Chpt 18 – TCP Connection Management

The "three−way handshake" used to establish a connection.

```
                              +---------+ ---------\      active OPEN
                              |  CLOSED |            \    -----------
                              +---------+<---------\   \   create TCB
                                |     ^              \   \  snd SYN
                   passive OPEN |     |   CLOSE        \   \
                   ------------ |     | ----------       \   \
                    create TCB  |     | delete TCB         \   \
                                V     |                      \   \
                              +---------+            CLOSE    |    \
                              |  LISTEN |          ---------- |     |
                              +---------+          delete TCB |     |
                   rcv SYN      |     |     SEND              |     |
                  -----------   |     |    -------            |     V
 +---------+      snd SYN,ACK  /       \   snd SYN          +---------+
 |         |<-----------------           ------------------>|         |
 |   SYN   |                    rcv SYN                     |   SYN   |
 |   RCVD  |<-----------------------------------------------|   SENT  |
 |         |                    snd ACK                     |         |
 |         |------------------           -------------------|         |
 +---------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +---------+
   |           --------------   |     |   -----------
   |                  x         |     |     snd ACK
   |                            V     V
   |  CLOSE                   +---------+
   | -------                  |  ESTAB  |
   | snd FIN                  +---------+
   |                   CLOSE    |     |    rcv FIN
   V                  -------   |     |    -------
 +---------+          snd FIN  /       \   snd ACK          +---------+
 |  FIN    |<-----------------           ------------------>|  CLOSE  |
 | WAIT-1  |------------------                              |   WAIT  |
 +---------+          rcv FIN  \                            +---------+
   | rcv ACK of FIN   -------   |                            CLOSE  |
   | --------------   snd ACK   |                           ------- |
   V        x                   V                           snd FIN V
 +---------+                  +---------+                   +---------+
 |FINWAIT-2|                  | CLOSING |                   | LAST-ACK|
 +---------+                  +---------+                   +---------+
   |                rcv ACK of FIN |                 rcv ACK of FIN |
   |  rcv FIN       -------------- |    Timeout=2MSL -------------- |
   |  -------              x       V    ------------        x       V
    \ snd ACK                 +---------+delete TCB         +---------+
     ------------------------>|TIME WAIT|------------------>| CLOSED  |
                              +---------+                   +---------+
```

TCP Connection State Diagram
Figure 6.

Objectives of connection management

Agree on initial sequence numbers
Exchange "options" such as MSL
Detect and recover from
  half open connections caused by crashes
  delayed duplicate packets

Connection process has 4 states:

Closed:	Not involved in session activation or in an established session:
SYN−Sent:	Actively trying to open a session.
SYN−Recvd:	Received a SYN and responded with an ACK:
Established:	SYN Acked.

Normally initiated by one TCP and responded to by another TCP.
Also works if two TCP simultaneously initiate the procedure.
Initialization packets carry:

Seq #, Ack #
  Initial sequence # selection is based on a clock.
  The procedure can help minimize the delayed duplicate problem
  (but its not fully realized in TCP)

Flag bits:

SYN ==> The seq # specified is my initial seq #.
ACK ==> The ack # specified is acking your choice of sequence #.
RST ==> Error!
  In an initializing state return to LISTEN.
  In a connected state abort the connection.

Normal connection establishment:

```
1.  CLOSED                                                LISTEN

2.  SYN-SENT    --> <SEQ=100><CTL=SYN>                --> SYN-RECEIVED

3.  ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

4.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>       --> ESTABLISHED

5.  ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED
```

Simultaneous connection establishment:

```
      TCP A                                                TCP B

   1.  CLOSED                                               CLOSED

   2.  SYN-SENT     --> <SEQ=100><CTL=SYN>                  ...

   3.  SYN-RECEIVED <-- <SEQ=300><CTL=SYN>                  <-- SYN-SENT

   4.               ... <SEQ=100><CTL=SYN>                  --> SYN-RECEIVED

   5.  SYN-RECEIVED --> <SEQ=100><ACK=301><CTL=SYN,ACK> ...

   6.  ESTABLISHED  <-- <SEQ=300><ACK=101><CTL=SYN,ACK> <-- SYN-RECEIVED

   7.               ... <SEQ=101><ACK=301><CTL=ACK>     --> ESTABLISHED
```

Recovery from old duplicate SYN.

```
       TCP A                                                   TCP B

   1.  CLOSED                                                  LISTEN

   2.  SYN-SENT    --> <SEQ=100><CTL=SYN>                  ...

   3.  (duplicate) ... <SEQ=90><CTL=SYN>                 --> SYN-RECEIVED

   4.  SYN-SENT    <-- <SEQ=300><ACK=91><CTL=SYN,ACK>  <-- SYN-RECEIVED

   5.  SYN-SENT    --> <SEQ=91><CTL=RST>                 --> LISTEN


   6.              ... <SEQ=100><CTL=SYN>                  --> SYN-RECEIVED

   7.  SYN-SENT    <-- <SEQ=400><ACK=101><CTL=SYN,ACK>  <-- SYN-RECEIVED

   8.  ESTABLISHED --> <SEQ=101><ACK=401><CTL=ACK>        --> ESTABLISHED
```

At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP B, on receiving the RST, returns to the LISTEN state.

