CS168 Fall 2018 Project 2 – Transport

Due: December 3, 2018 11:59 PM

Contents

1 Introduction

The goal of this project is to implement a Socket that implements a TCP protocol similar to those you can find in the Internet. A socket is an abstraction between the application layer and the transport layer that allows an application to easily use the underlying transport protocol (TCP in this case). While sockets are usually implemented by the operating system, your socket will be a user space implementation written in Python. We will provide a network simulator and a Python TCP/IP stack, minus parts that you must implement (which is the core of the actual TCP protocol). In this project, you won't be implementing applications that use this Socket. Instead, you will be implementing core parts of the protocol, and the tests we provide will act as applications. These applications will use the Socket class you write and expect the behavior defined in this specification.

This is not a short project. Please start as soon as possible to give yourself ample time to complete it.

2 Background

2.1 Sockets

It would be difficult for application developers to think in terms of packets. Following the trend of using abstractions when dealing with complexity, it is much easier to use a logical pipe that connects a sender and a receiver. In such abstraction, a sender calls a function to send data without having to worry about the details of how that message is actually sent and delivered. This abstraction is called a Socket and is controlled by its API.

Sockets abstract away establishing a connection, and sending and receiving data. Further, a TCP socket provides reliable, ordered, and error-checked delivery of a stream of bytes. Usually, each connection is composed of two sockets: the *local* socket, and the *remote* socket. For example, the local socket might be created and utilized by your web browser, and the remote socket could be created and managed by a web server. In typical usage, one socket *actively connects* (this would be your web browser), and the other *passively connects* (this would be the web server). In a typical scenario, the server creates a socket and then calls `bind` and `listen` functions to create a socket waiting for a connection. Clients can establish TCP connections to the server by calling `connect`, which the server can `accept`. Underlying this sequence of function calls is the TCP 3-way handshake you have learned about in lecture. Once the connection is established, either side can send data with `send` and the other side can receive data with `recv`.

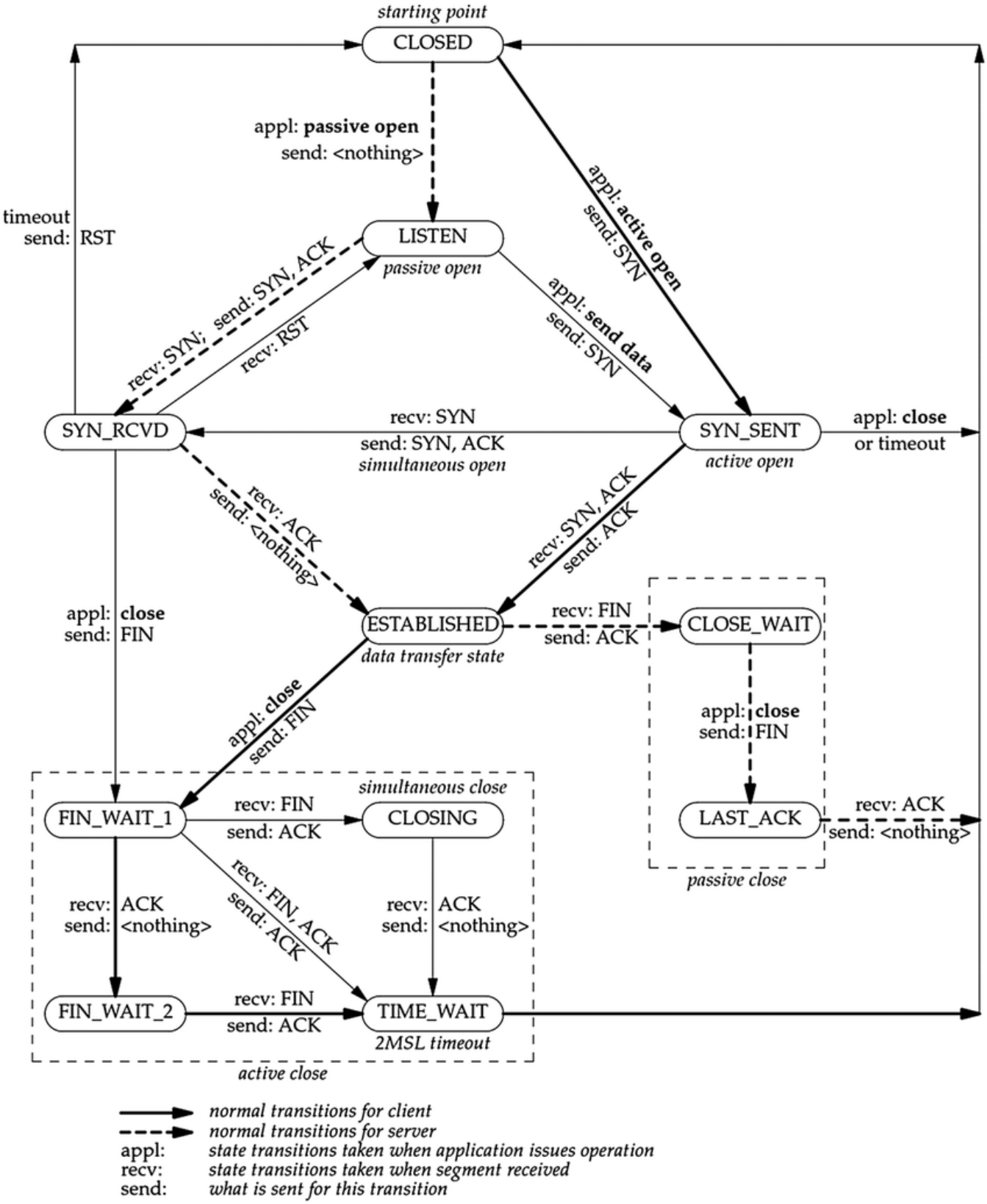
2.2 TCP

TCP (Transmission Control Protocol) is a Layer 4 protocol that provides a byte stream abstraction. The stream of bytes are broken into packets (or *segments* in TCP lingo) and delivers those packets. It provides multiplexing and demultiplexing, reliability, connection orientated communication, and flow and congestion control.

