Video 1
Hi and welcome to Computer Networking lesson 3. We are going to discuss transport layer today. Our goal is to understand the principles behind the transport layer. We are going to look at things like multiplexing and demultiplexing. We are going to discuss reliable data transfer flow control congestion control and finally we are going to learn about two core transport protocols on the Internet UDP and TCP. We may also cover TCP congestion control at the end of the lecture.

Video 2
Let's take a look at some of the transport layer services. Transport services and protocols provide logical communications between the application processes running on different hosts. Transport protocol runs on end systems. On the send side we take the messages that come from the application we break them into different parts called segments and we send them through the network. On the receiving side the transport layer takes those messages reassembles them and then pushes them up to the application. There are more than one transport protocols available but for the Internet purposes we will be discussing TCP and UDP.

Video 3
As we discussed before there are two types of transport layer protocols. TCP – Transport Control Protocol it is a reliable in order delivery. It has congestion control flow control and it uses connection set up just like when we meet somebody we would say "Hi how are you?" TCP does the same thing. UDP is unreliable unordered delivery no frills protocol but it is really really really fast. Services are not guaranteed for bandwidth or delays.

Video 4
Let's take a closer look at multiplexing and demultiplexing. Multiplexing at the sender means that we can handle data from multiple sockets add the transport layer header and then send it through the network. On the demultiplexing side the receiver is going to use that header information to deliver the received segments into the specific sockets that the application is using to listen for our transmission.

Video 5
So how does demultiplexing really work? Well hosts receive an IP datagram. Each IP datagram has the source IP address and the destination IP address. Each datagram carries one transport layer segment each segment has a source and a destination port number. The host uses IP address and a port number to direct segments to the appropriate socket where the application is listening on.

Video 6
So let's take a look at connectionless demuxing. When the hosts receives the UDP segment which is a connectionless protocol it is going to check the destination port number in that segment and it is going to direct the UDP segment to the socket with that port number. IP datagrams with the same destination port number for example on the same host but different source IP addresses and/or source port numbers will

be directed to the same socket at the destination. We will learn about DNS protocol for example that uses connectionless demultiplexing to connect.

Video 7
Let's take an example of how connectionless demultiplexing works. You could see two hosts on both side of the screen communicating with a server in the middle. It is probably DNS server. You could see how source port and destination port numbers change when the packets travel from one side to the other and back. You should notice that the socket that is used on the server in the middle is the same one where the packets are being forwarded to. Can you guess what the source and destination port numbers are on the right side of the screen communicating with the host with application before?

Video 8
TCP uses connection-oriented demux. TCP socket is identified by four parameters. Source IP source port destination IP and destination port. It is needed so we can identify multiple connections coming from the same host to the server or even multiple connections from multiple hosts to the same server. We should remember that web servers have different sockets for each connecting client. And non-persistent HTTP will have different sockets for each request.

Video 9
And here is an example of connection-oriented demux. You could see how we are using all four parameters source IP destination IP source port and destination port to define each individual segment connection. Imagine you are using two browsers and the same computer - Firefox and Chrome. That's exactly what is going to happen because your destination IP address is the same your destination port number port 80 is the same. Your source IP address is the same because you are using the same computer so the only way that connection parameter is going to differ and the system is going to know whether to use Firefox or Chrome is that original source port number that is going to be different for Firefox and for Chrome. Well let's take a look at the connection-oriented example of how demultiplexing works. Take a look at the host C. You could see two different applications working on a same laptop communicating with a server in the middle. It could be your laptop using two different browsers – Chrome and Firefox going to Yahoo.com. You could see that the destination port number is the same destination socket is the same port 80 our source IP address is the same because we are using the same laptop. So the only differentiating factor is that application socket that is on an originating host that was created by two different programs. That's how it finds its way back so and that's how it knows to send the information back to Chrome or to the Firefox. You could see how there are three different segments all destined to the same IP address on host B with the destination port 80 used demultiplexing to different sockets in an originating host.

Video 10
User Datagram Protocol also known as a connectionless transport. There is no handshaking between the receiver and the sender. Each UDP segment gets handled independently. UDP is typically used for streaming multimedia applications that are loss tolerant you could lose a few segments and still listen to the music right. It is used for DNS as we said before it is best effort service. The segments can be lost or delivered out of order but in exchange we get great and speedy delivery.

