Project
bTCP: basic Transmission Control Protocol

bTCP project, Wed 23 Feb 2022, version 1.0

Deadline for FSM drawings: *Wed*, 2022-03-09, 23:59 (Amsterdam time). No late submissions!

Final deadline: Fri, 2022-04-29, 23:59 (Amsterdam time). No late submissions!

Contents

Revision History

Revision	Date	Author(s)	Description
1.0	2021-02-23	Pol Van Aubel	Initial version for 2022

1 General remarks

Please read the following carefully:

- Work in groups of two.
- Enroll in a project group, *preferably* the one with the same group number as your assignment group if you're partnering with the same person.
- Each deliverable (python source file, drawing, pdf file, etc) should contain the names and student numbers of all group members either in the contents or the filename.
- Create a .tar.gz or zip file containing all of your deliverables.
  - Please remove `large_input.py` from your submission!
- The name of that file should contain the student numbers of all group members (e.g. `s123456_s987654.tar.gz`).
- Submit the file via Brightspace (`http://brightspace.ru.nl`).
- Only *one* member of the group should submit the file, using the group submission.

You can earn up to 110 points. Point distribution is as follows:

- 10 points for the bTCP connection management finite state machines.
- 10 points for working connect & disconnect mechanisms.
- 40 points for a working implementation of reliability.
  - 5 points for sending and respecting correct sequence numbers.
  - 5 points for a working acknowledgement mechanism.
  - 10 points for handling corruption.
  - 5 points for handling packet loss.
  - 10 points for handling reordering.
  - 5 points for handling packet duplication.
- 20 points for a working implementation of flow control.
  - 5 points for window negotiation during the handshake.
  - 5 points for (correct) window updating during the connection.
  - 10 points for *respecting* the current window size.
- 10 points for *combining* both client and server socket into one class.
- 10 points for the report.
- (Up to) 10 BONUS points for including extra useful features. Make sure to document these in your report as well. Think things like extending the application to send & receive multiple files, more aggressive test cases, a fast retransmit mechanism, etc.

We *may* take away some points if you deviate too far from the instructions even though your code technically works, e.g. if you do not implement the bTCP state machine.

You *are not allowed* a Fail for this project, you *must* make a serious attempt. This project will account for 20% of your final grade if and only if the grade for the project is higher than your exam grade *and* you receive a passing grade for the exam (5.5 or higher).

2 Overview

In this project, you will write the sending and receiving transport-layer code for a reliable data transfer protocol that we call bTCP, short for basic Transmission Control Protocol. As the name suggests, bTCP borrows a number of features from TCP:

- First, it guarantees reliable delivery of application layer data to the destination host. One of your tasks will be to define exactly how it accomplishes this.
- Second, it provides flow control (*not* congestion control!), i.e., the process of managing the rate of data transmission between sender and receiver to prevent a fast sender from overwhelming a slow receiver.
- Finally, it is connection oriented. Connections are established through a three-way handshake and shut down through a different kind of handshake.

Note that bTCP implements some of these features differently from TCP, and you have a lot of freedom to choose your particular implementation method. Unlike TCP, bTCP does not implement congestion control, as this is a complex topic and you'll already have your hands full. Also unlike TCP, bTCP does *not* implement multiplexing! Only one connection is supported, there are no port numbers. We will describe each feature in more detail in the following subsections.

As a transport layer protocol, bTCP sits between an application layer protocol and a network layer protocol:

- To the application layer, it provides the service of reliable data transfer. This service is available to applications through the bTCP **socket interface**.
- bTCP uses the (possibly unreliable) segment delivery service provided to it by the network layer.

So, the software you will be writing takes data from the application, and gives data to the application, via the socket API, i.e. the `send` and `recv` methods. It turns that data into segments, and sends them into the network layer. Rather than *use* a socket that Python provides for you, *you* are writing the socket itself.

We will provide you with a lossy layer that you use as the network layer and on top of which you will build your bTCP protocol. The general idea is visualized in Figure 1.

