Chapter 18:
TCP Connection Establishment and Termination

Introduction

- TCP is a *connection-oriented* protocol. Before either end can send data to the other, a *connection* must be established between them.
- This establishment of a connection between the two ends differs from a *connectionless* protocol such as UDP.

Connection Establishment and Termination

- **Scenario**
  - We type the following command on the system svr4:

```
svr4 % telnet bsdi discard
Trying 140.252.13.35 …
Connected to bsdi.
Escape character is '^]'.
^]
telnet> quit
connection closed.
```


Connection Establishment and Termination (Cont.)

- ***tcpdump* Output**

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                  win 4096 <mss 1024>
2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                  ack 1415531522 win 4096
                                                  <mss 1024>
3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1823083522 win 4096
4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1415531522:1415531522(0)
                                                  ack 1823083522 win 4096
5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 1415531523 win 4096
6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1823083522:1823083522(0)
                                                  ack 1415531523 win 4096
7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 1823083523 win 4096
```

Figure 18.1 tcpdump output for TCP connection establishment and termination.

Connection Establishment and Termination (Cont.)

- ***flag* characters output by *tcpdump* for flag bits in TCP header**

flag	3-character abbreviation	Description
S	SYN	synchronize sequence numbers
F	FIN	sender is finished sending data
R	RST	reset connection
P	PSH	push data to receiving process as soon as possible
.		none of above four flags is on

Figure 18.2 *flag* characters output by tcpdump for flag bits in TCP header.

- **The other two TCP header flag bits - ACK and URG - are printed specially by *tcpdump*.**

Connection Establishment and Termination (Cont.)

- **Time Line**

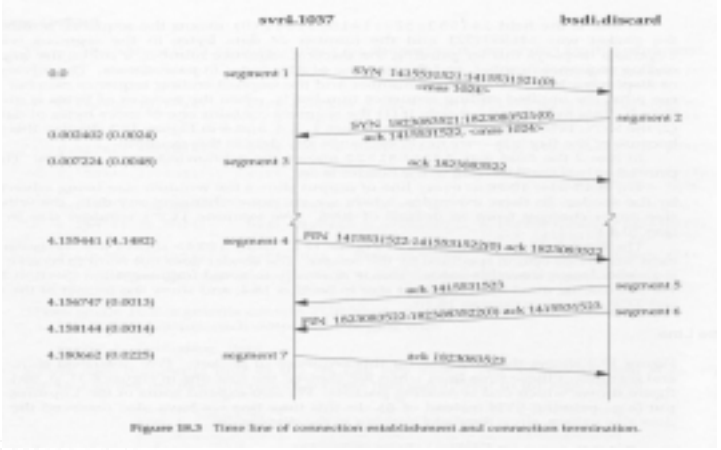


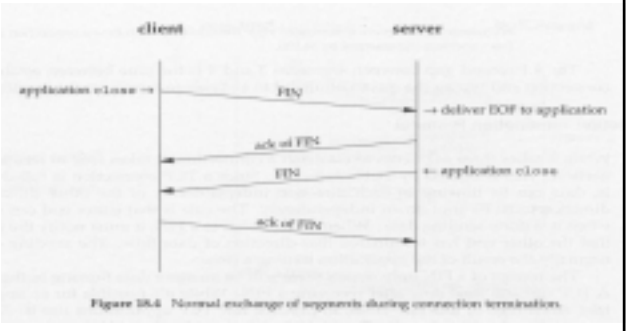
Figure 18.3 Time line of connection establishment and connection termination.

Connection Establishment and Termination (Cont.)

- **Connection Establishment Protocol**
  - *Three-way handshaking*:
    - The client sends a SYN segment specifying the port number of the server that the client wants to connect to, and the client's ISN.
    - The server responds with its own SYN segment containing the server's ISN. The server also acknowledges the client's SYN by ACKing the client's ISN plus one. A SYN consumes one sequence number.
    - The client must acknowledge this SYN from the server by ACKing the server's ISN plus one.
  - Active open
  - Passive open
  - The ISN should change over time, so that each connection has a different ISN.
- **ISN: To prevent packets delayed in the network from being delivered later and then misinterpreted as part of an existing connection.**

Connection Establishment and Termination (Cont.)

- **Connection Termination Protocol**



  - half-close
  - active close
  - passive close
- **The receipt of a FIN only mean there will be no more data flowing in that direction.**

Connection Establishment and Termination (Cont.)

- A TCP can still send data after receiving a FIN.
- A FIN consumes a sequence number, just like a SYNC.
- Sending the FINs is caused by the ***applications*** closing their end of connection, whereas the ACKs of these FINs are ***automatically*** generated by the ***TCP*** software.

Connection Establishment and Termination (Cont.)

- Normal *tcpdump* Output

