

## What transport layer does

## Lecture 7 Transport Layer

In this chapter we see how the OS of the hosts use the facility of the network layer to build a communication mechanism for use by its programs.

### References:

- Textbook Section 8.4 and 8.5.
- RFCs: 768 (UDP), 793 (TCP), 1122 (Clarifications of TCP), 1323 (Extensions of TCP), 2581 (TCP Congestion control), 3168 (ECN in TCP).

Network(0234B)

Network(0234B)-7.1

- With network layer, any computer in a (inter-)network can send a packet to any other computer.
- Note that we talk about **computers** which are communicating. But when we write programs, **processes** in the computers actually communicate.
- The transport layer allow computers to **multiplex** its dialog with other computers to serve for multiple processes.
- E.g., for UDP, the network packet data contains a **port number**, and each process listens to some ports that it wants to use.
- It also allow **additional functions** to be provided other the network layer. E.g., in Internet, TCP provide reliability over the unreliable ("best-effort") IP network.
- The Internet has two primary transport layer protocol (UDP and TCP), and we will look at the inner workings of them in turn.

## UDP versus TCP

**Most applications should use TCP**, because it is easier to use:

- **Reliable:** it does flow control, duplicate handling, lost or misordered frame handling, congestion avoidance, etc.  
Shield ourselves from the network unreliability, and be nice to the network.
- **Connection oriented:** once the client contact with the server, a connection is allocated, which won't be interfered by other communications.

Some applications have **specific needs** that make TCP inappropriate:

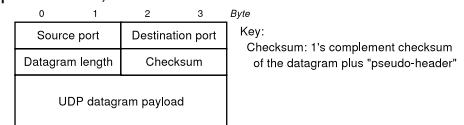
- They **don't want reliability** (e.g., communicate "streaming data" where timing is more important than avoiding loss).
- They **multicast** (can't connect with unknown number of peers).

Before specific protocol is designed for them in the OS, they can use UDP and implement whatever protocol they need.

Network(0234B)-7.2

## UDP packets

- UDP: **do just multiplexing**. Each UDP "datagram" is put into an IP packet of protocol 17, which looks like this:



- The sender put this into a single IP packet, thus size can thus be up to 65535 minus 20 (for IP header) minus 8 (for UDP header), i.e., **65507**.
- **IP fragmentation** might happen when the datagram is sent over the network. This can be **avoided** using the `IP_MTU_DISCOVER` socket option to find Maximum transfer unit (MTU) and `IP_MTU` to retrieve it.
- This works by the OS setting the DF option in IP and remembering the MTU advertised by the ICMP "fragmentation needed" packet.  
The application must then retransmitting the lost packet, though.

Network(0234B)-7.3

## UDP processing in the OS

- Applications using UDP will create a socket of the UDP type. Each such socket has a **local** and **remote** address and port number.
- When a **datagram arrives**, its destination is compared with the local part of all the sockets to see whether one of them matches. If so, it is queued into the socket if the remote part matches the source, and discarded otherwise.
- The **local part** normally allows any address to match, but only for one port. This can be set explicitly (using `bind()`), or chosen automatically during the first sending operation.
- The **remote part** is either set explicitly (using `connect()`), or left unset which means everything will be received.
- When the local part has the address unset, the address of the outgoing packet will be chosen during the send operations.

Network(0234B)-7.4

## TCP: introduction

- TCP (Transmission Control Protocol) has **much more features**, and correspondingly more complicated packet format and processing.
- Two primary differences to UDP as seen by applications:
  1. Applications see a **reliable byte stream**, not a message queue. I.e., segments can be merged or split. (For UDP, one `send()` is "normally" corresponding to one `recv()`).
  2. Applications see a **connection**, not just a port number. Multiplexing is based on **both** the local and remote parts. (For UDP, multiplexing is based on local part, remote part serves only for filtering.)
- So TCP has **2 basic parts: connection** management, and **reliability**.
- Reliability is achieved with sliding windows protocols (each byte has a sequence number).

