7. TCP and UDP

In this section, I want to review TCP and UDP, the common Layer 4 protocols used in the Internet. In particular, many of the other protocols used in the Internet suite of protocols use TCP or UDP as their foundation. You MUST take a look at Figure 31.1 on P. 576 of [Comer00] which clearly shows that over a dozen other protocols that we will later discuss specifically sit on top of TCP and/or UDP.

In addition, I want to make very clear the difference between a Layer 4 port, connection, and socket (which are three very different things).

Readings: Chapters 13, 12, and 11 of [Comer00], plus Figure 31.1 on P. 576.

Table of Contents

7.1 TCP

TCP is the one of the oldest and most common transport layer protocols.   Its job is to provide reliable data transfer.

- This requires checksums on the payload, because as we have seen, IP does not provide this. (Ethernet provides this, but you can't assume that you are using Ethernet; you may be using SLIP or some other unreliable Layer 2 protocol).
- Recall that IP does not provide true sequencing. Packets could arrive out of order, or in fact packets could go missing. To provide these features requires a 'connection-oriented' protocol.
- Finally, in order not to put too much flow control responsibility on the network (which is hard to do in a connectionless network using IP), TCP provides end-to-end flow control using credit allocation.

These requirements dictate the use of a connection-oriented protocol.

TCP is normally associated with the "socket" concept and Unix operating system. But, more and more we are finding these mixed with other systems. e.g. Sockets are showing up as a new MS-Windows API, and TCP/IP is used on systems other systems than Unix (after all, internetworking amongst only Unix computers is rather restrictive).

TCP is a stream-oriented protocol. Once a connection is made, you send bytes, not packets. If you send 1000 bytes, they may arrive in two groups of 500, or 10 groups of 100, etc.   Or two 1000 byte packets may arrive as one 2000 byte

group. i.e. There is no transport layer "framing" in the stream-oriented sub-class of a connection-oriented transport protocols. It just models a byte pipe stream. Bytes put in one end are guaranteed to get through, or you will be notified.

This instructor hates this particular aspect of TCP, as it means that EVERY application or higher-level protocol that uses TCP must add its own Layer 5 framing and transparency mechanism! This is stupid; this mechanism should have been available at least as an option at Layer 4 so that you each wouldn't have to do your own framing! Some RFCs to fix this have been proposed (see the interesting discussion in Section 24.7 of [Stevens94]).

Since TCP is stream-oriented, its (end-to-end) flow and error control do not work on the concept of packets, but instead each byte is given its own sequence number. These are measured modulo 2^{32}. Similarly, ACK numbers are modulo 2^{32}, and are use the 'next byte expected' convention. These acknowledge numbers are of the cumulative type, and take care of lost or duplicated Transport Protocol Data Units (TPDUs).   For efficiency in noisy environments, an option has been recently added to selectively NACK a range of byte numbers, rather than having to otherwise use Go-Back-N.

TCP also has an optional frame check sequence, which is usually enabled (but you should check your implementation).

Though TCP is stream oriented, it still has the concept of TPDUs. When a reasonable amount of data has been given to TCP by the application to transmit, TCP will send out a packet with the appropriate flow control, error control, and other information. On the other hand, the telnet application 'pushes' each individual byte out as soon as possible. In any case, in order to work, TCP has a specific TPDU format and to define its time semantics, a fairly complex state machine.

7.1.1 The TCP Header

To further discuss TCP it is best to show you a TCP header. One of the unfortunate things about the TCP frame format is there is not a header field to indicate the type of data that the TCP frame is carrying. Is it carrying user data, or does its payload contain yet a higher level protocol? Or does it carry some network layer packets being tunneled through this particular net? In Unix and now many systems, this is sometimes differentiated by well-known machine port number; for instance if the TCP is carrying File Transfer Protocol (FTP), it is normally sent to port 21. Note that a port is an IP address concept. All local processes on a CPU using a particular IP address (i.e. network interface card - NIC) share the same port address space.

TCP TPDU FORMAT:

Source Port (16)			Destination Port (16)
Sequence Number (32)			
Acknowledgment Number (32)			
HLen (4)	Reserved (6)	Code (6)	Window (16)
Checksum (16)			Urgent Pointer (16)
Options (if any)			Padding
Payload (maximum 65535-20-20 bytes)			

The first long word of a TCP TPDU header is the source and destination port numbers (source port numbers are included so you can know which one to reply to; they also are essential to uniquely identifying 'connections').

