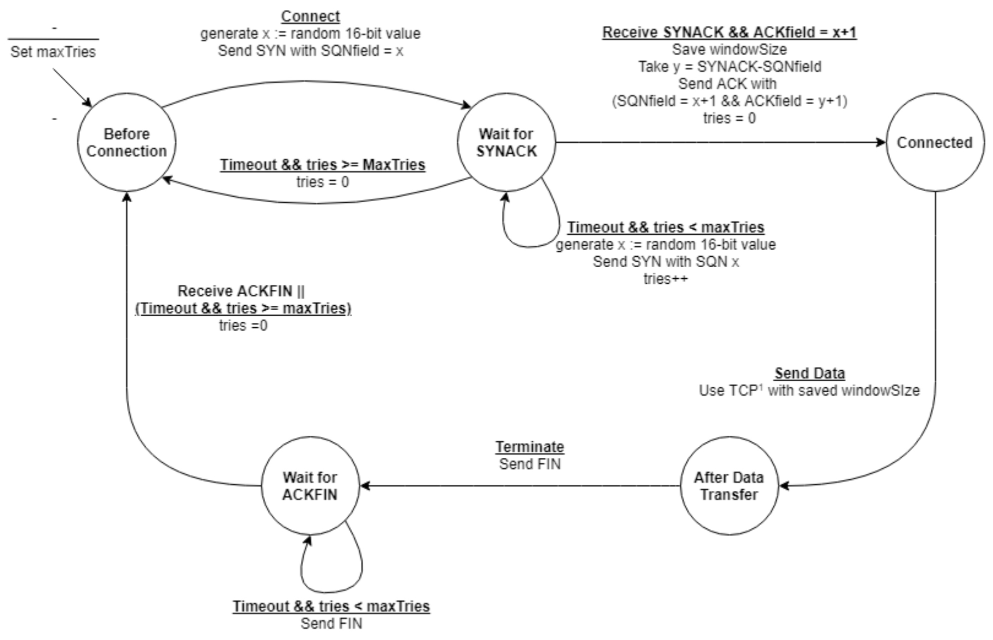
Networks and Distributed Systems
bTCP Project

Tom Aarsen (s1027401)
Bart Janssen (s4630270)

January 2020

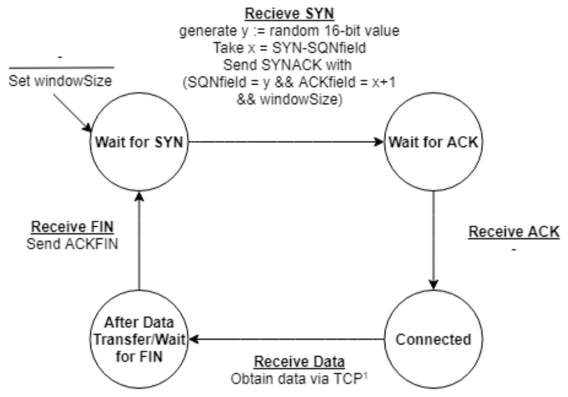
1 Automata

Client



-
Set maxTries
-
Connect
generate x := random 16-bit value
Send SYN with SQNfield = x
Receive SYNACK && ACKfield = x+1
Save windowSize
Take y = SYNACK-SQNfield
Send ACK with
(SQNfield = x+1 && ACKfield = y+1)
tries = 0
Before Connection
Wait for SYNACK
Connected
Timeout && tries >= MaxTries
tries = 0
Timeout && tries < maxTries
generate x := random 16-bit value
Send SYN with SQN x
tries++
Receive ACKFIN ||
(Timeout && tries >= maxTries)
tries =0
Send Data
Use TCP1 with saved windowSIze
Terminate
Send FIN
Wait for ACKFIN
After Data Transfer
Timeout && tries < maxTries
Send FIN

Server



1.1 Explanation

We made this automata at the start of our project, and used it throughout development for guidance.
For the Client and Server side we opted to not to reinvent the wheel and depict how the data is sent when using TCP. We chose this approach because in our eyes this is a very well-known protocol and it would be a little redundant.
We won't dive deeper into the specifics of this automata, since our actual implementation will be covered by section **2**.

