Computer Networks
UDP and TCP

Saad Mneimneh
Computer Science
Hunter College of CUNY
New York

"I'm a system programmer specializing in TCP/IP communication protocol on UNIX systems. How can I explain a thing like that to a seven-year-old?"

1 Introduction

So far, we have studied the DLC layer. The next layer up is the network layer, upon which the transport layer operates.

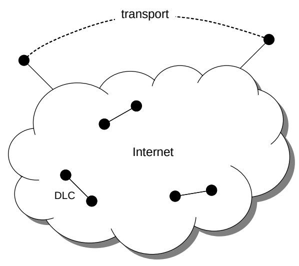

Figure 1: DLC, network, transport

While it may be natural to discuss the network layer next, DLC and the transport layer share many features in terms of reliable communication (but one is on a single link and the other is over a network). Many of the algorithms and protocols that we have seen for DLC apply for the transport layer. For this reason, we will consider the transport layer next.

From the viewpoint of the transport layer, the underlying network has certain limitations in the level of service it can provide. Some of the more typical limitations are:

- loss, reordering, and duplicate delivery of packets
- limited packet size (MTU, Maximum Transmission Unit)
- arbitrary long delays
- host to host communication

Such a network is said to provide *best-effort* level of service, as exemplified by the Internet. Therefore, it is the role of the transport layer to turn the unreliable host based communication of the underlying network into a reliable communication among application programs. The following list shows some of the common properties that a transport protocol can be expected to provide:

- reliable message delivery: once, only once, and in order
- support for arbitrarily large messages
- synchronization between sender and receiver through flow control
- support for multiple application processes on each host

Two transport protocols are particularly famous for the Internet: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

2 User Datagram Protocol (UDP)

UDP is a simple transport protocol that extends the host-to-host delivery of packets of the underlying network into a process-to-process communication. Since there are many processes running on a given host (e.g. multiple Internet browsers), UDP needs to add a level of demultiplexing, allowing multiple application processes on each host to share the network. Therefore, the only interesting issue in UDP is the form of address used to identify a process. Although it is possible to *directly* identify a process with the operating system (OS) assigned id (pid), such an approach is only practical in a close distributed system with one OS that assigns unique ids to all processes (does not work for the entire world!). Instead, a process is *indirectly* identified using an abstract locator, often called a *port*. A source process sends a message to a port, and a destination process receives a message from a port. The UDP header contains a 16 bit port number for both the sender (source) and the receiver (destination).

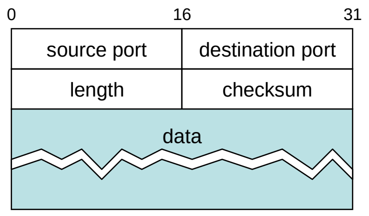

Figure 2: UDP header

- source port: 16 bit port number of the source
- destination port: 16 bit port number of the destination
- length: 16 bit number representing the length in bytes of the UDP datagram (including the header)
- checksum: 16 bit checksum used for error detection (later)
- data: the message

The 16 bit port number means that there are up to 64K possible ports, clearly not enough to identify all the processes on all hosts in the Internet. For this reason, ports are not interpreted across the entire Internet, but only within a single host. Therefore, a process is really identified by a port on some particular host, i.e. a $< port, host >$ pair. This pair constitutes the demultiplexing key for the UDP protocol.

The port number of a process is automatically assigned by the operating system through the socket API, i.e. the socket API provides an implementation of ports. But how does a process learn the port number for the process to which it wants to send a message? Typically, a client process initiates a message exchange with a server process (recall from the socket API that a server listens and a client connects). Once the client has contacted the server, the server knows the client's port number from the UDP header and can reply to it. The real problem, therefore, is how the client learns the server's port number in the first place. The common approach is for the server to listen on a well-known port that is widely published, much like the emergency service available at the well-known telephone number 911. For example, an http server listens on port 80. The telnet port is 23. The Domain Name Service (DNS) port is 53. The mail port is 25.

In the socket API, each port is typically implemented as a message queue.

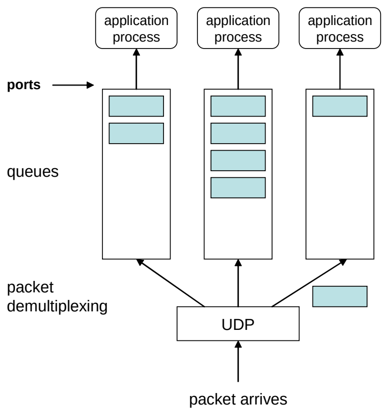

Figure 3: Ports as message queues

When a message arrives, UDP appends it to the end of the queue. If the queue is full, the message is dropped (unlike TCP, there is no flow control to instruct the sender to slow down). When the application process wants to receive a message, one is removed from the head of the queue. If the queue is empty, the application process blocks until a message becomes available.

