Transport Layer

The transport layer is used for:

- Demultiplexing: decide which application should receive the data.

  This is accomplished using port numbers, which are in the transport header and designate the application (Web server is port 80 for example).

  UDP (User Datagram Protocol) does only this.

  A unit in the transport layer is called a segment.

- If segments get lost, retransmission has to be done

- If the routes change, we should reorder the segments to get the original order the segments were sent in (multiple segments may constitute a single file)

  TCP does the above two.
  ↳ Transmission Control Protocol

→ It does not send segment (n+1) until segment (n) has been received

The issue is that routers do not cooperate to ensure reliable data transfer. We should be able to infer packet loss!

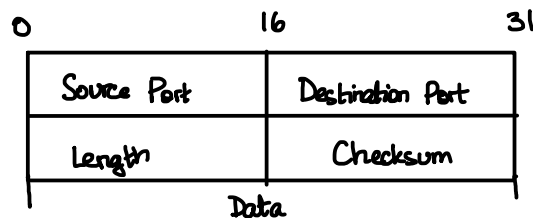If we want something between UDP and TCP, we should use UDP and build our own protocol on top of it.

The mechanism via which we drop packets at a router if the queue becomes full is called a drop-tail mechanism. → congestion

TCP is able to infer that congestion has occurred and does congestion control: the segment sending rate is reduced Enough of the input links back off to allow the queue to clear up.
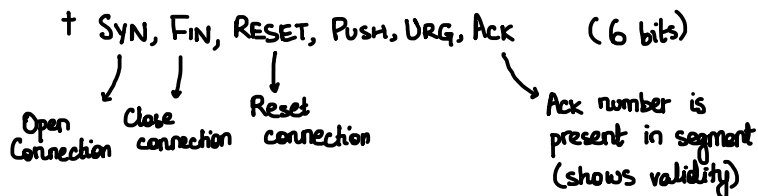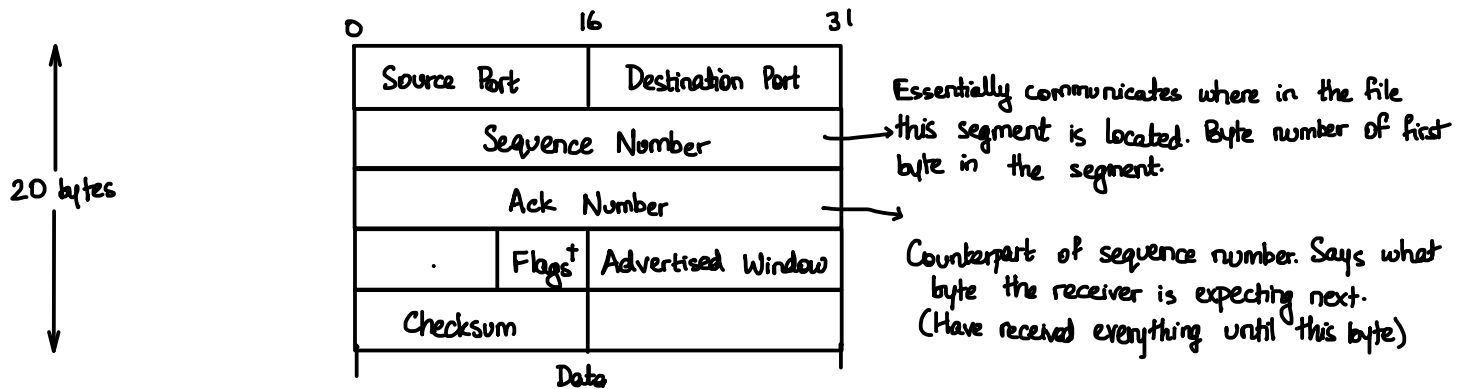
TCP also does flow control, which is just congestion control at the destination. This is different because the source and destination can communicate. The destination may take time to process information, and may not have read all the information from the transport layer yet (there is a buffer).

Van Jacobson and Sally Floyd did a lot of work on TCP in the 80s.

