

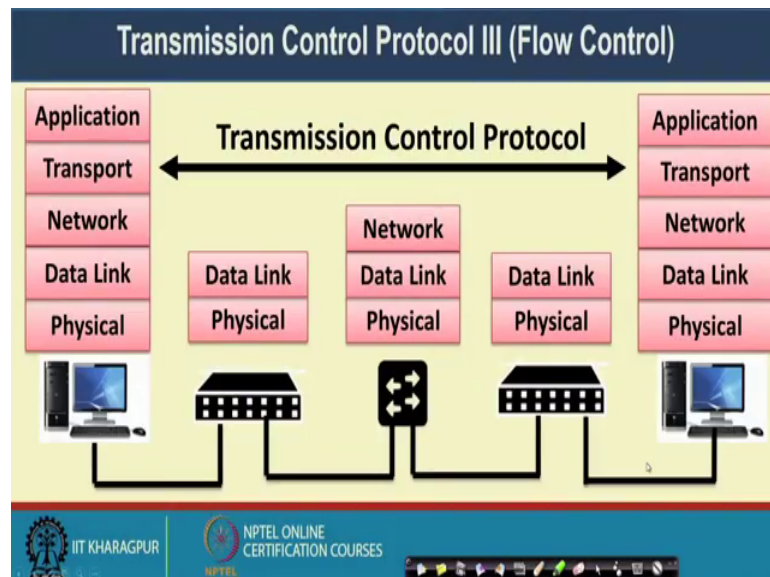
Computer Networks and Internet Protocol
Prof. Sandip Chakraborty
Department of Computer Science and Engineering.
Indian Institute of Technology, Kharagpur

Lecture -21

Transmission Control Protocol - III (Flow Control)

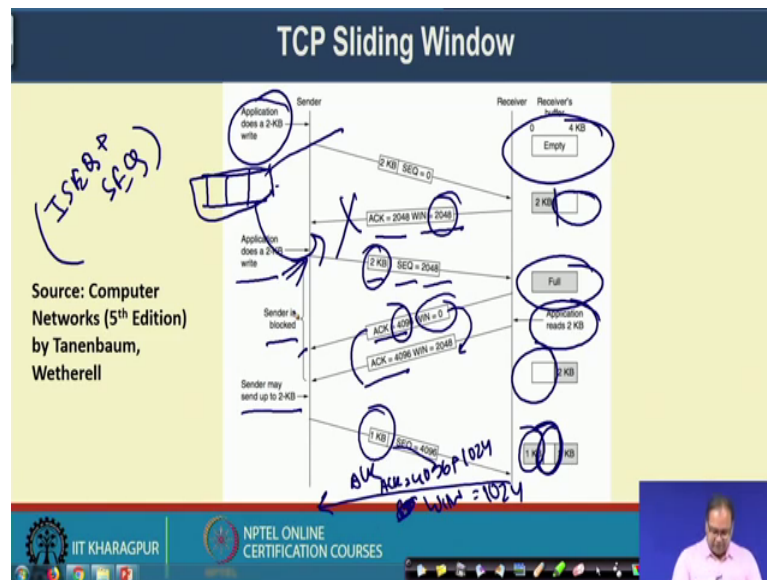
Welcome back to the course on Computer Network and Internet Protocol. So, in the last class, we have looked into the details of TCP connection establishment.

(Refer Slide Time: 00:25)



Now in this particular class, we look into the further details of TCP; the flow control algorithm, which is used by TCP to manage the rates of sender rates at sender's eye. And the different timers associated to it this flow control algorithm and how you set a proper value of those timers.

(Refer Slide Time: 00:48)



Well starting with the flow control algorithm. So, TCP uses a sliding window flow control algorithm with this go back in principle go back in ARQ principle. Where, if there is a time out, then you retransmit all the data which is there, inside your sender window.

So, the idea is something like this in a very simplified notion, that you say start to its subsequent number 0. So, remember that 0 is not the initial sequence number rather here just for explanation we are utilizing this sequence number as 0. But ideally it will start from the initial sequence number that has been established during the hand shaking phase of the connection establishment.

So, here the sequence number is like, if it is sequence number so, the initial sequence which is being established for last the sequence number that we are talking about here. So, just for simplicity of explanation we are starting with sequence number 0. So, let us start with sequence number 0 and at this stage the application does a 2 kB write at the transport layer buffer.

When the application does a 2 kB write a the transport layer buffer, so, you send 2 kB of data and you are sending a 2 kB of data with sequence number 0. So, initially this is the receiver buffer. So, the receiver buffer can hold up to 4 kB of data. So, once you are getting that so, the receiver buffer it says you that well; it has to receive 2 kB of data. So, it has only 2 kB of data which it can accept. So, it sends back with an acknowledgement

number of 2048 2 kB is equivalent to 2048 bytes so, because we are using byte sequence number. So, it sends an acknowledgement to it 2048 and window size has 2048 so, it can hold 2 kB of more data.

So, at this stage, that then again application does a 2 kB upload. So, when the application does a 2 kB upload you are sending 2 kB of data further data along with the sequence number starting from 2048. So, it is received by the receiver. So once it is received by the receiver. So, here because you have already sent 2 kB of data and the earlier advertised window size was 2048.

So, the sender is blocked from this point, because the sender has already utilized the entire buffer space that the receiver can hold, the sender cannot send anymore data. So, the same that is blocked at this stage so, the receiver buffer becomes full after receiving this 2 kB of data. So, the receiver sends an acknowledgement saying that it has received up to 4096 bytes and the window size is 0. So, it is not able to receive any more data so, the same that is blocked at this point.

Now, at this stage the application reads 2 kB of data from the buffer. Once the application reads 2 kB of data from the buffer, so, it has this it has read this fast 2 kB. So, again this fast 2 kB becomes full. So, when the fast 2 kB becomes full, the receiver sends an again an acknowledgement that well the acknowledgement number was 4096 the 1 which was there earlier, but the window size now changes from 0 to 2048. So, it can get 2 kB of more data.

