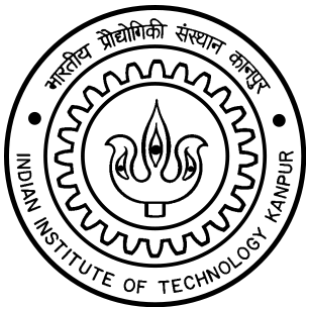
Indian Institute of Technology Kanpur

CS425 - Computer Networks

Project 3

Nishant Rai, 13449



Course Instructor
Dr. Sandeep Shukla

CS425 Project 3

Nishant Rai, 13660

October 1, 2016

1 Aim

To implement a STCP (Simple Transmission Control Protocol) layer that sits between the mysocket and network layers.

2 Variable description

This section describes the global fields added in the context_t struct. They are used to store state information about the current context and are updated accordingly. The description of the variables is provided in comments.

```
1  bool_t done;
2  /* TRUE once connection is closed */
3
4  int connection_state;
5  /* state of the connection (established, etc.) */
6  tcp_seq initial_sequence_num;
7
8  char sendBuff[WINDOW_SIZE];
9  /* The send buffer */
10 char recBuff[WINDOW_SIZE];
11 /* Receive buffer */
12
13 tcp_seq nextSeqSend;
14 /* Next sequence number to be sent from here */
15 tcp_seq lastSeqRecv;
16 /* Last sequence number received from the other host */
17 tcp_seq lastSeqAcked;
18 /* Last sequence number acked from the other host */
19
20 int sendWinSize;
21 int recWinSize;
22 int advWinSize;
23 /* Advertised window size by the other one */
24
25 bool closeInit;
26 /* Stores whether we've initiated the fin sequence */
```

For describing the state of the program, the following states are used to describe the current context. At different stages of the program, a connection state global variable stores the current state of the program. For example, when state is *CSTATE_WAIT_FOR_ACK*, the program is waiting for an *ACK* packet (This state is during the connection close procedure).

```
1 enum {
2       CSTATE_WAIT_FOR_ACK,
3       CSTATE_SEND_FIN,
4       CSTATE_FIN_RECVD,
5       CSTATE_WAIT_FOR_FIN,
6
7       CSTATE_CLOSED,
8       CSTATE_ESTABLISHED
9 };
```

3 Implementation Details

- Flags have been added for debugging which make it easier to track the flow. Just changing the value of the debug flag will lead to a quiet execution.
- The initial sequence number is generated randomly by using the rand() function and limiting it to MAX_SEQ_NUM (256).
- The send window size is computed using the following formula,

$$\begin{aligned} sendWinSize = min(& \\ & WIN\_SIZE - (nextSeqSend - lastSeqAcked - 1), \\ & advWinSize \\ &) \end{aligned}$$

  Here, WIN_SIZE is the maximum possible window size (3072). $(nextSeqSend - 1)$ gives us the sequence number of the last packet sent, and $lastSeqAcked$ gives us the sequence number of the last acked packet. Note that the packets are transferred and received in order. The available window for sending is thus given by their difference. Minimum is taken with $advWinSize$ (The advertised window size by the other host) since we do not want to overflow the buffers.
- Initially in the three way handshake, I've included a procedure to try atleast specific amount of time before claiming to give an error. The same is done in case of the final Fin-Ack sequence.

4 Testing Procedure

The STCP implementation has been checked with the provided server and client programs. Multiple files of varying sizes have been transferred and verified (0KB - 0.5MB).

5 Summary

A simplified TCP layer is implemented which solves the problem by assuming inorder transfer and receive. We also maintain sequence numbers and window sizes for avoiding overflows. Initialization and connection teardown take place through standard procedures i.e. *3-way handshake* and *4-way FIN-ACK sequence* respectively.