Another equally important question is how does UDP obtain the host address to form the $< port, host >$ demultiplexing key? This information is available in the IP header of the network layer. Recall, however, that the basic tenets of layering is that the header (and trailer if any) of a packet at a given layer contains all the information necessary for that layer to do its job.

- the header of a given layer is stripped off when the packet is passed to an upper layer
- the body of the packet at a given layer (which contains the headers for the higher layers) should not be consulted by the module at the given layer

Here's a figure from before as a reminder:

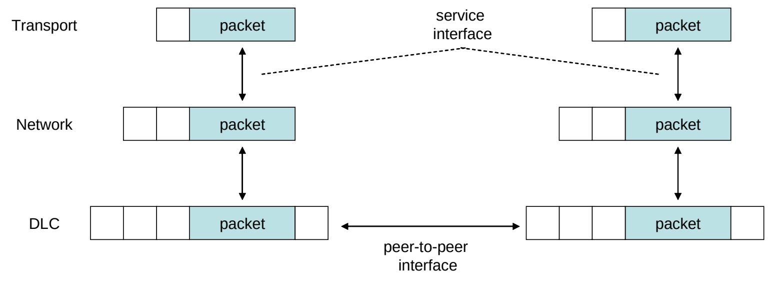

Figure 4: Headers and trailers

Nevertheless, when a packet is passed from the transport layer to the network layer, there are extra parameters passed along with the packet to provide the network layer with addressing information. Similarly, when the datagram arrives at its destination, the IP header is stripped off, but the addressing information and other header information can be passed up as parameters. That's how UDP obtains such information. In fact, three fields from the IP header - protocol number, source IP address, destination IP address - plus the UDP length field, form what is known as the *pseudoheader*.

Finally, although UDP does not implement flow control or reliable/ordered delivery of messages, it does a little more work than to simply demultiplex messages to some application process. As a last line of defense, it ensures the correctness of the message by the use of a checksum. The checksum is computed over the UDP header, the data, and the pseudoheader (so the UDP length field is included twice in the checksum calculation). The checksum algorithm is the following: think of the data to be checksummed as a sequence of 16 bit integers, add them (that's why it's a checksum) using ones complement arithmetic, and compute the ones complement of the result. The pseudoheader is included in the checksum to verify that the message has been delivered between the correct two endpoints. For example, if the destination IP address was modified while the packet was in transit, causing the packet to be misdelivered, this fact would be detected by the UDP checksum. The reason why the checksum is obtained using ones complement arithmetic will be explored in a homework. But this has mainly two advantages: (1) easy to update if the packet undergoes small changes, and (2) independent of how the machine represents bytes in memory.

In ones complement arithmetic, a negative number $-x$ is represented as the complement of x, i.e. each bit of x is inverted. When adding two numbers in ones complement arithmetic, a carryout from the most significant bit needs to be added to the result. Consider, for example, the addition of -5 and -2 in ones complement arithmetic using 4 bits. $+5$ is 0101, so -5 is 1010; $+2$ is 0010, so -2 is 1101. If we add 1010 and 1101 ignoring the carry, we get 0111. In ones complement arithmetic, the fact that this operation caused a carry from the most significant bit causes us to increment the result by 1, giving 1000, which is the ones complement representation of -7, as we would expect. The following routine gives a straightforward implementation of the checksum algorithm:

```
checksum(short * buf, int count) {
  int sum = 0;
  while (count--) {
    sum += *buf++;
    if (sum & 0xFFFF0000) {
      sum &= 0xFFFF;
      sum ++;
    }
  }
  return ~(sum & 0xFFFF);
}
```

Figure 5: The Internet checksum (also used in IP at network layer)

3 Transmission Control Protocol (TCP)

In contrast to a simple demultiplexing protocol like UDP, a more sophisticated transport protocol in one that offers reliable communication. The Internet's Transmission Control Protocol (TCP) is probably the most widely used protocol of this type. As a transport protocol, TCP provides reliable, in order delivery of messages. It is a full duplex protocol, meaning that each TCP connection supports a pair of streams, one flowing in each direction. It also includes a flow control mechanism for each of these streams that allows the receiver (on both ends) to limit how much data the sender can transmit at a given time (we will look at this feature later). Of course, TCP supports the demultiplexing mechanism of UDP to allow multiple application programs on a given host to simultaneously communicate over the Internet. However, the demultiplexing key used by TCP is the 4-tuple $<$ *source port*, *source host*, *destination port*, *destination host* $>$ to identify the particular TCP connection.

