UCSD
DEPARTMENT OF COMPUTER SCIENCE

CS123 Computer Communications: TCP
Fall 2006

Recall that at the start of this course we had shown that TCP is the topmost layer in a hierarchy of abstractions starting with raw bits (see Figure 1) to frames on a link, to packets across a network. If every application had to worry about routing and reliable data delivery, writing a new application like YouTube would be very slow.

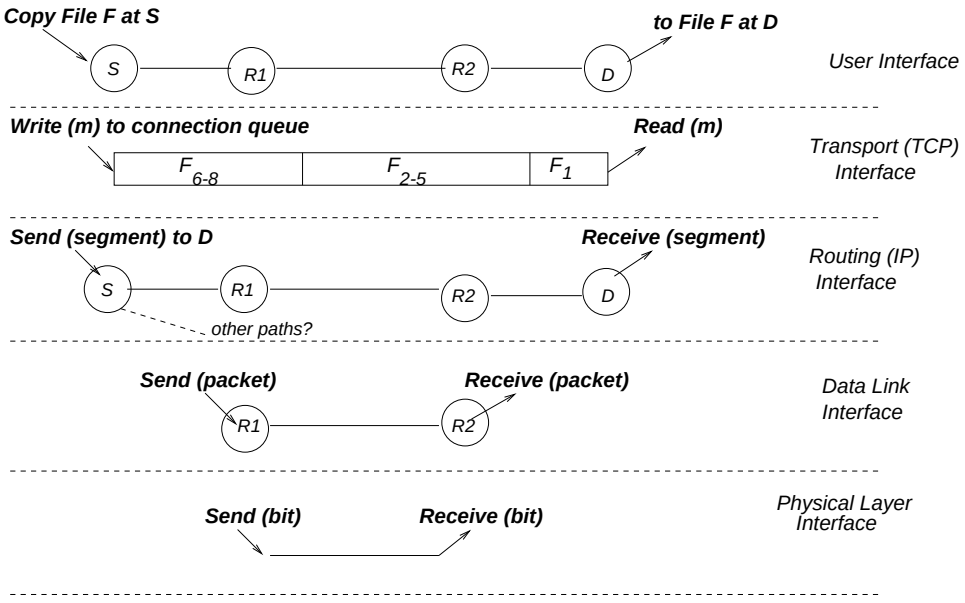


Figure 1: The hierarchy of abstractions we have studied so far. TCP provides the topmost abstraction, a shared queue abstraction

TCP is a transport protocol that provides the abstraction of a shared queue (topmost abstraction in Figure 1 between two processes on two different machines separated on the Internet). These queues are often called sockets. Machine A writes to a socket S1 on machine A. Assuming that the socket S1 is "connected" to a corresponding socket S2 on Machine B, then whenever a process on machine A writes to the socket S1, the data "magically" appears on socket S2 in Machine B. The same happens in reverse when data is written on B to socket S2. The magician is TCP.

Thus a connection is the (virtual) association between a pair of (socket) queues on one Internet machine and a similar pair of queues on another machine. It is virtual because these two machines may not be directly connected to each other. The machines are only connected via IP routing.

To make this abstraction true, TCP must take whatever data is written on a socket queue and deliver it reliably to the remote socket queue. It does using the standard machinery we learnt about in reliable data link protocols. TCP first *packetizes* the data (breaks the queue data into packets as IP can only deliver packets not streams of data), then uses a standard sliding window protocol

to deliver packets reliably to the remote socket with the usual mechanisms of retransmission, acks etc.

Thus to implement this reliable sliding window protocol, TCP needs *state* information such as the current sequence number, current window size etc. If the machine is a server talking to multiple clients, the server must keep separate sequence numbers and state for each client. Even if there are multiple applications talking to different applications between the same pair of machines, it makes sense to keep the data for each pair of machines on a separate connection. Otherwise, if one application is going slow, it can interfere with the other application.

Thus a connection is the shared state information at two machines that describes the progress of a data exchange between two applications at two different machines. In general, each machine has an array of sockets. where each socket has data queues and state information for the sequence numbers. A connection is an association between a socket at Machine A and a socket at Machine B. If two different applications talk concurrently between Machine A and B, they are generally mapped to separate connections.

Unfortunately, this means that when a conversation between two applications is finished, one must *disconnect* the connection, so that the socket queues and state can be used for future connections. This is because there are too many possible clients and servers, to statically pre-allocate connections. This is turn means that when a client wishes to connect to a server (for example, to transfer a file), a new connection must be set up.

This is very similar to Data Link protocols that we studied except for the following differences:

- **Network instead of single FIFO link** The data links we studied assumed that the physical wire was a FIFO data link. TCP works over a network where packets can be delayed for large amounts of time (stored in routers, for example), duplicates can be created by packet looping, and packets can be sent on different routes leading to re-ordering. This implies the need for very large sequence numbers.

  For example, suppose a network can send 64K packets in 1 second and packets can live in the network for 1 second. If the sequence number space is 16 bits (64 K), then it is possible for a sender to send packet number 0 at time 0. Assume packet number creates a duplicate that lives for 1 second. In the meanwhile, the sender keeps sending sequence numbers 1 through 64K - 1 from time 0 to time 1 second. When 1 second comes up, the sequence number space has wrapped to 0. Thus the receiver has no way to distinguish the old duplicate (which could be delivered late by the network) from the newest sender packet. In general, the sequence number space should be larger than the Rate at which the sender can generate sequence numbers * Max Lifetime of the network. TCP uses 32 bit sequence numbers; even this is considered too small today with high speed senders and requires extensions like PAWS (Protection Against Wrapped Sequence Numbers) for adequate protection.

  It should also be noted that unlike in the Data Link protocols we studied, TCP uses a sequence number for each byte of data and not just one sequence number per packet. For example, if a packet has sequence number 10 (which really represents the sequence number of the first byte) and has 50 bytes, the next packet will start not with sequence number 11 but with 60. This allows the receiver to acknowledge the starting part of a sent packet (if it has limited

receive buffer space) without acknowledging the whole packet.