A connection can progress through a series of states during its lifetime. The possible states are listed in the diagram below. For this project you only need to implement a subset of TCP for the client and not the server. You will **not** have to implement the LISTEN or SYN RECEIVED state transitions, and you'll be given a large fraction of the rest of state transitions. The entire state transition diagram is detailed below for reference. Further, the TCP stack you will implement will **only** support the ACK, SYN and FIN control

bits.

Further, we will also **not** implement congestion control. Specifically, we will not do fast retransmit (retransmit on 3 dup ACKs), fast recovery, slow start, or congestion avoidance. You are welcome to learn about this on your own and congestion control has been the focus of much great debate and research. However, due to its complex and gritty details, we will skip it for this project.


starting point
CLOSED
appl: passive open
send: <nothing>
appl: active open
send: SYN
timeout
send: RST
LISTEN
passive open
recv: SYN; send: SYN, ACK
recv: RST
appl: send data
send: SYN
SYN_RCVD
recv: SYN
send: SYN, ACK
simultaneous open
SYN_SENT
active open
appl: close
or timeout
recv: ACK
send: <nothing>
recv: SYN, ACK
send: ACK
appl: close
send: FIN
ESTABLISHED
data transfer state
recv: FIN
send: ACK
CLOSE_WAIT
appl: close
send: FIN
LAST_ACK
recv: ACK
send: <nothing>
passive close
appl: close
send: FIN
simultaneous close
FIN_WAIT_1
recv: FIN
send: ACK
CLOSING
recv: ACK
send: <nothing>
recv: FIN, ACK
send: ACK
recv: ACK
send: <nothing>
FIN_WAIT_2
recv: FIN
send: ACK
TIME_WAIT
2MSL timeout
active close
normal transitions for client
normal transitions for server
appl: state transitions taken when application issues operation
recv: state transitions taken when segment received
send: what is sent for this transition

3 Before You Start

This project requires Python 2.7. Please make sure you have that installed. Other versions will not work.

3.1 Getting Skeleton Code

```
$ git clone https://github.com/NetSys/cs168-fall18-student.git
```

You should end up with a directory at cs168-fall18-student/proj2_transport/pox; all commands in this spec (unless they say otherwise) assume this is your current directory.

3.2 Consider using an IDE

You can use an IDE (Integrated Development Environment) to assist you with navigating and editing the project code. A good IDE will let you jump to symbol definitions, go forward and backward to different methods, and can infer the types of many variables. PyCharm Community Edition is a free, fully featured IDE that checks off all of these boxes and more. Simply open the pox directory with PyCharm to get started. Ensure that you change the Python version to 2.7 in the Project Interpreter settings.

3.3 The POX Simulation Environment

POX is a networking software platform written in Python. POX started life as an OpenFlow controller, but can now also function as an OpenFlow switch, and can be useful for writing networking software in general. [1]

We will be using POX to create virtual end hosts and routers in a simulated network. This simulation is so realistic that after your TCP implementation is complete, you could use it to communicate with real servers on the Internet (even using your web browser) with only a bit of extra configuration. If you are interested in this please post a question in Piazza after the project is due.

3.4 Trying out POX

After cloning the repos, we can now try out a test to make sure that everything has been installed and set up correctly. From the top level pox directory, try the following command to run POX using a configuration file that just checks that things seem to be working.

```
$ python pox.py config=ext/cs168p2/tests/sanity_test.cfg
```

And you should see the following message if everything is correct:

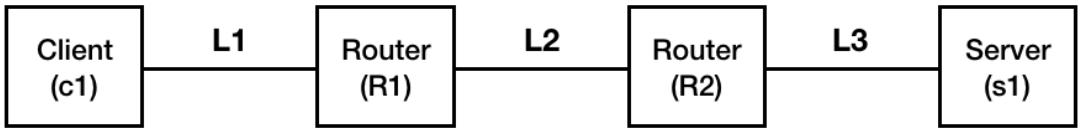
```
$[test ] [00:00:02] All checks passed, test PASSED
```

[1] Some interesting tidbits not directly related to this project: OpenFlow is an open network protocol for remotely controlling programmable network switches in a networking paradigm known as Software Defined Networking, which will be touched on in a future lecture. The switches may be either hardware switches or software *virtual switches*, of which Open vSwitch (OVS) is one of the most notable. Scott's company Nicira was a major early developer of OpenFlow and OVS. Despite POX's origins in this type of work, this project is actually entirely unrelated to it.

3.5 Testing

To help you check your work, we provide you with unit tests. These tests will be the **only** tests that we'll use to grade your submission, i.e., there will be no hidden tests (see Section 7 for details on grading). *However*, there will be some hidden tests not used for grading but for identifying various types of cheating.

The tests are run on the following network topology,



where your socket runs as c1 and the staff socket runs as s1. Links L1 and L3 have infinite bandwidth and zero propagation delay. Link L2 is the one the tests can control, with variable specifications depending on the test.

The tests are grouped to correspond with the stages of the project, and each test one aspect of the implementation as independently as possible. As you work on each stage, you should run the unit tests for that stage (and all previous stages) and make sure you pass them. For example, after you finish Stage 5, you can run unit tests by invoking:

```
$ cd ext/cs168p2
$ python autograder.py s5 # Runs the unit tests for Stage 5
$ python autograder.py all # Runs all unit tests
$ python autograder.py all 5 # Runs all unit tests up to and including stage 5
```

While you are working on a particular stage (or a bug!), you may also want to run particular tests, and you may want to read the code of the test that's failing to see exactly what it's doing. Each test is run by invoking POX with a particular configuration file. Most of these then load up a specific Python module. See the `tests` directory for a list of them. In general, you can invoke a particular test by doing something like the following:

```
$ cd ext/cs168p2
$ python autograder.py s5_t1
```

This will also show you the test console output.