So, once it the sender receives back sender comes out of the blocking stage and once the sender is coming out of the blocking stage. So, the sender may send up to 2 kB of more data. So, at this stage say the sender is sending 1 kB of data with sequence number 4096. So, that 1 kB is received by the receiver it put it in the buffer and it has 1 kB of free. So, if the receiver wants to send an acknowledgement, in that acknowledgement number it will use the sequence acknowledgement number as 4096 plus 1024 that it has received.

And the sequence and a this window size has window size has 1 kB so, window size at 1024 So, that acknowledgement it will send back to the sender.

So, that way the entire procedure goes on and whenever sender is sending some data that at this stage the sender this has send some data of 2 kB of data. Then in the sender buffer

that 2 kB of data is still there until it receives 10 acknowledgement. If this acknowledgement is lost, then based on that go back in principle it will retransmit this entire 2 kB of data which was there in the sender buffer ok.

(Refer Slide Time: 05:17)

Delayed Acknowledgements

- Consider a telnet connection, that reacts on every keystroke.
- In the worst case, whenever a character arrives at the sending TCP entity, TCP creates a 21 byte TCP segment, 20 bytes of header and 1 byte of data. For this segment, another ACK and window update is sent when the application reads that 1 byte. This results in a huge wastage of bandwidth.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, the algorithm is pretty simple considering this sliding window protocol we go back in principle. But there are certain tricks in it. Let us look into those tricks. First of all consider an application called telnet, I am not sure how many of you have used telnet. So, telnet is an application to make a remote connection to a server. So, with this telnet application you can make a remote server remote connection to a server and then execute the comments on top of that.

So, whenever you are making this remote connection to a server and executing the comments on that, say you have just written `ls` the liners comment `ls` to listing all the directives which are there in the current folder. So, that `ls` command need to be send to the server site over the network because, that is remote connection using telnet that you have done.

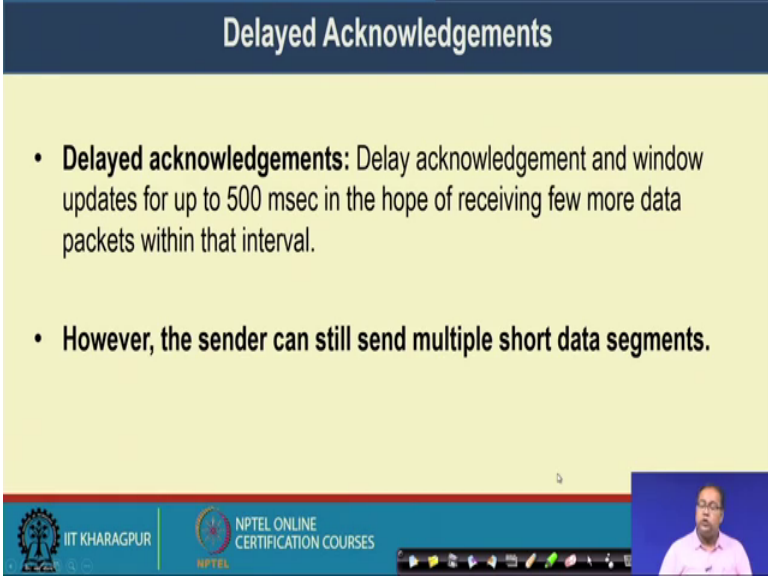
So, this telnet connection it reacts on every such keystroke, in the worst case that it may happen that whenever a character arrives at the sending TCP entity. TCP it creates a 21 byte of TCP segment, where 20 byte is there in the header and 1 byte of data. TCP segment header is 20 byte of long, but telnet is sending the data to the server byte by

byte. So, telnet application at the client side it has just received 1 byte of data and that 1 byte of data it is trying to send with the help of a TCP segment.

So, in that TCP segment what will happen, that the TCP segment side will contain 20 byte of the header and only 1 byte of data. So, you can just think of that what is the amount of overwrite you have. So, with that 21 byte of packet or data going to 1 byte of segment, you are sending only 1 byte of data. And for this segment another ACK and window update is sent when the application reads that 1 byte.

So, the application reads that 1 byte and application sends back an acknowledgement. So, these results in a huge wastage of bandwidth, just you are not sending any important data to the server side rather you are sending very small amount of data and the huge amount of resources utilized because of the headers.

(Refer Slide Time: 07:28)



Delayed Acknowledgements

- **Delayed acknowledgements:** Delay acknowledgement and window updates for up to 500 msec in the hope of receiving few more data packets within that interval.
- **However, the sender can still send multiple short data segments.**

The slide is part of an NPTEL presentation from IIT Kharagpur. It features a dark blue header with the title 'Delayed Acknowledgements' in white. The main content area is light yellow with two bullet points. At the bottom, there is a blue footer with logos for IIT Kharagpur and NPTEL, and a small video inset of a speaker on the right.

So, to solve this problem, we use the concept called delayed acknowledgement. So, in case of delayed acknowledgement, you delay the acknowledgement and window updates for up to some duration 500 millisecond in the hope of receiving few more data packets within that interval. So, it says that well whenever you are getting a character from the telnet application, you do not send it immediately. Rather you wait for certain amount of duration that is the 500 millisecond by default in TCP. And your hope is that by that time you may get some more data and you will be able to send a packet where with 20 kilobyte of sorry 20 byte of header you will have more than 1 byte of data.

However, in this case, the sender can still send multiple short data segments because, if the sender wants. So, it is just like that whenever you are sending the acknowledgement to the sender, you are delaying the acknowledgement.

You are delaying the acknowledgement; that means; you are not sending any immediate acknowledgement. And a sender to remember that, a sender unless it gets an acknowledgement with the available buffer space, the sender will not send anymore of data. So, the receiver just keep on waiting that, whenever it will get sufficient data from the sender it will have sufficient space at the receiver, that then only it will send back that acknowledgement to the sender. So, the receiver will not send immediate acknowledgement to the sender to prevent the sender to send further data to the receiver.

(Refer Slide Time: 09:01)

Nagle's Algorithm

- When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged.
- Then send all buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
 - Only one short packet can be outstanding at any time.