Network(0234B)-7.5

## TCP packets

TCP segments are transferred using IP protocol number 6.

0	1	2	3	Byte
Source port		Destination port		
Sequence number				
Piggyback Acknowledgement number				
HdrLen/Options		Window size		
Checksum		Urgent pointer		
Options				
TCP segment payload				

Key:

Checksum: 1's complement checksum of the datagram plus "pseudo-header"

HdrLen/Options: 4 bits header length: in terms of 4-byte words  
6 bits reserved: must be 0  
URG bit: urgent pointer is used  
ACK bit: Acknowledgement number is used  
PSH bit: Ask receiver to send it to upper layer immediately, don't wait for more segments  
RST bit: Connection is reset, i.e., closed without normal disconnection procedure  
SYN bit: Request connection  
FIN bit: No more data will be sent  
Urgent pointer: byte offset of last data within segment that is considered "urgent"

Network(0234B)-7.6

## TCP Sliding Windows: similarity with Datalink

- All bytes have a **sequence number** associated with it. Both the sender and the receiver has a **window** of sequence number.
- The sender window specifies **what bytes can be sent before waiting**. When acknowledgement correctly arrives, the window is advanced.
- The receiver window specifies **what bytes it expects**. Data outside the receiver window will be dropped without further consideration.
- Acknowledgement may be sent by a dedicated packet, or piggybacked via a packet sent over **traffic of the reverse direction**.
- After a segment is sent, a timer is set up. If it **timeouts**, the segment will be **resent**. The timer is removed on receiving acknowledgements.

But we will see some differences from datalink windowing soon. They are about window size, NAK, timeouts, etc.

Network(0234B)-7.7

## TCP Connection strategy

- Each connection has **two directions**.
- A connection and the direction of flow is **identified** by a 4-tuple (Source-IP, Source-port, Destination-IP, Destination-port).
- Usually, the server chooses its port, the client use any port chosen by the OS. But TCP also **allows the client to specify its port**.  
E.g., for distributed applications where everybody uses the same port number.
- When a connection is **established**, both directions are made available for data transfer.
- TCP supports **graceful disconnect**, where each side **independently** declares that it has no data to send for its sending direction.
- Like connection data, such operations are **communicated** using TCP segments, **given sequence numbers**, and are fully acknowledged.

Network(0234B)-7.8

## Connection establishment

- When a **client** calls *connect()*, a segment is sent to the peer to **request for a connection**. The **server** accepts the connection it by **sending an ACK**. But...
- Suppose the network is so congested that packets are resent a few times. What would happen if packets resent is **replayed** to the host at a later time when the connection is established again?
- The problem: **old** data might be mis-interpreted as **new**.
- Quick fix: keep a state to **disallow the connection** (i.e., the 4-tuple) to be used again for some amount of time.  
For how long? The IP TTL is decremented for each hop and for each second waiting for forwarding. So we can assume after a certain time (called MSL, Maximum Segment Lifetime, standard says 2 minutes) there will be no replay.
- But... what if **the host crashed** and lost the state?  
The standard says if the host doesn't have any state, it should keep quiet for 1MSL (2 minutes). Few users want to need such waiting on every boot.

Network(0234B)-7.9

## Initial Sequence Numbers (ISN)

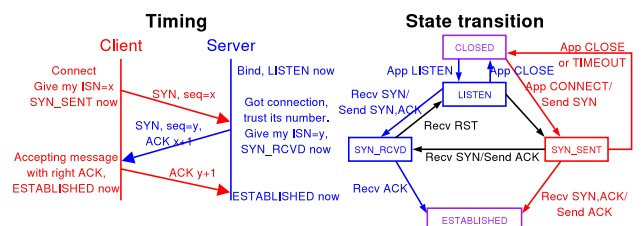
If the computer has a **clock** that is not reset upon crash, TCP has a safeguard against such situation, so that the connection won't be re-made.