When the original SYN (pun intended) finally arrives at line 6, the synchronization proceeds normally. *If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RST's sent in both directions.*

Dealing with half open connections:

```
    TCP A                                               TCP B

1.  (CRASH)                                        (send 300,receive 100)

2.  CLOSED                                              ESTABLISHED

3.  SYN-SENT -->         <SEQ=400><CTL=SYN>      --> (??)

4.  (!!)     <--     <SEQ=300><ACK=100><CTL=ACK> <--  ESTABLISHED

5.  SYN-SENT --> <SEQ=100><CTL=RST>              --> (Abort!!)

6.  SYN-SENT                                          CLOSED

7.  SYN-SENT --> <SEQ=400><CTL=SYN>              -->
```

When the SYN arrives at line 3, TCP B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half−open connection. TCP B aborts at line 5. TCP A will continue to try to establish the connection; the problem is now reduced to the basic 3−way handshake of figure 7. What if the RST should be lost? Then TCP B will remain in the established state. Eventually TCP A will retransmit and the scenario will be repeated. What if TCP A gives up? Then TCP B will remain in the established state until TCP B tries to send some data.

Dealing with delayed dups (part 2):

```
    TCP A                                               TCP B

1.  (send 100,receive 300)                         (send 300,receive 100)

2.  ESTABLISHED                                           ESTABLISHED

3.  (delayed dup) --> <SEQ=400><CTL=SYN>           --> (??)

4.                <-- <SEQ=300><ACK=100><CTL=ACK>  <-- ESTABLISHED

5.  ESTABLISHED                                           ESTABLISHED
```

When the SYN arrives at line 3, TCP B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment reflects its current view of the state of the connection, ignores it, and remains in the established state.

Active Side Causes Half−Open Connection Discovery:

```
      TCP A                                                  TCP B

  1.  (CRASH)                                        (send 300,receive 100)

  2.  (??)    <-- <SEQ=300><ACK=100><DATA=10><CTL=ACK> <-- ESTABLISHED

  3.          --> <SEQ=100><CTL=RST>                   --> (ABORT!!)
```

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection. This is illustrated in figure 11. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

Dealing with delayed SYNs:

```
    TCP A                                               TCP B

   1.  LISTEN                                             LISTEN

   2.       ... <SEQ=Z><CTL=SYN>                  -->   SYN-RECEIVED

   3.  (??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>   <--   SYN-RECEIVED

   4.       --> <SEQ=Z+1><CTL=RST>                -->   (return to LISTEN!)

   5.  LISTEN                                             LISTEN
```

We see the two TCPs A and B with passive connections waiting for SYN. An old duplicate arriving at TCP B (line 2) stirs B into action. A SYN−ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP B accepts the reset and returns to its passive LISTEN state.

Connection termination protocol

Disconnection is more complicated

It has 6 states
Requires 4 messages

Connections are *duplex* and each end can terminate independently

(After one direction is closed, data can continue to flow in the other direction!)
Receipt of FIN => no more data will be *received* but doesn't preclude additional sends

The *shutdown(int s, int how)* system call can be used to effect a unidirectional close. If how is 0, further receives will be disallowed. If how is 1, further sends will be disallowed. If how is 2, further sends and receives will be disallowed.

The *close(int s)* system call forces closing of both directions.

Termination states

FIN_WAIT_1:	Have sent FIN but not yet received an ack for it
FIN_WAIT_2:	Have received an ACK from my FIN but haven't received FIN from the other end yet.
TIME_WAIT:	Have sent and received FIN and have ACKed other end's FIN (State occupancy time is 2MSL).
CLOSING:	Received FIN from the other end before receiving ACK of my FIN from the other end (simultaneous close).
CLOSE_WAIT:	Received FIN from the other end and am waiting for close on my end. When I get the close I'll send my FIN.
LAST_ACK:	Sent the final FIN (I know its final because I already received FIN from the other end) and am waitng for ACK.

Exercise: Suppose an application is blocked on a read at the time a FIN is received. Does the receipt of the FIN cause the read to complete with an EOF indication??