- **Connection management** We have seen that for a Data Link a connection is only set up when a link crashes or comes up. As we have seen above, for TCP we have lots of clients dynamically requesting connections. Recall also that the HDLC restart method did not work in all cases. Since there is more at stake here, we have to do it right. We will describe the method called *3-way handshakes* to allow reliable initialization of connections in these notes.

- **Congestion Control:** Data link only needs speed matching between receiver and sender (flow control). Here we also need speed matching between sender and network (congestion control). This is because some link in the network between sender and receiver could suddenly lower in effective bandwidth during a connection (for example, because several other users began to send packets using that link). Without congestion control, the network could collapse akin to traffic gridlock in a busy city.

- **Minor differences:** Transport needs to dynamically compute round-trip delay to set retransmit timers unlike Data Links. Also, TCP uses a device called fast retransmit to avoid going to Selective Acknowledgement (although TCP SACK is now increasingly being used).

In what follows we first give a quick general introduction to TCP. We then dive into more details such as TCP addressing, the 3-way handshake idea, congestion control and other details.

1 A Quick Introduction to TCP

Figure 2 is an example of a time-space figure with time flowing downwards and space represented horizontally. A line from S to D that slopes downwards represents the sending of a message from S to D that arrives at a later time. Time-space figures will be used throughout this book to depict protocol execution scenarios.

To set up a connection, the sending TCP (Figure 2) sends out a request to start the connection called a SYN message with a number X the sender has not used recently. If all goes well, the destination will send back a SYN-ACK to signify acceptance along with a number Y that the destination has not used before. Only after the SYN-ACK is the first data message sent.

The messages sent between TCPs are called TCP *segments*. Thus to be precise, we will refer to TCP segments and to IP packets (often called *datagrams* in IP terminology).

In Figure 2, the sender is a web client, whose first message is a small (say) 20 byte HTTP GET message for the web page (e.g., index.html) at the destination. To ensure message delivery, TCP will retransmit all segments until it gets an acknowledgement. To ensure that data is delivered in order and to correlate acks with data, each byte of data in a segment carries a sequence number. In TCP only the sequence number of the first byte in a segment is carried explicitly; the sequence numbers of the other bytes are implicit based on their offset.

When the 20 byte GET message arrives at the receiver, the receiving TCP delivers it to the receiving web application. The web server at D may respond with a web page of (say) 1900 bytes that it writes to the receiver TCP input queue along with an HTTP header of 100 bytes, making a

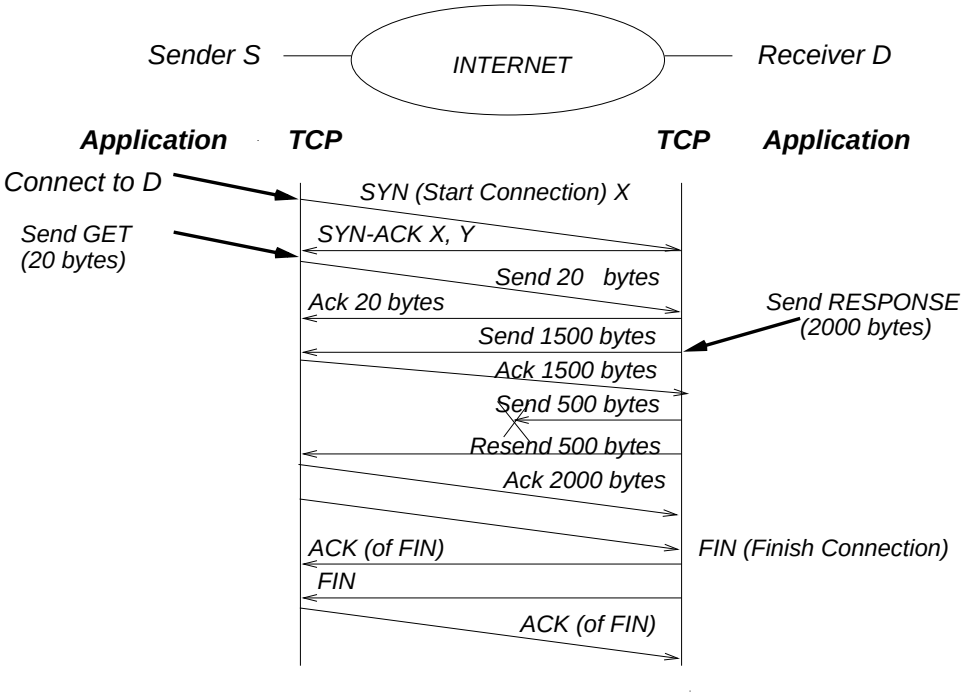


Figure 2: Time-space figure of a possible scenario for a conversation between a Web Client S and a Web Server D as mediated by the reliable transport protocol TCP. Assume that the ack to the SYN-ACK is piggybacked on the 20 byte GET message.

total of 2000 bytes. TCP can choose to break up the 2000 byte data arbitrarily into segments; the example of Figure 2 uses two segments of 1500 and 500 bytes.

Assume for variety that the second segment of 500 bytes is lost in the network; this is shown in a time-space picture by a message arrow that does not reach the other end. Since the receiver does not receive an ACK, the receiver retransmits the second segment after a timer expires. Note that ACKs are cumulative: a single ACK acknowledges the byte specified and all previous bytes. Finally, if the sender is done, the sender begins closing the connection with a FIN message that is also acked (if all goes well), and the receiver does the same.

Once the connection is closed with FIN messages, the receiver TCP keeps no sequence number information about the sender application that terminated. But networks can also cause duplicates (because of retransmissions, say) of SYN and DATA segments that appear later and confuse the receiver. This is why the receiver in Figure 2 does not believe any data that is in a SYN message until it is validated by receiving a third message containing the unused number Y the receiver picked. If Y is echoed back in a third message, then the initial message is not a delayed duplicate since Y was not used recently. Note that if the SYN is a retransmission of a previously closed connection, the sender will not echo back Y because the connection is closed.

This preliminary dance featuring a SYN and a SYN-ACK is called TCP's 3-way handshake. It allows TCP to forget about past communication at the cost of increased latency to send new data. In practice, the validation numbers X and Y do double duty as the initial sequence numbers of the data segments in each direction. This works because sequence numbers need not start at 0 or 1 as

long as both sender and receiver use the same initial value.