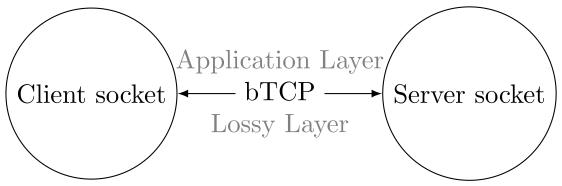

Figure 1: The bTCP stack.

In addition to writing Python 3.X code, you will be asked to draw a few finite state machines for the bTCP connection management and write a report containing your design decisions and the justification for these decisions.

Please note that the drawing of these state machines has an **early deadline**. This is so that we can give feedback on and publish the correct statemachine. If you do *not* submit this state machine in time, the points available for it will *not* be awarded even if you later submit a correct state machine in your final submission.

To be able to test your socket for correctness, we have provided a simple file transfer application between a client and server which uses the services provided by bTCP. A bTCP server socket cannot accept more than one client simultaneously, so this application is not required to accept more than one client either. For basic bTCP operation, you *should not need* to alter these client and server applications. They already call the right socket methods, you just need to make those methods behave correctly. Still, you should feel free to inspect their code and, if your bTCP implementation requires it, alter it accordingly[1]. Furthermore, if you combine the server- and client socket into *one* implementation, you will need to make the application use that combined implementation instead.

We will go over the concrete tasks in a later section.

2.1 Segment structure of bTCP

The data stream originating from the application layer is divided into individual units of data transmission called bTCP segments. These segments are what is sent between client and server. A bTCP segment consists of a segment header of 10 bytes and a data section with a fixed size of 1008 bytes. The data section follows the header and consists of payload data received from the application layer, padded with zero-bytes if it cannot fill the entire 1008 bytes with payload data. Note that the data section may be empty, i.e., it might contain 1008 bytes of padding. Segments with an empty data section can be found, for example, during connection establishment and termination.

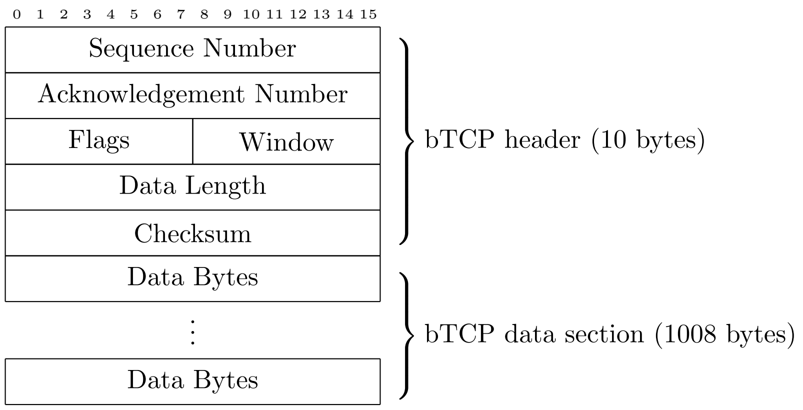

Figure 2: A bTCP segment.

The fields shown in Figure 2 have the following meaning:

- Sequence Number (16 bits): If the SYN flag is set, this contains a randomly chosen initial value intended as the first sequence number for this new connection. If the SYN flag is clear, this value is used to order different segments; its value then depends on how you chose to implement reliable data transfer[2].
- Acknowledgement Number (16 bits): If the ACK flag is set, this value is used to acknowledge received segments/data. Its value depends on how you chose to implement reliable data transfer. If the ACK flag is clear, this field has no meaning.
- Flags (8 bits): This contains the three flags ACK, SYN, and FIN. It is up to you how you represent these within this single byte, although we have provided you with a reference implementation that packs them into the three least significant bits of the Flags byte.
- Window (8 bits): Part of flow control, this field specifies how many segments the receiver is willing to receive. (Note that since a segment can carry 1008 data bytes, and the Window is only 8 bits, it is not reasonable to specify the Window in number of data bytes like in TCP.)
- Data Length (16 bits): Specifies the number of bytes in the data section that correspond to real payload data bytes. Even though the bTCP segment has a fixed length, we still need to be able to get the actual payload data from the segment without the padding. Without a length field it would be impossible to differentiate between "a long run of zeroes that is actual payload data" and "a long run of zeroes that is padding".