```
1  0.0                    svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>
2  0.002402 (0.0024)      bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522
                                                    win 4096 <mss 1024>
3  0.007224 (0.0048)      svr4.1037 > bsdi.discard: . ack 1 win 4096
4  4.155441 (4.1482)      svr4.1037 > bsdi.discard: F 1:1(0) ack 1 win 4096
5  4.156747 (0.0013)      bsdi.discard > svr4.1037: . ack 2 win 4096
6  4.158144 (0.0014)      bsdi.discard > svr4.1037: F 1:1(0) ack 2 win 4096
7  4.180662 (0.0225)      svr4.1037 > bsdi.discard: . ack 2 win 4096
```

Figure 18.5 Normal tcpdump output for connection establishment and termination.

Timeout of Connection Establishment

- **Scenario**
  - There are several instances when the connection cannot be established. To simulate this scenario we issue our telnet command after disconnecting the Ethernet cable from the server's host.
  - tcpdump output

```
1   0.0              bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                win 4096 <mss 1024>
                                                [tos 0x10]
2   5.814797 ( 5.8148)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                win 4096 <mss 1024>
                                                [tos 0x10]
3  29.815436 (24.0006)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                win 4096 <mss 1024>
                                                [tos 0x10]
```

Figure 18.6 tcpdump output for connection establishment that times out.

Timeout of Connection Establishment (Cont.)

- To see this we have to time the telnet command:

```
bsdi % date ; telnet svr4 discard ; date
Thu Sep 24 16:24:11 MST 1992
Trying 140.252.13.34 …
telnet: Unable to connect to remote host: Connection timed out
Thu Sep 24 16:25:27 MST 1992
```

- The time difference is 76 seconds. Most Berkeley-derived system set a time limit of 75 seconds on the establishment of a new connection.

Timeout of Connection Establishment (Cont.)

- **First Timeout Period**
  - What's happening here is that BSD implementations of TCP run a timer that goes off every 500 ms.

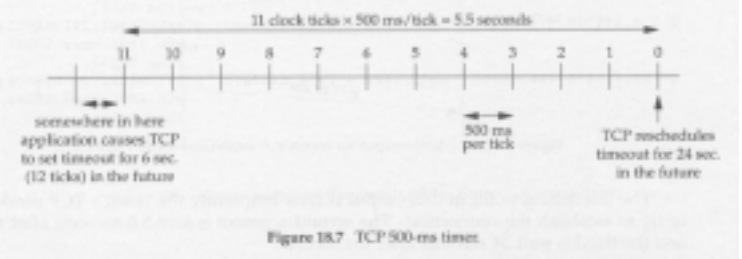


Figure 18.7 TCP 500-ms timer.

- **Type-of-Service Field**
  - The BSD/386 Telnet client sets the field for minimum delay.

Maximum Segment Size

- **The MSS is the largest "chunk" of data that TCP will send to the other end.**
- **When a connection is established, each end has the option of announcing the MSS it expects to receive.**
- **In general, the larger the MSS the better, until fragmentation occurs.**
- **If the destination IP address is "nonlocal", the MSS normally default to 536.**
- **Example**

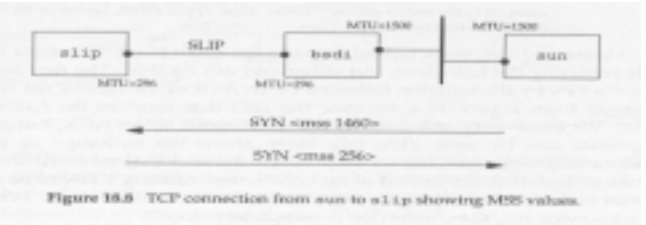


Figure 18.8 TCP connection from sun to slip showing MSS values.

Maximum Segment Size (Cont.)

- We initiate a TCP connection from sun to slip and watch the segment using tcpdump.

