



PROJECT 3 : COMPUTER NETWORKS (CS425A)

TCP PROTOCOL IMPLEMENTATION

30.09.2016

PRAMOD CHUNDURI

ROLL NO 13221

chpramod@iitk.ac.in

IMPLEMENTED FEATURES :

3-WAY HAND SHAKE

4 WAY CLOSING OF CONNECTION

SUPPORT ON DIFFERENT OS (hton and ntohs)

SLIDING WINDOW

Overview of Features

3-WAY HAND SHAKE

3-Way hand shake is implemented as per the TCP protocol. Active side sends a SYN packet, Passive replies with a SYN_ACK packet, and Active side then acknowledges this packet.

4 WAY CLOSING OF CONNECTION

A 4 way closing of connection is implemented. One side sends a FIN packet, the other side acknowledges and once the other side also sends the FIN packet, the receiving side closes the connection after sending an acknowledgement. A call to close(sd) always closes the connection smoothly and the test server keeps running flawlessly. Also, the case when Ctrl+C is pressed as an end of connection is also handled.

SUPPORT ON DIFFERENT OS

The code can be run on and receive packets from hosts with any byte orders, as care has been taken to change the STCP headers to a universal format ().

SLIDING WINDOW

The sender side sliding window has been implemented by taking two pointers, left and right, and ensuring that the sequence number of the current packet is within the limits. The limits are appropriately changed whenever an acknowledgement packet arrives.

Testing and Design features

All the functionalities implemented have been verified for on the provided reference server and client for ideal behavior.

All the edges cases have been tested, like closing the server after data is received, repeatedly sending data over the same connection, connecting multiple clients to the server etc.

“Errno” has been set at appropriate places, to check for errors like invalid packets, incorrect handshake etc.

The possibility of receiving a FIN+DATA packet has been handled.

The working of the sliding window has been tested by deliberately dropping the acknowledgement packets and printing the response.

Appendix (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"
#include <limits.h>
// #include
uint8_t MSS = 255;
```

```
enum { CSTATE_ESTABLISHED }; /* you should have more states */
```

```
/* 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;

    /* 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;
    int errno;
    ctx = (context_t *) calloc(1, sizeof(context_t));
    assert(ctx);

    generate_initial_seq_num(ctx);
```

```

/* 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.
 */

//If this is the active side of connection
if (is_active){
    //3 way handshake rules for the active side (client side in our example)
    tcp_seq curr_active_seq = ctx->initial_sequence_num;
    tcp_seq curr_passive_seq;
    STCPHeader *syn = (STCPHeader *)malloc(sizeof(struct tcphdr));
    STCPHeader *syn_ack = (STCPHeader *)malloc(sizeof(struct tcphdr));
    STCPHeader *ack = (STCPHeader *)malloc(sizeof(struct tcphdr));
    syn->th_seq = curr_active_seq;
    syn->th_off = sizeof(STCPHeader)/4;
    syn->th_seq = htonl(syn->th_seq);
    syn->th_ack = htonl(syn->th_ack);
    syn->th_flags = TH_SYN;
    //Send SYN packet to network
    ssize_t syn_send = stcp_network_send(sd,syn,sizeof(struct tcphdr),NULL);
    syn->th_seq = ntohl(syn->th_seq);
    syn->th_ack = ntohl(syn->th_ack);
    if (syn_send <= 0){
        errno = 111;
    }
    //Receive SYN_ACK packet

```

```
ssize_t syn_ack_rcv = stcp_network_rcv(sd,syn_ack,sizeof(struct tcphdr));
syn_ack->th_seq = ntohl(syn_ack->th_seq);
syn_ack->th_ack = ntohl(syn_ack->th_ack);
if (syn_ack_rcv <= 0){
    errno = 111;
}
if (syn_ack->th_flags != (TH_SYN | TH_ACK)){
    errno = 111;
}
if (syn_ack->th_ack != (syn->th_seq+1)){
    errno = 111;
}
//Send acknowledgement packet for SYN_ACK received
curr_passive_seq = syn_ack->th_seq;
curr_passive_seq++;
ack->th_seq = curr_active_seq+1;
ack->th_ack = curr_passive_seq;
ack->th_flags = TH_ACK;
ack->th_off = sizeof(STCPHeader)/4;
ack->th_seq = htonl(ack->th_seq);
ack->th_ack = htonl(ack->th_ack);
ssize_t ack_send = stcp_network_send(sd,ack,sizeof(struct tcphdr),NULL);
if (ack_send == -1){
    errno = 111;
}
}
else{
    //3 way handshake rules for the passive side (server side in our example)
    tcp_seq curr_active_seq;
```

