STCP: Implementing a Reliable Transport Layer

Overview

In the first two assignments, 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 reliable transport layer! That's what you'll be doing in this assignment. You'll get to learn how the socket interface is implemented by the kernel and how a reliable transport protocol like TCP runs on top of an unreliable delivery mechanism. We're going to call your socket layer MYSOCK and it will contain all the features and calls that you used in Assignment #1. Your socket layer will implement a transport layer that we'll call STCP (Simple TCP), which is in essence a stripped-down version of TCP. STCP is compatible with TCP, and provides a reliable, 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 the MYSOCK and STCP layers. In fact, the MYSOCK layer has already been implemented for you; you get to add the functionality needed for the transport layer. The skeleton consists of a network layer, a bogus transport layer, and also a dummy client and server application to help you debug your socket and transport layer.

This assignment can be seen as two shorter assignments (named HW3.A and HW3.B), where you will be adding new functionality incrementally. This is a long assignment and we want you to get started early. To stage your work, we've structured the assignment as two milestones with progressively increasing functionality. You must submit two versions of your code, one for each milestone. This is helpful because say you have completed the coding necessary for milestone HW3.A --- you can then immediately check in your code, and proceed to add the functionality required for milestone HW3.B. In doing so, if it so happens that you break something which was working fine for milestone HW3.A, you will still get points for completing milestone HW3.A.

The milestones are as follows. Please also read the detailed information about each of the milestones here after you have read the basic STCP functionality in this file

HW3.A: Implement the socket/transport layer so that the supplied programs (a dummy client and a dummy server) work in reliable mode using your implementation.

HW3.B: The supplied client and server programs also work in unreliable mode using your implementation.
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 assignment description should be used in your implementation.

The 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 an unreliable datagram communication service with a peer application; i.e. it will send and receive data between a client and a server, but does not guarantee that the data will arrive, or that it will arrive in order. As you'll see if you delve into our code, we actually implemented the so-called unreliable datagram service over a regular TCP connection. For the purposes of this assignment, it just appears to you and your code as a network layer.

You're going to find it helpful to force the network layer to be unreliable. To emulate the behavior of a congested multi-path network, setting the is_reliable parameter to false when creating a socket (use the -U flag when running the program) will cause the network layer to randomly reorder and drop packets. You'll see an example of how this is done in our dummy client/server code.

Transport Layer

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.

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; in fact, we might grade your project with some other (simple and similar) application. Both client and server accept the -U flag to make the network layer unreliable. The client also accepts the -q option, which suppresses the output of the received data to the file.

Getting Started

Download the STCP tarball in klms board and extract it into a new directory in your Unix 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 standard Makefile when you submit!

MYSOCK Layer Definition

This section details the protocol your transport layer will implement. Be sure to also read RFC 793, which

describes TCP in more detail.

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 HW#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.

STCP Packet Format

An STCP packet has a maximum 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 */
#ifdef _BIT_FIELDS_LTOH
        u_int    th_x2:4,                /* (unused) */
                 th_off:4;               /* data offset */
#else
        u_int    th_off:4,               /* data offset */
                 th_x2:4;                /* (unused) */
#endif
        uint8_t th_flags;
```

```
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
#define TH_PUSH 0x08
#define TH_ACK  0x10
#define TH_URG  0x20
        uint16_t th_win;                /* window */
        uint16_t th_sum;                 /* checksum */
        uint16_t th_urp;                /* urgent pointer */
        /* options follow */
};