6 Execution Screenshots

7 Source Code

```c
/*
 * 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 <sys/time.h>

#define MAX_SEQ_NUM 256
#define WINDOW_SIZE 3072
#define MAX_SEND 6
#define OFFSET 5
#define TIME_OUT 2

#define DEBUG 1

enum {
    CSTATE_WAIT_FOR_ACK,
    CSTATE_SEND_FIN,
    CSTATE_FIN_RECVD,
    CSTATE_WAIT_FOR_FIN,

    CSTATE_CLOSED,
    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;

    char sendBuff[WINDOW_SIZE];      /* The send buffer */
    char recBuff[WINDOW_SIZE];       /* Receive buffer */

    tcp_seq nextSeqSend;    /* Next sequence number to be sent from here */
    tcp_seq lastSeqRecv;      /* Last sequence number received from the other host */
    tcp_seq lastSeqAcked;         /* Last sequence number acked from the other host */

    int sendWinSize;
    int recWinSize;
    int advWinSize;      /* Advertised window size by the other one */

    bool closeInit;      /* Stores whether we've initiated the fin sequence */

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


void printContext(context_t *ctx) {
    printf("---------------START_CONTEXT-------------------\n");
    printf("nextSeqSend : %u\n", ctx->nextSeqSend);
    printf("lastSeqRecv : %u\n", ctx->lastSeqRecv);
    printf("lastSeqAcked : %u\n", ctx->lastSeqAcked);
    printf("sendWinSize : %d\n", ctx->sendWinSize);
```

```
72      printf("recWinSize : %d\n", ctx->recWinSize);
73      printf("advWinSize : %d\n", ctx->advWinSize);
74      printf("closeInit : %d\n", (int) ctx->closeInit);
75      printf("------------FINISH_CONTEXT------------------\n");
76  }
77
78  void printPack(struct tcphdr *pack) {
79      printf("--------------START_PACK--------------------\n");
80      printf("Seq : %u\n", pack->th_seq);
81      printf("Ack : %u\n", pack->th_ack);
82      printf("Flags : %u\n", pack->th_flags);
83      printf("Window : %u\n", pack->th_win);
84      printf("------------FINISH_PACK--------------------\n");
85  }
86
87  static void generate_initial_seq_num(context_t *ctx);
88  static void control_loop(mysocket_t sd, context_t *ctx);
89  int tearDown(mysocket_t sd, context_t *ctx);
90
91  /* initialise the transport layer, and start the main loop, handling
92   * any data from the peer or the application.  this function should not
93   * return until the connection is closed.
94   */
95  void transport_init(mysocket_t sd, bool_t is_active)
96  {
97
98      if (DEBUG)
99          printf ("Entering func transport_init()\n");
100
101     context_t *ctx;
102
103     ctx = (context_t *) calloc(1, sizeof(context_t));
104     assert(ctx);
105
106     generate_initial_seq_num(ctx);
107     // Updates the initial seq num field in ctx
108
109     /* XXX: you should send a SYN packet here if is_active, or wait for one
110      * to arrive if !is_active.  after the handshake completes, unblock the
111      * application with stcp_unblock_application(sd).  you may also use
112      * this to communicate an error condition back to the application, e.g.
113      * if connection fails; to do so, just set errno appropriately (e.g. to
114      * ECONNREFUSED, etc.) before calling the function.
115      */
116
117     int sendCount = 0;
118
119     ctx->nextSeqSend = ctx->initial_sequence_num;
120     ctx->sendWinSize = ctx->recWinSize = WINDOW_SIZE;
121     ctx->advWinSize = 0;
122     ctx->lastSeqRecv = 0;
123     ctx->lastSeqAcked = 0;
124     // Invalid for now
125
126     if (DEBUG)
127         printf ("Initial sequence number: %d\n", ctx->initial_sequence_num);
128
129     if (is_active) {
130
131         if (DEBUG)
132             printf ("In active mode: Sending SYN packet\n");
133
134         struct tcphdr *synHead;
135         synHead = (struct tcphdr*) calloc(1, sizeof(struct tcphdr));
136
137         synHead->th_flags = TH_SYN;
138         synHead->th_win = ctx->recWinSize;
139         synHead->th_seq = ctx->nextSeqSend;
140
141         sendCount = 0;
142
143         if (DEBUG)
144             printPack(synHead);
```

```
145             // Add debug statement
146
147         while (1) {
148             if (sendCount >= MAX_SEND) {
149                 errno = ECONNREFUSED;
150                 // Setting errno to the approporiate state
151                 printf("ERROR: Maximum number of sends exhausted\n");
152                 return;
