basic Transmission Control Protocol (bTCP)

Networks and Distributed Systems

Project

Onno de Gouw

Stefan Popa

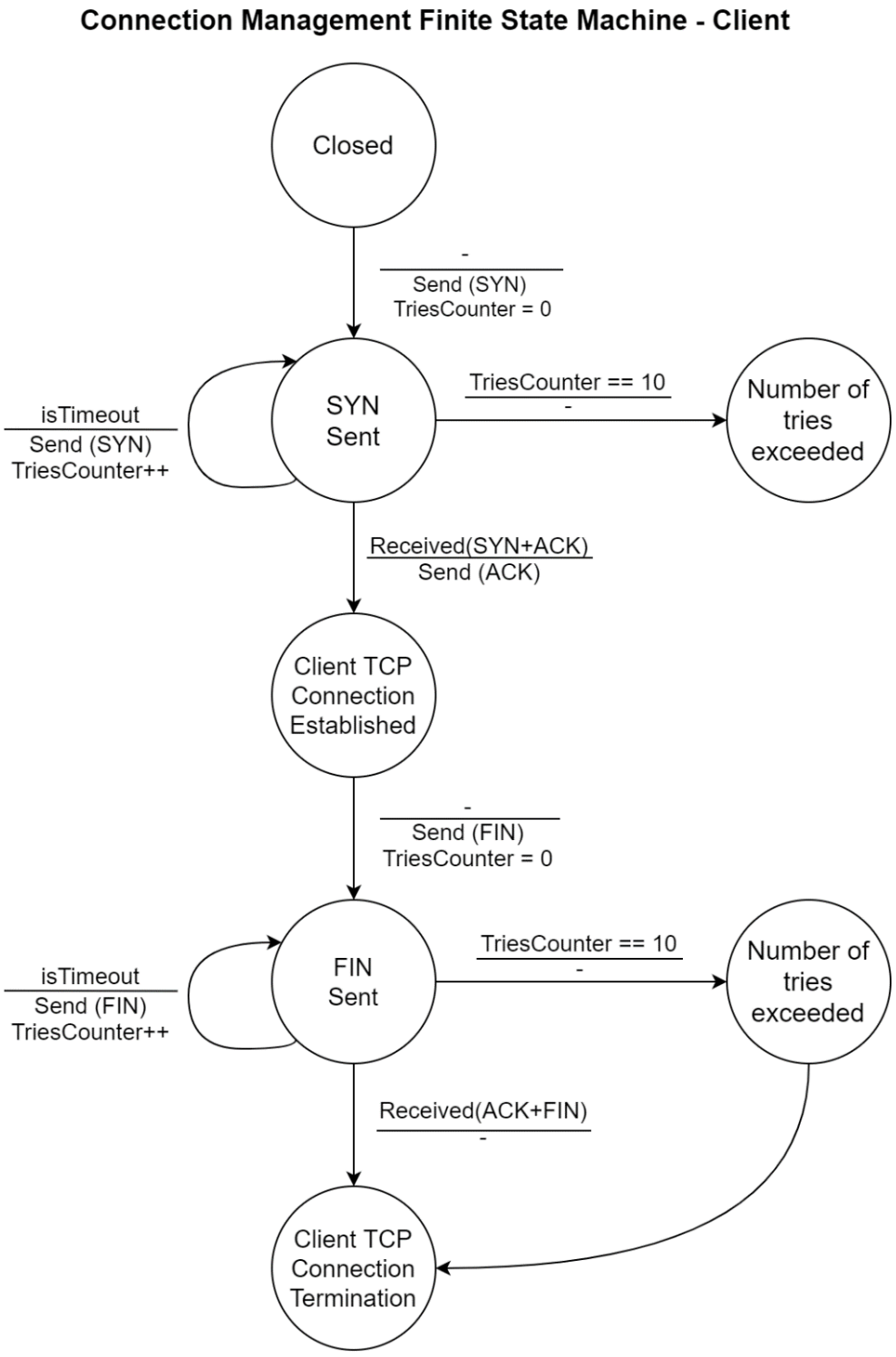
08-05-2020

1. Introduction

For this project, we have written the sending and receiving transport-level code for a reliable data transfer protocol, called bTCP using Python. bTCP stands for basic Transmission Control Protocol and it borrows a number of features from TCP.
In this report we are going to show finite state machines for the bTCP connection management and explain their meaning. Also, we are going to shortly explain the implementation and document the design decisions that we made including a justification for these decisions.

