STCP Transport Layer

CS425 - Computer Networks

Vaibhav Nagar(14785)

Email: vaibhavn@iitk.ac.in

September 30, 2016

Elementary features of Transport Layer

- It establishes connection between two peers after three way handshake.
- STCP provides a 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.
- STCP treats application data as a stream i.e., no artificial boundaries are imposed on the data by the transport layer.
- It runs continuously until connection is either closed by peer or by the application itself. In between it waits for events when network packet arrives or application sends some data.
- The whole code is developed only in C language using various libraries.

Design Choices

The code is written in imperative manner.

- An STCP packet has a maximum segment size of 536 bytes and with fixed congestion window of 3072 bytes.
- The local receiver window has a fixed size of 3072 bytes.

Test Procedure

Testing has been done on the VM provided and as well as on the Ubuntu 16.04 machine. Server and client runs perfectly using my implemented tranport layer.
Also provided binary file of server works smoothly with the client using my transport layer and also the client binary makes flawless connection and make requests with server using my transport layer.

Summary

STCP layer gracefully bridges the gap between application layer with the network layer. An application sends data which is then passed to network layer after encapsulating the payload with appropriate TCP headers. Similarly network data received is pushed up to application after removing TCP headers.

Acknowledgements are sent to peer to notify successful delivery. Proper FIN/ACK is sent to and fro when at least one of the peer requested to close the connection.