153             }
154             // Sending prepared packet
155             stcp_network_send(sd, synHead, sizeof(struct tcphdr), NULL);
156             sendCount++;
157
158             if (stcp_wait_for_event(sd, NETWORK_DATA, NULL) == NETWORK_DATA)
159                 break;
160         }
161
162         if (DEBUG)
163             printf("SYN Header sent! SeqNum: %d\n", synHead->th_seq);
164
165         sendCount = 0;
166         struct tcphdr *synAck;
167         synAck = (struct tcphdr *) calloc(1, sizeof(struct tcphdr));
168         stcp_network_recv(sd, synAck, STCP_MSS);
169
170         if (!((synAck->th_flags) & TH_ACK)) {
171             // Check the corresponding bit
172             printf("ERROR: Ack bit is not set\n");
173             return;
174         }
175
176         if (!((synAck->th_flags) & TH_SYN)) {
177             printf("ERROR: SYN is not set in packet\n");
178             return;
179         }
180
181         if (DEBUG)
182             printf("ACK received for: %d\n", synAck->th_ack);
183
184         ctx->lastSeqRecv = synAck->th_seq;
185         ctx->nextSeqSend++;
186         ctx->advWinSize = synAck->th_win;
187         ctx->sendWinSize = MIN(ctx->advWinSize, WINDOW_SIZE);
188
189         struct tcphdr *ackMsg;
190         ackMsg = (struct tcphdr *) calloc(1, sizeof(struct tcphdr));
191
192         ackMsg->th_seq = ctx->nextSeqSend;
193         ackMsg->th_flags = TH_ACK;
194         ackMsg->th_win = WINDOW_SIZE;
195         ackMsg->th_ack = ctx->lastSeqRecv + 1;
196
197         stcp_network_send(sd, ackMsg, sizeof(struct tcphdr), NULL);
198
199         if (DEBUG)
200             printf("ACK sent! SeqNum: %d, Ack for: %d\n", ackMsg->th_seq, ackMsg->th_ack
                    );
201
202         if (synHead)
203             free(synHead);
204
205         if (synAck)
206             free(synAck);
207
208         if (ackMsg)
209             free(ackMsg);
210
211     }
212     else {
213
214         if (DEBUG)
215             printf ("In passive mode: Waiting for SYN packet\n");
216
```

```
217         stcp_wait_for_event(sd, NETWORK_DATA, NULL);
218
219         struct tcphdr *synPack;
220         synPack = (struct tcphdr *) calloc(1, sizeof(struct tcphdr));
221         stcp_network_recv(sd, synPack, STCP_MSS);
222         // Note that STCP_MSS has the maximum segment length
223
224         if (!((synPack->th_flags) & TH_SYN)) {
225             printf("ERROR: SYN is not set in packet\n");
226             return;
227         }
228         if (DEBUG)
229             printf("SYN received! Seq Num : %d\n", synPack->th_seq);
230
231         ctx->advWinSize = synPack->th_win;
232         ctx->sendWinSize = MIN(ctx->advWinSize, WINDOW_SIZE);
233         ctx->lastSeqRecv = synPack->th_seq;
234
235         struct tcphdr *synAck;
236         synAck = (struct tcphdr *) calloc(1, sizeof(struct tcphdr));
237         synAck->th_ack = ctx->lastSeqRecv + 1;
238         synAck->th_win = WINDOW_SIZE;
239         synAck->th_flags = (TH_SYN | TH_ACK);
240         synAck->th_seq = ctx->nextSeqSend;
241
242         ctx->nextSeqSend++;
243
244         sendCount = 0;
245
246         while (1) {
247             if (sendCount >= MAX_SEND) {
248                 errno = ECONNREFUSED;
249                 // Setting errno to the approporiate state
250                 printf("ERROR: Maximum number of sends exhausted\n");
251                 return;
252             }
253
254             // Sending prepared packet
255             stcp_network_send(sd, synAck, sizeof(struct tcphdr), NULL);
256             sendCount++;
257
258             if (stcp_wait_for_event(sd, NETWORK_DATA, NULL) == NETWORK_DATA)
259                 break;
260
261         }
262
263         if (DEBUG)
264             printf("SYN ACK sent! SeqNum: %d, Ack for: %d\n", synAck->th_seq, synAck->
                    th_ack);
265
266         struct tcphdr * ack;
267         ack = (struct tcphdr *) (calloc(1, sizeof(struct tcphdr)));
268         stcp_network_recv(sd, ack, STCP_MSS);
269
270         if (!((ack->th_flags) & TH_ACK)) {
271             // Check the corresponding bit
272             printf("ERROR: Ack bit is not set\n");
273             return;
274         }
275
276         // if ((ack->th_ack) != (ctx->initial_sequence_num + 1)) {