```
1  0.0                  sun.1093 > slip.discard: S 517312000:517312000(0)
                                                 <mss 1460>
2  0.10 (0.00)          slip.discard > sun.1093: S 509556225:509556225(0)
                                                 ack 517312001 <mss 256>
3  0.10 (0.00)          sun.1093 > slip.discard: . ack 1
```

Figure 18.9 tcpdump output for connection establishment from sun to slip.

- It's OK for a system to send less than the MSS announced by the other end.
- If both hosts are connected to Ethernets, and both announce an MSS of 536, but an intermediate network the MTU of 296, fragmentation will occur. The only way around this is the path MTU discovery mechanism.

TCP Half-Close

- Half close: TCP provides the ability for one end of a connection to terminate its output, while still receiving from the other end.
- A typical scenario for a half close:

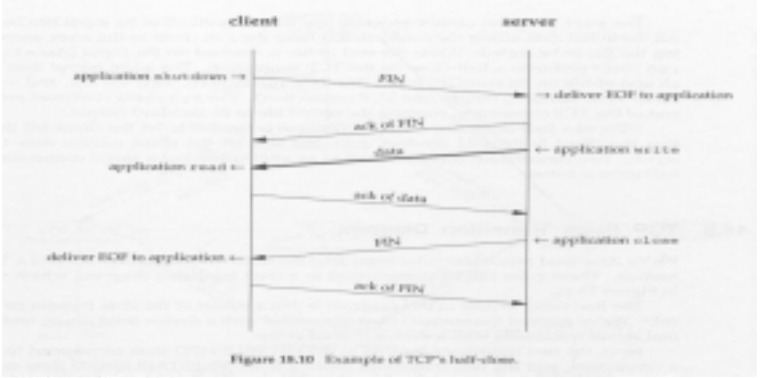


Figure 18.10 Example of TCP's half-close.

TCP Half-Close (Cont.)

- **Why is there a half-close? One example is the Unix *rsh(1)* command, which executes a command on another system. The command**

  sun % rsh bsdi sort < datafile

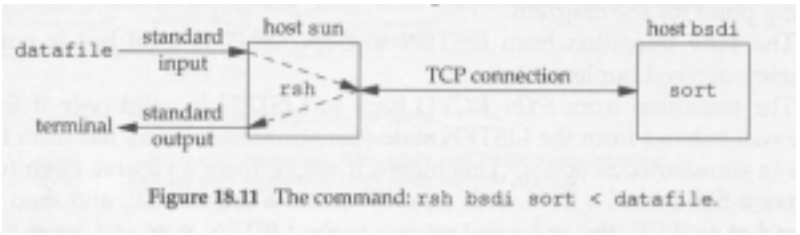
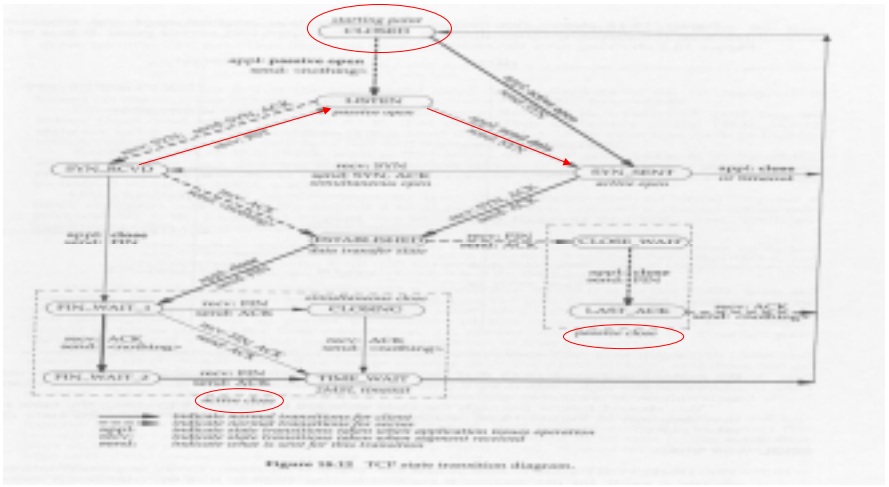


Figure 18.11 The command: `rsh bsdi sort < datafile`.

TCP State Transition Diagram



TCP State Transition Diagram (Cont.)

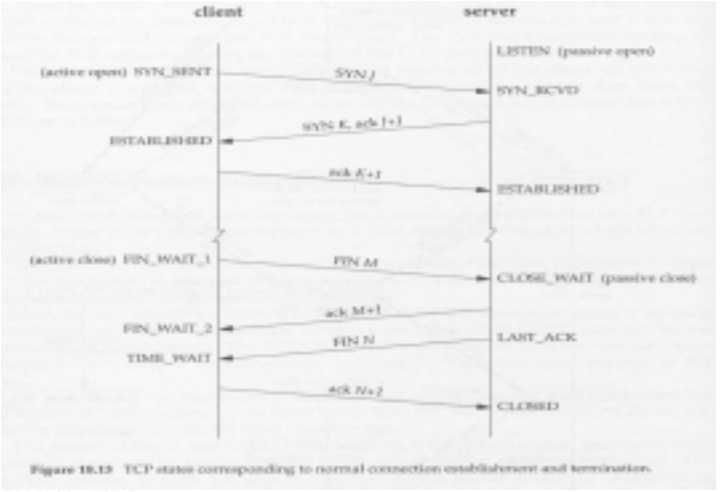


Figure 18.13 TCP states corresponding to normal connection establishment and termination.

2MSL Wait State

- The TIME_WAIT state is also called the 2MSL wait state.
- *Maximum segment lifetime (MSL):* the maximal amount of time any segment existing in the network before being discarded.
- TCP segment are transmitted as IP datagrams, and the IP datagram has the TTL field that limits its lifetime.
- The rule is: when TCP performs an active close, and sends the final ACK, that connection must stay in the TIME_WAIT state for twice the MSL.
- Why 2MSL wait state?
  - Let TCP resend the final ACK in case this ACK is lost (the other end will time out and retransmit its final FIN).
  - The socket pair defining that connection cannot be reused, until the 2MSL wait is over.

2MSL Wait State (Cont.)

- What are the differences between IP TTL and TCP MSL?
- A local port number cannot be reused while that port number is the local port number of a socket pair that is in the 2MSL wait.
- Any delayed segments that arrive for a connection while it is in the 2MSL wait are discarded.
- In client-server model, the server usually does the passive close, and does not go through the TIME_WAIT state.

2MSL Wait State (Cont.)

- **Example I**
  - Scenario:

```
Sun % sock –v –s 6666
Connection on 140.252.13.33.6666 from 140.252.13.35.1081
^?
Sun % sock –s 6666
Can't bind local address: Address already in use
Sun % netstat
Active Internet connections
Proto   Recv-Q  Send-Q  Local Address      Foreign Address  (state)
Tcp     0       0       sun.6666           bsdi.1081
TIME_WAIT
```

2MSL Wait State (Cont.)

- **Example II**
  - Scenario

```
Sun % sock –v bsdi echo
Connected on 140.252.13.33.1162 to 140.252.13.35.7
Hello there
Hello there
^D
Sun % sock –b1162 bsdi echo
Can't bind local address: Address already in use
```


2MSL Wait State (Cont.)

- **Example III**
  - Scenario

```
Sun % sock –v –s 6666
Connection on 140.252.13.33.6666 from 140.252.13.35.1098
^?
Sun % sock –b6666 bsdi 1098
Can't bind  local address: Address already in use
Sun % sock –A –b6666 bsdi 1098
Active open error: Address already in use
```

SO_REUSEADDR

Quiet Time Concept

- The 2MSL wait provides protection against delayed segments from an earlier incarnation of a connection. But this works only if a host with connections in the 2MSL wait does not crash.
- What if a host with port in the 2MSL wait crashes, reboots within MSL seconds, and immediately establishes new connections using the same local and foreign IP address and port numbers corresponding to the local ports what were in the 2MSL wait before the crash?
- To protect against this scenario, RFC 793 states that TCP should not create any connections for MSL seconds after rebooting. It is called the ***quiet time***.

FIN_WAIT_2 State

- Unless we have done a half-close, we are waiting for the application on the other end to recognize that it has received an end-of-file notification and close its end of the connection, which sends us a FIN. Only when the process at the other end does this close will our end move from the FIN_WAIT_2 to the TIME_WAIT state.

Reset Segments

- **A bit in TCP header named RST is for "reset."**
- **In general, a reset is sent by TCP whenever a segment arrives that doesn't appear correct for the referenced connection.**
- **Connection Request to Nonexistent Port**
  - Scenario