2 Design Choices

2.1 Segment Structure of bTCP

For the segment structure of bTCP we chose a simple approach.
We made a central class `BTCPSegment` for every segment we would be handling. Furthermore we made several classes for special segments that e.g. are intended for the establishing and termination of the connection. These include the `ACKSegment`, `SYNSegment`, `FINSegment` and their respective combinations like `SYNACKSegment` and `FINACKSegment`.
`struct`'s we packed/processed all the incoming segments. Since we knew all the byte sizes of the different variables this was easy to accomplish. When a segment came in, we first packed it with a `checksum` of `0`, and then computed the 'real' `checksum`. With that new `checksum`, we packed the segment again, essentially only updating the checksum.

Before receiving any segment, Client or Server side, we unpacked all the variables from the packed segment, and immediately recomputed and checked the `checksum` of the that segment. If it contained errors, we would discard such a segment.

We need to store any combination of 3 flags in our Segment: `ACK`, `SYN` and `FIN`. To do this, we have 8 bits at our disposal.
We have opted to leave the 5 leftmost bits blank, and use the remaining 3 bits for these flags. The least significant bit represents the `FIN` flag, the second to least significant bit represents the `SYN` flag and the remaining bit represents the `ACK` flag.
So, a `SYNSegment` is represented by `0000 0010`, a `SYNACKSegment` is represented by `0000 0110` and a regular segment with no flags looks like `0000 0000`.
This way, we can easily represent all possible combinations of these 3 flags using just 3 bits. The data section of the segment would be filled with `1008` bytes of padding when any of the flags would be set, otherwise this section logically will be filled with the content of the file.

2.2 bTCP Connection Establishment

Using the notion of 'states' on both the Client and Server side, we wanted to achieve a clear overview of the different stages we could perform actions in. We tried to follow the path of our automata as closely as possible. For an overview of the states, we refer to the code in both `server_socket.py` and `client_socket.py`.

When the client wants to connect, he wants to initiate a three-way handshake. He does this by simply sending out a `SYNSegment` with a random 16-bit `seq_n`, an `ack_n` set to `0` and the current `window` size to the server.

After sending such a SYNSegment, we store all generated seq_n in a list. We store all of them, so that even if the Client is on connection attempt 5, it may still be able to receive a message responding to connection attempt 1, rather than only being able to respond to a timely response for connection attempt 5.

The Server looks out for an incoming segment with the SYN flag set, and would respond with an SYNACKSegment with a random 16-bit new_seq_n, an new_ack_n as the seq_n of the just received SYNSegment incremented by 1 and the current window size.
Would the Server not respond in time, then the Client would sent out SYNSegment's to connect until the amount of tries is surpassed.
If the Server has sent out the SYNACKSegment, the Client then would then look at an incoming segment with the SYN and ACK flags set. At the same time, he would check if the ack_n matches one of his seq_nś + 1 we stored earlier. If this all works out, then the Client would send out an ACKSegment to the Server with the new_seq_n being the ack_n of the received SYNACKSegment, the new_ack_n being the seq_n of that same segment incremented by 1 and the current window size. After sending this ACKSegment, the Client considers itself to be connected. The Server then would look out for a segment with the ACK flag set, and then he also considers himself to be connected to the Client.

2.3 bTCP Connection Termination

Connection termination is a little less involved than the three-way handshake as described in section **2.2**.
When the Client wants to disconnect (if it was not disconnected already), he would send out a FINSegment to the Server. Just as the Client had a maximum number of tries to connect, it also has for disconnecting. In this case it means that, even if the Server does not respond in time, after a maximum number of tries with no response, the Client considers himself to be disconnected. This FINSegment would have seq_n of 0 and the current window_size.
The Server will look out for a segment with the FIN flag set, and will send out a FINACKSegment to the Client telling him that the connection is closed. The Server also considers himself to be closed after that (although he may get another FINSegment due to a timeout).

2.4 bTCP Reliability

For reliability we stick fairly closely to the TCP implementation. We have a timeout for each segment, but if one is triggered, all unacknowledged segments are resent. Whenever 3 duplicate ACKSegment are received, another timeout is triggered.