The diagram illustrates the algorithm with a 'Sender' and a 'Receiver'. The Sender has a buffer containing three small data pieces labeled (1), (2), and (3). An arrow shows the first piece (1) being sent to the Receiver. The Receiver has received (1) and is sending back an acknowledgement. The Sender's buffer now contains (2) and (3), which are being sent together as a single larger packet to the Receiver. The Receiver has received (2) and (3) and is sending back another acknowledgement. The footer of the slide includes the IIT Kharagpur logo and the text 'NPTEL ONLINE CERTIFICATION COURSES'.

Well now, we have another algorithm. So, in the earlier case what we have seen that well with the delayed acknowledgement, you are expecting that unless you are sending an acknowledgement to the sender, the sender will not send any further data. But sender is not restricted to that sender is that whenever it will get data from the telnet application it will immediately send the data.

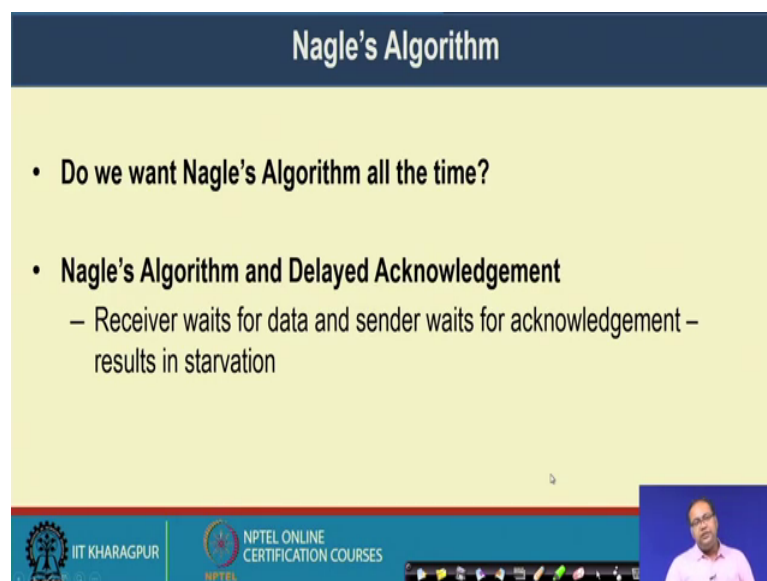
Now, to prevent sender for sending this kind of small packets or small segments, we use the concept of Nagle's algorithm. What is this? The Nagle's algorithm tells that, when the data come into the sender in small pieces just send the first piece and buffer all the rest until the first piece of acknowledgement. So, it is just like that, you have received a

small data segment or single bytes you have received byte A, you send that byte A from the sender say this is the sender and this is your receiver.

And you keep on buffering all the subsequent characters A B C D until you get the acknowledgement from the sender. So, the hope here is that whenever you are sending some short packet in the internet, you are not sending multiple short packets one of after another. That means, you are not sending a packet A packet B B packet C like segment A segment B segment C over the network rather only one short packet will be outstanding in the network at any given instance of time.

So, that way by the time you will get the acknowledgement for this packet your expectation is that, you will get multiple other characters in the sender buffer. Whenever you are getting multiple other buffer characters in the sender buffer, you can combine them together construct a single segment and send it over the network.

(Refer Slide Time: 10:57)



The slide is titled "Nagle's Algorithm" in a dark blue header. The main content area is yellow and contains two bullet points. The first bullet point asks "Do we want Nagle's Algorithm all the time?". The second bullet point is "Nagle's Algorithm and Delayed Acknowledgement", which includes a sub-point: "Receiver waits for data and sender waits for acknowledgement – results in starvation". In the bottom right corner, there is a small video inset showing a man in a pink shirt. The bottom of the slide features a blue footer with the IIT Kharagpur logo, the text "NPTEL ONLINE CERTIFICATION COURSES", and a navigation bar with various icons.

Nagle's Algorithm

- Do we want Nagle's Algorithm all the time?
- Nagle's Algorithm and Delayed Acknowledgement
 - Receiver waits for data and sender waits for acknowledgement – results in starvation

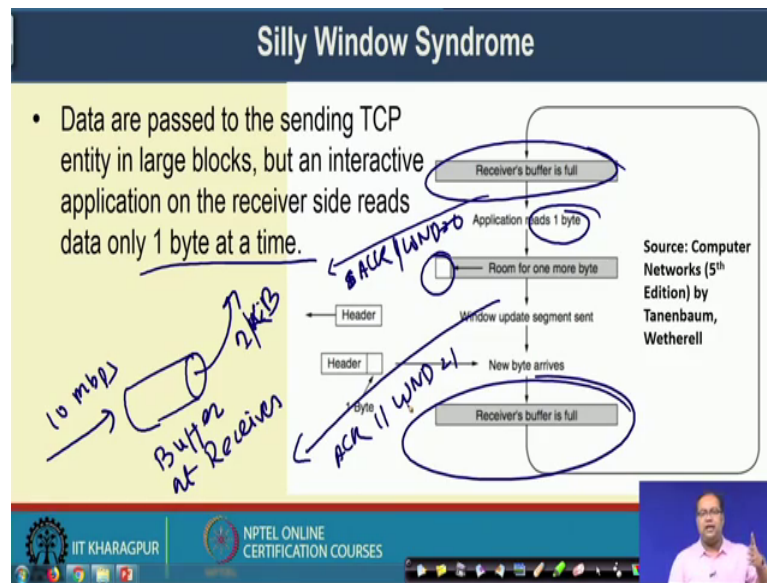
The question comes here that we want to use Nagle's algorithm all the time? Because Nagle's algorithm intentionally increasing the delay in transfer. So, if you are just using telnet's application and applying Nagle's algorithm, your response time for the application will be slow. Because although you are trying something that TCP is preventing that single byte to reach at the server site unless it is getting an acknowledgement for the previous short packet.

And that is why do not want to use Nagle's algorithm for less sensitive application. And there is another interesting observation here that, if you implement Nagle's algorithm and delayed acknowledgement altogether, what may happen? That the in the Nagle's algorithm the sender is waiting for the acknowledgement. The sender has sent one small packet or a small segment and the sender is waiting for the acknowledgement, but the receiver is delaying that acknowledgement. Now if the receiver is delaying the acknowledgement and the sender is waiting for that acknowledgement. So, the sender may go to starvation and you may have a significant amount or considerable amount of delay in getting the response from the application. So, that is why if you are implementing Nagle's algorithm and delay the acknowledgement altogether it may result in a scenario, where you may experiences low response time from the application because of the starvation.