Appendix

Source Code of Transport Layer

```
1
2   /*
3   * transport.c
4   *
5   * Project 3
6   *
7   * This file implements the STCP layer that sits between the
8   * mysocket and network layers. You are required to fill in the STCP
9   * functionality in this file.
10  *
11  */
12
13
14  #include <stdio.h>
15  #include <stdarg.h>
16  #include <string.h>
17  #include <stdlib.h>
18  #include <assert.h>
19  #include <arpa/inet.h>
20  #include <sys/time.h>
21  #include "mysock.h"
22  #include "stcp_api.h"
23  #include "transport.h"
24
25
26  enum { LISTEN, SYN_SENT, SYN_RECEIVED, CSTATE_ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2,
      CLOSE_WAIT, CLOSING, LAST_ACK, TIME_WAIT, CLOSED };     /* you should have more
     states */
27
28  #define MAX_SEQ_NUM 4294967296
29  #define ISN_RANGE 256
30  #define WIN_SIZE 3072
31  #define CONGESTION_WIN_SIZE 3072
32  #define OFFSET 5
33  #define MSS 536
34  #define HDR_SIZE sizeof(STCPHeader)
35  #define OPTIONS_SIZE 40
36
37  /* this structure is global to a mysocket descriptor */
38  typedef struct
39  {
40  bool_t done;    /* TRUE once connection is closed */
41
42  int connection_state;   /* state of the connection (established, etc.) */
43  tcp_seq initial_sequence_num;
44  tcp_seq initial_ack_num;
45
46  /*Used when send data from app to network and to determine
     remote_advertised_window*/
47  tcp_seq sequence_num;
```

```
48  tcp_seq last_byte_acked;
49
50  /*Used when send data from network to app and to determine local_advertised_window
     */
51  tcp_seq ack_num;
52  tcp_seq last_byte_read;
53
54  /* Remote and local advertised window */
55  uint16_t remote_advertised_window;
56  uint16_t local_advertised_window;
57
58  /* any other connection-wide global variables go here */
59  } context_t;
60
61
62  static void generate_initial_seq_num(context_t *ctx);
63  static void control_loop(mysocket_t sd, context_t *ctx);
64
65  /* My Functions */
66
67  uint16_t getRemoteWindowSize(context_t *ctx);
68  uint16_t getLocalWindowSize(context_t *ctx);
69  void setSTCPheader(STCPHeader *hdr, tcp_seq seq, tcp_seq ack, uint32_t data_offset
     , uint8_t flag, uint16_t win);
70  void getSTCPheader(STCPHeader *hdr, char *buf);
71  void printSTCPheader(STCPHeader *hdr);
72  void send_segment_to_app(mysocket_t sd, context_t *ctx, char *segment, size_t
     segment_size, STCPHeader *hdr);
73  void printContext(context_t *ctx);
74
75  uint32_t myhtonl(uint32_t hostlong)
76  {
77  #if __BYTE_ORDER == __LITTLE_ENDIAN
78  return htonl(hostlong);
79  #elif __BYTE_ORDER == __BIG_ENDIAN
80  return hostlong;
81  #else
82  #error __BYTE_ORDER must be defined as __LITTLE_ENDIAN or __BIG_ENDIAN!
83  #endif
84  printf("BYTEORDER: MUST NOT reach Here\n");
85  return -1;
86  }
87
88  uint16_t myhtons(uint16_t hostshort)
89  {
90  #if __BYTE_ORDER == __LITTLE_ENDIAN
91  return htons(hostshort);
92  #elif __BYTE_ORDER == __BIG_ENDIAN
93  return hostshort;
94  #else
95  #error __BYTE_ORDER must be defined as __LITTLE_ENDIAN or __BIG_ENDIAN!
96  #endif
97  printf("BYTEORDER: MUST NOT reach Here\n");
98  return -1;
```

```
99  }
100
101 uint32_t myntohl(uint32_t netlong)
102 {
103 #if __BYTE_ORDER == __LITTLE_ENDIAN
104 return ntohl(netlong);
105 #elif __BYTE_ORDER == __BIG_ENDIAN
106 return netlong;
107 #else
108 #error __BYTE_ORDER must be defined as __LITTLE_ENDIAN or __BIG_ENDIAN!
109 #endif
110 printf("BYTEORDER: MUST NOT reach Here\n");
111 return -1;
112 }
113
114 uint16_t myntohs(uint16_t netshort)
115 {
116 #if __BYTE_ORDER == __LITTLE_ENDIAN
117 return ntohs(netshort);
118 #elif __BYTE_ORDER == __BIG_ENDIAN
119 return netshort;
120 #else
