

Design Engineering of Computer Systems
Professor. Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 33
Transport Protocols

Hello, everyone. Welcome to the twenty third lecture in the course Design and Engineering of Computer Systems. In this lecture, we are going to understand a little bit more detail about what happens at the transport layer of the Internet. So, let us get started.

So, the IP layer that we have seen in the previous lecture basically provides a host to host delivery of IP datagrams. That is one host on the Internet sends an IP datagram to some other host on the Internet, all the IP routers along the path, look at the destination IP address and forward this IP datagram to the other end host. We have seen various mechanisms for how this is done using routers, that run routing protocols, and between two IP routers also how the link layer is present all of that we have seen in the previous lecture.

(Refer Slide Time: 01:13)

Transport layer

- IP layer provides host-to-host delivery of IP datagrams
 - Packets can get dropped, no reliability guarantees
- Transport layer deals with process-to-process delivery of messages
 - TCP (Transmission Control Protocol) guarantees reliable in-order delivery
 - UDP (User Datagram Protocol) does not guarantee any reliability
 - SCTP (Stream Control Transmission Protocol) provides multiple reliable streams over a connection
 - Choice depends on application requirements
- Transport protocols run only at end hosts (end to end argument)
 - Application layer writes messages into sockets, OS does transport layer processing, send packet over network
 - NIC receives packet, OS does transport layer processing, application reads message from socket

+

And we have also seen that the IP layer there is no mechanism to provide any guarantees that is there are no reliability guarantees anywhere in the IP layer. If an IP datagram comes if you can

forward you forward, otherwise you can drop it. So, what the transport layer does is the transport layer runs above the IP layer.

So, the IP layer provides a host to host delivery of IP datagrams and with no reliability, then the transport layer takes this mechanism and on top of it, it builds other mechanisms in order to do a process to process delivery of messages. That is there is one process running on one host, another process on another host. So, the transport layer deals with this process to process delivery. And in addition to this it tries to provide various other guarantees like in-order delivery, reliable delivery and so on.

Of course, there are many different protocols at the transport layer. TCP provides all of these guarantees. UDP is a much simpler transport layer that only deals with process to process and no other guarantees. And you have other transport layer protocols like SCTP, which provides multiple streams over a connection. TCP is just one in order reliable stream, SCTP is multiple streams, but we will not study this in this lecture.

So, there are many transport layer protocols. And on top of this transport layer, you have your application layer. And an application can choose between different transport layer protocols that at once. You open a TCP socket, you will get TCP guarantees. If you open UDP socket, you will get UDP guarantees. And note that the transport layer only runs at the end hosts. This is the end to end argument. The end host, the client and the server will run TCP processing. All these routers in between they are just dumb routers that are just forwarding IP datagrams. They do not know anything about this end to end mechanisms.

So, and this transport layer runs inside operating systems. As we have seen, when you write something into a socket, the transport layer processing is done in the OS and then the packet is sent over the network. Similarly, when you receive a packet, the OS does all the transport layer processing and then gives the application layer message alone to the application that is reading from the socket.

(Refer Slide Time: 03:30)

TCP connection setup

- TCP is connection-based protocol
 - End-to-end connection established between client and server
 - IP routers on path not aware of connection, only forward datagrams
- Connection establishment via 3-way TCP handshake
 - Server opens listen socket and waits to accept, client starts connect system call
 - Client's TCP sends special SYN packet to server
 - Server replies with SYN ACK, client replies back with SYN ACK ACK
 - Connect and accept system calls return after 3-way handshake completes
- After connection established, client and server can exchange data in both directions of connection
- When data transfer done, send FIN and FIN ACK from each side to tear down connection
- UDP has no such concept of connection setup, just send packets directly

So, now, let us understand a little bit more detail about the TCP protocol, which is the most widely used transport layer protocol on the Internet. So, what is TCP? TCP is a connection based protocol. That is when you, if you recollect TCP sockets, you will connect two TCP sockets with each other.

So, what is this connection, what is setup when you do this connection is that, an end to end connection is established between the client and the server, even though no router along the path is aware of this connection. So, what happens when, during this end to end connection is that you have a what is called the three way TCP handshake between your client and your server. Your client and server exchange some messages in order to establish this connection.

How is this done? The server has opened a listen socket and it is waiting for new connections and the client starts a connect system call. And at this point, what happens is the, client machine will send a special packet to the server called the SYN packet saying, hello, I want to talk to you. Then the server, the operating system transport layer code at the server will respond with an SYN-ACK saying, okay, hi, I heard your request. I am here. Let us talk.

