

Exercise #1

2.1 We have mentioned IP versions 4 and 6. What happened to version 5 and what were versions 0, 1, 2, and 3? (Hint: Find the IANA's "Internet Protocol" registry. Feel free to skip ahead to the solution if you cannot visit <http://www.iana.org>.)

Ans: Visit <http://www.iana.org/numbers.htm> and find the registry called "IP Version Number." Version 0 is reserved, versions 1 -3 are unassigned, and version 5 is the Internet Stream Protocol.

2.2 Where would you look to find more information about the protocol that is assigned IP version 5?

Ans: All RFCs are available at no charge through electronic mail, anonymous FTP, or the Web. A starting point is <http://www.ietf.org>. The directory <ftp://ftp.rfc-editor.org/in-notes> is one location for RFCs. To start, fetch the current RFC index, normally the file `rfc-index.txt`, also available in an HTML version at <http://www.rfc-editor.org/rfc-index.html>. If we search the RFC index (see the solution to the previous exercise) with an editor of some form, looking for the term "Stream," we find that RFC 1819 defines Version 2 of the Internet Stream Protocol. Whenever looking for information that might be covered by an RFC, the RFC index should be searched.

2.3 With Figure 2.15, we said that TCP assumes an MSS of 536 if it does not receive an MSS option from the peer. Why is this value used?

Ans: With IPv4, this generates a 576-byte IP datagram (20 bytes for the IPv4 header and 20 bytes for the TCP header), the minimum reassembly buffer size with IPv4.

2.5 A connection is established between a host on an Ethernet, whose TCP advertises an MSS of 1,460, and a host on a Token Ring, whose TCP advertises an MSS of 4,096. Neither host implements path MTU discovery. Watching the packets, we never see more than 1,460 bytes of data in either direction. Why?

Ans: The host on the token ring cannot send packets with more than 1,460 bytes of data because the MSS it received was 1,460. The host on the Ethernet can send packets with up to 4,096 bytes of data, but it will not exceed the MTU of the outgoing interface (the Ethernet) to avoid fragmentation. TCP cannot exceed the MSS announced by the other end, but it can always send less than this amount.

3.1 Why must value-result arguments such as the length of a socket address structure be passed by reference?

Ans: In C, a function cannot change the value of an argument that is passed by value. For a called function to modify a value passed by the caller requires that the caller pass a pointer to the value to be modified.

3.2 Why do both the `readn` and `writen` functions copy the `void*` pointer into a `char*` pointer?

Ans: The pointer must be incremented by the number of bytes read or written but C does not allow a void pointer to be incremented (since the compiler does not know the data type pointed to).

4.1 In Section 4.4, we stated that the `INADDR_` constants defined by the `<netinet/in. h>` header are in host byte order. How can we tell this?

Ans: Look at the definitions for the constants beginning with `INADDR_` except `INADDR_ANY` (which is all zero bits) and `INADDR_NONE` (which is all one bits). For example, the class D multicast address `INADDR_MAX_LOCAL_GROUP` is defined as `0xe00000ff` with the comment “224.0.0.255”, which is clearly in host byte order.

4.2 Modify Figure 1.5 to call `getsockname` after `connect` returns successfully. Print the local IP address and local port assigned to the TCP socket using `sock_ntop`. In what range (Figure 2.10) are your system's ephemeral ports?

Ans: Here are the new lines added after the call to `connect`:

```
len = sizeof(cliaddr);
Getsockname(sockfd, (SA *) &cliaddr, &len);
printf("local addr: %s\n", Sock_ntop((SA *) &cliaddr, len)) ;
```

This requires a declaration of `len` as a `socklen_t` and a declaration of `cliaddr` as a `struct sockaddr_in`. Notice that the value-result argument for `getsockname(len)` must be initialized before the call to the size of the variable pointed to by the second argument. The most common programming error with value-result arguments is to forget this initialization.

4.3 In a concurrent server, assume the child runs first after the call to `fork`. The child then completes the service of the client before the call to `fork` returns to the parent. What happens in the two calls to `close` in Figure 4.13?

Ans: When the child calls `close`, the reference count is decremented from 2 to 1, so a FIN is not sent to the client. Later, when the parent calls `close`, the reference count is decremented to 0 and the FIN is sent.

4.4 In Figure 4.11, first change the server's port from 13 to 9999 (so that we do not need superuser privileges to start the program). Remove the call to `listen`. What happens?

Ans: `accept` returns `EINVAL`, since the first argument is not a listening socket.

4.5 Continue the previous exercise. Remove the call to `bind`, but allow the call to `listen`.

Ans: Without a call to `bind`, the call to `listen` assigns an ephemeral port to the listening socket.

5.1 Build the TCP server from Figures 5.2 and 5.3 and the TCP client from Figures 5.4 and 5.5. Start the server and then start the client. Type in a few lines to verify that the client and server work. Terminate the client by typing your EOF character and note the time. Use `netstat` on the client host to verify that the client's end of the connection goes through the `TIME_WAIT` state. Execute `netstat` every five seconds or so to see when the `TIME_WAIT` state ends. What is the MSL for this implementation?

Ans: The duration of the `TIME_WAIT` state should be between 1 and 4 minutes giving an MSL between 30 seconds and 2 minutes.

5.3 What is the difference between our echo client/server and using the Telnet client to communicate with our echo server?

Ans: Telnet converts the input lines into NVT ASCII (Section 26.4 of TCPv1), which terminates every line with the two-character sequence of a CR (carriage return) followed by an LF (linefeed). Our client adds only a newline, which is actually a linefeed character. Nevertheless, we can use the Telnet client to communicate with our server as our server echoes back every character, including the CR that precedes each newline.