```
tcp_seq curr_passive_seq = ctx->initial_sequence_num;
STCPHeader *syn = (STCPHeader *)malloc(sizeof(struct tcphdr));
STCPHeader *syn_ack = (STCPHeader *)malloc(sizeof(struct tcphdr));
STCPHeader *ack = (STCPHeader *)malloc(sizeof(struct tcphdr));
//Receive Syn packet
ssize_t syn_recv = stcp_network_recv(sd,syn,sizeof(struct tcphdr));
syn->th_seq = ntohl(syn->th_seq);
syn->th_ack = ntohl(syn->th_ack);
fflush(stdout);
if (syn_recv <= 0){
    errno = 111;
}
if (syn->th_flags != TH_SYN){
    errno = 111;
}
curr_active_seq = syn->th_seq;
curr_active_seq++;
syn_ack->th_seq = curr_passive_seq;
syn_ack->th_ack = curr_active_seq;
syn_ack->th_flags = TH_ACK | TH_SYN;
syn_ack->th_off = sizeof(STCPHeader)/4;
syn_ack->th_seq = htonl(syn_ack->th_seq);
syn_ack->th_ack = htonl(syn_ack->th_ack);
//send syn_ack packet
ssize_t syn_ack_send = stcp_network_send(sd,syn_ack,sizeof(struct tcphdr),NULL);
if (syn_ack_send == -1){
    errno = 111;
}
//receive ack packet
```



```
ssize_t ack_rcv = stcp_network_rcv(sd,ack,sizeof(struct tcphdr));
ack->th_seq = ntohl(ack->th_seq);
ack->th_ack = ntohl(ack->th_ack);
fflush(stdout);
if (ack_rcv == -1){
    errno = 111;
}
if (ack->th_flags != TH_ACK){
    errno = 111;
}
if (ack->th_ack != (curr_passive_seq+1) ){
    errno = 111;
}
}
ctx->connection_state = CSTATE_ESTABLISHED;
stcp_unblock_application(sd);

control_loop(sd, ctx);

/* do any cleanup here */
free(ctx);
}

/* 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);
    tcp_seq curr_seq_number = ctx->initial_sequence_num + 1;
    tcp_seq curr_ack_number;
    tcp_seq left=curr_seq_number;
    tcp_seq right = left+3072;
    tcp_seq curr_window;
    int my_fin = 0;
    int opp_fin = 0;
    int fin_ack = 0;
```

```
unsigned int SW_FLAG = 0;
while (!ctx->done)
{
    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 ^ SW_FLAG), NULL);
    /* check whether it was the network, app, or a close request */
    if (event & APP_DATA){
        //receive data from the app
        char buffer[536];
        //if the sequence number of the data to be sent is within the limits
        if (curr_seq_number >= left && curr_seq_number < right){
            size_t app_rcv_size;
            if ((right - curr_seq_number) < sizeof(buffer))
                app_rcv_size = stcp_app_rcv(sd, buffer, right - curr_seq_number);
            else
                app_rcv_size = stcp_app_rcv(sd, buffer, sizeof(buffer));
            STCPHeader *header = (STCPHeader *)malloc(sizeof(STCPHeader));
            header->th_seq = curr_seq_number;
            header->th_flags = 0;
            header->th_off = sizeof(STCPHeader)/4;
            header->th_seq = htonl(header->th_seq);
            char *packet = (char *)malloc(sizeof(STCPHeader) + app_rcv_size);
            memcpy(packet, header, sizeof(STCPHeader));
            memcpy(packet + sizeof(STCPHeader), buffer, app_rcv_size);
            stcp_network_send(sd, packet, sizeof(STCPHeader) + app_rcv_size, NULL);
            curr_seq_number += app_rcv_size;
        }
    }
}
```