So, in broad sense, the delayed acknowledgement what you are doing? You are presenting the receiver to sending small window updates. And you are delaying this acknowledgement at the receiver side with the expectation that the sender will accumulate some more data at the sender buffer. And it will be able to send the large segment rather than a small segment.

Whereas, in case of Nagle's algorithm you are just waiting for the acknowledgement of a small segment with the expectation that by that time the application will write few more data to the sender buffer and these two together can cost a starvation. So, that is why we do not want to implement delayed acknowledgement and Nagle's algorithm altogether.

(Refer Slide Time: 13:14)



So, one possible solution comes from, another problem in this window update message, which we will call as the silly window syndrome. So, let us see that what is silly window syndrome? So, it is like that data are passed to the sending TCP entity in large block, but an interactive application under receiver side reads data only one byte at a time. So, it is just like that, if you look into the receiver side, the receiver this is the receiver buffer say, this is the receiver buffer.

So, the sending application is sending data at a rate of 10 mbps say, the sender has lots of data to send, but you are running some kind of interactive application at the receiver side. So, it is receiving data at a very slow rate like; at a rate of 1 kB at a time or 1 byte at a time the example that is given here at 1 byte at a time.

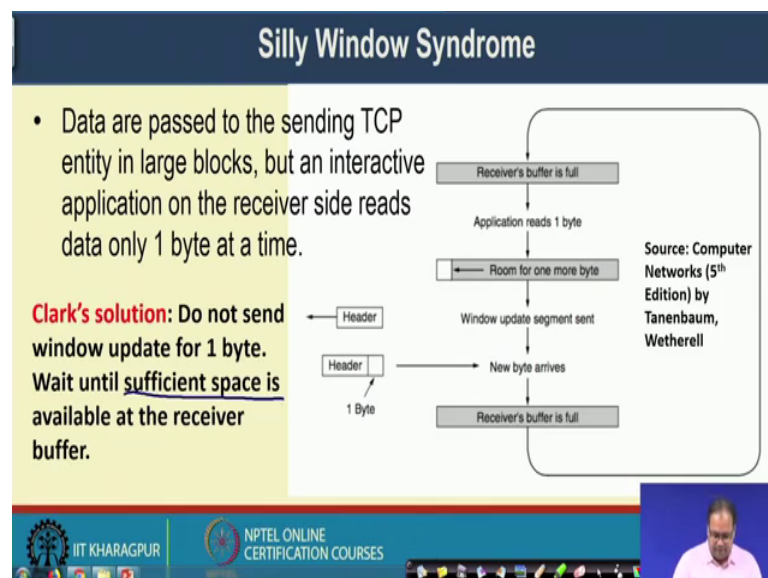
Now, if it happens, so, this is the kind of problem. Initially, say the receiver buffer is full when the receiver buffer is full you are sending an acknowledgement to the sender saying that the acknowledgement the corresponding acknowledgement number followed by the window value as 0. So, the sender is blocked here, now the application needs 1 byte of data. The moment application needs 1 byte of data; you have a free space here in the buffer. Now say, the receiver is sending another acknowledgement to the sender saying that the window size is 1.

So, if it sends this window size small window size advertisement to the sender what the sender will do? Sender will send only 1 byte of data. And once it sends 1 byte of data

with that 1 byte of data again the receiver buffer becomes full. So, this becomes in a loop and because of this window update message of 1 byte, the sender is tempted to send 1 byte of segment with every window update message. So, this again creates the same problem that we were discussing earlier that you are sending multiple small segments one after another.

And we do not want to send those multiple small segments, because it has such significant overhead from the network perspective. It conceives a huge amount of bandwidth without transferring any meaningful data to the receiver intake.

(Refer Slide Time: 15:43)



So, to solve this problem, we have a solution which is proposed by Clark, we call it as a Clark solution. So, the Clark solution says that do not send window update for 1 byte, you wait for sufficient space is available at the receiver buffer. Once some sufficient space is available at the receiver buffer then only you send the window update message.

Now, the question comes that what is the definition of the sufficient space. That depends on the TCP implementation that if you are using some buffer space, then you use certain percentage of the buffer space. If that is become available then only you send the window update message to the sender.

(Refer Slide Time: 16:23)

The slide is titled "Handling Short Segments – Sender and Receiver Together". It contains the following bullet points:

- Nagle's algorithm and Clark's solution to silly window syndrome are **complementary**
- **Nagle's algorithm:** Solve the problem caused by the sending application delivering data to TCP a byte at a time
- **Clark's solution:** Receiving application fetching the data up from TCP a byte at a time
- Exception: The PSH flag is used to inform the sender to create a segment immediately without waiting for more data

The slide footer includes the IIT KHARAGPUR logo, the NPTEL ONLINE CERTIFICATION COURSES logo, and a small video feed of a man in a pink shirt.

Well, here the interesting fact is that to hand glass handle the short segments at the sender and receiver altogether. That this Nagle's algorithm and the Clark's solution to see the window syndrome. They are complementary, just like the earlier case like the Nagle's algorithm and the delayed acknowledgement can create a starvation that will not happen here.

So, the Nagle's algorithm it solves the problem caused by the sending application delivering data to TCP 1 byte at a time. So, the sending it prevents the sending application to send small segments. Whereas, the Clark solution, here it prevents the receiving application for sending window update of 1 byte at a time. So, the receiver is receiving application fetching the data from the TCP layer 1 byte at a time for that you will not send immediate window update message.

There is certain exception to that because; whenever you are applying this Nagle's algorithm and the Clark solution. Again it will have some amount of delay on the application perspective. The application response time will be still little slow, because you are waiting for sufficient data to get accumulated and then only create a segment.

Similarly, on the receiver side you are waiting for sufficient data to read by the application and then only you will send the window update message, this may still have some higher response time from the application perspective, may not be as high as like a starvation which was there for Nagle's algorithm and delayed acknowledgement. But, for

certain applications say for some real time application, you want that the data is transferred immediately by pressing the Nagle's algorithm and the Clark solution; in that case in the TCP header you can set the PSH flag.