277         //     printf("ERROR: Received out of sequence packet\n");
278         //     return;
279         // }
280
281         if (DEBUG)
282             printf("ACK received! Seq Num: %d, Ack for: %d\n", ack->th_seq, ack->th_ack)
                    ;
283
284         ctx->advWinSize = ack->th_win;
285         ctx->lastSeqRecv = ack->th_seq;
286         // ctx->lastSeqAcked = ack->th_seq;
287
```

```
288         if (ack)
289             free(ack);
290
291         if (synAck)
292             free(synAck);
293
294     }
295
296     if (DEBUG)
297         printf("Handshake successful: Connection established!\n");
298
299     ctx->connection_state = CSTATE_ESTABLISHED;
300     stcp_unblock_application(sd);
301
302     control_loop(sd, ctx);
303
304     /* do any cleanup here */
305     if (ctx)
306         free(ctx);
307
308 }
309
310 /* generate random initial sequence number for an STCP connection */
311 static void generate_initial_seq_num(context_t *ctx)
312 {
313     assert(ctx);
314
315 #ifdef FIXED_INITNUM
316     /* please don't change this! */
317     ctx->initial_sequence_num = 1;
318 #else
319     /* you have to fill this up */
320     /* Generate a random number in the range 0-255 */
321     ctx->initial_sequence_num = rand() % MAX_SEQ_NUM;
322     if (DEBUG)
323         printf("Initial sequence number: %d\n", ctx->initial_sequence_num);
324 #endif
325 }
326
327 // Create a timespec timeout object with the passed delay
328 timespec getTimeout(int timeSec) {
329     timeval currentTime;
330     gettimeofday(&currentTime, NULL);
331     timespec timeout;
332     timeout.tv_sec  = currentTime.tv_sec + timeSec;
333     // Add dealy from cuurent
334     timeout.tv_nsec = currentTime.tv_usec * 1000;
335     // Conversion form nano to micro
336     return timeout;
337 }
338
339 /* control_loop() is the main STCP loop; it repeatedly waits for one of the
340  * following to happen:
341  *   - incoming data from the peer
342  *   - new data from the application (via mywrite())
343  *   - the socket to be closed (via myclose())
344  *   - a timeout
345  */
346 static void control_loop(mysocket_t sd, context_t *ctx)
347 {
348     assert(ctx);
349     assert(!ctx->done);
350
351     if (DEBUG)
352         printContext(ctx);
353
354
355     // Reinitalize the ctx variables, for consistency
356     ctx->lastSeqAcked = ctx->lastSeqRecv + 1;
357     ctx->sendWinSize = MIN(ctx->advWinSize, WINDOW_SIZE);
358     ctx->recWinSize = WINDOW_SIZE;
359
360     while (!ctx->done)
```

```
361     {
362
363         unsigned int eventId;
364
365         timespec timeout = getTimeout(TIME_OUT);
366
367         /* see stcp_api.h or stcp_api.c for details of this function */
368         /* XXX: you will need to change some of these arguments! */
369         eventId = stcp_wait_for_event(sd, ANY_EVENT, &timeout);
370
371         ctx->sendWinSize = (WINDOW_SIZE - (ctx->nextSeqSend - ctx->lastSeqAcked - 1));
372
373         /* check whether it was the network, app, or a close request */
374         if (eventId & APP_DATA)
375         {
376
377             char *rcvData;
378             int rcvSize;
379
380             int availRec = (MIN(ctx->advWinSize, WINDOW_SIZE)) - sizeof(struct tcphdr);
381             int availSend = ctx->sendWinSize;
382
383             if (MIN(availSend,availRec) > 0) {
384
385                 if (DEBUG) {
386                     printf ("App Data mode: App has requested data to be sent\n");
387                     // printContext(ctx);
388                 }
389
390                 struct tcphdr *header;
391                 header = (struct tcphdr *) calloc(1, sizeof(struct tcphdr));
392                 header->th_off = OFFSET;
393                 header->th_win = ctx->recWinSize;
394                 header->th_seq = ctx->nextSeqSend;
395                 header->th_flags = 0;
396                 // Forming the header to send
397
398                 availRec = MIN(availSend, availRec);
399                 if (availRec > STCP_MSS)
400                     availRec = STCP_MSS;
401
402                 rcvData = (char *) malloc(availRec * sizeof(char));
403                 rcvSize = stcp_app_recv(sd, rcvData, availRec);
404
405                 if (DEBUG) {
406                     printf("availRec, availSend: %d, %d\n", availRec, availSend);
407                 }
