CPSC 4510/5510 Computer Networks

Project 3: Implementing a Simple TCP

Note:

In this project, you can form a team of up to 4 members.

Attention:

1. THREE WEEKS are used for this project, and no excuses for deadline extension will be accepted!
2. # lines of code are not many, but need some time for going through the provided code and TCP RFC docs as well as team discussions. Right design choices will be the key to the success of this project.
3. Form a team of members who can really sit down together for discussions and work as a real team.
4. Individual projects are highly discouraged for most of you, if not all!

I. Overview:

In the previous projects, you learned about the socket interface and how it is used by an application. By now, you're pretty much an expert in how to use the socket interface over a reliable transport layer, so now seems like a good time to implement your own socket layer and transport layer! That's what you'll be doing in this project. You'll learn how the socket interface is implemented by the kernel and how a reliable transport protocol like TCP runs on top of the network layer. We're going to call your socket layer MYSOCK and it will contain all the features and calls that you used in Project #1 and #2. Your socket layer will implement a transport layer that we'll call STCP (Simple TCP), which is a stripped down version of TCP. STCP provides a connection-oriented, in-order, full duplex end-to-end delivery mechanism. It is similar to early versions of TCP, which did not implement congestion control or optimizations such as selective ACKs or fast retransmit.

To help you get started, we're providing you with a skeleton system in which you will implement MYSOCK. In fact, the MYSOCK application socket layer has already been implemented for you; you add the functionality needed for the transport layer. The skeleton (stcp_skeleton.tar is provided for download) consists of:

1. a network layer,
2. a ***bogus transport layer that you need to fill in***,
3. the MYSOCK socket interface,
4. and a dummy client and server application to help you debug your socket and transport layer.

To simplify your implementation, you only need to implement the transport layer so that the supplied programs (a dummy client and a dummy server) work in ***reliable*** mode. Thus, you need **NOT** worry about dropped packets, reordering, retransmissions and timeouts in the underlying network layer.

Important: STCP is not TCP! While STCP is designed to be compatible with TCP, there are many distinct differences between the two protocols. When in doubt, the specifications in this project description should be used in your implementation.

II. Structure of the Code:

Network layer

At the lowest layer is the *network* layer. We provide you with a fully functional network layer that emulates a reliable datagram communication service with a peer application. As you'll see if you delve into our code, we actually implemented the reliable datagram service over a regular TCP connection. For the purposes of this project, it just appears to you and your code as a network layer.

Transport layer (transport.c)

The next layer up is the *transport layer*. We provide you with a bogus minimal transport layer in which some basic functions are already implemented. It is provided only so that the client and server will compile (but **NOT** run), and to give you an example of how to use the socket/transport/network layer calls. This is where you will implement the STCP functionality.

Application layer

The *application* layers that we give you are the dummy client and dummy server. The dummy client and server are very simple and are provided to aid you with the debugging of your transport layer. When executed, the client prompts for a filename which it sends to the server. The server responds by sending back the contents of the file. The client stores this file locally under the filename "rcvd". The client can also ask for a file from the server on the command line in a non-interactive mode. The client and server work as expected if the file "rcvd" on the machine where the client is running is identical to the file asked for at the server machine. You may change the client and server as much as you like for debugging purposes. We will not use your versions of the dummy client and server for grading. The client accepts the -q option, which suppresses the output of the received data to the file.

III. Getting Started:

Download the STCP tarball (stcp.tar) for this project and extract it into a new directory in your cs1/cs2 account. A Makefile is included for you in the tarball. If for some reason you need to do something different with `make` for testing purposes, please create your own Makefile and build with it by calling `make -f YourMakefile` during development. **Your code must build with the provided standard Makefile when you submit!**

IV. MYSOCK Layer Definition:

This section describes the protocol your transport layer will implement. RFC 793 details the TCP state machine, which if followed carefully will likely decrease the amount of time it will take you to implement STCP and reduce bugs in the process. Some corrections to this RFC were addressed in RFC 1122 (section 4.2) that you will also want to read.

