Mitchell Bailey
Department of Electrical and Computer Engineering
EECE.5830 - Network Design: Principles, Protocols, and Applications
9 December 2022

EECE 5830 Project Phase 6 (TCP) Design File

Introduction

Transmission control protocol (TCP) consists of a number of key features. These include connection setup, connection teardown, dynamic window sizing, dynamic timeouts, internet checksums, flow control, and congestion control.

Please note that while I try to show as much code as possible here, the actual implementation of TCP is much more verbose and *tcpnet.py* should likely be viewed as reference in parallel with this document.

File Descriptions

tcpnet.py

Contains the code which defines the protocol implementation.

unittests.py

Extensive tests ranging from typical to edge cases to ensure the robustness of the protocol implementation.

tcptest.py

Called by the graphical user interface (GUI) to send and receive a file numerous times.

gui.py

Contains the code defining the interactable GUI.

Feature Explanation

All of the key TCP features are fully implemented.

TCP Header

```python
def make_hdr(self, seq_num: int, ack_num: int, checksum: int, flags: int = 0b00000000):
    hdr_len = 0 # Header length = Header length field value x 4 bytes
    urg_ptr = 0

    header: bytearray = bytearray(self.SOURCE_PORT.to_bytes(2, 'big') + self.DEST_PORT.to_bytes(2, 'big') +
        seq_num.to_bytes(4, 'big') + ack_num.to_bytes(4, 'big') + hdr_len.to_bytes(1, 'big') + flags.to_bytes(1,
        'big') + self.rx_win_size.to_bytes(2, 'big') + checksum.to_bytes(2, 'big') + urg_ptr.to_bytes(2, 'big') +
        time.time_ns().to_bytes(8, 'big'))

    return header
```

Figure 1

The header follows a standard TCP header format. 8 bytes of the options field has been used to store timestamp information.

Connection Setup

```python
    def _handshake(self, flags = 0):
        if self.handshake_complete:
            return

        if flags == 0b0 and not self.handshake_begun:
            self._handshake_syn()
        elif flags == 0b000010: # SYN
            self.handshake_begun = True
            # print(self.whois, 'Got SYN.', self.last_rxed_seq_num, self.last_rxed_ack_num)
            self._handshake_syn_ack()
        elif flags == 0b010010: # SYN-ACK
            self.handshake_begun = True
            # print(self.whois, 'Got SYN-ACK.', self.last_rxed_seq_num, self.last_rxed_ack_num)
            self._handshake_ack()
        elif flags == 0b010000:
            # print(self.whois, 'Got ACK.', self.last_rxed_seq_num, self.last_rxed_ack_num)
            if self.sent_syn_ack:
                self.handshake_complete = True
                # print(self.whois, 'HANDSHAKE COMPLETED', self.last_rxed_seq_num, self.last_rxed_ack_num)

    # CLIENT STEP 1 (Part 1)
    def _handshake_syn(self):
        # print(self.whois, 'Sending SYN')
        self.sent_syn = True
        self.curr_seq_num = 1
        self.curr_ack_num = 0
        syn_pkt: bytearray = bytearray(self.make_hdr(seq_num=self.curr_seq_num, ack_num=self.curr_ack_num,
            flags=0b000010, checksum=0)) # 2
        self._udt_send(syn_pkt)

    # SERVER STEP 1 (Part 2)
    def _handshake_syn_ack(self):
        # print(self.whois, 'Sending SYN-ACK')
        self.sent_syn_ack = True
        self.curr_seq_num = 2
        self.curr_ack_num = self.last_rxed_seq_num + 1
        self.zero_index = self.curr_ack_num
        syn_ack_pkt: bytearray = bytearray(self.make_hdr(seq_num=self.curr_seq_num, ack_num=self.curr_ack_num,
            flags=0b010010, checksum=0)) # 18
        self._udt_send(syn_ack_pkt)

    # CLIENT STEP 2 (Part 3)
    def _handshake_ack(self):
        # print(self.whois, 'Sending ACK')
        self.sent_ack = True
        self.curr_seq_num = self.last_rxed_ack_num
        self.curr_ack_num = self.last_rxed_seq_num + 1
        self.zero_index = self.curr_seq_num
        ack_pkt: bytearray = bytearray(self.make_hdr(seq_num=self.curr_seq_num, ack_num=self.curr_ack_num,
            flags=0b010000, checksum=0)) # 16
        self.handshake_complete = True
        # print(self.whois, 'HANDSHAKE COMPLETE', self.curr_seq_num, self.curr_ack_num)
        self._udt_send(ack_pkt)
```