[1]For example, `connect` *could* be implemented to return failure upon timeout, in which case you may want to make the application loop waiting for it to succeed.

[2]Note that the maximum value of 65535 makes it inefficient to keep track of individual bytes – that would only allow for 30 simultaneous 1008-byte segments in flight at the same time, but the network and buffers can handle *much* more than that.

- Checksum (16 bits): The bTCP checksum, computed according to the Internet Checksum specification (refer to lecture 3, assignment 3, and assignment 5 for details). The checksum is computed over the *entire* segment with the checksum field initially set to zero. You do *not* need to add a pseudo-header with network layer addresses in front of the segment. We chose the number of bytes in the data section so that the total length is an even number of bytes, for your convenience.

In the above we referred to flags as being set or clear. All of our flags consist of single bits only. Hence, we say that a flag is set if the corresponding bit is one and clear if the corresponding bit is zero.

2.2 bTCP connection establishment

Connection establishment in bTCP is the same as in TCP. Before any segments are exchanged between sender and receiver, a three-way handshake is performed.

The steps taken are the following:

1. The client randomly generates a 16-bit value, say x, puts this in the Sequence Number field of a bTCP segment with the SYN flag set, and sends the segment to the server.

2. The server receives that segment. The server puts $x + 1$ in the Acknowledgement Number field of a bTCP segment with both the SYN and ACK flags set. The server then generates a random 16-bit value of its own, say y, and puts this in the Sequence Number field of that segment. The server then sends the segment to the client.

3. The client receives that segment. The client puts $y + 1$ in the Acknowledgement Number field of a bTCP segment with the ACK flag set. The Sequence Number field of that segment is $x + 1$.

   - Just like in TCP, segments without data (i.e. Data Length set to 0) do *not* advance the Sequence Number.
   - The Sequence Number field of the first segment carrying *data* to the bTCP server should also be $x + 1$.
   - You *may* choose to piggy-back the data on this Acknowledgement segment. In that case, the Sequence Number advances normally.

The exact Acknowledgement Numbers and Sequence Numbers may differ depending on how you choose to implement acknowledgements. The numbers shown here are for TCP-style acknowledgements, where the number actually equals the next sequence number that the receiver *expects* to receive. Go-back-N- and selective-repeat-style acknowledgements as shown in lecture 3 and 4 might use x and y as Acknowledgement Numbers instead.

Should the client never get a response from the server to its initial SYN segment within some specified time interval, it will simply try the first step again. The number of retries should be bounded by some specified constant. The timeout value and number of tries are up to you to choose.

The handshake accomplishes a number of things:

1. exchange initial sequence numbers,

2. prove that both parties can reach each other, and

3. learn the advertised window size.

We haven't shown you how to send or learn the advertised Window size here. For more information on that, read section 2.5.

2.3 bTCP connection termination

Connection termination in bTCP is similar to connection termination in TCP, but simplified. The steps taken are the following:

1. The client sends a bTCP segment with the FIN flag set.
2. The server responds with a bTCP segment with the ACK and FIN flags set.
3. The client responds to the FIN with an ACK and closes the connection.

The difference between TCP and bTCP is that in bTCP we can *always* combine the FIN and ACK of step 2, resulting in a *three*-way, rather than four-way, shutdown.

Should the client never get a response from the server to its initial FIN segment within some specified time interval, it will simply try the first step again. The number of retries should be bounded by some specified constant. If the number of retries has been exceeded and a bTCP segment with ACK and FIN flags set has still not been received, the client should just assume that the server had enough time to close its end of the connection.