```
Bsdi % telnet svr4 20000
Trying 140.252.13.34 …
telnet: Unable to connect to remote host: Connection refused
```

  - This message is output by the Telnet client immediately.
  - The packet exchange corresponding to this command

```
1  0.0                  bsdi.1087 > svr4.20000: S 297416193:297416193(0)
                                                 win 4096 <mss 1024>
                                                 [tos 0x10]
2  0.003771 (0.0038)    svr4.20000 > bsdi.1087: R 0:0(0) ack 297416194 win 0
```

Figure 18.14 Reset generated by attempt to open connection to nonexistent port.

Abort a Connection

- **Orderly release: FIN is sent after all previously queued data has been sent, and there is normally no loss of data.**
- **Abortive release**
  - Any queued data is thrown away and the reset is sent immediately.
  - The receiver of the RST can tell that the other end did an abort instead of a normal close.
- **Scenario**

```
Bsdi % sock –L0 svr4 8888
Hello, world
^D
```

Abort a Connection

- Show the tcpdump output

```
1  0.0                    bsdi.1099 > svr4.8888: S 671112193:671112193(0)
                                                  <mss 1024>
2  0.004975 (0.0050)      svr4.8888 > bsdi.1099: S 3224959489:3224959489(0)
                                                  ack 671112194 <mss 1024>
3  0.006656 (0.0017)      bsdi.1099 > svr4.8888: . ack 1
4  4.833073 (4.8264)      bsdi.1099 > svr4.8888: P 1:14(13) ack 1
5  5.026224 (0.1932)      svr4.8888 > bsdi.1099: . ack 14
6  9.527634 (4.5014)      bsdi.1099 > svr4.8888: R 14:14(0) ack 1
```

Figure 18.15 Aborting a connection with a reset (RST) instead of a FIN.

- The RST segment elicits no response from the other end – it is not acknowledged at all. The receiver of the reset aborts the connection and advises the application that the connection was reset.

Abort a Connection

- We get the following error on the server for this exchange:

```
Svr4 % sock –s 8888
Hello, world
Read error: Connection reset by peer
```

Detecting Half-Open Connections

- **Scenario: we'll execute the Telnet client on *bsdi*, connecting to the discard server on svr4. We type one line of input, and watch it and reboot the server host. This simulates the sever host crashing.**

```
Bsdi % telnet svr4 discard
Trying 140.252.13.34 …
Connected to svr4.
Escape character is '^]'.
Hi  there
                                        here is where we reboot the server
   host
Another line                            and this one elicits a reset
Connection closed by foreign host.
```


- Tcpdump output

```
1   0.0                  bsdi.1102 > svr4.discard: S 1591752193:1591752193(0)
2   0.004811 (  0.0048)  svr4.discard > bsdi.1102: S 26368001:26368001(0)
                                                    ack 1591752194
3   0.006516 (  0.0017)  bsdi.1102 > svr4.discard: . ack 1

4   5.167679 (  5.1612)  bsdi.1102 > svr4.discard: P 1:11(10) ack 1
5   5.201662 (  0.0340)  svr4.discard > bsdi.1102: . ack 11