- The idea: start the connection in a way that is **different when started at different time**.
- What information can be made different? Recall that the receiver will drop segments that are not within the receiver window. So **just start with different window!**
- TCP standard suggest the **sender side** of a connection to choose the initial sequence number **based on its own clock**, that **increases at a rate of 1 per 4μs**.  
So that if data rate is slower than  $1/4\mu = 250k$  bytes per second, the data uses numbers slower than the ISN increase rate. This ensures data of an earlier connection will not be accepted by a later connection. (Except when wrap-around.)
- So during connection establishment, **ISNs must be exchanged**.

Network(0234B)-7.10

## Three-way handshake

To exchange ISNs, we add one more message, resulting in a connection protocol called **three-way handshake**:



The "color coding": blue = server normal path, red = client normal path, purple = both, black = exceptional cases.

If things go well, only the first SYN packet can be accepted, no other. Because their sequence numbers are wrong.

Network(0234B)-7.11

### Fast connections and Wrap-around

- What will happen if sequence number wraps around? E.g., **ISN wraps around every 2.39 hours**. The reliability could suffer, because it opens a small window for old packets to be accepted as new.
  - 2.39 hours =  $2^{31} \times 4\mu$  seconds.
- The more dangerous assumption made: nobody wants more than **1 sequence number per 4μs**, i.e., 250KBps. This is simply untrue.
- For faster connections, the original TCP depends only on waiting time to be reliable. The **sequence number can't guarantee** that old duplicates of **previous** connections won't be accepted as new ones.
- It is even worse: using gigabit networks, the 32-bit counter could **wrap around** in 17 seconds, **before MSL passed!**
- So there is real danger that an old duplicate of the **connection itself**, rather than a **previous** connection, get accepted as a new packet!!

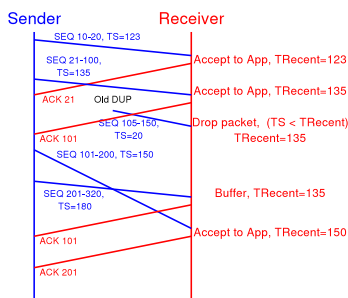
Network(0234B)-7.12

### Time-stamp option: PAWS

- Solution: an extension to TCP to add a **timestamp** to **all** packets.
  - In Linux, this is configured with `/proc/sys/net/ipv4/tcp_timestamp`, normally enabled.
- If both the server and the client uses a SYN packet with the timestamp option, all packets following it will carry **two 32-bit timestamps**.
  - For the two directions, currently we are only interested in one direction. RFC 1323 says the timestamp should tick at an interval between 10ms to 1s, to avoid itself wrapping within 1MSL while supporting up to 8 Tbps speed.
- **Protection Against Wrapped Sequence numbers (PAWS)**: drop a segment if its timestamp < a recent one accepted for app layer.
- With this, old duplicates of the same connection have no chance to be accepted: it will either have too low sequence numbers or timestamp.
  - There's a slim chance of dropping re-ordered packets needlessly, though.

Network(0234B)-7.13

### Illustration



Note that the buffered packet won't cause TRecent to increase, otherwise the delayed 101-200 packet would get dropped incorrectly.

TRecent is updated to 150 rather than 180, as it is easier (Receiver won't need to remember all the timestamps of the buffered packets).

Network(0234B)-7.14

### Graceful disconnection

TCP allows connections to be closed **gracefully**.

1. Connection is not closed until **no more data transfer** is needed.

The FIN bit ensures this: FIN means "I have nothing more to send", while the other side can continue to send data. A connection can be close **only after** both side sends a segment with FIN bit set.

2. After closing connections, **both** sides can free all resources.

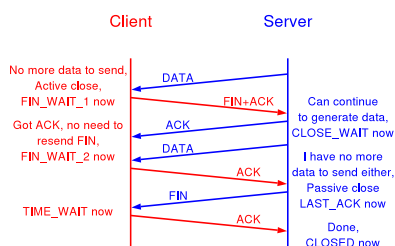
Difficulty: the FIN packets can get **lost!** The FIN packet would thus need to be resent. So we can close once we receive an ACK for the FIN.