121 #error __BYTE_ORDER must be defined as __LITTLE_ENDIAN or __BIG_ENDIAN!
122 #endif
123 printf("BYTEORDER: MUST NOT reach Here\n");
124 return -1;
125 }
126
127
128 /* Send STCP header with appropriate flags to peer  */
129 int sendTCPheader(mysocket_t sd, STCPHeader *hdr, context_t *ctxt, uint32_t
      data_offset, uint8_t flag)
130 {
131 hdr->th_seq = myhtonl(ctxt->sequence_num);
132 hdr->th_ack = myhtonl(ctxt->ack_num + 1);
133 hdr->th_off = data_offset;
134 hdr->th_flags = flag;
135 hdr->th_win = myhtons(ctxt->local_advertised_window);
136 return stcp_network_send(sd, hdr, sizeof(STCPHeader), NULL);
137 }
138
139
140
141 /* initialise the transport layer, and start the main loop, handling
142 * any data from the peer or the application.  this function should not
143 * return until the connection is closed.
144 */
145 void transport_init(mysocket_t sd, bool_t is_active)
146 {
147 // check
148 context_t *ctx;
149 ssize_t bytes_transfer;
150 STCPHeader *header = (STCPHeader*)calloc(1, sizeof(STCPHeader));
151 ctx = (context_t *) calloc(1, sizeof(context_t));
```

```
152  assert(ctx);
153  generate_initial_seq_num(ctx);
154
155  ctx->done = FALSE;
156  ctx->local_advertised_window = WIN_SIZE;
157  ctx->remote_advertised_window = WIN_SIZE;                    // Default window
      size
158
159  /* XXX: you should send a SYN packet here if is_active, or wait for one
160  * to arrive if !is_active.  after the handshake completes, unblock the
161  * application with stcp_unblock_application(sd).  you may also use
162  * this to communicate an error condition back to the application, e.g.
163  * if connection fails; to do so, just set errno appropriately (e.g. to
164  * ECONNREFUSED, etc.) before calling the function.
165  */
166  // Active connection
167  if(is_active == TRUE)
168  {
169  while(ctx->connection_state != CSTATE_ESTABLISHED)
170  {
171  if(sendTCPheader(sd, header, ctx, OFFSET, 0 | TH_SYN) == -1)        // Send SYN
172  {
173  errno = ECONNREFUSED;
174  ctx->connection_state = CLOSED;
175  printf("Error: Receive -1 from stcp_network_send method\n SYN Not sent => CLOSING
      connection\n");
176  break;
177  }
178  else
179  {
180  ctx->sequence_num++;
181  ctx->connection_state = SYN_SENT;
182  bzero(header, HDR_SIZE);
183  bytes_transfer = stcp_network_recv(sd, (void*)header, sizeof(STCPHeader));  //
      Receive SYN+ACK
184  if(bytes_transfer <= 0 && !(header->th_flags & (TH_SYN | TH_ACK)))
185  {
186  printf("ERROR: SYN and ACK Flag NOT set by server\nCLOSING CONNECTION\n");
187  ctx->connection_state = CLOSED;
188  break;
189  }
190  else
191  {
192  ctx->connection_state = SYN_RECEIVED;
193  ctx->initial_ack_num = myntohl(header->th_seq);
194  ctx->ack_num = ctx->initial_ack_num;
195  ctx->last_byte_read = ctx->ack_num;
196  ctx->last_byte_acked = myntohl(header->th_ack);
197  ctx->remote_advertised_window = MIN(CONGESTION_WIN_SIZE, myntohs(header->th_win));
198
199  bzero(header, HDR_SIZE);
200  if(sendTCPheader(sd, header, ctx, OFFSET, 0 | TH_ACK) == -1)        // Send ACK
201  {
202  errno = ECONNREFUSED;
```

```
203   ctx->connection_state = CLOSED;
204   printf("ERROR: Unable to send ack to server\nCLOSING CONNECTION\n");
205   break;
206   }
207   else
208   {
209   ctx->connection_state = CSTATE_ESTABLISHED;
210   }
211   }
212   }
213   }
214   }
215   // Passive connection
216   else
217   {
218   ctx->connection_state = LISTEN;
219   bytes_transfer = stcp_network_recv(sd, (void*)header, sizeof(STCPHeader));
       /*Receive SYN*/
220
221   if(bytes_transfer <= 0 && !(header->th_flags & TH_SYN))
222   {
223   printf("SYN Flag NOT set\n");
224   }
225   else
226   {
