Lab 6

Fast Reliable File Transfer over TCP/IP

Group 4
Ali Afzal
Zaine Pazooki
Spencer McDonough

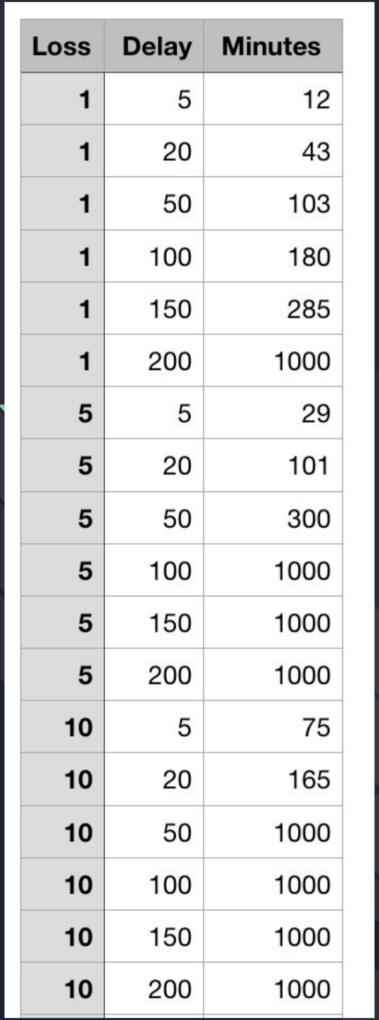
Initial Experiments

scp was slow as expected, 1000 minutes indicates stalled process.

Interestingly, the standard congestion control could not distinguish between loss or delay beyond a certain threshold. At either 15% loss or 200 ms one way delay, the protocol failed. The graphs are presented in the next slide.

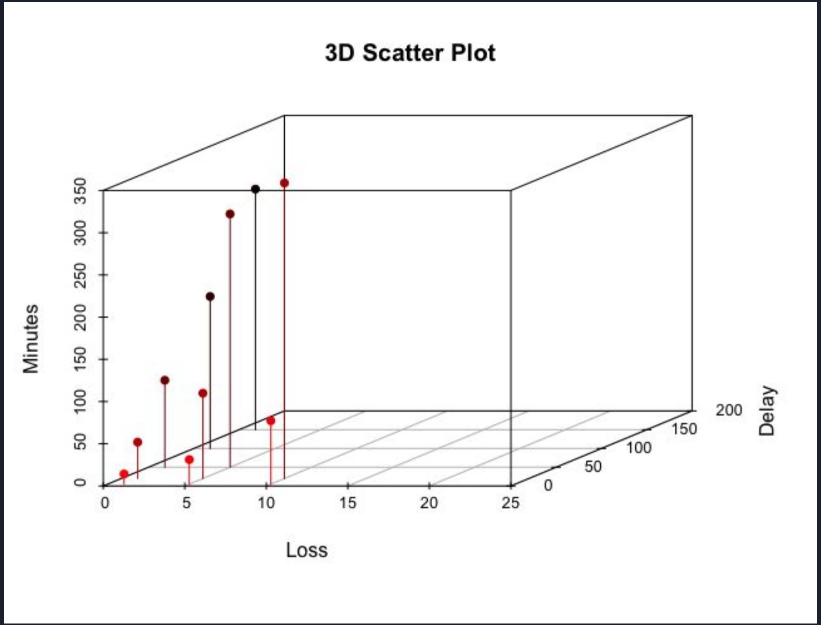
On our tests, by changing the congestion protocol to RENO and Westwood, we were able to at least send the file through, albeit at kilobytes per second speeds.

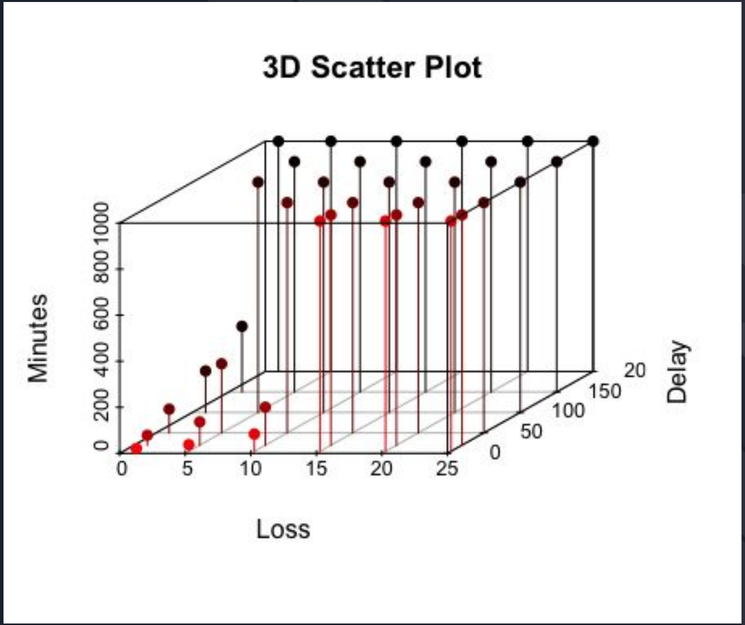
To deal with the high loss on 20% we turned to veno which is optimized for lossy links and wireless networks.