Then the client will once again send an SYN-ACK ACK. And this completes your connection setup. At this point, the server, this accept system call will return at the server, the connect system call will return at the client and both client and server sockets are connected with each other. This is called the 3-way TCP handshake.

Why is this three way needed? The client has to call the server, server has to say, okay, I hear you. Then when the server sends a message, the server also needs to know that the client got its message. So, if you think about it, this is common sense between two people talking. Hello, are you there? Yes, I am there. Can you hear me? Yes, I can hear you. With this, this connection is established. Both these hosts can now confidently talk to each other.

Similarly, when the data transfer is done also you will basically have a similar handshake using what are called FIN and FIN ACK messages from each side in order to tear down a connection that is also there. Note that the UDP has no such concept of any connection. Any socket can send to any socket. You just send packets directly. But with TCP since you are maintaining some kind of a reliability and everything, you will do this TCP 3-way handshake in order to connect two different sockets.

(Refer Slide Time: 06:28)

Transport layer segments

- **Segmentation:** message written into socket is split into chunks of MSS (maximum segment size), headers added and sent over network
 - MSS depends on underlying link technologies, which limit max packet size
 - Message boundaries may not be preserved in segments (e.g., one message may span many segments)
- TCP/UDP add source/destination port numbers to header, to help identify different sockets (source/destination IP address is part of IP layer header)
- How to ensure reliability? TCP adds sequence number and acknowledgment number in header
 - Sender puts sequence number of the starting byte present in the packet
 - Receiver replies with sequence number of the next byte it is expecting
 - Receiver's ack is cumulative, and indicates that everything up to that sequence number has been received
- TCP is bidirectional stream, each side sends a sequence number and ack number
 - Ack piggybacked with data in other direction, or sent in a separate packet
- Other fields in TCP/UDP segment header: packet size, check sum

Handwritten notes: "write 64 KB" with an arrow pointing to a sequence of boxes. A diagram shows sequence numbers 0, 100, 200 and acknowledgment numbers 100, 200. A red 'S' and 'C' with arrows indicate bidirectional communication.

And once this connection setup is done, then the transport layer will start sending segments. The client and server will start sending segments to each other, transport layer segments to each other. What is a segment? A segment is nothing but whatever application layer message you receive plus some TCP headers added to it.

So, the message that is written into the socket is first split into smaller chunks and this chunk size is called the maximum segment size. So, you cannot, if the application writes a large message you cannot send all of it in one transport layer segment, because the underlying link layer

technologies the Ethernet has certain constraints on how big these messages can be. Therefore, what you will do is the transport layer will first split a message into smaller chunks of size MSS or the maximum segment size.

And note that message boundaries are not preserved when you make these segments. That is if you write 64 kilobytes into a socket in a write system call all the 64 bytes would not be in one segment. If your maximum segment size is 1 kb, then it will be split into 64 segments of 1 kb each and sent out over the network. So that is one thing that the transport layer does. It creates these segments which later on become packets over the network.

And what else will you add to the segment. Of course, one thing you will add is port numbers. You have multiple processes on computer which are identified different sockets are distinguished by port numbers. Therefore, in the transport layer TCP, UDP both of them will add port numbers. And the IP layer will add the IP addresses. Together between the transport and IP layers you have the source destination port numbers as well as the source destination IP addresses added to the headers in the packet.

And the other fields that are added are various things like your packet size, checksum and all of that, which again both TCP and UDP will do. But in addition to these port numbers, IP address, packet size, checksum which are the bare minimum things, TCP also adds a few extra fields to the packet header for the purpose of reliability, which is the sequence number and the acknowledgement number.

Note that this is not added by UDP which does not care about reliability but only by TCP. So, what is the sequence number? Sequence number basically tells you the starting byte that is present in the packet. Packets are given sequence numbers so that you can keep track. This is the first packet. This is the second packet. This packet is lost. This is received. In order to be able to do this you need to be able to identify the packets in some way, number them in some way. That is what you use sequence numbers for.