Figure 2

Connection setup is conducted via the TCP three-way handshake, facilitated by the functions *_handshake*, *_handshake_syn*, *_handshake_syn_ack*, and *_handshake_ack* (see: Figure 2). When the TCPNet class object is instantiated, the *_tcp_rx_thread* thread, which handles all incoming packets, is started. If it receives a handshake SYN packet, it responds with a SYN-ACK packet and sets its sequence and ack numbers appropriately (see: Figure 3).

```python
if not self.handshake_complete: # Handshake is incomplete.
    if self.send_data is not None: # We are the sender.
        if flags is None: # We listened but heard nothing and are the sender.
            self._handshake(0) # Fire the initial handshake.
        else:
            self._handshake(flags)
    elif flags is not None and flags > 0: # We are the receiver (or not ready to send) and got a flag.
        self._handshake(flags)
```

Figure 3

The original SYN would have been sent if a send data request was made. In our example, the sender now receives a SYN-ACK and responds with an ACK. Once this handshake is completed by both sides, the data transfer can begin.

Note: Handshake failure is unlikely or impossible since on a timeout, the last sent packet is re-sent.

Connection Teardown

```python
def _teardown(self):
    self._teardown_fin()

def _teardown_fin(self):
    # 4-way handshake
    # print(self.whois, 'TEARDOWN HAS BEEN CALLED!')
    self.teardown_initiated = True

    fin_pkt: bytearray = bytearray(self.make_hdr(seq_num=self.curr_seq_num, ack_num=self.curr_ack_num,
        flags=0b000001, checksum=0))
    self._udt_send(fin_pkt)

    # self.done = True

def _teardown_ack(self):
    # print(self.whois, 'TEARDOWN HAS BEEN REQUESTED!')

    ack_pkt: bytearray = bytearray(self.make_hdr(seq_num=self.curr_seq_num, ack_num=self.curr_ack_num,
        flags=0b010000, checksum=0))

    self._udt_send(ack_pkt)

    if not self.teardown_initiated:
        # If teardown initiated, then we must be the initiator and have already sent a FIN.
        fin_pkt: bytearray = bytearray(self.make_hdr(seq_num=self.curr_seq_num, ack_num=self.curr_ack_num,
            flags=0b000001, checksum=0))

        self._udt_send(fin_pkt)

    self.teardown_initiated = True
    # self._shutdown()
    self.done = True
    # self.log_final()
    # print('self.done?', self.done)
```

Figure 4

Connection teardown is handled via TCP’s four-way handshake and is facilitated by the functions *_teardown*, *_teardown_fin*, and *_teardown_ack* (see: Figure 4). A teardown is initiated by the sender when the last byte it wants to send has been acknowledged. It then sends a FIN packet, which is received and responded to with an ACK and another FIN packet. At this point both ends of the connection close (but remain instantiated for another connection if desired).

Dynamic Window Sizing and Congestion Control

```
    def _handle_winsize(self, timedout):
        ack = self.last_rxed_ack_num
        seq = self.curr_seq_num

        # This is where we reset win_size to 1/2 due to packet loss.
        if ack != seq + self.MAX_DATA_SIZE:
            self.consecutive_nacks += 1
        else:
            self.consecutive_nacks = 0
        if timedout:
            self.consecutive_nacks = 0
            self.rx_win_size = 1
        elif self.consecutive_nacks > 3:
            self.consecutive_nacks = 0
            self.rx_win_size = int(self.rx_win_size / 2)
            if self.rx_win_size < 1:
                self.rx_win_size = 1
        else:
            self.rx_win_size += 1
```

Figure 5

Dynamic window sizing is handled by *_handle_winsize*, which is called by the receiving thread each timeout or packet receive (see: Figure 5). This implementation follows the TCP Tahoe protocol, where multiple consecutive NACKs result in a halving of the window size, and timeouts result in resetting the window size to 1. Otherwise, the window size is increased.