The UDP header is just

0	16	31
Source Port	Destination Port	
Length	Checksum	

Data

The TCP header on the other hand, looks like

20 bytes

0	16	31
Source Port	Destination Port	
Sequence Number		
Ack Number		
.	Flags†	Advertised Window
Checksum		

Data

Essentially communicates where in the file this segment is located. Byte number of first byte in the segment.

Counterpart of sequence number. Says what byte the receiver is expecting next. (Have received everything until this byte)

† SYN, FIN, RESET, PUSH, URG, ACK   (6 bits)

Open Connection   Close connection   Reset connection

Ack number is present in segment (shows validity)

The protocol field in the IP layer says whether the transport layer is using TCP or UDP.
6   17

Lecture 23   TCP. Opening and Closing a Connection

Recall that there is a sequence number and an ACK number in TCP.
This allows to send (data + ACK) together.

How is a Connection established?
Suppose there is a web server, which is a passive participant, listening on some port and an active client that is trying to open a connection.
→ First, a SYN flagged packet is sent. Suppose it has a seq. no. x.
→ The server responds with a (SYN+ACK) flagged packet with ack. no. $x+1$. Suppose it has seq. no. y.
→ The client responds with ACK $y+1$.
This is called a three-way handshake.
While SYN/FIN packets don't have data, they are considered to have one byte.
Usually, the value x is randomly selected.
ACK z acknowledges that we have received everything from start to $z-1$.

Why do we want a random X and Y?
Suppose we want to transmit two files of size N and M and we start with seq. no. at 0. If some segment from the first packet arrives late, then it could cause problems. If the port numbers are the same, we could interpret it to belong to the second file, which we do not want.
We choose the initial seq. no. randomly in the hopes that the sequence number spaces do not overlap.
If the file is large enough, it is possible to wrap around from $(2^{32}-1)$ to 0.

Let us draw the state transition diagram.
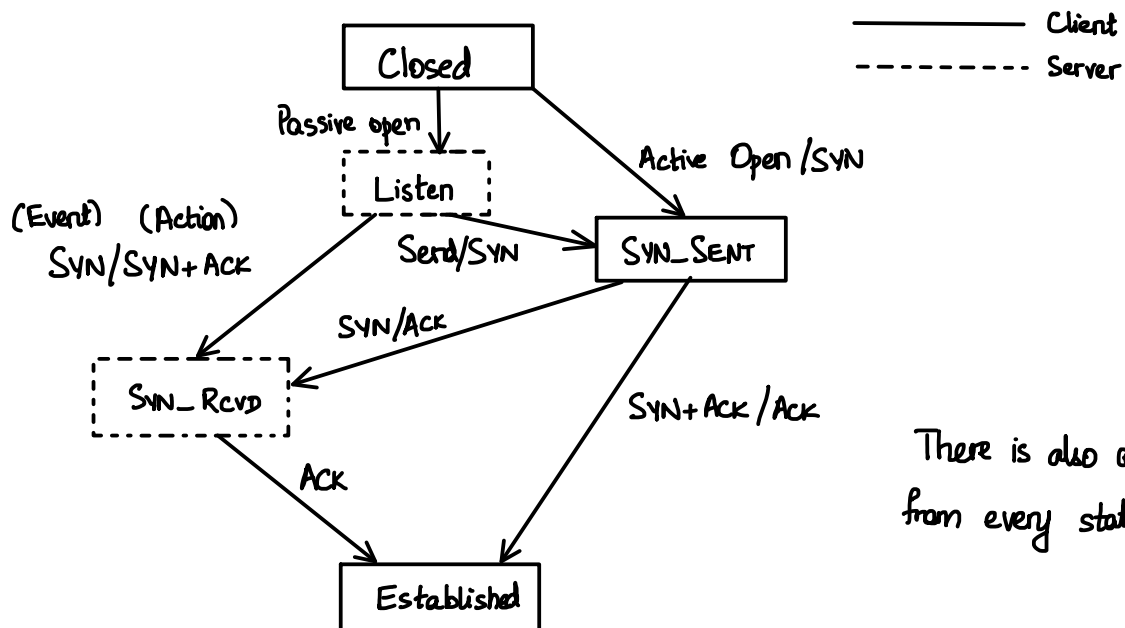
The above is complicated because we do not know what the other side is doing. We have to account for them sending a SYN at the same time as us. The actual diagram is in fact far more complicated.