So, this PSH flag it will help you to send the data immediately, it will help make inform the sender to create a segment immediately, without waiting for more data from the application side. So, you can reduce the response time by utilizing the PSH flag.

(Refer Slide Time: 18:43)

The slide is titled "Handling Out of Order in TCP". It contains two bullet points:

- TCP buffers out of order segments and forward a duplicate acknowledgement to the sender.
- Acknowledgement in TCP – Cumulative acknowledgement

Below the text is a diagram illustrating the receiver buffer and cumulative acknowledgements. A horizontal bar represents the "Receiver Buffer". The buffer is divided into segments. The first segment is labeled "1024" and is shaded. The second segment is labeled "2048" and is also shaded. The third segment is labeled "2048" and is unshaded. A curved arrow labeled "Duplicate ACK & DUPACK" points from the buffer back to the sender. A straight arrow labeled "ACK (SEQ = 1024)" points from the buffer back to the sender. Another straight arrow labeled "ACK (SEQ = 1024)" points from the buffer back to the sender. The diagram shows that the receiver buffer contains segments starting from 1024, and the sender receives a duplicate acknowledgement for sequence number 1024.

The slide footer includes the IIT KHARAGPUR logo and the text "NPTEL ONLINE CERTIFICATION COURSES".

Well now, the second thing is that handling out of order segments in TCP. So, what TCP does? The TCP buffer space out of order segments and forward duplicate acknowledgement. So, this is an interesting part of the TCP this concept of duplicate acknowledgement. So, what TCP does that whenever you are receiving certain out of order segment say for example, I am just trying to draw a ye so, I am trying to say this is the receiver buffer. In the receiver buffer, we have received up to say this segment and the receiver is say this is say 1024. It has received up to 1023 and it is expecting from 1024 and you have received the segment from say 2048 to something else.

Now, at this case, whenever it has received this previous segment, it has sent an acknowledgement with sequence number as 1024; that means, the receiver is expecting and segment starting from byte 1024, but it has received this out of order segment. So, it will put the out of order segment in the buffer, but it will again send an

acknowledgement with this same sequence number, that it is still expecting sequence number 1024.

So, this acknowledgement we call it as a duplicate acknowledgement. So, this called a duplicate acknowledgement or in short form DUPACK. So, this DUPACK, we will inform the sender application that will ah; it has this particular receiver has not received the byte starting from 1024, but it has received certain other bytes after that.

So, this has an important consequence in the design of TCP congestion control algorithm. So, we look into the details of this consequence, when we discuss about the TCP congestion control algorithm in the next class.

(Refer Slide Time: 21:14)

Handling Out of Order in TCP

- Receiver has received bytes 0, 1, 2, 2, 4, 5, 6, 7
 - TCP sends a cumulative acknowledgement with ACK number 2, acknowledging everything up to byte 2
 - Once 4 is received, a duplicate ACK with ACK number 3 (next expected byte) is forwarded – **triggers congestion control**
 - After timeout, sender retransmits byte 3
 - Once byte 3 is received, it can send another cumulative ACK with ACK number 8 (next expected byte)

The slide includes logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES. A small video inset in the bottom right corner shows a man in a pink shirt speaking.

So, here is an example, say the receiver has received the bytes 0 1 2 and it has not received the bytes 3 and then it has received bytes 4 5 6 7. So, TCP sends a cumulative acknowledgement with acknowledgement number 2 which acknowledges everything up to byte 2.

So, once this four is received a duplicate ACK with acknowledgement number 3 that is the next expected byte it is forwarded. This triggers a congestion control algorithm which we look into the details in the next class, after time out the sender retransmits byte 3. So, whenever the sender is retransmitting byte 3 so, you have received byte 3 here.

So, the moment you have received byte 3 here, you have basically received all the bytes up to byte 7. So, you can send another cumulative acknowledgement with acknowledgement number 8; that means you have received everything up to 7 and now you are expecting byte 8 to receive ok.

(Refer Slide Time: 22:15)

TCP Timer Management

- **TCP Retransmission Timeout (RTO):** When a segment is sent, a retransmission timer is started
 - If the segment is acknowledged before the timer expires, the timer is stopped
 - If the timer expires before the acknowledgement comes, the segment is retransmitted
- What can be an ideal value of RTO ?
- Possible solution: Estimate RTT, and RTO is some positive multiples of RTT
- RTT estimation is difficult for transport layer – why?

The diagram shows a vertical line representing time. A horizontal arrow labeled 'seg' starts from the line and points to the right. A horizontal arrow labeled 'Timer' starts from the same point on the line and points to the right, ending at a point further to the right than the 'seg' arrow. This illustrates the timer running alongside the segment transmission.

The slide footer includes the IIT KHARAGPUR logo, NPTEL ONLINE CERTIFICATION COURSES text, and a small video inset of a speaker in the bottom right corner.

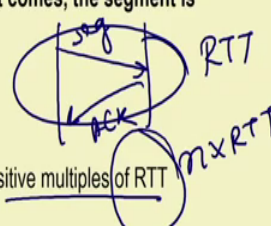
TCP has multiple timers implementation. So, let us look into those timers in detail. So, one important timer it is TCP retransmission timeout or TCP, we call it in short form TCP RTO. So, this retransmission timeout helps in the flow control algorithm. So, whenever as segment is sent, this retransmission timer is started if the segment is acknowledged so, if the segment is acknowledged before the timer expires the timer is stopped and if the timer expires before the acknowledgement comes, the segment is retransmitted. So, once you have transmitted a segment from the sender side you start the timer, say within this timeout if you receive the acknowledgement, then you restart the timer otherwise once timeout occurs, then you retransmit this segment.



So, timeout occurs means, something bad has happened in the network and simultaneously it also triggers the congestion control algorithm that we will discuss during the discussion of the congestion control algorithm, but it also retransmit the loss segment. So, if it does not receive the acknowledgement within the timeout, it assumes that the segment has lost.

(Refer Slide Time: 23:36)

TCP Timer Management

- **TCP Retransmission Timeout (RTO):** When a segment is sent, a retransmission timer is started
 - If the segment is acknowledged before the timer expires, the timer is stopped
 - If the timer expires before the acknowledgement comes, the segment is retransmitted
- What can be an ideal value of RTO ?
- Possible solution: Estimate RTT, and RTO is some positive multiples of RTT
- RTT estimation is difficult for transport layer – why?

