Features and Documentation

This section briefly discusses some of the TCP related features that I've implemented, and how various functions work.

Overall Design

TCP Client basically functions as follows:

1. Given a set of command line arguments in `args`
2. Construct a `TCP_CLIENT` instance which constructs a `UDP_CLIENT` under the hood, and intializes parameters such as `port` number to send to, `port` number to receive from, etc
3. Start a thread for doing blocking receive in a loop. In particular, it runs the following method

```python
def __receive(client:TCP_CLIENT):
    while client.state == TCP_CLIENT.ESTABLISHED:
        # check again
        client.rcv_lock.acquire()
        if client.state == TCP_CLIENT.ESTABLISHED:
            received = client.receive() # blocking
            logging.info(f'state {client.state}')
            if client.state != TCP_CLIENT.ESTABLISHED:
                client.update_fin_packets(received)
            logging.info(f'thread recevied: {received}')

        client.rcv_lock.release()
        time.sleep(0.1)
    return
```

here, a lock is required when calling receive, because during the scenario when it starts `FIN` sequences to end the connection, the **main thread** who is constantly sending packets will send a `FIN` packet and **wait for** `ACK` and etc. Without this lock, there is a chance that the `ACK` packet will be stolen by this thread before the main thread calls `rcv` in the following coed (for instance)

```python
def __wait_server_ack(self, fin_packet:Packet):
    logging.debug(f'at __wait_server_ack')
    fin_seq = fin_packet.header.seq_num
    self.__state = TCP_CLIENT.FIN_WAIT_1

    # find FIN ACK packet
    check_threading = True
    while self.__state == TCP_CLIENT.FIN_WAIT_1:
        # wait
        time.sleep(1)
        self.rcv_lock.acquire()
        """
        Obtain LOCK here so that:
        Case 1. the other thread in tcpclient.py obtained the lock and went rcv.
            - self.__thread_fin_packets is 100% updated. This works
        Case 2. I got the lock
            - the other thread obviously went rcv. Also works
        """
        # check list first
        if check_threading:
            for packet in self.__thread_fin_packets:
                # check if is the ACK for fin
                logging.debug(f'fin ack wait: checking {packet.header}')
                if packet.header.ack_num >= fin_seq + 1 and packet.header.is_ack():
                    self.__state = TCP_CLIENT.FIN_WAIT_2
                    self.__thread_fin_packets.remove(packet)
                    self.__window = []
```

```python
                    self.rcv_lock.release()
                    return packet
                check_threading = False
            # receive
            packet = self.receive()
            self.rcv_lock.release()

            logging.debug(f'fin ack wait: {packet.header}')
            if packet.header.ack_num == fin_seq + 1 and packet.header.is_ack():
                self.__state = TCP_CLIENT.FIN_WAIT_2
                return packet
        return
```

4. Start a loop that constantly reads `globals.MSS` (which is set to 512) bytes from the file, and attempt to send:

```python
# inside the send_file method
with open(args.file, 'rb') as openfile:
    receiv_thread.start()
    data = openfile.read(globals.MSS)
    while data != b'':
        ret = client.send(data)
        while ret == -1:
            time.sleep(1)
            ret = client.send(data)
        data = openfile.read(globals.MSS)
    client.terminate()
    receiv_thread.join()
```

   where the `TCP_CLIENT` could reject packet by returning `0` from `client.send(data)`, which will happen when the window size is full. (To see how I implemeneted Pipelined Sending, checkout the next section.)

5. Once all files contents are sent, terminate the connection by doing the `FIN` handshakes inside the method `terminate`, which will be gone into details in the next section as well.

TCP Receiver/Server does the following:

1. Given a set of command line arguements in `args`
2. Construct a `TCP_SERVER` instance which constructs a `UDP_SERVER` under the hood, and intializes parameters such as `port` number to send to, `port` number to receive from, etc
3. Start the server with `start()`, which will constantly attempt to `service_client` by receiving from the underlying buffer:

```python
def start(self, args):
    """Start the server
    Start to listen and accept clients.
    Reset when client intiates FIN requests and completed the handshake

    Args:
        args (namespace): command line arguments for the program, e.g. which file to write to
    """
    server = self._socket
    server.bind(self._serveraddress)
    # other application related init
    init(args)
    self.__state = TCP_SERVER.LISTEN
    print("The server is ready to receive")

    while True:
        try:
            self.__state = TCP_SERVER.ESTABLISHED
            service_client(self, args)
            self.__state = TCP_SERVER.LISTEN
```

```python
        except Exception as err:
            print(err)
            pass
    return
```