In preparation of sending, we split up the data to be sent up in chunks of 1008 bytes, exactly the largest possible payload size in a bTCP segment. Each

data chunk is entered into a `BTCPSegment`, alongside a `seq_n` and a `ack_n`. The first `seq_n` is 0, while the `ack_n` is always the `seq_n` of the segment plus the data length of that segment. All segments that follow have as `seq_n` the `ack_n` of the previous segment.
However, that is not all. As you can imagine, with larger files, these values will quickly rise above a value possible to display within 16 bits. This is why we have opted for cyclic segment numbers, by performing a modulo operation. So, each `seq_n` and `ack_n` is modulo `2**16 - 1`, or `65535`.
The reason we chose modulo `2**16 - 1` instead of exactly `2**16` is because with modulo `2**16` sequence numbers will start to overlap after segment 4096, while for modulo `2**16 - 1` we don't experience duplicate segment numbers until segment number `21845`.
As a result, our program works for files smaller than `21845` segments, which is equivalent to `22019760` bytes, or roughly `21` gigabytes.

After preparing the segments to be sent, we add one final segment with an empty body, no flags and a custom set data length of `65534`. This data length is only possible for this final custom segment, as the data length of segments with real data must be between `0` and `1008` bytes.

When we start to send data, we first calculate how many segments we can send without overflowing the server's buffer, using a variable for last acknowledged and last sent, as well as a receive window and a total number of segments to send.
For each sent segment, we store the time it was sent at, and after each iteration we compare the times to see if there has been a timeout. In that event, we move the last sent back to the last acknowledged value, causing the program to resend the unacknowledged segments.

The server will receive this data, and check whether the `seq_n` is what we would expect it to be: the next segment. If it isn't, we simply resend the previous acknowledgement. If it is the expected segment, we store the data it contains, and send a new acknowledgement for the segment.

Then, eventually the client will receive this acknowledgement. It will compare the `ack_n` with the last acknowledged `ack_n`, and see if they are equivalent. Upon receiving three such duplicate `ACKSegment` in a row, we also trigger a timeout.
However, such a duplicate `ACKSegment` timeout can only occur once per `ack_n`, as otherwise if the client sent 20 segments, and the first one is lost, the client would receive 19 duplicate `ACKSegment` and trigger 6 different timeouts. To resolve this, we allow only one duplicate timeout trigger per `ack_n`.

Eventually, the server will acknowledge the final special segment with data length `65534`, and the client will receive this acknowledgement. The server will now know that it is finished receiving data, and can continue closing down if it wishes.

2.5 bTCP Flow Control

In order to avoid overflowing the buffer on the server side, every message sent by the server that the client receives includes a window size showing the amount of space left in the buffer. The client will update its receive window accordingly so that it can ensure the buffer will never overflow. The formula used for determining how many segments to send is the following:

```python
# Get number of segments to send
num_to_send = min(
    self.receive_window - (self.last_sent - self.last_acked),
    self.n_segments - self.last_sent
)
```

2.6 Additions

2.6.1 Function Termination

We have decided to make the `accept()` method on the `BTCPServerSocket` only return whenever the connection has been successful, so that the user does not need to wait themselves for a connection to properly establish before continuing with the remainder.
This functionality is also present in `BTCPClientSocket`'s `accept()` method.
Furthermore, methods like `connect()` and `send()` may throw Exceptions if connecting with the server failed.

2.6.2 Mutex Locks

Due to the existence of state variables that are modified and accessed by both the sending and the receiving threads for both the server and the client, we have opted to include mutex locks for all accesses to the variable `state` to help avoid race conditions. The server also has a mutex lock surrounding accesses of its buffer, and the client has them surrounding its `last_acked`, `last_sent` and `sent_times` variables, as both threads interact with those as well.

2.6.3 Additional parameters

We have added two additional parameters: `debug` and `retries`. The former will, when `True`, output some debugging statements regarding the states of the client and server, as well as special events like timeouts or the final segment being received.
The latter affects the amount of times the client will attempt to connect or disconnect before giving up.

The default values for these parameters are `False` and `20` respectively, and have proven useful for testing purposes.