It is better to actively terminate a connection because this tells the other side that they can clear their buffer and any state-related data.
To do this, we use a FIN-flagged packet.
1. Half-close: Closed by one end at a time. The other end can continue to send some data, but it need not receive a response. After it is done, it sends a FIN flag as well, and receives an ACK.

2. 3-Way Handshake Termination: Both sides close at the same time.
   The client sends a FIN, and the server responds with (FIN + Ack)
   together. The client responds with an Ack (to the FIN).

When the sender sends a packet, they expect a response by a certain time
If we do not receive an ACK within the timeout, we retransmit.

What do we set the timeout to?
A couple of issues are.
1. The RTT can vary quite a lot depending on the network path.
2. The RTT can vary across time as well due to the queue lengths.
                                    (even for the same path)
3. Packet loss can occur. When we receiven an ACK, could it be due to
   a previous transmission of the same packet?
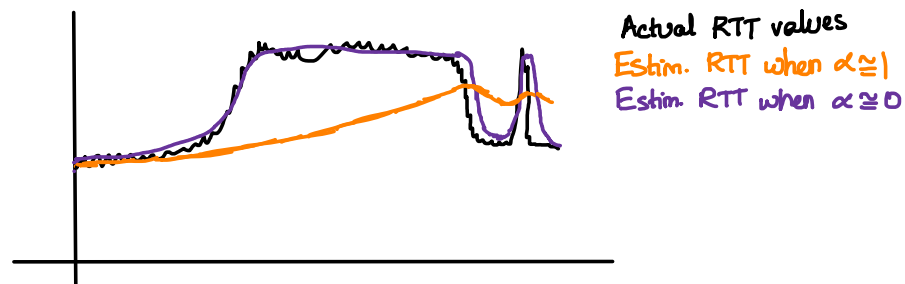
We would like to get an average RTT.
                        ↳ in what sense?

Let the "sample RTT" be the most recent RTT.
We take the new estimated RTT as a weighted average
$$\alpha \times \text{Estim. RTT} + (1-\alpha) \times \text{Sample RTT.}$$

For example,

Actual RTT values
Estim. RTT when $\alpha \cong 1$
Estim. RTT when $\alpha \cong 0$

Both the extreme values of α are not very good.

The old algorithm uses
$$\text{Timeout} = 2 \times \text{Estimated RTT}$$

In the new algorithm, we try to take the variation of RTT into account.
Suppose we have a Gaussian distribution $N(\mu, \sigma)$. If the RTT follows this distribution, then $\mu + 3\sigma$ serves as a good value for the timeout.

Measure Estim. RTT (mean) as before.
Let Diff = Sample RTT − Estim. RTT
Since standard deviation is tricky due to the square root, we use the mean deviation insted.
That is, we let

$$Dev = (1-\beta) \times Dev + \beta \times |Diff|$$

The timeout is then set as

$$\mu \times Estim.\ RTT + \phi \times Dev.$$

Usually 1   Usually 4

We usually take $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$.
The initial timeout value is quite large.

What about the packet loss issue?
   We do <u>not</u> use the RTT measurement for retransmitted packets.

Now, let us move on to congestion control.
Initially, because the internet was for military purposes, TCP was kept to not have state information (it is localized to the end nodes).
Consider the receiver buffer.

MAX RECV BUFFER   (a circular buffer)

↑
Application has
read till here.
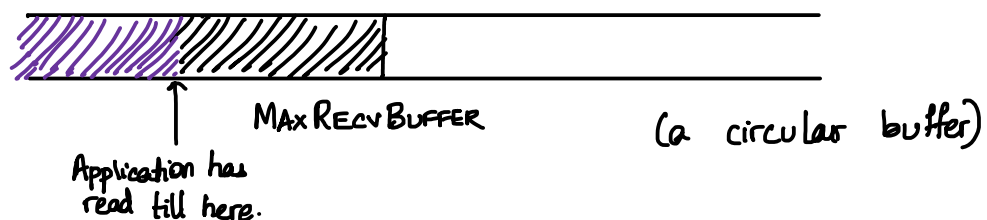
The left and right sections are free. If congestion is at the receiver, we can send this buffer information.
What if congestion is at an intermediate router? There is no direct communication so TCP must infer this.
   (Rarely, we use Explicit Congestion Notification (ECN))

How does TCP infer this?

