Report: Project 3

CS425A Computer Networks 30th Sep 2016

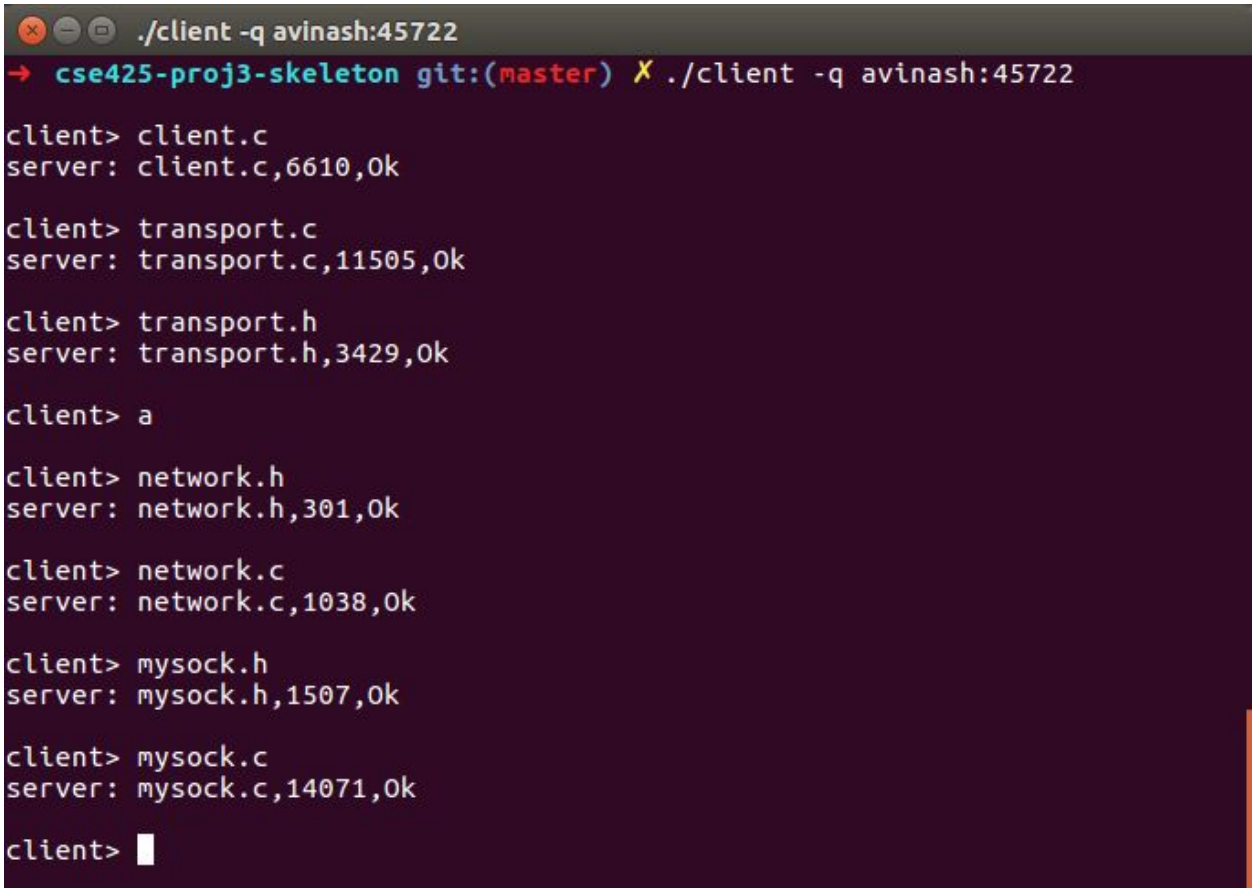
Avinash Mohak, 13177 (amohak@iitk.ac.in)

Implemented Options:

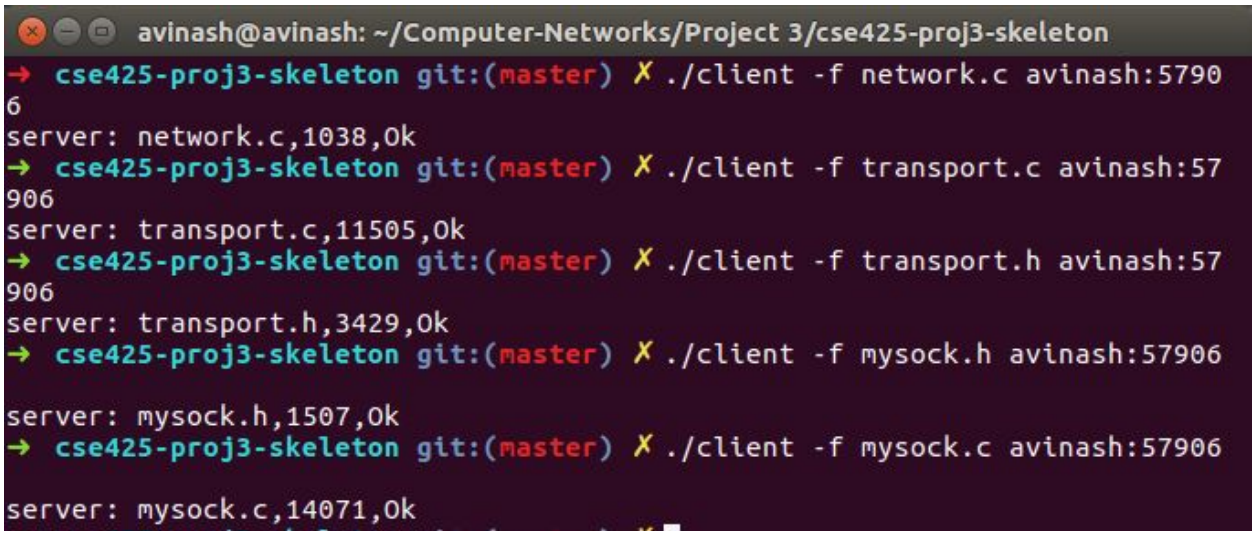
- Network Initiation (3-way handshake)
- Network Termination(4-way tear down)
- Sliding Window for Data transfer

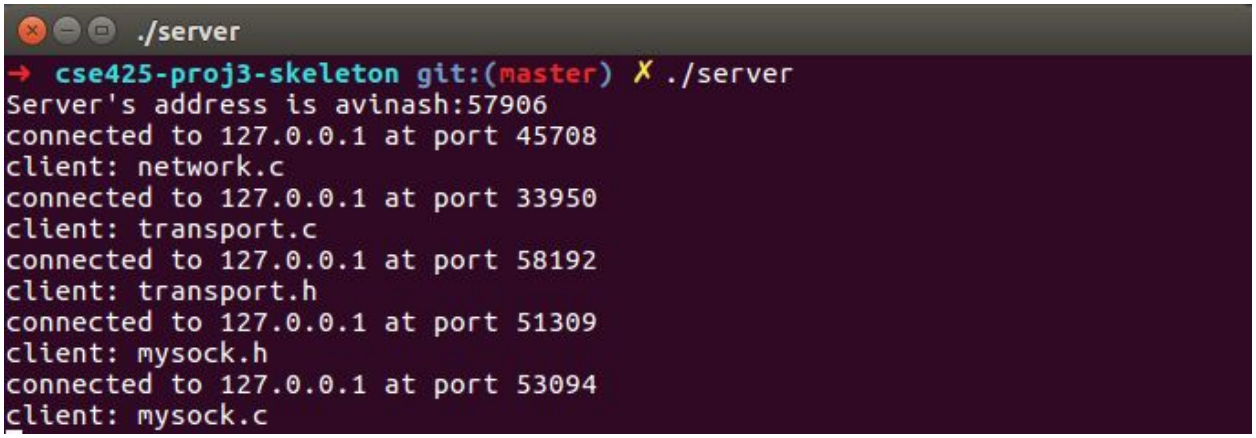
Testing Results:

1. The given client and server were used for testing purposes. Successful multiple file transfer between them in interactive mode. (In the given snapshot, output has been suppressed for clarity).