Loss	Delay	Minutes
1	5	12
1	20	43
1	50	103
1	100	180
1	150	285
1	200	1000
5	5	29
5	20	101
5	50	300
5	100	1000
5	150	1000
5	200	1000
10	5	75
10	20	165
10	50	1000
10	100	1000
10	150	1000
10	200	1000

As is evident in the graphs, even without the stalled data points present, the throughput decreases exponentially as the delay or loss increases incrementally.





Testing Different TCP Congestion Implementations

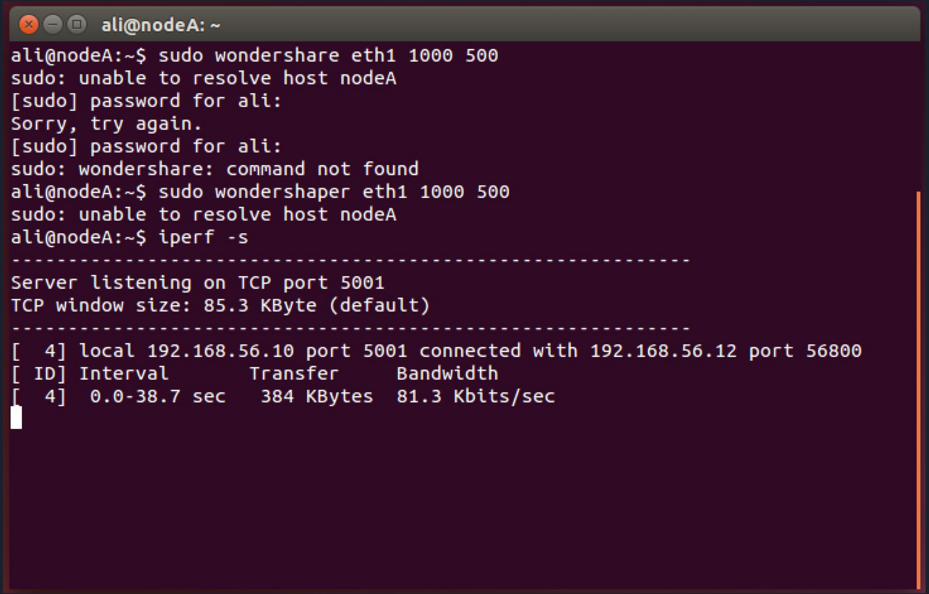
Although we did compile latest kernel on Deter, due to space limitations and the fact that we needed both sides to be modified to the latest stable kernel (because of modifications to the tcp files) we are presenting the results on an emulated on our VMs here, which we will replicate on Deter for the demo.

Set up: we use wondershare for link simulation between two 16.04 ubuntu VMs with a link set up between them.

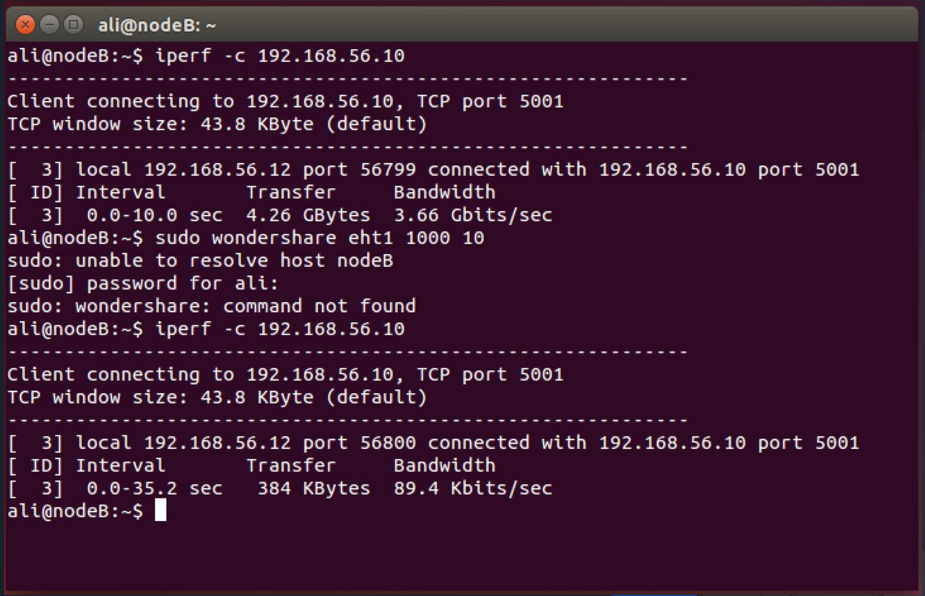
- The screenshots are presented in the next slide