But... the ACK can also be lost as well... when should the other side decide that it is safe to close??!

Solution: just stop after sending ACK. If the ACK is lost, the FIN sender will try again a few times, timeout, and close anyway.

Network(0234B)-7.15

### Disconnection sequence



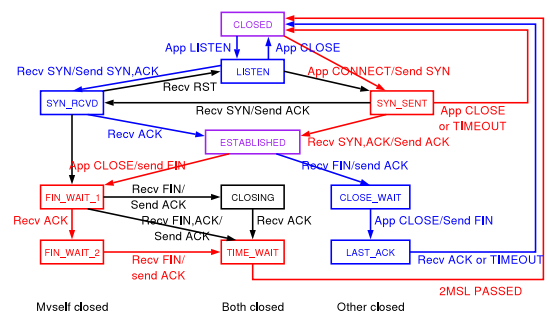
To give the full flexibility, the socket API allows a one-side disconnection operation, using `shutdown()`.

The `shutdown()` call can also indicate that "I'll receive no more data". If the sender continue to send data, a RST (Reset) segment is replied, and the connection is completely shut down non-gracefully. The `close()` call shutdowns both directions. The sender is indicated by a signal (`SIGPIPE`).

Network(0234B)-7.16

### The full state transition diagram

This sums up the connection and disconnection procedure:



Note that the 2MSL time wait is for the "quick fix" for duplicate avoidance we mentioned earlier.

Network(0234B)-7.17

### SYN flood

- Suppose somebody (an attacker) sends a **lot of SYN packets** to a port that your server is listening to, without intention make a connection.
- Your server go to the SYN\_RCVD state for the corresponding connections. But... **the client does not respond** at all.
- What will happen? The TCP protocol will **timeout** after some time, e.g., 2MSL. But...
- During this time, there can be a real lot of such SYN packets received. Your server will **run out of space** in the table storing half-connections.
- Early OS would now reject all new SYNs, and the server is said to be **SYN-flooded**: no new client can connect to it—a Denial of Service.
- Worse, the attacker won't expect a connection, so the IP addresses he uses will be all wrong: **very difficult to trace the attacker** or stop him!

Network(0234B)-7.18

### Solving SYN flood

The problem can be solved by a few ways. 2 popular ones:

1. When buffer is full, **drop a random old connection**. Since the attacker have the largest number of connection in the table, the odds are that the dropped connection is that of the attacker.

Drawbacks: still cause trouble to the new connections.

2. **SYN cookies**: When buffer is full, **work without a buffer**. But how?

The idea: When the SYN is received, the server compute a **hash** based on the client IP and port, as well as the current time. It is used as the ISN, and an SYN-ACK is sent back to the client.

Upon receive of the client's ACK, the connection is created if (1) it is not too old, and (2) the hash calculation gives a correct value for ISN.

This works only if the attacker can't compute your hash, otherwise the attacker sends ACK instead to flood your ESTABLISHED table. Cryptography is needed.

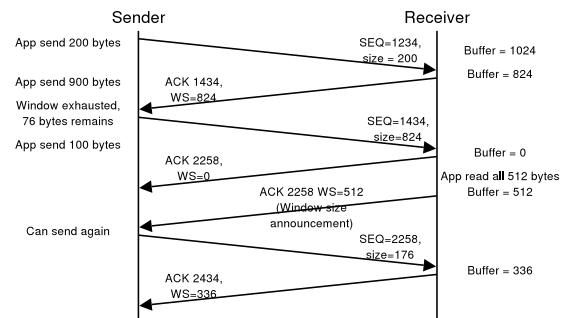
Network(0234B)-7.19

### Window sizes for the sliding window

- How to **set the window sizes**? Ideally, the sender window will be at most that of the receiver window, so that if the receiver does not have enough space, it won't try to send.  
Sender has its own reason to use a smaller window, e.g., memory low.
- How the sender knows the receiver's window? **The TCP header has a "Window size" field** which tells how large is it currently.
- This depends on **how fast the application is getting the data**. E.g., if the application get stuck, the OS kernel buffer will be full soon, and the Window Size would then be 0.
- Whenever the sender receives an ACK, it computes and records what is the **last byte that can be sent** without exceeding the receiver window. Data after that byte will be held in the sender's buffer.  
When sender buffer is full, the `send()` operation blocks.