2. The termination was tested with the *-f* option, which ran successfully, multiple times.





Summary:

1. The STCP (handshake, file transfer, and termination(or tear-down) works properly to the best of my knowledge.
2. The endianness/compatibility issue was handled correctly.
3. All states(according to the TCP state-diagram(RFC 793)), except TIME_WAIT, were implemented.
4. The server seems to misbehave when the client is closed with Ctrl-C instead of Ctrl-D(EOF).
5. Not much could be done for error handling, except a few diagnostic messages.

Appendix:

Source Code:
/*

```
 * transport.c
 *
 *        Project 3
 *
 * This file implements the STCP layer that sits between the
 * mysocket and network layers. You are required to fill in the STCP
 * functionality in this file.
 *
 */


#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <arpa/inet.h>
#include "mysock.h"
#include "stcp_api.h"
#include "transport.h"


#define MaxWindowSize 3072

void h_to_n(STCPHeader *Header)
{
        Header->th_seq = htonl(Header->th_seq);
        Header->th_ack = htonl(Header->th_ack);
        Header->th_win = htons(Header->th_win);
        Header->th_sum = htons(Header->th_sum);
        Header->th_urp = htons(Header->th_urp);
}

void n_to_h(STCPHeader *Header)
{
        Header->th_seq = ntohl(Header->th_seq);
        Header->th_ack = ntohl(Header->th_ack);
        Header->th_win = ntohs(Header->th_win);
        Header->th_sum = ntohs(Header->th_sum);
        Header->th_urp = ntohs(Header->th_urp);
}

enum {          CSTATE_ESTABLISHED = 1,
```

```
                LISTEN = 2,
                SYN_SENT = 3,
                SYN_RECD = 4,
                FIN_WAIT_1 = 5,
                FIN_WAIT_2 = 6,
                CLOSING = 7,
                CLOSE_WAIT = 8,
                LAST_ACK = 9,
                CLOSED = 0
        };    /* you should have more states */  /* I have it already */


/* this structure is global to a mysocket descriptor */
typedef struct
{
        bool_t done;    /* TRUE once connection is closed */

        int connection_state;   /* state of the connection (established, etc.) */
        tcp_seq initial_sequence_num;

        tcp_seq next_byte_expected;
        tcp_seq last_byte_acked;
        tcp_seq next_sequence_num;
        tcp_seq fin_seq_num;

        int AdvertisedWindowSize;
        /* any other connection-wide global variables go here */
} context_t;


static void generate_initial_seq_num(context_t *ctx);
static void control_loop(mysocket_t sd, context_t *ctx);


/* initialise the transport layer, and start the main loop, handling
 * any data from the peer or the application.  this function should not
 * return until the connection is closed.
 */
void transport_init(mysocket_t sd, bool_t is_active)
{
        context_t *ctx;

        ctx = (context_t *) calloc(1, sizeof(context_t));
```

```
assert(ctx);

generate_initial_seq_num(ctx);
ctx->AdvertisedWindowSize = MaxWindowSize;
ctx->next_sequence_num = ctx->initial_sequence_num;
ctx->connection_state = CLOSED;

STCPHeader *synPacket = (STCPHeader *) calloc(1,sizeof(STCPHeader));
STCPHeader *synAckPacket = (STCPHeader *) calloc(1,sizeof(STCPHeader));
STCPHeader *ackPacket = (STCPHeader *) calloc(1,sizeof(STCPHeader));
STCPHeader *inPacket = (STCPHeader *) calloc(1,sizeof(STCPHeader));



while(ctx->connection_state == CLOSED)
{
        if(is_active)
        {
                synPacket->th_seq = ctx->initial_sequence_num;
                synPacket->th_off = 5;
                synPacket->th_flags = TH_SYN;

                h_to_n(synPacket);
                stcp_network_send(sd,synPacket,sizeof(STCPHeader),NULL);

                ctx->connection_state = SYN_SENT;
                ctx->next_sequence_num++;

                unsigned int event_flag = stcp_wait_for_event(sd,ANY_EVENT,NULL);

                if(event_flag == NETWORK_DATA)
                {
                        bool_t simultaneous_open = false;

                        stcp_network_recv(sd,inPacket,sizeof(STCPHeader));
                        n_to_h(inPacket);
                        // if(synAckPacket->th_ack == ctx->initial_sequence_num + 1)

                        if((inPacket->th_flags ^ TH_SYN) == 0)
                        {
                                simultaneous_open = true;
                                ctx->connection_state = SYN_RECD;
```

```
