ECE 333: Introduction to Communication Networks
Fall 2002

Lecture 27: Transport layer II

In the last lecture we began discussing the transport layer in a communication network. In the Internet there are primarily two transport layer protocols, TCP and UDP. UDP simply provides a unreliable, connectionless multiplexing service. In addition to multiplexing, TCP provides a reliable, connection-oriented service, over the unreliable, connectionless network layer provided by IP. TCP also provides for ***flow control*** and ***congestion control***; these are mechanisms that regulate the rate at which packets enter the network. In this lecture we look at how TCP provides reliability and flow control. In the next lecture, we will discuss the congestion control algorithm used by TCP.

Reliable Service

TCP provides a reliable *byte stream* service - it transmits data as a stream of bytes and does not preserve message on data given to it from the application layer. Specifically, TCP arbitrarly breaks up the data (on byte boundaries) into a sequence of segments (packets).

To provide reliability in TCP, a version of the Selective Repeat ARQ protocol is used. We have previously considered ARQ protocols at the Data Link Layer; the protocol used by TCP behaves similarly. Each segment is numbered, and acknowledged by the receiver. If a segment is not acknowledged it is retransmitted.

However, with TCP several new complications arise because packets can be delayed and can arrive out of order. Thus it is more difficult to ensure that two packets with the same sequence number cannot be present in the network at the same time. This is addressed by using a larger set of sequence numbers. This is further complicated because host may be rebooted. For example, suppose two hosts establish a TCP connection and begin sending data. The hosts then crash, are rebooted and reestablish the TCP connection. How do the hosts make sure that an old packet from the previous connection is not confused with a new packet with the same sequence number? This is addressed through the connection establishment algorithm used in TCP.

Connection establishment

In TCP, a connection establishment phase is required, to ensure that the receiving process is available and to synchronize the sequence numbers used for the ARQ protocol (In TCP, connections are duplex, so a starting sequence number is needed for both hosts). Connection establishment is a basic problem that arises in many other places as well. We look at the generic problem of connection establishment first, and then discuss some details of how this is implemented in TCP.

Consider the following simple connection establishment protocol

1. The client transmits a special packet requesting a connection.
2. The server responds to a connection request with an "accept" packet.
3. After receiving the accept packet the client and server begin sending data using some ARQ protocol with initial sequence numbers of zero (for each direction).
4. If the client does not receive a reply to a connection request after a certain time, it times-out and retransmits the request.

This simple scenario is complicated by the fact that the network can lose, store and re-order packets.

Example: Suppose the client wants to send one data segment. First it requests a connection, then sends the data (with sequence number 0) and closes the connection. Suppose the client times out after doing each of these steps and then retransmits. This results in the following sequence.

Client Transmits:

CONNECT CONNECT DATA(0) DATA(O) CLOSE CLOSE

Since data can be reordered in the network, the server could receive:

CONNECT DATA(0) CLOSE CONNECT DATA(0) CLOSE

In this case the server may think that the client opened two sessions and sent two different data packets.

Similar problems can arise if a client crashes and then re-opens a connection, while a packet from a previous connection is still in the network.

We will consider some possible solutions to these problems next.

First solution: The server could keep track of the last sequence number used for a given host and assign the next sequence number to start the next connection (this could be sent to the server in the "connection accept" packet). In the above case, when the server received the second connect, it would tell the client to begin using sequence number 1. In this case when the server receives the second data packet 0 it would know that this was from a previous connection.

Problems: There are still several problems with the above solution:

(1) This requires the server to store in memory a sequence number for each connection even after the connection has been released. For large servers this is undesirable.

(2) Even if the server keeps track of sequence numbers, they will eventually "wrap around" and be reused. If a packet is delayed in the subnet long enough an old packet can still be confused with a new packet that has the same sequence number.

(3) Either the client or server could crash and loose track of the sequence numbers.

A connection establishment algorithm that avoids the above problems can be designed if the following two assumptions hold

(1) There is a maximum time that packets can be delayed in the subnet. (This type of guarantee must be provided by the network layer, e.g., with the use of the time-to-live field in IP packets.)

(2) Each host has a **clock** available, which it uses to decide on initial sequence numbers. This clock can be thought of as a counter with N bits, that increments every r seconds (N is the number of bits used for sequence numbers). We assume this clock keeps time even if the host crashes (to avoid the third problem above). These clocks do not need to be synchronized between two different hosts.

Given the above assumptions the following connection establishment procedure is used:

(1) The client picks an initial sequence number based on its clock and includes it in the connection request packet.

(2) The server responds to the connection request acknowledging the clients sequence number and including its own initial sequence number (based on its own clock).