Network(0234B)-7.20

### Buffer management



Network(0234B)-7.21

### Other subtleties of windows management

- But what if **window size change** get lost? Both sender and receiver would be waiting...
- Answer: The sender sends at least 1 byte of new data every 2 minutes to get out of the loop. If there is no buffer, this segment is dropped.
- There are some **efficiency** problems, though. What happen if the sender application **write 1 byte a time** (typical for telnet etc)?
- A lot of 1-byte message will be sent... but 1-byte message means a 21-byte segment, or 41-byte packet, and even larger PPP frame... and there's a similar frame for the reverse traffic on ACK!  
Not usually a problem for LAN, but on WAN this will add fire to congestion.
- There's a reverse problem: receiver gets 1-byte at a time, causes many window size changes each asking for 1 more byte (**silly window syndrome**).

Network(0234B)-7.22

### Nagle and Clark

TCP is modified by a few rules to fix the problems.

- **Nagle**: When sending less than 1 MSS, suppress further sending until it is acknowledged. (At that time we probably have more bytes.)

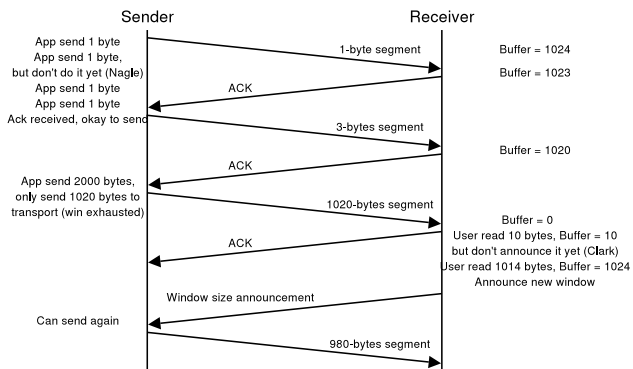
This is **self-adjusting**: For fast links like LANs, the ACK comes quickly, so the Nagle algorithm is effectively suppressed. The slower the link, the more the Clark algorithm delays sending.

- **Clark**: don't send a window size update until it reaches either the receiver's maximum transfer unit or half the allocated buffer.
- **ACK piggybacking**: ACK with no data is not sent until 200ms passed. (But this interfere with Nagle!)

Problem: Nagle requires an ACK but the other side wait for 200 ms!  
Temporary solution: for concerned applications, set the TCP\_NODELAY socket option to disable Nagle's algorithm. But this increases network load.

Network(0234B)-7.23

### Events in Nagle and Clark



Network(0234B)-7.24

### Window size

- The **size of window** is expressed as a **16-bit integer**. So the window size is at most 65535. For "long fat pipe", this hinders performance. Long Fat Pipe (LFN) means the link has large delay bandwidth product, so the pipe has a larger capacity and needs a larger buffer to be fully utilized.
- The solution: allow the Window size to be **scaled**. This is negotiated by a TCP option in the SYN packet:
  - Each side sends a **Window Scale** option with the scaling factor it wants to use, expressed as the number of bits to shift the window size.
  - The scale is actually used if both sides send such an option.
- The maximum window can then be at most  $2^{30}$  (to avoid wrap-around). This is selected based on the **receive buffer size** when the connection is made.