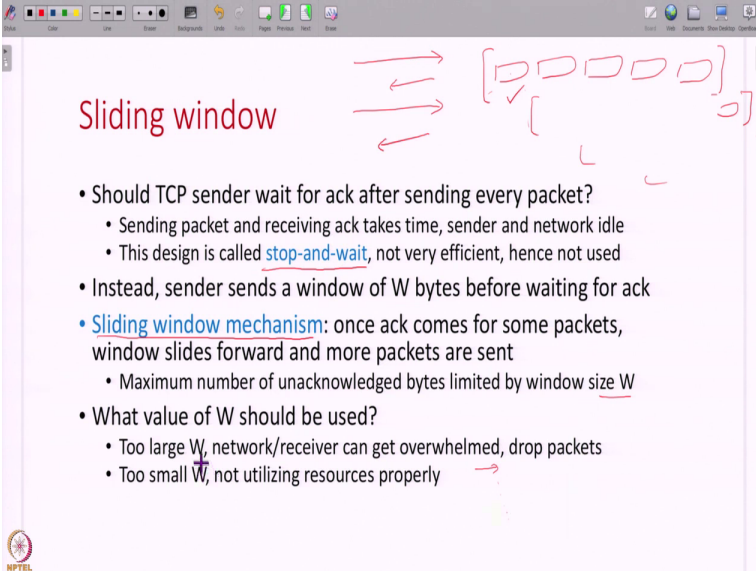
So, the sender puts the sequence number of the starting byte in a packet. If a packet has the first 100 bytes the starting sequence number will be 0. The next 100 bytes the starting sequence number will be 100. The next 100 bytes a starting sequence number will be 200 and so on. So, we have byte based sequence numbers put in packet headers by TCP.

And when the receiver receives a packet it will send back an acknowledgement number which is the sequence number of the next byte it is expecting. So when the sender sends the 0 to the first 100 bytes, byte 0 to 99 then the receiver will send an acknowledgement with a number equal to 100 which is saying I got everything before 100 send me 100 next. I am waiting for 100. When this packet is received, the receiver will send an acknowledgement saying 200 which is I got everything up to 200 send me byte number 200 next.

So, that is the, of course, you could have put sequence numbers to be packet base sequence number, acknowledgement numbers to be packet level acknowledgement numbers, but TCP prefers to use this byte wise semantics. The other thing to note is the receivers' acknowledgement is cumulative. It indicates that everything up to this byte has been received. And the sequence numbers and acknowledgement numbers are there in both directions. TCP is a bidirectional stream. Once a client and the server have connected with each other, the client can send bytes, the server can also send bytes.

So, for this direction you will have a sequence number and for this direction also you will have a sequence number. Similarly, for this direction you will have acknowledgement, for this direction you will have acknowledgement and these acknowledgement numbers can be sent with the data in the reverse direction or as separate packets. So, this concept of sequence numbers and acknowledgement numbers is used by TCP to guarantee reliability.

(Refer Slide Time: 11:22)



Sliding window

- Should TCP sender wait for ack after sending every packet?
 - Sending packet and receiving ack takes time, sender and network idle
 - This design is called stop-and-wait, not very efficient, hence not used
- Instead, sender sends a window of W bytes before waiting for ack
- Sliding window mechanism: once ack comes for some packets, window slides forward and more packets are sent
 - Maximum number of unacknowledged bytes limited by window size W
- What value of W should be used?
 - Too large W, network/receiver can get overwhelmed, drop packets
 - Too small W, not utilizing resources properly

So, now, the next question comes up the sender is sending a packet, waiting for an acknowledgement, then how do you do this? Do you wait for every packet? If I send one packet do I wait for an acknowledgement for every packet? Note that waiting for acknowledgement has to be done at some point that is essential for reliability. Otherwise, if you simply send packets do not see what is being acknowledged then you would not have reliability. But the question is when do you wait? Do I send a bunch of packets and wait or do I wait for each packet.

So, there are two different ways of doing this. You can do what is called a stop and wait design, which is sender sends one packet waits for acknowledgement, then sends the next packet waits for acknowledgement, but this is very inefficient, because packets take a long time to reach the other side and you are wasting a lot of time waiting for acknowledgements.

Instead, what TCP does is it does what is called a sliding window protocol. That is, the sender will send a bunch of packets, a window of packets and then wait for acknowledgement. Instead of just sending each packet and waiting you send a bunch of packets and then wait for acknowledgement. And of course, you cannot keep sending forever. You will have to put some limit on this window size.