Again, the timeout value and number of tries are up to you to choose. It is perfectly fine to have the same constants for connection establishment and termination.

2.4 bTCP reliability

The bTCP protocol provides reliable data transfer: data is delivered from sending process to receiving process, correctly and in order. It is up to you how it accomplishes this. At a minimum, your solution should make use of:

- checksums, to verify the integrity of segments,
- sequence numbers, to identify segments,
- acknowledgement numbers, to determine *what* segments to retransmit, and
- timers, to determine *when* to retransmit segments.

Possible solutions include Go-Back-N, Selective Repeat, and TCP. Each has its own advantages and disadvantages. You are expected to explain which one you chose, and what advantages and disadvantages made you choose it ("simplicity of implementation" *is* a valid reason).

2.5 bTCP flow control

The idea behind bTCP flow control is that the receiver sends feedback to the sender about the state of its receive buffer as to not get overwhelmed by the sender, i.e., to make sure that this buffer does not overflow, because this would lead to data being discarded. To control the amount of data that can be sent, the receiver advertises a receive window, which communicates the spare room in the receive buffer. Every bTCP segment received by the receiver is acknowledged. The receiver advertises its receive window in these acknowledgements.

As already mentioned, the Window field in the bTCP header is only 8 bits wide. Since a segment can carry 1008 data bytes it is not reasonable to specify the Window in number of data bytes directly, like in TCP. You can specify the window size in "whole segments". Alternatively, you can multiply the window size by some constant value; e.g. a window value of 30 could communicate that the server is able to receive up to $30 \times 256 = 7680$ bytes.

During the three-way handshake, your client socket should observe the advertised window size. After connection establishment it should send as many segments as it can within the window, and wait for acknowledgement segments before sending any more segments. E.g. if the advertised window has size 100, then your client socket should send 100 segments (assuming your window specifies entire segments) and wait. Once the server has acknowledged the first segment, your client socket will be able to send another segment. This way there should never be more than 100 segments travelling through the network at any given time.

If you implement updating receive windows, i.e. where the server actively changes its receive window based on a fixed amount of buffer space available, you should ensure that if the window ever drops to 0, there is still enough communication to ensure that the receiver can notify the sender of an increased window size.

We assume you'll be able to figure out how to do all this from the lecture recordings of lectures 3 and 4 and the lecture slides.

3 Structure of the bTCP transport layer implementation

The bTCP transport layer implementation is sandwiched between the application layer and network layer. *Each* of those layers has *independently triggered* events. A few examples of events:

- segments can arrive on the network layer, requiring your bTCP application to inspect them for errors, reorder them, and make the ordered data available to the application.
- acknowledgement segments can arrive on the network layer, which requires no interaction with the application layer at all.
- the application may want to send some data, requiring your bTCP implementation to take that data, turn it into bTCP segments, and send them.
- a timer may expire, triggering a retransmit of unacknowledged segments.

Clearly, we are not dealing with software that has an easily defined flow through it, like a Read-Execute-Print-Loop. This is an event-driven process. In general, there are different ways of handling such an event-driven process, like asynchronous I/O. The solution we have chosen is a simple form of multi-threading:

- There is an "application thread". This is the primary thread of the client- and server programs, the one that would exist anyway. It is the thread in which the program calls the functions of the socket API, like `send`, `recv`, `accept`, etc.
- Then, there is a second "network thread". We explicitly create this thread in the lossy layer. This thread calls two methods in your bTCP implementation, which is clearly documented in the framework. This is the thread in which *most* of your code implementing the actual bTCP protocol will run.

3.1 The lossy layer

We simulate the network layer by building a lossy layer using UDP, which, as you should recall, is an unreliable transport layer protocol. There are two reasons for doing it like this:

1. it avoids complexity, and
2. it teaches you that the theoretical layered model as presented in both the textbook and lectures is not limited to the simple stack presented until now. We can wrap protocols in other protocols, and nothing prevents you from doing this. The data section of UDP may contain anything.