Network(0234B)-7.25

### Setting timeouts

- The sliding window algorithm needs **timeout** to resend segments. If the **packet is most likely lost**.
- The probability depends on the current **average round-trip-time (RTT)** and its **variance**, but they change **during the connection**.
- When a packet is acknowledged, its **round trip time** is measured, and is used to **update an estimate** of the average *RTT* by a rule  $RTT = \alpha RTT + (1 - \alpha)M$ , where  $0 < \alpha < 1$ , e.g.,  $7/8$ .
- We can estimate the **mean variance  $D$**  (i.e., average deviation from mean) of *RTT* similarly. The update rule:  $D = \alpha D + (1 - \alpha) |RTT - M|$ .
- With an update to *RTT* and *D*, the **retransmission timeout** is updated to  $RTT + kD$  for some *k*. In practice, *k* is chosen to be 4 to make it easy to calculate.

Network(0234B)-7.26

### Measuring delay

- The delay can be measured by the sender **record its send time**, and compare it with the time **when an ACK is received**.
- What if the segment is **resent**? We don't know whether the ACK correspond to the old or new packet.
- Karn's rule**: don't update *RTT* on retransmitted segments. When retransmit, double the current timeout value to slow down retries. For retransmitted segments, we don't know which transmit an ACK refers to.
- Defect: a **whole window** is usually resent, so the whole window will have no usable sample for the estimate.
- A better tool: make use of the **timestamp**. The receiver **echos** the timestamp (actually, the TRecent value) to the sender, so that the sender estimate the RTT by just subtracting it from the current time.

Network(0234B)-7.27

### Congestion control

- Network layer of intermediate routers has **limited capacity**, due to limited memory, bandwidth and CPU cycles.
- If **many transport level connections** (perhaps between different pairs of computers) must go through the same router, the **queue** of routers builds up, so everybody waits longer.
- If that persists, routers memory will get used up, so packets must be **discarded**. The transport layer segments are thus lost.
- Note the difference between flow control and congestion control:
  - Flow control solves a **local** problem that **either end** has no capacity to process the data.
  - Congestion control solves a **global** problem that **intermediate** router runs out of capacity.

Network(0234B)-7.28

### Two involved layers

Both the network layer and transport layer are involved in any congestion control scheme.

- The end-points don't know whether a connection or new segment will cause congestion **in the middle** of the network. Only the **network layer** can **detect** them.
- The network layer cannot **stop traffic** by itself, since transport layer must **reduce rate of sending segments** or **connecting** before a congestion get resolved.

In general, congestion are **detected** in the network layer, and the transport layer is **notified** by the network layer in some way.

Since Internet does not mandate a scheme to "reserve" bandwidth, congestion handling relies on **feedback**.

Drawback: no bandwidth and delay guarantee. This is being changed by IPv6 by the provision of flow labels and differentiated services.

Network(0234B)-7.29

### Detecting congestions

- If a router receives a packet, wants to forward it to an interface, but it runs out of buffer, the router **knows** there is congestion.
- At this time, the router has no option but to **drop some packet**.
- Which packet to drop? Usually, the best is simply a **random** one.
- But this is usually **too late**. A more aggressive scheme is to keep a run average of the **queue length** of each outgoing buffer.
- If **queue length is long** (e.g., consistently more than 80% of buffer), the router considers that there is congestion and takes action.
- This is called **Active Queue Management (AQM)**.

Network(0234B)-7.30

### Reporting congestion problem

- If buffer is completely full, the network layer must **drop a packet**. This can be seen as a reporting mechanism.  
This assumes most drops are due to congestion, which is really the case.
- What if AQM is used and the router want to take early action? It can drop a random packet anyway even when queue is not full: **Random Early Detection (RED)**.  
This must be configured to the network software explicitly.
- Another way: **send a packet** to the sender. The ICMP packet **Source quench** is used for that purpose.
- But extra packets adds to congestion, so it's not recommended.
- New alternative to wastefully dropping an otherwise healthy packet: continue to send the packet with an **explicit congestion notification**.