6 194.909929 (189.7083)  bsdi.1102 > svr4.discard: P 11:25(14) ack 1
7 194.914957 (  0.0050)  arp who-has bsdi tell svr4
8 194.915678 (  0.0007)  arp reply bsdi is-at 0:0:c0:6f:2d:40
9 194.918225 (  0.0025)  svr4.discard > bsdi.1102: R 26368002:26368002(0)
```

Figure 18.16 Reset in response to data segment on a half-open connection.

Simultaneous Open

- **What is Simultaneous Open?**
  - It is possible, although improbable, for two applications to both perform an active open to each other at the same time.
  - Each end must transmit a SYN, and the SYNs must pass each other on the network.
  - It also requires each end to have a local port number that is well known to the other end.

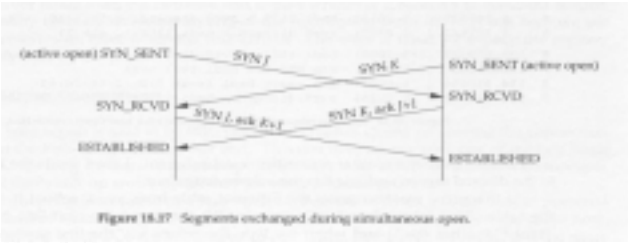


Figure 18.17 Segments exchanged during simultaneous open.

Simultaneous Open (Cont.)

- **An Example**
  - Scenario: We'll execute one end on our host bsdi, and the other end on the host bangogh.cs.berkeley.edu. Since there is a dialup SLIP link between them, the round-trip time should be long enough to let the SYNs cross.

Simultaneous Open (Cont.)

- On bsdi:

```
Bsdi % sock –v –b8888 vangogh.cs.berkeley.edu 7777
Connected on 140.252.13.35.8888 to 128.32.130.2.7777
TCP_MAXMEG = 512
Hello, world
And hi there
Connection closed by peer
```

- On the other end:

```
Vangogh % sock –v –b7777 bsdi.tuc.onao.edu 8888
Connected on 128.32.130.2.7777 to 140.252.13.35.8888
TCP_MAXMEG = 512
Hello, world
And hi there
^D
```


Simultaneous Open (Cont.)

- Tcpdump output

```
1   0.0                 bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                  win 4096 <mss 512>
2   0.213782 (0.2138)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                  win 8192 <mss 512>
3   0.215399 (0.0016)   bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                  ack 1058199042 win 4096
                                                  <mss 512>
4   0.340405 (0.1250)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                  ack 91904002 win 8192
                                                  <mss 512>
5   5.633142 (5.2927)   bsdi.8888 > vangogh.7777: P 1:14(13) ack 1 win 4096
6   6.100366 (0.4672)   vangogh.7777 > bsdi.8888: . ack 14 win 8192
7   9.640214 (3.5398)   vangogh.7777 > bsdi.8888: P 1:14(13) ack 14 win 8192
8   9.796417 (0.1562)   bsdi.8888 > vangogh.7777: . ack 14 win 4096
9  13.060395 (3.2640)   vangogh.7777 > bsdi.8888: F 14:14(0) ack 14 win 8192
10 13.061828 (0.0014)   bsdi.8888 > vangogh.7777: . ack 15 win 4096
11 13.079769 (0.0179)   bsdi.8888 > vangogh.7777: F 14:14(0) ack 15 win 4096
12 13.299940 (0.2202)   vangogh.7777 > bsdi.8888: . ack 15 win 8192
```

Figure 18.18 Exchange of segments during simultaneous open.

Simultaneous Close

- It is also possible for both sides to perform an active close, and the TCP protocol allow.

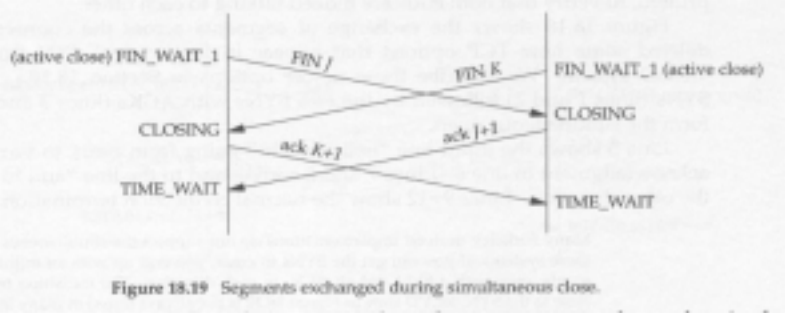


Figure 18.19 Segments exchanged during simultaneous close.

TCP Options

- The only options defined in the original TCP specification are the end of option list, no operation, and the maximum segment size option.

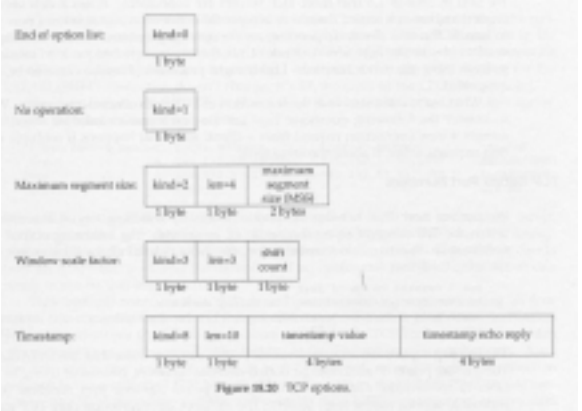


Figure 18.20 TCP options.

TCP Options (Cont.)

- kind: type of option
- len: the total length of option field, including *kind* and *len* bytes.
- NOP: no operation option is to allow the sender to pad fields to a multiple of 4 bytes.
- Example:

  <mss 512, nop, wscale 0, nop, nop, timestamp 146647 0>

TCP Server Design

- When a new connection request arrives at a server, the server accepts the connection and invokes a new process to handle the new client.
- Under Unix the common technique is to create a new process using the fork function. Ligntweight processes (threads) can also be used, if supported.
- We need to answer the following questions: how are the port numbers handled when a server accepts a new connection request from a client, and what happens if multiple connection requests arrive at about the same time.

TCP Server Design (Cont.)

- **TCP Server Port Numbers**
  - We'll watch the Telnet server using the netstat command.