The lossy layer that is provided to you spawns its own network thread that continuously reads from the UDP socket.

Whenever a bTCP segment can be read from the socket, the lossy layer reads this segment and calls a method called `lossy_layer_segment_received`, passing the segment to it. You are supposed to write its implementation. That method should handle the incoming segment, i.e. things like checksum verification, receiving acknowledgements and registering their corresponding segment as being acknowledged, etc.

If no segment has been received for a set number of milliseconds (default 100, but you are free to change that constant), the lossy layer will call a method called `lossy_layer_tick`. Again, you are supposed to write its implementation. This allows you to do work in the network thread without having to wait for a segment to arrive. For example, you could trigger checking for timeouts on acknowledgements there.

Be careful, however: `lossy_layer_tick` will *not* be called "every 100 milliseconds", but only "every 100 milliseconds since the last segment arrived". In particular, if segments are continuously arriving 50ms apart, this method will *never* be called. So you cannot rely on counting the calls to this method to implement your timer! You will need some helper methods called from both `lossy_layer_tick` *and* `lossy_layer_segment_received` in order to ensure correct functionality if you do work in the network thread that is not directly handling received segments, which you *almost certainly will be.*

We have provided a *very* rudimentary implementation of these functions that manages to transfer a file on a *perfect* network with a fast enough receiver. It's using bTCP segments but none of the functionality (except Data Length) is implemented. It behaves like UDP: unreliable, with no transport layer error handling, or indeed acknowledgement of data reception, at all. The reason for this implementation is that many people got stuck on getting *any* segments across the network, and transferring data between application and network threads, without some demonstration of how to do so. You should build on this, adding the bTCP state machine, the buffer of unacknowledged segments for retransmission, the sending and proper handling of acknowledgements, etc. Of course, you should feel free to alter the given implementation where needed.

See the documentation in `client_socket.py` and `server_socket.py` for details about how to use all this.

3.2 bTCP socket interface

The interface between the file transfer application and bTCP is a set of bTCP socket methods. For the client these are `connect`, `send`, `shutdown`, and `close`.

- `connect` will play the client's part in connection establishment.
- `send` will take data from the application layer, encapsulate it in a number of segments and send these in a reliable way to the server.
- `shutdown` will perform connection termination.
- `close` cleans up any state.

For the server, these are `accept`, `recv`, and `close`.

- `accept` will play the server's part in connection establishment. Note that our bTCP implementation has no separation between `accept` and `listen` (i.e. there is no `listen` at all).
- `recv` will take data from bTCP and deliver it to the application layer.
- `close` cleans up any state.

You are free to make changes to this interface as you see fit, but the basic idea behind the interface should be respected. If you *do* make changes, be mindful that you alter the provided `client_app.py` and `server_app.py` accordingly.

3.3 Distinction between client- and server socket

Initially, you will implement all these methods in *two separate socket classes*: one for the client socket, and one for the server socket. Although bTCP is inherently symmetrical, you can start by implementing the data communication one-way and your file transfer application should be able to function using those one-way sockets.

The reason we do it this way is that for the initial implementation you may find it easier to have a clear separation: the sending side, responsible for sending payload-carrying segments and registering their acknowledgements, and the receiving side, responsible for receiving payloads and sending their acknowledgements. The `lossy_layer_segment_received` and `lossy_layer_tick` methods in both these classes will probably look quite different from each other.

Eventually however, once you have gotten these sockets working, for full credits you should combine the two sockets into a *single* socket implementation that is capable of handling *bidirectional* data transfer. The only difference between a server and a client socket then is on which one an application calls `accept` and on which one an application calls `connect`. For this, you will need to *merge* the implementations of the `lossy_layer_segment_received` and `lossy_layer_tick` methods, and alter `client_app.py` and `server_app.py` accordingly.