Note: The unit tests are **not** guaranteed to be comprehensive—it is possible for your implementation to have a defect in one stage that manifests itself by failing unit tests for a later stage.

3.6 Debugging

We identify two different ways to debug your code. One is to trace the packets between components in the network (see diagram in 3.5), and the other one is to use a more programmatic approach.

3.6.1 Programmatic

You can use self.log to print debug messages; e.g. self.log.error(string) anywhere within the StudentUSocket class. You can also set breakpoints with pdb in your python code; e.g. "import pdb; pdb.set_trace()" wherever you want to set a breakpoint. If you've never used pdb you can read about it here. However, if you wish to use pdb you'll need to run the test without the autograder. Example: if you want to set breakpoints in your code and run test s1_t1, you'll need to do the following from the root pox directory:

```
$ python pox.py config=ext/cs168p2/tests/s1_t1.cfg
```

When you execute tests this way, you might see logs starting with the string "tcp_sockets". This are logs coming from the server code. Logs coming from your code will begin with "student_socket". Important note: remove **all** breakpoints from your code before turning it in. You'll only be given points for tests that pass in the autograder without timing out, so if a test pauses because it hit a breakpoint in your code, you'll lose those points.

3.6.2 Tracing packets

Every time a test is run, the command line output will specify a location for a pcap file. POX can produce pcap (packet capture) files that allows you to use a network analyzer such as WireShark. The pcap is automatically captured by Pox and written to the `trace` directory. To debug, download WireShark, go to the File menu, and Open the pcap file. Note: you'll see sequence numbers always starting at 0 on WireShark. This is because the tool shows relative numbers (deltas) with respect to the initial sequence number.

3.7 Resources

Listed below are all the RFCs that went into the creation of this project. Note that these are only for reference, and if there's any contradiction between this specification and the RFCs, this document takes priority.

- **RFC 793** - Transmission Control Protocol
  There are a bunch of RFCs that update or modify this. Additionally, the API it describes is not a perfect match to the sockets API, which we follow more closely.
- **RFC 1122** - Requirements for Internet Hosts – Communication Layers
  Most notably, this contains some fixes/updates to RFC 793. Also contains discussion of zero window probes.
- **RFC 6298** - Computing TCP's Retransmission Timer
  The most up-to-date and comprehensive RFC on the retransmission timer. This includes all the Karn/Partridge stuff.

3.8 Requirements

Before we get started with the implementation, let's lay down some ground rules:

- Your `TCP Socket` implementation must live entirely in `ext/cs168p2/student_socket.py`; no other files will be considered or usable during grading (so you may, for example, write additional test cases,

but correct operation of your socket code must not rely on anything not in the `student_socket.py` file).

- You should **not** touch the POX code itself. Nor should you write code that dynamically modifies POX, the simulator, or the tests. Additionally, don't override any of the methods which aren't clearly intended to be overridden, and don't alter any "constants." *You will receive zero credit for turning in a solution that modifies the simulator itself or otherwise subverts the assignment. If you're not sure about something: ask.*
- Your `TCP Socket` instances should communicate with other `TCP Socket` instances **only** via the sending of packets. Global variables, class variables, calling methods on other instances, etc., are not allowed—each `TCP Socket` instance should be entirely standalone!
- The skeleton code we provide for `TCP Socket` will already have several instance variables defined for the class. **Do not** modify or remove these definitions; you'll be using them during the project. Similarly, **do not** remove any existing method definitions; you'll be filling in their implementations.
- However, feel free to add your own instance variables and/or helper methods, as long as they don't break the provided tests.
- You should not need any additional `import` statements. It would be fine for you to use, say, Python's `collections` module. However, you should **not** use (or need to use!) the `time`, `threading`, or `socket` modules. If you have questions, ask!

4 Getting Familiar

The most common way in which programs interact with the networking capabilities of operating systems is via the Berkeley sockets API, invented right here at Berkeley. Applications call the socket API functions in order to manage connections and to communicate.

All the code you write will be inside the StudentUSocket class. This is a subclass of StudentUSocketBase that contains the interface to POX and a number of things which you shouldn't really need to worry about for this project. You'll be writing up code in StudentUSocket to implement the rest of TCP.

4.1 Methods in skeleton code

All methods in the skeleton code you are provided have docstrings, but we summarize all of them here for easier reference later in this specification.

```
class RXControlBlock (object)
    nxt = 0  # next expected receive sequence number
    wnd = 0  # receive window
    irs = 0  # initial receive sequence number

class TXControlBlock (object)
    una = 0  # oldest unacknowledged sequence number
    nxt = 0  # next send sequence number to use
    wnd = 0  # send window
    wl1 = 0  # seg sequence num used for last window update
    wl2 = 0  # seg ack num used for last window update)
```