Setting up Emulab on our VMs

100 MB link, 100ms delay, and 20% loss

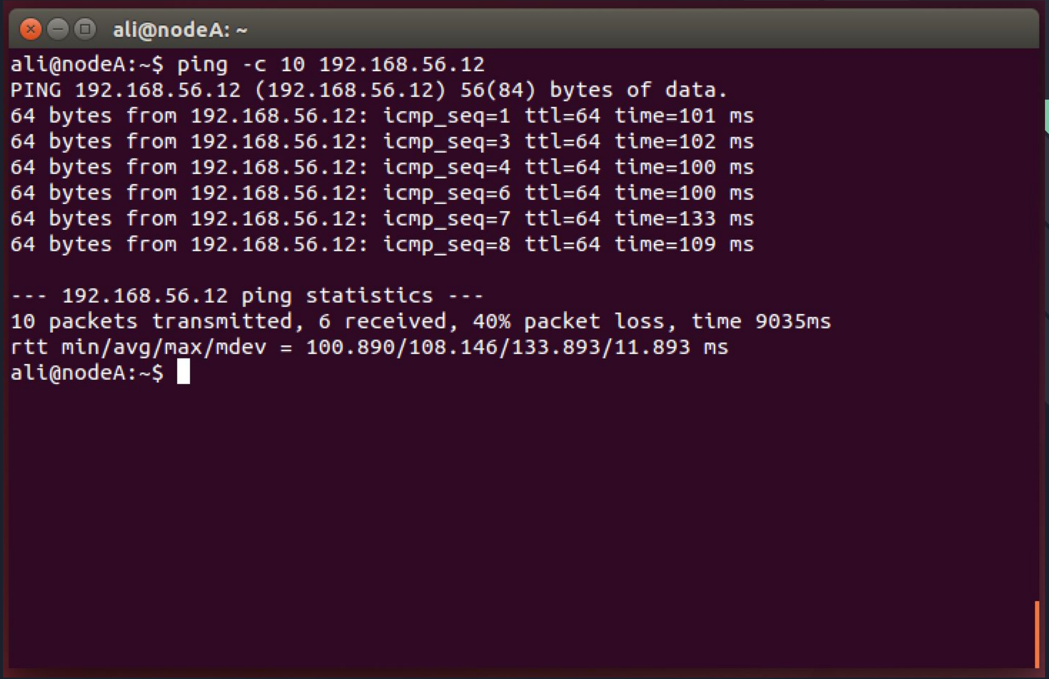


```
ali@nodeA:~$ sudo wondershare eth1 1000 500
sudo: unable to resolve host nodeA
[sudo] password for ali:
Sorry, try again.
[sudo] password for ali:
sudo: wondershare: command not found
ali@nodeA:~$ sudo wondershaper eth1 1000 500
sudo: unable to resolve host nodeA
ali@nodeA:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.56.10 port 5001 connected with 192.168.56.12 port 56800
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-38.7 sec   384 KBytes  81.3 Kbits/sec
```



```
ali@nodeB:~$ iperf -c 192.168.56.10
------------------------------------------------------------
Client connecting to 192.168.56.10, TCP port 5001
TCP window size: 43.8 KByte (default)
------------------------------------------------------------
[  3] local 192.168.56.12 port 56799 connected with 192.168.56.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  4.26 GBytes  3.66 Gbits/sec
ali@nodeB:~$ sudo wondershare eht1 1000 10
sudo: unable to resolve host nodeB
[sudo] password for ali:
sudo: wondershare: command not found
ali@nodeB:~$ iperf -c 192.168.56.10
------------------------------------------------------------
Client connecting to 192.168.56.10, TCP port 5001
TCP window size: 43.8 KByte (default)
------------------------------------------------------------
[  3] local 192.168.56.12 port 56800 connected with 192.168.56.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-35.2 sec   384 KBytes  89.4 Kbits/sec
ali@nodeB:~$
```