And then by the time you have sent all of these packets suppose an acknowledgement for this packet comes, then your window of packets has moved, then you will send one more packet. Now, if this is also acknowledged, then you will send these next set of packets, these set of packets. You keep sliding your window as acknowledgements come. That is why it is called sliding window. You send a window of packets, as the window packets keep getting acknowledged, you keep moving your window forward. This edge of the window keeps moving forward.

Now, the question comes up what is this maximum window size that you should use. You cannot clearly keep sending forever. You have to at some point wait for acknowledgements. What is this value this maximum window size that we have to use. That is the question that TCP tries to answer. If you use too larger window size, then what will happen, you are just dumping packets into the network. You can cause congestion, bad things can happen.

If you use too smaller window size like stop and wait, then what happens. You are not using your resources properly. You are sending a packet. Then for a long time you are not doing anything

because you are waiting for acknowledgement. Therefore, your window size should be optimally tuned. So, now that TCP uses a sliding window protocol, let us fully understand how it handles reliability.

(Refer Slide Time: 14:05)

The slide is titled "Reliability" in red. Above the title is a diagram showing a sequence of boxes representing data segments with sequence numbers 0, 100, 200, and 300. Arrows indicate the flow of segments and acknowledgments. A bracket under the first three segments is labeled "window".

- TCP sender transmits multiple segments, pauses if window is full
- Upon receiving a TCP segment, receiver sends ack back to sender
 - Ack sequence number is next in-order byte number expected
 - Out-of-order segments received are not reflected in ack number
- When sender receives ack, window slides forward, more data can be sent
- If a segment is lost, will result in duplicate acks from receiver (dupack)
 - A single dupack can also be due to reordering, so sender does not panic
 - If 3 dupacks for a sequence number, sender infers loss, retransmits lost segment
- What if severe congestion, all segments/acks are lost?
 - Sender maintains a retransmission timer for every segment
 - On timer expiry, timeout and retransmit everything
- What if data received at receiver, but ack is lost?
 - Sender retransmits segment unnecessarily, receiver identifies and discards duplicates
- Receiver assembles segments, sorts by sequence number, delivers to app in order

So, the sender sends multiple segments with increasing sequence numbers and it has some notion of what is the maximum window size that I have to use and until that window size is hit it will keep on sending segments. Now, when a receiver receives a segment, it will send an acknowledgement back to the sender. So this is a basic mechanism needed for reliability. And these acknowledgement sequence number will indicate the next in order byte expected.

So suppose this sequence numbers are 0, 100, 200, 300 and so on, when the receiver gets this packet it will send an acknowledgement number of 100. If it gets, after it gets this packet it will send an acknowledgement number of 200, 300 and so on. And of course if you receive any jumbled in packets, you will not send those acknowledgement numbers. You will only send a cumulative acknowledgement number.

If the receiver has received some packet over here and then it will still continue, and not this packet, then it will still send this acknowledgement number only. It will not acknowledge out of order packets that is not possible in TCP. It is not possible to do. Now, when the sender, so once the receiver sends this acknowledgement, when the sender gets this acknowledgement, it will advance its window size and the window will keep moving forward.

Now, what happens when data is lost? How are we guaranteeing the reliability here? Now, suppose some segment this segment is lost so, and then this segment is received, if the segment starting at 100, byte number 100 is lost, but segment at byte number 200 is received, then what will the receiver do? It will still send an acknowledgement back for byte 100 saying I am waiting for byte 100. When segment 300 is received, it will once again send an acknowledgement saying I want byte 100. The next segment is received, it will once again send an acknowledgement saying I want byte 100 that is we will have duplicate acknowledgements.

Every time the receiver receives a packet it will send back an acknowledgement for the next in sequence byte it is expecting. And therefore, with these duplicate acknowledgements, a single duplicate acknowledgement can be due to some reordering of packets that is okay. But once you get three duplicate acknowledgements, the sender thinks something is lost, something bad has happened and it will retransmit that lost segment.

And once you retransmit the segment when the receiver gets it, then all of these segments are also received, then it might send back one big, the next acknowledgement can be over here. You can skip all of these and say oh I got all of these send this byte next. The receiver can do that. So, of course, this duplicate acknowledgement to detect loss can only happen if your some packets are going through, some are loss, some are going through. But what if everything is lost? The sender has sent some 10 packets everything is lost segments, acknowledgements, everything. In that case, you would not get these duplicate acknowledgements.