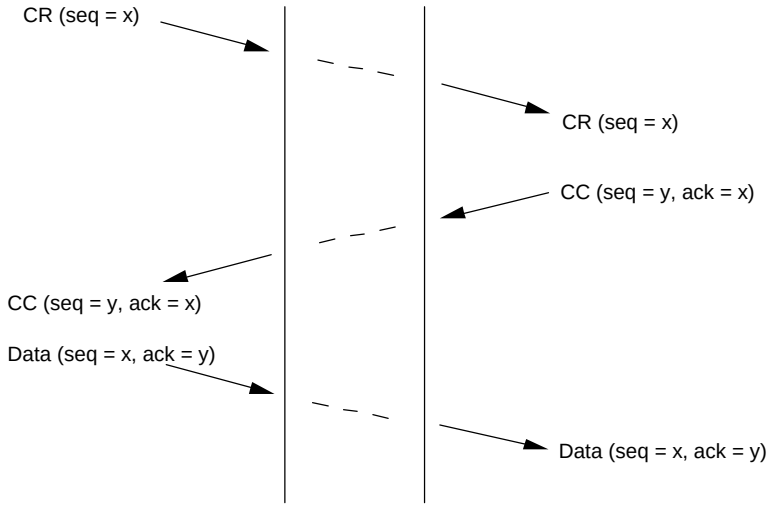
(3) The client then sends its first data packet, using its initial sequence number and acknowledges the servers initial sequence number.

Both sides agree that the connection is established only after all three of these steps have taken place. This type of exchange is referred to as a ***3 –way handshake.***

After a connection is established, the hosts increment their sequence numbers according to the rules of the ARQ protocol.

Three Way Handshake

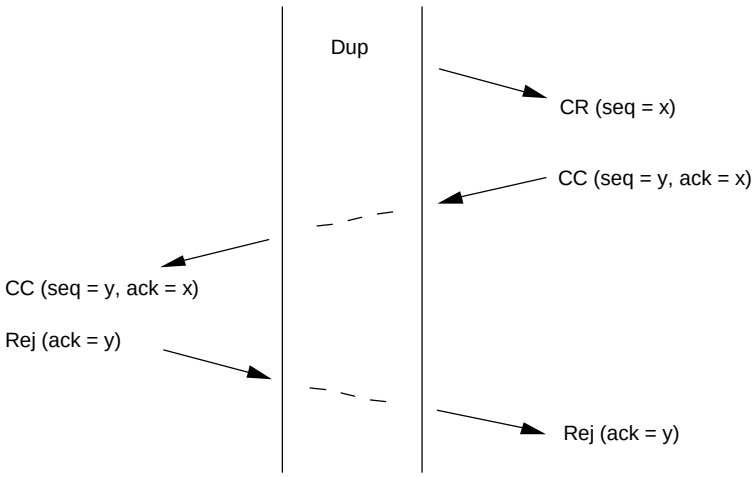
Schematically, it looks like this:



CR= connection request, CC = connection confirmed
In TCP, the CR and CC packets are called SYN messages - these are TCP segments that have a special SYN bit flag set in their header.

Avoiding Duplicate Connections

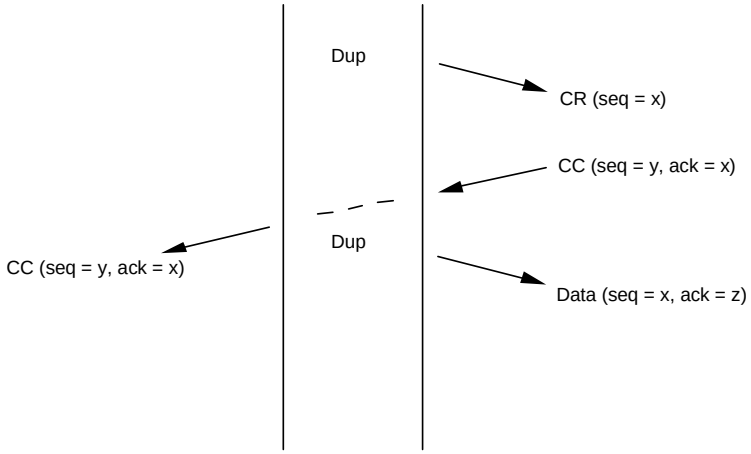
What happens if a duplicate CR shows up? The originator is informed as follows:



The client knows that it has already given up on starting a connection with seq=x. It may tell the server, so that the server does not hold on to false connection (or the server may simply time-out and close the connection.)

Delayed Request and Delayed Third Leg

As in the above example, suppose that both the connection request packet and the first data packet sent from the client where repeated and show up after the connection is closed.



In this case, how does the server know to not accept the data?

For the above protocol to work correctly, it needs to be guaranteed that two different packets in the network cannot have the same sequence numbers. For example, suppose that in the example on page 11, the sequence number in the server's response, $y=z$. Then the server would not be able to tell that the second packet was a duplicate.