Overview
STCP is a full duplex, connection oriented transport layer that guarantees in-order delivery. Full duplex means that data flows in both directions over the same connection. Guaranteed delivery means that your protocol ensures that, short of catastrophic network failure, data sent by one host will be delivered to its peer in the correct order. Connection oriented means that the packets you send to the peer are in the context of some pre-existing state maintained by the transport layer on each host.

STCP treats application data as a stream. This means that no artificial boundaries are imposed on the data by the transport layer. If a host calls **mywrite()** twice with 256 bytes each time, and then the peer calls **myread()** with a buffer of 512 bytes, it will receive all 512 bytes of available data, not just the first 256 bytes. It is STCP's job to break up the data into packets and reassemble the data on the other side.

STCP labels one side of a connection active and the other end passive. Typically, the client is the active end of the connection and server the passive end. But this is just an artificial labeling; the same process can be active on one connection and passive on another (e.g., the HTTP proxy of Project 2 that "actively" opens a connection to a web server and "passively" listens for client connections).

The networking terms we use in the protocol specification have precise meanings in terms of STCP. Please refer to the glossary later.

STCP Packet Format
A STCP packet has a maximum segment size of 536 bytes. It has the same header format as TCP. The header format is defined in transport.h as follows:

```
typedef uint32_t tcp_seq;

struct tcphdr {
        uint16_t th_sport;              /* source port */
        uint16_t th_dport;              /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgment number */
#if     __BYTE_ORDER == __LITTLE_ENDIAN
        uint8_t th_x2:4;                /* unused */
        uint8_t th_off:4;               /* data offset */
#elif   __BYTE_ORDER == __BIG_ENDIAN
        uint8_t th_off:4;               /* data offset */
        uint8_t th_x2:4;                /* unused */
#else
#error  __BYTE_ORDER must be defined as __LITTLE_ENDIAN or __BIG_ENDIAN!
#endif
        uint8_t th_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04                     /* you don't have to handle this */
#define TH_PUSH 0x08                     /* ...or this */
#define TH_ACK  0x10
#define TH_URG  0x20                     /* ...or this */
        uint16_t th_win;                 /* window */
        uint16_t th_sum;                 /* checksum */
        uint16_t th_urp;                 /* urgent pointer (unused in STCP) */
} __attribute__ ((packed)) STCPHeader;
```

For this project, you are not required to handle all fields in this header. Specifically, the provided network layer wrapper code sets th_sport, th_dport, and th_sum, while th_urp is unused; you may thus ignore these fields. Similarly, you are not required to handle all legal flags specified here: TH_RST, TH_PUSH, and TH_URG are ignored by STCP. The fields STCP uses are shown in the following table. Note that any relevant *multi*-byte fields of the STCP header will entail proper endianness handling with `htonl`/`ntohl` or `htons`/`ntohs`.

Descriptions of the fields you need to handle are as follows:

Field	Type	Description
th_seq	tcp_seq	Sequence number associated with this packet.
th_ack	tcp_seq	If this is an ACK packet, the sequence number being acknowledged by this packet. This may be included in any packet.
th_off	4 bits	The offset at which data begins in the packet, in multiples of 32-bit words. (The TCP header may be padded, so as to always be some multiple of 32-bit words long). If there are no options in the header (which there should not be for the packets you send), this is equal to 5 (i.e. data begins twenty bytes into the packet).
th_flags	uint8_t	Zero or more of the flags (TH_FIN, TH_SYN, etc.), OR'd together.
th_win	uint16_t	Advertised receiver window in bytes, i.e. the amount of outstanding data the host sending the packet is willing to accept.

Sequence Number
STCP assigns sequence numbers to the streams of application data by numbering the bytes. The rules for sequence numbers are:

- Sequence numbers sequentially number the SYN flag, FIN flag, and bytes of data, starting with a randomly chosen sequence number between 0 and 255, both inclusive. (The sequence numbers are exchanged using a 3-way SYN handshake in STCP as explained later).
- Both SYN and FIN indicators are associated with one byte of the sequence space which allows the sequence number ACKing mechanism to handle SYN and non-data bearing FIN packets despite the fact that there is no actual associated payload.
- The transport layer manages two streams for each connection: incoming and outgoing data. The sequence numbers of these streams are independent of each other.
- The sequence number should always be set in every packet. If the packet is a pure ACK packet (i.e., no data, and the SYN/FIN flags are unset), the sequence number should be set to the next unsent sequence number. (This is purely for compatibility with real TCP implementations; they discard packets with sequence numbers that fall well outside the sender's window. STCP itself doesn't care about sequence numbers in pure ACK packets).
- The Initial Sequence Number (ISS) can be chosen randomly.