                                synAckPacket->th_ack = inPacket->th_seq + 1;
                                synAckPacket->th_seq = ctx->initial_sequence_num;
                                synAckPacket->th_off = 5;
                                synAckPacket->th_flags = (TH_SYN | TH_ACK);

                                h_to_n(synAckPacket);

stcp_network_send(sd,synAckPacket,sizeof(STCPHeader),NULL);

                                unsigned int event_flag =
stcp_wait_for_event(sd,ANY_EVENT,NULL);

                                if(event_flag == NETWORK_DATA)
                                {

stcp_network_recv(sd,inPacket,sizeof(STCPHeader));
                                        n_to_h(inPacket);
                                        simultaneous_open = false;
                                }
                                else
                                {
                                        //error handling : Non-network data
                                }
                        }

                        if(((inPacket->th_flags ^ (TH_SYN | TH_ACK)) == 0) &&
!simultaneous_open)
                        {
                                ctx->next_byte_expected = inPacket->th_seq + 1;

                                ackPacket->th_seq = ctx->initial_sequence_num + 1;
                                ackPacket->th_ack = inPacket->th_seq + 1;
                                ackPacket->th_off = 5;
                                ackPacket->th_flags = TH_ACK;

                                ctx->last_byte_acked = inPacket->th_ack;

                                h_to_n(ackPacket);

stcp_network_send(sd,ackPacket,sizeof(STCPHeader),NULL);
                        }
                        else
                        {
```

```
                                        //error handling
                                }
                        }
                        else
                        {
                                //error handling : Non-network data
                        }
                }

                else
                {
                        ctx->connection_state = LISTEN;

                        stcp_network_recv(sd,synPacket,sizeof(STCPHeader));
                        n_to_h(synPacket);
                        if(synPacket->th_flags ^ TH_SYN)
                        {
                                //error handling
                        }
                        else
                        {
                                ctx->next_sequence_num++;

                                synAckPacket->th_ack = synPacket->th_seq + 1;
                                synAckPacket->th_seq = ctx->initial_sequence_num;
                                synAckPacket->th_off = 5;
                                synAckPacket->th_flags = (TH_SYN | TH_ACK);

                                h_to_n(synAckPacket);

stcp_network_send(sd,synAckPacket,sizeof(STCPHeader),NULL);

                                ctx->connection_state = SYN_RECD;

                                unsigned int event_flag =
stcp_wait_for_event(sd,ANY_EVENT,NULL);

                                if(event_flag == NETWORK_DATA)
                                {
                                        stcp_network_recv(sd,ackPacket,sizeof(STCPHeader));
                                        n_to_h(ackPacket);
```

```
                                        if(ackPacket->th_flags != TH_ACK || ackPacket->th_ack !=
ctx->initial_sequence_num + 1)
                                        {
                                                //error handling
                                        }
                                        else
                                        {
                                                ctx->next_byte_expected = ackPacket->th_seq;
                                                ctx->last_byte_acked = ackPacket->th_ack;
                                        }
                                }
                                else
                                {
                                        // Non-Network Data
                                }
                        }
                }
        }
        /* XXX: you should send a SYN packet here if is_active, or wait for one
         * to arrive if !is_active.  after the handshake completes, unblock the