Flow Control deals with congestion at the receiver

The advertisement window field in the TCP header is equal to the space left in the receiver buffer.

The sender has a window, which is the maximum amount of data (in bytes) which is outstanding — sent but not ACKed.

If we send window many bytes, the data rate is $\cong \frac{window}{RTT}$

First, the window must be less than the advertisement window.

(to ensure that we take care of flow control)

We calculate the Congestion window and set

$$window = \min \{ congestion\ window,\ adr.\ window \}$$

result of inference about
congestion at routers

This is approximately equal to

$$DataRate \times RTT \longrightarrow Delay-Bandwidth\ Product$$
(maximum possible)

It remains to choose the congestion window.

Lecture 25   Congestion Control

When congestion occurs,

1. packets are dropped (and routers cannot retransmit them, retransmission will have to be done from the source).

2. queuing delays may become very large.

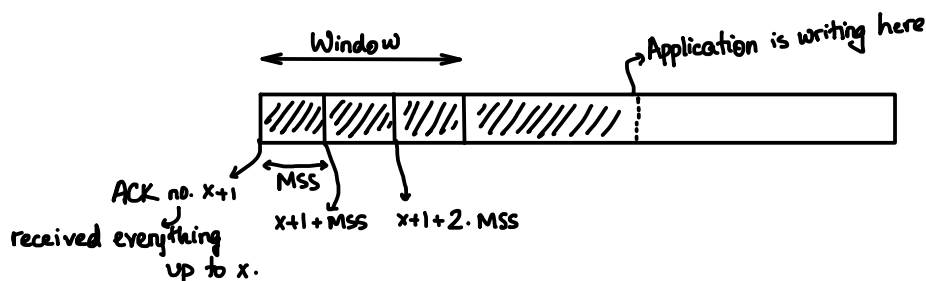As mentioned, the Congestion window tries to fix congestion control.

How do we infer congestion?

• High RTT could signify congestion (but how high is high?).

• Packet loss could signify congestion. We can use a timeout or ACK feedback to infer whether a packet loss has occurred.

Let us see what TCP does.

Consider the buffer at the transport layer of the sender. Suppose seq. no. x corresponds to SYN, so the first packet has seq. no. $x+1$. Let MSS be the "maximum segment size".

How many segments do we send? This depends on the window size

If the initial window is $3 \times$ MSS, we send out 3 segments.

Window

Application is writing here

ACK no. $x+1$
received everything up to x.

MSS

$x+1+$MSS $x+1+2 \cdot$MSS

Consider the buffer of the receiver, which is initially empty. Suppose it receives the first segment. It replies with ACK no. $x+1+$MSS.

next expected byte

The ACK no. (in the diagram above) gets updated to $x+1+$MSS — it represents that the sender has got everything until $x+$MSS. This is called cumulative ACK.

Ack no. z means that everything until $z-1$ has been received.

Now, the window at the sender is pushed to the right.

Window

Application is writing here

$x+1$

ACK no. $x+1+$MSS

$x+1+2 \cdot$MSS
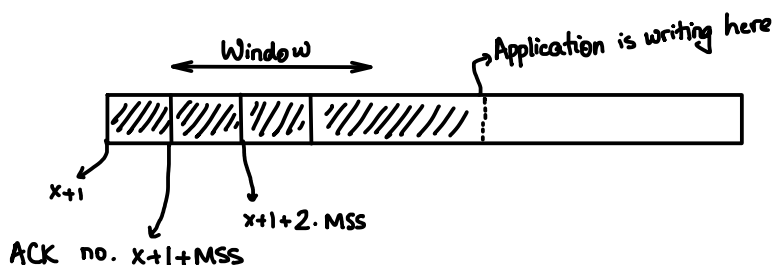
This is called a sliding window mechanism. The window itself might be increased/decreased depending on congestion.

What if the second packet is lost and the third packet is received?

