

TCP Header: How it works and background information on the TCP protocol

Introduction

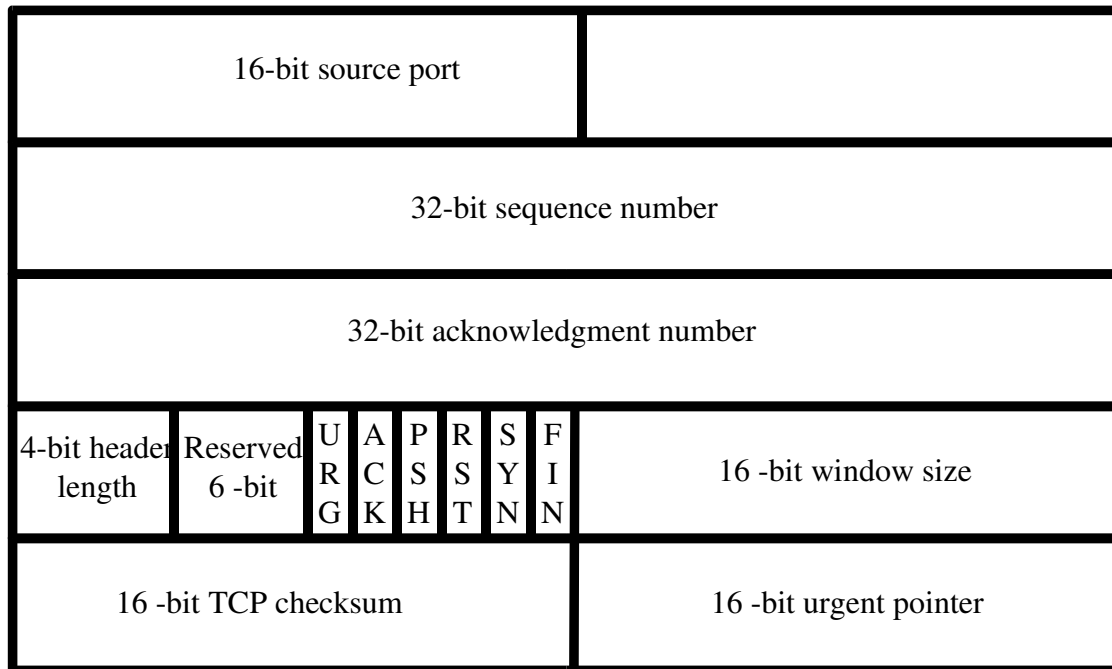
In the last section we talked about the UML tool. This section we will start to talk about the theory. I would like to give an overview of the TCP layer and then in the next section go into more detail of how the Linux kernel implements the TCP protocol. These next two sections are helpful to understanding the kernel modifications, the role each layer in the protocol stack, and how the Linux kernel handles each of these roles. Also the INET (Linux implementation of the TCP IP suite) is not the same as the standards and RFC' so some explanation is needed.

Overview

The TCP is the transport communication protocol. In the protocol stack it is four (transport) right above the Internet Protocol (IP). The TCP guarantees the arrival of the packets. The way the Linux kernel uses TCP is the packet arrives at the network card and the information (digital signals) is transferred to memory. The Linux kernel creates a structure in memory (called a `sk_buff`) which has pointers to the newly received digital transmission which is now in memory. Each transmission is a packet – a part of the original transmitted data with headers attached. Each packet (for our purposes) consist of three different types of headers (Ethernet, IP, TCP) and the actual data body. So if I am on the Internet and download the latest `root_fs` from http://user_mode_linux.sourceforge.org, the file would be 16 megs (for the newest slackware `root_fs`). Transferring all these files in one big packet (one set of headers) would be cumbersome. First, I would have to keep all the values together. If someone else wants to use the bandwidth, they would have to wait until my transmission of 16 megs. So we all can share bandwidth, the data is partition into small packets (roughly 1500 bits a piece) each having their own set of headers. These headers tell the packet where to go and in our case ensures the packet gets there. This is what the Ethernet and IP headers do. The Ethernet and IP headers contain sets of address (kind of like street address) to tell the packet where to go. The TCP has the harder job. Since our packet is divided into pieces and the Internet is not all the same bandwidth and reliability. Some packets will arrived mixed up, corrupted, twice or not at all. The TCP manages the arrival of the packets and makes sure the application receives the data in order. On the packet, the TCP header is the mechanism which is allows the INET code to do this.

TCP Header and Sk_buff

TCP' header information shown in the graphic below:



Above is the TCP header in the Linux code. The total length of the TCP header is 20 bytes; four bytes for each row. Just a reminder, 8 bits equal 1 byte. So the TCP header is actually 160 bits. Of those 160 bits, 64 bits are used for the sequence and acknowledgment numbers (32 bits for the sequence and 32 bits for the acknowledgment number). The Seq and Ack numbers are how the TCP keeps packets in order (just a bit useful information mentioned later).

So the header has all sorts of information. In the kernel, TCP header is accessed by pointer the `sk_buff` (stands for socket buffer). The `sk_buff` is probably the most important structure in the networking of the kernel. The `sk_buff` is essentially the incoming network packet. The `sk_buff` is a "c lang-struct" which has pointers to each of the segments in memory. Below is a code sniped:

```

struct sk_buff {
    /* These two members must be first. */
    struct sk_buff * next; /* Next buffer in list */
    struct sk_buff * prev; /* Previous buffer in list */

    struct sk_buff_head * list; /* List we are on */
    struct sock *sk; /* Socket we are owned by */
    struct timeval stamp; /* Time we arrived */
    struct net_device *dev; /* Device we arrived on/are leaving by */

    /* Transport layer header */
    union
    {
        struct tcphdr *th;
        struct udphdr *uh;
        struct icmphdr *icmph;
        struct igmpchr *igmpchr;
        struct iphdr *iph;
        struct spxhdr *spxh;
        unsigned char *raw;
    } h;

    /* Network layer header */
    union
    {
        struct iphdr *iph;
        struct ipv6hdr *ipv6h;
        struct arphdr *arph;
        struct ipxhdr *ipxh;
        unsigned char *raw;
    } nh;

    /* Link layer header */
    union
    {
        struct ethhdr *ethernet;
        unsigned char *raw;
    } mac;

    struct dst_entry *dst;

```

As you can see, the pointers are arranged in a union fashion for the tcp. If you look at the h union, h has many different choices one of them is tcphdr (TCP) another of them is udphdr (UDP). The sk_buff is able to handle the different types of protocols; the sk_buff is versatile in that it able to support all the different protocols in the protocol stack. I can talk all day about how well the sk_buff was designed, but the main point of this is the TCP.