The TCP sequence numbers are carried in a TCP header contained in each segment. The TCP header contains 16 bits for the destination port (recall that a port is like a telephone extension that helps identify the receiving application), 16 bits for the sending port (analogous to a sending application extension), a 32 bit sequence number for any data contained in the segment and a 32 bit number acknowledging any data that arrived in the reverse direction. There are also flags that identify segments as being SYN, FIN etc. A segment also carries a routing header[1] and a link header that changes on every link in the path.

If the application is (say) a videoconferencing application that does not want reliability guarantees, it can choose to use a protocol called UDP (User Datagram Protocol) instead of TCP. Unlike TCP, UDP does not need acks or retransmissions because it does not guarantee reliability. Thus the only sensible fields in the UDP header corresponding to the TCP header are the destination and source port numbers. Like ordinary mail versus certified mail, UDP is cheaper in bandwidth and processing but offers no reliability guarantees.

2 TCP Addressing

TCP uses a version of the initial process server model, except with one process for every well known service such as mail. Thus initially a sender may send a connect request to the well known destination mail port, say Port 25. The initial packet is sent to the mail process. However, the sending process also picks an unused source port number, say 11,000. When the receiving server gets the message, it knows it should be sent to the mail. To keep different connections apart that are sent to the same well known port, TCP connections are separated by the entire 4-tuple: Source IP, Dest IP, Source Port, Destination Port. Thus even if two clients happen to pick the same source port number by chance, they will have different source IP addresses and so can be distinguished.

3 3-way Handshakes for Reliable Connection Set up in TCP

Since three way handshakes are such a nice concept and are used to TCP, I hope you will be patient and let me explain them to you using the following analogy. Imagine that John often calls up Martha in the morning and they go through the following sequence:

1. John calls Martha
2. Phone rings at Martha's house and Martha picks up phone
3. John says "Hi"
4. Martha says "Hi"
5. John says "Let's meet today at noon at the student center"

[1]The routing header is often called the Internet Protocol or IP header

6. Martha says "OK"

7. John says "Bye"

8. Martha says "Bye" and hangs up.

9. John hangs up.

This is a simple protocol to coordinate meeting together at the cafeteria. If the phone goes dead after John says "let's meet" John may not know whether Martha has agreed to come. However, John can always call her again to confirm and if he keeps doing this he should be able to get through. The two generals impossibility result only applies if the phone can remain dead till noon. Lets ignore that problem and consider another one, that of detecting duplicates.

Assume that the phone company changes its implementation to packetized voice using datagram routing that can lose, duplicate, and delay voice packets up to a day. Lets say John calls Martha on Monday and they meet on Monday using the protocol shown above. Assume, however, that the network makes duplicates of all the packets that John sends. Then on Tuesday, John goes to the beach for the day. Unfortunately, on Tuesday all the old duplicates of the packets from John arrive at Martha's phone. Thus we have a scenario in which:

1. A duplicate call packet arrives at Martha's home which causes Martha's phone to ring. Martha picks up phone

2. A duplicate packet arrives at Martha's home in which John says "Hi"

3. Martha says "Hi"

4. A duplicate packet arrives at Martha's home in which John says "Let's meet today at noon at the student center"

5. Martha says "OK"

6. A duplicate packet arrives at Martha's home in which John says "Bye"

7. Martha says "Bye" and hangs up.

Martha then goes to the cafeteria. John isn't there. Martha is so mad she breaks off their engagement. John protests later that he was at the beach. Later when he finds out about the real problem, John sues the telephone company. The telephone company calls in Peter Protocol to discuss options to avoid this problem in the future.

3.1 Timer Based Transports

Peter Protocol says that the problem was caused by the fact fact that Martha has no way of distinguishing old voice packets from new. So a simple way to solve the problem is that John would have to number all packets that he sends with a sequence number. Also Martha would have to

remember the last number she received from John in order to detect duplicates (which she discards) because they have smaller or equal numbers than the last one she has received. However, when Martha is told about this she protests that she doesn't want to remember the last sequence number from every possible person who calls her (including toothpaste salesmen, wrong numbers, and other crank callers). Peter Protocol reassures her that she only has to remember the last number from every caller for a period equal to the maximum time that a voice packet can live in the network. However, Martha refuses to remember even the last sequence numbers from all callers who called in the last day (remember the voice network could delay packets for up to a day).

Peter Protocol says "Yeah, that's a problem with timer based transports. They need you to remember information after a call finishes, and that's a real problem if you have lots of calls and the packet lifetimes are large." Timer based protocols were introduced by Fletcher and Watson.

3.2 Clock Based Transports

Peter says that John could always number his packets with the latest time on John's wristwatch. Now a naive scheme based on clocks will not work. For instance, suppose John sends a packet at time t. The packet may take time $t + T$ to get to Martha. Thus, even if Martha has a perfectly synchronized watch, Martha must accept all packets timestamped with time t that arrive at actual times anywhere between t and $t + T$. But now consider that John sends a packet at t and it takes almost no time to arrive at Martha. Then John hangs up and an old duplicate arrives at Martha's before time $t + T$. This duplicate will be accepted! Thus we need more sophisticated schemes in which the receiver keeps some memory. Such a scheme was introduced by Liskov and Shirra.

However, once we add the memory requirement, and the need for clock synchronization between John and Martha, this scheme no longer is as attractive.

3.3 Three Way Handshakes

Peter finally suggests the following modified protocol that he calls a three way handshake.