Dynamic Timeouts and Flow Control

```
            # Calculates the updated timeout based on RTT.
            if timestamp is not None:
                sample_rtt = (time.time_ns() - timestamp) / 1e9
                self.estimated_rtt = (1 - TCPNet.TYPICAL_RTT) * self.estimated_rtt + TCPNet.TYPICAL_RTT * sample_rtt
                self.dev_rtt = (1 - TCPNet.TYPICAL_BETA) * self.dev_rtt + TCPNet.TYPICAL_BETA * abs(sample_rtt - self.
                    estimated_rtt)
                self.timeout_interval = self.estimated_rtt + 4 * self.dev_rtt
                # print('sample_rtt:', sample_rtt)
                # print('TO: %f s'%(self.timeout_interval))
                self.set_timeout(self.timeout_interval)
```

Figure 6

The timeout is constantly recalculated based on the round-trip-time (RTT) obtained by subtracting now from the timestamp of the last received packet. An estimated RTT is then calculated using exponential averaging, and the timeout is calculated by adding four standard deviations to the RTT.

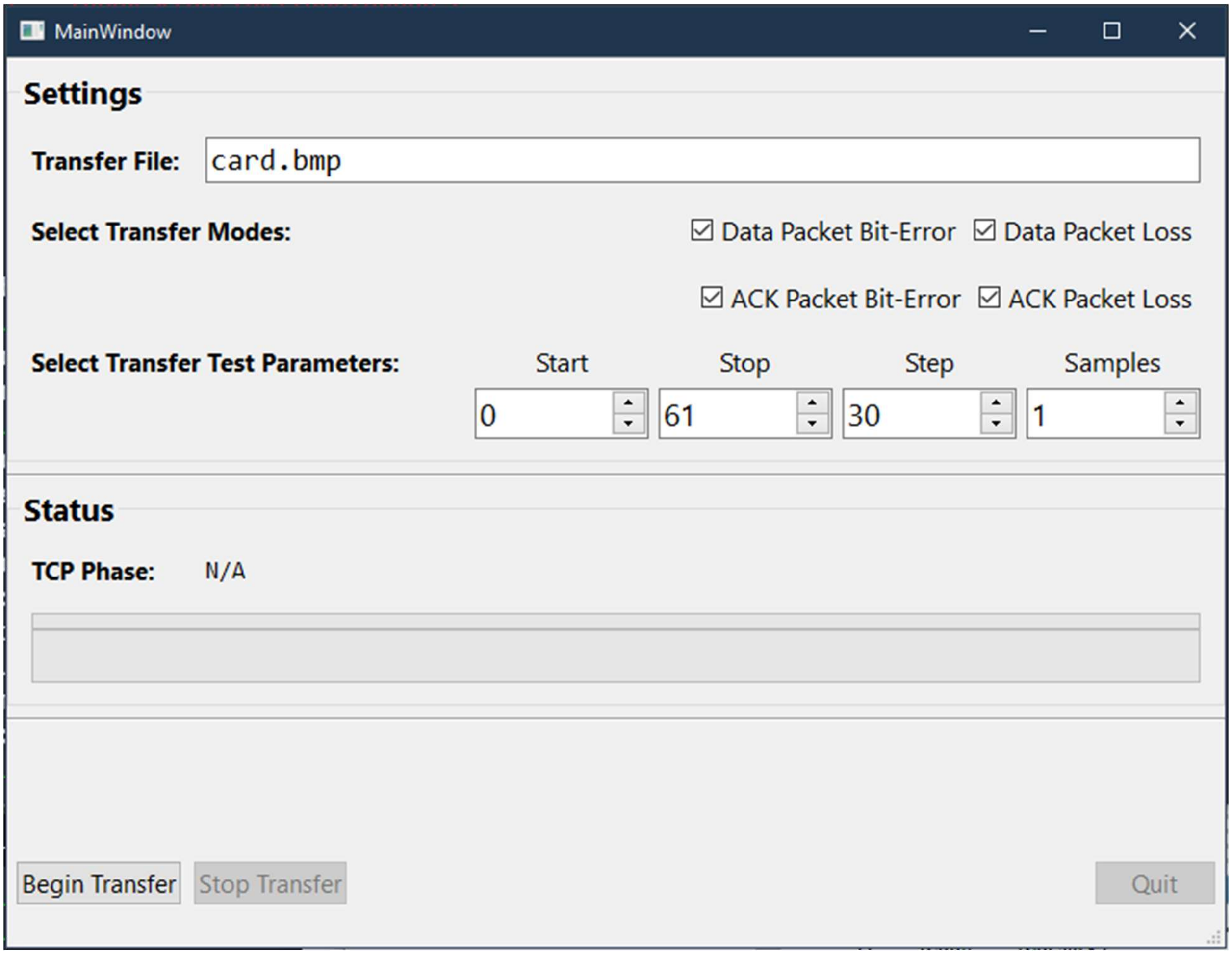
Internet Checksums

```python
    #Define bit16sum() function: Computes the packet checksum based on the checksum algorithm learned in class.
    def bit16sum(self, data)->bytearray:
        checksum = 0 #Initialize checksum variable, set equal to zero
        for i, byte in enumerate(data): #For pyte in data
            if (i % 2 == 0): #If i modulus 2 == 0, then sll data by eight bits
                bit16 = data[i] << 8
                if (i+1 < len(data)): #If the next byte is less than the length of the data, then compute addition w/
                    carryover
                    bit16 = (data[i] << 8) | (data[i+1])
            checksum = (checksum + bit16) & 0xFFFF #Compute 1s complement (addition result AND'd with 0xFFFF mask) to
                yield final checksum
        return checksum.to_bytes(2, 'big')
```