where the `service_client` function basically is the entry point for all the receiving and acking:

```python
def service_client(server:TCP_SERVER, args):
    """Specifies what to do when received something from client

    Essentially does 1) receive 2) check packet received
    3) write to file 4) send ACK

    Args:
        server (TCP_SERVER): running instance of TCP_SERVER
        args (namespace): command line arguments
    """
    # receive packet
    received, client_address = server.receive()
    logging.info(f"[LOG] serviced {client_address}")
    logging.info(f"{received or 'Discarded or Residual'}")

    if received is not None:
        # write to file
        to_file(received, dst=args.file)

    # send ACK
    if server.state == TCP_SERVER.ESTABLISHED:
        server.send('')
    return
```

which basically a) receives from the buffer, b) write to file if non-corrupt (not-None) packet received, c) send `ACK`

4. When a client initiated a `FIN` sequence, the `server.receive()` will detect a `FIN` packet and basically does an `ACK` back and sends a `FIN`. Implemnetation details of this will be gone over in the next section.

   Once this "handshake" is done, it will reset its current status of `seq_num` and `ack_num` and etc, such that it can be ready to service the next "new client".

5. This `service_client` mentioned above will continue running forever, and should work across multiple runs of `tcpsender.py` without the need to restarting the server everytime.

TCP Related Features

- **TCP Packet**

  This is implemented as the following class

```python

class Packet(util.Comparable):
    """TCP Packet

    This abstraction gives you a human-readable packet on the "surface",
    but during transmission, it will be "serialized" by struct.pack to become
    bytes.
    """

    def __init__(self, header:TCPHeader, payload:str) -> None:
        """Construct a packet from header and payload

        Args:
            header (TCPHeader): a constructed TCP header
            payload (str or bytes): payload
        """
        self.__header = header
        self.__payload = payload

```

```
    @property
    def header(self):
        return self.__header

    @property
    def payload(self):
        return self.__payload

    @header.setter
    def header(self, value):
        self.__header = value

    @payload.setter
    def payload(self, value):
        self.__payload = value

    def compute_checksum(self):
        checksum = self.__compute_checksum()
        checksum = ~checksum & 0xffff # 1s complement and mask
        self.__header.set_checksum(checksum)
        return

    def __compute_checksum(self):
        """Computes the 1s complement treating self.__header=0

        Returns:
            [int]: 1s complement of checksummed packet
        """
        prev_checksum = self.__header.checksum
        self.__header.set_checksum(value=0)
        # computes checksum without header
        all_bytes = serialize(self)
        checksum = 0
        for i in range(0, len(all_bytes), 2):
            if i + 1 == len(all_bytes):
                checksum += all_bytes[i]
                break
            checksum += (all_bytes[i] << 8 + all_bytes[i+1])
        # reset
        self.__header.set_checksum(prev_checksum)
        return checksum

    def is_corrupt(self):
        current_checksum = self.__compute_checksum()
        logging.debug(f'checksum result {current_checksum & self.__header.checksum}')
        return current_checksum & self.__header.checksum != 0

    def __str__(self):
        content = f"""
        ---
        [HEADER]: {self.__header}
        [Payload]: {self.__payload}
        ---"""
        return content
```

basically it contains human-readble formatting of a Packet, which is very useful for logging and debugging. When actually sending/receiving the packet, it will be serialized into bytes by the following two methods:

```
def serialize(packet:Packet):
    """Converts the Human-readable Packet to bytes

    Args:
        packet (Packet): Packet abstraction

    Returns:
        bytes: actual bytes of the packet
        (i.e. 20 bytes header + up to 512 byte payload)
    """
```

```python
    line_1 = struct.pack('HH', packet.header.src_port, packet.header.dst_port)
    line_2 = struct.pack('I', packet.header.seq_num)
    line_3 = struct.pack('I', packet.header.ack_num)
    # convert flag
    flag = packet.header.flags
    flag_map = int(f"{flag.ack}{flag.cwr}{flag.ece}{flag.fin}{flag.syn}", 2)
    line_4 = struct.pack('BBH', packet.header.header_len, flag_map, packet.header.rcvwd)
    line_5 = struct.pack('HH', packet.header.checksum, 0)
    # final
    final = line_1 + line_2 + line_3 + line_4 + line_5
    if len(packet.payload) != 0:
        final += packet.payload
    return final

def deserialize(packet):
    """Converts bytes to a HUman-readable Packet

    Args:
        packet (bytes): network transmitted bytes

    Returns:
        Packet: human-readable Packet
    """
    total_size = len(packet)
    src_port, dst_port, \
        seq_num, ack_num, \
            header_len, flags, rcvwd, \
                checksum, urg, \
                    data = struct.unpack(f'HHIIBBHHH{total_size-20}s', packet)
    flags = format(flags, '#07b')
    header = TCPHeader(
        src_port=src_port,
        dst_port=dst_port,
        seq_num=seq_num,
        ack_num=ack_num,
        _flags=Flags(
            cwr=int(flags[3]),
            ece=int(flags[4]),
            ack=int(flags[2]),
            syn=int(flags[6]),
            fin=int(flags[5])),
        rcvwd=rcvwd)
    header.set_checksum(checksum)
    packet = Packet(header, data)
    return packet
```