If the sk_buff->h.th is opened up, we would see the tcphdr. The tcphdr contains the declaration of all the important header information. Looking at this code sniped, we can see that big endian and little endian have been taken into account. Based on the architecture type (in our case i386). The c-preprocessor automatically sets the tcp header

structure' s byte order correct.

```
struct tcphdr {
    __u16 source;
    __u16 dest;
    u32 seq;
    __u32 ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u16 res1:4,
        doff:4,
        fin:1,
        syn:1,
        rst:1,
        psh:1,
        ack:1,
        urg:1,
        ece:1,
        cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
    __u16 doff:4,
        res1:4,
        cwr:1,
        ece:1,
        urg:1,
        ack:1,
        psh:1,
        rst:1,
        syn:1,
        fin:1;
#else
#error    "Adjust your <asm/byteorder.h> defines"
#endif
    __u16 window;
    __u16 check;
    __u16 urg_ptr;
};
```

SMALL CODE SNIPED OF THE SEQ AND ACK circled

Establishing a TCP connection

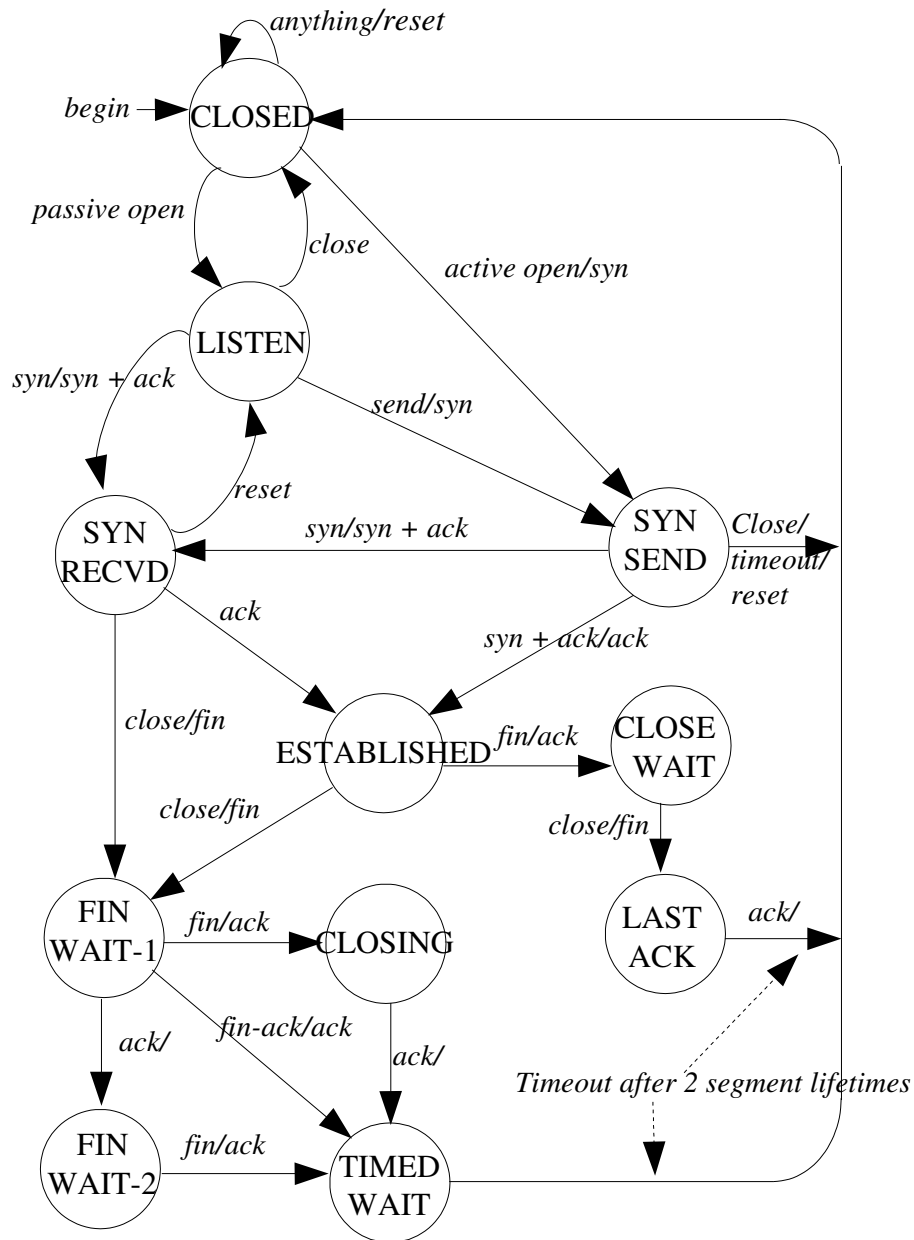
The sequence and acknowledgment numbers mentioned earlier manages the arrival and order of the packets. The way it does this is by labeling each packet that is sent with a beginning sequence number on each side. This number is incremented a known value and the receiver is expecting that value. If the packets become out of sequence, the receiver can tell by the sequence number.

The initial sequence number is determined by the sender. It creates a packet (sk_buff) and sends it down the protocol stack and out to the Internet. The TCP header has a section of one bit FLAGS. There are six flags: URG, ACK, PSH, RST, SYN, and FIN. The initial packet send by the sender not only has the initial sequence number, but also has the SYN flag set.

When the receiver gets the initial packet. It processes the packet and sends its own sequence number and places the just received sequence number in the acknowledgment field. The sender then sets two flags in the TCP header: SYN and ACK.

When the receiver receives the SYN and ACK packet. It sends a packet with the ACK flag marked to acknowledge the SYN/ACK packet.