Figure 7

A standard 16-bit checksum is calculated.

Graphical User Interface



The GUI allows the user to begin a file transfer and specify the parameters of the transfer. Two progress bars are present, one shows how far along the current file is while the second one shows how complete the entire process is.

Charts and Analysis

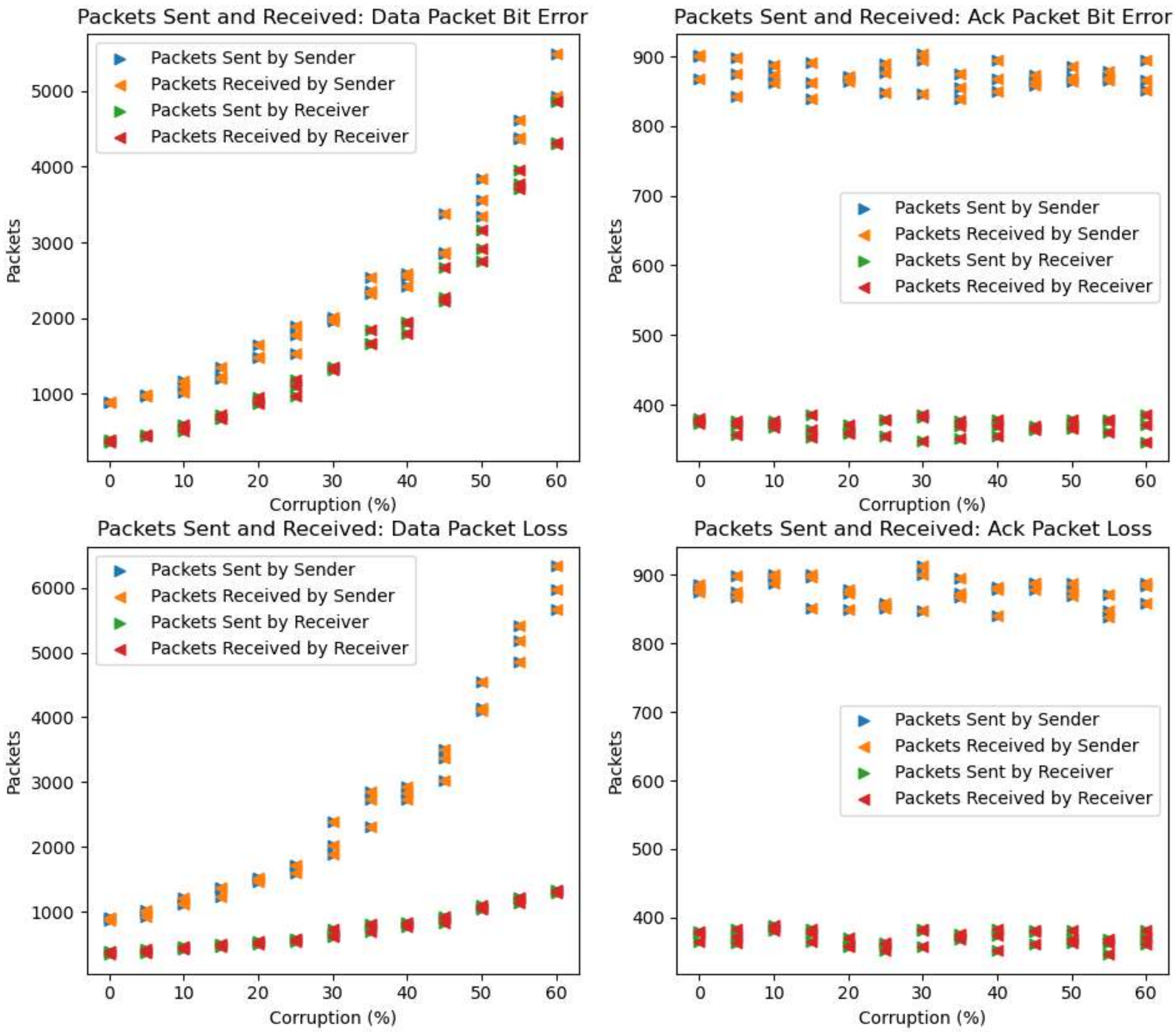


Figure 8

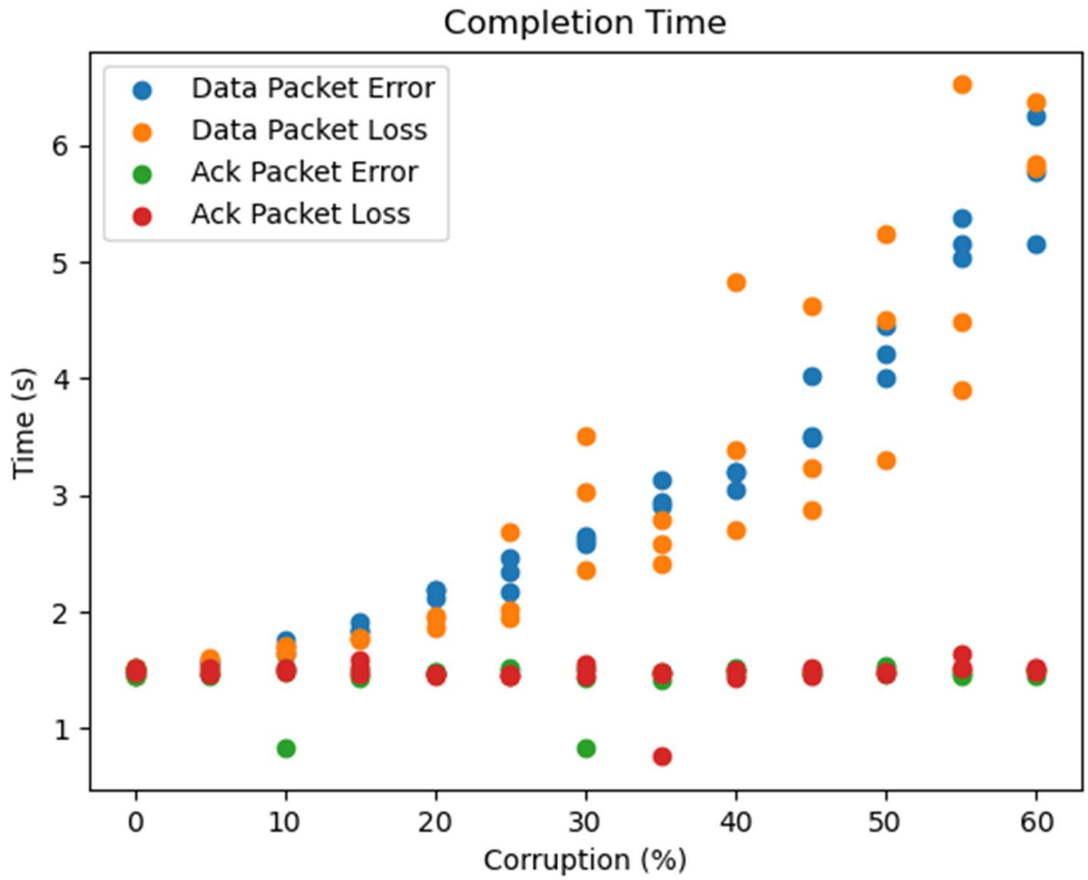


Figure 9

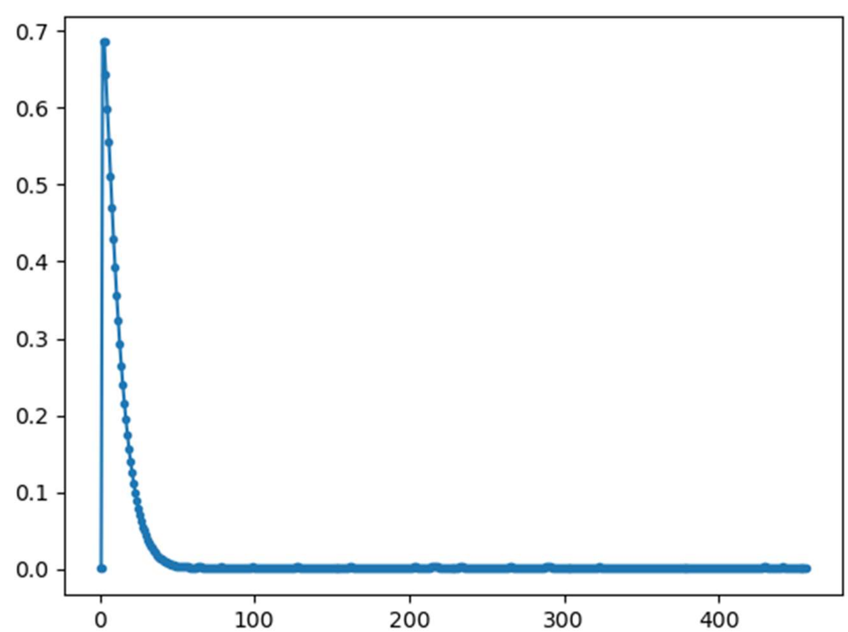


Figure 10

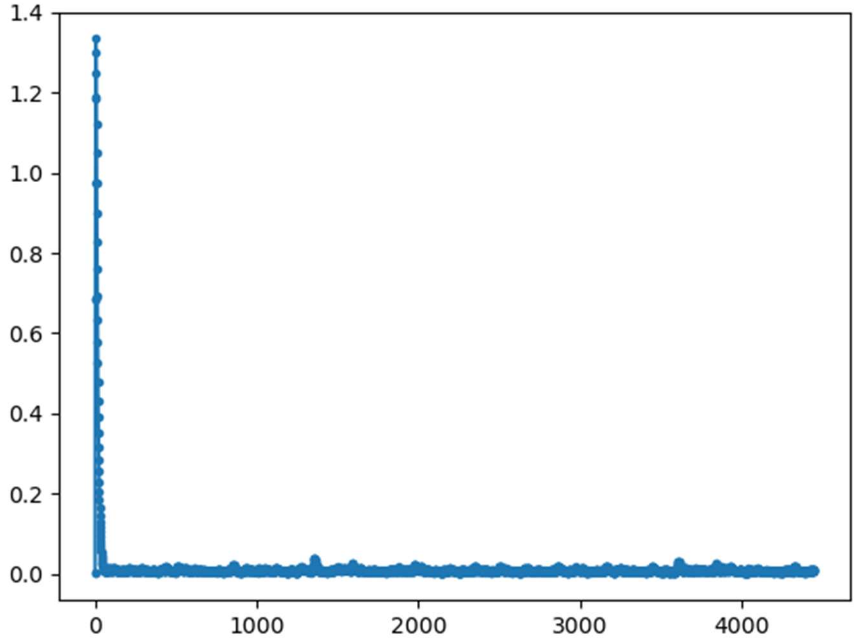


Figure 11

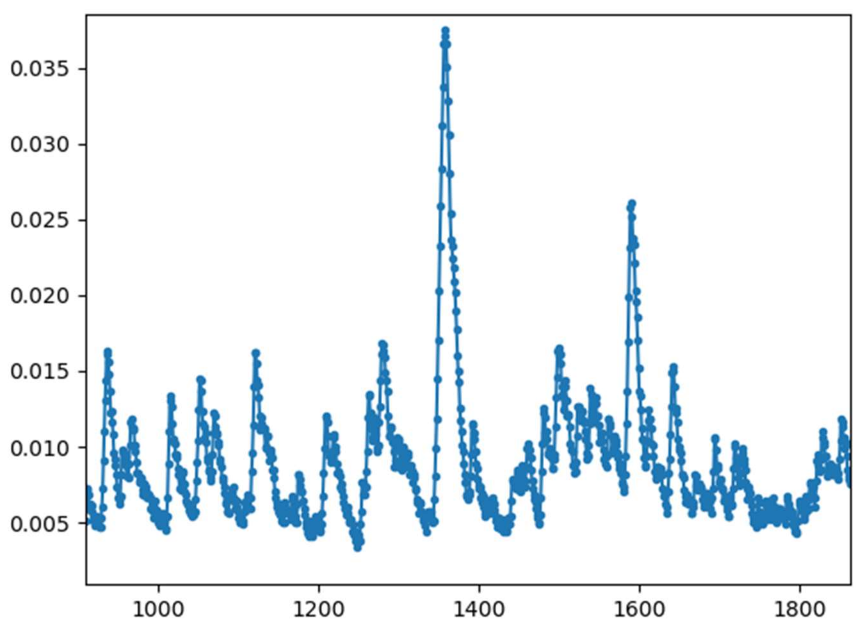


Figure 12

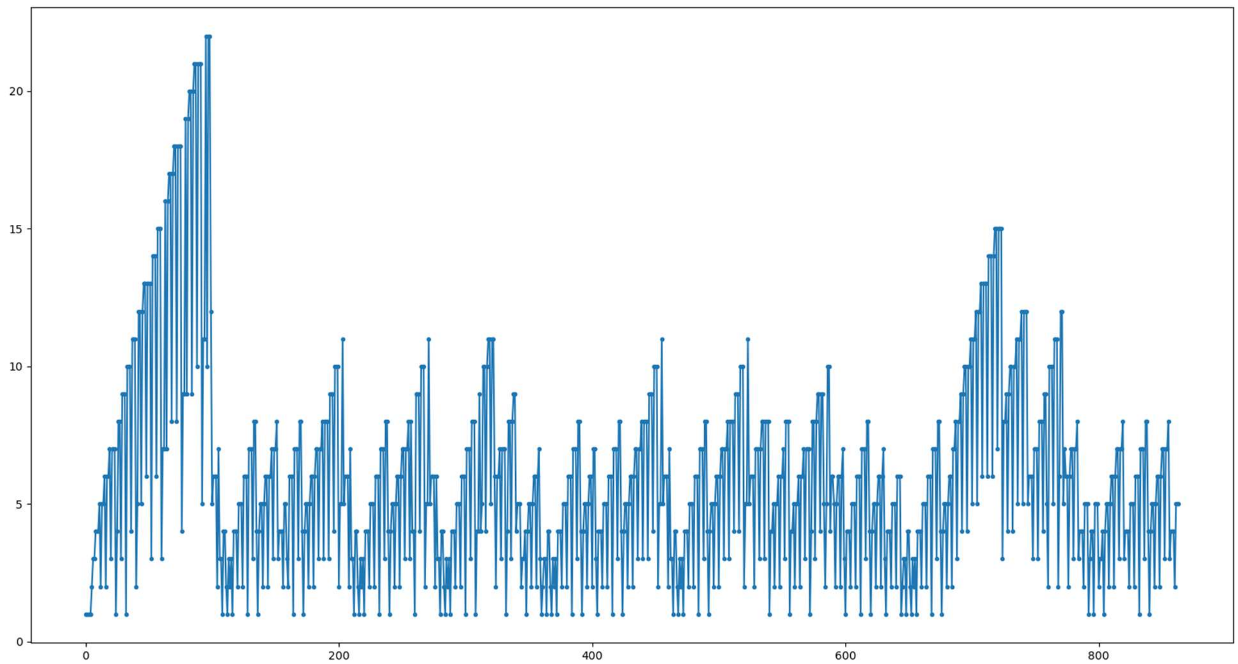


Figure 13

Figure 8 shows how many packets are sent and received for each type of corruption. Note that the number of sent packets includes dropped packets, since they appear to be sent from the sender's perspective.