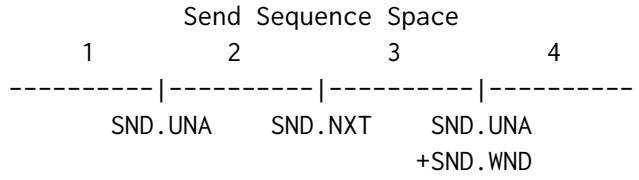
```
    iss = 0  # initial send sequence number

class FinControl (object)
    def acks_our_fin (self, ack)
    def set_pending (self, next_state=None)
    def try_send (self)

class RetxQueue(object)
    def push (self, p)
    def pop (self)
    def pop_upto (self, seq_no)
    def get_earliest_pkt (self)
    def empty (self)
    def peek (self)

class RecvQueue(RetxQueue)
    def push (self, p)

class StudentUSocket(object)
    def _do_timers (self)
    def new_packet (self, ack=True, data=None, syn=False)
    def close (self)
    def acceptable_seg (self, seg, payload)
    def connect (self, ip, port)
    def tx (self, p, retxed=False)
    def rx (self, packet)
    def handle_synsent (self, seg)
    def update_rto (self, acked_pkt)
    def handle_accepted_payload (self, payload)
    def update_window (self, seg)
    def handle_accepted_ack (self, ack)
    def check_ack (self, seg)
    def handle_accepted_seg (self, seg, payload)
    def maybe_send (self)
    def start_timer_timewait (self)
    def check_timer_timewait (self)
    def check_timer_retx (self)
    def set_pending_ack (self)
    def maybe_send_pending_ack  (self)
```

5 Sequence spaces and segments

Before we go on and talk about the project stages, we need to talk about the different kinds of sequences spaces that TCP uses. Since you'll be implementing the protocol itself, we will have to go into significant detail here.

Each host in TCP needs to maintain a *send* sequence space and a *receive* sequence space. In this project, the RXControlBlock is the data structure that keeps the send sequence space and is instantiated for you in self.snd. TXControlBlock keeps the receive sequence space and is instantiated in self.rcv. 1 sequence space usually corresponds to 1 octet, or 1 byte, but some packets occupy sequence space even though they don't carry any payload; more on this later.

5.1 Sending data



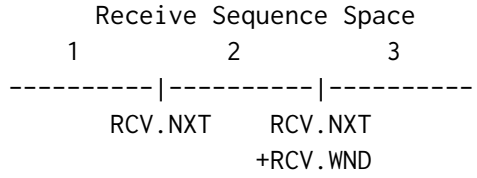

```
1 - old sequence numbers which have been acknowledged
2 - sequence numbers of unacknowledged data
3 - sequence numbers allowed for new data transmission
4 - future sequence numbers which are not yet allowed
```

Within the diagram, the definitions of each label are:

- SND.UNA: the oldest unacknowledged sequence number
- SND.NXT: the next sequence number to be sent
- SND.WND: the current send window (the receive window of the peer)

Therefore, when we send segments that are valid and occupy sequence space, we must increment SND.NXT. **FINs and SYNs occupy 1 sequence space** and segments with payload occupy **size of payload** sequence space. Furthermore, when the peer acknowledges new segments we have sent, we move SND.UNA to the right to match the segment ack number. Finally, every segment we receive comes with a window field (the window advertisement) which is the maximum buffer size that we can send to the peer (including packets in flight).

5.2 Receiving data




```
1 - old sequence numbers which have been acknowledged
2 - sequence numbers allowed for new reception
3 - future sequence numbers which are not yet allowed
```

Within the diagram, the definitions of each label are:

- RCV.NXT: next sequence number expected on an incoming segments, and is the left or lower edge of the receive window
- RCV.WND: the current size of our receive window

Therefore, when we receive segments they are stored starting at RCV.NXT, and up to RCV.WND bytes.

5.3 Segments

We refer to segments as any TCP packet that we send or receive. TCP segments are usually encapsulated in IP packets– we mostly ignore IP packets in this project, however, sometimes you'll see that methods in the socket class receive or send IP packets. The reason is because POX deals with IP packets. If at any point the documentation says that p is an IP packet, you can get the TCP segment with doing p.tcp. In this project, each TCP segment contains the following fields:

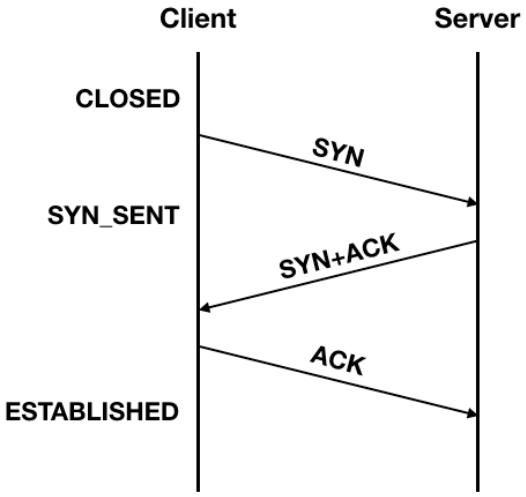
```
seg.seq - segment sequence number
seg.ack - segment acknowledgment number
seg.len - segment length
seg.win - segment window
seg.ACK - whether this segment has the ACK flag set
seg.SYN - whether this segment has the SYN flag set
seg.FIN - whether this segment has the FIN flag set
```

The function new_packet() creates new segments for us so we can transmit them. Further, it sets the segment's sequence number equal to self.snd.nxt and the ack number to self.rcv.nxt.

6 Stages

To guide your implementation, we have split the implementation process into **ten stages**, each of which covers one aspect of the project. Some of these are split into sub-stages. You should follow along stage by stage, and by the end, you will have implemented a functional version of TCP, similar to what you find in the Internet!

6.1 Three-way handshake



Every TCP connection begins with the three-way handshake. In this procedure, the client will try to establish a connection with the server by first sending a packet, with the SYN flag set, of some sequence number x. When the server receives this request, it will respond with a segment, with the ACK and SYN flags

set, with ack number $x + 1$ and some sequence number y. The client will in turn respond to the server's SYNACK with a final segment with ACK flag set, sequence number $x + 1$ and ack number $y + 1$.