227   ctx->initial_ack_num = myntohl(header->th_seq);
228   ctx->ack_num = ctx->initial_ack_num;
229   ctx->last_byte_read = ctx->ack_num;
230   ctx->remote_advertised_window = MIN(CONGESTION_WIN_SIZE, myntohs(header->th_win));
231   ctx->connection_state = SYN_RECEIVED;
232   bzero(header, HDR_SIZE);
233   if(sendTCPheader(sd, header, ctx, OFFSET, 0 | TH_SYN | TH_ACK) == -1)         /*
       Send SYN+ACK */
234   {
235   errno = ECONNREFUSED;
236   ctx->connection_state = CLOSED;
237   printf("ERROR: Unable to send syn+ack to peer\nCLOSING CONNECTION\n");
238   }
239   else
240   {
241   ctx->sequence_num++;
242   ctx->connection_state = SYN_SENT;
243   bzero(header, HDR_SIZE);
244   bytes_transfer = stcp_network_recv(sd, (void*)header, sizeof(STCPHeader));   /*
       Receive ACK */
245   if(bytes_transfer <= 0  && !(header->th_flags & TH_ACK))
246   {
247   ctx->connection_state = CLOSED;
248   printf("ERROR: ACK Flag NOT set by peer\nCLOSING CONNECTION\n");
249   }
250   else
251   {
252   ctx->last_byte_acked = myntohl(header->th_ack);
253   ctx->remote_advertised_window = MIN(CONGESTION_WIN_SIZE, myntohs(header->th_win));
```

```
254   ctx->connection_state = CSTATE_ESTABLISHED;
255   }
256   }
257   }
258   }
259
260   stcp_unblock_application(sd);
261
262   if(header)
263   {
264   free(header);
265   header = NULL;
266   }
267
268   /* Going to Control Loop if connection is estblished*/
269   if(ctx->connection_state == CSTATE_ESTABLISHED)
270   control_loop(sd, ctx);
271
272   /* do any cleanup here */
273   free(ctx);
274   }
275
276
277   /* generate random initial sequence number for an STCP connection */
278   static void generate_initial_seq_num(context_t *ctx)
279   {
280   assert(ctx);
281
282   #ifdef FIXED_INITNUM
283   /* please don't change this! */
284   ctx->initial_sequence_num = 1;
285   #else
286   /* you have to fill this up */
287   /*ctx->initial_sequence_num =;*/
288   srand(time(NULL));
289   ctx->initial_sequence_num = (tcp_seq)(rand()%ISN_RANGE);
290   ctx->sequence_num = ctx->initial_sequence_num;
291   ctx->ack_num = ctx->sequence_num;                    /* Can be Random, so
       initializing to its own sequence_num */
292   ctx->last_byte_acked = ctx->sequence_num;
293   #endif
294   }
295
296
297   /* control_loop() is the main STCP loop; it repeatedly waits for one of the
298   * following to happen:
299   *   - incoming data from the peer
300   *   - new data from the application (via mywrite())
301   *   - the socket to be closed (via myclose())
302   *   - a timeout
303   */
304   static void control_loop(mysocket_t sd, context_t *ctx)
305   {
306   assert(ctx);
```

```
307   assert(!ctx->done);
308
309   /* Header variables for network and application */
310   STCPHeader *app_hdr = (STCPHeader*)calloc(1, sizeof(STCPHeader));
311   STCPHeader *network_hdr = (STCPHeader*)calloc(1, sizeof(STCPHeader));
312
313   if(app_hdr == NULL || network_hdr == NULL)
314   {
315   printf("Error: Unable to allocate space. Insufficient Memory!\n");
316   ctx->connection_state = CLOSED;
317   return;
318   }
319
320   ssize_t bytes_transfer;
321
322   uint16_t curr_remote_window;
323   uint16_t curr_local_window;
324
325   /* Variables to handle Application Data */
326   size_t app_segment_size;
327
328   /* Variables to handle Network Data */
329   size_t network_packet_size;
330   size_t network_segment_size;
331   size_t payload_size;
332
333   tcp_seq ack_value = 0;
334   uint8_t dataOffset;
335
336   /* Boolean Variables  */
337   /* LocalFINcalled: Set to 1 when I send FIN first*/
338   int LocalFINcalled = 0;
339   /* RemoteFINcalled: Set to 1 when peer sends FIN first*/