Figure 9 Completion Time shows the time for completion of each round of file transfer. Three samples have been performed for each corruption-corruption type combination. The protocol is mostly impervious to acknowledgement losses since they are cumulative, and extras end up being sent anyway. Data packet errors and losses cause similar amounts of delay, indicating that the calculated timeouts are likely ideal. The high variability in transfer time during data packet loss corruption is notable.

Figures 10, 11, and 12 show the timeout over time. Recording of the timeout value begins when it is uninitialized (0), allowing us to see the initial estimate of around 1 second. It then finds equilibrium at around one-hundredth of one second. Figure 10 shows a typical connection, while Figure 11 shows a 60% corrupted line. Figure 12 offers a zoomed-in view of Figure 11 to see in detail how the timeout reacts to changes in RTT.

Figure 13 shows the window size over time for one file transfer. Note that the window size data for this graph is sampled more than once per window size adjustment, resulting in consecutively equivalent values.

References

[1] J. Kurose, “3.4.1 Building a Reliable Data Transfer Protocol,” in *Computing Networking: A Top-Down Approach*, 7th ed., K. Ross, Ed. Pearson.

[2] “Qt for Python (PyQt5),” *Qt for Python*. [Online]. Available: https://doc.qt.io/qtforpython/. [Accessed: 04-Dec-2022].