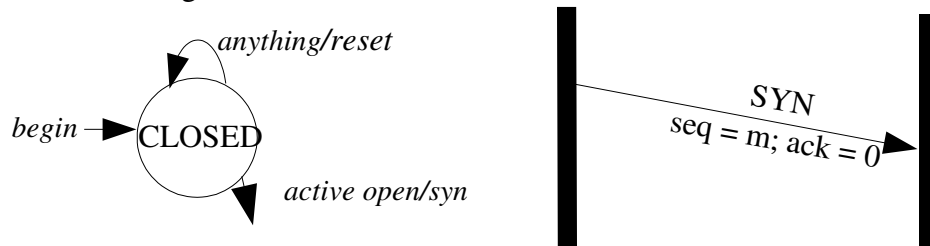
Below is a finite state diagram of the TCP establishing a connection.



TCP STATE DIAGRAM

Now this complex flowchart may seem intimidating at first, especially if you have never had Automaton theory class (or even taught it). This diagram is actually from the RFC (Request for comment – or the guidelines) for the TCP protocol. Let us go through this diagram for the section we just discussed (called the three-way handshake). We will use a simple web (http) web request as our example.

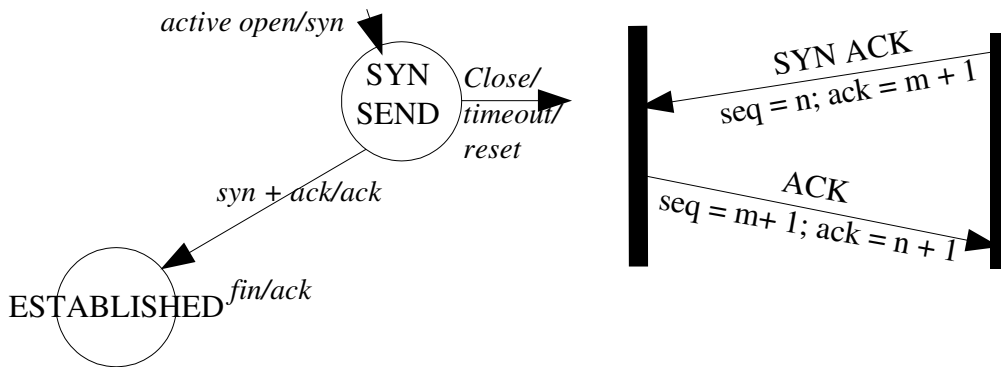
The three-way hand shake, actually named after the three packets sent back and forth, is used to establish the connect between two computers. Looking at the intimidating finite state diagram, we start at the “begin” label. One thing to keep in mind, this finite state diagram is for each side of the connection; one state diagram represents the sender and the other represents the receiver. So there are two flow charts or we have two positions on the finite state diagram.



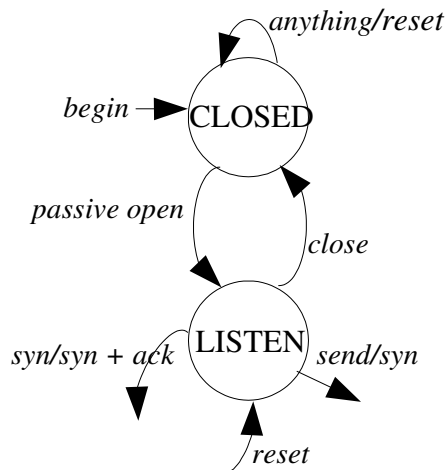
Starting with the sending side (since it is easier), we are at the “begin” label at the upper left. The TCP socket is in closed state. The user (at the computer) decides to open a web page, send some email, telnet, or something requiring a TCP connection. The socket receives a system call from the “userland” (believe or not many publications actually call the user level, “userland”). The socket then creates a packet and sets the SYN flag. From the “CLOSED” state we move to the “SYN SENT” state since a packet with a SYN flag was sent. The flags in the TCP header correlate to the SYN, ACK, FIN, and RST shown on the diagram. When a packet is sent with one of these flags. We traverse the state diagram from one state to another based on the flags on the packets we send. Also we populate the 32-bit sequence number field with an ‘m’ value (any value between $0-2^{32}$).

We just send a SYN packet and are in the SYN SENT state. From the line connecting the CLOSED and the SYN SENT state there is a label with two words “active open / syn”. This might be even more confusing. The right side of the front slash is what the sender is sending. The left side is either a state, in this case active open, or what was received from the other machine.

Now the sender is sitting at the SYN SEND state. It receives a packet with the SYN ACK flags set in the TCP header from the receiver. The sequence number we sent is now in the 32-bit acknowledge field with 1 added to ‘m’ ($m+1$). The socket is following our state diagram too (or programmed to) and then sends a packet with an ACK message. Now on the finite state diagram follows the line labeled ‘SYN + ACK/ ACK’ is at ESTABLISHED. The connection is established.

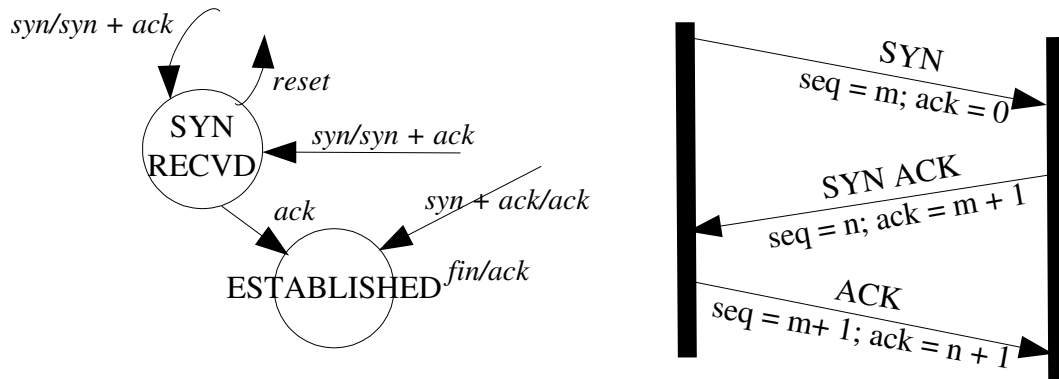


Now we will talk about the receiver' s diagram. The receiver starts the same place, the beginning label. Now the receiver needs to be in a listening state. This is because Apache or any other services depending on TCP cannot just automatically wake up and receive packets. The services (Apache, email, or telnet) are communicating to the kernel space through a socket. This socket needs to bound (this is where BIND comes in) to a port number. So when a packet with the correct port number comes in knows where to go (I am being repetitive – I know). If the TCP socket is not listening for this packet, it just zooms right by into the bit bucket. When the TCP socket is listening for the packet, it receives the packet and sends it up to the application (or service).