the function `serialize` converts a `Packet` instance to bytes using `stuct.pack` , and the other one unpacks the bytes into a `Packet` by `struct.unpack` .

The serialize/deserialize happens **only right before** the `send_to` and **right after** the `recv_from` of the UDP channel, which means the entire TCP code can treat all data as a `Packet` (which makes the program easier):

```python
# insude UDP_SERVER
def send_packet(self, packet:Packet, client_address):
    socket = self.__socket
    packet = structure.packet.serialize(packet)
    ret = socket.sendto(packet, client_address)
    return ret
def receive_packet(self):
    raw_packet, _ = self.__socket.recvfrom(self.__buffersize)
    return structure.packet.deserialize(raw_packet)
```

and similarly in the `UDP_CLIENT` :

```python
def send_packet(self, packet:Packet, client_address):
        socket = self.__socket
```

```python
        packet = structure.packet.serialize(packet)
        ret = socket.sendto(packet, client_address)
        return ret

def receive_packet(self):
    server = self.__socket
    raw_packet, client_address = server.recvfrom(self.__buffersize)
    try:
        # e.g. corruption
        packet = structure.packet.deserialize(raw_packet)
        logging.debug(f'rcvd {packet}')
    except:
        packet = None
    return packet, client_address
```

- **Timer** (used for timeouts)
  This in TCP is basically based on the `threading.Timer` module:

```python
class TCPTimer(object):
"""TCP timer implementation. Used for multithreading mainly
"""

def __init__(self, interval: float, function: Callable[..., Any], *args, **kwargs) -> None:
    """TCP Timer implementation. Essentially triggers @function when timedout.

    Args:
        interval (float): TimeoutInterval
        function (Callable[..., Any]): function to call when timedout
    """
    self.__interval = interval
    self.__function = function
    self.__args = args
    self.__kwargs = kwargs
    self.__timer = None

@property
def interval(self):
    return self.__interval

def start(self):
    if self.__timer is not None and self.__timer.is_alive():
        logging.error('timer already running')
        return
    self.__timer = Timer(self.__interval, self.__function, args=self.__args, kwargs=self.__kwargs)
    self.__timer.start()
    logging.debug('timer started')
    return

def is_alive(self):
    if self.__timer is None:
        return False
    return self.__timer.is_alive()

def cancel(self):
    if self.__timer is None:
        return
    self.__timer.cancel()
    return

def restart(self, new_interval=None):
    if self.__timer is not None and self.__timer.is_alive():
        self.__timer.cancel()

    # calling self.start() causes problem as the thread might NOT be finished

    # (e.g. function still executing)

```

```
        self.__interval = new_interval or self.__interval
        self.__timer = Timer(self.__interval, self.__function, args=self.__args, kwargs=self.__kwargs)
        self.__timer.start()
        return
```

where whenever you start a `TCPTimer`, it basically starts another thread using `Timer`. When that `Timer` object timed out, it will essentially call the `self.__function` (which will be the `retransmit` function). Therefore, in this way, retranmissions of packets will not interfere with the main program of sending/receiving.