[3] “Socket - low-level networking interface,” *Python 3.10.7 documentation*. [Online]. Available: https://docs.python.org/3/library/socket.html. [Accessed: 29-Sep-2022].

[4] “Random - generate pseudo-random numbers,” *random - Generate pseudo-random numbers - Python 3.11.0 documentation*. [Online]. Available: https://docs.python.org/3/library/random.html. [Accessed: 30-Oct-2022].

[5] “Time - Time Access and conversions,” *time - Time access and conversions - Python 3.11.0 documentation*. [Online]. Available: https://docs.python.org/3/library/time.html. [Accessed: 30-Oct-2022].

[6] “Collections - Container Datatypes,” *collections - Container datatypes - Python 3.11.0 documentation*. [Online]. Available: https://docs.python.org/3/library/collections.html. [Accessed: 04-Dec-2022].

[7] “Sys - system-specific parameters and functions,” sys - System-specific parameters and functions - Python 3.10.7 documentation. [Online]. Available: https://docs.python.org/3/library/sys.html. [Accessed: 29-Sep-2022].

[8] “Os.path - common pathname manipulations,” *os.path - Common pathname manipulations - Python 3.11.0 documentation*. [Online]. Available: https://docs.python.org/3/library/os.path.html. [Accessed: 30-Oct-2022].

[9] “Matplotlib API Reference,” *API Reference - Matplotlib 3.6.0 documentation*. [Online]. Available: https://matplotlib.org/stable/api/index.html. [Accessed: 30-Oct-2022].

[10] “Signal - set handlers for asynchronous events,” *signal - Set handlers for asynchronous events - Python 3.10.7 documentation*. [Online]. Available: https://docs.python.org/3/library/signal.html. [Accessed: 29-Sep-2022].