Then what will the sender do, how will the sender realize that a loss has happened? The sender, that is why the sender also maintains a timer. For every segment you send you will maintain a timer. Within that timer if duplicate acknowledgements are received, you will realize the segment is lost, you will transmit it. If no duplicate acknowledgements are received and no acknowledgement has come at all then when the timer expires, you will timeout and you will retransmit everything.

So, in this way using some combination of timeouts and duplicate acks losses are detected and the sender will retransmit. And eventually after sending multiple times it will eventually reach the end. You will get an acknowledgement. The other thing that can happen is your data can go through but your acknowledgements can be lost, in which case the sender will think something bad has happened and retransmit, but that is okay. The receivers, TCP receivers identify these

duplicate packets and they want deliver duplicate packets to the application, the TCP layer will filter out these duplicates using sequence numbers.

In the end, the receiver all the packets it will assemble them, sort them in order of sequence number and when an application reads from a socket this in-order stream is delivered to the application. In this way, TCP takes care of reliability. When you write and read using TCP sockets, you are getting this reliability, you are guaranteed an in-order reliable byte stream. So, the next thing is, of course, we have conveniently skipped this question of what is your window size. Your window size cannot be too big, it cannot be too small. What should it be?

(Refer Slide Time: 18:53)

Bandwidth delay product

- How to compute window size in sliding window protocol?
- Consider the following toy example
 - Suppose network can send 10 packets/sec (bandwidth)
 - Round trip time (RTT) is 2 seconds, i.e., it takes 2 seconds for a packet to reach receiver and ack to come back
 - After sender sends 20 packets (bandwidth delay product), the ack for the first packet would have come back, and sender can send more
- Ideal sliding window size = bandwidth delay product (BDP) of connection
 - If window size > BDP, congestion in network
 - If window size < BDP, sender is idle
 - But BDP is hard to estimate (bandwidth and RTT highly variable)
- TCP sender computes congestion window size (cwnd) using heuristics

So, let us just see a small example. Suppose your network speed is such that you can send 10 packets per second, anything faster than that your network kind of gets blocked. Then your round trip time, when you send a packet it goes to the other side, traverses all the routers on the Internet and comes back, your round trip time is say 2 seconds. So, what does that mean? Once the sender starts sending packets, once the sender sends 20 packets, you can only send 10 packets per second in 2 seconds that is the product of this bandwidth and delay that is called the bandwidth delay product.

Once you send 20 packets in 2 seconds, what has happened? The acknowledgement for your first packet has come back. Your first packet, then 19 more packets, at the end of which your first packet has been acknowledged, therefore, you can send one more, then your next packet is

acknowledged, you can send one more. So, if you are sending at the rate that your network supports and you have a certain delay, then by the time you send your bandwidth delay product or BDP worth of packets the ack for the first one would have come through. And therefore, your ideal window size in this ideal world is basically your bandwidth delay product.

You take the bandwidth of your network, the rate at which your network is able to send your traffic, you multiply it with the round trip time, you get your BDP, use this as your BDP. If you use this as your BDP, then everything will be perfect. You are going at a smooth rate. If you send more than this BDP what will happen?

Your network is not able to handle that. Your network will get congested. If you send less than your BDP, what will happen? Your network is idle. Your sender is idle. If you send one packet and wait for two seconds, then you are just wasting the network. But if you send at your BDP at this exact rate at which your network can take it then you have an ideal situation. So, this is what TCP will want to do.

However, finding out this BDP is very hard in real life and it is not possible because the, what do you mean, the Internet is large, complex, you do not know what the bandwidth available to you is, the delays are highly varying. So, this BDP is not, in general, in real life, it is not possible to know it. This toy example we could calculate but real life is not like that. Therefore, in real life what TCP will do is it somehow tries to estimate this BDP using very rough heuristics.

So, TCP calculates what is called the congestion window or the cwind using some heuristics to approximate the BDP in some sense. So, that is called congestion control. Calculating this congestion window and limiting yourself to this congestion window in order to not cause too much congestion in the network at the same time to send as much data as possible that algorithm inside the TCP logic that does this is called the congestion control algorithm.