340   int RemoteFINcalled = 0;
341
342   /* Sequence Number at which FIN is sent or received */
343   tcp_seq FIN_seq_num;
344
345   struct timespec *abs_time = NULL;
346   struct timeval tv;
347
348   unsigned int waitFlags = ANY_EVENT;
349
350   while (!ctx->done)
351   {
352   unsigned int event;
353
354   curr_remote_window = getRemoteWindowSize(ctx);                    /* Remote
       window Size Left */
355   curr_local_window = getLocalWindowSize(ctx);                      /* Local
       window Size Left */
356
357   /* If FIN has been sent or received then No APP_DATA would be entertained */
358   if(ctx->connection_state == FIN_WAIT_1 || ctx->connection_state == FIN_WAIT_2)
```

```
359  waitFlags = 0 | NETWORK_DATA;
360  /* Wait for any event if there is sufficient remote window */
361  else if(curr_remote_window > 0)
362  waitFlags = 0 | ANY_EVENT;
363  /* No Remote Window size left then only wait for NETWORK_DATA or
      APP_CLOSE_REQUESTED*/
364  else
365  waitFlags = 0 | NETWORK_DATA | APP_CLOSE_REQUESTED;
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
370  event = stcp_wait_for_event(sd, waitFlags, abs_time);
371
372  /* check whether it was the network, app, or a close request */
373  /* Application Data */
374  if (event & APP_DATA)
375  {
376  /* the application has requested that data be sent */
377  /* see stcp_app_recv() */
378
379  app_segment_size = MSS;                          /* Payload Limit is a MSS */
380
381  if(curr_remote_window > 0)
382  {
383  if(curr_remote_window < MSS)
384  app_segment_size = (size_t)curr_remote_window;
385
386  char *app_segment = (char*)calloc(1, app_segment_size) ;
387  app_segment_size = stcp_app_recv(sd, app_segment, app_segment_size);
388
389  if(app_segment_size > 0)
390  {
391  /* No TCP options are set */
392  setSTCPheader(app_hdr, ctx->sequence_num, (ctx->ack_num + 1)%MAX_SEQ_NUM, OFFSET,
      0|TH_ACK, ctx->local_advertised_window);
393
394  bytes_transfer = stcp_network_send(sd, app_hdr, HDR_SIZE, app_segment,
      app_segment_size, NULL);
395
396  if(bytes_transfer < 0)
397  {
398  /* Peer may have closed connection abruptly*/
399  /* No other possibilities, as no packet loss is assumed*/
400  errno = ECONNREFUSED;
401  ctx->connection_state = CLOSED;
402  ctx->done = TRUE;     /* I don't think I should try again */
403  if(app_segment)
404  {
405  free(app_segment);
406  app_segment = NULL;
407  }
408  continue;
409  }
```

```
410
411 ctx->sequence_num = (ctx->sequence_num + app_segment_size)%MAX_SEQ_NUM;
412 }
413
414 if(app_segment)
415 {
416 free(app_segment);
417 app_segment = NULL;
418 }
419 bzero(app_hdr, HDR_SIZE);
420 }
421 else
422 {
423 waitFlags = 0 | NETWORK_DATA | APP_CLOSE_REQUESTED;
424 }
425 }
426 /* Network Data */
427 if(event & NETWORK_DATA)
428 {
429 payload_size = MSS;
430
431 if(curr_local_window > 0)
432 {
433 if(curr_local_window < MSS)
434 {
435 /* Removing OPTIONS_SIZE also from payload_size because if there are no options
      then more payload will be
436 read due to OPTIONS_SIZE*/
437 /* Such case won't occur as every time local window is advertised 3072 to peer*/
438 payload_size = curr_local_window - OPTIONS_SIZE;
439 }
440
441 /* Full Payload with header and options*/
442 network_packet_size = HDR_SIZE + OPTIONS_SIZE + payload_size;
443
444 char *network_packet = (char*)calloc(1, network_packet_size);
445
446 /* Receive network packet from peer*/
447 network_packet_size = stcp_network_recv(sd, network_packet, network_packet_size);
448
449 /* Received packet should be of length greater than equal to STCP header Size */
450 if(network_packet_size >= HDR_SIZE)
451 {
