Networks and Distributes Systems
Basic Transmission Control Protocol

Jordy Aaldering, s1004292
Maria Zhekova, s1049083

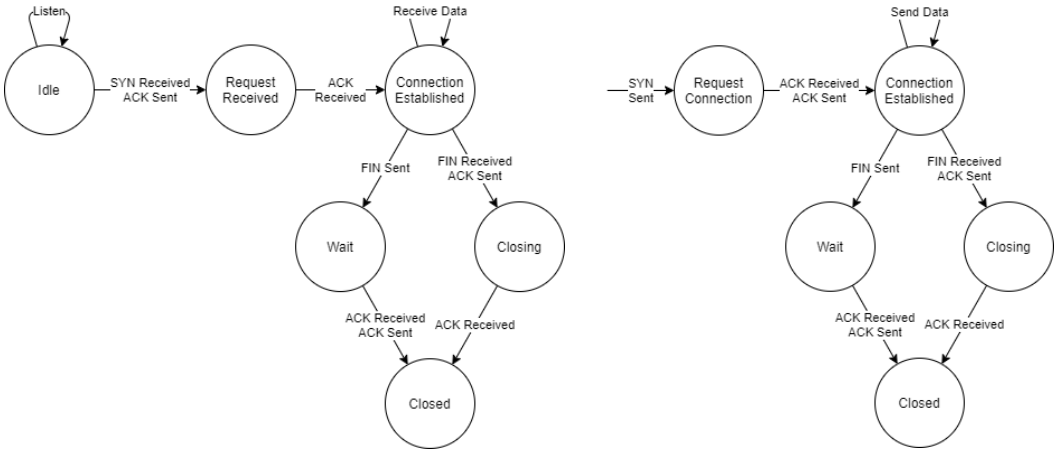
May 4, 2020

1 Introduction

This project presents the production of a reliable data transfer protocol called basic Transmission Control Protocol, bTCP, which borrows futures from TCP. This protocol sits between an application layer protocol and the network layer protocol. In our case, the server will be able to accept only one client.

2 Finite State Machines

The transitions of the protocol in response to input or user commands can be specified with the help of a finite state machine of the server and the client. Figure 1a describes the finite state machine of the server and figure 1b describes the finite state machine of the client.



(a) Finite state machine of receiver

(b) Finite state machine of sender

Figure 1: Finite state machines of bTCP

3 Production

The file structure of the project can be seen in figure 2 on the right. We started by creating a file *header.py* which is responsible for the header of the bTCP segment and contains the sequence number, the acknowledgement number, the flags, the window which represents the maximum packets the receiver is willing to receive, the data length, and the checksum. Supported flags are the synchronise flag SYN, the finish flag FIN, and the acknowledge flag ACK. The format for transforming the header into bytes and back is "*HHBBHH*". The file *packet.py* represents the bTCP segment which uses the *header.py* file in its implementation. The packet also holds the rest of the received data as a list of bytes.

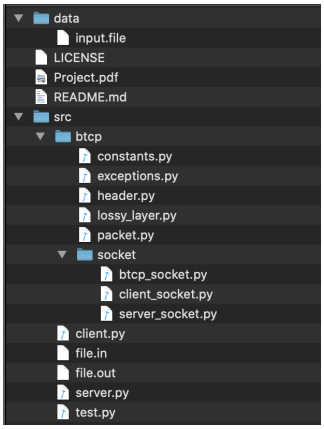


Figure 2: File structure

The Internet checksum is calculated in the *btcp_socket.py* file where we initialise the socket and raise an error if the checksum could not be calculated correctly. In subsection 3.1 and 3.2 we are going to look at the methods from the files *client_socket.py* and *server_socket.py*. In order to handle the exceptions as *ChecksumsNotEqual* and *InvalidChecksum* of the bTCP header, we created a separate *exceptions.py* file.

3.1 Connection Establishment

As we saw in figure 1 we need to perform a three-way handshake between the client and the server before exchanging any segments. The connection is established by several steps:

- The client connects to the server with the given IP address and port with the help of the boolean method `connect(ip:str, port:int, tries=5)`. The default number of tries before stopping is set to five. If this number of tries is exceeded, the method returns false and prints a client connection failure message, otherwise it starts the three-way handshake.
  This operation is done by calling the `_initiate_handshake(x, syn=True)` method, which verifies if there were any errors in the received flags and returns the packet with the SYN flag set. The sequence number x is a randomly generated 16-bit number that is passed as a parameter.

- The server acknowledges the connection inside `accept(recv_packet:Packet)`. This method generates a random 16-bit number y as a new sequence number and sets the new x to be the reply of `_acknowledge_handshake (y, recv_packet, syn=True)`, using the newly generated y. It stores the received x in the acknowledgement field and increases it by one. Then it sends the packet back to the client with the flags SYN and ACK set to true.

- If there are no errors in the received packet, the client acknowledges it with the help of the method `_acknowledge_handshake(x, recv_packet, syn=True)` by increasing the value of y by one and sending this new packet to server.

3.2 Connection Termination

In order to disconnect after exchanging the desired packets we also need to perform a three-way handshake which is similar to the establishment of the connection. Except this time we exchange the FIN flag instead of the SYN flag.

- This operation is started by the client in the `disconnect(tries=5)` method where, just like before, we generate a random sequence number x and use the method `_initiate_handshake(x, fin=True)` with the FIN flag set to true. Here we again have a default maximum of five tries to establish the termination of the connection.
- The server responds with the help of the method `disconnect(recv_packet:Packet)`, where the random number y is regenerated for the acknowledgement number and passed as a parameter to `_acknowledge_handshake(y, recv_packet, fin=True)`, which return the new value of x. These two values; x and y are passed as parameter to `_finish_handshake(x, y, fin=True)` where we check if the flags FIN and ACK are correctly set before sending the segment to the client.
- On the client side, if the connection is established successfully and the correct value of the flags are received we close the connection.