The next long word is the 32 bit *send* sequence number. It indicates the *byte* number of the first byte of payload measured modulo-4G from the beginning of the connection. Since TCP has no sense of Layer 4 user packets which need to be Acknowledged or Negatively Acknowledged, acknowledgments are by specified to the individual byte!

The third long word is the byte number (also modulo-4G) the sender is expecting next on that connection (i.e. you are sending a **piggy-backed** **cumulative ACK**).

Next comes a complicated long word, the first field of which is a 4 bit measure of the header length (measured in units of long words). This tells the destination how many option fields there are, and more importantly, where the transport payload actually begins.

Following 6 unused bits, are the 6 bit flags of the code field:

URG signals that the payload carries urgent data which should be handled by the receive process with higher priority than other data. This urgent data begins with the first byte of the payload and contains as many bytes as specified by the urgent pointer field. The urgent pointer can be thought of pointing to the end of the urgent data (normal data could follow the urgent data in the payload). Note: when you call the transport layer from a program and tell it to send urgent data, this causes an immediate push of the urgent data ('push' will discussed in a moment).

ACK signals that the number in the acknowledge field is valid (i.e. this TPDU is actually acknowledging something). If this flag is false, the 32 bit acknowledge field is still

present (as the first part of a TCP header is of fixed format), but contains junk.

PSH signals that this data was pushed by the source application (i.e. it was sent immediately rather than buffered up waiting to see if more source user data could be accumulated before bothering to send a TPDU). This is frequently done in Telnet, as you want what the user typed send within a second (especially if you are using remote echo), and not have the source wait to see if the user will type anything more. On reception, the push flag true indicates that the data should not be buffered upon reception. Blocked calls to recv() should be unblocked right away, even if for just a byte or two that has arrived in a short payload.

RST indicates a reset request. A port that receives a TCP packet for which it does not have a connection replies with a RST. It is also used to reject a request for connection.

SYN is used to indicate the TPDU is a connection request or connection accepted TPDU. ACK distinguishes between these two cases. If ACK=0, then the send sequence number is an randomly chosen number to eliminate any problems with lost or duplicated control packets. If ACK=1, the TPDU is the second of a 3 way connect handshake. The acknowledgment sequence number would be one higher (next number expected) than the random one received. I believe even the first TPDU of a 3 way connection handshake can carry data, though such data probably should not be passed to higher layers until the connection is fully confirmed.

FINnish is a connection release, and indicates the sender has no more data to send.

End-to-end flow control is handled using a credit allocation technique. The window size indicates how many bytes starting with the next number expected the sender is willing to receive. There is a bit of a problem here as the 16 bit credit only allows 64K to be sent max. This is ridiculously small on high speed, long delay (e.g. satellite) channels. RFC 1323 specifies a option negotiation technique whereby this window size can be considered to be scaled up by up to 16 powers of 2. This allows a credit for 4 GB to be sent!

There is a 16 bit frame check. If all zeroes, it indicates it is not in use. It should definitely be used on unreliable layer 2 links like SLIP. It is a 1's compliment checksum calculated over the payload, the TCP header, and a weird pseudo header (composed of the source and destination IP addresses, plus the protocol field from the IP header, plus the length of the TPDU). The length of the TPDU is not even an actual field in the IP header (its length field specifies the entire length of the IP packet including the IP header). This weird mixing of stuff from layer 3 into layer 4 is really poor. In fact, when transmitting, layer 4 has to reach down into layer 3 to find this stuff out to calculate the layer 4 checksum! If IP's header were not checked I could understand this; a packet might have got to the wrong destination. By everything I read says IP's payloads are not checked, but their headers are definitely checked, so why indirectly check them again via this pseudo-header concept?

The options field allows various things like window credit scaling, and *Selective* NACKs to be specified. The later

would specifying the range of bytes (first and last, or first and length) that have not been received properly.

7.1.2 Note Re Framing

Receiving half a packet is not possible just because the other half hasn't arrived yet. Nor because of IP fragmentation. The only two reasons you might get half of a send()'s data is:

- You called recv() and asked for too few bytes.
- The TCP send process thought it best to send the first half in a separate packet. This would be wise on error prone radio data channels.

Interestingly, there are also only two reasons you might get more than one TPDU in a call to recv():

- Several TPDUs had completely arrived, and you asked recv() for a large number of bytes.
- The source TPDU process had gotten a second call to send() before the data from an earlier send() had been transmitted. So it packaged both hunks together into one TPDU to transmit.

7.1.3 The TCP State Machine

The TCP frame format is not all that is required to specify the protocol. In addition, you need a call interface that can be used by the client programmer (i.e. sockets). And you need a finite state machine to carefully specify the actions/reactions appropriate in each mode. Figure 13.13 of [Comer95] shows the TCP state machine.