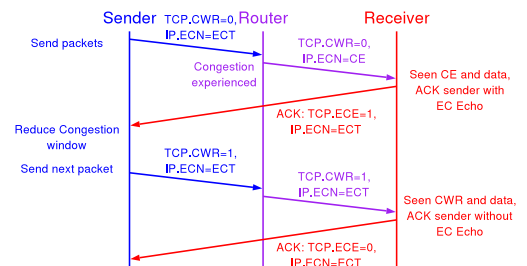
Network(0234B)-7.31

### Explicit Congestion Notification

- Last 2 bits of the IP packets TOS field (previously unused) is originally 00. If hosts supports ECN, it is set to 01, **Explicit Congestion Transport (ECT)**, i.e., transport layer supports ECN.
- If a router detects collision with AQM, want to drop a packet and see it is ECN enabled, the router changes it to 11, **Congestion Experienced (CE)**.  
It means: this packet would have been dropped.
- The transport layer receiver then **acknowledge** with a transport layer bit (in TCP, within the "reserved" bits called "EC Echo") to tell the sender.
- The sender can thus deal with the congestion, **without having to deal with a lost packet**.  
In TCP, it also has set another bit in the reserved TCP header in the next packet it send, called CWR ("Congestion Window Reduced") to acknowledge it. It should become clear in the next slide why it is named as such.

Network(0234B)-7.32

### Congestion handling in TCP



Note the use of CWR: this makes sure that if the ACK carrying EC-Echo is lost, the next ACK will still carry the EC echo to slow down the sender.

Network(0234B)-7.33

### Congestion handling in TCP

When a TCP connection **timesouts waiting for acks**, we assume there is congestion, so the connection must **slow down** sending. But **how**?

- Answer: **reduce the size of sending window**.
- Until now we assume that the sending window depend only on the **window size** announced by the other end. This is not the full picture.  
That deals with flow control, not congestion control.
- There is another window, called **congestion window**. It **estimates** how much data can be sent without problem.
- Size of sending window is the **minimum** of the *window size* announced by the receiver and the *size of the congestion window*.
- On timeout or EC Echo, the size of **congestion window** is **reduced**.
- When segments start to get acknowledged normally (congestion is over), it is enlarged again to speed up the connection.

Network(0234B)-7.34

### Congestion control algorithm: AIMD

- 2 variables per connection: **slow start threshold**  $ssthres$ , **congestion window size**  $cwnd$ . Let's focus on  $cwnd$  first.  
In implementation, both are integers counted in bytes. But for convenience of discussion, we treat them as floating point numbers representing number of segments of size MSS, the maximum segment size.
- **Increase**: When each new segment is acknowledged, it increases  $cwnd$  by  $1/cwnd$ . So when a whole window is received (i.e.,  $cwnd$  segments arrived),  $cwnd$  is approximately incremented.  
Or, keep a counter so that  $cwnd$  is incremented after ACK of a full window.
- **Reduction**: on congestion,  $cwnd$  is set to  $cwnd/2$ .  
Not really the case: as we will see,  $cwnd$  is set to 1.
- So Increase is **Additive**, Decrease is **Multiplicative**. We call it AIMD.
- The congestion control algorithm is used whenever the TCP connection has a peer in **another network**.  
Otherwise, the transmission is limited only by the receiver window.