3.3 Socket Interface

In this part we will explain the methods of the socket interface between the file transfer application and the bTCP sockets.

- **client_socket.py**:
  In this file the important methods are `connect()`, `send(data:bytes)`, `disconnect()`, and `close()`. The `connect()` method was explained in 3.1, the `send(data:bytes)` method first separates the header of the data passed as parameter in order to extract the information, it then calculates the checksum and sends the packet in a reliable way to the server. The connection termination is performed by the `disconnect(tries=5)` method, which was discussed in section 3.2. Finally, in order to clean up any state we have the `close()` method which destroys the lossy layer and closes the socket.
- **server_socket.py**:
  The main methods for the server are `listen()`, `accept(recv_packet:Packet)`, `recv()`, `disconnect(recv_packet:Packet)`, and `close()`. The server is constantly listening for new connection. If a packet is received that has the SYN flag set to true, the server starts a three-way handshake, as explained in 3.1. If the FIN flag is set to true, a disconnection is desired and the method from section 3.2 for disconnection is called. The method `recv()` will take the data from the bTCP and deliver it to the application layer in order to keep track. Finally we have the closing method similar to the one in the socket of the client, which destroys the lossy layer and closes the socket.

3.4 The server and the client

- The server is constantly listening for packages and new connections, it writes the received information to a file, and in the end it closes.
- The client, on the other side, connects to the desired IP address and port of the server and sends its packets. In the end, it disconnects and closes.

4 Tests

We tested our code by first running the server file and then the client file from the terminal. The file that we transferred between them was a randomly generated lorem ipsum text with some numbers.

For a better understanding of what is going on we print a number of messages explaining the performed actions and states of the both server and client. For instance, if we look at Figure 3a, after the first packet is sent we see:
`Client send packet:  01926, 00000 | ACK: 0, SYN: 1, FIN: 0 | 09424 | ''`
Here the first number *01926* corresponds to the generated SYN value, the second, which is only 0's in this case, is the ACK number. After that we have the three flags next to 0 or 1, this represents whether flags are set to true or false, where 0 corresponds with false and 1 corresponds with true. Next we see some different number for each action, which represents the checksum of the sent or received packet. Finally, is the decoded data, which is only filled once in this test case when the server successfully receives the packet and displays it, otherwise we have an empty string.

We can see clearly in figure 3b, when the server receives the data and displays it, the flags are set to 0. Therefore, there are no flags and the first three-way handshake has terminated after the successful connection. Afterwards, the client again sends a segment, but this time the flag FIN is set to true in order to initialise a connection termination using a second three-way handshake.

```
Client connecting: localhost:30000
Client send packet: 01926, 00000 | ACK: 0, SYN: 1, FIN: 0 | 09424 | ''
Client recv packet: 61817, 01927 | ACK: 1, SYN: 1, FIN: 0 | 24851 | ''
Client send packet: 01927, 61818 | ACK: 1, SYN: 1, FIN: 0 | 09040 | ''
Client connected successfully
Client disconnecting
Client send packet: 13386, 00000 | ACK: 0, SYN: 0, FIN: 1 | 20391 | ''
Client recv packet: 53468, 13387 | ACK: 1, SYN: 0, FIN: 1 | 06206 | ''
Client send packet: 13387, 53469 | ACK: 1, SYN: 0, FIN: 1 | 05950 | ''
Client disconnected successfully
```

(a) Client test

```
Server connecting: 127.0.0.1:50122
Server recv packet: 01926, 00000 | ACK: 0, SYN: 1, FIN: 0 | 09424 | ''
Server send packet: 61817, 01927 | ACK: 1, SYN: 1, FIN: 0 | 24851 | ''
Server recv packet: 01927, 61818 | ACK: 1, SYN: 1, FIN: 0 | 09040 | ''
Server connected successfully
Server recv packet: 00000, 00000 | ACK: 0, SYN: 0, FIN: 0 | 04779 | 'Hello, World!
 This is a very fancy message containing lorem ipsum and some numbers.
Lorem5 ipsum do4lor sit amet, id tota verterem menandri pro. Etiam civibus accusam
 id duo. Tollit iisque assentior no per. Eu timeam corpora duo, ignota impedit viv
endum usu in.

Ius agam tation 4eirmod3 no, agam iuv5556aret mnesarchum vix ad, in pri tamquam ct
onseq556uat philos3ophia. E2u2m cu repdudia7r8e ad0ver6sarium,4456 eirmod principe
25s quaerndum vis ut. Vel vidisse elaboraret no. Ferri scribentur id sed, ne mei u
num velit dolor. Tritani intellegebat comprehensam vis cu. Cum et corpora definieb
as consequuntur, ea quo populo aliquip appetere, dicat vocibus insolens has eu. Se
a te oblique dolores maiestatis, duo deleniti voluptaria ei.

'
Server recv packet: 13386, 00000 | ACK: 0, SYN: 0, FIN: 1 | 20391 | ''
Server disconnecting
Server send packet: 53468, 13387 | ACK: 1, SYN: 0, FIN: 1 | 06206 | ''
Server recv packet: 13387, 53469 | ACK: 1, SYN: 0, FIN: 1 | 05950 | ''
Server disconnected successfully
```

(b) Server test

Figure 3: Test connection