Main point is the socket needs to be in a listening state to receive packets going to a specific application. So from the CLOSED state, the socket does a “passive open” by order of a system call originating from the applications. So the TCP state moves from CLOSED state to LISTEN state by a “passive open”. The socket is now listening for any packet that comes in with the correct port number.

When the receiver gets the initial packet with the SYN flag set. It generates its own sequence number and places the received sequence number in the acknowledge field. The socket then sets the SYN ACK flags and sends the packets. Now the TCP socket moves to the SYN RECVD state.

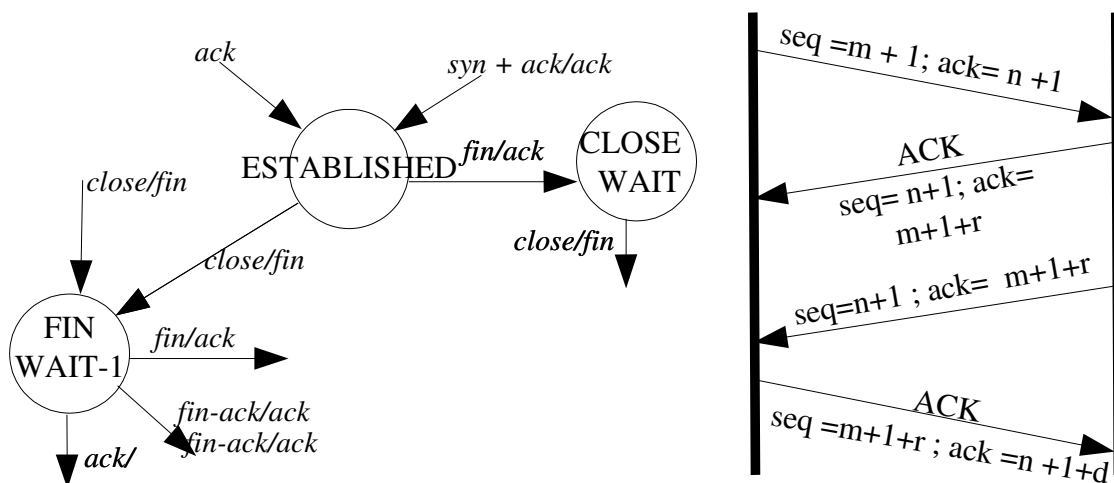


The sender will then send the receiver an ACK packet saying it has received the SYN ACK packet. The sender's socket then moves to the ESTABLISHED state.

Ok. That is a bit complicated. But that is what happens on both sides. Also, no information regarding the user's request is sent. So no requesting a webpage (http requests), no initial SMTP (Simple Mail Transport Protocol), no nothing from any server or client requesting information is sent in these three packets. These three packets are completely used for establishing a connection, for the sender's and receiver's TCP sockets to exchange sequence numbers and prepare for the actual data transfer.

Connection established

The transfer of requests and data between the server and client occurs after the connection is established. Based on the finite state diagram, both the sender and the receiver are on the ESTABLISHED state.



Now both the sender and receiver are on the ESTABLISHED state. They also send packets to each other (which we will talk about here in a moment), but their state does not change. When the sender sends a request and the receiver responds. The state does not change. The finite state diagram does not seem to cover the transfer of packets for the application. The finite state diagram only covers the connection and closing of the socket. I could be wrong. But all the different pieces of literature I have read does not explain how the finite state diagram accounts for the flags that are set during the packet transfer.

We are going to continue with our example. This example will use web requests for the packet transfer. We will assume the request and the web data is not more than 1500 bytes each (roughly the maximum number of bits a packet can send at any one time). So the webpage we are requesting does not have any packets and for this example, we will not have to fragment the packet and reassemble them. We will just talk about the sender (web client) sending one request and the receiver (web server) sending just one packet of web information back.

When the sender sends the request. It uses its sequence number (which is returned in the acknowledge, incremented by one). Our sequence number is $(m + 1)$, m being the original number sequence number.

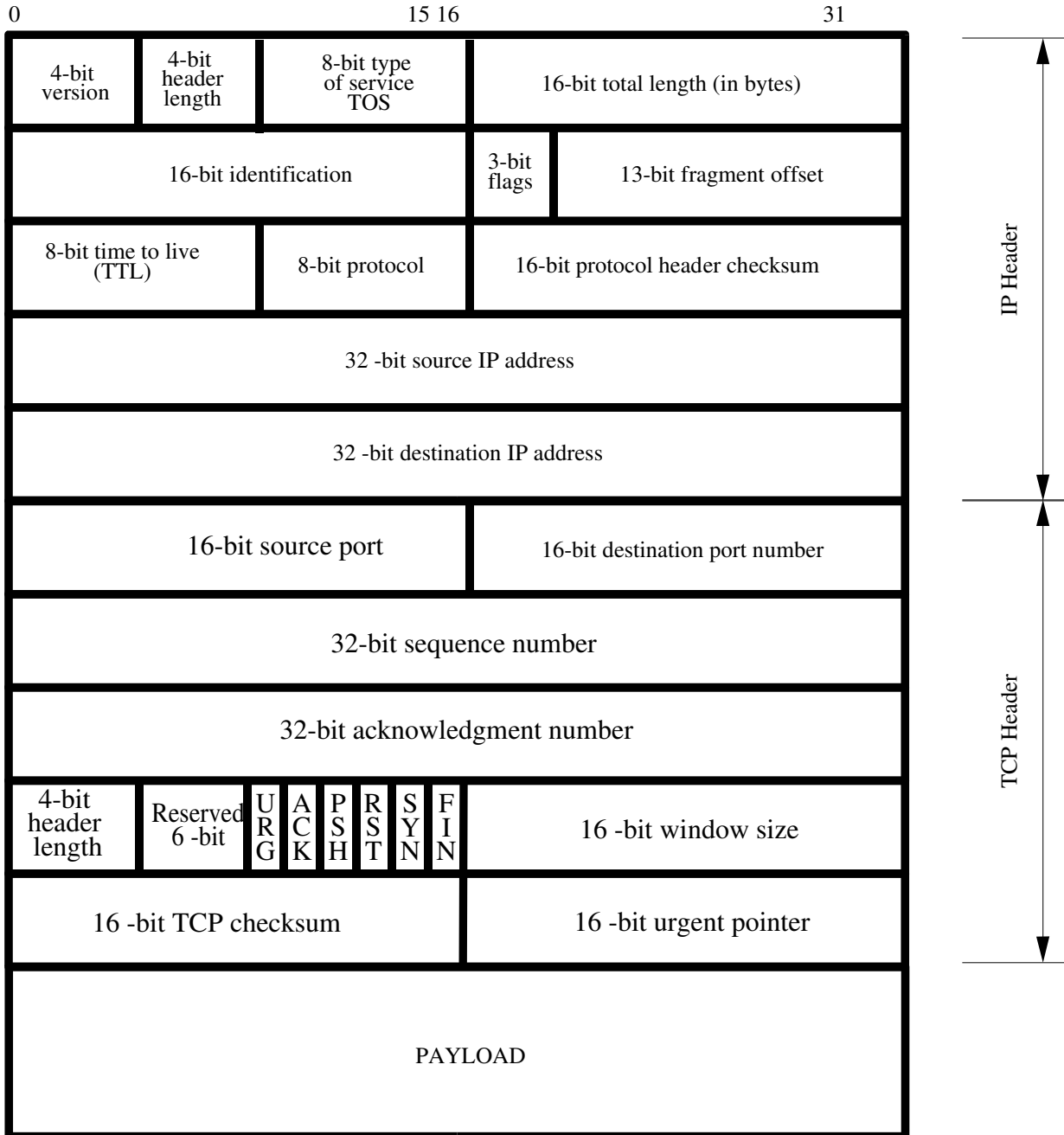
Just a little bit about how the TCP protocol increments sequence numbers. If you have not figured out by my writing so far. I am all about trying to find the pattern in something then explain the pattern. So for this incremental of the sequence number I am going to do just that, explain the pattern I see oppose to quoting something from Comer, Stevens, or an RFC (those authors are great by the way and highly recommend).