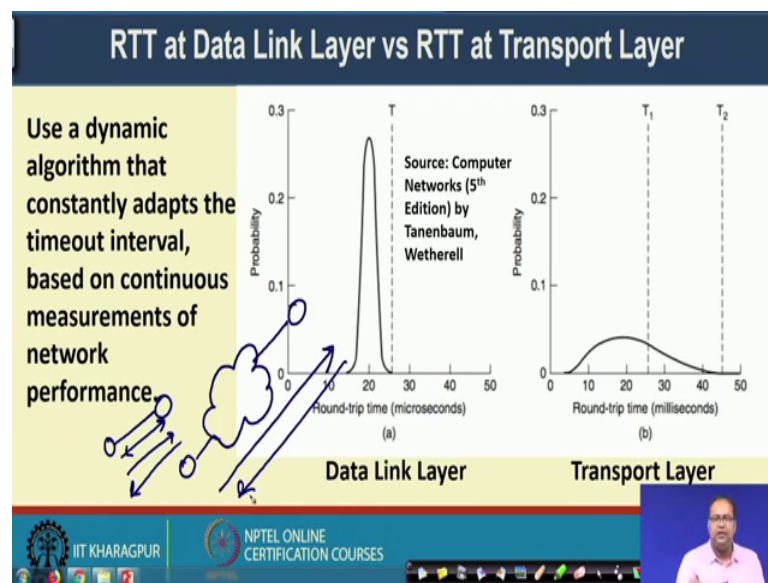


Now, the question comes that what can be an ideal value for this retransmit timeout. So, how will you say this retransmit timeout? So, one possible solution is that to estimate the round trip time because, you have sent a segment and you are waiting for the corresponding acknowledgement. So, ideally if everything is good in the network, then this segment transmission and the acknowledgement transmission it will take one round trip time.

So, it is one round trip time it is expected to get everything, but because of the network delay and something, you can think of that well I will say that the retransmission timeout to some positive multiples of RTT. Some n cross RTT where n can be 2, 3; something like that based on your design choice. But then the question comes that how you make an estimation of RTT? Because your network is really dynamic and this RTT estimation is a difficult for transport layer. So, let us see that why this difficult for transport layer.

(Refer Slide Time: 24:32)

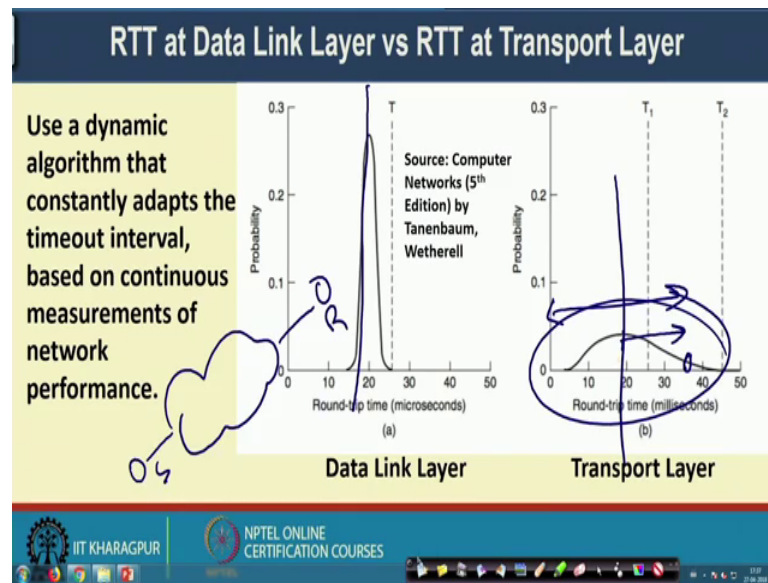


So, if we make a plot something like this so we are trying to plot the RTT that out trip time and the data link clear at the transport layer. So, the difference is that in case of data link clear you have two different nodes, which are directly connected via link. So, if these two different nodes are directly connected via link. So, how much time it will take to send the message and get back the reply.

But in case of your network layer, in between the two nodes you have this entire internet and then another node and then you are trying to estimate that, if you are sending a message to this end host and receiving back a reply what is the average round trip time it is taking.

Now, if we just plot this round trip time, the distribution of this round trip time, we will see that the variation is not very high whenever you are at the data link here because, it is just the single link and in that single link this dynamicity is very less because, the dynamicity is very less for a single link you can make a good estimation, if you take the average with that average we will give you a good estimation of that round trip time.

(Refer Slide Time: 25:37)



But that is not true for the transport layer, in case of transport layer because there are lots of variability in between this intermediate network between the sender and the receiver. So, your round trip time varies significantly so the variance in round trip time it is very high.

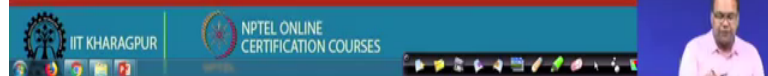
So, if you just take an average, the average will never give you a right estimation it may happen that well, the actual value falls somewhere here and there is a significant deviation from the average. And if you say retransmission timeout by considering that RTT estimation you will get some spurious RTO's. So, the solution here is that you use a dynamic algorithm that constantly adopts the timeout interval, based on some continuous measurement of network performance.