Due to the cumulative ACK mechanism, we send ACK no. $x+1+$MSS itself. (receiver)

These ACKs with a repeated number are called duplicate acknowledgements.

Each dup ACK means that one more segment has been received, but we may not know which. They do not push the window to the right.

Next, let us look at some principles of congestion control.

1. If there is no congestion, increase the congestion window conservatively.

   We do "additive increase of window size". If there is no congestion, we grow the window linearly as 1 MSS per RTT. This is a slow increase.

   TCP uses self-clocking to measure RTT — we increase the window using ACK arrivals.

   Suppose we have sent out $\frac{Congestion\ Window}{MSS}$ packets   + ~1 RTT
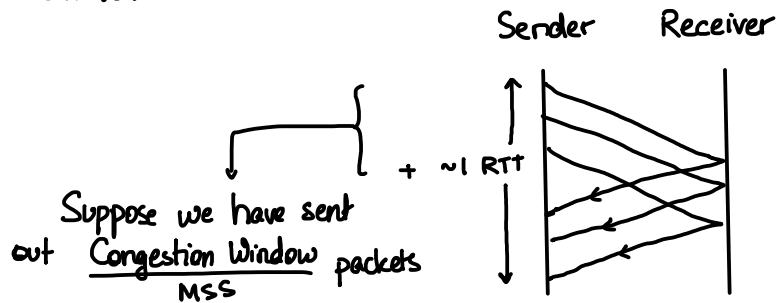
   How much should we increase window for each ACK?

   Just $\dfrac{MSS}{no.\ of\ ACKs} = \dfrac{MSS}{CW/MSS} = \dfrac{(MSS)^2}{CW}$

   So, the additive increase rule says that for each ACK (assuming no congestion), we increment CW by MSS^2/CW.

2. If congestion is high, decrease window size drastically.

   We want to relieve the congestion as soon as possible — more packets would increase congestion

   Some people suggest

   $$CW \leftarrow 1 \quad (\text{In TCP Tahoe})$$
   $$CW \leftarrow CW/2 \quad (\text{In TCP Reno}) \quad \text{— Multiplicative decrease}$$

   AIMD stands for Additive Increase Multiplicative Decrease.

3. Initially, set window size small (because we do not know the appropriate data rate) but increase aggressively (additive increase is too slow if we set it too low)

   So, we do exponential increase.

   For each RTT, $CW \leftarrow 2 \times CW$ (initially at least). We use self-clocking again.

   Per received ACK, increase the CW by 1 MSS.
   $$\left(CW + \frac{CW}{MSS} \times MSS = 2 \times CW \right)$$

   This is called slow-start (which is sort of a misnomer).

There is also a certain s.s. threshold. If the s.s. window crosses it, we switch to additive increase This is known as **congestion avoidance.**

Congestion Avoidance
(The s.s. threshold is adjusted after learning more about the network)
(set by admin initially)

(used in Linux)
Nowadays, we use TCP Cubic, which also tests if there is congestion around the s.s. threshold, and increases drastically if not.
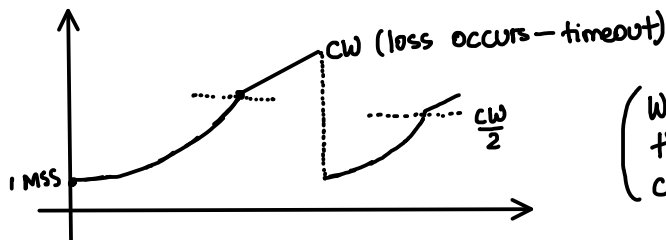
TCP implementations are described in RFC (Request For Comments).

