Project
bTCP: basic Transmission Control Protocol

March 19, 2020

Deadline: Friday 8 May 2020, 15:00 (Amsterdam time). Late submissions will not be accepted!

1 General Remarks

Please read the following carefully:

- Work in groups of two (or individually).
- Each deliverable should contain the names and student numbers of all group members.
- Create a ZIP file containing all of your deliverables.
- The name of the ZIP file should contain the student numbers of all group members (e.g. `s123456_s987654.zip`).
- Submit the ZIP file via Brightspace (`http://brightspace.ru.nl`).
- Only *one* member of the group should submit the ZIP file.

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

- 10 points for the bTCP connection management finite state machine.
- 25 points for a working implementation of reliability.
- 25 points for a working implementation of flow control.
- 5 points for the file transfer application.
- 20 points for tests.
- 15 points for the report.
- (Up to) 10 BONUS points for including extra useful features. Make sure to document these in your report as well.

This project will account for 20% of your final grade.

2 Overview

In this project, you will write the sending and receiving transport-level 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, 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 tore down through a different kind of handshake. Note that bTCP implements some of these features in a different way compared to TCP. Unlike TCP, bTCP does not implement congestion control, as this is a complex topic. We will describe each feature in more detail in the following subsections.

Recall that bTCP, as a transport layer protocol, sits between an application layer protocol and network layer protocol. To the application layer, it provides the service of reliable data transfer. This service is accessed through the bTCP socket interface. In turn, bTCP uses the unreliable segment delivery service provided to it by the network layer. In order to understand the interplay between the different layers better, you will also write a simple file transfer protocol between a client and server on top of the services provided by bTCP. A server is not required to accept more than one client. We will provide you with a lossy layer that models the network layer and on top of which you will build your bTCP protocol. This idea is visualized in Figure 1.

In addition to writing Python 3.X code, you will be asked to draw a finite state machine for the bTCP connection management, write a number of tests for your code, and write a report containing your design decisions and the justification for these decisions. We will go over the concrete tasks in a later section.

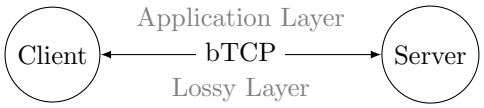

Figure 1: The bTCP stack.

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 of a fixed size, 1008 bytes. The data section follows the header and consists of payload data received from the application layer, possibly padded with zero bytes up to 1008 bytes. Note that the data section may be empty, i.e., it might contain 1008 bytes of padding. Segments of this kind can be found, for example, during connection establishment and termination.

A bTCP segment has the following format:

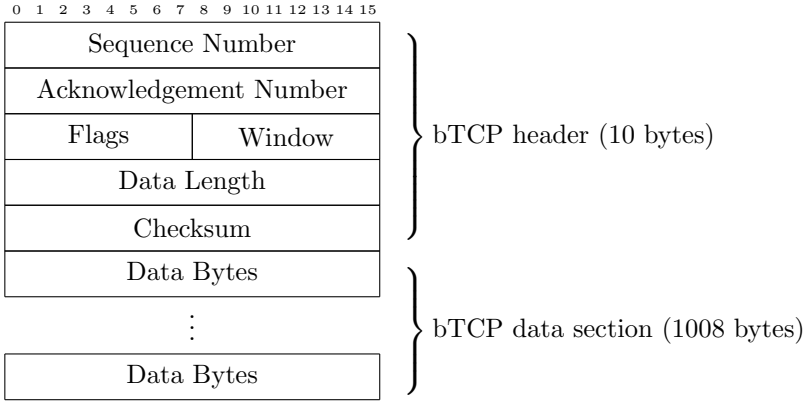

Figure 2: A bTCP segment.

The fields have the following meaning:

- Sequence Number (16 bits): If the SYN flag is set, then this contains a randomly chosen value. If the SYN flag is clear, then this value is used to order different segments and its value depends on how you chose to implement reliable data transfer.
- Acknowledgement Number (16 bits): If the ACK flag is set, then 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, then 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.
- Window (8 bits): Part of flow control, this field specifies how many packets the receiver is willing to receive.
- Data Length (16 bits): Specifies the number of bytes in the data section that correspond to real payload data bytes.
- Checksum (16 bits): This is the Internet Checksum. The checksum is computed over the entire segment with the checksum field initially set to zero. We chose the number of bytes in the data section to be divisible by 16 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 sends a bTCP segment with the SYN flag set. It randomly generates a 16-bit value, say x, and stores this in the Sequence Number field.

2. The server responds with a bTCP segment with both the SYN and ACK flags set. It generates a random 16-bit value of its own, say y, and stores this in the Sequence Number field. It stores $x + 1$ in the Acknowledgement Number field.

3. The client responds with a bTCP segment with the ACK flag set, a Sequence Number field equal to $x + 1$ and an Acknowledgement Number field equal to $y + 1$.