Data Packets
The following rules apply to STCP data packets:

- The maximum payload size is 536 bytes.

- The th_seq field in the packet header contains the sequence number of the first byte in the payload.
- Data packets are sent as soon as data is available from the application. STCP performs no optimizations for grouping small amounts of data on the first transmission.

ACK Packets
In order to guarantee delivery, data must be acknowledged. The rules for acknowledging data in STCP are:

- Data is acknowledged by setting the ACK bit in the STCP packet header flags field. If this bit is set, then the th_ack field contains the sequence number of the next byte of data the receiver expects (i.e. one past the last byte of data already received). This is true no matter what data is being acknowledged, i.e. payload data, SYN, or FIN sequence numbers. The th_seq field should be set to the sequence number that would be associated with the next byte of data sent to the receiver. The th_ack field should be set whenever possible (this again isn't required by STCP, but is done for compatibility with standard TCP), even for a non-ACK packet.
- Data may accompany an acknowledgment. STCP is not required to generate such packets (i.e. if you have outstanding data and acknowledgments to send, you may send these separately), but it is required to be capable of handling such packets.
- Acknowledgments are not delayed in STCP as they are in TCP. An acknowledgment should be sent as soon as data is received.
- If a packet is received that contains duplicate data, a new acknowledgment should be sent for the current next expected sequence number.

Sliding Window
There are two windows that you will have to take care of: the receiver and sender windows.

The receive window is the range of sequence numbers which the receiver is willing to accept at any given instant. The window ensures that the transmitter does not send more data than the receiver can handle.

Like TCP, STCP uses a sliding window protocol. The transmitter sends data with a given sequence number up to the window limit. The window "slides" (increments in the sequence number space) when data has been acknowledged. The size of the sender window, which is equal to the other side's receiver window, indicates the maximum amount of data that can be "in flight" and unacknowledged at any instant, i.e. the difference between the last byte sent and the last byte ACK'd.

The rules for managing the windows are:

- The local receiver window has a fixed size of 3072 bytes.
- The congestion window has a fixed size of 3072 bytes. STCP does not perform adaptive congestion control.
- The sender window is the minimum of the other side's advertised receiver window, and the congestion window. Note that STCP should always respect the most recently advertised receiver window.

- Note that received data may cross either end of the current receiver window; in this case, the data is split into two parts and each part is processed appropriately.
- Do not send any data outside of your sender window.
- The first byte of all windows is always the last acknowledged byte of data. For example, for the receiver window, if the last acknowledgment was sequence number 8192, then the receiver is willing to accept data with sequence numbers of 8192 through 11263 (=8192+3072-1), inclusive.

TCP Options
The following rules apply for handling TCP options:

- STCP does not generate options in the packets it sends.
- STCP ignores options in any packets sent by a peer. It must be capable of correctly handling packets that include options by offsetting any included data payload appropriately.

Retransmissions
Because this project assumes that the network layer is *reliable*, you do not need to implement retransmissions since there will be no packet loss, timeouts, or reordering.

Network Initiation
Normal network initiation is always initiated by the active end. Network initiation uses a three-way SYN handshake exactly like TCP, and is used to exchange information about the initial sequence numbers. The order of operations for initiation is as follows:

- The requesting active end sends a SYN packet to the other end with the appropriate seq number (seqx) in the header. The active end then sits waiting for a packet with the SYN flag set.
- The passive end sends a SYN_ACK with its own sequence number (seqy) and acks with the number (seqx+1). The passive end waits for an ACK.
- The active end records seqy so that it will expect the next byte from the passive end to start at (seqy+1). It ACKs the passive end's SYN request, and changes its state to the established state.