452 /* Copies first 20 bytes i.e. Header Size from network_packet to network_hdr */
453 getSTCPheader(network_hdr, network_packet);
454
455 ctx->remote_advertised_window = MIN(CONGESTION_WIN_SIZE, myntohs(network_hdr->
      th_win));
456
457 /* ACK Flag is set by peer */
458 if(network_hdr->th_flags & TH_ACK)
459 {
460 ack_value = myntohl(network_hdr->th_ack);
461
```

```
462  /* ack_value lies between window */
463  if(ack_value >= ctx->last_byte_acked && ack_value <= ctx->sequence_num)
464  {
465  ctx->last_byte_acked = ack_value;
466
467  /* Case I: Got ACK of my FIN which was sent before the peer's FIN */
468  if(ctx->connection_state == FIN_WAIT_1 && (ack_value-1) >= FIN_seq_num)
469  ctx->connection_state = FIN_WAIT_2;
470
471  /* Case II: Got ACK of my FIN which was sent after the peer's FIN */
472  if(RemoteFINcalled == 1 && (ack_value-1) >= FIN_seq_num)
473  {
474  /* All FINs are sent and ACKed so Now closing connection */
475  ctx->connection_state = CLOSED;
476  ctx->done = TRUE;
477  RemoteFINcalled = 0;
478
479  if(network_packet)
480  {
481  free(network_packet);
482  network_packet = NULL;
483  }
484  continue;
485  }
486  }
487  else
488  {
489  ;/* ACK flag is not set. Such case can occur only when */
490  /* ACK flag is not set deliberately by the peer */
491  }
492  }
493  if(network_hdr->th_flags & TH_FIN)
494  {
495  ctx->ack_num = myntohl(network_hdr->th_seq);
496
497  /* Already sent the FIN and now got the FIN of peer */
498  if(ctx->connection_state == FIN_WAIT_2)
499  {
500  /* Well, application must not be blocked at this point on myread call as */
501  /* app queue was  cleared before . But Still to remain safe. */
502  stcp_fin_received(sd);
503
504  /* Send ACK of peer's FIN  */
505  bzero(network_hdr, HDR_SIZE);
506  setSTCPheader(network_hdr, ctx->sequence_num, (ctx->ack_num + 1)%MAX_SEQ_NUM,
      OFFSET, 0|TH_ACK, ctx->local_advertised_window);
507
508  stcp_network_send(sd, network_hdr, HDR_SIZE, NULL);
509
510  /* NOW close the connection */
511  ctx->connection_state = CLOSED;
512  ctx->done = TRUE;
513  if(network_packet)
514  {
```

```
515   free(network_packet);
516   network_packet = NULL;
517   }
518   continue;
519   }
520   /* Receive FIN from peer */
521   if(ctx->connection_state == CSTATE_ESTABLISHED)
522   {
523   /* Suspending all transmission to and fro from application */
524   /* As packets are assumed to be arrived in order so no more data packets will
       arrive */
525   stcp_fin_received(sd);
526   RemoteFINcalled = 1;
527   }
528
529   }
530
531   dataOffset = (network_hdr->th_off);
532   network_segment_size = network_packet_size - (dataOffset*4);
533
534   /* There must be some payload in network packet */
535   /* If connection state is not established then no need to send data to application
       */
536   if(network_segment_size > 0 && ctx->connection_state == CSTATE_ESTABLISHED)
537   {
538   if(ctx->local_advertised_window < network_segment_size)
539   ctx->local_advertised_window = 0;
540   else
541   ctx->local_advertised_window -= network_segment_size;
542
543   assert(network_segment_size <= MSS);
544
545   char *network_segment = (char*)calloc(1, network_segment_size);
546   bcopy(network_packet + (dataOffset*4), network_segment, network_segment_size);
547
548   send_segment_to_app(sd, ctx, network_segment, network_segment_size, network_hdr);
549
550   ctx->local_advertised_window += network_segment_size;
551   if(ctx->local_advertised_window > WIN_SIZE)
552   ctx->local_advertised_window = WIN_SIZE;
553
554   /* Send Ack */
555   bzero(network_hdr, HDR_SIZE);
556   setSTCPheader(network_hdr, ctx->sequence_num, (ctx->ack_num + 1)%MAX_SEQ_NUM,