```
Sun % netstat –a –n –f inet
Active Internet connections (including servers)
Proto     Recv-Q  Send-Q  Local Address      Foreign Address    (state)
Tcp       0       0       *.23               *.*                LISTEN
```

  - We now start a Telnet client on the host slip (140.252.13.65) that connects to this server.

```
Proto     Recv-Q  Send-Q  Local Address      Foreign Address    (state)
Tcp       0       0       140.252.13.33.23   140.252.13.65.1029 ESTABLISH
Tcp       0       0       *.23               *.*                LISTEN
```

TCP Server Design (Cont.)

- **Restricting Local IP Address**
  - If we specify an IP address to our sock program when we invoke it as a server, that IP address becomes the local IP address of the listening end point

    Sun % **sock –s 140.252.1.29 8888**

    restricts this server to connections arriving on the SLIP interface (140.252.1.29). The netsate output:

```
Proto     Recv-Q  Send-Q  Local Address      Foreign Address    (state)
Tcp       0       0       140.252.1.29.8888  *.*                LISTEN
```

  - If we connect to this server across the SLIP link, from the host solaris

```
Proto     Recv-Q  Send-Q  Local Address      Foreign Address    (state)
Tcp       0       0       140.252.1.29.8888  140.252.1.32.34614 ESTABLISH
Tcp       0       0       140.252.1.29.8888  *.*                LISTEN
```

TCP Server Design (Cont.)

- **But if we try to connect to this server from a host on the Ethernet (140.252.13), the connection request is not accepted by the TCP module.**