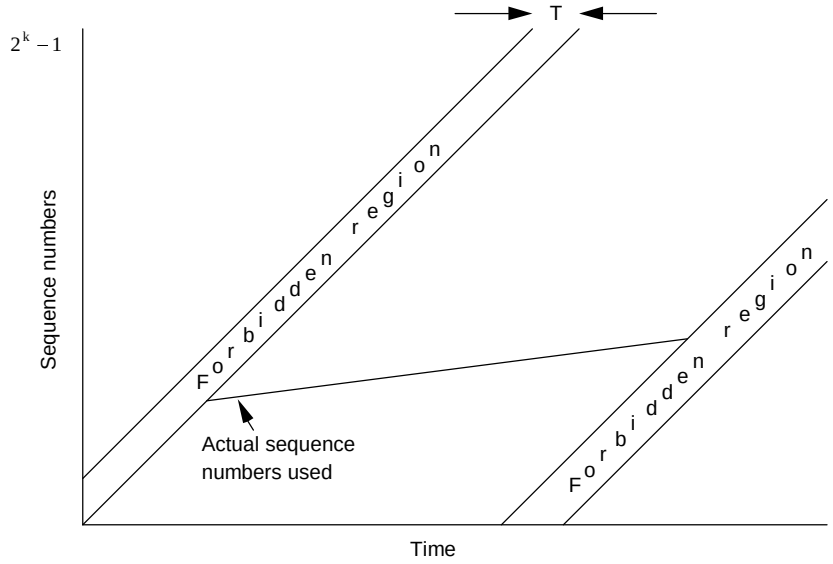
To prevent this, two things are needed:

(1) The clock has to advance fast enough, so that two consecutive packets from the same host do not have the same sequence numbers.

(2) The time it takes for the sequence numbers to wrap around is longer than the time a packet can be delayed in the network. (This time depends on the number of bits in a sequence number and the rate of the clock.)

Even if the above two conditions are met, there is still the possibility of duplicate sequence numbers, when a host has a very slow transmission rate and connections last a very long time. This is illustrated below.

Problems with Wraparound of Initial Numbers



To avoid this, one solution is to resynchronize sequence numbers after a certain time. In TCP, 32 bit sequence numbers are used, and the clock increments every 4 μsec. So wrap-around takes over 4 hours. Thus, this is not a major problem.

Flow Control and Congestion Control

In a network, it is often desirable to limit the rate at which a source can send traffic into the subnet. If this is not done and sources send at too high of a rate, then buffers within the network will fill-up resulting in long delays and eventually packets being dropped. Moreover as packets gets dropped, retransmission may occur, leading to even more traffic. When sources are not regulated this can lead to ***congestion collapse*** of the network, where very little traffic is delivered to the destination.

Two different factors can limit the rate at which a source sends data. The first of these is the inability of the destination to accept new data. Techniques that address this are referred to as ***flow control***.

The other factor is the number of packets within the subnet; techniques that address this are referred to as ***congestion control***.

Flow control and congestion control can be addressed at the transport layer, but may also be addressed at other layers. For example, some DLL protocols perform flow control on each link. And some congestion control approaches are done at the network layer. Both flow control and congestion control are part of TCP. We'll first look at flow control in TCP.

Buffering

During a TCP connection, both hosts allocate a certain amount of buffer space (memory) to be used as a receive buffer for the connection. Received packets are stored in this buffer until the receiving application removes them. A host may allocate certain amount of memory per connection or dynamically allocate buffers as each TCP segment arrives. Here we focus on the former case, where each session has a fixed size buffer.

If the receiving application reads the data out of the buffer at a slower rate than it is arriving, then eventually the available buffer space will be depleted. In this case flow control is used to slow the sender down.

In TCP, flow control is accomplished by having the receiver include its remaining buffer space in each acknowledgement, this is called the ***receive window***. The sender basis the window size for its sliding window ARQ protocol on the receive window. Some implementation details are discussed next.

TCP flow control

TCP uses 32 bit sequence numbers. In TCP, the sequence numbers are based on the number of bytes in each segment. Application data is viewed as a stream of bytes; the sequence number in a segment is the number of the first byte in that segment.

Example: suppose the first segment in a session sent by a host had sequence number 50 (based on the clock at that host). If this segment contained 1000 bytes, then the next segment would have sequence number 1050.

Each TCP segment received from the destination will contain an acknowledgement number N_{ack} indicating the first byte the receiver is expecting (these acknowledgments are interpreted as cumulative ACK's, see lecture 9). A received segment will also contain the receive window in bytes, let R_w indicate this value. In this case the last byte the sending host can transmit (before receiving a new acknowledgment) must be no greater than $N_{ack} + R_w$.

One additional problem that can arise in TCP is if the receive window = 0. In this case, unless the destination sends new data, the sending host may become deadlocked. To avoid this TCP allows the sending host to send segments with one byte of data in them, when the receive window is zero. (This is referred to as Nagle's algorithm, see Sect. 8.5 of Leon-Garcia)