With the exception of sending data, as a general rule, you add one to the sequence number to packets with the SYN, both [SYN ACK], FIN, and [FIN ACK] flags set. This means you the only time you do NOT add one to the incoming sequence number is when the ACK flag is set. Fairly simple. Side note, almost half the packets have only the ACK field set, the other half you add one too the incoming sequence number. Generally adding one to the sequence number and then placing it in the acknowledge field tells the other side, you have received the packet. The adding one to the incoming sequence (outgoing acknowledge) is used by the socket to keep track that the packet has been received.

Here is another example up in the diagrams. Sender sends a packet with sequence number 1000 and acknowledge of 3000. Receiver receives the packets and takes the sequence number 1000 and places it in the acknowledge field and adds one, becomes 1001. The acknowledge field is now the sequence field, so the receiver is sending the sender his sequence number of 3000 (untouched) and an acknowledge of 1001. This is how the acknowledge and sequence numbers work for setting up and closing a connection. For sending and receiving data requests, the TCP does something a little

different to change the incoming sequence number.

Sending and receiving data the TCP uses the size of the data (or the size of the payload) to increment the sequence number. My guess is this is to save a spot on payload size and just put it in the sequence number, considering there is not such field for payload size. So just remember the incoming sequence number is updated based on the size of the data payload. Below, there is a more extensive picture of the TCP header with where the payload would be located. Generally, TCP packets are setup as: Ethernet header, IP header, TCP header, then data payload.



MORE EXTENSIVE HEADER OF THE IP, TCP , AND PAYLOAD.

The data payload is where the server to client data request and data responses are kept. Again, the three way handshake (three packets opening a connection) and four way close do not contain payloads; these packets are used merely for establishing and closing a connection.