Lecture 26   TCP Tahoe and Reno
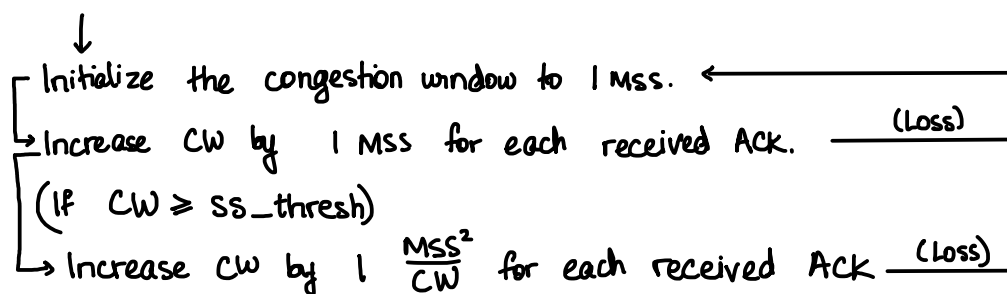
We first look at TCP Tahoe.

As expected, we start with a low congestion window.
We do slow-start, and switch to AI after crossing the threshold. If at some point we detect packet loss — there is no reply within the retransmission timeout, the CW is dropped to 1 MSS and the s.s. threshold is set to max $\{\frac{Window}{2}, 2 \times MSS\}$
$\quad\quad$ ↳ not CW!
(we only use timeout to detect loss in Tahoe)

CW (loss occurs — timeout)
$\frac{CW}{2}$
1 MSS

(We halve it because the threshold may have been crossed when in s.s.)

The rules are:

↓
┌ Initialize the congestion window to 1 MSS. ←
└→ Increase CW by 1 MSS for each received ACK. —— (Loss)
│ (If CW ≥ ss_thresh)
└→ Increase CW by 1 $\frac{MSS^2}{CW}$ for each received ACK —— (Loss)

When there are multiple sources, it is possible that one source faces packet loss and another doesn't (depending on the queue).
It is very difficult to predict exactly what would happen in general.

Suppose some webpage is loading slowly, so we stop and reload, and the page loads much faster. What happens? This may be because the s.s. threshold is set too low after some packets are lost, and additive increase slows everything down tremendously. When we stop and reload, the s.s. threshold is reset. If we are not unlucky and many losses do not occur this time around, the page may load faster.

There are two questions:
1. Do we have to be so aggressive by setting CW = 1 MSS after a loss?
2. Can we detect loss in other ways?

For the second, there are a couple of enhancements (implemented in TCP Reno).

1. **Fast retransmit**: Suppose we send out 6 segments and all but the third are successfully transmitted. We then get 3 duplicate ACKs (for segments 4,5,6). The fast retransmit rule is that if we receive 3 duplicate ACKs for a particular segment, we retransmit it (and disable the old timer).
   (Triple dup. ACK $\Rightarrow$ loss inferred)

2. **Fast recovery**: What do we do when loss is inferred from triple duplicate ACK? We decrease ss_thresh to $\max\{\frac{Window}{2}, 2 \times MSS\}$ like earlier. The difference here is that we set the CW to ss_thresh (and not 1 MSS like earlier). Triple duplicate ACK means that the congestion is not so severe that no segments are getting through.
   Also, the timer is started afresh (we disable the old timer).
   So, timeout loss is considered far worse than dup. ACK loss.

Lecture 27   TCP Vegas

RFC 5681 and RFC 6218 look at congestion control and the timer respectively. Suppose that in slow start, we receive an ACK which acknowledges N bytes of new data. We then increment the CW by $\min\{N, MSS\}$ — not every segment needs to be MSS bytes long.
This also protects against some attacks (like sending many ACKs for a single segment — splitting one ACK into many)

In congestion avoidance, for every Ack which acknowledges new data, we increment the window by MSS^2/CW (we do not take the number of Acked bytes into account).

The initial RTD is usually set to 1s or higher. When loss occurs, we also back off by doing $RTD = \min \{ 2 * RTD, \max\_RTD \}$.

Now, let us look at TCP Vegas.

It uses RTT to detect congestion. Is this a good idea? Congestion is not the only source of high RTT.

Something like VoIP (on UDP) is latency-sensitive, whereas TCP is built to fill up queues. This is the idea behind Vegas — waiting for loss to occur might affect other applications.

First of all, it has Reno's rules and detects loss using timeout and triple dup. ACK, modifying CW and SS_thresh. The SS rules are the same as well.

Congestion avoidance is slightly different.

Denote the minimum RTT (when queues are not a problem) by BaseRTT.