5.4 In our example in Section 5.12, we verified that the first two segments of the connection termination are sent (the FIN from the server that is then ACKed by the client) by looking at the socket states using `netstat`. Are the final two segments exchanged (a FIN from the client that is ACKed by the server)? If so, when, and if not, why?

Ans: No, the final two segments of the connection termination sequence are not sent. When the client sends the data to the server, after we kill the server child (the "another line"), the server TCP responds with an RST. The RST aborts the connection and also prevents the server end of the connection (the end that did the active close) from passing through the `TIME_WAIT` state.

5.5 What happens in the example outlined in Section 5.14 if between Steps 2 and 3 we restart our server application on the server host?

Ans: Nothing changes because the server process that is started on the server host creates a listening socket and is waiting for new connection requests to arrive. What we send in Step 3 is a data segment destined for an ESTABLISHED TCP connection. Our server with the listening socket never sees this data segment, and the server TCP still responds to it with an RST.

5.6 To verify what we claimed happens with `SIGPIPE` in Section 5.13, modify Figure 5.4 as follows: Write a signal handler for `SIGPIPE` that just prints a message and returns. Establish this signal handler before calling `connect`. Change the server's port number to 13, the daytime server. When the connection is established, `sleep` for two seconds, write a few bytes to the socket, `sleep` for another two seconds, and write a few more bytes to the socket. Run the program. What happens?

Ans: Figure E.I shows the program. Running this program under Solaris generates the following:

```
Solaris % tsigpipe 192.168.1.10
SIGPIPE received
write error: Broken pipe
```

The initial `sleep` of two seconds is to let the daytime server send its reply and close its end of the connection. Our first write sends a data segment to the server, which responds with an RST (since the daytime server has completely closed its socket). Note that our TCP allows us to write to a socket that has received a FIN. The second `sleep` lets the server's RST be received, and our second write generates `SIGPIPE`. Since our signal handler returns, `write` returns an error of `EPIPE`.

5.8 In our example output from Figure 5.20, when the client and server were on different endian systems, the example worked for small positive numbers, but not for small negative numbers. Why? (Hint: Draw a picture of the values exchanged across the socket similar to Figure 3.9.)

Ans: Our client was on a little-endian Intel system, where the 32-bit integer with a value of 1 was stored as shown in Figure E.2.

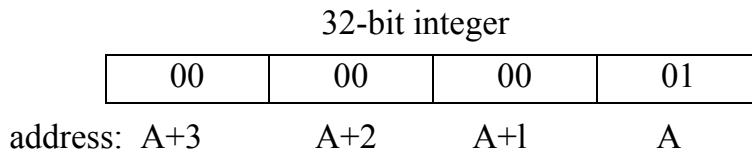


Figure E.2 Representation of the 32-bit integer 1 in little-endian format.

The 4 bytes are sent across the socket in the order A, A+1, A+2, and A+3 where they are stored in the big-endian format, as shown in Figure E.3.

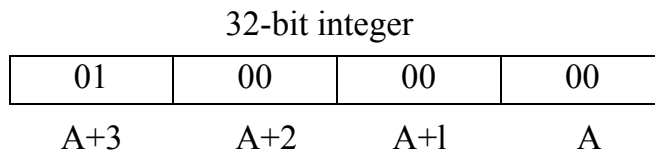


Figure E.3 Representation of the 32-bit integer from Figure E.2 in big-endian format.

This value of 0x01000000 is interpreted as 16,777,216. Similarly, the integer 2 sent by the client will be interpreted at the server as 0x02000000, or 33,554,432. The sum of these two integers is 50,331,648, or 0x03000000. When this big-endian value on the server is sent to the client, it is interpreted on the client as the integer value 3.

The 32-bit integer value of -22 is represented on the little-endian system as shown in Figure E.4, assuming a two's-complement representation of negative numbers.

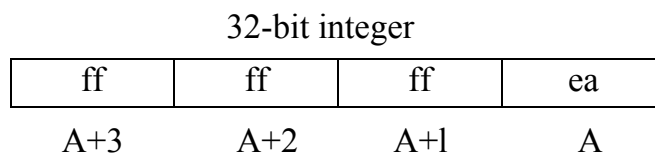


Figure E.4 Representation of the 32-bit integer -22 in little-endian format.

This is interpreted on the big-endian server as `Oxeaffffff`, or -352,321,537. Similarly, the little-endian representation of -77 is `Oxfffffb3`, but this is represented on the big-endian server as `Oxb3ffffff`, or -1,275,068,417. The addition on the server yields a binary result of `Ox9effffff`, or -1,627,389,954. This big-endian value is sent across the socket to the client where it is interpreted as the little-endian value `Oxffff9e`, or -16,777,314, which is the value printed in our example.

5.10 What happens in Figures 5.19 and 5.20 if the client is on a SPARC that stores a long in 32 bits, but the server is on a Digital Alpha that stores a long in 64 bits? Does this change if the client and server are swapped between these two hosts?

Ans: In the first scenario, the server blocks forever in the call to `readn` in Figure 5.20 because the client sends two 32-bit values but the server is waiting for two 64-bit values. Swapping the client and server between the two hosts causes the client to send two 64-bit values, but the server reads only the first 64 bits, interpreting them as two 32-bit values. The second 64-bit value remains in the server's socket receive buffer. The server writes back one 32-bit value and the client will block forever in its call to `readn` in Figure 5.19, waiting to read one 64-bit value.