For more details, be sure to read RFC 793. Pay special attention to each state in the connection setup, including the simultaneous open scenario

Network Termination
As in TCP, network termination is a four-way handshake between the two peers in a connection. The order of closing is independent of the network initialization order. Each side indicates to the other when it has finished sending data. This is done as follows:

- When one of the peers has finished sending data, it transmits a FIN segment to the other side. At this point, no more data will be sent from that peer. The FIN flag may be set in the last data segment transmitted by the peer. You are not required to generate FIN + data packets, but your receiver must be capable of handling them.
- The peer waits for an ACK of its FIN segment, as usual. If both sides have sent FINs and received acknowledgements for them, the connection is closed once the FIN is acknowledged. Otherwise, the peer continues receiving data until the other side also initiates a connection close request. If no ACK for the FIN is ever received, it terminates its end of the connection anyway.

RFC 793 includes more details on connection termination; pay special attention to the TCP state diagram as you will need to implement the majority of the FSM in the transport layer. Note that you are not required to support TIME_WAIT.

Glossary

ACK packet

> An acknowledgment packet; any segment with the ACK bit set in the flags field of the packet header.

Connection

> The entire data path between two hosts, in both directions, from the time STCP obtains the data to the time it is delivered to the peer.

Data packet

> Any segment which has a payload; i.e. the th_off field of the packet header corresponds to an offset less than the segment's total length.

FIN packet

> A packet which is participating in the closing of the connection; any segment with the FIN bit set in the flags field of the packet header.

Payload

> The optional part of the segment which follows the packet header and contains application data. Payload data is limited to a maximum size of 536 bytes in STCP.

Segment

> Any packet sent by STCP. A segment consists of a required packet header and an optional payload.

Sequence Number

> The uniquely identifying index of a byte within a stream.

Network

> The data path between two hosts provided by the network layer.

Stream

An ordered sequence of bytes with no other structure imposed. In an STCP connection, two streams are maintained: one in each direction.

Window

Of a receiver's incoming stream, the set of sequence numbers for which the receiver is prepared to receive data. Defined by a starting sequence number and a length. In STCP, the length is fixed at 3072.

V. Transport Layer Interface

The interface to the transport layer is given in transport.h. The interface consists of only one function:

extern void transport_init(mysocket_t sd, bool_t is_active);

This initializes the transport layer, which runs in its own thread, one thread per connection. This function should not return until the connection ends. sd is the 'mysocket descriptor' associated with this end of the connection; is_active is TRUE if you should initiate the connection.

To implement the STCP transport layer, **the only file you should modify is transport.c**. While STCP is a simplified version of TCP, it still implements a lot of the TCP finite state machine (FSM). Within transport.c, aside from transport_init(), there is also a stub for a local function, control_loop(), where you should implement the majority of the "event-driven" STCP transport FSM. By event-driven we mean use of the stcp_wait_for_event() function to receive signals from the application layer for data or connection close, and the network layer for incoming packets. Each iteration of the control_loop() should handle the current set of pending events and update the state of the transport FSM accordingly.

VI. Network Layer Interface

The network layer provides an interface to the datagram service delivery mechanism. Underpinning this interface are a pair of send/recv queues used for communication between the transport and network layer threads. Your transport layer will build reliability on top of this layer using the functions implemented in the network layer. The interfaces are defined in stcp_api.h. Study it well. You are not required, but are highly recommended, to study the implementation of the functions in the network layer. Note that stcp_network_send() takes a variable number of arguments, but in general use, you will either use it with a single argument (full STCP packet buffer or just a STCP header buffer) or with two arguments (STCP header buffer, STCP data buffer). The last argument to stcp_network_send() must be NULL to demarcate the end of the vararg list.

VII. Application Layer Interface

The application level socket interface is used by the client/server programs to establish STCP connections: myconnect(), mybind(), mylisten(), myaccept(), etc and to send/recv data. Underlying the interface between the application and transport layer are a pair of send/recv queues for communication between the two threads. All the transport layer needs to know is when there is data available on the recv queue and when the application has closed the connection, which is communicated via the stcp_wait_for_event() mechanism. Once again, study the interface functions defined in stcp_api.h well as they will be the essential interface for communication and control between the transport layer and the application layer above and the network layer below.

