 8, 2020

1 Introduction

For this assignment we had to create an implementation of a network protocol similar to TCP, called basic Transmission Control Protocol, or bTCP. bTCP should provide functionality for flow control and reliability (in our case Go-Back-N). The protocol has a three-way handshake for both connection establishment and termination.

2 Design decisions and justifications

We chose to implement reliability using the Go-Back-N solution with one slight alteration from the one in the course book. Instead of using arrays for our buffers, we chose to use the lists python uses instead of arrays as queues, which meant we did not explicitly need to maintain a buffer start or end location. We chose to use Go-Back-N because it is the most light-weight of the three, sorting is neither required at sender side nor at receiver side and it is the most widely used.

For our timeouts, we chose to use pythons asyncio framework. This works well, even though our implementation is slightly complicated, but this may be done in a simpler way. We chose to do it like this, because this we we know for sure a timeout is detected.

To build packets, we made a 'factory' class, which is easily used to build packets. The same class can also be used to easily decode packets.

3 Extra features

We chose to implement functionality for sending regular messages from the server side. Both client and server side are now able to use both the `send()` and `recv()` methods.

The original code that implemented `netem` didn't work, as using the command required sudo rights. We fixed this by piping the password of our own user accounts into the password request prompt. You can run the test framework by changing the 'password' field at the top of the testframework.test file manually, or by adding your password after argument -p.

4 Finite State Machine

Variables used:

seqnum	the variable that maintains the location of the last sent message of the sending buffer
expseq	the variable that maintains the expected sequence number of the next received packet
rwindow	the advertised window size of the receiver
swindow	the receiving window size of the sender

Abbreviations used:

START	application layer creates socket and calls connect()
TIMEOUT	a timeout has occured
EXCEED	The number of tries has exceeded the threshold.
MSG(x,y,z)	A message (could be ACK, SYNACK, SYN, FIN, ACKFIN or MSG) with x the Sequence Number field, y the Acknowledgement Number field and z the window size. MSG is a message without any SYN, ACK or FIN flags
APP-SEND	the sending logic is called from application layer
APP-RCV	the receiving logic is called from the application layer
ADD-SB	add the message to the end of the receiving buffer
ADD-RB	add the message to the end of the receiving buffer
RMV-SB	remove the message at the start of the receiving buffer
PASS	pass the receiving buffer to the application layer and clear it
REFUSE	client refuses to send data from upper layer
RBF	the advertised window size is 0
RBOK	the advertised window size is higher than 0
SBF	the sending buffer is full
SBOK	the sending buffer has room for the packets to be sent
SEND	send the message at seqnum of buffer
SEND-ALL	send all messages from base to seqnum of buffer, but not more than rwindow packets.
N-CORR	the packet is not corrupted
CORR	the packet is corrupted
N-SEQ	expseq does not match that of the received package
SEQ	expseq matches that of the received package
IGNORE	nothing is done
OTHER	A package is received that has no specified behaviour
CLOSE	the connection is closed
ABORT	send an en error message to the application layer and close the connection
APP-CLOSE	the application layer closes the socket
RCV	The size of the receiving buffer.

Finite state machine for the client followed by the finite state machine of the client: