Project 2
bTCP: basic Transmission Control Protocol

05 May 2017

1 TCP

The Transmission Control Protocol (TCP) is one of the most widely used protocols on the internet. It is used so much because it provides reliability, flow control, and congestion control to data streams over unreliable infrastructure. TCP provides these features by defining a new packet header which is put inside the internet protocol (IP) packet payload segment. Important fields of this header include the sequence and acknowledgment numbers, the window size and the header checksum. The full TCP header[1] can be seen here:

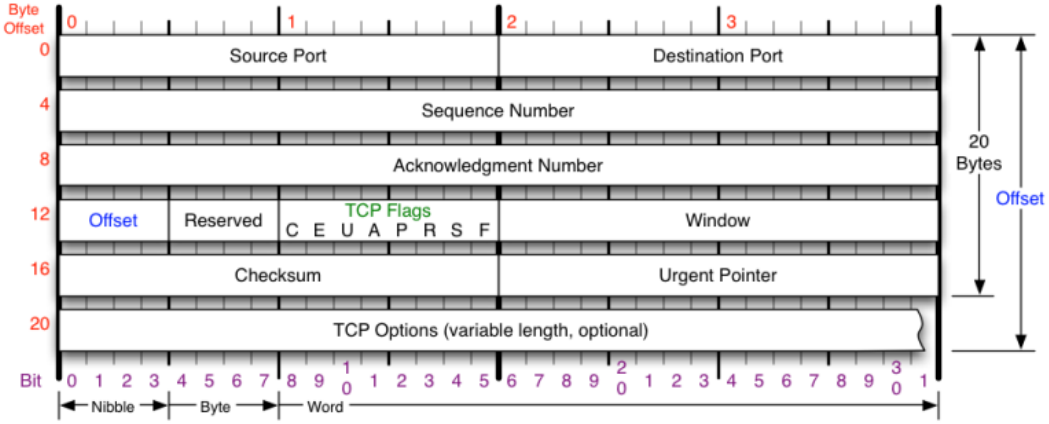

Figure 1: TCP header

[1]

Of the three basic features, reliability, flow control, and congestion control, we will concern ourselves with only reliability and flow control for this project. Reliability in TCP is handled primarily with the sequence and acknowledgment numbers. The basic idea is that a sender will increment the sequence number with each packet sent and the receiver will reply with a packet which has the acknowledgment number set to correspond to the sequence numbers received. In this way the sender knows what packets have and have not reached their destination and can resend as needed. Flow control is handled through the TCP window which is a 16 bit integer. The window value simply defines how many packets can be sent at a time. The idea being that if the receiver has a low amount of memory or is otherwise unable to handle a flood of data it can define a smaller window and limit the amount of incoming traffic.

1.1 TCP Flags

In the TCP header, we can see a section called ‘TCP Flags’. In TCP these flags are one bit each and are considered `off` if they are zero and `on` if they are one. In discussing TCP packets you will often come across ‘ACK Packets’, ‘SYN Packets’, ‘FIN-ACK Packets’ and others. What these mean is simply that the flag is `on`. That is, an ACK packet refers to a TCP packet where the ACK flag (A in figure 1) is set to 1. A FIN-ACK packets refers to a TCP packet which has both the FIN and ACK flags set to one. These packets can have a payload of data, but can also be empty as they are in the opening handshake.

1.2 Connection establishment: The TCP handshake

TCP has a three-way handshake which occurs before any payload data is exchanged. This handshake procedure initialises a beginning sequence number, a window size, and ensures that a connection can be made. The process is

1. Sender: The first SYN packet is sent to the receiver. Sometimes referred to as the original SYN.
2. Receiver: The SYN-ACK packet is sent back with the acknowledgment number set one higher than the original SYN.
3. Sender: An ACK packet is sent back and the sequence number is set to the number acknowledged by the receiver.

In this exchange the receiver sends one packet to the sender and thus the sender knows the window size that the receiver allows for. [2]

1.3 Connection termination

Similar to the connection establishment handshake there is also a connection termination handshake. This one is a four-way handshake.

1. Whichever party wishes to terminate the connection sends a FIN packet.
2. The receiver of the FIN packet responds with a FIN-ACK packet.
3. The initiator then ACKs the FIN-ACK and closes its connection.
4. The receiver gets this ACK and closes its connection.

[2] The TCP RFC can be found here https://tools.ietf.org/html/rfc761

2 bTCP

For this project, you will be implementing a basic form of the Transmission Control Protocol which will be dubbed basicTCP or bTCP. This protocol will use a simplified header and **will not** concern itself with the idea of congestion control. If you would like to read up on network congestion and the dreaded congestion collapse wikipedia has a section here: Network Congestion. You will code a client and a server where the client opens a connection and transfers a file. Your bTCP packets should be a fixed size of 1016 bytes. That is 16 bytes for the header and 1000 bytes for the payload of the packet.

2.1 bTCP header

The bTCP header is as follows:

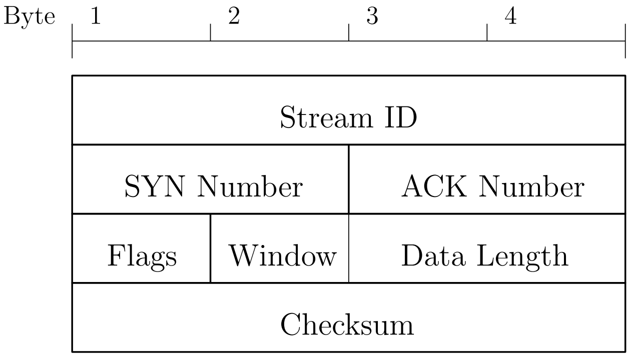

Figure 2: bTCP header

The field definitions are

1. Stream ID (32bits): A unique identifier given to each bTCP stream. Used to differentiate packets of different origin and destination.
2. SYN Number (16bits): Used to order packets in a given bTCP stream.
3. ACK Number (16bits): Used to acknowledge received packets.
4. bTCP flags (8bits): Used to define the flag state of a given bTCP packet
5. Window Size (8bits): Defines the number of packets allowed in transit with a maximum of 255.
6. Data Length (16bit): Defines how much of the 1000 bytes is data.
7. Checksum (32bits): A checksum of the header and payload data.