```
}
//if seq.number of data out of limits
else{
    SW_FLAG = 1;
}
/* the application has requested that data be sent */
/* see stcp_app_rcv() */
}
if (event & NETWORK_DATA){
    //receive fixed size buffer(packet) from the network
    char buffer[556];
    ssize_t recvd_packet_size = stcp_network_rcv(sd,buffer,sizeof(buffer));
    if (recvd_packet_size == 0)
        return;
    //separate the header from the packet
    STCPHeader *header = (STCPHeader *)malloc(sizeof(STCPHeader));
    memcpy(header,buffer,sizeof(STCPHeader));
    header->th_seq = ntohl(header->th_seq);
    header->th_ack = ntohl(header->th_ack);
    header->th_win = ntohs(header->th_win);
    //if the received packet is not an acknowledgement
    if (header->th_flags != TH_ACK){
        size_t data_size = recvd_packet_size-(TCP_DATA_START(buffer));
        char *data;
        //if not an empty packet
        if (data_size!=0)
            data = (char *)malloc(data_size);
        //if it is an empty packet
        else{
```

```
if (header->th_flags != TH_FIN){
    errno = 74; //Not a data message - EBADMSG
    break;
}
else{
    //the given packet is FIN, so tell app that fin is received
    stcp_fin_received(sd);
    opp_fin = 1;
    curr_ack_number = header->th_seq + 1;
    STCPHeader *ack_packet = (STCPHeader *)malloc(sizeof(STCPHeader));
    ack_packet->th_seq = header->th_seq + 1;
    ack_packet->th_ack = curr_ack_number;
    ack_packet->th_win = 3072;
    ack_packet->th_off = sizeof(STCPHeader)/4;
    ack_packet->th_win = htons(ack_packet->th_win);
    ack_packet->th_seq = htonl(ack_packet->th_seq);
    ack_packet->th_ack = htonl(ack_packet->th_ack);
    ack_packet->th_flags = TH_ACK;
    //send acknowledgement for FIN
    stcp_network_send(sd,ack_packet,sizeof(STCPHeader),NULL);
    if (fin_ack == 1)
        return;
    else
        continue;
}
}

//send acknowledgement for the received packet
curr_ack_number = header->th_seq + data_size;
STCPHeader *ack_packet = (STCPHeader *)malloc(sizeof(STCPHeader));
```

```
ack_packet->th_off = sizeof(STCPHeader)/4;
ack_packet->th_seq = header->th_seq + 1;
ack_packet->th_ack = curr_ack_number;
ack_packet->th_win = 3072;
ack_packet->th_win = htons(ack_packet->th_win);
ack_packet->th_seq = htonl(ack_packet->th_seq);
ack_packet->th_ack = htonl(ack_packet->th_ack);
ack_packet->th_flags = TH_ACK;
stcp_network_send(sd,ack_packet,sizeof(STCPHeader),NULL);
//and send data to the application layer
memcpy(data,buffer+sizeof(STCPHeader),data_size);
stcp_app_send(sd,data,data_size);
//if it is a FIN+DATA packet
if (header->th_flags == TH_FIN){
    stcp_fin_received(sd);
    opp_fin = 1;
    curr_ack_number = header->th_seq + 1;
    STCPHeader *ack_packet = (STCPHeader *)malloc(sizeof(STCPHeader));
    ack_packet->th_off = sizeof(STCPHeader)/4;
    ack_packet->th_seq = header->th_seq + 1;
    ack_packet->th_ack = curr_ack_number;
    ack_packet->th_win = 3072;
    ack_packet->th_win = htons(ack_packet->th_win);
    ack_packet->th_seq = htonl(ack_packet->th_seq);
    ack_packet->th_ack = htonl(ack_packet->th_ack);
    ack_packet->th_flags = TH_ACK;
    stcp_network_send(sd,ack_packet,sizeof(STCPHeader),NULL);
}
}
```

```

//Given packet is an acknowledgement packet
else{
    SW_FLAG = 0;
    curr_window = header->th_win;
    left = header->th_ack;
    right = left + curr_window;

    //code for handling the 4 way hand shake (my_fin and opp_fin as network states
w.r.t reception of fin packets)
    if (my_fin==1){
        if (curr_seq_number==header->th_ack-1){
            if (opp_fin==1)
                return;
            else
                fin_ack = 1;
        }

    }
}


// if there is a close request
if (event & APP_CLOSE_REQUESTED){
    STCPHeader *fin = (STCPHeader *)malloc(sizeof(STCPHeader));
    fin->th_off = sizeof(STCPHeader)/4;
    fin->th_flags = TH_FIN;
    fin->th_seq = curr_seq_number;
    fin->th_seq = htonl(fin->th_seq);
    my_fin = 1;
    //send a FIN packet
    stcp_network_send(sd,fin,sizeof(STCPHeader),NULL);
}

```

```
    /* etc. */
}
}

/*****
/* 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);  
}
```