```
1  0.0                  bsdi.1026 > sun.8888: S 3657920001:3657920001(0)
                                                win 4096 <mss 1024>
2  0.000859 (0.0009)    sun.8888 > bsdi.1026: R 0:0(0) ack 3657920002 win 0
```

Figure 18.21 Rejection of a connection request based on local IP address of server.

The server application never sees the connection request – the rejection is done by the kernel's TCP module.

TCP Server Design (Cont.)

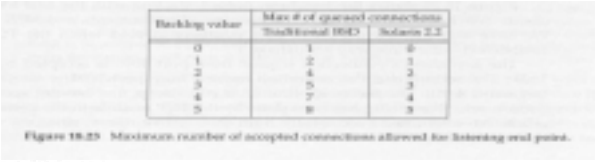
- **Restricting Foreign IP Address**
  - The interface functions shown RFC 793 allow a server doing a passive open to have either a fully specified foreign socket or a unspecified foreign socket.
  - Unfortunately, most APIs don't provide a way to do this.
  - The three types of address bindings that a TCP sever can establish for itself.

Local Address	Foreign Address	Description
localIP.lport	*foreignIP.fport*	restricted to one client (normally not supported)
localIP.lport	*.*	restricted to connections arriving on one local interface: localIP
.lport	*.*	receives all connections sent to lport

Figure 18.22 Specification of local and foreign IP addresses and port number for TCP server.

TCP Server Design (Cont.)

- **Incoming Connection Request Queue**
  - How dose TCP handle these incoming connection requests while the listening application is busy?
    - Each listening end point has a fixed length queue of connections that have been accepted by TCP, but not yet accepted by the application.
    - The application specifies a limit to this queue, commonly called the backlog. This backlog must be between 0 and 5, inclusive.
    - When a connection request arrives, an algorithm is applied by TCP to the current number of connections already queued for this listening end point, to see whether to accept the connection or not.



Backlog value	Max # of queued connections	
	Traditional BSD	Solaris 2.1
0	1	0
1	2	1
2	4	2
3	5	3
4	7	4
5	8	5

Figure 18.23 Maximum number of accepted connections allowed for listening end point.

TCP Server Design (Cont.)

- If there is room on this listening end point's queue for this new connection, the TCP module ACKs the SYN and completes the connection.
- If there is no room on the queue for the new connection, TCP just ignores the received SYN.
- If the listening server doesn't get around to accepting some of the already accepted connections that have filled its queue to the limit, the client's active open will eventually time out.

- An example:

bsdi % **sock -s -v -q1 -o30 7777**

Set the backlog of the listening end point to 1

Cause the program to sleep for 30 seconds before accepting any client connections.

TCP Server Design (Cont.)

- The *tcpdump* output

```
 1   0.0                 sun.1090 > bsd1.7777: S 1617152000:1617152000(0)
 2   0.002310 ( 0.0023)  bsd1.7777 > sun.1090: S 4164096001:4164096001(0)
                                                 ack 1617152001
 3   0.003098 ( 0.0008)  sun.1090 > bsd1.7777: . ack 1

 4   4.291007 ( 4.2879)  sun.1091 > bsd1.7777: S 1617792000:1617792000(0)
 5   4.293349 ( 0.0023)  bsd1.7777 > sun.1091: S 4164672001:4164672001(0)
                                                 ack 1617792001
 6   4.294167 ( 0.0008)  sun.1091 > bsd1.7777: . ack 1

 7   7.131981 ( 2.8378)  sun.1092 > bsd1.7777: S 1618176000:1618176000(0)
 8  10.556783 ( 3.4248)  sun.1093 > bsd1.7777: S 1618688000:1618688000(0)
 9  12.695916 ( 2.1391)  sun.1092 > bsd1.7777: S 1618176000:1618176000(0)
10  16.195772 ( 3.4999)  sun.1093 > bsd1.7777: S 1618688000:1618688000(0)
11  24.695571 ( 8.4998)  sun.1092 > bsd1.7777: S 1618176000:1618176000(0)

12  28.195454 ( 3.4999)  sun.1093 > bsd1.7777: S 1618688000:1618688000(0)
13  28.197810 ( 0.0024)  bsd1.7777 > sun.1093: S 4167808001:4167808001(0)
                                                 ack 1618688001
14  28.198619 ( 0.0008)  sun.1093 > bsd1.7777: . ack 1

15  48.694931 (20.4963)  sun.1092 > bsd1.7777: S 1618176000:1618176000(0)
16  48.697292 ( 0.0024)  bsd1.7777 > sun.1092: S 4170496001:4170496001(0)
                                                 ack 1618176001
17  48.698145 ( 0.0009)  sun.1092 > bsd1.7777: . ack 1
```

Figure 18.24 tcpdump output for backlog example.

Summary

- Before two processes can exchange data using TCP, they must establish a connection between themselves.
- We used tcpdump to show all the fields in the TCP header.
- Fundamental to understanding the operation of TCP is its state transition diagram.
- A TCP connection is uniquely defined by a 4-tuple: the local IP address, local port number,foreign IP address, and foreign port number.