Note that you may **ONLY** call the functions declared in stcp_api.h from your code. You must **not** call any other (internal) functions used in the mysock implementation.

VIII. Test Your Code

The provided file transfer server and client should be used to test your code. You may modify the code for the client and server however you wish for testing purposes, but your modifications will not be taken into account for grading.

Make sure that your implementation does not drop packets or send them out of order.

Miscellaneous Notes and Hints

1. The calls myconnect() and myaccept() block till a connection is established (or until an error is detected during the connection request). To cause them to unblock and return to the calling code, use the stcp_unblock_application() interface found in stcp_api.h.
2. mybind() followed by mygetsockname() does not give the local IP address; mygetsockname() (like the real getsockname()) does not return the local address until a remote system connects to that mysocket.
3. We will be testing your code in the Linux environment (cs1 & cs2) provided. Make sure your code compiles and runs correctly in it.
4. Correct endianness will be tested. Don't neglect to include your ntohs() and htons(), etc. calls where appropriate. If you forget them, your code may seem to work correctly while talking to other hosts of similar endianness, but break when talking to systems running on a different OS.
5. If you get the error stcp_api.c:xxx: ssize_t stcp_network_send(mysocket_t, const void*, size_t, ...): Assertion `packet_len + next_len <= sizeof(packet)' failed, then you're forgetting to put NULL as the last argument.

IX. Deliverables

The deliverables for this assignment are:

1. Your modified transport.c. *Do not modify any other .c or .h files found in the stub code.*

2. A README describing the design of your transport layer, and any design decisions/tradeoffs that you had to consider. One page is enough for the writeup. For team projects, README must include all team members' names.

X. <u>Grading</u>

Label	Grading components	Comments
3-A	[5 pts] Complete file submission: transport.c and well-documented README -- README file should list each team member's name and contributions -- README file should describe your design decisions for STCP	1. Missing file: -1 pt 2. README does not follow the requirement: -1 to -4 pts
3-B	[10 pts]Basic Functionality: -- Work with the provided client/server and able to transmit any reasonably sized file from the server to the client without corruption. You can use "diff oldfile newfile" to check if two file contents are identical or not.	If your STCP does not work as expected, you get at most 40% of 10 points, depending on your implementation.
3-C	[4 pts] Well-written code: for readability, error checking, byte orders, and comments.	
3-D	[1 pt] Your STCP should run silently: any status messages or diagnostic output should be off by default.	
3-E	Either the program cannot be compiled Or fatal segmentation faults	Zero points for this project!

XI. Frequently Asked Questions (FAQ)

Q1. There seems to be lots of stuffs, where should I start?

A1. Read the TCP chapters in textbook, and RFC 793. Make sure to clearly understand how TCP works before you start. Then study the code and try to understand how the files (especially those related to mysock and transport layer) are structured and connected with each other. Note that the only file you need to modify is transport.c, and that stcp_api.h contains all the functions you may use.

Q2. What am I allowed to modify in transport.c? What should I leave alone?

A2. You can change anything you like in transport.c, except for code inside "#ifdef FIXED_INITNUM." (Grading depends on this staying the same!) You can add any other functions or data types that you wish.

Q3. I'm confused about how connection termination works.

A3. When the application requests that a mysocket is closed (via myclose()), your transport layer will receive a 'close requested event' from stcp_wait_for_event(); this is your cue to send a FIN segment once all pending data has been successfully transmitted. When you receive a FIN from the peer, and all expected data from the peer has been passed up to the application, you indicate that subsequent myread()s should return zero by calling stcp_fin_received().

Q4. Do I have to consider the case where a packet is truncated? (i.e. the data length implied by the STCP header is less than the actual size of the sent packet)

A4. No. You can assume that the implied payload length: len_from_stcp_network_recv - stcp_data_offset is correct and that no bytes at the tail end of the packet were lost, i.e. datagram transmission is all or nothing. If your buffer can fit it, you always get the whole datagram from the network layer as transmitted by the sender.

Q5. Why this weird threaded model? It makes everything harder to debug. Why not just have the application directly call the transport layer?

