CS425 - Project 3 Report

Nishant Gupta, nishgu@iitk.ac.in, 13447

September 30, 2016

STCP Protocol

1 Implementation Specs

- Sliding window is implemented
- Proper endianness handling is insured
- Abrupt disconnects are handled
- Proper errno is set

2 Test results

- All Combinations of two clients(reference client and client.c in skeleton) and two server(reference server and server.c in skeleton) are working properly
- The client exits cleanly with `Ctrl+D` and `Ctrl+C` interrupts and the server with `Ctrl+C` interrupt.
- Single file transfer with `-f` flag is working properly
- Multiple client connections are working fine with one client queued after other

3 Appendix

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

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;
  tcp_seq initial_sequence_num_peer;
  tcp_seq current_sequence_num_peer;
  tcp_seq current_sequence_num;
```

```
  tcp_seq sender_window, unacked, congestion_window,
      last_acked;
  /* 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);


inline void hton(STCPHeader* header){
  header->th_seq = htonl(header->th_seq);
  header->th_ack = htonl(header->th_ack);
  header->th_win = htons(header->th_win);
}

inline void ntoh(STCPHeader* header){
  header->th_seq = ntohl(header->th_seq);
  header->th_ack = ntohl(header->th_ack);
  header->th_win = ntohs(header->th_win);
}
/* 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 bytes_recieved, send_status;
  ctx = (context_t *)calloc(1, sizeof(context_t));
  assert(ctx);

  generate_initial_seq_num(ctx);
  STCPHeader *hdr = (STCPHeader *)calloc(1, sizeof(STCPHeader));
  /* 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 (is_active) {
    /**
    printf("Active Connection initiation request\n");
    */

    // send SYN packet
    hdr->th_flags = TH_SYN;
    hdr->th_seq = ctx->current_sequence_num++;
    hdr->th_off = sizeof(STCPHeader)/4;

    hton(hdr);
    send_status = stcp_network_send(sd, hdr, sizeof(STCPHeader), NULL);
    if(send_status == -1) {
      errno = ECONNREFUSED;
      return;
    }
```

```
  // recv and process SYN+ACK packet
  bytes_recieved = stcp_network_recv(sd, hdr, sizeof(STCPHeader));
  if(bytes_recieved == 0) {
    errno = ECONNREFUSED;
    return;
  }
  ntoh(hdr);
  assert(hdr->th_flags == (TH_SYN | TH_ACK));
  assert(ctx->current_sequence_num == hdr->th_ack);
  ctx->initial_sequence_num_peer = hdr->th_seq;
  ctx->current_sequence_num_peer = ctx->initial_sequence_num_peer + 1;

  // send ACK packet
  memset(hdr, 0, sizeof(STCPHeader));
  hdr->th_flags = TH_ACK;
  hdr->th_seq = ctx->current_sequence_num;
  hdr->th_off = sizeof(STCPHeader)/4;
  hdr->th_ack = ctx->current_sequence_num_peer;
  hton(hdr);
  send_status = stcp_network_send(sd, hdr, sizeof(STCPHeader), NULL);
  if(send_status == -1) {
    errno = ECONNABORTED;
    return;
  }


  // Handshake complete
} else {
  /**
  printf("Passive Connection initiation request\n");
  */
  // recieve SYN packet
  bytes_recieved = stcp_network_recv(sd, hdr, sizeof(STCPHeader));
  if(bytes_recieved == 0) {
    errno = ECONNABORTED;
    return;
  }
  ntoh(hdr);
  ctx->initial_sequence_num_peer = hdr->th_seq;
  ctx->current_sequence_num_peer = ctx->initial_sequence_num_peer + 1;

  // send SYN+ACK packet
  memset(hdr, 0, sizeof(STCPHeader));
  hdr->th_flags = TH_ACK | TH_SYN;
  hdr->th_seq = ctx->current_sequence_num++;
  hdr->th_off = sizeof(STCPHeader)/4;
  hdr->th_ack = ctx->current_sequence_num_peer;
  hton(hdr);
  send_status = stcp_network_send(sd, hdr, sizeof(STCPHeader), NULL);
  if(send_status == -1) {
    errno = ECONNABORTED;
    return;
  }
```

```
    // recieve ACK
    stcp_network_recv(sd, hdr, sizeof(STCPHeader));
    if(bytes_recieved == 0) {
      errno =  ECONNABORTED;
      return;
    }
    ntoh(hdr);

    assert(hdr->th_flags & TH_ACK);
    assert(ctx->current_sequence_num == hdr->th_ack);
  }
  ctx->connection_state = CSTATE_ESTABLISHED;
  /**/
  //printf("Handshake Completed\n"); /**/
  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 = 1; // testing

  ctx->initial_sequence_num = rand() % 256;
  ctx->current_sequence_num = ctx->initial_sequence_num;
#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);
  unsigned int event;
  bool_t this_end_closed = FALSE, other_end_closed = FALSE;
  STCPHeader *hdr = (STCPHeader *)calloc(1, sizeof(STCPHeader));
  int off = sizeof(STCPHeader) / 4 + ((sizeof(STCPHeader) % 4) ? 1 : 0);
  ctx->unacked = 0;
  ctx->sender_window = 3072;
  ctx->congestion_window = 3072;
```