       OFFSET, 0|TH_ACK, ctx->local_advertised_window);
557
558   stcp_network_send(sd, network_hdr, HDR_SIZE, NULL);
559   /* ACK sent */
560   }
561   /* If remote sent FIN first and no payload is appended then sending ACK of FIN */
562   else if(RemoteFINcalled == 1 && ctx->connection_state == CSTATE_ESTABLISHED)
563   {
564   ctx->last_byte_read = ctx->ack_num;
565
```

```
566   bzero(network_hdr, HDR_SIZE);
567   setSTCPheader(network_hdr, ctx->sequence_num, (ctx->ack_num + 1)%MAX_SEQ_NUM,
       OFFSET, 0|TH_ACK, ctx->local_advertised_window);
568
569   stcp_network_send(sd, network_hdr, HDR_SIZE, NULL);
570   }
571
572   bzero(network_hdr, HDR_SIZE);
573   }
574   /* Detected a EOF of socket or peer may have presses Ctrl+C */
575   else if(network_packet_size == 0)
576   {
577   stcp_fin_received(sd);
578
579   ctx->connection_state = CLOSED;
580   ctx->done = TRUE;
581
582   if(network_packet)
583   {
584   free(network_packet);
585   network_packet = NULL;
586   }
587
588   continue;
589   }
590   if(network_packet)
591   {
592   free(network_packet);
593   network_packet = NULL;
594   }
595   }
596   }
597   /* APP_CLOSE_REQUESTED */
598   if((event & APP_CLOSE_REQUESTED || LocalFINcalled == 1 || RemoteFINcalled == 1) &&
       (ctx->connection_state == CSTATE_ESTABLISHED))
599   {
600   LocalFINcalled = 1;
601
602   /* Some data left in APP queue or network queue.*/
603   if(event & (APP_DATA|NETWORK_DATA) && ctx->connection_state == CSTATE_ESTABLISHED
       && RemoteFINcalled == 0)
604   {
605   gettimeofday(&tv, NULL);
606   abs_time = (struct timespec*)(&tv);
607   abs_time->tv_sec += 1;                    /* Wait for a sec */
608   }
609   /* No APP_Data in the app_queue remains. NOW send the FIN to peer */
610   else
611   {
612   /* Send FIN */
613   setSTCPheader(app_hdr, ctx->sequence_num, (ctx->ack_num + 1)%MAX_SEQ_NUM, OFFSET,
       0|TH_ACK|TH_FIN, ctx->local_advertised_window);
614
615   stcp_network_send(sd, app_hdr, HDR_SIZE, NULL);
```

```
616   bzero(app_hdr, HDR_SIZE);
617
618   FIN_seq_num = ctx->sequence_num;
619   ctx->sequence_num++;
620   ctx->connection_state = FIN_WAIT_1;
621
622   LocalFINcalled = 0;
623   abs_time = NULL;
624   }
625   }
626   /* etc. */
627   }
628
629   /* Clean Up */
630   if(app_hdr)
631   free(app_hdr);
632   if(network_hdr)
633   free(network_hdr);
634
635   app_hdr = NULL;
636   network_hdr = NULL;
637   }
638
639
640   /*****************************************************************/
641   /* our_dprintf
642   *
643   * Send a formatted message to stdout.
644   *
645   * format               A printf-style format string.
646   *
647   * This function is equivalent to a printf, but may be
648   * changed to log errors to a file if desired.
649   *
650   * Calls to this function are generated by the dprintf amd
651   * dperror macros in transport.h
652   */
653   void our_dprintf(const char *format,...)
654   {
655   va_list argptr;
656   char buffer[1024];
657
658   assert(format);
659   va_start(argptr, format);
660   vsnprintf(buffer, sizeof(buffer), format, argptr);
661   va_end(argptr);
662   fputs(buffer, stdout);
663   fflush(stdout);
664   }
665
666
667   void send_segment_to_app(mysocket_t sd, context_t *ctx, char *segment, size_t
       segment_size, STCPHeader *hdr)
668   {
```

```
669  assert(segment);
670  assert(ctx);
671  assert(hdr);
672
673  tcp_seq seq_num = myntohl(hdr->th_seq);
674  tcp_seq expected_seq_num = (ctx->ack_num +1)%MAX_SEQ_NUM;
675
676  /* sequence number is the same as we expected */
677  if(seq_num == expected_seq_num)
678  {
679  ctx->ack_num = (seq_num + segment_size -1)%MAX_SEQ_NUM;
680  stcp_app_send(sd, segment, segment_size);