408
409                 while (stcp_network_send(sd, header, sizeof(struct tcphdr), rcvData,
                        rcvSize, NULL) == -1);
410                 // Repeatedly keep sending
411
412                 int startNum = ctx->nextSeqSend % WINDOW_SIZE;
413                 int endNum = (startNum + rcvSize);
414
415                 if (DEBUG)
416                     printf("start, end: %d, %d\n", startNum, endNum);
417
418                 int i;
419                 if (endNum < WINDOW_SIZE) {
420                     for (i = startNum; i < endNum; i++)
421                         ctx->sendBuff[i] = rcvData[i - startNum];
422                         // Storing the data at the appropriate place in the buffer
423                 }
424                 else {
425                     // Wrap around, Extracting the elements in the first part
426                     for (i = startNum; i < WINDOW_SIZE; i++)
427                         ctx->sendBuff[i] =  rcvData[i - startNum];
428                     // Extracting the elements in the second part
429                     for (i = 0; i < endNum % WINDOW_SIZE; i++)
430                         ctx->sendBuff[i] = rcvData[WINDOW_SIZE - startNum + i];
431                 }
432
```

```
433                 ctx->nextSeqSend += rcvSize;
434
435                 if (DEBUG)
436                     printf("Data packet of size: %d, Seq Num: %d sent\n", rcvSize, ctx->
                            nextSeqSend);
437
438                 if (rcvData)
439                     free(rcvData);
440
441             }
442         }
443
444         if (eventId & NETWORK_DATA)
445         {
446
447             if (DEBUG) {
448                 printf ("Network Data mode: Network wants to deliver data\n");
449                 // printContext(ctx);
450             }
451
452             // Get segment from network
453             int segLen = 2 * STCP_MSS;
454             // Upper limit to size
455             char *segData;
456             segData = (char *) malloc(segLen * sizeof(char));
457             segLen = stcp_network_recv(sd, segData, segLen);
458             // Update segLen to the actual amount of data
459
460             if (DEBUG)
461                 printf("Segment accepted from network\n");
462
463             tcphdr *recvMsg = (tcphdr *) segData;
464             // Read the initial part of the message
465             ctx->advWinSize = recvMsg->th_win;
466
467             if (DEBUG)
468                 printPack(recvMsg);
469
470             if (recvMsg->th_flags & TH_ACK) {
471                 // If it is an acknowledgement
472                 if (DEBUG)
473                     printf("ACK message received, seqNum: %d, ack for: %d\n", recvMsg->
                            th_seq, recvMsg->th_ack);
474                 ctx->lastSeqAcked = recvMsg->th_ack;
475             }
476
477             int segSize = segLen - recvMsg->th_off * 4;
478             // Getting the part with the data
479             ctx->lastSeqRecv = recvMsg->th_seq;
480
481             if (segSize != 0) {
482
483                 if (DEBUG) {
484                     printf("Entering data transfer part. Segment contains data with
                            seqNum: %d\n", recvMsg->th_seq);
485                     printf("Current Send Window Size: %d\n", ctx->sendWinSize);
486                 }
487
488                 char *actData = segData + recvMsg->th_off * 4;
489
490                 struct tcphdr *ackMsg;
491                 ackMsg = (struct tcphdr *) (calloc(1, sizeof(struct tcphdr)));
492                 ackMsg->th_flags = TH_ACK;
493                 ackMsg->th_off = OFFSET;
494                 ackMsg->th_win = ctx->recWinSize;
495                 ackMsg->th_ack = ctx->lastSeqRecv + segSize;
496                 ackMsg->th_seq = ctx->nextSeqSend;
497                 // Prepare ack message
498
499                 stcp_network_send(sd, ackMsg, sizeof(tcphdr), NULL);
500                 // Send the ack message
501
502                 if (DEBUG) {
```

```
503                     // print(ackMsg);
504                     printf("ACK message sent!: SeqNum: %d, Ack for: %d\n", ackMsg->
                            th_seq, ackMsg->th_ack);
505                 }
506
507                 stcp_app_send(sd, actData, segSize);