Back up your project (or use version control) before starting on this! That way, if you don't succeed, you still have the version with separate sockets to submit. Of course you are allowed to work on a merged implementation from the start, but be aware that we made the conscious decision to provide you with this simpler unidirectional version for the initial implementation for a reason. The project is complex enough as it is.

3.4 Thread safety

There will obviously need to be some form of thread-safe data transfer between the application thread and network thread and back. Booleans and enums can be used to signal state changes from one thread to another. You can assign the value `true` or `false`, or e.g. `bTCPStates.CLOSED` to an attribute in one thread, and read it (in a loop, or otherwise) in another thread. We do not think you will need more advanced synchronization (e.g. locks) in this project, but you may find otherwise. However, for passing around the bytes of payload data and the prepared segments, Python's `List` **is not inherently thread-safe**. To ensure that you can pass these between the two threads, you should use some instances of `Queue` — an inherently thread-safe FIFO collection — or other inherently thread-safe collections.

The rudimentary implementation we've provided uses Queues that are bounded in size: at most 1000 elements, where elements are `bytes` objects of at most 1008 bytes each. Until you get flow control working, you may find you need to make the Queue on the receiving side (much) larger to avoid losing data to Queue overflows.

Our rudimentary implementation uses these Queues to pass the *data* to the network thread. Turning the data into segments and sending these is only done in the network thread. Alternatively, you *can* send segments directly from the application thread by calling the right function of the Lossy Layer without issue, but because acknowledgements and retransmissions will need to happen in the network thread you will still need to pass the segments to the network thread.

4 What you are supposed to do

We have provided you with a number of Python 3.X files which contain the emulated network layer, i.e. the lossy layer, and the stubs for any procedures that you should write, with some already having a rudimentary implementation to transfer a file over a perfect network. These procedures are by no means exhaustive, i.e. you should feel free to add any helper methods you need.

In addition to this, you will find a file transfer application and a test framework, which will be described shortly. The flow which the program follows can be seen in Figure 3.

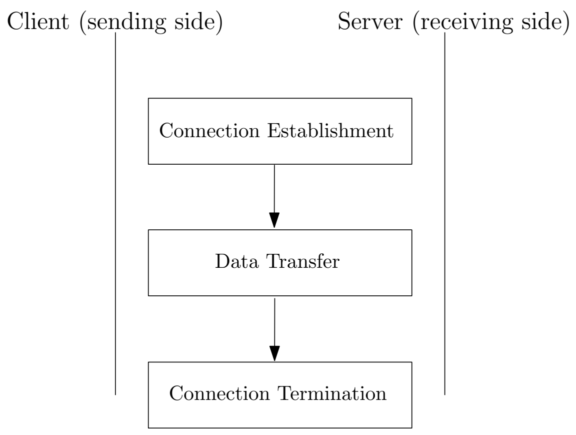

Figure 3: The different stages of bTCP

The rudimentary implementation already *calls* the requisite methods, even those not yet implemented.

Your tasks are the following:

1. Draw and submit the bTCP connection management finite state machines for both client, server, and a combined version for the combined socket. See the details in section 4.1. **Note that there is an early deadline for this!**
2. Think about how you want to implement the reliable data transfer and make a choice.
3. Implement the bTCP protocol using your solution for reliable data transfer and flow control. If you need an idea of how to split this into smaller steps, look at the description of the grade distribution in Section 1.
4. Use the test framework, described below, to test your implementation. Do not think of the test framework as something that you only use once you believe you have finalized everything. Testing and developing go hand in hand and happen concurrently.
5. Alter the file transfer applications where applicable to work with your implementation of the bTCP protocol. The test framework uses it to test your code.
6. Write a report documenting all design decisions and the justification for these decisions. Again, it is best if you write while you develop as opposed to writing the report at the end when you have forgotten about all the details. Writing things down also makes you reason about what you are doing.

4.1 Drawing the bTCP connection management finite state machines

You should draw and submit three finite state machines (FSMs), according to the format we have been using in the course, for bTCP:

1. One FSM for the progression of connection states for the *server*.
2. One FSM for the progression of connection states for the *client*.