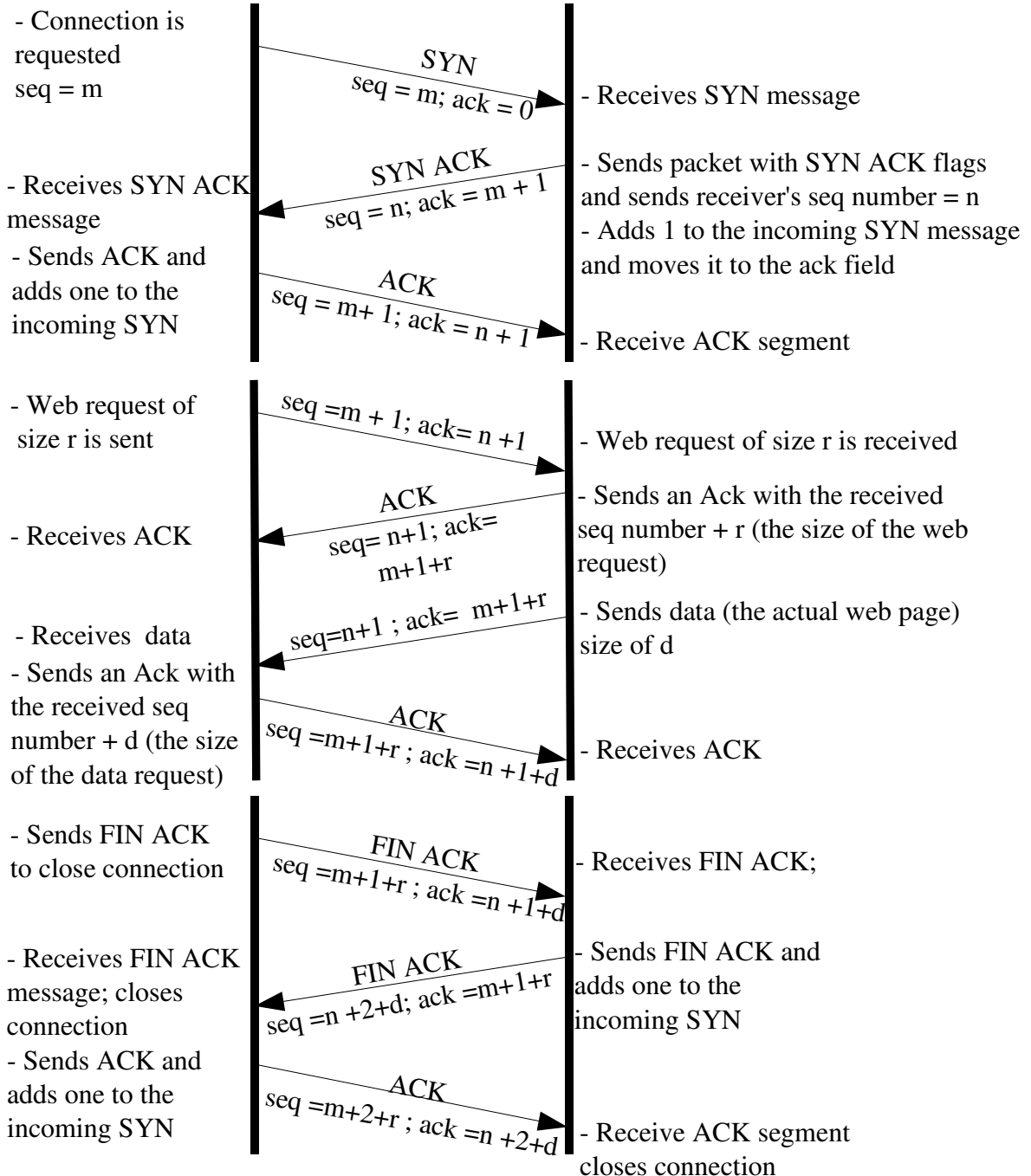


DIAGRAM OF THE TRANSFER OF PACKETS DIAGRAM WITH EXAMPLE NUMBERS

Now we will just give a quick example of how the incoming sequence number is updated. The sender sends a web request. Something like "GET / HTTP/1.0\r\n" to request a webpage. The size of the request is say, 190 bytes (actual size of a request taken from an ethereal capture of the network driver). Again, this web request is going to be in the payload. Let us continue with the example above and start with a sequence number of 1001 and an acknowledge number of 3001. When the web request is received by the sender. The receiver looks at the size of the payload (web request). The payload is 190 bytes. The receiver adds 190 to the incoming sequence number, sets the ACK flag, and places the incoming sequence number in the outgoing acknowledgment making it 1192 with an outgoing sequence number of 3001 (untouched).

The sender will then send the request up to userland and the application (in our case a web server) will process the request. Make a system call to send data out as a packet. The sender then sends another packet (two in a row) with the data requested (webpage). For this example, the webpage is only 598 bytes (from an ethereal network driver capture) and is only one packet since the entire size of the packet containing the data requested is under 1500 bytes. The sender sends the webpage data. The PSH and ACK flags are set. Why I am not sure. It is not in any of the references I have been reading, but the ethereal network tap shows these flags sets. I think very highly of Linux, but this might be somewhere where the network protocol (INET) differs from the RFC's and the commonly published standards.

The requested data is received by the receiver. The receiver then looks at the size of the payload (size of the webpage data) which is 598 bytes. The incoming sequence number is 3001. The sender takes the incoming sequence number and adds 598 for a total of 3599 and sets the ACK flag with an acknowledge field of 3599.

That is how a web request is processed and transferred using TCP. Couple of notes. In the appendix there is a sample network capture of a simple web request, if more hands on examples are needed. Just remember there are a minimum of four packets transferred when requesting a webpage, there are a minimum of ten packets including the TCP overhead packets of opening a closing a connection. But, sending the actual request requires four packets: two ACK's, one request, and one data packets. Remember that the only time the packet is updated is either when something other than an ACK (only ACK) packet is received. If the packet is received to setup a connection or close a connection. Only one is added the the incoming sequence number than transferred to the outgoing ACK packet in the acknowledgment field. For data, the size of the payload (either the data request or the actual data) is increment to the incoming sequence number and then placed in the acknowledgment field of the outgoing ACK packet. I have almost depleted the amount I can talk about regarding transferring a packet. The next section I will quick talk about closing a connection then fill in details about the other fields in the TCP header. Then the next chapter will take all the background information about the TCP and apply it to how the INET source code processes the packets.

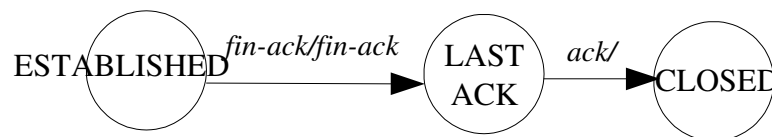
Closing the connection

Finally we reach the end of our connection. Closing a connection entails three packets being sent. This part the INET code probably did not follow standards, because I have studied the finite state diagram and does not match what the incoming packets suggest. Below is a modified finite state diagram with how the INET does this process. What I am going to do is just talk about what the INET does oppose to the actual published way of closing a connection. The thesis is to cover Linux and its implementation of the TCP protocol, so we will talk about it oppose to the standard published way.

There are three packets involved (published way there are 4). The sender sends a packet with the FIN and ACK flags set. Data transfer is done so there is no data in these packets. Once the receiver receives the FIN ACK packet, the receiver adds one to the incoming sequence number and sends and with the incremented incoming sequence number in the outgoing acknowledgment field, also sets the FIN ACK flags on the outgoing packet.

Once the receiver's FIN ACK packet is received by the sender. The receiver increments the incoming sequence number by one and sets the outgoing packet with the just the ACK flag. The sequence and acknowledgment numbers are switched and the packet is sent. The connection on the sender's side is then closed to the receiver.

Once the receiver gets the last ACK packet. The connection is closed on both sides. So the transfer of data through sockets can be partitioned into three parts: opening the connection, transferring the data, and closing the connection.