During the three-way handshake, the TCP socket will transition between a set of states. When the socket is idle, it is in the CLOSED state. When a connection has been successfully established, it is in the ESTABLISHED state. After the SYN has been sent but before the SYNACK is received, it is in the SYN_SENT state. When we receive the final ACK, we move to ESTABLISHED. The internal state of the socket is kept in self.state.

In this project, you do **not** need to implement a client that can respond to a separate SYN and ACK packets. You can assume that the server will respond with a single SYNACK packet.

1. Begin by taking a look the the connect() function. This function is called by the the application and should start the procedure by sending the initial SYN packet. Build a new packet with the SYN flag set, transmit this packet, and change the connection state to SYN_SENT. Remember to manually set the sequence number for this packet to the initial sequence number. **Hint:** The send() and maybe_send() functions do additional processing we don't need here. Use the raw tx() function.
2. The SYNACK the client receives should be handled by the handle_synsent() function. All packet receives are handled in rx()– modify the function such that if the current state is SYN_SENT, it calls handle_synsent() with the new segmen.
3. In handle_synsent(), if the ACK for the SYN is acceptable (this code is given to you), set the variables self.rcv.nxt, self.snd.una to appropriate values. **Hint:** You want to use modulo operators, check out modulo_math.py, which performs arithmetic operations assuming 32-bit unsigned integers that can wrap around.
4. Still in handle_synsent(), if the new oldest unacknowledged sequence number is larger than our initial sequence number, then this ACK is acking our SYN. If so, move state to ESTABLISHED and then express that we want to send an ACK (see set_pending_ack()). We don't directly send the ACK because we want to send as few packets as possible, so we try to merge ACKs and combine ACKs with data. Once there is a pending ack, rx() will take care of the actual sending. As we will see in more detail in a later stage, every segment we receive advertises a send window, and this ACK is no exception, so make sure to also call update_window().

At this point your socket should do 3 way handshake correctly. You can now run Stage 1 tests.

6.2 Receiving In Order Data

In this stage you'll add support for in-order data arrival to the client. Received in-order data should be stored in rx_data. Consider the scenario where you are receiving 3 packets with payloads that don't overlap: p1, p2, p3. If all of these packets arrive, and they do so in order, then your implementation should simply copy p1, p2, p3 to rx_data and you are done. However, if we assume p2 is lost, then your client will receive and store p1, and then it will receive p3. At this point, you can't store p3 in rx_data, as you would be corrupting your receive buffer (you don't know how large p2 could be!). Instead, for now, you'd need to drop p3 and send an ack back to the server requesting p2.

You will implement the functions handle_accepted_seg() and handle_accepted_payload(). handle_accepted_seg()

is called by rx() after a segment has been validated by acceptable_seg() to be within the receive window; these are some tricky checks, but we provide this code for you.

1. Begin by modifying rx(). If the segment is acceptable, call handle_accepted_seg() on the segment. However, you should only do this if the segment is in-order, i.e., it is the next segment you are expecting (see rcv.nxt)! If you receive an out-of-order packet, set a pending ack and drop the packet here (but allow the rest of rx() to execute).
2. TCP can only process the payload if the state is one of ESTABLISHED, FIN_WAIT_1, or FIN_WAIT_2, and if the length of the payload is non-zero. Implement this functionality in handle_accepted_seg(). If both conditions are true, call handle_accepted_payload() on the payload.
3. In handle_accepted_payload(), increment then next expected byte (self.rcv.nxt). Also, decrement the size of the receive window size by the same quantity. Then we append the payload to rx_data. Finally set a pending ACK.
   Hint: Every time you add numbers or increment number in your send or receive sequence space, think if an overflow could occur.

6.3 Receiving Out of Order Data

In the previous stage we dropped packets that were out of order, while acking the segment we were waiting for. In this stage we will improve on this by temporarily keeping any out-of-order segments until we can process them in-order. For example, if we are supposed to receive packets p1, p2, p3, but p2 is lost, we would only get p1 and then p3. Instead of dropping p3 (as done in the previous stage), we can store it in a *receive queue* and send an ack requesting p2. For the receive queue, we use rx_queue, an object of type RecvQueue. When p2 arrives, we process it (by calling handle_accepted_seg() on it) and this allows us to process p3 as well. Although we can temporarily store packets that arrive out of order, we must process them in order.

A payload overlap can occur for various reasons, but it essentially means that you need to handle situations where the payloads partially overlap across subsequent segments. If this happens, you need to remove some octets from the beginning of one of the segment's payload.

1. Back in rx(), we just called handle_accepted_seg() if the state was in one of many that allowed incoming segments. Instead of doing that, simply insert the segment into the rx_queue—we will process the queue in the next step.
2. The rx() function a good place to check if we can process any in-order segments from the receive queue. Every time rx() is called, there is a chance that the next in-order segment is available for processing in the queue.
   (a) If the packet with the smallest sequence number from the queue is larger than the currently next expected sequence number for incoming segments, then the packet is out of order. In this case, simply set a pending ack and allow the rest of rx() to execute.
   (b) Otherwise, pop that packet from the queue, extract the payload (or a subset of the payload in case of an overlap) and call handle_accepted_seg() on it. Continue doing this until the queue is empty or until the next packet is not in-order.

Hint: When doing comparisons or arithmetic on any sequence space number use modulo operators!
Hint: self.rcv.nxt - p.tcp.seq gives the start of the slice of the payload that is so far unprocessed—useful if there is a payload overlap!

6.4 Simple Sending of Data

Now that we can set up a connection and receive data, the next step is to implement the ability to send data and react to ACKs. When sending data, ACKs from the peer tells us what they have successfully received so far. Applications call the socket function send() with data, and the function fills the transmit buffer (tx_data) with the data. The function maybe_send() is called by send() and by rx() and is responsible for sending as much as the send window allows (snd.wnd).