```
int send_status;
void *buffer = (void *)malloc(STCP_MSS + 4 * off);
while (!ctx->done) {

  event = stcp_wait_for_event(sd, ANY_EVENT, NULL);

  if ((event & APP_DATA) && !this_end_closed) {

    int data_limit = MIN(STCP_MSS, ctx->sender_window - ctx->unacked);

    hdr->th_seq = ctx->current_sequence_num;
    hdr->th_off = off;
    hdr->th_flags = 0;
    hdr->th_win = ctx->congestion_window;

    int bytes_to_send = stcp_app_recv(sd, buffer, data_limit);
    hton(hdr);
    send_status = stcp_network_send(sd, hdr, sizeof(STCPHeader), buffer, bytes_to_send,
                      NULL);
    if(send_status == -1) {
      errno = ECONNABORTED;
      return;
    }
    ctx->unacked += bytes_to_send;
    ctx->current_sequence_num += bytes_to_send;
  }

  if (event & NETWORK_DATA) {
    int bytes_received = stcp_network_recv(sd, buffer, STCP_MSS + off * 4);
    if(bytes_received == 0) {
      errno = ECONNABORTED;
      ctx->done = TRUE;
      continue;
    }

    memcpy((void *)hdr, buffer, sizeof(STCPHeader));
    ntoh(hdr);
    ctx->sender_window = MIN(hdr->th_win, ctx->congestion_window);

    if ((hdr->th_flags & TH_ACK) && ctx->last_acked < hdr->th_ack - 1) {
      ctx->last_acked = hdr->th_ack - 1;
      ctx->unacked = ctx->current_sequence_num - hdr->th_ack;
      ctx->sender_window = MIN(hdr->th_win, ctx->congestion_window);
      if(this_end_closed) { // we won't send any data after this
                            // so this should hold
        assert(hdr->th_ack <= ctx->current_sequence_num + 1);
      }
    }

    if (bytes_received - off * 4 > 0) {
      if(ctx->current_sequence_num_peer > hdr->th_seq + bytes_received - off * 4){
        // data is duplicate but send new ack
        hdr->th_ack = hdr->th_seq + bytes_received - off * 4;
        hdr->th_flags = TH_ACK;
        hdr->th_win = ctx->congestion_window;
```

```
      hdr->th_off = off;
      hdr->th_seq = ctx->current_sequence_num;

      hton(hdr);
      send_status = stcp_network_send(sd, hdr, off * 4, NULL);
    } else {
      int new_data_offset = 0;
      new_data_offset = ctx->current_sequence_num_peer - hdr->th_seq;
      ctx->current_sequence_num_peer = hdr->th_seq + bytes_received - off * 4;
      hdr->th_ack = ctx->current_sequence_num_peer;
      hdr->th_flags = TH_ACK;
      hdr->th_win = ctx->congestion_window;
      hdr->th_off = off;
      hdr->th_seq = ctx->current_sequence_num;

      hton(hdr);
      send_status = stcp_network_send(sd, hdr, off * 4, NULL);
      stcp_app_send(sd, (char *)buffer + off * 4 + new_data_offset, bytes_received - off * 4 - new_
    }
    if(send_status == -1) {
      errno = ECONNABORTED;
      return;
    }
  }


  if (hdr->th_flags & TH_FIN) {
    ctx->current_sequence_num_peer++; // FIN takes one Segment space
    //ACK for FIN
    memset(hdr, 0, sizeof(STCPHeader));
    hdr->th_ack = ctx->current_sequence_num_peer;
    hdr->th_seq = ctx->current_sequence_num;
    hdr->th_flags = TH_ACK;
    hdr->th_win = ctx->congestion_window;
    hdr->th_off = off;

    hton(hdr);
    send_status = stcp_network_send(sd, hdr, off * 4, NULL);
    if(send_status == -1) {
      errno = ECONNABORTED;
      return;
    }
    other_end_closed = TRUE;
    stcp_fin_received(sd);
  }

}


if(event & APP_CLOSE_REQUESTED) {
  //printf("App close requested\n");
  // send FIN
  memset(hdr, 0, sizeof(STCPHeader));
  hdr->th_seq = ctx->current_sequence_num++;
  hdr->th_flags = TH_FIN;
```

```
      hdr->th_off = sizeof(STCPHeader)/4;
      hdr->th_win = ctx->congestion_window;

      hton(hdr);
      send_status = stcp_network_send(sd, hdr, off * 4, NULL);
      if(send_status == -1) {
        errno = ECONNABORTED;
        return;
      }


      this_end_closed = TRUE;
    }
    if (event & TIMEOUT) {
      ctx->done = TRUE;
    }

    if(this_end_closed && other_end_closed) ctx->done = TRUE;
    /* etc. */
  }
  free(hdr);
  free(buffer);
}

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