(Refer Slide Time: 26:27)

RTT Estimation at the Transport Layer

Jacobson's algorithm (1988) - used in TCP

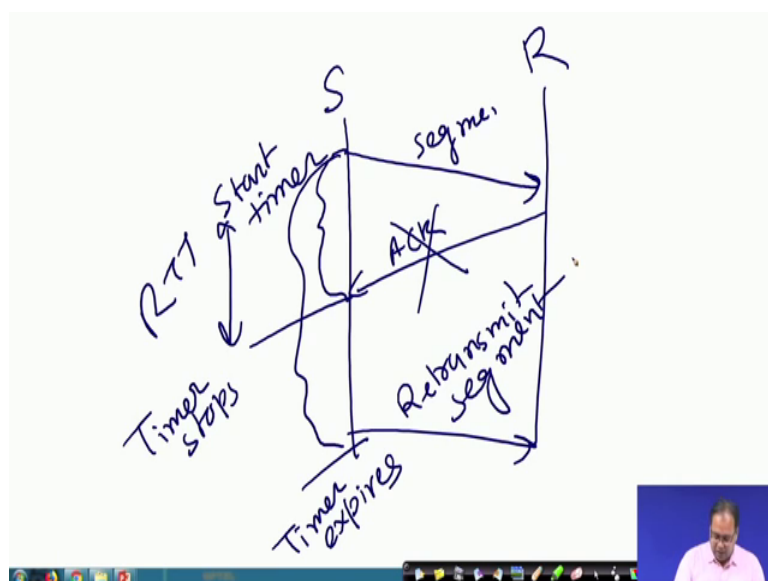
- For each connection, TCP maintains a variable, **SRTT (smoothed Round Trip Time)** – best current estimate of the round trip time to the destination
- When a segment is sent, a timer is started (both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long)
- If the ACK gets back – measure the time (say, R)
- Update SRTT as follows
$$SRTT = \alpha SRTT + (1 - \alpha)R \text{ (Exponentially Weighted Moving Average)}$$
- α is a smoothing factor that determines how quickly the old values are forgotten. Typically $\alpha = 7/8$



So, how will you do that? So, to do that we have something called the Jacobson algorithm proposed in 1988 which is used in TCP. So, the Jacobson algorithm says that for each connection TCP maintains a variable called SRTT the full form is Smoothed Round Trip Time which is the best current estimate of the round trip time to the destination.

Now, whenever your segment whenever you are sending a segment you start a timer. So, this timer has two different purposes like it can be used to trigger the timeout and at the same time it can be used to find out that how much time it takes to receive the acknowledgement.

(Refer Slide Time: 27:09)



So, whenever you have sent a message say this is the sender this is the receiver you have send the segment and you have start the timer. So, the timer the clock will keep on ticking. So, if you receive the acknowledgement here so at this stage you can think of that well this the timer stops here and this difference will give you an estimation of round trip time. But if you do not receive this acknowledgement, then after some timeout this timer expire say, here and once the timer expires you retransmits the segment.

So, it can be used for two different purposes this same timer. So, ah; so, you measure the time if you receive back an acknowledgement and you update the SRTT as follows. So, SRTT would be some alpha times the previous estimation of SRTT plus 1 minus alpha of this measured value R. So, this algorithm this mechanism we call as exponentially weighted moving average or EWMA. Now alpha is a smoothing factor that determines that how quickly the old values are forgotten like what weight you are going to give in the old values typically in case of t TCP Jacobson's set this alpha to a value of 7 by 8.

(Refer Slide Time: 28:39)

The slide is titled "Problem with EWMA" and contains a bulleted list of points. The first point states that choosing a suitable RTO is nontrivial even with a good SRTT value. The second point notes that the initial TCP implementation used $RTO = 2SRTT$. The third point, which is underlined, states that experience showed a constant value is too inflexible because it fails to respond when variance goes up (RTT fluctuation is high), which happens normally at high load. The fourth point, also underlined, suggests considering the variance of RTT during RTO estimation. The slide footer includes the IIT Kharagpur logo and the text "NPTEL ONLINE CERTIFICATION COURSES".

Problem with EWMA

- Even given a good value of SRTT, choosing a suitable RTO is nontrivial.
- Initial implementation of TCP used $RTO = 2SRTT$
- Experience showed that a constant value was too inflexible, because it failed to response when the variance went up (RTT fluctuation is high)
– happens normally at high load
- Consider variance of RTT during RTO estimation.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, this EWMA algorithm has a problem like; even you give a good value of SR SRTT, choosing a suitable RTO is nontrivial. Because the initial implementation of PCP it used RTO equal to two times of SRTT, but it has found out that still there is a significant amount of variance say ah; from the practical experience people have seen that a constant value, this constant value of RTO it is very inflexible because, it fail to response when the variance went up.

So, if your RTT has a measured RTT has too much deviation from the estimated RTT, then you will get the spurious RTO. So, in case your RTT fluctuation is high you may lead to a problem. So, it happens normally at high load so when your network load is very high your RTT fluctuation will become high.

So, in that case the solution is that apart from the average one, you consider the variance of RTT during the RTO estimation. Now how we consider the variance of RTT?

(Refer Slide Time: 29:41)

RTO Estimation

- Update RTT variation (RTTVAR) as follows.

$$RTTVAR = \beta RTTVAR + (1 - \beta) |SRTT - R|$$
- Typically $\beta = 3/4$
- RTO is estimated as follows.

$$RTO = SRTT + 4 \times RTTVAR$$
- Why 4?
 - Somehow arbitrary
 - Jacobson's paper is full of clever tricks – use integer addition, subtraction and shift – computation is lightweight

The slide includes a video feed of a presenter in the bottom right corner and logos for IIT Kharagpur and NPTEL Online Certification Courses at the bottom.

Now, to consider the variance of RTT so, you update the RTT variance variation which is termed as RTTVAR as follows. So, RTTVAR will be equal to beta time previous estimation of RTTVAR plus 1 minus beta of current estimation of the variance, that is the difference between current estimation of the RTT and the measured RTT that will give you the current variance and we set beta equal to 3 by 4.

Now, you estimate the RTO as follows so, you will take the SRTT value so; that means, the estimation of the round trip type into plus 4 times of RTT variance. So, you are here considering the variance as well so, if your network load becomes high. So, the system will get adapted to this variation. Now a question may come to in your mind that why 4? So, god knows so, why 4 so, it was somehow arbitrary. So, Jacobson's paper if you look into the Jacobson paper that had deal with this RTO estimation, in that case it is full of many such clever tricks. So, they have used integer addition subtraction and shift to make all this computation lightweight.

So, he has used this value 4 s o, that 4 is 2 square. So, you can apply the binary shift operation to make this computation. So, that is just a reason possibly Jacobson has utilized all this values and set this values set this particular.

(Refer Slide Time: 31:14)

Karn's Algorithm

- How will you get the RTT estimation, when a segment is lost and retransmitted again?