3. One FSM that *combines* the two, merging states that are conceptually the same.

Make sure you include the transition labels, i.e. the events that trigger a transition between states and the actions that are taken in that transition.

To understand what we mean by all this, figures 4 and 5 show the connection management FSMs for a TCP server and client, as shown in lecture 4. The states that would be merged in a combined FSM would be `CLOSED` and `ESTABLISHED`. The merged states have all the transitions coming in and going out of the individual states of the separate FSMs.

Of course, bTCP has a different state machine, so don't just copy these (although it *will* look similar to these). We have provided an incomplete list of possible states in `btcp_socket.py` as an enum.

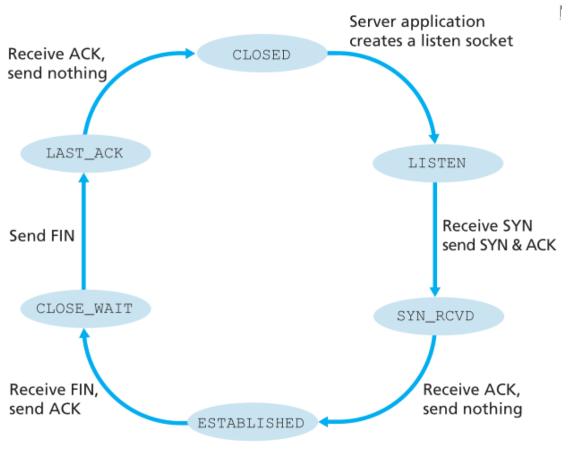
Server application creates a listen socket
CLOSED
Receive ACK, send nothing
LAST_ACK
LISTEN
Send FIN
Receive SYN send SYN & ACK
CLOSE_WAIT
SYN_RCVD
Receive FIN, send ACK
ESTABLISHED
Receive ACK, send nothing

Figure 4: Connection FSM for TCP server

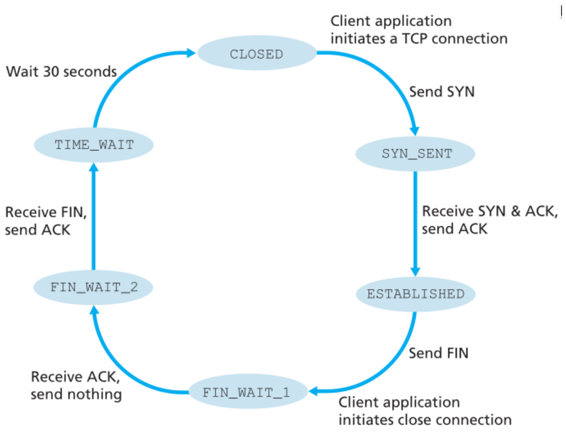
Client application initiates a TCP connection
CLOSED
Wait 30 seconds
Send SYN
TIME_WAIT
SYN_SENT
Receive FIN, send ACK
Receive SYN & ACK, send ACK
FIN_WAIT_2
ESTABLISHED
Send FIN
Receive ACK, send nothing
FIN_WAIT_1
Client application initiates close connection

Figure 5: Connection FSM for TCP client

The deadline for this part is separate from the other tasks! That way we can be sure you have started work on the project and provide you with the correct state machine before you get too far into implementing it.

4.2 Developing your implementation

First, consider using PyCharm. PyCharm is an integrated development environment (IDE) that provides, among other things, code analysis, integration with version control systems, and a graphical debugger. Free educational licenses are available.

Once you are done reading this document, **read the code we have provided!** We have added a lot of documentation that hopefully clarifies the details for each individual method.

You are strongly encouraged to develop your bTCP implementation in an incremental manner. Make your implementation work under ideal conditions and introduce other conditions incrementally:

1. ideal network (no packet loss, bit flips, etc).
2. network with spurious bit flips.
3. network with packet loss.
4. network with delays (sometimes exceeding the timeout value).
5. network with reordered packets.
6. network with duplicate packets.
7. network with all of the above problems.