1. John picks a number x which he has never used before (at least in the last few days). John calls Martha.
2. Phone rings at Martha's house and Martha picks up phone
3. John says "Hi, my validation number is x".
4. Martha picks a number y which she has never used before (at least in the last few days) and says "Hi. My validation number is y, yours is x"
5. John says "Let's meet today at noon at the student center. My validation number is x, yours is y"
6. Martha says "OK. My validation number is y, yours is x"
7. John says "Bye. My validation number is x, yours is y"

8. Martha says "Bye. My validation number is y, yours is x" and hangs up.

9. John hangs up.

Now lets see what happens in the bad scenario above:

1. A duplicate call packet arrives at Martha's home which causes Martha's phone to ring. Martha picks up phone

2. A duplicate packet arrives at Martha's home in which John says "Hi. My validation number is x."

3. Martha picks a number y which she has never used before (at least in the last few days) and says "Hi. My validation number is y, yours is x"

4. John says "Let's meet today at noon at the student center. My validation number is x, yours is z"

5. Martha checks that $y \neq z$ and hangs up, realizing that its a duplicate.

Why did this happen? Since all duplicates were sent by John before the first packet of the duplicate call arrives at Martha, John must be using some validation number $z \neq y$ for Martha. Why? Because Martha has never used the number y before and all validation numbers that John uses for Martha are only numbers that John has received in the past: thus the old number z cannot be equal to the new number y. This allows Martha to reject the duplicate call.

Martha, however, is not convinced this is an improvement. She protests:

- Don't Martha and John have to remember these validation numbers? In that case isn't it better to use timer based transport protocols? Peter answers that that Martha and John need only remember the last validation number they have used in the last call they were involved in. For the next call, they just need to increment the number. This will ensure unique validation numbers. So only one number needs to be remembered as opposed to one per call. (However, if John and Martha are forgetful, Peter suggests they write the number down on a piece of paper. Or else, if they don't want to remember any numbers, he suggests they use random numbers for validation numbers, thus achieving high probability synchronization.)

- Why do both John and Martha need to pick validation numbers? Peter explains that Martha's unique validation number protects Martha against duplicates while John's unique number protects John from duplicates. Notice that John cannot depend on the properties of Martha's number to detect duplicates himself. What's sauce for gander, is sauce for the goose.

3.4 Why is this relevant to networks?

Consider the following protocol to transfer money between John's bank and Martha's bank (after the engagement is broken, John is sending money to Martha to pay her back for all the presents she bought him. Sad story.)

1. John's bank sends a Connect Initiate Packet to Martha.
2. Martha's computer gets the packet and wakes up the bank transfer application.
3. Martha's transport replies with a connect confirm. Sets up a connection with sequence numbers initialized.
4. John's bank sends a packet that says "Transfer 100 dollars to Martha's account."
5. Martha's transport sends a packet that says "OK"
6. John transport sends a packet that says "Disconnect"
7. Martha transport sends a "Disconnect Ack" packet and breaks connection. (i.e., forgets all sequence numbers from John).
8. John then breaks the connection.

Suppose the network duplicates all the packets from John's bank a few minutes later. Then Martha's bank will register another 100 dollars to Martha's account. Martha may be quite happy in this case, but the bank will not be!

Once again all the solutions we talked about in the phone conversation problem apply to this case. We could use Timer Based Transports or we could use three-way handshakes. We can implement validation numbers by keeping a unique number on disk (so it can survive crashes) or we can use random numbers.

TCP uses a 3-way handshake. So does the OSI Transport Protocol. The OSI transport protocol makes the pair of validation numbers (nonces) the connection Identifier and uses separate sequence numbers. TCP uses the 4-tuple: Source IP, Destination IP, Source Port, Destination Port as the connection identifier. Instead of using separate validation numbers, TCP reuses the initial sequence numbers in each direction as validation numbers. The insight here is that sequence numbers do not need to start at 0 as we assumed in the Data Link protocols. They could start at any number as longer as the receiver knows about this. Thus both ends of a TCP connection pick initial sequence numbers they have never used before as validation numbers.

3.5 Crash failures

Transport protocols can be designed to avoid accepting delayed duplicates. But what if you are transferring a file and the system crashes in between. You can't really tell whether the file was written before the crash or after if you don't get any acks back from the receiver. This is a fundamental problem. All a transport can do is to guarantee that your data gets to the other end *at most once* in the face of crashes.

In practice (see example of file transfer done in class), we work around this problem in two ways. We rely on an application level (i.e., file transfer ack) that is sent only after the modified file is written to disk. Of course, if a crash occurs before the application level ack, we can't tell whether the file was written or not. Typically, if this happens the sender will retry later and do the

file transfer again. This relies on the fact that if we do the write of the file one or more times with the same data the result will be the same. Thus, at least in terms of crash failures we rely on the idempotency (i.e., operation can be repeated without any side effects) of application operations. This will not work if the application operation is an operation like "Increment X" which is not idempotent. But its OK for an operation like "Write X = 23".

4 Congestion Control

Recall that besides connection management, a second important difference between a Data Link and a Transport like TCP is the need for congestion control.

Before we describe congestion control, let us describe TCP's fast retransmission policy which is important to understand congestion control. Recall that TCP uses a sliding window protocol and we had studies two variants: Go-back-N and Selective Reject. Go-back N does not accept out-of-order packets, and Selective Reject needs to have acks specify which packets were received out of order. The most common versions of TCP use an intermediate strategy that does better than G-back-N in common cases without the effort of modifying Ack formats.

The idea is that a receiving TCP buffers out of order segments. It also guesses based on duplicate acks that 1 packet is lost. For example, suppose packets 6, 7, 8, 9, and 10 are sent, and the packet 7 is lost. When packets 8, 9, and 10 are received, the receiver buffers them and sends an ack for packet 6. When the sender gets 3 acks for packet 6, it assumes that packet 7 is lost and retransmits that early (without waiting for a full timeout). This leads to a fast recovery from a single packet failure. If there is a full-scale timeout without receiving the oldest expected ack, then the entire window is retransmitted as usual.

There is a move to modify TCP with Selective ACKS (TCP-SACK) to make it approximate Selective Acknowledgement. Once again because of finite space limitations, it does not completely emulate Selective Acknowledgement as we studied it.

Going back to congestion control, the following should be noted about congestion control:

- It is Dynamic Problem, not a static problem. Unlike routing, congestion on links can change every second as new connections come up and down in the network, and depending on user's data usage patterns.
- It can't be solved by rerouting traffic within network; must stop admission to network. At some point one can admit more traffic than the network can handle. In that case, the network drops packets and no packet makes it through completely (each packet makes it partway and then is dropped). This is called congestion collapse. The Internet was in danger of congestion collapse before TCP congestion control was added. (Note that highways have a different model for handling congestion: since cars can't be dropped, it just results in waiting at the previous hop, which leads to gridlock.)