Inside the TCP client, some of the usages look like:

```
class TCP_CLIENT(UDP_CLIENT):
    def __init__(self, udpl_ip, udpl_port, window_size, ack_lstn_port):
        """TCP reliable sender implementation

        Args:
            udpl_ip (str): udpl IP address to send to (proxy address)
            udpl_port (int): udpl port address to send to
            window_size (int): number of packets allowed in current window
            ack_lstn_port (int): port number of receiving ACK from server
        """
        super().__init__(udpl_ip, udpl_port, ack_lstn_port)
        self.__seq_num = 0
        self.__ack_num = 0 # assumes both sides start with seq=0
        self.__timer = timer.TCPTimer(TCP_CLIENT.INIT_TIMEOUT_INTERVAL, self.retransmit)
        # other initialization omitted

    # other class methods omitted
    def __post_send(self, packet:Packet):
        logging.debug("at __post_send")
        # some code omitted

        # 3. check if timer is running
        self.__rtt_sampling.double_interval(enabled=False, restore=False)
        if not self.__timer.is_alive():
            self.__timer.restart(new_interval=self.__rtt_sampling.get_interval())
        # some code omitted
    return
```

where notice that when there is a timeout, it will go doubling the interval by `self.__rtt_sampling.double_interval(enabled=False, restore=False)` and then the timer will restart (if not running) with an interval of `self.__timer.restart(new_interval=self.__rtt_sampling.get_interval())`.

To see how the RTT Sampler works, see the next bullet point on RTT Sampler.

- **RTT Sampler**

  Again, this is implemented as an object so that detailed updating mechanism can be hidden away from the main logics of TCP sending. In details, it is implemented as follows:

```
class RTTSampler(object):
    """TCP RTT Sampler

    This class essentially allows you to input a measured RTT and updates
    TimeoutInterval internally. So the next time, you can get the computed
    TimeoutInterval by :func:self.get_interval
    """
    def __init__(self, init_interval) -> None:
        super().__init__()
        self.__timeout_interval = init_interval

        # used for estimation
        self.__estimated_rtt = init_interval
        self.__alpha = 0.125
        self.__dev_rtt = 0
        self.__beta = 0.25
        self.__gamma = 2
```

```python

        # used for doubling timeout
        self.__within_timeout = False
        pass

    def double_interval(self, enabled=True, restore=True):
        self.__within_timeout = enabled
        if enabled is False and restore: # when sending new packets, do not restore
            self.__timeout_interval = round(self.__estimated_rtt + self.__gamma * self.__dev_rtt, 3)
        return

    def update_interval(self, sample_rtt):
        # we received something, switch back to using normal timeout
        self.__within_timeout = False

        alpha = self.__alpha
        beta = self.__beta
        self.__estimated_rtt = (1-alpha) * self.__estimated_rtt + alpha * sample_rtt
        self.__dev_rtt = (1-beta) * self.__dev_rtt + beta * (abs(sample_rtt - self.__estimated_rtt))
        self.__timeout_interval = round(self.__estimated_rtt + self.__gamma * self.__dev_rtt, 3)
        logging.debug(f"""
        sample with {sample_rtt}
        new self.__estimated_rtt {self.__estimated_rtt}
        new self.__dev_rtt {self.__dev_rtt}
        rounded new timeout interval {self.__timeout_interval}
        """)
        return

    def get_interval(self):
        if self.__within_timeout:
            self.__timeout_interval *= 2
        return self.__timeout_interval
```

where the most important method is basically the `update_interval`. This basically performs the calculation of the new timeout interval, and it is done everytime when `TCP_CLIENT` received an `ACK`, and there is a packet that we are actively tracking:

```python
# inside TCP_CLIENT
def __post_recv(self, packet:Packet):
    # some code omitted here
    # 1. update window, received ACK
    if packet.header.ack_num > self.__send_base:
        # 2. new ACK received
        # some code omitted here

        # 3. update RTT
        start_time = self.__waiting_packets.get(packet.header.ack_num)
        if start_time is not None: # not retransmitted
            end_time = time.time()
            self.__rtt_sampling.update_interval(end_time - start_time)
            self.__waiting_packets.pop(packet.header.ack_num, None)
        return
    # some code omitted here
    return
```

where basically it will check the `self.__waiting_packets` dictionary, which is added and updated by the `send` and `retransmit` methods, and see if we should sample this RTT or not.

- **Pipelined Sending** (window)

  This is now simple due to the multithreading receive. In essense, whenever we send something, it a) check the current window size to see if it is full, b) if not full, send c) perform `__post_send` actions such as updating the next sequecen number, adding packet into window, and start a timing track for RTT sampler

```python
def send(self, payload:str):
    """Reliably send a packet with payload @payload

    Args:
        payload (str or bytes): payload

    Returns:
        int: success=0
    """
    # 0. consult window
    if len(self.__window) == self.__window_size:
        return -1
    # 1. construct packet
    _, src_port = self.get_info()
    header = TCPHeader(
        src_port=src_port,
        dst_port=self.dst_addr[1],
        seq_num=self.__seq_num,
        ack_num=self.__ack_num,
        _flags=Flags(cwr=0, ece=0, ack=0, syn=0,fin=0),
        rcvwd=10)
    packet = Packet(header, payload)
    packet.compute_checksum()

    # 2. send packet
    self.send_packet(packet)

    # 3. update seq_num, etc
    self.__post_send(packet)
    return 0
```

(notice that we were able to construct a `Packet` in an entirely human-readable form because all the byte conversion is done secretly in the end by `serialize/deserialize` !)