The test framework was created in order to help you with this incremental approach.

4.3 Testing your implementation

4.3.1 File transfer on top of the bTCP protocol

In order to test your transport layer, you need an application to use it. We have provided a pair of *very* simple python processes:

- one server creating a bTCP server socket, accepting a connection, and receiving (a lot of) data; and
- one client creating a bTCP client socket, connecting to the server, sending (a lot of) data, and shutting down the connection.

You should feel free to alter `client_app.py` and `server_app.py` if you change the way the functions they call behave. E.g. if `connect` were to return a boolean indicating connection success, the application should probably handle it if it returns `false`.

The main method of testing your implementation will be by seeing that the data has not been altered for the recipient. Our implementation simply reads `large_input.py`, a 128 MiB file, from the filesystem and transfers its contents. This is enough to properly test your implementation. You can tell it to use a different file through command line arguments.

If our implementation of the file transfer application and our test framework are insufficient for your needs, you can write your own. In that case, you do not have to send an actual *file* from the file system through bTCP, you can just operate on `bytes` objects. As an example, `large_input.py` is also a python source file that contains a `bytes` object containing 128 MiB of data. You can use that object directly if you import the file.

Regardless of what you do, make sure to test with data larger than 64 MiB. We'll let you figure out yourself why we insist on that.

4.3.2 Test Framework

The test framework has test cases for each set of conditions. Each test checks if reliability is achieved for the respective context, that is, if the content sent by the client is received in unaltered form by the server. The test cases are already written for you – you just need to run the test framework and ensure your implementation passes all tests. As-is, the given rudimentary implementation should pass 1 of the tests: the one with an ideal network.

The test framework simply runs the file transfer client and file transfer server, checking that the file / blob of data sent by the client is the same as the data received by the server. If this does not suit your needs, you should feel free to come up with a different way of testing your sockets and implementing that in the test framework.

The framework makes use of the `tc netem` utility to simulate each of the problematic environments. This utility is found in all recent Linux distributions. A guide on `tc netem` is found here[6]. See Appendix A for some details on `tc netem`, including how to recover from a crashed test framework.

[6] `https://wiki.linuxfoundation.org/networking/netem`

5 Deliverables

To summarize, you are supposed to hand in the following:

- Drawings of the bTCP connection management finite state machines.
- Your Python 3.X implementation of the bTCP transport layer.
- A *simple* file transfer application using your bTCP transport layer to transfer a (large-ish) file (or hardcoded blob of data; you don't *have* to read this from the filesystem).
- A completed test framework.
- A small report documenting your design decisions and justifications for these decisions.

For instructions on how to submit these deliverables and their deadlines, please read the first page of this document carefully.

A tc netem – the network emulator

Our test framework makes the loopback interface misbehave on purpose. Note that you may have local applications running, like a caching DNS resolver, that do not operate correctly when this is the case. Should the test framework crash without cleaning up,

```
sudo tc qdisc del dev lo root netem
```

should reconfigure to a normal state.

For a quick understanding of the capabilities of tc netem, say we want reordering of 25 percent of packets with a correlation of 50 percent (those packets will be delayed 10 ms). Open a terminal on a Linux machine and add a reordering rule by running:

```
tc qdisc add dev lo root netem delay 10ms reorder 25% 50%
```

Then we play with nc to create a localhost UDP server and client which sends the content of the text file 'file.txt' to the server.

On one terminal run the following command to start a UDP server listening on localhost:

```
nc -u -l localhost 40000
```

On the other terminal run the following command to start a UDP client to send a file over UDP:

```
cat file.txt | nc -4u -q1 localhost 40000
```

On the server side, provided your file was large enough, you will notice re-ordering of the text.

Once you are done, we clear the reordering rule by running:

```
sudo tc qdisc del dev lo root netem
```

Remember the above command, you may need it to restore normal operation of the network if your application crashes during running of the test framework.