A general framework for congestion control is shown in Figure 3. The pictus shows a network where there is a very fast link followed two 19.2 Kbps links. Without congestion control, the sender

will send at very high rate and many packets will be dropped at the first router. The question is: how should the sender S realize that it should send at 19.2 Kbps. Also, if a new sender S2 wants to also send to D, how should S realize it should now lower its transmission rate to 9.6 Kbps. The answer essentially: is start small, and increase rate: on feedback denoting congestion, reduce rate.

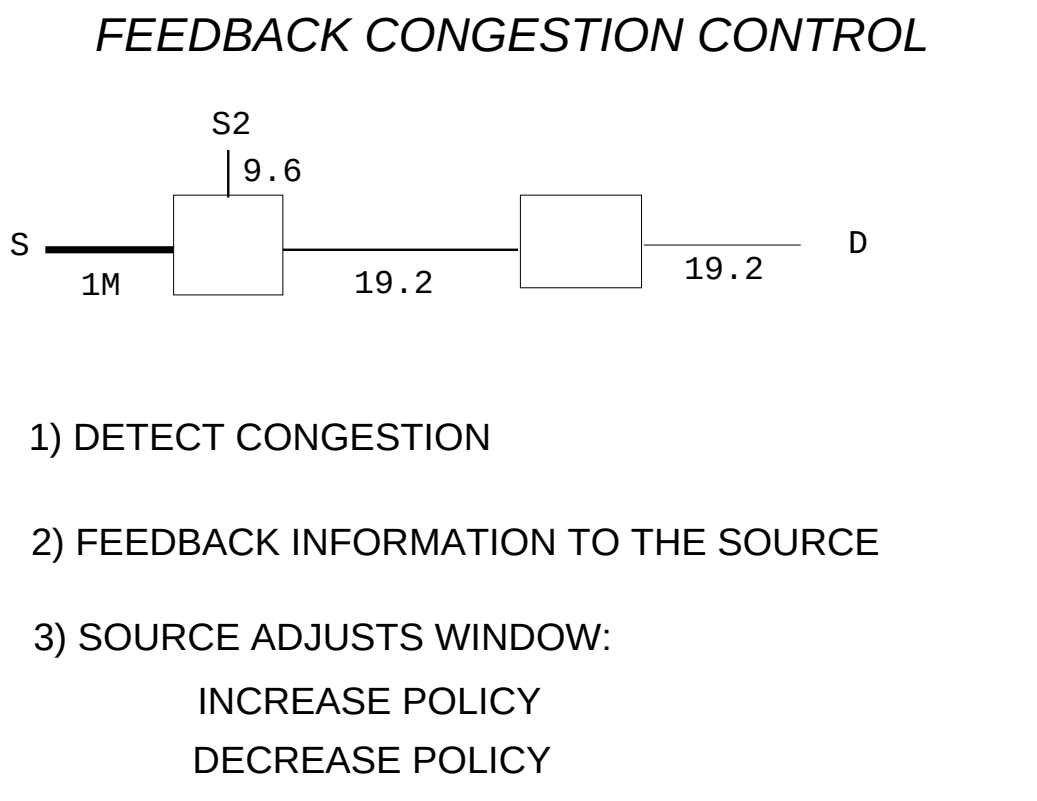


Figure 3: Congestion Control Framework

More specifically, the idea is to first detect and feedback congestion to a source. One method

called DECBit in OSI) was to have a router detecting congestion (for example, the leftmost router) detect congestion on an outbound queue for a link (for example, on the bottlenecked leftmost 19.2 Kbps link) by setting a bit in the packet which gets to the receiver and is sent back to the sender in an ack. Another method used in IP is for a router to simply drop a packet. The sender finds out about this later by a timeout or by fast retransmission.

Sources react to congestion (or lack thereof) by adjusting their window. When a source detects no congestion, it increases its window. When it detects congestion, it decreases its window. Most sources, increase their window slowly, but often multiplicatively decrease their window (e.g., cut their window size by half).

When looking at the dynamics of a congestion control scheme, two simple scenarios are often helpful. First, look at how a single source reaches steady state around the bandwidth of the bottlenecked link. Since a source gradually increases its rate, one question is how fast it takes for a source to reach steady state. Second, an interesting question is what happens when a second source starts sharing the bottlenecked link. The second source should start at a low value and gradually increase its share while the first source should reduce, and the two sources receive approximately equal treatment.

Finally, two terms are helpful. The OSI scheme uses what is known as *congestion avoidance*: in other words, the endnodes try and make sure that they never try and send more than the bottlenecked bandwidth and that the queues in routers never fill up. Another possibility is *congestion control* where congestion occurs packets are dropped and then endnodes backoff to reduce their traffic. Avoidance is better than reactive methods.

4.1 TCP Congestion Control

With the general framework described in the last section (which can fit several other schemes including the OSI DECBit scheme), we now describe the TCP/IP solution.

The top of Figure 4 shows a network connecting source S and destination D. Imagine the network had links with capacity 1 Mbps and a file transfer can occur at 1 Mbps. Now suppose the middle link is replaced by a faster 10 Mbps link. Surely it can't make things worse, can it? Well, in the old days of the Internet it did. Packets arrived at a 10 Mbps rate at the second router which could only forward packets at 1 Mbps; this caused a flood of dropped packets that led to slow retransmissions. This resulted in a very low throughput for the file transfer.

Fortunately, the dominant Internet transport protocol TCP added a mechanism called TCP congestion control that is depicted in Figure 4. The source maintains a window size W, which is the number of packets the source will send without an acknowledgement. Controlling window size controls the source rate because the source is limited to a rate of W packets in a trip delay to the destination. As shown in Figure 4, a TCP source starts W at 1. Assuming no dropped packets, the source increases its window size exponentially, doubling every round trip delay, till W reaches a threshold. After this, the source increases W linearly.