Should the client never get a response from the server in the second step within some specified time interval, then it will simply try the first step again, where the number of tries is 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 of initial sequence numbers, 2) proof that both parties can reach each other, and 3) learning the advertised window size.

2.3 bTCP Connection Termination

Connection termination in bTCP is similar to that 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 closes the connection.

Should the client never get a response from the server in the second step within some specified time interval, then it will simply try the first step again, where the number of tries is bounded by some specified constant. If the number of tries has been exceeded and a bTCP segment with ACK and FIN flags set still has not been received, then the client should just assume that the server had enough time to close its end of the connection. The timeout value and number of tries are up to you to choose. If 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 sequence numbers, acknowledgement numbers, and timers in order to determine when to retransmit a segment. Possible solutions include Go-Back-N, Selective Repeat, and TCP. Each has its own advantages and disadvantages.

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 segments being dropped. To control the amount of data that can be sent, the receiver advertises a receive window, which is the spare room in the receive buffer. Every bTCP segment sent is acknowledged at some point by the receiver and it is within this acknowledgement segment that the window is advertised.

During the three-way handshake, your client should observe the advertised window size, send as many segments as it can within the window, and wait for acknowledgment segments before sending any more segments. For example, if the advertised window has size 100, then your client should send 100 segments and wait. Once the server has acknowledged the first segment, then your client will be able to send another

segment. In this way there should never be more than 100 segments travelling through the network at any given time.

2.6 bTCP Socket Interface

The interface between your file transfer application and bTCP is a set of bTCP socket methods. For the client these are connect, send, disconnect, 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. Disconnect 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. 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.

2.7 The Lossy Layer

The bTCP protocol is sandwiched between the application layer and the unreliable network 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 often violated in practice. In fact, 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 thread that continuously reads from the UDP socket. Whenever data can be read from the socket, it reads this data and calls a routine that you are supposed to write, passing the data to it.

3 What you are supposed to do

You are provided 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. These procedures are by no means exhaustive. In addition to this, you will find a test framework which will be described shortly. The flow which your program needs to follow can be seen in Figure 3.

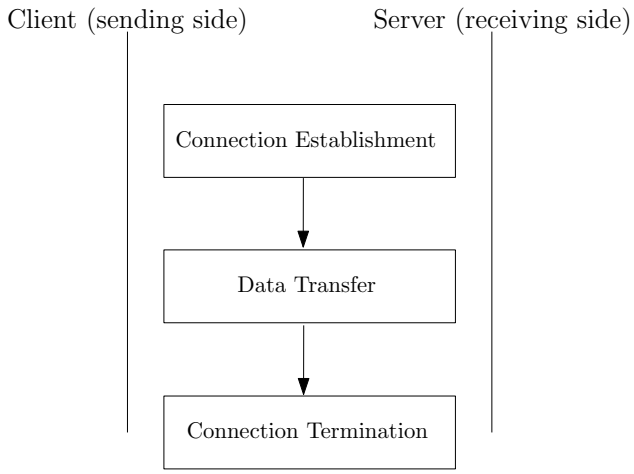

Figure 3: The different stages of bTCP

Your tasks are the following:

- Draw the bTCP connection management finite state machines for both client and server.
- Think about how you want to implement the reliable data transfer and make a choice.
- Implement the bTCP protocol using your solution for reliable data transfer. In parallel, work on the test framework that is described below. 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.
- Implement the file transfer application on top of the bTCP protocol.
- 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. In fact, writing things down makes you think about what you are doing.

You are strongly recommend to develop your software in an incremental manner. Make your program 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 duplicate packets.
4. network with packet loss.
5. network with delays (sometimes exceeding the timeout value).
6. network with reordered packets.

7. network with all of the above problems.

The test framework was created in order to help you with this incremental approach. We will describe it in the next subsection.

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.

3.1 Test Framework

The test framework has test cases for each set of conditions. Each test should check 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. You have to fill up the test cases, and ensure your implementation passes all of them.

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]. For a quick walkthrough, 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 (server listening on localhost)

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

On the other terminal run (client sending 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 all is done, we clear the reordering rule by running:

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

As an aside, packet loss can also be simulated using `iptables`, for example:

```
iptables -A INPUT -m statistic --mode random --probability 0.1 -j DROP
iptables -A OUTPUT -m statistic --mode random --probability 0.1 -j DROP
```

Each of these commands drops 10% of packets. The first drops packets which arrive at the machine and the second drops packets leaving the machine.

Finally, write a report that includes all of your design decisions and a justification for these decisions.

4 Deliverables

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

- A drawing of the bTCP connection management finite state machine.
- Your Python 3.X implementation (of the file transfer application and the bTCP protocol).
- A completed test framework.
- A report documenting your design decisions and justification for these decisions.

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

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