A5. Several reasons. For one, it actually makes things easier in the long run. If there was just one process/thread, how would the transport layer get control of the CPU when a timeout occurs or when a packet comes in? The only way to do this would be with signals. The signal model can work, but is much harder to debug with regards to things like race conditions. You have to explicitly deal with reentrancy issues and critical sections.

The event-driven model we have allows the transport layer to sleep until an interesting event happens. In some respects, this is a simpler model than you'd face in implementing a real TCP stack.

The second reason is that it abstracts the socket layer away from the transport layer. The layer you are writing is much more like a protocol stack that would actually fit into the operating system: you have a defined interface above you and below you, and you don't really need to know or care what the user processes above you are doing.

Q6. What is the meaning of 'absolute time' in stcp_wait_for_event()?

A6. It is the system time at which the function should return, if there is no pending event of interest. It is *not* a time interval relative to the current time, i.e. it has the same origin as functions like gettimeofday(2). Note that timespec used by stcp_wait_for_event() has a tv_nsec (nanosecond) field as opposed to the more common timeval which has a tv_usec (microsecond) field. If you use gettimeofday(2) to obtain the current time you will need to convert between the two struct types

Q7. What functions and data types can I use from the stub code in my solution?

A7. Anything declared in stcp_api.h or transport.h is fair game. You should *not* use any other methods or data types defined by the mysock implementation. You should study stcp_api.h in detail as these will be the primary (re: only) network/application I/O functions that transport.c will use. For example, you must use stcp_network_send() and stcp_network_recv() to send and receive packets from the network.

Q8. Do I need to deal with sequence number wrap around (goes over 2^32 and starts from 0 again)?

A8. No. This does add some complexity into the assignment. You're not required to deal with it. We're not going to test your programs to transfer 4 gigabytes of data :-) But keep in mind this happens in real TCP.

Q9. What is the meaning of __attribute__ ((packed))?

A9. In modern computer architectures, the internal representation of a struct with a bunch of fields might contain some bytes of inner padding to align the fields with the word boundaries (this is done for performance reasons). However when you try to send this data, you do not want to send unnecessary padding, because the receiving machine might have a different internal representation.

So, what you want the compiler to do is to eliminate any padding, and to pack all fields back to back, so that the internal and the network representation are the same (except perhaps for byte ordering). This is the role of the PACK directives.

Q10. Do I have to set the source and destination ports in the TCP header?

A10. Nope. Although this would be handled by 'real' TCP, because of the way we simulate the underlying unreliable network layer, the specifics of how ports are assigned depends on the underlying network layer in the mysock implementation. Consequently, the mysock layer takes care of assigning ports and setting these values accordingly. You do not have to handle demultiplexing of connections.

Q11. Do I have to check for packet corruption with a checksum?

A11. No, but you will probably still want to do other error checking in the header.

Q12. How can I accurately measure elapsed time?

A12. You can use the function gettimeofday(2). You cannot use any function from the real-time library (-lrt).

Q13. The receiver window has closed so I can't send any more data until I get an ACK. What should I do with the extra data that I can't send?

A13. You should only read as much data from the application as you're planning on sending. Let the rest of it sit there until you're ready to send it (i.e. an ACK opens up the receive window again).

Note that the transmitter has to keep track of the sender window, based on the 3072-byte receiver window, the current ACK sequence number and the outstanding amount of un-acknowledged data, in order to know how much to send.

Q14. If I receive a segment with a sequence number less than the current ACK sequence number, can I just discard it?

A14. No. You have to look at both the start and end of the segment. For example, if the ACK sequence number is 1000 (1000 is the next byte you expect), and a segment comes in with a sequence number of 800 and a length of 300 (i.e. ends at 1099), then you have to keep the data from 1000 to 1099. A similar case exists at the end of the receive window.

XII. **Submission (**Deadline: 3:45PM, Thursday, May 26, 2016. THREE weeks for this project. NO excuses for deadline extension will be accepted! **)**

Your proxy should be a multi-threading program (using PThread) and it handles synchronization and race condition. You will need to submit a tarball file (i.e., project3.tar) containing the following two files:

- transport.c
- A README file describing your code and the design decisions that you made.

Run the command to submit the tarball file:

```
/home/fac/zhuy/class/network/submit   p3   project3.tar
```