(Refer Slide Time: 21:56)

Congestion control

- Ideally, sender sets cwnd to be BDP, but BDP is difficult to estimate
- Instead, sender relies on feedback from network to adjust cwnd
 - If packets are going through, maybe cwnd is below BDP, send more
 - If packets are getting lost, cwnd may have crossed BDP, slow down
 - Packet loss is simplest form of feedback about congestion
- A simple congestion control algorithm to compute cwnd
 - Start with cwnd = 1 MSS
 - Initially, ramp up cwnd quickly, double cwnd every RTT (slow start)
 - After a threshold, be more careful, increase cwnd by 1 MSS every RTT (additive increase)
 - 3 dupack, slow down, halve the value of cwnd (multiplicative decrease)
 - If timeout, restart from beginning
- Different TCP variants use different congestion control algorithms for different types of applications, networks
 - Approximate heuristics, no one best congestion control algorithm

So, this congestion control algorithm relies on some feedback from the network. How do you adjust your congestion window? You do not know what is happening in the network, but you can only infer. So, if packets are going through, then maybe you are sending below the BDP. You can probably increase your congestion window. You can increase your window size. If packets are getting lost that means something bad is happening, the network is congested, some switch is not able to, some router is not able to handle all this traffic and it is dropping packets, therefore, you may have to reduce your congestion window. You may have to slow down.

So, most congestion control algorithms today simply rely on packet loss. If packets are getting lost, slow down. If they are not getting lost, continue to send. So, this is called the congestion control algorithm inside the operating system, inside the TCP layer of the operating system. So, very simple congestion control algorithm could look like this.

Start with sending one segment. And initially, you can ramp up quickly, every RTT double your congestion window. I have sent 1 segment, in the next round trips send 2, in the next round trip send 4 segments, in this way you keep on doubling your congestion window that is called the slow start.

But of course, this is initially to ramp up quickly. But after some time, you have to be more careful. What if you know congestion will happen then you do not want to be sending a large

window size and realizing packets are getting lost. So, instead what you will do is you will be more careful.

After some time you will get into what is called the additive increase phase where you will only increase your congestion window by one segment every round trip time. The, if your round trip time is 2 seconds send 5 packets, next 2 seconds, after 2 seconds send 6 packets, after 2 seconds send 7 packets in a window. That is how you will increase your window size more slowly.

That is if you plot the congestion window as a function of time initially you will rise rapidly then you will slowly increase, linear increase. Then if something bad happens, if for dupack comes you know packet has been lost then you half your condition window. You come down. Then again you increase slowly. Then again some loss happens you slow down. Then again you increase slowly, slow down. This is called the additive increase, multiplicative decrease.

When you decrease you will decrease drastically. You will half it. When you increase you will increase slowly. And of course if some timeout or something happens then you will restart all of this again. You will go back to slow start. You will start from the beginning. And this is a very simple algorithm but different TCP variants will use different congestion control algorithms like this and of course there is no one standard method to do this, but there are different heuristics that are adopted by different algorithms.

(Refer Slide Time: 24:57)

Understanding congestion

- What happens inside a router?
 - Look up destination IP address of received datagram, find next hop and outgoing link
 - If outgoing link is busy, packet is queued up until it can be transmitted
- What is congestion?
 - Network is a pipeline of links, the slowest link becomes the **bottleneck**
 - Queue builds up at the head of the bottleneck link, in bottleneck router
 - If queue at bottleneck router overflows, packets are dropped
 - Different connections may have different bottleneck links
- Can we detect congestion before it causes packet drop?
 - Some routers can warn when queue starts to grow, before buffer fills up completely
 - When queue size crosses threshold, Random Early Detection (RED) routers drop packets with some probability, or set Explicit Congestion Notification (ECN) mark
 - ECN-aware TCP can use these warnings to adjust cwnd before packet drop

So, now let us just look inside a router to actually understand what is happening when there is congestion. A router it gets packets from multiple links and it will look up some routing table, the forwarding table, look up the destination IP address and find out should I send the next packet to this other guy or this guy or this guy the router makes that decision. And the packet comes in, datagrams are coming in, the decision of, forwarding decision has been made and then if this link is free the packet will be sent on the outgoing link, but if the link is not free, then the packet will be buffered up.

You will have some small buffer here where you are storing all the packets. And this queue is of a finite size. So, if this link is very slow, in any network there will be one link. If the network is composed of multiple links, if this link can send 100 packets per second, this can only send 10, then at this point, the queue will build up. So, on a road, on a narrow road is where the traffic will stop. We all know this. It becomes the bottleneck.

So, the slowest link in the network becomes the bottleneck that is called the bottleneck link. And at that bottleneck router at the queue will build up that router will no longer, a lot of data is coming in the router is no longer able to send it out on the link and the packets will keep getting buffered. They will keep getting stored at the router at some point the space at the router, the storage space at the router runs out and packets start getting dropped and this is how congestion happens.