Figure 13.15 of [Comer00]

7.2 UDP - User Datagram Protocol

The connectionless transport layer protocol widely used on the Internet is UDP. UDP is for single send and receive messages, where the overhead of setting up a connection would not be efficient.

A good example is that of a remote procedure call (RPC). This is a simple send of the function designator and parameters, and the return of the return parameters. Since this procedure might not be called again for days, why set up and tear down a connection. UDP is also used for broadcasting routing tables to other nearby routers by RIP and OSPF, where a connection is just not required.

Note that UDP is an unreliable protocol because neither it, nor the IP it depends on, is connection-oriented. This is hardly serious for RIP or OSPF, as the routers will broadcast their known routes again in a minute or so anyway. But in the case of RPC, the calling process must wait for the return. If no return is forthcoming, it must be programmed to try the call again. But what if the call increments some remote variable, and it was the first return, not first call that was lost. In that case, the second (repeat) call has the potential to increment that variable, unfortunately, a second time!!!!

7.2.1 The UDP Frame Format

The UDP header is simply four 16 bit words: source port, destination port, TPDU total length (including the header), and the weird UDP checksum.

Source Port	Destination Port
Message Length	Checksum
Payload	

Notice that UDP has the same limitation that TCP has in that it has no header field describing the nature/protocol of the payload. Again, like TCP, the destination port number sort of by convention implies this.

It seems to me that the length field is not needed. There isn't one in TCP; the length from the underlying network (IP) layer can be passed up. After subtracting the IP header length and UDP header length, you know how much UDP payload there will be. This, and the crazy, layer-spanning checksum mechanism of TCP and UDP, is an example of how the Internet was thrown together without a lot of planning, and with free labor (mainly grad students being supervised by university professors in the U.S. being supported by U.S. Government research money). Its main redeeming quality is that it is cheap, and it has been heavily tested and adapted over the years.

UDP uses the same kind of optional checksum that TCP uses, checking the UDP header, UDP payload, and weird IP pseudo-header. If the UDP checksum appears to be zero, it is not being used, because this is not a valid checksum.

7.3 Sockets vs. Connections vs. Ports

Students and even professionals get confused between:

- ports - of which there are two of each number per network interface/IP address (one for UDP and one for TCP).
- sockets - which are what individual processes use to identify an incoming or outgoing data stream (I won't call it a connection so as to be able to cover connectionless streams as well).
- connections - which represent a 3 tuple on the client end and a 5 tuple on the server end.

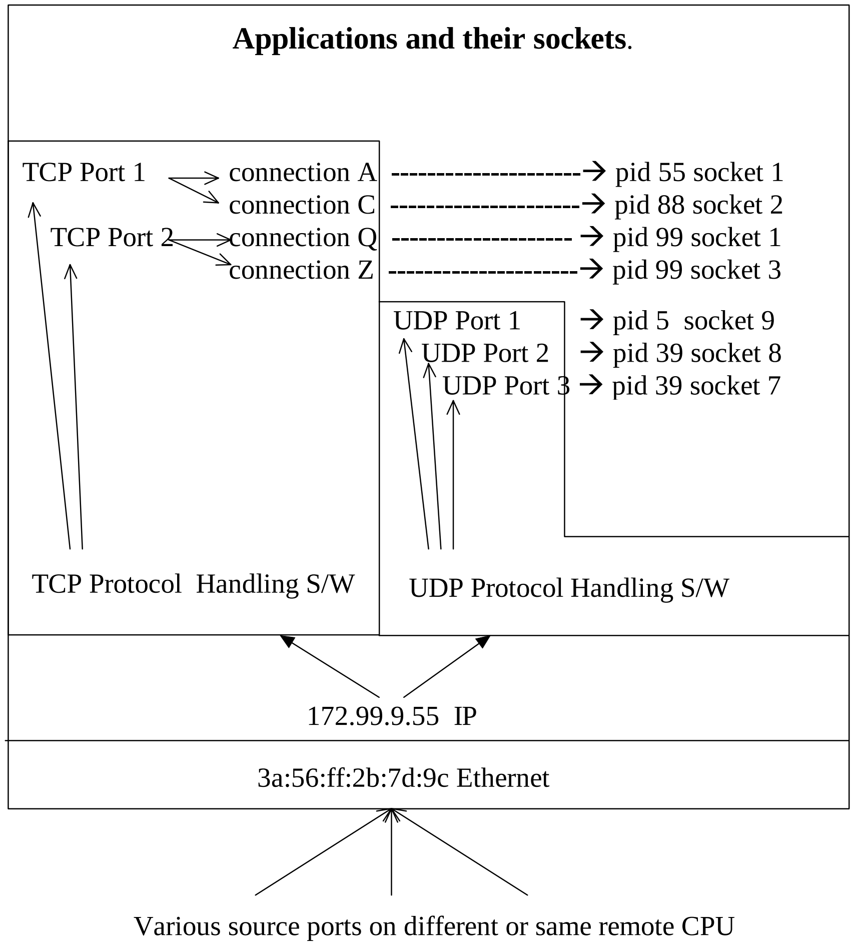
Every TCP/UDP machine uses short integers to identify particular ports. Note that port 99 for UDP is a different port than port 99 for TCP. Which one of these a packet is destined for is differentiated by the protocol field of the IP packet. Many people do not realize this, and thus often when certain ports are reserved for certain universal services, the service reserves both the UDP and TCP versions of that port number. This also allows for future offering of the same service with the opposite connectivity (i.e. connectionless vs. connection-oriented).