If there is a single dropped packet (this can be inferred by a number of acknowledgements with the same number), the "gap" is repaired by only retransmitting the dropped packet; this is called "fast retransmit" as we said earlier. In this special case, the source detects some congestion and

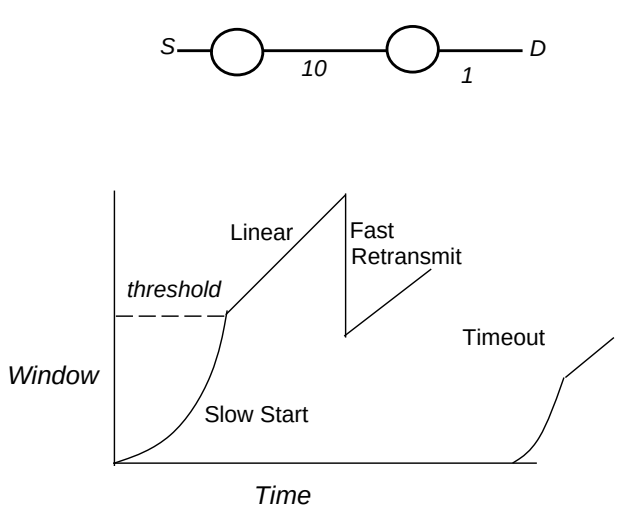


Figure 4: An illustration of TCP congestion control as a prelude to RED.

reduces its window size to half its original size (Figure 4) and then starts trying to increase again. If several packets are lost, the only way for the source to recover is by having a slow, 200 msec timer expire. In this case, the source infers more drastic congestion, and restarts the window size at 1, as shown in Figure 4.

For example, with tail-drop routers, the example network shown at the top of Figure 4 will probably have the source ramp up until it drops some packets, then return to a window size of 1 and start again. Despite this oscillation, the average throughput of the source is quite good as the retransmissions occur comparitively rarely compared to the example without congestion control. However, wouldn't it be nicer if the source could drop to half the maximum at each cycle (instead of 1) and avoid expensive timeouts (200 msec) completely? The use of a RED router makes this more likely.

The main idea in a RED router (Figure 5) is to have the router detect congestion *early*, before all its buffers are exhausted, and warn the source. The simplest scheme, called the DECbit scheme would have the router send a "congestion experienced" bit to the source when its average queue size goes beyond a threshold. Since there is no room for such a bit in current IPv4 headers, RED routers simply drop a packet with some small probability. This makes it more likely that a flow causing congestion will just drop a single packet, which can be recovered by the the more efficient fast retransmit instead of a drastic timeout.[2]

The implementation of RED is more complex than it seems. First, we need to calculate the output queue size using a weighted average with weight w. Assuming that each arriving packet uses the queue size it sees as a sample, the average queue length is calculated by adding $(1 - w)$ times the old average queue size to w times the new sample. In other words, if w is small, even if the sample is large, it only increases the average queue size by a small amount. The average queue size

[2]But what of sources that do not use TCP and use UDP? Since the majority of traffic is TCP, RED is still useful; the RED drops also motivate UDP applications to add TCP-like congestion, a subject of active research. A more potent question is whether RED helps small packet flows such as Web traffic which accounts for a large percentage of Internet traffic.

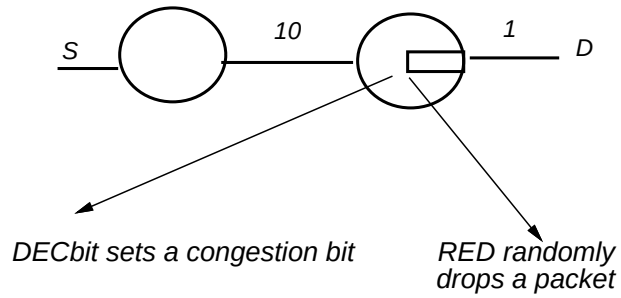
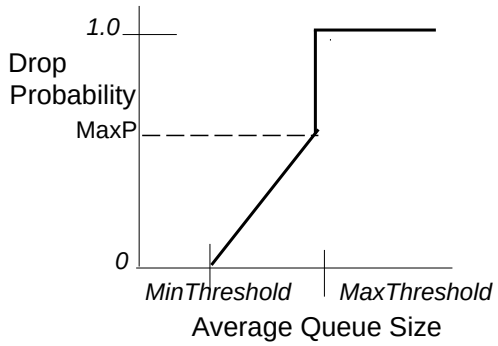


Figure 5: RED is an early warning system that operates *implicitly* by packet dropping, instead of *explicitly* by sending a bit as in the DECbit scheme.

changes slowly as a result and needs a large number of samples to change value appreciably. This is done deliberately to detect congestion on the order of round trip delays (100 msec) rather than instantaneous congestion that can come and go. However, we can avoid unnecessary generality by only allowing the w to be a reciprocal of a power of two; a typical value is 1/512. There is a small loss in tunability compared to allowing arbitrary values of w. However, the implementation is more efficient as the multiplications reduce to easy bit shifting.

However, there's further complexity to contend with. The drop probability is calculated using a function shown in Figure 6. When the average queue size is below a minimum threshold the drop probability is zero, it then increases linearly to a maximum drop probability at the maximum threshold; beyond this all packets are dropped. Once again, we can remove unnecessary generality and use appropriate values such as MaxThreshold being twice MinThreshold, and MaxP a power of 2. Then the interpolation can be done with two shifts and a subtract.



*AverageQ = (1 −w) * AverageQ + (w * SampleQsize)*

Figure 6: Calculating drop probabilities using RED thresholds

But wait, there's more. The version of RED so far is likely to drop more than one packet in a burst of closely spaced packets for a source. To make this less likely and make fast retransmit more likely to work, the probability calculated above is scaled by a function that depends on the number of packets queued since the last drop. This makes the probability increase with the number of non-dropped packets, making closely spaced drops less likely.

But wait, there's even more. There is also the possibility of adding different thresholds for different types of traffic; for example bursty traffic may need a larger Minimum Threshold. Cisco has introduced Weighted RED where the thresholds can vary depending on the TOS bits in the IP header. Finally, there is the thorny problem of generating a random number at a router. This can be done by grabbing bits from some seemingly random register on the router; a possible example is the low order bits of a clock that runs faster than packet arrivals. The net result is that RED which seems easy takes some care in practice, especially at Gigabit speeds. Nevertheless, RED is quite feasible, and is almost a requirement for routers being built today.