And when packets get dropped, then the senders will get that signal and the congestion window will reduce and this congestion will reduce that is the sort of the inner details of how congestion is happening on the network. So, you might say why should we wait for packets to get dropped and then slow down. When this queue is building up itself cannot we realized if you are going into a traffic junction, you see a lot of vehicles backed up you might just take a different route beforehand itself, like you do not sit there for one hour and then realize there is congestion.

So, this is possible. There are modern routers today, where, when the queue size starts to increase itself, they will detect it and that is called random early detection. So, before the queue overflows itself you can either drop packets or you can set a mark on packets, you can, there is something called explicit congestion, notification, support available in routers. So, you can say the queue is building up and you can set that signal on the packet.

So, that if your TCP is aware of all of these things, it can slow down even before packets are lost. So, especially in networks like data center networks, where performance is very important, you can use all of these optimizations. So, now, let us understand end to end, you are sending a window of packets from the sender to the receiver, every packet, what are all the delays that this packet experiences.

(Refer Slide Time: 27:55)

End-to-end delay in the network

- On every link in the network path, a packet experience delays
 - Transmission delay: time taken to put a packet onto the link
 - Propagation delay: time taken for the signal to reach the other end of link
 - Processing delay: time taken to process packet, look up forwarding table, ...
 - Queueing delay: time spent waiting in queue at router
- Round trip time RTT = sum of all delays for both data packet and ack
- BDP = bottleneck link bandwidth X RTT
- Varying network characteristics across different networks
 - Data center network paths have high bandwidth, low RTT (few milliseconds)
 - Internet-wide network paths have lower bandwidth, higher RTT (tens to hundreds of milliseconds)

So, when you take a packet and you have to transmit it, there is a basic transmission delay. Your link has a certain speed. It might take a few microseconds to actually transmit the packet on your link that is called the transmission delay. Then you have the propagation delay. Once you have put out a signal on the wire there is the speed of light. It takes some time for the signal to reach the other end that is the propagation delay.

Then every router has to process a packet. Once the packet is transmitted, propagates, receives, the other endpoint has to process the packet, look up the forwarding table, make some decisions, all of that is the processing delay. Then finally, if the packet cannot be sent out on any link, because the link is slow then the packet will get queued up and it will wait in a queue that is called the queuing delay.

So, for every packet you add up all of these delays on every link that is your round trip time. When you send a packet for all the data, for all the links of the data packet you add up these delays then for all the links of the ACK you add up these delays that is the round trip time in

your network. That is the amount of time if you send a packet it will take this much time to get an acknowledgement back. And your BDP is nothing but your bottleneck link bandwidth multiplied by this RTT.

And note that different networks will have different values of bottleneck bandwidth and RTT and so on. In data centers you might have very high bandwidth links connecting, different servers and you can have very low RTTs, like a few milliseconds, but on the Internet if you are talking to, if you have a network, wide area network and your client is on one side of the Earth and the server is on the other side then your RTT can be very large. It can be tens to hundreds of milliseconds and you can have lower bandwidth also, because there are, there could be many some slow link along the path.

So, different networks will have different characteristics. The BDP value is different. There is no easy way to find out what this BDP is and TCP uses some heuristics in its congestion control algorithm to estimate what is the best window size.

(Refer Slide Time: 30:05)

The slide is titled "Flow control" in red. It contains a list of bullet points discussing network flow control scenarios and solutions. A hand-drawn diagram in red ink shows a circle labeled "read" with an arrow pointing to a box labeled "TX" and another arrow pointing from a box labeled "RX" back to the "read" circle.

- What if network is fast, but receiver is slow?
 - If receiver OS is slow to handle interrupts, device RX ring will overflow, packet drop
 - If receiver application is slow to read from socket, socket RX queue will fill up
- TCP receiver indicates space left in socket RX queue in every ack (called receive window size)
- Sender sets window size to be minimum of cwnd, receive window size
- Flow control: sender slows down in order to not overwhelm receiver
 - Different reason for slowing down as compared to congestion control
- Ideally, receiver must set socket RX queue size to be at least equal to BDP, so that receive window is not reason for low throughput
- Network tools (e.g., iperf) run client and server on two hosts and report end to end TCP throughput achieved
 - If TCP throughput is low, but network is uncongested, can increase RX buffer size

And the final concept with respect to transport layers that I would like to discuss is what is called the flow control. So, so far we have just been concerned about the network. The network is getting congested. Network is dropping packets. But what if the network is fast, but your receiver is slow. If you recollect what is happening at the receiver, when a packet comes in, the packet is put into, your device driver has TX RX rings. You put the packet in here.