Video 11
Here is an example of the UDP segment header. 32 bits source port number destination port number we defined the length in bits of the UDP segment that includes the header. And the question becomes is why do we use UDP? There is no connection establishment which can add the delay it is simple and there is no connection control which means that UDP can blast as fast as possible. And we see this in services like Youtube Spotify or other multimedia applications you can use.

Video 12
We are now going to take a look at the principles of the reliable data transfer. It is in fact one of the top 10 most important networking protocols. Can you guess why? Primarily it is because we want to ensure that for certain applications there is a guaranteed delivery of our packets.

Video 13
Some of our applications such as SNTP for email or HTTP for web require reliable channel to move the packets from source to destination. But we know that our networks are inherently not reliable. So we have to build mechanisms inside our protocols to ensure reliability and delivery of the information from source to destination.

Video 14
What we are going to do is to actually deliver hypothetically reliable data transfer protocol. We are going to take information from our source host and see how we can ensure reliable delivery through the network over the channel that is inherently not reliable.

Video 15
We are going to look at certain computing functions. Reliable data send unreliable data send reliable data receive and the delivery of data. Those on the top are the ones that are called by application or to the application. The functions at the bottom are the ones that are used by the transport layer.

Video 16
There is a dependency between the event and the state. So what we are going to do is to define which event causes the transition from one state to the next.

Video 17
Our first example is RDT 1.0. Reliable transfer over reliable channel. Never happens in the real world but nevertheless exists hypothetically. The underlying channel is perfectly reliable in this case there are no bit errors there are no loss of packets and we just take the data from one site send it to another site there are no controls and it gets delivered reliably and in order.

Video 18
Let's take a look at RDT 2.0. Channel with bit errors. How do we get bit errors? Well there are things like electromagnetic induction EMI. Which may affect our transmission the bits the zeroes and ones may get flipped. We have to be able to detect them and recover from those errors. The question is how do we humans recover from errors? Usually we start the conversation we ask "Could you please repeat it again?" We go back and forth until we are clear. Let's take a look at how it is done in Computer Networking. RDT 2.0 is channel with bit errors. The underlying channel can actually flip bits in transmission sometimes through EMI or other natural forces. The question is how do we recover from those errors? Well we as humans provide feedback we ask the question "Can you please repeat it?" "Can you please say different words?" in Computer Networking RDT 2.0 we introduce two new concepts ACKs and NACKs – acknowledgements and negative acknowledgments. ACK means that the receiver explicitly tells the sender that the packet was received intact. NACK means that the receiver explicitly tells the sender that the packet was received with errors and in that case the sender is going to have to retransmit the packet.

Video 19
But RDT 2.0 has a fatal flaw. The NACKs and ACKs themselves can also get corrupted and because the computers don't see each other like humans do they don't know whether or not the ACKs and the NACKs were corrupted or were intact themselves. We also need to figure out a way how to handle duplicates when the sender retransmits the current packet if in fact ACK or NACK was corrupted. The sender needs to add the sequence numbers to each packet and then the receiver has to be able to identify and discard duplicates. We also have to remember that it is what is called a stop and wait operation. Where the sender sends one packet at a time and then waits for the receiver to respond either with an ACK or an NACK.

Video 20
We are now going to improve our 2.0 protocol with a 2.1 modification. We are going to add sequencing to each packet. For now we are going to use 1 and 0 as sequence numbers and they are going to suffice. You may ask yourself why well because it is a stop and wait operation so we have to send and receive information and know what is happening on both sides of the network. The receiver is going to check if the packet is duplicate and if it is it is going to indicate whether or not it already received it and if it is in fact a duplicate information it is going to junk it. Receiver cannot know however if the last ACK or NACK was received ok at the sender.

Video 21
Let's continue modifying our reliable data transfer protocol. The next modification 2.2 is a NACK free protocol. It has the same functionality as 2.1 but we are now only going to be using acknowledgements. So instead of a NACK the receiver is going to send an ACK for last successfully received packet. Let's look at the example if we send a series of packets 1 2 3 4 we send a packet 1 receiver is going to send acknowledgement 1. We send packet 2 receiver is going to send acknowledgement 2. If packet 3 arrived corrupted in 2.1 we would have sent a NACK in 2.2 we are going to resend acknowledgement 2 telling the sender that the last successfully received packet was packet 2. That would be the indication for the sender to resend packet 3.