A few notes: The stream ID is an arbitrary label. You are free to define the stream ID as you see fit. The checksum has 32 bits of to accommodate the use of the crc32 algorithm[3][4]. SYN and ACK numbers should increment as the packets flow, but where they start and how they increment is left to you to define.

2.2 Reliability

Like full TCP, your bTCP implementation will support reliable data transfer. With proper acknowledgment the client will keep track of which packets have been acknowledged by the server and retransmit those that

[3] https://docs.python.org/2/library/binascii.html#binascii.crc32

[4] https://docs.python.org/3/library/zlib.html#zlib.crc32

have not been acknowledged after some period. It is left to you to determine how retransmission should be handled, but a timeout on each packet is not a bad solution. Your server should be able to reassemble any out of order packets.

2.3 Flow Control

Flow control is defined by the window size in the bTCP header. During the initial handshake your client code should observe the window size available, send as many packets as it can up to that window and wait for acknowledgments before sending more packets. For example, if there is a window size of 100 then your client code should send 100 packets and wait. When the server acknowledges the first packet your client can then send another packet. In this way there should never be more than 100 packets on the wire at any given time.

2.4 Flags

In this project we will emulate three of the TCP flags. SYN, ACK and FIN. These flags are single bits in the full TCP spec and thus can be either on or off. We will represent them with an 8-bit integer for simplicity of implementation. The states can be defined as you see fit, but be sure to have only one integer correspond one state of the flags. You are free to determine these states.

3 Project description

As mentioned above, you will be implementing a basic form of TCP over UDP. This is being done over a UDP socket simply to ease the process of implementing a new protocol and to aid in testing. You are tasked to implement bTCP and to make a working file transfer client-server combination which uses your bTCP implementation. You will need to fill in the provided server and client code in order to make a command line file transfer program. Your code should implement a handshake, provide reliable transport, honour window sizes, and reassemble the file correctly. Your code should transfer arbitrary files. It is important to note that you are free to improve upon this protocol (for which you will be awarded BONUS points) so long as you maintain:

- The opening handshake
- Reliable data transfer
- Flow control with the window size
- The termination handshake

What you need to code?

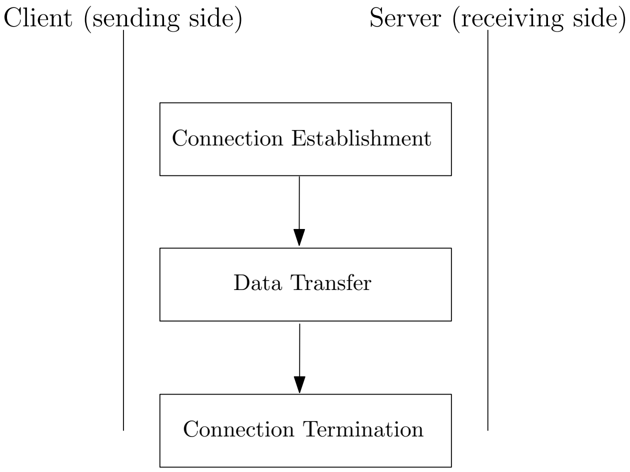

Figure 3: The different phases of bTCP

3.1 Tests

You will have to ensure that the implementation achieves reliable data transfer in the following contexts:

1. ideal network (no packet loss/corruption...)
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

To that end, you are provided with a test framework which has test cases for each of these contexts. 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[5]. 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 within your NetKit VM, 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.

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

4 Submitting your project

Requirements about the delivery of this project:

- Submit via Blackboard (http://blackboard.ru.nl);
- Upload one single .zip archive with your code, tests, and documentation.

Deadline:

Final Project by Friday, 02 June 2017, 20:00. Late submissions are not accepted.

Organization: It is strongly encouraged that you work in pairs. You can also work individually. Groups of more than two members are not allowed.

Marks: You can score a total of 100 points. The points are distributed as follows:

- 10 points for a state machine for bTCP
- 25 points to demonstrate (an efficient implementation of) working reliability
- 25 points to demonstrate (an efficient implementation of) working flow control
- 25 points for tests
- 15 points for report (including clearly-described design process and justifications for choices made during implementation)
- (Up to) 10 BONUS points for extra features implemented. Make sure to document these in your report as well.

Implementation: You must use the latest Python 3.x version (3.6). A Python framework compatible with this version is provided on Blackboard. You are strongly encouraged to build your project on top of this framework. The framework is designed to simplify and streamline your project. You are allowed to change the framework for your learning purposes, but the submitted version must work with an unmodified framework.

We suggest you use versioning. In particular, we recommend Git. In case you are unfamiliar with it, a good presentation on Git can be found on Giso Dal's page at: http://www.cs.ru.nl/~gdal/files/gittutorial.pdf. The faculty is hosting a GitLab server, accessible with your science login, at https://gitlab.science.ru.nl/. For more information on it, see http://wiki.science.ru.nl/cncz/GitLab.