typedef struct tcphdr STCPHeader;
```

For this assignment, you are not required to handle all fields in this header. Specifically, the provided 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.

The packet header field format (for the relevant fields) is 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	4bits	The offset at which data being 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, 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'ed 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 Numbers

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).
The transport layer manages two streams for each connection: incoming and outgoing data. The sequence numbers of these streams are independent of each other.
Both SYN and FIN indicators are associated with one byte of the sequence space.

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).
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 reliable delivery, data must be be acknowledged. The rules for acknowledging data in STCP are:

Data is acknowledged by setting the ACK bit in the flags field in the packet header. 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. The th_seq field should be set to the sequence number that would be associated with the next byte of data sent to the peer. The th_ack field should be set whenever possible (this again isn't required by STCP, but is done for compatibility with standard TCP).
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 might be 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.
Acknowledgments are only sent for the last contiguous received piece of data. For example, assume the last acknowledgment sent was 10. A packet containing data with sequence number 15 of length 5 is received. An acknowledgment for this packet cannot be sent because of the gap at sequence number 10 of length 5. Instead, an ACK should be sent for 10 again. However, once all the data in this range has been received, the correct acknowledgment to send is for sequence number 20.

Sliding Windows

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

The receiver 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

as and when data has been acknowledged. The size of the sender window indicates the maximum amount of data that can be unacknowledged at any instant. Its size is equal to the other side's receiver window.

The rules for managing the windows are:

The local receiver window has a fixed size of 3072 bytes.
The sender window is the minimum of the other side's advertised receiver window, and the congestion window.
The congestion window has a fixed size of 3072 bytes.
As stated in the acknowledgment section, data which is received with sequence numbers lower than the start of the receiver window generate an appropriate acknowledgment, but the data is discarded.
Received data which has a sequence number higher than that of the last byte of the receiver window is discarded and remains unacknowledged.
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.

Retransmissions

It is an ugly fact of networking life that packets are lost. STCP detects this when no acknowledgment is received within a timeout period. The rules for timeouts are:

Whenever a SYN, data, or FIN segment (i.e. an STCP packet containing anything more than just an acknowledgment) is transmitted, a timeout is scheduled some milliseconds from the time of transmission. You may implement the timeout as you desire--e.g. a fixed timeout, the Karn-Partridge algorithm, etc. Whatever works for you. Note that a fixed RTO may not be sufficient--we strongly suggest that you implement RTT estimation in some form. (This is only a few lines of code).
If the data in a segment is acknowledged before the corresponding timer expires, the timer is canceled and the segment can be safely discarded.
If the timeout for a segment is reached, the segment is examined. If it has been sent a total of 6 times (once from the original send plus 5 retransmissions), then the network is assumed to have failed and the network terminated. Otherwise the segment is retransmitted and the timeout is reset again.
STCP, like TCP, is a Go-back-N protocol. This means that if the transport layer decides that its peer has not received the segment with sequence number n, then it will retransmit all data starting at n, not just that

segment.
Your receiver must buffer any data it receives, even if it arrives out of order. (This isn't strictly required by TCP, but almost any "real" TCP implementation would perform this optimization, and you must too).

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 SYN_ACK.
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.

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 (other than possibly retransmissions, including the FIN). The FIN flag may be set in the last data segment transmitted by the peer. (You are not responsible for generating such packets, but your receiver must be capable of handling them). The usual rules for retransmission and sequence numbers apply for the FIN segment.
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, as per the rules for retransmission, it terminates its end of the connection anyway.

http://www.faqs.org/rfcs/rfc793.html includes more details on connection termination; see in particular the state diagram. 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.

Transport Layer

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.

Network Layer

The network layer provides an interface for the connectionless and unreliable datagram service delivery mechanism. 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. You are not required, but are highly recommended, to study the implementation of the functions in the network layer.

Please note that you may only use the interfaces declared in stcp_api.h in your own code. You must not call any other (internal) functions used in the mysock implementation.

Descriptions of Milestones

Milestone HW3.A: You need to do everything that is required to make the client and server programs work in the reliable mode. Thus, you need not worry about dropped packets, reordering, retransmissions, timeouts in the code for this milestone. To grade your implementation, we will first make sure that no packets are dropped or appear out of order and then test whether your code meets the remaining functionality.
Milestone HW3.B: In this milestone, you need to add to your code in the previous milestone such that the client and server work even in unreliable modes. Thus you will need to worry about go-back-N, retransmissions and timeouts.

Testing Your Code

The provided file transfer server and client should be used to test your code in both reliable and unreliable mode. You may modify the code for the client and server however you wish for testing purposes. We will be grading your submission using our own clients and servers, which will be similar to the provided client/server pair.

Miscellaneous Notes and Hints

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.
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.
We will be testing your code on the 32-bit Linux machines in the lab machines. Make sure your code compiles and runs correctly on those systems. Also make sure that you kill all your jobs when you are not running them.
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.

Deliverables

The deliverables for milestones HW3.A and HW3.B of this assignment are:

Your modified transport.c (named as transport.c for HW3.A part and transportb.c for HW3.B part). You are not allowed to modify or submit any other .c or .h files not found in the stub code download.
README describing the design of your transport layer, and any design decicsions/tradeoffs that you had to consider. One page is enough for the writeup.

Grading

Submission : 10 Points

- ·Submission should be compiled without any errors or warnings.
- ·Submission should include a correct set of files: transport.c README file is enough

Basic Functionality : 30 Points

·Your transport.c file should Include explanation about each functions as a comment

·Before making C code, Just thinking or writing what kind of functions you should implement. It will be written in your comment and is enough to explain your functions.

Correct behavior in HW3.A : 30 Points

·Without error

·Submission can transmit without shuffling of order.

Correct behavior in HW3.B : 30 Points

·Without error

·Submission can transmit with shuffling of order.

***There is a part score in each category. Please do not give up.

Assignment FAQ

A FAQ is also available. Please look over it before asking your question to your TA.