If the queues are empty, the expected rate is $CW/BaseRTT = ExptRate$

The actual rate is $CW/RTT \leq ExptRate$

            ↓
           We smooth out the recent RTTs, similar to
           Estim. RTT earlier.

Letting $Diff = ExptRate - ActualRate$,

in Congestion avoidance ($CW \geq SS\_thresh$)

    $Diff. < \alpha \Rightarrow CW$ is increased by 1 MSS per RTT
                        (Eg. CW increases by MSS^2/CW)
           ↳This is like in Reno.

    $Diff. > \beta \Rightarrow CW$ is decreased by 1 MSS per RTT.
                (Additive decrease because congestion is high)

    $\alpha < Diff. < \beta \Rightarrow CW$ is unchanged.
                (Congestion is neither too high nor too low)

What should α and β be set to? We also want compatibility with Reno, etc. Due to this, Vegas never really became popular.

We have Diff. $= \dfrac{CW}{Base\ RTT} - \dfrac{CW}{RTT}$

$$= CW \left(\underbrace{\dfrac{RTT - Base\ RTT}{Base\ RTT \times RTT}}_{} \right) \rightarrow \text{queuing delays}$$

$$= \dfrac{(\text{Actual Rate}) \times (\text{Queuing delays})}{Base\ RTT}$$

In a single queue, this would be $\simeq \dfrac{\text{Data in Queue}}{Base\ RTT}$
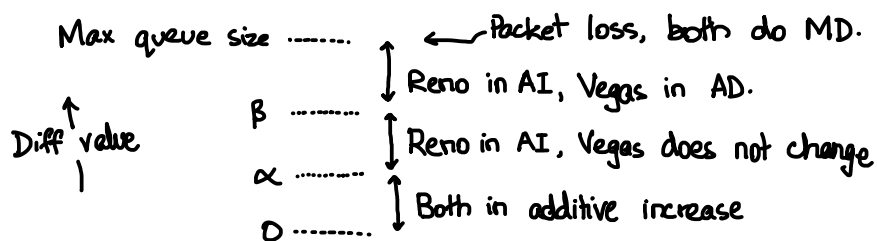
Vegas targets to have $\alpha < Diff < \beta$

$$\alpha \times Base\ RTT < \text{Data in Queue} < \beta \times Base\ RTT$$

The initial proposed values (in the Vegas paper) are $\alpha = 30$ kBps and $\beta = 60$ kBps.
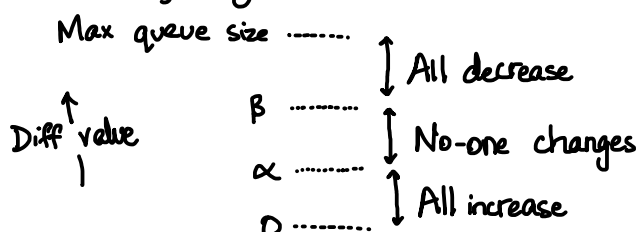
The primary drawback is that we do not know what to set α, β to.

Now, what if Vegas flows compete with Reno flows?
    what if all Reno is replaced with Vegas?

1. Reno v. Vegas


```
Max queue size ------ ↑ ← Packet loss, both do MD.
                      ↓ Reno in AI, Vegas in AD.
           β ------   ↑ Reno in AI, Vegas does not change
Diff value ↑          ↓
           α ------   ↑ Both in additive increase
           0 ------   ↓
```

Vegas' packet loss is faced due to Reno's aggressiveness. If both compete, Reno will dominate Vegas.

2 What if only Vegas?

```
Max queue size ------ ↑ All decrease
           β ------   ↓
Diff value ↑          ↑ No-one changes
           α ------   ↓
           0 ------   ↑ All increase
```

Advantages in this case:
- There is no packet loss.
- The queuing delays remain in the interval corresponding to (α, β)
- The queue is always active (never idle). This is unlike the all Reno case, so throughput may be significantly higher. The Vegas paper claims to have 50% more throughput than Reno

A TCP flow is defined by a 5-tuple.
　　　　　(Source & Dest. IP and port numbers and protocol field)

Nowadays, we primarily use TCP Cubic, which is even more aggressive than Reno.