Video 22
In our last modification of the reliable data transfer protocol is the modification 3.0. Which means that we now understand that the channel can have errors and loss. Well we now know how to deal with errors right? Through acknowledgements and duplicate acknowledgements what do we do in terms of loss? How do we know if the packet was lost? Well with 3.0 we are going to introduce a timer where a sender is going to wait a reasonable amount of time and then assume that the packet was lost and resend it again.

Video 23
So 3.0 in action. Example A – no loss very easy send packet receive packet. And you could see how acknowledgement numbers change. If in fact we experience a loss of packet as it is shown in example B the timer is going to tell the sender we never got the acknowledgement back let's resend the packet. In a couple more examples what happens if the packet was delivered successfully but the acknowledgement was lost? Well we already know how to handle duplicates through acknowledgements. So if the timer times out the packet was received successfully by the receiver but the acknowledgment was lost the timer is going to trigger the send to resend the packet and yet once it is received on the receiver side we are going to know it is a duplicate discard it and continue our operation. And finally in example D we have something called delayed ACK where everything is going successfully but there was traffic on the road and our response our acknowledgement was delayed. Well in that case the time out is going to trigger an action on the sender to resend the packet but on the receiver side we are going to receive the duplicate we already know how to deal with this issue same thing is going to happen on the sender once the packet acknowledgement successfully received and the system is going to come in balance and the operation is going to resume once both sides know that this transmission was successful.

Video 24
RDT 3.0 is in fact reliable data transfer protocol we know how to handle packet loss sequencing duplicates but the performance is very very bad. If we look at the fraction of time when the sender is transmitting information is very low compared to the throughput of the network. Why is that? Well primarily it is because of the

stop and wait operation. We have to wait until the whole packet arrives at the receiver side send the acknowledgement receive the acknowledgement and then and only then we can start transmitting the second packet. We need to do something better to increase the utilization of the network.

Video 25
In order to improve the performance of our RDT protocols we are going to introduce the concept called pipelining. Which means that the sender is going to allow multiple inflight but yet to be acknowledged packets in the transmission. Think about it as stuffing the line with information without waiting for each individual packet to be acknowledged. There are two generic forms of pipeline protocols one is called go back N and another one that is called select and repeat.

Video 26
Here is an example of how pipelining actually increases utilization if we send three packets at a time without waiting for acknowledgments it is going to increase the utilization of the line by a factor of three.

Video 27
So if we look at the pipeline protocols a little bit more closer we could see that go back N means that the receiver is going to send back a cumulative acknowledgement for all of the packets that were successfully received. For example if we have a series of packets that went out one through ten and the first six were received correctly and the last four were not the receiver is going to send a cumulative acknowledgement six for the first six successfully received packets. The sender then will know to resend the other four. Select and repeat means that the receiver will individually acknowledge packets that were not successfully received and then the sender is going to know which individual packets to resend. For example out of the series of ten packets if packet two and packet six were not received correctly the sender is going to know that only two and those two would need to be resent.

Video 28
Here is a more detail example of a go back N pipeline protocol on a sender's side. You could see these different packets that are marked with different colors the green ones are the ones that are already ACK'd. The yellow ones are the ones that were sent into the pipe waiting to be acknowledged. The blue ones are the ones that are usable waiting to be sent into the network and then the grey ones are not usable at all. As we receive acknowledgements for the packets that are successfully received the window is going to move and the yellow packets are going to be marked green and the blue packets are going to become yellow and be prepared to be sent down to the network.

Video 29
Go back N in action goes exactly like this if out of the four packets that were sent out the third one was lost the time out is going to indicate it on the sender side and then once it happens the sender is going to receive packets two three four and five back

to the receiver. With the first two successfully acknowledged and already waiting on the receiver side to be processed.

Video 30

On the select and repeat side the receiver is going to individually acknowledge all correct received packets. It is going to buffer the packets as needed for eventual in order delivery to the application. The sender then is only going to resend the packets for which acknowledgements were never received and that's how it knows which packets it has to resend to the receiver. We are going to maintain the sender window which is going to keep intact those packets that we are keeping track of. Similar to go back N protocol here is an example of a sender and receiver windows that use select and repeat. You could see how our yellows and greens are a little bit different but nevertheless we are keeping track of those packets that we have successfully received acknowledgements from versus those that we sent and still waiting for. On a receiver side we are marking the packets that we are successfully receiving versus the ones we are waiting to receive from the sender and that operation allows us to actually wait and fill in the gaps individually rather than resending the whole chain of packets back to the receiver.