Finally, as we said earlier, RED is a substitute for the fact that IP routers could not set a bit in the IP header as in the DECBit proposal. There is a proposal on table for a ECN bit for IPv6. This will become more common.

5 Some Remaining Details

The big differences between TCP and a sliding window protocol are indeed the use of a large sequence number, connection management using handshakes, and congestion control. There are a few other details that are worth knowing about.

First, every TCP connection is to a different destination. Thus to set retransmit timers properly, the round-trip delay must be measured. TCP does so today by measuring the average round-trip delay using an exponentially weighted average; round-trip delay samples are calculated using the time to send a packet and receive an ack. The variance is also calculated and the retransmit timers are set to the Average Round Trip (called RTT) + K * Variance, where K is a small constant.

Second, acks are not send for every data packet received. There is an algorithm called Nagle's algorithm that waits for some time to see if there is reverse data that an ack can be piggybacked. It also tries to send cumulative acks so that one ack often is used to ack two segments. The best source for details is TCP/IP Illustrated by Richard Stevens.

Third, besides congestion control, just as in Data Links, there is also flow control. Flow control is about reacting to slowdowns by the receiving host (which the receiving endnode knows precisely) rather than congestion control (which is speed matching with all the links in the network path). This is shown in Figure 7.

The first part of Figure 7 shows that the window size can always be reduced to reduce the sender rate. The sender keeps two window sizes, a flow control window and a congestion control window (sometimes called *cwind*). The real window is the minimum of the two windows reflecting the fact that the real rate the sender should send at is the smaller of the receiver rate and the network bottlenecked link rate.

A subtlety in flow control is that the receiver could completely shut off the sender if the receiver is going really slow. In that case, when the receiver finally is ready to allow more data, it may send a message increasing the window size. If this message is sent just once, this may result in a deadlock with the receiver thinking it has opened the window once again, but the sender remains unaware of this. In the OSI Transport protocol, the receiver keeps resending its reopened window size until the sender sends an ack; in TCP, the sender must keep probing a closed (i.e, zero value)

window to check if it is now open.

Another complication is out of order reductions in window sizes. For example, at the bottom of Figure 7, an OSI receiver can reduce its window size first to 5 and then to 3. If the 3 comes before the 5, the receiver may take the later 5 to mean that the window has opened again to 5. To avoid this misinterpretation, the receiver in OSI also adds subsequence numbers so that it is clear tha the $A(1,3)$ is the later flow control update. Again, these are subtleties that you can safely ignore in a first reading and they are not even used in TCP.

6 BSD Shared Queue (Socket) Abstraction for TCP

We began this chapter (and the course) by saying that TCP offers a shared queue abstraction called sockets. It is worth ending this chapter by showing you the detailed interface to TCP offered by the kernel to applications. The interface shown in Figure 8 is the classic BSD TCP interface first popularized by UNIX BSD. This interface is still widely used even in other Operating Systems such as Windows.

On the left are the client calls, where a client application (e.g., a web browser or a mail client) calls TCP. It starts with a `socket` call by which the client opens a socket. The TCP module in the kernel allocates a socket to the requesting application[3] and returns the application a small descriptor (much like a file descriptor is returned when a file is opened in UNIX). At this point, the client can `bind` this socket to a particular source port and IP address. While the IP address is most often a single IP address, if there are multiple interfaces (each of which can be a separate IP address and can even have different prefixes), the application must specify *which* IP address this socket binds to.

Next, the client application can request that its local socket be connected to a destination host and port using the `connect` call.

The `connect` interface call will result in the famous TCP initial exchange we know so well, with the SYN being sent with the initial sequence number, and the SYN-ACK returning. When this happens the `connect` call returns. From this point on, the client can send data to the socket queue by using the `send` call (there are other calls as well which we ignore here for simplicity). The `send` call can queue an arbitrary large chunk of data to the socket queue. When this happens, TCP will *internally* break up the socket queue data into packets, add a sequence number, and retransmit them till they are acked.

Finally, when the server sends back data in a TCP packet, any new data is queued in order in the receive socket buffer at the client TCP. It is only delivered to the client when the client does a `receive` call specifying how many bytes it wants to receive.

Turning our attention to the server side, the server application (say a web server or a mail server) also opens a socket and does a `bind`. When it is has finished it does a `listen` call to say that it is willing to listen to data sent (say) on a particular destination port, say Port 80 for a web server. Finally, when a SYN packet comes in for say Port 80, the server application is alerted. It can then choose to `accept` the connection. This results in a SYN-ACK being sent to the client

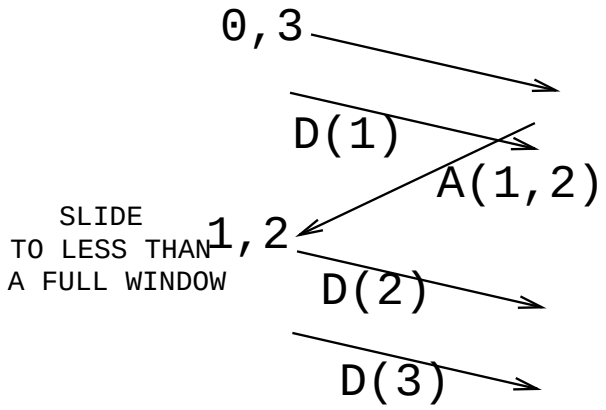
[3]recall that a socket has state for send and receive data queues and other TCP state such as sequence numbers

and the connection being established. Because each new connection requires a new socket at the receiver (you can't share a receive queue from several sending connections), when the `accept` call returns, the application is given a new socket descriptor for this particular connection. This allows the original socket to continue to listen to new connections, while the established connection can do data transfer on its new socket. Thus every concurrent established connection will have its own socket at the server, and hence have separate data transmit and receive queues.

From that point on, the server application can also use the same *send* and `receive` calls to send and receive data.