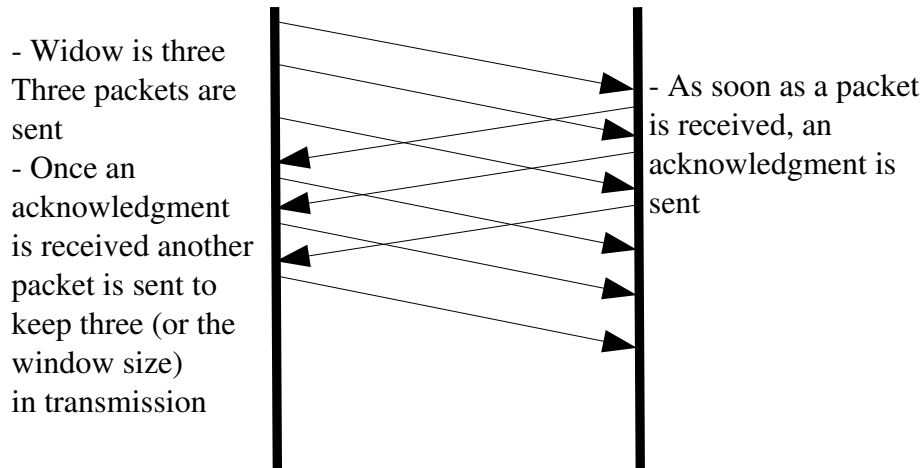
PICTURE OF THE MODIFIED CLOSE

Window

Up to this point, there has been a minimum of ten packets fired between the sender and receiver for one packet of information (little bit of overhead). About half the packets are acknowledged stating the packet has been received. The connection stays open for a set time called "Keep alive" it is actually programmed inside of the INET code. Strangely enough the INET code has timers, many timers. The finite state diagram even indicates the state can timeout and reset. Once a packet is fired (such as a SYN packet or a web request) a timer inside the INET code starts which schedules for the packet to be resent in the future. If the packet arrives the socket cancels the scheduled event. We will talk a little more about this in the next chapter.

The timers and the sequence in which packets arrive are all part of a bigger concept called "flow control". Flow control makes sure the packets arrive at a given rate but also in the

correct order, removing any duplicates and removing any erroneous packets. I briefly mentioned the sequence and acknowledge numbers are used for keeping the packets in order. Another field called the window (which is 16 bits). The window tells the other side how many packets that can be sent before an ACK packet is sent. For example, our small webpage request only had one packet. Assume for a second we are going to download a howto from the Linux document project (www.tldp.org). The webpage would contain pictures and pages and pages of very helpful information. When the connection is established (three-way handshake), the window is sent in the SYN ACK. The window is the amount of data that are allowed to be sent before receiving an acknowledgment. The INET has a lot of “s marts” built into it and can figure out how many packets and amount of information the computer can handle before problems occur. The sender or client is the one who sets up the initial window. Then both move the amount of data that can be sent over the network. The main point, the window is used for flow control. It controls the amount of data that can be sent by both sides. Below is a small diagram to illustrate.



WINDOW CONTROL DIAGRAM

In this diagram, we can see that the window is three packets. Only three packets can be sent before receiving an acknowledgment. When the acknowledgment for one of the packets is received another packet can be sent. If the window is three, only three packets can be sent without having an acknowledgment.

Checksum

We have talked about how the connections are opened and closed and how flow control is done. Now we move on to assuring the accuracy of the TCP header. There is another field called the checksum. Below is a diagram, from Comer's book, of the one's complement addition of the fields in the TCP header.

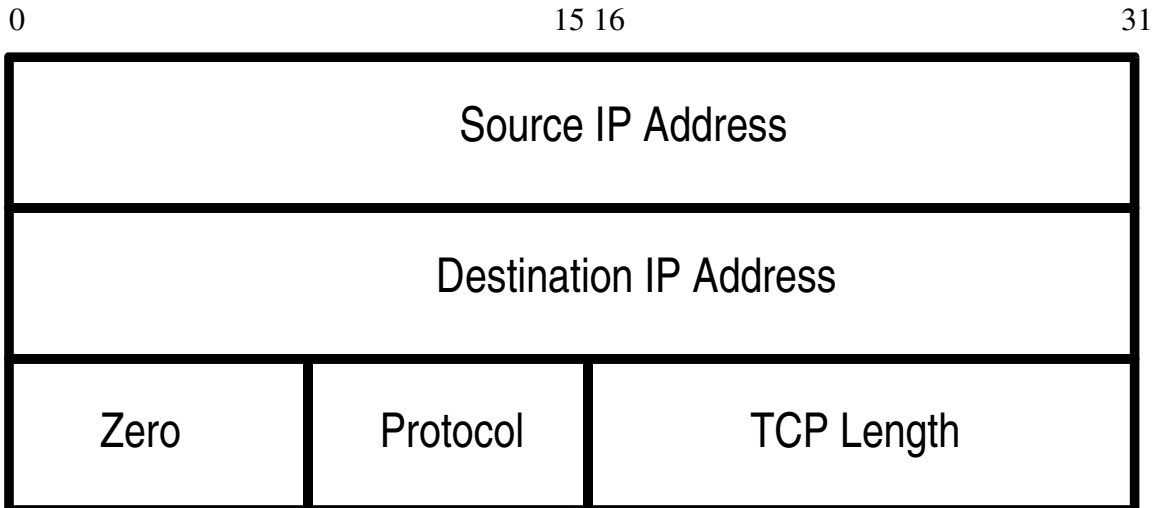
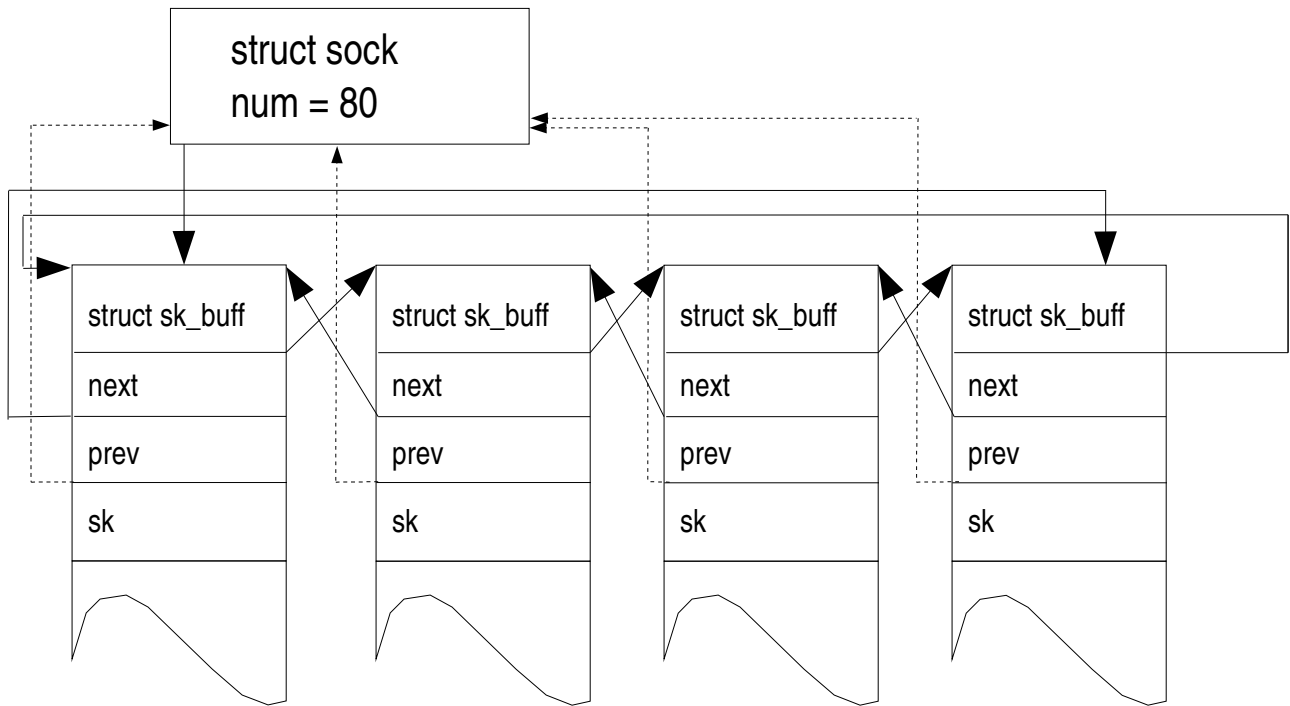