2. bTCP connection management finite state machines

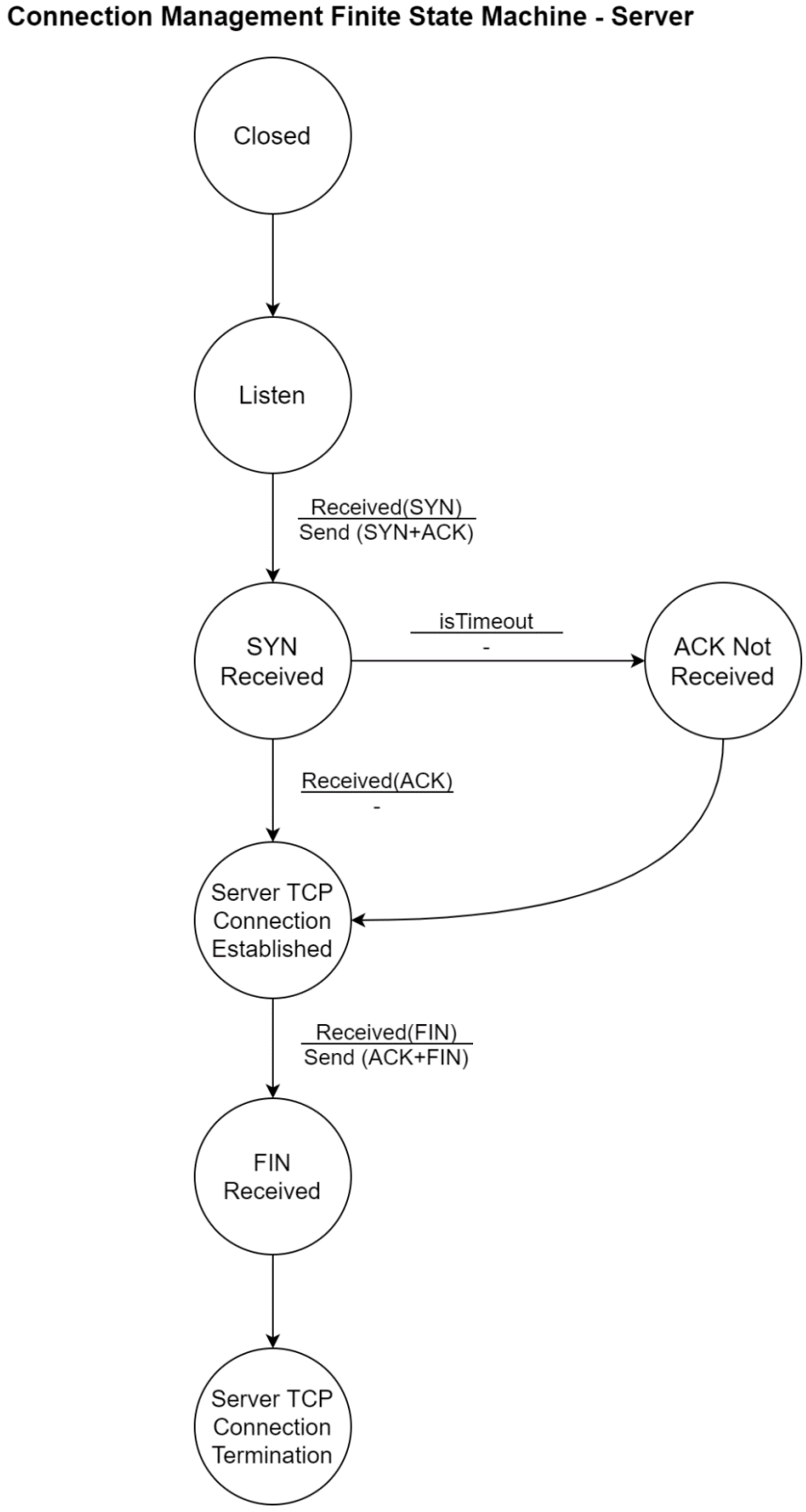
2.1. Client



As can be seen from this finite state machine, the client will send a segment with the SYN flag set to the server and it will also initialise the counter for the number of times it should try to resend the segment to zero. In the state SYN Sent, the client will wait to

get a reply from the server. If the timeout is reached, it will resend the SYN segment and increment the counter by one. If the maximum number of tries has been reached, which in our case is ten, the connection fails and an error message will be displayed. If the SYN+ACK segment has been received, the client will send an ACK segment back to the server and the connection is successfully established. Now, when the client wants to end the connection it will send a segment with the FIN flag set to the server and it will also initialise the counter for the number of times it should try to resend the segment to zero. In the state FIN Sent, if a timeout is reached the client will resend the segment and it will increment the counter by one. If the number of tries has been reached, the client will just assume the server had enough time to close the connection and end its own side of the connection. However, if the ACK+FIN segment is received, the client will close the connection immediately.

2.2. Server



As can be seen from this finite state machine, the server will listen for any connection requests. If the server receives a segment with the SYN flag set it will reply with a segment that has the SYN+ACK flag set. In the state SYN Received, the server will wait

for the acknowledgement coming from the client. However, it does not care if the ACK is received or not and it will automatically start the connection if the ACK is not received within a certain time. Now, after all data has been sent, when the server receives a segment with the FIN flag set it will reply with a segment that has the ACK+FIN flag set. After this, it does not have to wait for any other segment and therefore it will automatically close the connection.

3. Implementation and Design Decisions

We are going to explain in short what we decided to implement within our bTCP protocol and we will go more into depth about certain design decisions that we made within our project and that we consider being important:

Implementation:
The connection establishment part of this protocol is implemented in the way it is explained in the assignment, which is similar to how TCP uses a three-way handshake. The same thing can be said about the connection termination part.
When it comes to flow control, the only thing that we implemented is the window size check, which is explained in the assignment as well.
For reliable data transfer, we have decided to use the cumulative acknowledgement scheme, because we think it is the most efficient out of the three possibilities that we had. Also, we felt that it was the most clear and understandable way to implement the reliable data transfer.