Video 31

And here is an example of the select and repeat in action. We are going to send a series of packets in a pipeline fashion down to the network. On the receiver side we are going to receive packet zero send acknowledgement zero. Receive packet one send acknowledgement one. Receive packet three buffer it send acknowledgement three right. So now we are detecting the gap in a network. On the sender side once the timer times out on a packet two it is going to resend that packet receiver is going to receive it put it in order with other packets and then push it up to the application for successful delivery.

Video 32

Now that we've learned about reliable data transfer protocol and pipelining let's take a look at an actual TCP and how it is implemented. It is point to point meaning there is only one sender and one receiver it is reliable in order byte stream means that all of the packets have to be sent and received eventually by the application in the same order that they left. It is pipelined in TCP it uses congestion control and flow control to manage the information that goes through the network. It is full duplex meaning that the data and the control signals go bidirectionally. It is connected oriented we have to agree on a connection parameters before we start transmitting "Hi how are you?" "What language do you speak?" "English" "Let's continue". And finally it is flow controlled meaning that the sender is never going to overwhelm the receiver with more information than the receiver can handle. And this is what the TCP segment structure actually looks like you could see everything we've talked about up until now. There is a place for source port number and the destination port number there is a place for sequence number there is a place for acknowledgement number checksum receive window and finally a place to carry our payload or the actual data.

Video 33
You are actually going to be surprised to learn that there is a dependency between sequence numbers and acknowledgement numbers of the TCP packets. The sequence numbers are defined as byte stream number of a first byte in the segment's data. The acknowledgements are calculated as the sequence number of a next byte expected from the other side we are going to tell the sender what we are expecting from him to receive. Here is an example of the dependency between sequence numbers and acknowledgements. Host A is going to send a letter C through the network to host B host B is going to take that letter move it into the capital form and send it back. That's a simple Telnet scenario. Take a look at the host A where it sends the packet with a sequence number 42 acknowledgement 79 and the data is the actual letter C that's the payload that we are carrying. Host B is going to receive it acknowledge it convert the lower case C to uppercase C and send it back. The sequence number now becomes 79 acknowledgement number becomes 43. You could see how these things change as both hosts telling each other what they are expecting in terms of the information coming from them the acknowledgement numbers and the sequence numbers.

Video 34
One important concept to remember is the time outs. We don't want them to be set too short or too long if they are set too short we are going to be sending a lot of information without ability to receive our acknowledgements back. If they are too long we are going to slow down our network so we use the concept of RTT which we discussed before to develop the time out timing that needs to be set for all of our TCP transmissions. We are going to sample our network and combine the parameters of estimated RTT and the sample RTT to define the actual time out that every TCP packet is going to be set to before it receives the acknowledgement back.

Video 35
And here are the few examples of the TCP retransmission scenarios. What actually happens in a packet? Well let's look at the scenario number one lost acknowledgement. We send the packet through it has a sequence number 92 and the 8 bytes of data. The acknowledgement number 100 was lost on a time out the sender is going to resend that packet again and then hopefully receive the acknowledgement back. In another example of a TCP retransmission on a premature time out the sender never received the acknowledgments for successfully delivered packets because they were delayed in transmission. So it had to resend the original packet sequence number 92 with 8 bytes of data however once the acknowledgements were received later it realized that the packets were received successfully because the last acknowledgment for successful received packet was at 120 which indicated both packets were successfully received. And finally the great example about cumulative ACK as you could see on the slide the original acknowledgement for the first packet was lost but because the second one went through the sender knows that both packets were received successfully. Why? Because we are using cumulative acknowledgements the last acknowledgement for

the sequence number 100 with 20 bytes of data was received successfully and we know through pipelining protocols that we are acknowledging all the packets that were received successfully up to the last one which is an indication for the sender to continue moving forward. A question for you what would the next acknowledgement number will be?