Normal Close Seqeuence

```
    TCP A                                                   TCP B

1.  ESTABLISHED                                             ESTABLISHED

2.  (Close)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

3.  FIN-WAIT-2  <-- <SEQ=300><ACK=101><CTL=ACK>      <-- CLOSE-WAIT

4.                                                      (Close)
    TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

5.  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>      --> CLOSED

6.  (2 MSL)
    CLOSED
```

Simultaneous Close Sequence

```
    TCP A                                                   TCP B

1.  ESTABLISHED                                             ESTABLISHED

2.  (Close)                                                 (Close)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  ... FIN-WAIT-1
                <-- <SEQ=300><ACK=100><CTL=FIN,ACK>  <--
                ... <SEQ=100><ACK=300><CTL=FIN,ACK>  -->

3.  CLOSING     --> <SEQ=101><ACK=301><CTL=ACK>      ... CLOSING
                <-- <SEQ=301><ACK=101><CTL=ACK>      <--
                ... <SEQ=101><ACK=301><CTL=ACK>      -->

4.  TIME-WAIT                                           TIME-WAIT
    (2 MSL)                                             (2 MSL)
    CLOSED                                              CLOSED
```

The 2 MSL wait

Designed to prevent the delayed duplicate problem
A *connection* can't be reused until the 2 MSL (up to 4 minutes depending on system has expired)

Unfortunately some implementations use *port* rather than *connection.*

Not a problem for if the client initiates the close (since clients use ephemeral ports)
Potential big problem if server should initiate the close (since servers use well known ports)

Some TCP's allow recycling of port in the 2 MLS state *if* the initial sequence # is greater than the final number of the previous connection.

Exercise: What pitfall of this practice is discussed in RFC 1185

TCP Server Design

```
/* rcmdserv.c */

/* This program is an example of how a remote command server */
/* like rsh might work.                                      */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>

char buffer[200];

struct hostent *hp;
struct hostent *gethostbyname();
struct sockaddr_in name;
struct sockaddr_in client;
extern int errno;
int srvpid;

main(argc, argv)
int argc;
char *argv[];
{
   int i;
   int session = 0;
   unsigned char c;
   int sock;
   int msgsock;
   int status;
   int ramt;
   int wamt;
   int namesize;
   int hostaddr;

/* Create a stream (TCP) socket */

   srvpid = getpid();
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0)
   {
      printf("Socket create failed \n");
      exit(1);
   }
```

This server uses the strategy, previously declared to be unclever, of binding the local port to a local interface address!

```
/* Get nameserver to convert host name to address */

   hp = gethostbyname(argv[1]);
   if (hp == 0)
   {
      printf("Host %s not found", argv[1]);
      exit(2);
   }

/* Fill in host and port address in name structure */

   bcopy((char *)hp->h_addr,
            (char *)&name.sin_addr, hp->h_length);
   name.sin_family = AF_INET;
   name.sin_port = htons(atoi(argv[2]));

/* Bind socket to network,port address */

   status = bind(sock, (struct sockaddr *)&name, sizeof(name));
   printf("Bind status = %d \n", status);
   if (status < 0)
     exit(3);

/* Signify availabilty for network connections */

   status = listen(sock, 4);
   if (status < 0)
   {
      printf("Listen failed with code %d. \n", errno);
      exit(4);
   }
```

In the main loop the server waits for connection requests and the forks a new instance of itself to handle the service.

```
   namesize = sizeof(name);
   while(1)
   {

   /* Accept a connection from the network */
   /* Address of the connecting party will be in name after */
   /* the connection is established.                         */

      status = accept(sock, (struct sockaddr *)&name, &namesize);
      if (status < 0)
      {
         printf("Accept  failed with code %d. \n", errno);
         exit(5);
      }

   /* Value returned by accept is a handle for reading or writing */

      printf("Accept status = %d \n", status);
      msgsock = status;

   /* Invoke the remote command processor */

      session += 1;
      if (fork() == 0)
      {
         rcmd(msgsock, session);
         exit(6);
      }
   }
}
```

```
/* This is the remote command server.. It reads commands */
/* from a connected client, redirects the standard out   */
/* back across the net and executes the commands.        */

rcmd(comfile, id)
int comfile;
int id;
{
   int infile;
   int amtread;
   int killrc;
   int *loc = 0;
   char msgbuf[10];
   int  i;

/* Redirect standard output back to network connection */

   sprintf(msgbuf, "SVR%d>", id);
   close(1);
   dup(comfile);

   while(1)
   {
      write(comfile, msgbuf, 6);

   /* @ informs the other end its ok to send.. Its a SNA-like  */
  /* "Dataflow" protocol.                                      */

      write(comfile, "@", 1);
      amtread = read(comfile, buffer, sizeof(buffer));
      *(buffer + amtread - 1) = 0;
      fprintf(stderr, "Cmd was: %s \n", buffer);

   /* Session level protocol ... x ==> terminate this session */
   /*                            z ==> terminate server.      */
      if (buffer[0] == 'x')
      {
         close(comfile);
         close(1);
         return(1);
      }

      if (buffer[0] == 'z')
      {

         close(comfile);
         killrc = kill(getppid(), SIGKILL);
         fprintf(stderr, "Kill RC = %d \r\n", killrc);
         return(-1);
      }

    /* Execute the command that was transferred. */

      system(buffer);
   }
}
```