The first thousand or so ports are reserved for particular services. These can be seen either in RFC 1700, or on FreeBSD Unix, in file /etc/services. A sample from the latter is shown below. Some ports that are lesser well know but might cause conflicts if you use them are mentioned in RFC 1700 ranging up to about port 7000.

```
# WELL KNOWN PORT NUMBERS
#
rtmp          1/ddp        #Routing Table Maintenance Protocol
tcpmux              1/tcp        #TCP Port Service Multiplexer
tcpmux              1/udp        #TCP Port Service Multiplexer
nbp           2/ddp        #Name Binding Protocol
compressnet   2/tcp        #Management Utility
compressnet   2/udp        #Management Utility
compressnet   3/tcp        #Compression Process
compressnet   3/udp        #Compression Process
echo          4/ddp        #AppleTalk Echo Protocol
rje           5/tcp        #Remote Job Entry
rje           5/udp        #Remote Job Entry
zip           6/ddp        #Zone Information Protocol
echo          7/tcp
echo          7/udp
discard             9/tcp        sink null
discard             9/udp        sink null
systat             11/tcp        users    #Active Users
systat             11/udp        users    #Active Users
daytime            13/tcp
daytime            13/udp
qotd         17/tcp        quote    #Quote of the Day
qotd         17/udp        quote    #Quote of the Day
msp          18/tcp        #Message Send Protocol
msp          18/udp        #Message Send Protocol
chargen            19/tcp        ttytst source  #Character Generator
chargen            19/udp        ttytst source  #Character Generator
ftp-data     20/tcp        #File Transfer [Default Data]
ftp-data     20/udp        #File Transfer [Default Data]
ftp          21/tcp        #File Transfer [Control]
ftp          21/udp        #File Transfer [Control]
ssh          22/tcp    #Secure Shell Login
ssh          22/udp    #Secure Shell Login
telnet             23/tcp
telnet             23/udp
#            24/tcp        any private mail system
#            24/udp        any private mail system
smtp         25/tcp        mail           #Simple Mail Transfer
smtp         25/udp        mail           #Simple Mail Transfer
nsw-fe             27/tcp        #NSW User System FE
nsw-fe             27/udp        #NSW User System FE
msg-icp            29/tcp        #MSG ICP
msg-icp            29/udp        #MSG ICP
msg-auth     31/tcp        #MSG Authentication
msg-auth     31/udp        #MSG Authentication
dsp          33/tcp        #Display Support Protocol
dsp          33/udp        #Display Support Protocol
#            35/tcp        any private printer server
#            35/udp        any private printer server
time         37/tcp        timserver
time         37/udp        timserver
rap          38/tcp        #Route Access Protocol
rap          38/udp        #Route Access Protocol
rlp          39/tcp        resource #Resource Location Protocol
rlp          39/udp        resource #Resource Location Protocol
graphics     41/tcp
graphics     41/udp
nameserver   42/tcp        name           #Host Name Server
nameserver   42/udp        name           #Host Name Server
```

7.3.1 UDP Sockets

A given UDP port can accept incoming traffic from several different source computers (or different source ports or sockets on the same computer). The following diagram illustrates this flow of inbound traffic



Notice in particular how traffic bound for UDP port 1 ends up at a different socket than traffic bound to TCP port 1.

For UDP communications, a socket is a 5 tuple:

1. An I/O descriptor.
2. A process ID (pid) of the process owning the socket.
3. A local IP address.
4. A protocol designator.
5. And a local port number.