Video 36
And now that we looked at the actions on the TCP sender side let's look at the TCP ACK generation on the receiver side. Let's look at the various events that actually happen for example if we have an arrival of in order with expected sequence number all data up to the expected sequence number was already acknowledged. What are we going to do? We are going to delay the ACK we are going to wait for up to 500 milliseconds for the next segment and if there is no next segment arrival we are going to send the acknowledgement back. If we have an arrival of in order segment with expected sequence and one other segment has an ACK pending we are going to immediately send the cumulative ACK back to the sender ACK in both in order segments. If we receive our order segments with higher than expected sequence numbers we are going to detect the gap we are going to immediately send duplicate ACK indicating the sequence number of the next expected byte. We are telling the sender what information we are waiting for them to send to us. And finally when we get the arrival segment that partially or completely fills in the gap we are going to immediately send the acknowledgement provided that the segment starts at the lower end gap put the entire transmission together and send it up to the application.

Video 37
TCP also has another feature called fast retransmit. Fast retransmit means that if the sender if the sender receives three acknowledgements for the same data which is called triple duplicate ACK is going to resend all of the unacked segments with the smallest sequence number to the receiver. And here is an example of something called fast retransmit. You could see on this picture that the second packet sequences 100 20 bytes of data was lost. The receiver wants to detect the loss send three acknowledgements 100 back to the sender when the sender receives them without waiting for the time out it realized that the second packet was lost and resend it.

Video 38
Let's take a look at the TCP flow control. Flow control means that the receiver controls the sender in such a way that the sender won't overflow the receiver's buffer information by transmitting too much too fast. The way that actually works is the receiver advertises free buffer space by including the rwnd value in the TCP header of the receiver to sender's segment. Sender limits the amount of unacked in-flight data to receiver's rwnd value. That guarantees that the receiver buffer will not get overflown.

Video 39
As humans we easily agree on a handshake by introducing each other. But in a network there are so many different variables that come into play that are network delays we don't see each other so a 2-way handshake is not enough to agree on how we are going to communicate. TCP actually employs 3-way handshake we do that by first listening on both sides of the connections we then transition from the listening state on a sender side that's our client sending a SYNSENT packet to the receiver to our server. When the server receives that it moves its state to SYNRECEIVED parameters that means that the server is now ready to start accepting the information. Once the client receives the first acknowledgement bit it moves itself into the established position and then sends the acknowledgements along with the data to the server side. Once server receives that it itself moves to the established state and that's how the TCP 3-way handshake is done and now the connection is established.

Video 40
Similar to the 3-way handshake TCP also has to go through specific process for closing the connection. Client and server send each other information final bits or also known FIN bits which create a state announcing to everyone that now we are going to be closing the particular connection. Client which initiates the connection will also send the FIN bit to the server that indicates that the client no longer has any data to send but can continue to receive data from the server. Once the server gets that information it is going to initiate its own closing process it is going to send last acknowledgment packet back to the client the client once received it is going to move through a couple of different states eventually the connection is going to time out on the client side and it is going to get closed. Same thing is going to happen on the server side when after the last ACK the server is going to time out and also move to the established finished closed state.

Video 41
Now we are going to discuss congestion control which is a top 10 problem of the networking world. Informally it means that there is too many sources sending too much information all at once for network to handle. If you ever been stuck on traffic that's exactly what it is. Let's take a look at the congestion scenario where we have four senders multiple paths and then let's see what happens if and when all of them start transmitting at the same time. We have host A that's the red line that transmits the information to host C if host A continuously transmits it is going to effectively block all of the packets coming from host D to host B. The upper blue line is going to get blocked as we learned in this lesson once host D stops receiving the acknowledgements for the packets that it sent out it is going to time out and retransmit the packets causing further congestion. And that's what the congestion scenario looks like the network itself is going to keep retransmitting the packets and all of the hosts are going to get blocked. Another cost of congestion is that when the packets are dropped any upstream transmission capacity is going to become wasted the line will basically sit idle and unused.

Video 42

Video 43
So how do we prevent congestion from happening? TCP actually has a built-in mechanism to do that it is called additive increase multiplicative decrease. The approach is that the sender is going to keep increasing transmission rate the window size probing for usable bandwidth until loss occurs. We are going to increase by 1 MSS every RTT until loss occurs and then we are going to cut down the congestion window in half when that happens. Another part of TCP congestion control is something called TCP slow start. The way this works is when the connection begins we increase the rate exponentially until the first loss event occurs we start with a single segment double it to two then four eight and so forth and so on. Initial rate is slow but it ramps up exponentially and very very fast.

Video 44
Well so this was a really hard lesson we looked at the principles behind the transport layer services. We talked about muxing and demuxing reliable data transfer flow control congestion control we also discussed TCP and UDP. We are now going to be leaving the network edge and going right into the core. See you next time.