508                 // Sending data to app, note that appData contains the data part
509
510                 if (ackMsg)
511                     free(ackMsg);
512
513             }
514
515             if (recvMsg->th_flags & TH_FIN) {
516                 if (DEBUG)
517                     printf("FIN received: Initiating FIN-ACK sequence\n");
518                 ctx->connection_state = CSTATE_FIN_RECVD;
519                 ctx->closeInit = false;
520                 tearDown(sd, ctx);
521             }
522         }
523
524         if (eventId & APP_CLOSE_REQUESTED) {
525             if (DEBUG)
526                 printf("APP requests to close connection: Initiating FIN sequence\n");
527             ctx->connection_state = CSTATE_SEND_FIN;
528             ctx->closeInit = true;
529             tearDown(sd, ctx);
530         }
531     }
532 }
533
534 int tearDown(mysocket_t sd, context_t *ctx) {
535
536     int count = 0;
537     timespec timeout;
538     unsigned int stat;
539
540     // Keep trying to close till you pass a threshold
541     while (ctx->connection_state != CSTATE_CLOSED)
542     {
543
544         if (DEBUG)
545             printContext(ctx);
546
547         switch (ctx->connection_state) {
548
549             case CSTATE_SEND_FIN:
550
551                 if (count >= MAX_SEND) {
552                     errno = ECONNREFUSED;
553                     return -1;
554                 }
555                 if (DEBUG)
556                     printf("Preparing to send FIN\n");
557
558                 // Prepare fin header
559                 struct tcphdr *finHead;
560                 finHead = (struct tcphdr *) (calloc(1, sizeof(struct tcphdr)));
561
562                 finHead->th_flags = TH_FIN;
563                 finHead->th_seq = ctx->nextSeqSend;
564                 finHead->th_win = ctx->recWinSize;
565                 finHead->th_off = OFFSET;
566
567                 ctx->nextSeqSend++;
568
569                 if (DEBUG)
570                     printPack(finHead);
571
572                 while (stcp_network_send(sd, finHead, sizeof(struct tcphdr), NULL) ==
                        -1);
573
```

```
574                 if (DEBUG)
575                     printf("FIN message sent! seqNum: %d\n", finHead->th_seq);
576
577                 ctx->connection_state = CSTATE_WAIT_FOR_ACK;
578                 // Update connection state
579
580                 if (finHead) {
581                     free(finHead);
582                 }
583
584                 count++;
585                 break;
586
587             case CSTATE_WAIT_FOR_ACK:
588
589                 if (DEBUG)
590                     printf("Currently waiting for ACK\n");
591
592                 timeout = getTimeout(TIME_OUT);
593                 stat = stcp_wait_for_event(sd, NETWORK_DATA, &timeout);
594
595                 if (stat & NETWORK_DATA) {
596
597                     // Get segment from network
598                     int segLen = 2 * STCP_MSS;
599                     // Upper limit to size
600                     char *segData;
601                     segData = (char *) malloc(segLen * sizeof(char));
602                     segLen = stcp_network_recv(sd, segData, segLen);
603                     // Update segLen to the actual amount of data
604
605                     if (DEBUG)
606                         printf("Segment accepted from network\n");
607
608                     tcphdr *recvMsg = (tcphdr *) segData;
609
610                     if (!(recvMsg->th_flags & TH_ACK) || (recvMsg->th_ack != ctx->
                            nextSeqSend)) {
611                         if (DEBUG)
612                             printf("FIN ACK not received: Trying again\n");
613                         ctx->connection_state = CSTATE_SEND_FIN;
614                     }
615                     else {
616                         if (DEBUG)
617                             printf("FIN ACK received, seqNum: %d, ack for: %d\n",
                                    recvMsg->th_seq, recvMsg->th_ack);
618                         if (ctx->closeInit)
619                             // Now need to wait for a fin, since we started it
620                             ctx->connection_state = CSTATE_WAIT_FOR_FIN;
621                         else
622                             // Since we received a fin ack, thus the connection is