         * application with stcp_unblock_application(sd).  you may also use
         * this to communicate an error condition back to the application, e.g.
         * if connection fails; to do so, just set errno appropriately (e.g. to
         * ECONNREFUSED, etc.) before calling the function.
         */

        ctx->connection_state = CSTATE_ESTABLISHED;
        stcp_unblock_application(sd);

        control_loop(sd, ctx);

        /* do any cleanup here */
        free(ctx);
        free(synPacket);
        free(synAckPacket);
        free(inPacket);
        free(ackPacket);
}


/* generate random initial sequence number for an STCP connection */
static void generate_initial_seq_num(context_t *ctx)
{
```

```
	assert(ctx);

#ifdef FIXED_INITNUM
	/* please don't change this! */
	ctx->initial_sequence_num = 1;
#else
	/* you have to fill this up */
	ctx->initial_sequence_num = rand() % 256;
#endif
}


/* control_loop() is the main STCP loop; it repeatedly waits for one of the
 * following to happen:
 *   - incoming data from the peer
 *   - new data from the application (via mywrite())
 *   - the socket to be closed (via myclose())
 *   - a timeout
 */
static void control_loop(mysocket_t sd, context_t *ctx)
{
	assert(ctx);
	assert(!ctx->done);

	while (!ctx->done)
	{
		char sendBuffer[MaxWindowSize], recvBuffer[MaxWindowSize];

		unsigned int event;

		/* see stcp_api.h or stcp_api.c for details of this function */
		/* XXX: you will need to change some of these arguments! */
		event = stcp_wait_for_event(sd, ANY_EVENT, NULL);

		/* check whether it was the network, app, or a close request */
		if (event & APP_DATA)
		{
			int EffectiveWindow = ctx->AdvertisedWindowSize -
(ctx->next_sequence_num - ctx->last_byte_acked);
			if(EffectiveWindow > 0)
			{
				int app_rcvd =
stcp_app_recv(sd,sendBuffer,MIN(EffectiveWindow,STCP_MSS));
```

```
                                if(app_rcvd > 0)
                                {
                                        STCPHeader *DataPacketHeader = (STCPHeader *)
calloc(1,sizeof(STCPHeader));

                                        DataPacketHeader->th_seq = ctx->next_sequence_num;
                                        DataPacketHeader->th_ack = ctx->next_byte_expected;
                                        DataPacketHeader->th_off = 5;
                                        DataPacketHeader->th_win = MaxWindowSize;

                                        ctx->next_sequence_num += app_rcvd;

                                        h_to_n(DataPacketHeader);

stcp_network_send(sd,DataPacketHeader,sizeof(STCPHeader),sendBuffer,app_rcvd,NULL);
                                        free(DataPacketHeader);
                                }
                        }
                }

                if(event & NETWORK_DATA)
                {
                        int recvBufferSize =
stcp_network_recv(sd,recvBuffer,MaxWindowSize);
                        if(recvBufferSize>0)
                        {
                                STCPHeader *DataPacketHeader = (STCPHeader *)
calloc(1,sizeof(STCPHeader));
                                STCPHeader *ackPacketHeader = (STCPHeader *)
calloc(1,sizeof(STCPHeader));

                                int offset = TCP_DATA_START(recvBuffer);
// Header size

                                memcpy(DataPacketHeader,recvBuffer,sizeof(STCPHeader));
                                n_to_h(DataPacketHeader);

                                if(DataPacketHeader->th_flags & TH_ACK)
                                {
                                        ctx->last_byte_acked = DataPacketHeader->th_ack;

                                        if(DataPacketHeader->th_ack == ctx->fin_seq_num + 1)
                                        {
```

```
                                        if(ctx->connection_state == FIN_WAIT_1)
                                                ctx->connection_state = FIN_WAIT_2;
                                        if((ctx->connection_state == LAST_ACK) ||
(ctx->connection_state == CLOSING))
                                        {
                                                ctx->connection_state = CLOSED;
                                                ctx->done = TRUE;
                                        }
                                }