At the heart of TCP is the sliding window algorithm. Even though this is the same basic algorithm we have seen before for DLC, because TCP runs over the network rather than a single link, there are many important differences that complicate TCP.

3.1 Added complications of TCP

Here's a list of the most relevant complications added by TCP to the sliding window algorithm.

3.1.1 Out of order packets

One important aspect of the sliding window algorithm is the choice of a modulus to impose an upper bound on the sequence numbers. In our treatment of the DLC, we used the FIFO property of the link to obtain an appropriate modulus. Unlike the case of a single link, however, packets may be reordered as they cross the Internet. Therefore, TCP has to be prepared to deal with an old packet that suddenly shows up at the receiver with both SN and RN being in the appropriate windows, hence potentially confusing the sliding window algorithm.

3.1.2 Initialization and disconnection

Recall that for the DLC, whether the link is up or down is maintained by an alternating sequence of INIT and DISC messages (each side of the DLC remembers the last ack, whether ACKI or ACKD). A similar mechanism is required for TCP to initialize the sliding window algorithm. Unlike the link initialization of DLC, however, what makes the initialization and disconnection of TCP sessions tricky is that one does not want to save information about connections after they have been torn down (may connections may exist). In other words, while DLC initialization runs indefinitely, after the last ack is sent by TCP, no more data is sent! Therefore, TCP must use an additional mechanism to avoid the confusion between old and new connections.

3.1.3 Window size

Parameters of a link such as RTT and bandwidth are fixed and known. Therefore, a window size can be computed (recall that a window size of $\approx 1 + \frac{RTT \times bandwidth}{frame\ size}$ frames is appropriate). TCP has no idea what links will be traversed by the packets. Moreover, RTT may change depending on different times of the day even if the same set of links are used. This information is also relevant to determine appropriate time-outs. Finally, the sender and the receiver communicating over TCP might have different speeds; therefore, the receiver must be able to limit how much data the sender can transmit (window size). This is done through flow control, which we will study later.

3.1.4 Retransmission time-outs and congestion

At the DLC layer, the issue of retransmission time-outs is not very critical (besides the correctness of the distributed algorithms). At the transport layer, the issue is quite serious. Retransmissions from the transport layer causes congestion in the network (a link is shared by many TCP connections). This will cause large delays. When congestion becomes bad enough, packets will be discarded in the network. This triggers even more retransmissions, and eventually throughput is reduced to almost zero. TCP must deal with this issue, and we will study TCP congestion control later.

3.2 TCP speaks bytes

TCP is a byte oriented protocol. The sender writes bytes into a TCP connection, and the receiver reads bytes out of the TCP connection. TCP does not, itself, transmit individual bytes. That would be too much overhead (why?). Instead, the sender generates bytes into the TCP buffer (sliding window), and TCP collects enough bytes from this buffer to fill a reasonably sized packet called segment (one that is appropriate for the underlying network to prevent further segmentation at the network layer).

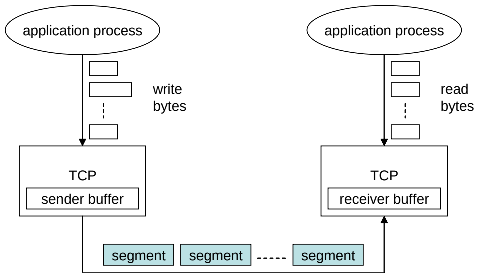

Figure 6: TCP segments

The segment contains a sequence number SN and a request number RN as usual. However, those numbers are used in a rather peculiar way in TCP. SN represents the sequence number of the first byte of data in the segment. Thus if a segment has a sequence number $SN = m$ and contains n bytes of data, then the next segment for that TCP session has $SN = m + n$. There appears to be no good reason for this, but it does not do any harm, other than adding complexity and overhead. TCP at the receiving side empties the content of the segment into the TCP buffer (sliding window), and the receiver read bytes from this buffer. As one might expect, RN denotes the next desired segment by giving the number of the first byte of data of that segment, usually piggybacked on segments going in the opposite direction. The following figure shows the segment format for TCP.