If you have ever done socket programming, you probably have had to do a bind() call (or one was made by default for you). This **bind is what forms the association between descriptor/pid and address/protocol/port**.

You can think of this association as a database table in RAM with primary and alternate compound keys.

Descriptor	PID	IP address	Protocol	Port

Given the inbound IP address, protocol, and port (i.e. the primary key), you can determine the pid and I/O descriptor of the socket which that incoming data should be sent to.

Similarly, for output, given the program's PID and socket I/O descriptor that the write applies to (i.e. the alternate key), you can determine the source port, (source) protocol, and source IP address that that outgoing datagram should leave via, and be labeled with. Therefore, a UDP socket is thought of as a 5-tuple: port, protocol, IP address, processID (PID), and I/O descriptor index.

7.3.2 TCP Client Sockets

For TCP, things are a little more complicated. At the client end (the end initiating the connection), a socket is a 7 tuple:

1. I/O descriptor of socket
2. local process id of socket
3. local port
4. local IP address
5. protocol
6. remote IP address
7. remote port

Each 7 tuple is associated with a compound key composed of the local pid and I/O descriptor. Given the I/O descriptor and pid, you can find out the other 5 things.

I/O Descr	PID	Local Addr	Local Port	Protocol	Remote Addr	Remote Port

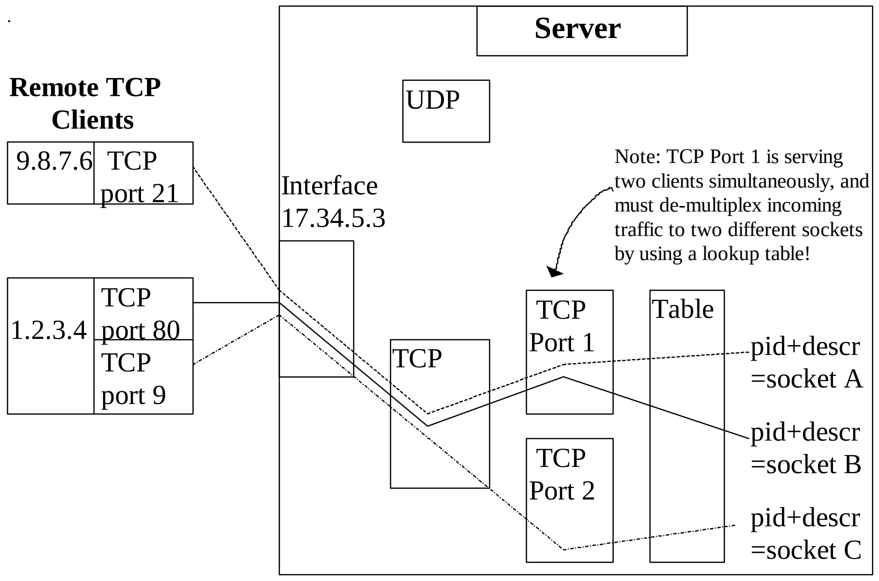
In addition, there is an alternate compound key used during reception. If receiving a reply, given the local port, local IP address, and local protocol the packet is addressed to, you can find out the local pid and socket the data is destined to.

Note that for TCP *clients*, a *particular* local TCP *port* can NOT be connected to more than one remote site. I.e. the unshaded columns above are not needed as part of the alternate key. This is because clients/callers do not have to be from a particular well-known port; the client can use any port number, and the server will reply to that particular client port that requested the connection. That is not to say that one computer or process cannot have two TCP client connections to a particular remote site, only that they cannot go out through the same socket or port. (This is not true of UDP; UDP can spray packets from one port to several different destinations on the Internet).

7.3.3 TCP Server Sockets

On the other hand, a TCP *server* can have many connections through one of its ports. This is particularly true when you handle multiple simultaneous connections. They are distinguished by the additional attributes: remote IP address and remote port. Therefore a suitable compound key which can be used to determine which socket (i.e. pid and descriptor) an inbound packet should be given to is the right hand 5 columns in the able above. (Note this is 2 more additional columns than are needed by UDP or client TCP).

For outgoing packets, I/0 descriptor and pid are still an adequate look up key for the server to find out the other 5 fields which need to be inserted in the outgoing packet.



7.4 References

[Comer00] “Internetworking with TCP/IP, Vol. 1, 4th ed.” by Douglas E. Comer, Prentice-Hall, 2000.

[Stevens94] “TCP/IP Illustrated, Vol. 1” by W. Richard Stevens, Addison-Wesley, 1994.