- **Checksum**

  To prevent against corrupted packet, the internet checksum basically performs the summing over data and check against the `checksum` field of the packet.

  On the sender side, a checksum is computed everytime we constructed a packet

```python
# code snipped inside send() of TCP_CLIENT
_, src_port = self.get_info()
header = TCPHeader(
    src_port=src_port,
    dst_port=self.dst_addr[1],
    seq_num=self.__seq_num,
    ack_num=self.__ack_num,
    _flags=Flags(cwr=0, ece=0, ack=0, syn=0,fin=0),
    rcvwd=10)
packet = Packet(header, payload)
packet.compute_checksum()
```

where the `compute_checksum` basically does

```python
def compute_checksum(self):
    checksum = self.__compute_checksum()
    checksum = ~checksum & 0xffff # 1s complement and mask
    self.__header.set_checksum(checksum)
    return
```

where the `__compute_checksum()` basically computes the raw sum without 1s complement:

```python
def __compute_checksum(self):
    """Computes the 1s complement treating self.__header=0

```

```python
    Returns:
        [int]: 1s complement of checksummed packet
    """
    prev_checksum = self.__header.checksum
    self.__header.set_checksum(value=0)
    # computes checksum without header
    all_bytes = serialize(self)
    checksum = 0
    for i in range(0, len(all_bytes), 2):
        if i + 1 == len(all_bytes):
            checksum += all_bytes[i]
            break
        checksum += (all_bytes[i] << 8 + all_bytes[i+1])
    # reset
    self.__header.set_checksum(prev_checksum)
    return checksum
```

On the receiver/server side, whenever a packet is received, it can invoke the `is_corrupt` method to check the checksum easily:

```python
def is_corrupt(self):
    current_checksum = self.__compute_checksum()
    logging.debug(f'checksum result {current_checksum & self.__header.checksum}')
    return current_checksum & self.__header.checksum != 0
```

- **FIN**

  When a client finishes sending all pieces of a file, it starts calling `terminate` function, which basically starts performing the `FIN` "handshake":

```python
def terminate(self):
    """Terminate the connection

    After the FIN handshake, close the underlying UDP socket.

    Returns:
        None: None
    """
    # 0. wait for all other retransmission to be done
    while len(self.__window) > 0:
        # the other thread will timeout and retransmit
        time.sleep(1)

    # change state so that the other thread will not receive packets
    self.__state = TCP_CLIENT.BEGIN_CLOSE

    # 1. construct FIN packet
    _, src_port = self.get_info()
    header = TCPHeader(
        src_port=src_port,
        dst_port=self.dst_addr[1],
        seq_num=self.__seq_num,
        ack_num=self.__ack_num,
        _flags=Flags(cwr=0, ece=0, ack=0, syn=0, fin=1),
        rcvwd=10)
    packet = Packet(header, b'')
    packet.compute_checksum()
    self.__fin_start_seq = self.__seq_num

    # 2. send packet
    self.send_packet(packet)

    # 3. start timers
    self.__post_send(packet)

    # 4. wait for acks and etc
```

```
        self.__post_fin(packet)
    return super().terminate()
```

where `super().terminate()` terminates the entire socket, so it is the end of transmission.

And similarly, the server will detect the `FIN` handshake during a `receive()` call:

```
def receive(self):
    """Blocking receive a packet

    Returns:
        (Packet, tuple): returns (Packet, client_address) if packet is not corrupt.
        Else, returns (None, client_address)
    """
    # 1. receive packet
    packet, client_address = self.receive_packet()
    # 2. check if packet is corrupt
    if packet is not None and not packet.is_corrupt():
        # 3. if not, update ack_num
        packet = self.__post_recv(packet)
    else:
        packet = None
    return packet, client_address
```

and inside `__post_recv()`, it does:

```
def __post_recv(self, packet:Packet):
    self.__ack_num = self.__next_ack(packet) # position of next byte
    if packet.header.is_fin() and packet.header.seq_num + 1 >= self.__ack_num:
        logging.info('closing connection')
        logging.info(packet)
        packet = self.close_connection(packet)
        logging.info('connection closed')
    return packet
```