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Now, another question comes which is like how will you get the RTT estimation when a segment is lost and retransmitted again. If a segment has lost and retransmitted again, then you will not get the proper estimation of RTT because this segment you have transmitted the segment. So, the segment has lost you have started the timer here. So, there is a time out you again after the timeout we transmitted the segment and you got the acknowledgement.

Now, if that is the case, then this will; obviously, not give you an estimation of the RTT because in between the segment got lost and you have made a duplicate transmission of the same segment.

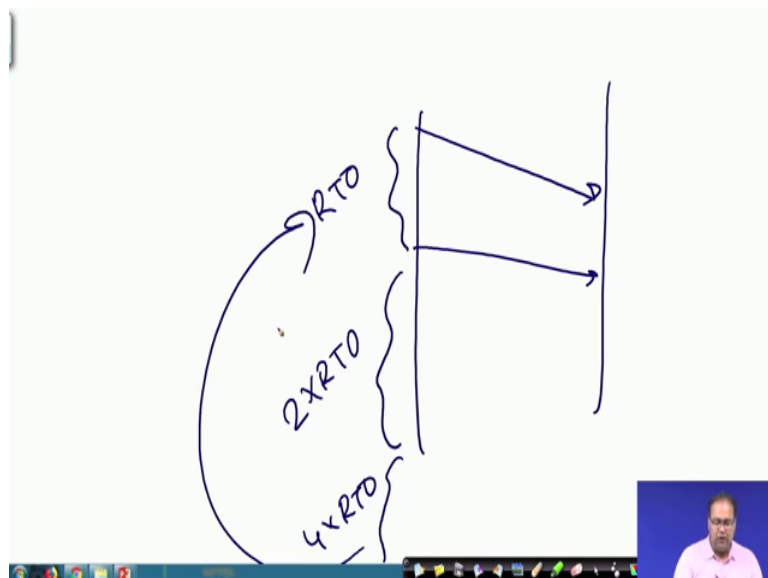
(Refer Slide Time: 31:56)

Karn's Algorithm

- How will you get the RTT estimation, when a segment is lost and retransmitted again?
- **Karn's algorithm:**
 - Do not update estimates on any segments that has been retransmitted
 - The timeout is doubled each successive retransmission until the segments gets through the first time

Now, to prevent this Karn's provides an algorithm and the Karn's algorithm says that do not update the estimates on any segments that has been retransmitted. So, you do not update your RTT estimation whenever you are retransmitting a segment. And a timeout is doubled on each successive transmission until the segment gets to through the first time. So, it is just like that once you have set a timer so, once you have set a timer say you got a timeout.

(Refer Slide Time: 32:17)

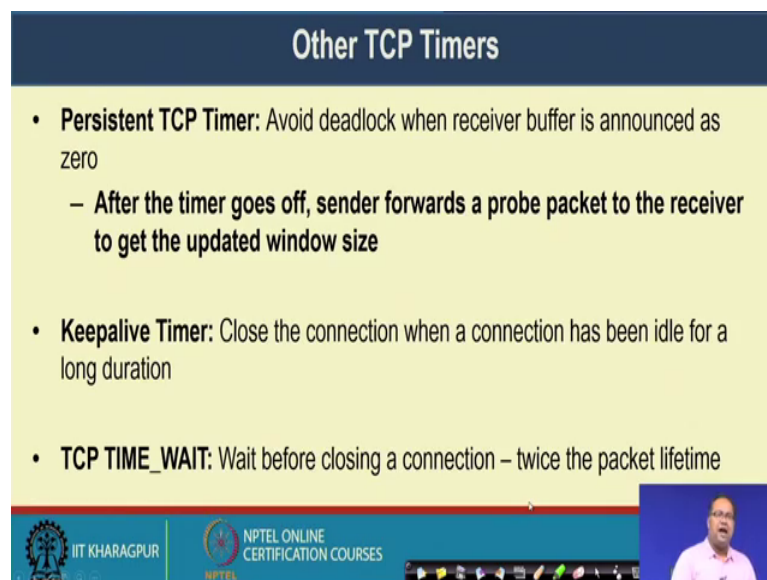


You retransmit the segment then you set say this was the value of RTO, but the next time you set is as 2 times RTO. So, you wait for more time to get back the response, if you get

back the response by that time it is good if you are not getting that then you make it 4 times the RTO.

So, that way you increment the RTO until you get back the acknowledgement whenever you are getting the acknowledgement again you reset it to the original implementation of the transmission timeout.

(Refer Slide Time: 32:56)



The slide is titled "Other TCP Timers" in a dark blue header. The main content area is yellow and contains three bullet points. The first bullet point is "Persistent TCP Timer: Avoid deadlock when receiver buffer is announced as zero", followed by a sub-bullet "After the timer goes off, sender forwards a probe packet to the receiver to get the updated window size". The second bullet point is "Keepalive Timer: Close the connection when a connection has been idle for a long duration". The third bullet point is "TCP TIME_WAIT: Wait before closing a connection – twice the packet lifetime". At the bottom of the slide, there is a blue footer bar with logos for IIT Kharagpur and NPTEL Online Certification Courses, and a small video inset of a speaker on the right.

- **Persistent TCP Timer:** Avoid deadlock when receiver buffer is announced as zero
 - After the timer goes off, sender forwards a probe packet to the receiver to get the updated window size
- **Keepalive Timer:** Close the connection when a connection has been idle for a long duration
- **TCP TIME_WAIT:** Wait before closing a connection – twice the packet lifetime

So, there are other TCP timers like this persistent TCP timer, which avoid deadlock when receiver buffer is announced as 0. So, after the timer goes off they send a forward flow packet to the receiver to get the updated window size there is something called Keepalive timer. So, this Keepalive timer it closes the connection when a connection has been idle for a long duration. So, you have set up a connection at not sending any data. So, after this Keepalive timer it will go off and then the time wait state which we have seen in case of connection closer. So, you wait before closing a connection which is in general twice the packet lifetime.

So, this is all about the flow control algorithm and different set up of your TCP timer values. In the next class we have we will see how we apply this loss or duplicate acknowledgement that we have seen here for the management of TCP congestion control.

Thank you all for attending this class.