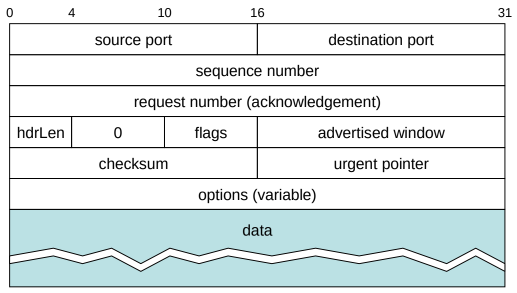

Figure 7: TCP segment format

- source port: 16 bit port number of the source
- destination port: 16 bit port number of the destination
- sequence number: 32 bit SN
- request number: 32 bit RN
- HdrLen: length of header in 32 bit words, needed because of options, also known as offset field
- flags: 6 bits for SYN, FIN, RESET, PUSH, URG, and ACK
    - SYN and FIN are used to establish and terminate a TCP connection (see next section)

  - RESET is set by the receiver to abort the connection, e.g. when the receiver is confused upon the receipt of an unexpected segment
  - ACK is set by the receiver whenever an acknowledgment must be read
  - PUSH is set by the sender to instruct TCP to flush the sending buffer, also used by the receiver to break the TCP byte stream into records (not supported by socket API)
  - URG is set by the sender to signify that the segment contains urgent data

- advertised window: 16 bit number used for flow control (later)
- checksum: 16 bit checksum computed over the TCP header, the TCP data, and the pseudoheader (same algorithm as for UDP)
- urgent pointer: when URG flag is set, urgent pointer indicates where the urgent data ends (it starts at the first byte of data)
- option: variable
- data: the message

3.3 Connection establishment and termination

TCP uses a three-way handshake similar to that of the DLC link initialization to establish a connection and initialize the sliding window algorithm. During this three-way handshake, the SYN flag is used in an analogous way to INIT. However, the main difference is that TCP does not start with $SN = 0$ (and $RN = 0$). The reason for this, as motivated in Sections 3.1.1, is to protect two incarnations of the same connection from using the same sequence numbers too soon. For instance, assume that a connection is aborted by the receiver after exchanging few segments. Assume also that some of these segments are still in the network. If the connection is re-established (with $SN = 0$ and $RN = 0$), some of these segments will qualify as legitimate segments by the sliding window algorithm because their SNs and RNs are still within the expected range.

Therefore, TCP requires that each side select an initial SN at random (well, in practice SN is initialized by the value of an internal clock counter that increments once every 4 μsec). The sequence numbers are then exchanged while establishing the connection using a three-way handshake.

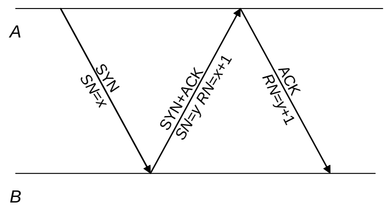

Figure 8: TCP three-way handshake

Here's a simplified state diagram for establishing a TCP connection in the form of event/action (ϵ means empty event or empty action):

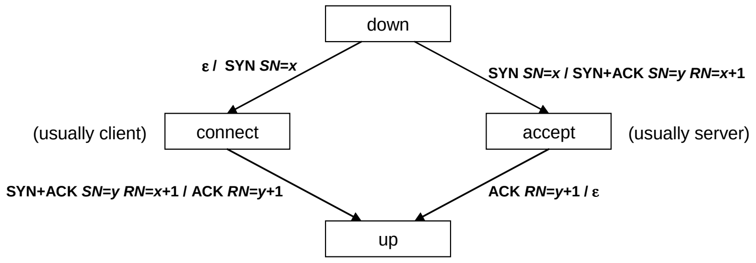

Figure 9: Establishing a TCP connection

From the above diagram, we can see that if the last ack was lost, the client thinks the connection is established but the server thinks it is not established yet. In this case, the client starts to send data, and the ack will eventually get to the server. But what if the client terminates the connection before the server knows it was established? In this case, the last ack with FIN (see below) will tell (acks are acted upon first).

To terminate a connection, another three-way handshake is done where the FIN flag is used in an analogous way to DISC. Here's a simplified equivalent (but wrong) state diagram for terminating a TCP connection in the form of event/action (ϵ means empty event or empty action).