                                    successfully closed now
623                             ctx->connection_state = CSTATE_CLOSED;
624                         ctx->lastSeqRecv = recvMsg->th_seq;
625                         count = 0;
626                     }
627
628                     if (segData)
629                         free(segData);
630
631                 }
632                 else
633                     ctx->connection_state = CSTATE_SEND_FIN;
634                     // Try again
635
636                 break;
637
638             case CSTATE_WAIT_FOR_FIN:
639
640                 if (DEBUG)
641                     printf("Currently waiting for FIN\n");
642
643                 timeout = getTimeout(TIME_OUT);
```

```
644                 stat = stcp_wait_for_event(sd, NETWORK_DATA, &timeout);
645
646                 if (stat & NETWORK_DATA) {
647
648                     // Get segment from network
649                     int segLen = 2 * STCP_MSS;
650                     // Upper limit to size
651                     char *segData;
652                     segData = (char *) malloc(segLen * sizeof(char));
653                     segLen = stcp_network_recv(sd, segData, segLen);
654                     // Update segLen to the actual amount of data
655
656                     if (DEBUG)
657                         printf("Segment accepted from network\n");
658
659                     tcphdr *recvMsg = (tcphdr *) segData;
660
661                     if (!(recvMsg->th_flags & TH_FIN)) {
662                         if (DEBUG)
663                             printf("FIN not received\n");
664                     }
665                     else {
666                         if (DEBUG)
667                             printf("FIN message received, seqNum %d\n", recvMsg->th_seq)
                                ;
668                         ctx->lastSeqRecv = recvMsg->th_seq;
669                         ctx->connection_state = CSTATE_FIN_RECVD;
670                     }
671
672                     if (segData)
673                         free(segData);
674
675                 }
676
677                 break;
678
679             case CSTATE_FIN_RECVD:
680
681                 if (DEBUG)
682                     printf("Currently preparing to send ACK\n");
683
684                 // Prepare ack header
685                 struct tcphdr *ackHead;
686                 ackHead = (struct tcphdr *) (calloc(1, sizeof(struct tcphdr)));
687
688                 ackHead->th_flags = TH_ACK;
689                 ackHead->th_win = ctx->recWinSize;
690                 ackHead->th_off = OFFSET;
691                 ackHead->th_seq = ctx->nextSeqSend;
692                 ackHead->th_ack = ctx->lastSeqRecv + 1;
693
694                 // Send the fin ack
695                 while (stcp_network_send(sd, ackHead, sizeof(struct tcphdr), NULL) ==
                        -1);
696
697                 if (ctx->closeInit)
698                     ctx->connection_state = CSTATE_CLOSED;
699                 else
700                     ctx->connection_state = CSTATE_SEND_FIN;
701
702                 if (DEBUG)
703                     printf("ACK message sent!\n");
704
705                 if (ackHead)
706                     free(ackHead);
707
708                 break;
709
710             default:
711                 break;
712
713         }
714     }
```

```
715
716     if (DEBUG)
717         printf("Connection successfully closed!\n");
718
719     ctx->done = true;
720
721     return 1;
722 }
723
724 /*****************************************************************/
725 /* our_dprintf
726  *
727  * Send a formatted message to stdout.
728  *
729  * format               A printf-style format string.
730  *
731  * This function is equivalent to a printf, but may be
732  * changed to log errors to a file if desired.
733  *
734  * Calls to this function are generated by the dprintf amd
735  * dperror macros in transport.h
736  */
737 void our_dprintf(const char *format,...)
738 {
739     va_list argptr;
740     char buffer[1024];
741
742     assert(format);
743     va_start(argptr, format);
744     vsnprintf(buffer, sizeof(buffer), format, argptr);
745     va_end(argptr);
746     fputs(buffer, stdout);
747     fflush(stdout);
748 }
```