Then the OS handles the interrupt, processes the packet and then in your socket queue you will add this packet to the socket queue and then your application will read from the socket and that is when the socket queue is empty. We have seen all of this last week. Now, if your receiver application is too slow or your operating system is too slow, then all of these queues also get filled up.

For example, if your application is not reading packets fast enough, your application is very slow, then even if your network is very fast all the packets will come and wait here, in that case also the sender has to slow down that is called flow control. The sender slowing down in response to a slower receiver is called flow control, whereas a sender slowing down in response to a slower network is called congestion control.

So, how does the sender know that the receiver is slow? In every acknowledgement you will tell to the sender how much space is left in your socket received queue. If your socket received queue has very few bytes left, you will tell that in the acknowledgement. So that then the sender will set the window size to be the minimum of the congestion window and the receiver window.

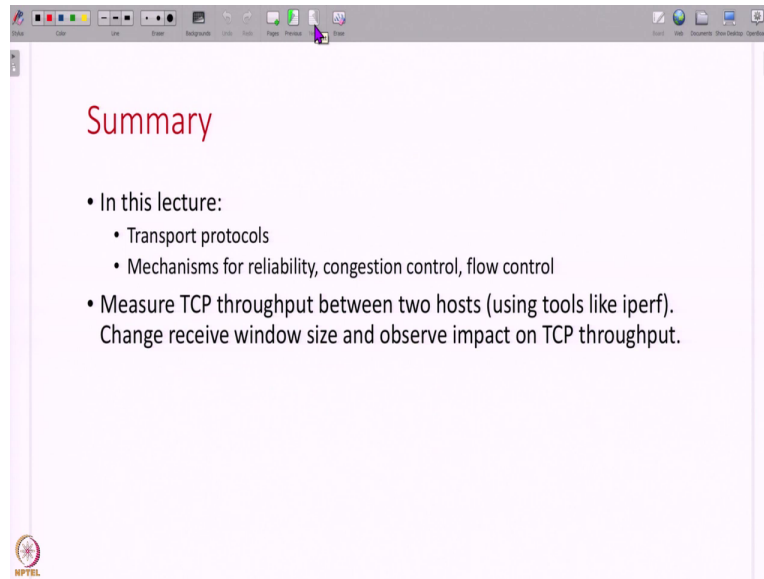
If the network is slow, the congestion window will be low, you will use this value. If the network is fast your congestion window can be very high. But the receive window will be smaller. Therefore, the minimum of these two values will be used. So, TCP does both congestion control as well as flow control.

And the other thing that you should do is you must set your socket queues, receive queue size to be at least as big as your bandwidth delay product. If your receiver is fast, but then this queue does not have enough space, then that is not good enough. Therefore, if sometimes, you might find that your connection is very slow, but your network is fast, in that case, the problem could actually be your received buffer. Your receive queue is so small that your acknowledgement is telling I have no space left and your sender is slowing down. Even though your network is fast, that can also happen.

So, we have tools like iperf. For example, in Linux which lets you measure the bandwidth between a sender and a receiver, a client and a server in a system, and if you find that when you measure this iperf bandwidth and you see that the bandwidth is low, but your network is very fast, then this receive buffer size is something that in computer systems you should always be

mindful about and see if you want to tune it if that is actually limiting your sender slowing down your sender in a network.

(Refer Slide Time: 33:19)



So, that is all I have in today's lecture. We have discussed transport protocols. We have seen how transport protocols like TCP use a sliding window of packets. And they use acknowledgements for reliability, estimate the congestion window size in order to not cause congestion in the network, slow down if the receiver is slow using flow control mechanism. So, all of these together will ensure an end to end reliable in order byte stream delivery of packets on the Internet.

So, to understand these concepts better please use tools like iperf, try to measure the TCP bandwidth or the delay between two hosts in your network, change the receive window size and see what happens to TCP throughput. So these are all things you can try out to understand the concept of TCP congestion control and flow control better. Thank you all. That is all I have for this lecture. Let us continue our discussion in the next lecture.