1. First, we will add code to maintain our send sequence space; among other things, this will tell us what the peer has received so far. In check_ack() add code to implement the following checks and actions:

   (a) If snd.una $<$ seg.ack $\leq$ snd.nxt, the ack number of the received segment is in *2 of the send sequence space* (see diagram in 5.1), and this means one of the packets we sent was just ACKed! In this case, call handle_accepted_ack() on the segment.

   (b) Otherwise, if seg.ack $<$ snd.una (1 in send sequence space), then this is an old ACK. In this case, drop the packet, that is, don't allow the rest of handle_accepted_seg() to execute. However, do allow the rest of check_ack() to execute.

   (c) Otherwise, if the ack has a sequence number that you haven't sent, i.e., seg.ack $>$ snd.nxt (3 or 4 in send sequence space), then set a pending ack and drop the packet; don't allow the rest of check_ack() to execute.

2. Now implement handle_accepted_ack(). For this stage, we only handle in-order ACKs for data that we have sent, therefore, all you have to do for now is update the variable that keeps track of unacknowledged sequence numbers so far.

3. Now that we have the ability to handle ACKs for data we have sent, let's implement support to segmentize and send packets. The function maybe_send() is used to send as much of the transmit buffer as possible, that is, you need to send data currently stored in tx_data. While doing so, you need to maintain the following conditions:

   (a) Trivial, but you can't send more data than what tx_data has.

   (b) In sum, you can't send more data than what your send window allows, including packets in flight.

   (c) While segmentizing tx_data, each segment needs to be equal or smaller in size than the max segment size. This is provided for you in self.mss.

   Transmit each segment using tx().

   Hint: The size of the send window is self.snd.wnd. How can you use the other variables in your send sequence space to compute the amount of data in flight?

4. The last part is to actually transmit the data. Currently the function tx() hands off the packet to the manager, which does the actual sending. Before handing off the packet, update the next sequence number to be sent in your send sequence space for the case that you send a packet with payload.

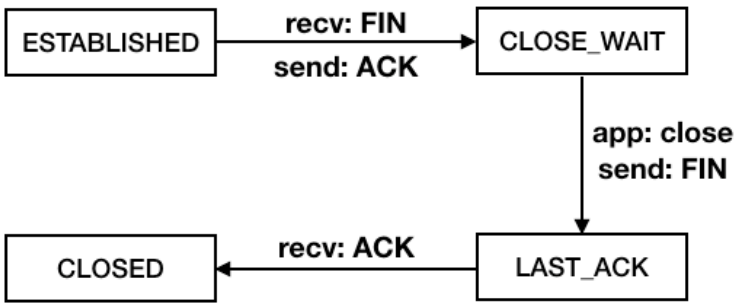
6.5 Honor Advertised Window

Welcome to the shortest stage of this project. In the previous stage, you already implemented obeying the send window when segmentizing data, but so far the value for self.snd.wnd was TX_MAX_DATA, which is not correct. Every time a packet is received, the sender advertises the maximum window size it wants the other side to use for sending.

1. Correctly assign the send window in update_window() to the value the segment advertises. On a full implementation of TCP, this would be set to the min of the advertised window and the congestion control window, but since we won't be implementing congestion control, you should ignore this fact.

6.6 Passive Close

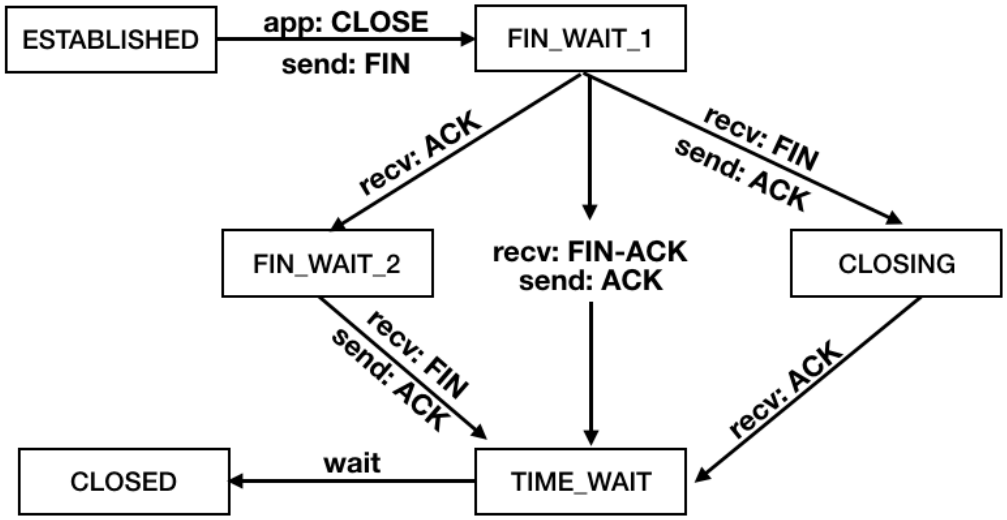
We now turn to connection teardown. Broadly speaking, in TCP there are two graceful shutdown procedures: active close and passive close. The difference lies in who initiates the close—the side that initiates it is defined as the active closer, and the side that responds to the close, the passive closer.



In passive close, a FIN packet is received by our client, and we must respond with an ACK for the FIN and enter CLOSE_WAIT. When an application calls close() in our client, we send a FIN. At this point we enter LAST_ACK, where we stay until our FIN is ACKed.