Design Decisions:

- We have decided to implement the flags in the following way:
  The flag part of the header is implemented with a format '00XXYYZZ', where XX is used for the SYN flag, YY is used for the ACK flag and ZZ is used for the FIN flag. If the flag is set then the two bits are '11' and if the flag is not set then the two bits are '00'. We have decided to do it in this way, such that we do not waste the space that we have reserved for the flags and at the same time keep the same formatting for all three flags.
- We decided to implement most of our reliable data transfer and flow control code in the *lossy_layer_input* method for multiple reasons. Firstly, since this is the only method from the client and server socket respectively that is used by the lossy layer this means that it runs in a different thread than all the other methods of the sockets. Therefore, as we have also noticed ourselves while creating and testing the protocol, if the code would have not been in this method, certain race condition problems would appear, for example the counter that keeps track of how many segments have been sent and not acknowledged would go out of sync between the thread of the lossy layer and the thread of the socket. A second reason is that since everything related to reliable data transfer and flow control is now in the same thread, the transmission also works faster.
- We have chosen the format string 'HHBBhH' to build the header of our segments because of the following: The first two characters – 'HH' – of this format string, mean that the first two values should be both interpreted as a two byte unsigned short. Because these two values are the sequence number and the acknowledgement

number respectively it makes sense to interpret them as unsigned values, since they should not be negative and it can also happen that they are large. The next two characters – 'BB' – represent the flags and the window, which are both one byte unsigned chars. The reason for this is that the flags do not have to be interpreted as integers and because the window should not be allowed to be negative either. Finally, the last two characters of the format string – 'hH' – represent the data length and the checksum. In this particular protocol, the data length cannot be larger than 1008 bytes and thus it does not matter if we allow negative values for this field. For the checksum, we should not allow negative values, because we are working with the bits themselves and we do not want to add any interpretation to these bits.

- We have decided to use many of our variables and buffers as global variables for the classes, such as the sequence number, the acknowledgement number, the segment counter or the send/receive buffer, because they are used in multiple parts of our code. It would have been harder to declare them locally in each method, because these methods contain multiple scopes and this would mean that we would have had superfluous declarations. Moreover, since we also have the most important code in the *lossy_layer_input* method which is called every time a segment arrives, we were forced to make some variables global to the class because they would have been reset every time *lossy_layer_input* would be called if they were declared locally.
- We decided to create methods such as *build_segment* or *unpack_segment*, because they do shorten our program quite a bit by avoiding superfluous and duplicate blocks of code. We also believe it improves the readability of our code.
- Since the acknowledgement sent by the client in the last step of the three-way handshake can be ignored, we have also decided to send that segment with both the sequence number and acknowledgement number zero. This is because in this way we avoid making the server send an acknowledgement for this segment if it is not received or if it has a wrong checksum.
- When it comes to our testing framework we noticed that there are multiple ways you can go about implementing it. After some thinking, we decided that the easiest way to implement it is by using the *_app.py file directly. This only requires two lines of added code in each testing method and the first line of code can combine launching the localhost, sending the data to the server and receival of data by the server. Finally we used *assert* to check whether the input and the output files contain the same content afterwards.
- We have decided to separate the data in *client_socket* rather than doing it in *client_app*, because we feel that it is easier to handle the connection termination properly if we keep track of how much data is left to send in *client_socket*. Therefore, in *client_app*, we simply read the whole data from the file and pass this data to the transport layer by giving it as argument to the *send* method.
- We also decided to not make use of the data length field in the bTCP segment header, because it is not needed in any part of our program and it makes it even more efficient to not send this length in the header.
- Finally, since our reliable data transfer is implemented in the *lossy_layer_input* method, we had to make a slight change to how cumulative acknowledgements work. Since we cannot tell the method to wait 500 milliseconds (as it is specified in

the book) and still receive data at the same time, we immediately send an acknowledgement if a segment is received. The place where cumulative acknowledgments come into place is the case where certain acknowledgments are lost. Therefore, when the client sends a segment that has already been received, the server will simply drop this segment and send an acknowledgement for the expected segment. For example, if the client sends a segment with sequence number 45 and the server expects a segment with sequence number 60, the server will send an acknowledgement for segment 60 and when the client receives it, it can just drop all the packets between 45 and 60 from its send buffer.

4. Conclusion

Now that we have finished this project we can conclude a few things. Firstly, we are able to conclude that we have learned a lot from it. Before working on this project, we did not have as much experience with programming in Python as we have now. We learned a lot about how to use it and we got used to not only program and run our code on a Linux machine, but also using the PyCharm IDE. Secondly, this project gave us a better insight in how a transmission control protocol - including its flow control and reliable data transfer - works in practice. It was interesting to practically see the interplay between the different layers of the internet protocol stack. In summary, we found this project very helpful in showing parts of a network in practice and we learned a lot from it.