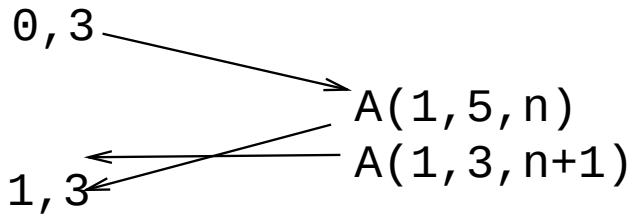
FLOW CONTROL

- Windows provide static flow control. Can provide dynamic flow control if receiver acks indicate what receiver will buffer.



- Need to avoid deadlock if window is reduced to 0 and then increase to c > 0. In OSI, receiver keeps sending c. In IP, sender periodically probes an empty window.

- In OSI, receiver can retract window from say 5 to 3. To detect out of order acks, use subsequence numbers.



- Real Window =Min Flow Control,Congestion Control windows.

Figure 7: Flow Control is used for speed matching with the receiver as opposed to congestion control which is used for speed matching with the network

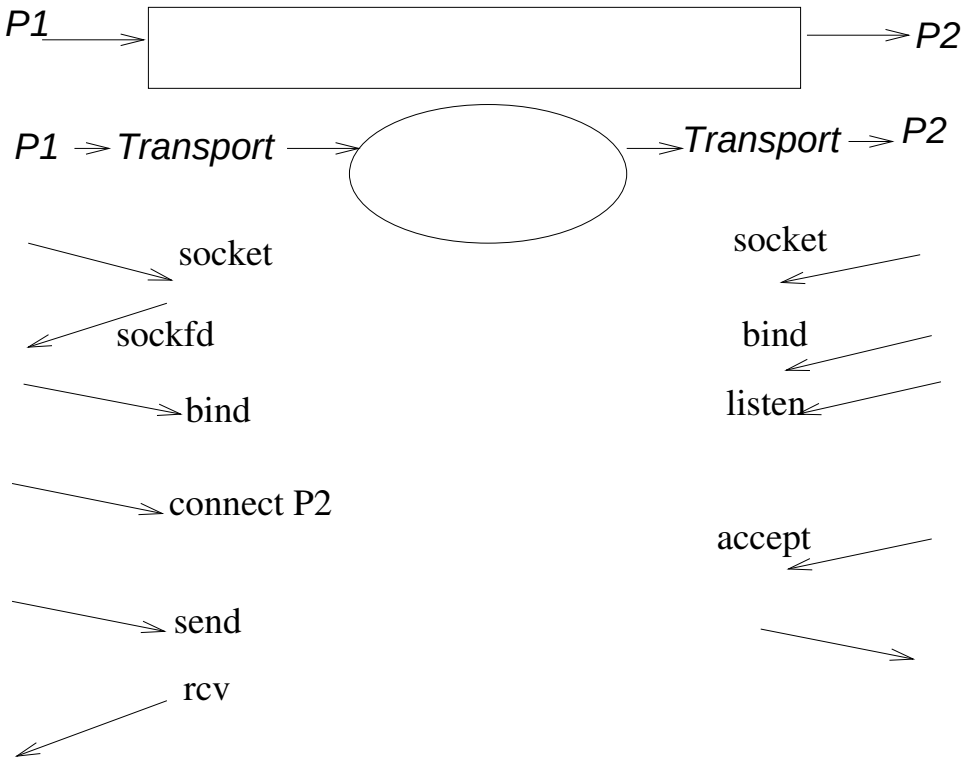
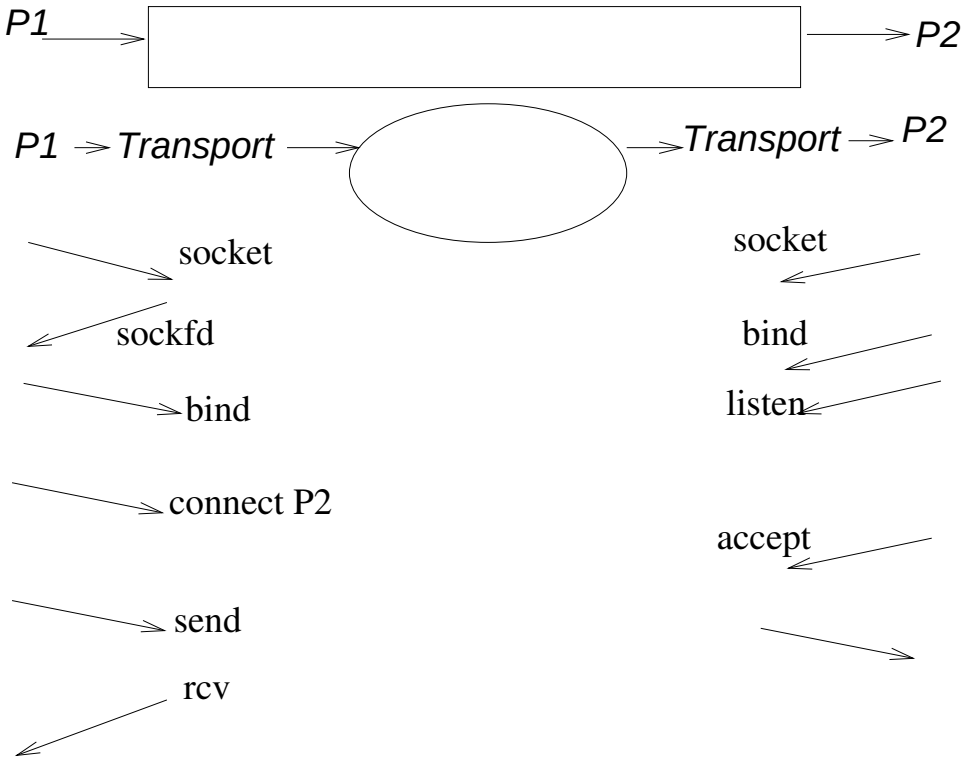


Figure 8: BSD Socket Abstraction API


P1
P2
P1
Transport
Transport
P2
socket
socket
sockfd
bind
bind
listen
connect P2
accept
send
rcv