1. When a FIN arrives, handle_accepted_seg() will call handle_accepted_fin(). Modify handle_accepted_fin() to update the next sequence number expected in our receive sequence space. We also set a pending ACK for the FIN. Then we need to update our state, move to the appropriate state if the current state is ESTABLISHED. Now the protocol will wait until the application in our host calls close()
2. In close(), do the following if the state is the one you moved to in the previous step. You need to set a pending FIN, and we provide code for you to do that—use FinControl. The function you'll use also receives a parameter to move to the next appropriate state when the FIN is actually sent, so be sure to fill that out.
3. Finally, once we get the ACK for our FIN, we can close the connection without further waiting. In check_ack(), add the following code if the state is the one you moved to in the previous step. Check if the ACK we are just received acks the FIN we sent (again, see FinControl). If so, we close the socket by calling _delete_tcb().

6.7 Active Close



On active close, our host is the one that calls close() first, so we send initial FIN packet. Depending on whether we get an ACK followed by a FIN, a FIN-ACK, or a FIN followed by an ACK, there are three paths our state transition can follow. The end result of all three paths is to transition into TIME_WAIT, where we wait for a unique amount of time (30 seconds in the project, see TIMER_TIMEWAIT) in case our last ACK gets lost. Afterwards, the socket closes and transitions into CLOSED.

The state all the paths in active close want to reach is TIME_WAIT. Every time you want to transition to this state, simply call start_timer_timewait(). This will start the TIME WAIT timer, and will move the state to TIME_WAIT too, so you don't have to do it manually.

1. The first step is to react accordingly when an application calls close(). Therefore, in close() change the state to FIN_WAIT_1, and set a pending FIN if the current state is ESTABLISHED.
2. In handle_accepted_fin(), do the following if the state is FIN_WAIT_1:
   (a) If the ACK received acks our FIN, it means that the FIN we just received was a FIN-ACK, so we can transition directly to TIME_WAIT.
   (b) If the FIN received does not ack our FIN, then we should transition to CLOSING. This is the simultaneous close, when we receive the other side's FIN before getting the ACK for our FIN.
3. In check_ack(), transition from FIN_WAIT_1 to FIN_WAIT_2 if the ACK we received acks the FIN we sent. This path in the diagram is if we receive the ACK before the other side's FIN.
4. In check_ack(), transition from CLOSING to TIME_WAIT if the ACK we received acks the FIN we sent. This path completes the transition for simultaneous close.
5. We are almost done! In handle_accepted_fin(), the only transition we are missing is from FIN_WAIT_2 to TIME_WAIT, which happens if we receive a FIN while in FIN_WAIT_2. Implement it and you are done with connection teardown!

6.8 Send retransmission

In stage 4 we implemented simple in-order send of segments. However, we didn't handle the case where sent packets are lost. In this stage, we'll handle sent drop packets by retransmitting them. Instead of an actual timer with an interrupt, retransmits in this project work by tagging every packet with a timestamp of the time at which they are originally transmitted. Before each packet is transmitted for the first time, it is added to a retrasmit queue (see retx_queue and RetxQueue). Packets are removed from the retransmit queue when they are ACKed. Every 100 milliseconds, the earliest packet (oldest) in the retransmit queue is inspected to see if it has expired and if so, it is retransmitted. Only SYNs, FINs and segments with payload are retransmitted–we don't need to retransmit ACKs.

The Retransmit Timeout (RTO) is how long we should wait before a packet times out and needs to be retransmitted, and is given by rto. On this stage, the RTO is fixed to 1 second.

1. In tx(), tag every IP packet that is being transmitted for the first time with the current simulation time stamp. Do this by setting the tx_ts attribute of the IP packet. The current time is defined by the attribute self.stack.now.
2. Further, we tag each IP packet that has been retransmitted at least once with the retxed attribute set to True. So, we add a packet to the retransmit queue only if it is the first time that is being transmitted.
3. Now let's fill out check_timer_retx() that checks the retransmit queue. If there are packets in the queue, peek the earliest, or ol packet from the queue. Compute how long that packet has been in the queue by using the time when it was added to the retransmit queue, and the current time. If the packet has been in the queue longer than or equal to rto, then retransmit this packet. Retransmit one packet at most every time check_timer_retx() is called.
4. Now we have to remove packets from the retransmission queue when they are ACKed. Implement this in handle_accepted_ack(). **Hint:** RetxQueue has a pop_upto method.

6.9 RTO Update & RTT Estimation

In the previous stage, we assumed the RTO was fixed at 1 second. As we saw in class, we need to actually update this value frequently to avoid sending duplicate packets on a high latency link. The RTT (Round Trip Time) is defined as the difference between the time a packet was sent and the time at which its ACK was received. We can compute this number by using the tx_ts attribute you set when transmitting a packet in the previous stage. The RTT is used to properly determine the RTO and must be estimated by measuring how long it took for a packet to be ACKed.

1. To begin, the RTO should be doubled every time there is a retransmitted packet. Implement this in check_timer_retx() when the packet is retransmitted. The RTO should never exceed MAX_RTO, so cap RTO to it.
2. We want to update the RTO whenever we get a new packet ACKed, but **only** if that packet is a clean sample. For every packet ACKed by a new received ACK, check if that packet has been retransmitted. If it is a clean sample, call update_rto() on the packet.

3. update_rto() is called by handle_accepted_ack(), and takes an acked packet and updates self.rto by re-estimating the RTT. The following attributes are already declared for you, and you must only modify rto, srtt and rttvar; do not modify the others.

```
self.rto = 1  # retransmission timeout
self.srtt = 0  # smoothed round-trip time (estimated RTT)
self.rttvar = 0  # round-trip time variation (estimated Deviation)
self.alpha =  1.0/8
self.beta = 1.0/4
self.K = 4
self.G = 0  # clock granularity
```

The following sections from RFC 6298 will also be very helpful:

```
(2.2) When the first RTT measurement R is made, the host MUST set

            SRTT <- R
            RTTVAR <- R/2
            RTO <- SRTT + max (G, K*RTTVAR)

(2.3) When a subsequent RTT measurement R' is made, a host MUST set

            RTTVAR <- (1 - beta) * RTTVAR + beta * |SRTT - R'|
            SRTT <- (1 - alpha) * SRTT + alpha * R'

      The value of SRTT used in the update to RTTVAR is its value
      before updating SRTT itself using the second assignment.  That
      is, updating RTTVAR and SRTT MUST be computed in the above
      order.

      The above SHOULD be computed using alpha=1/8 and beta=1/4 (as
      suggested in [JK88]).

      After the computation, a host MUST update
      RTO <- SRTT + max (G, K*RTTVAR)
```

Finally, clamp the new RTO to self.MAX_RTO and self.MIN_RTO.