DIAGRAM OF CHECKSUM FIELDS

According to Comer (INWTCIP /IP 224) "To compute the checksum, TCP software on the sending machine follows a procedure like the one described in Chapter 12 for UDE. It prepends a pseudo header to a segment, appends enough zero bits to make the segment a multiple of 16 bits, and computes the 16-bit checksum of the entire result. TCP does not count the pseudo header or padding in the segment length, nor does it transmit them. Also, it assumes the checksum field itself is zero for purposes of the the checksum computation. As with other checksums, TCP uses 16-bit arithmetic and takes the one's complement of the one's complement sum. As the receiving site, TCP software performs the same computation to verify the segment arrived intact."

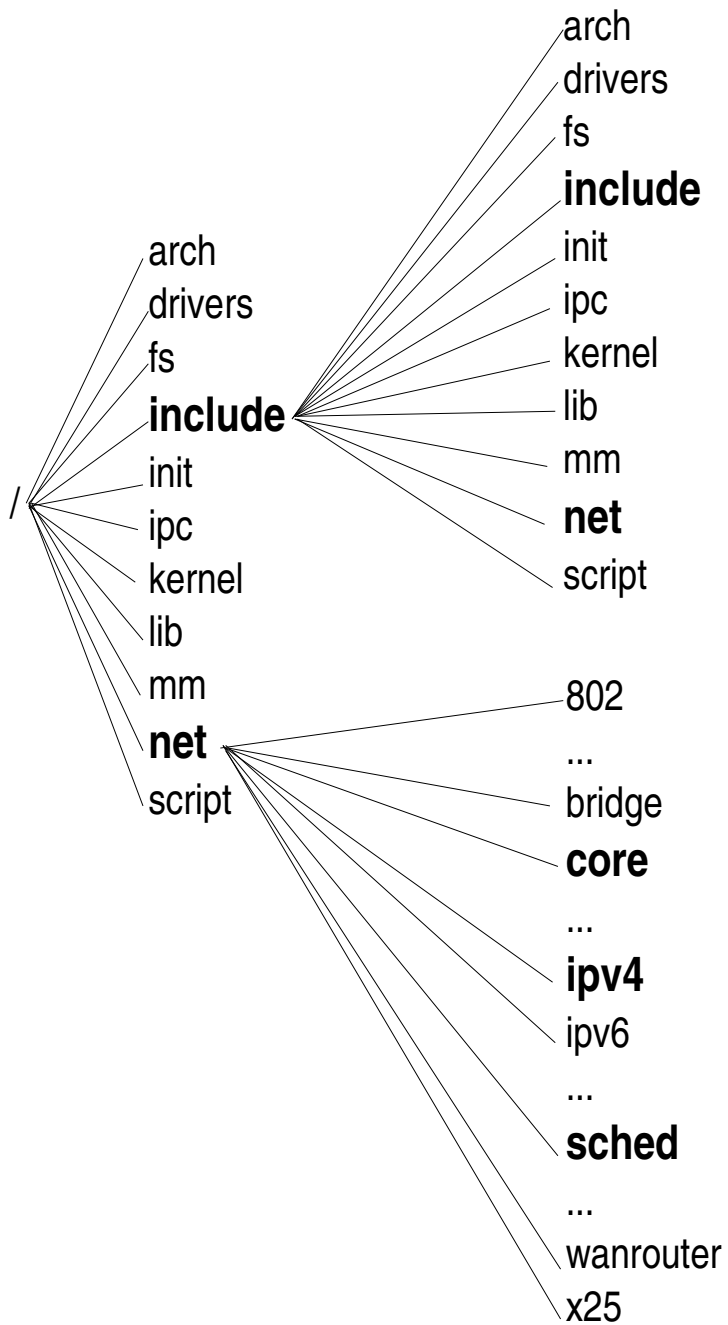
The checksum ensures the TCP header information is correct. The IP header also has a checksum. I remember in Linear Algebra there are mathematical ways to prove if one value is off it messes up the checksum and impossible to accidentally get the same checksums with only changing a few of the binary bit values. Now we are going to talk about the last field.



PICTURE OF THE SK_BUFF TO THE CORRECT SOCKET TO THE APPLICATION LAYER

At the beginning of the TCP header are the port numbers (in the sock struct, the source port is called “num”). The port numbers are used to bind a socket to a specific port number. If you have been in the computer networking field, BIND may sound similar. Bind is where a specific port number (one in the TCP header) is attached to a socket structure. The socket structure is the topic of the next section. But for now, consider the socket as the interface between the application and the kernel space. The port numbers route the packets from the sk_buff (kernel space) to the correct application. If a packet web request comes in for Apache web server it would have a port number 80. Again the port numbers help direct the packet to the correct application BIND to the correct socket which we will talk about next.

Now that you know more than you ever wanted to know about the TCP header and how it works. Below is a directory structure of where the IP version 4 and TCP are kept. We are going to go through this mess of code and I hope to convey to you the things I learned about the code, so all this learning will not have to be repeated.



DIRECTORY OVERVIEW OF WHERE ALL THE NETWORKING CODE IS KEPT