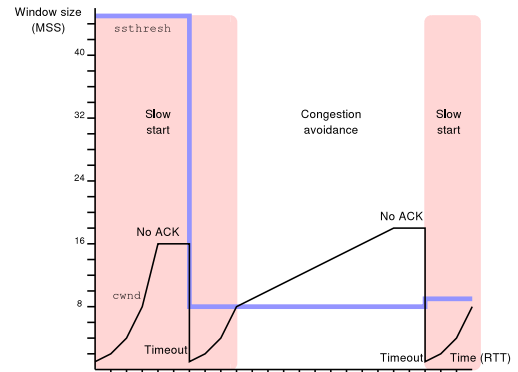
Network(0234B)-7.35

### Slow start

- The problem: what should be the initial  $cwnd$ ? If too small, it takes too long to fully utilize an LFN with additive increase.
- When  $cwnd < ssthresh$ , TCP doesn't use AIMD: **slow start** mode. E.g., initially,  $ssthresh$  is set large, e.g., 64KiB;  $cwnd$  is set to 1MSS. So  $cwnd < ssthresh$ —slow start.
- Slow start: when a segment is acknowledged,  $cwnd$  is increased by 1 **MSS**. So full window ACK  $\Rightarrow$  double  $cwnd$ : **exponential** increase!
- The first “slow start” stops when one segment is lost. Lost packets cause TCP to set  $ssthresh = cwnd / 2$ , and  $cwnd = 1$ —slow start again, with a much smaller threshold.
- So  $cwnd$  counts from 1. Suppose a packet is lost when  $cwnd = 16$ . Then  $ssthresh$  is set to 8. Counting restarts from 1 to 8.
- Then  $cwnd > ssthresh$ : **congestion avoidance mode**.  $cwnd$  increases much slower: to 8.125, 8.248, 8.369, etc.

Network(0234B)-7.36

### Example



“Slow” start means slower than the threshold. But the increase is not slow at all.

Network(0234B)-7.37

### Behaviour

- One consequence of AIMD: if two connections both loss a segment, the one with a **larger congestion window** gets **penalized** more heavily. Because it needs longer time for the additive increase to recover the lost congestion window size.
- If there are **multiple TCP connections with heavy traffic** through the **same congested router**, this has the tendency to **equalize** the congestion window of all these connections.
- This is conceived as being **fair**. There are problems, though...
  - Not all network packets are TCP. In particular, **UDP programs** has no obligation to use AIMD, and **can be unfair**.
  - Not all users of that router uses the **same number of connections**. One using more connections can use unfair amount of bandwidth.
- **New transport protocols will mandate** the use of AIMD, though.

Network(0234B)-7.38

### Fast retransmit, fast recovery

TCP tries to repeatedly test the network to see whether its bandwidth is enlarged. But there's a problem: a failure initiates a costly slow start.

- **Fast retransmit**: if the sender receives **3 duplicated ACKs**, conclude that a packet is most likely lost...
  - Resend the acknowledged frame, without timeout (acts like NAK).
- **Fast recovery**: after fast retransmit, or on ECN-Echo:
  - Set  $ssthresh = cwnd / 2$ , as it is congestion notification.
  - Don't set  $cwnd = 1$ . Instead... set  $cwnd = ssthresh + 3!$
  - Rationale? Each duplicated ACK means a later packet is received.
  - Each further duplicated ACK **inflate**  $cwnd$  by 1MSS, until a new ACK is received when we **deflate** it back by setting  $cwnd = ssthresh$ .

Network(0234B)-7.39

### Timer management

There is an interesting consequence of AIMD. Since we will set  $cwnd$  to 1 when a timeout occurs, we can only resend 1 segment... so we **don't need to keep multiple timers!** TCP recommendations:

- Keep a **single retransmission timer** for each connection.
- When **sending** a segment at a time when the timer is **not set**, set it to the current RTO (computed from the current RTT estimates).
- When **sending** a segment at a time when the timer is **already set**, just leave it unchanged.
- When receiving a **new** (non-duplicated) **ACK**, **reset the timer**: turn it off if no more data in transit, and set it to RTO if otherwise.
- When receiving a **duplicated ACK**, leave the timer unchanged.
- When **timeout**, resend. Double the RTO value to slow down retries.

Network(0234B)-7.40

### Further improvements

- What we have described is called the TCP **Reno** algorithm.
- It has a major problem: it always increase the window size until it is far too much, which leads to multiplicative decrease (or, worse, slow start).
- It is possible to do better. The TCP **Vegas** algorithm:
  - Continuously use the timestamps to track the current delay.
  - If the delay start to increase significantly, don't increase  $cwnd$ . And if the delay increase much, **decrease**  $cwnd$  slightly.
  - All these are done **before** ECN or packet loss occurs.
- But it is not yet a popular implementation, since with existing Reno implementation, Vegas tends to get smaller bandwidth.

Network(0234B)-7.41