6.10 Survey

Congrats on reaching the end! The last step is just to fill out the anonymous 2 minute survey for this project, which is worth 1% of the project. At the end of the survey is a secret word. Fill in the secret_word variable of the function proj2_survey to get this point.

7 Grading

Submission instructions will be posted on Piazza before the deadline. Be sure to familiarize yourself with the **late policy** outlined in the syllabus on the course website.

100% of your grade will come from the unit tests that we have provided to you. 99% of your grade is split equally among the 9 stages (i.e., 11% for each stage). The last 1% comes from Stage 10, the survey. Within each stage, all tests are weighted equally.

Any further details on grading will be posted on Piazza. To submit, upload your `student_socket.py` file to Gradescope.

You must solve this project **individually**. You may not share your code or show your code with anyone, including any custom test code that you may write. You may discuss the assignment requirements or your solutions—*away from a computer and without sharing code*—but you should not discuss the detailed nature of your solution. Also, don't put your code in a public repository.

We expect you all to uphold high academic integrity and pride in doing **your own work**. 23% of academic misconduct cases at a certain junior university are in Computer Science.[2] Let's be better than this. If you get stuck on the project, come to project office hours as early as possible. Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct[3].

7.1 Test Details

On all tests, we are the client, the peer is the server. All tests for a given stage assumes the previous stage's functionality is correct (except in stage 1).

7.1.1 Stage 1

1. 3 way handshake
   (a) Packet 1 comes from client, has SYN flags only
   (b) Packet 2 comes from server, has SYNACK flags, correct ack num, client state is SYN_SENT
   (c) Packet 3 comes from client, has SYNACK flags, correct ack num, client state is ESTABLISHED
2. Same as previous one but the server's initial sequence number is set at wraparound boundary.

7.1.2 Stage 2

1. Receive 1 packet with payload from server. Check theres only 1 packet with payload. Check correct number of packets. Check sequence numbers. Check that received data is correct.
2. Same as test 1 but 3 packets with payload.
3. Same as test 1 but 50 packets with payload.

[2] `http://www.pcworld.com/article/194486/why_computer_science_students_cheat.html`
[3] `http://students.berkeley.edu/uga/conduct.pdf`

4. Same as test 1 but router drops the packet with payload once, client should request it (ack it) and server should retransmit it.
5. Same as test 4 but receive 3 packets. Router will drop interleaved packets with payload (drop 1st, let 2nd pass, drop 3rd, etc.) Each packet is dropped once tops so retransmissions are not dropped.
6. Same as test 5 but receive 15 packets.

7.1.3 Stage 3

1. Receive 1 packet with payload from server. Check data is correctly received. Drop the packet once and then allow it to pass only once. So one successful transmission should be enough for client to get the packet.
2. Same as 1 but receive 3 packets and allow them to pass in an interleaved fashion. Each packet is allowed to pass once tops.
3. Same as 2 but receive 15 packets.
4. Same as 2 but set server's initial sequence number close to wrap around boundary.
5. Same as 3 but set server's initial sequence number close to wrap around boundary.

7.1.4 Stage 4

1. Client sends 1 packet to server, check the server receives all data correctly.
2. Same as 1, but send 3 packets.
3. Same as 1, but send 50 packet.
4. Same as 2, but set server's initial sequence number close to wrap around boundary.
5. Same as 3, but set server's initial sequence number close to wrap around boundary.

7.1.5 Stage 5

1. Set the server's receive window to 1 byte. Client sends 300 bytes, must send 300 packets.
2. Set the server's receive window to 199 bytes. Client sends 1990 bytes, must send 10 packets.

7.1.6 Stage 6

1. Test 3 way handshake. 4th packet is FIN+ACK from server. 5th packet ACK from client. 6th packet FIN+ACK from client. 7th packet ACK from server. Client transitions correctly on passive close states.
2. Test 1 plus set server's initial sequence number close to wrap around boundary.

7.1.7 Stage 7

1. Active close, close connection from client side. Client must send FIN, receives FIN+ACK, sends ACK. Their state transition is ESTABLISHED, FIN_WAIT_1, TIME_WAIT. The time-wait timer should go off and they should end in CLOSE state.
2. close() after send(). While data is still in the transmit buffer, client calls close(). Should wait until after all data is sent before sending the FIN.

7.1.8 Stage 8

1. Client sends data. Drop acks from server that correspond to the beginning of the payload, but allow acks that correspond to the end of the payload to reach the client. Client should not retransmit packets.
2. Client sends one packet. The packet is dropped once. Client should retransmit it after 1 second.
3. Client sends 10 packets of data. Drop interleaved packets but each one is dropped once tops. Client should retransmit dropped packets.

7.1.9 Stage 9

1. Client sends 100 packets, one packet every 25 ms. R1-R2 link latency is set to 200ms. By the end, SRTT should be within +-10% of the RTT. All data should reach server.
2. Test 1 but set link latency to 500ms. SRTT should be within +-5% of the RTT.
3. Test 1 but drop ≈4% of the packets going to server. By end, RTO must be less than 32, SRTT less than 16, and RTTVAR more than 1.5.

7.1.10 Survey

1. Check that hash matches.