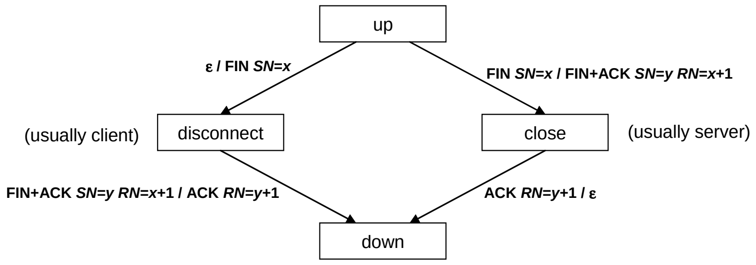

Figure 10: Terminating a TCP connection (wrong)

From the above diagram, we can see that if the last ack was lost, the client thinks the connection is terminated but the server thinks it is not terminated yet. As described in Section 3.1.2, the client is sending no more data! If another client with the same port and host establishes a connection with the server, the new connection will be confused with the old one. As a result, a delayed FIN+ACK from the server might cause the new connection to be terminated. While the use of 32 bit sequence numbers generated by a fast internal clock counter makes this possibility remote, TCP specification requires the client to wait before terminating the connection. A Maximum Segment Lifetime (MSL) is imposed on the segments, i.e. a segment cannot survive in the Internet for more than MSL, which is 120 seconds. The client waits for twice the MSL before assuming the connection is terminated.

Here's the updated simplified diagram for terminating a TCP connection in the form of event/action (ϵ means empty event or empty action).

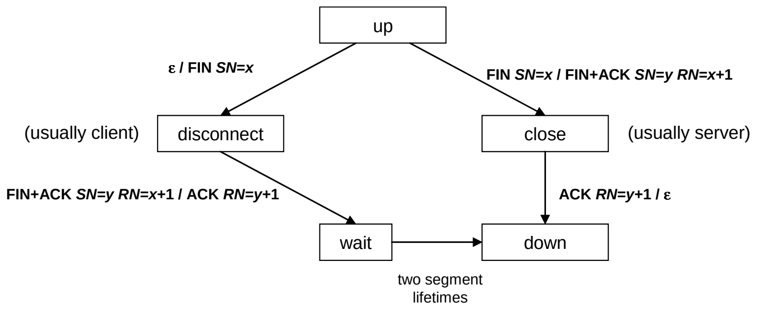

Figure 11: Terminating a TCP connection

3.4 Sequence numbers wrap around

The use of randomly initialized sequence numbers SN when establishing a connection, and the wait for twice the MSL when terminating a connection, make the possibility of error due to multiple TCP connections very small. However, even with one TCP connection, sequence numbers will eventually wrap around (SN is finite) if the connection remains active for a long time. Without the FIFO property, a delayed segment that suddenly shows up will cause the sliding window algorithm to make a mistake in accepting it. The way TCP fights against this possibility is through an appropriate choice of MSL (hence a segment cannot be delayed indefinitely). Currently, MSL is 120 seconds. Let's see how much time is needed for the sequence numbers to wrap around. This depends on the speed of the network. We have 2^{32} possible sequence numbers. On Ethernet networks, for instance, the bandwidth is 10 Mbps; therefore, we need $\frac{2^{32}\times 8}{10\times 10^6}$ seconds to transmit 2^{32} bytes (to wrap around). This is 57 min $>>$ MSL. On faster networks (e.g. 622 Mbps and 1.2 Gbps), the time to wrap around is much smaller than 120 seconds.

3.5 Effective window size

The window size W (in bits) must be large enough to use the full capacity of the network. Let MSS be the Maximum Segment Size. If the bandwidth is B, then the sender transmits W bits every $RTT + MSS/B \approx RTT$ (usually $RTT >> MSS/B$). Of course one of the difficulties in determining W is to obtain accurate measurements of RTT and B. But assuming these are known, the sender is limited to $\min(B, W/RTT)$ bps.

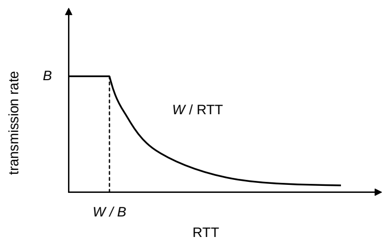

Figure 12: Window size

Therefore, we need $W \geq RTT \times B$. TCP can support windows up to $2^{16} - 1$ bytes ($\approx$ 64 KB), as we will see next.

References

Dimitri Bertsekas and Robert Gallager, Data Networks
Larry Peterson and Bruce Davie, Computer Networks: A Systems Approach