                        }

                        int data_size = recvBufferSize - offset;
                        if(data_size > 0)
                        {
                                if(DataPacketHeader->th_seq ==
ctx->next_byte_expected)
                                {
                                        ctx->next_byte_expected =
ctx->next_byte_expected + data_size;
                                        stcp_app_send(sd,recvBuffer+offset,data_size);
                                }

                                else if(DataPacketHeader->th_seq <
ctx->next_byte_expected)
                                {
                                        int diff = ctx->next_byte_expected -
DataPacketHeader->th_seq;
                                        if(diff < data_size)
                                        {
                                                offset += diff;
                                                ctx->next_byte_expected =
ctx->next_byte_expected + data_size - diff;

stcp_app_send(sd,recvBuffer+offset,data_size - diff);
                                        }
                                }
                                else
                                {
                                        //error handling -- Packet loss probably -- not to
be handled
                                }

                        }
```

```
                        if(DataPacketHeader->th_flags & TH_FIN)
                        {
                                stcp_fin_received(sd);

                                ctx->next_byte_expected++;
                                if(ctx->connection_state == CSTATE_ESTABLISHED)
                                        ctx->connection_state = CLOSE_WAIT;
                                if(ctx->connection_state == FIN_WAIT_1)
                                        ctx->connection_state = CLOSING;
                                if(ctx->connection_state == FIN_WAIT_2)
                                {
                                        ctx->connection_state = CLOSED;
                                        ctx->done = TRUE;
                                }
                        }

                        ackPacketHeader->th_seq = ctx->next_sequence_num;
                        ackPacketHeader->th_ack = ctx->next_byte_expected;
                        ackPacketHeader->th_flags = TH_ACK;
                        ackPacketHeader->th_off = 5;
                        ackPacketHeader->th_win = MaxWindowSize;

                        h_to_n(ackPacketHeader);

stcp_network_send(sd,ackPacketHeader,sizeof(STCPHeader),NULL);
                        }
                }

                if(event & APP_CLOSE_REQUESTED)
                {
                        STCPHeader *finPacket = (STCPHeader *)
calloc(1,sizeof(STCPHeader));
                        finPacket->th_seq = ctx->next_sequence_num;
                        finPacket->th_ack = ctx->next_byte_expected;
                        finPacket->th_flags = TH_FIN;
                        finPacket->th_off = 5;
                        finPacket->th_win = MaxWindowSize;

                        h_to_n(finPacket);
                        stcp_network_send(sd,finPacket,sizeof(STCPHeader),NULL);

                        ctx->fin_seq_num = ctx->next_sequence_num;
```

```
                        ctx->next_sequence_num++;

                        if(ctx->connection_state==CSTATE_ESTABLISHED)
                                ctx->connection_state = FIN_WAIT_1;
                        else if(ctx->connection_state == CLOSE_WAIT)
                                ctx->connection_state = LAST_ACK;
                }

        }

}


/*********************************************************************/
/* our_dprintf
 *
 * Send a formatted message to stdout.
 *
 * format           A printf-style format string.
 *
 * This function is equivalent to a printf, but may be
 * changed to log errors to a file if desired.
 *
 * Calls to this function are generated by the dprintf amd
 * dperror macros in transport.h
 */
void our_dprintf(const char *format,...)
{
        va_list argptr;
        char buffer[1024];

        assert(format);
        va_start(argptr, format);
        vsnprintf(buffer, sizeof(buffer), format, argptr);
        va_end(argptr);
        fputs(buffer, stdout);
        fflush(stdout);
}
```