681  ctx->last_byte_read = ctx->ack_num;
682  }
683  /* Sequence number less than expected */
684  else if(seq_num < expected_seq_num)
685  {
686  /* Case I: If there is some new data */
687  if((seq_num + segment_size -1) >= expected_seq_num)
688  {
689  ctx->ack_num = (seq_num + segment_size -1)%MAX_SEQ_NUM;
690  stcp_app_send(sd, segment + (expected_seq_num - seq_num), seq_num + segment_size -
       expected_seq_num);
691  ctx->last_byte_read = ctx->ack_num;
692  }
693  /* Case II: No new data */
694  else
695  {
696  ; /* Duplicate Data */
697  }
698  }
699  /* Wrap Around */
700  else if(seq_num > expected_seq_num && (seq_num + segment_size -1)%MAX_SEQ_NUM >=
     expected_seq_num)
701  {
702  ctx->ack_num = (seq_num + segment_size -1)%MAX_SEQ_NUM;
703  stcp_app_send(sd, segment + (MAX_SEQ_NUM - seq_num + expected_seq_num),
     segment_size - (MAX_SEQ_NUM - seq_num + expected_seq_num));
704  ctx->last_byte_read = ctx->ack_num;
705  }
706  else
707  {
708  /* Such case where sequence number > expected_seq_num can never occur since
     packets are assumed to be in order */
709  printf("Must not be printed as packets are assumed to be in order \n");
710  }
711  }
712
713
714  void printContext(context_t *ctx)
715  {
716  printf("########################Context#########################\n");
717  printf("ctx->sequence_num = %u\n", ctx->sequence_num);
718  printf("ctx->last_byte_acked = %u\n",ctx->last_byte_acked);
```

```
719   printf("ctx->ack_num = %u\n", ctx->ack_num);
720   printf("ctx->last_byte_read = %u\n", ctx->last_byte_read);
721   printf("ctx->remote_advertised_window = %u\n", ctx->remote_advertised_window);
722   printf("ctx->local_advertised_window = %u\n", ctx->local_advertised_window);
723   printf("##########################################################\n");
724   }
725
726
727   void printSTCPheader(STCPHeader *hdr)
728   {
729   printf("========================HEADER=================================\n");
730   printf("Sequence Number: %u\n", myntohl(hdr->th_seq));
731   printf("Ack Number: %u\n", myntohl(hdr->th_ack));
732   printf("Offset: %u\n", hdr->th_off);
733   printf("Flags: %u\n", hdr->th_flags);
734   printf("Window: %u\n", myntohs(hdr->th_win));
735   printf("================================================================\n");
736
737   return;
738   }
739
740
741   void getSTCPheader(STCPHeader *hdr, char *buf)
742   {
743   assert(hdr);
744   bcopy(buf, hdr, HDR_SIZE);
745   return;
746   }
747
748
749   void setSTCPheader(STCPHeader *hdr, tcp_seq seq, tcp_seq ack, uint32_t data_offset
        , uint8_t flag, uint16_t win)
750   {
751   hdr->th_seq = myhtonl(seq);
752   hdr->th_ack = myhtonl(ack);
753   hdr->th_off = data_offset;
754   hdr->th_flags = flag;
755   hdr->th_win = myhtons(win);
756   return;
757   }
758
759
760   uint16_t getLocalWindowSize(context_t *ctx)
761   {
762   assert(ctx);
763
764   if(ctx->ack_num < ctx->last_byte_read)                    /* Wrap around */
765   return (uint16_t)(ctx->local_advertised_window - (MAX_SEQ_NUM - (ctx->
        last_byte_read - ctx->ack_num)));
766   else
767   return (uint16_t)(ctx->local_advertised_window - (ctx->ack_num - ctx->
        last_byte_read));
768   }
769
```

```
770
771 uint16_t getRemoteWindowSize(context_t *ctx)
772 {
773 assert(ctx);
774
775 if(ctx->sequence_num < ctx->last_byte_acked)          /* Wrap around */
776 return (uint16_t)(ctx->remote_advertised_window - (MAX_SEQ_NUM - (ctx->
     last_byte_acked - ctx->sequence_num)));
777 else
778 return (uint16_t)(ctx->remote_advertised_window - (ctx->sequence_num - ctx->
     last_byte_acked));
779 }
780
```

Listing 1: Transport Layer