Testing the Link

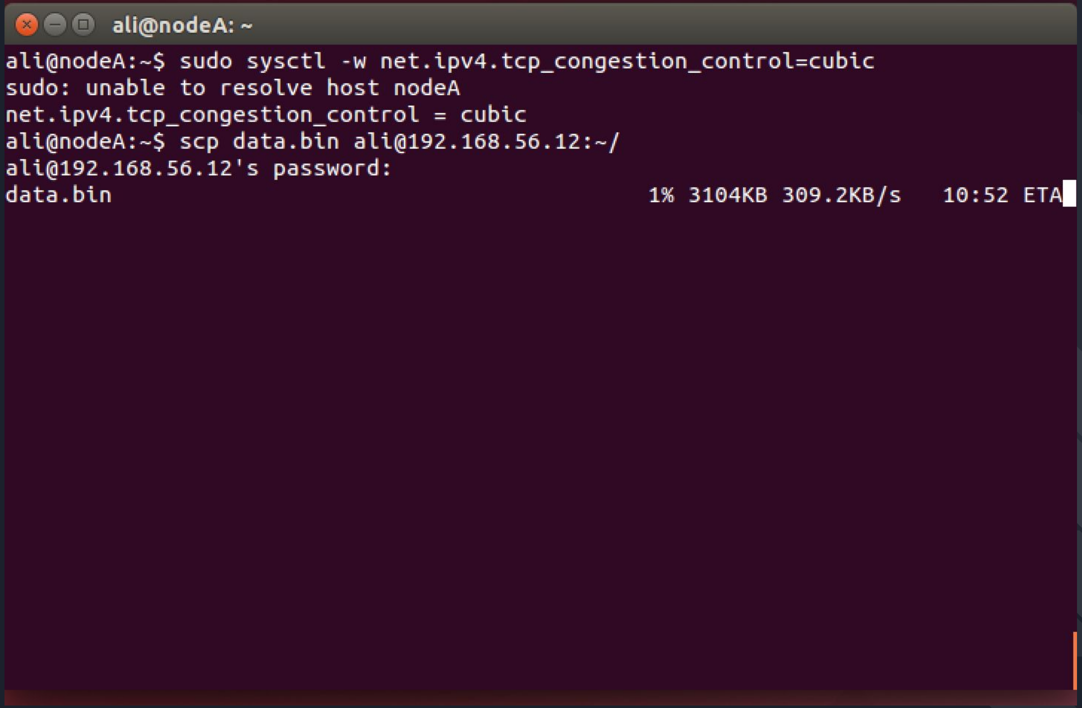


```
ali@nodeA:~$ ping -c 10 192.168.56.12
PING 192.168.56.12 (192.168.56.12) 56(84) bytes of data.
64 bytes from 192.168.56.12: icmp_seq=1 ttl=64 time=101 ms
64 bytes from 192.168.56.12: icmp_seq=3 ttl=64 time=102 ms
64 bytes from 192.168.56.12: icmp_seq=4 ttl=64 time=100 ms
64 bytes from 192.168.56.12: icmp_seq=6 ttl=64 time=100 ms
64 bytes from 192.168.56.12: icmp_seq=7 ttl=64 time=133 ms
64 bytes from 192.168.56.12: icmp_seq=8 ttl=64 time=109 ms

--- 192.168.56.12 ping statistics ---
10 packets transmitted, 6 received, 40% packet loss, time 9035ms
rtt min/avg/max/mdev = 100.890/108.146/133.893/11.893 ms
ali@nodeA:~$
```

Changing TCP Implementation to Cubic

We did test various implementations, but on our links and on deter’s (westwood, reno, vegas, …) but we found cubic to be one of the better ones as mentioned in the SCP test results. The improvement was slight but considerable



```
ali@nodeA:~$ sudo sysctl -w net.ipv4.tcp_congestion_control=cubic
sudo: unable to resolve host nodeA
net.ipv4.tcp_congestion_control = cubic
ali@nodeA:~$ scp data.bin ali@192.168.56.12:~/
ali@192.168.56.12's password:
data.bin                                        1% 3104KB 309.2KB/s   10:52 ETA
```

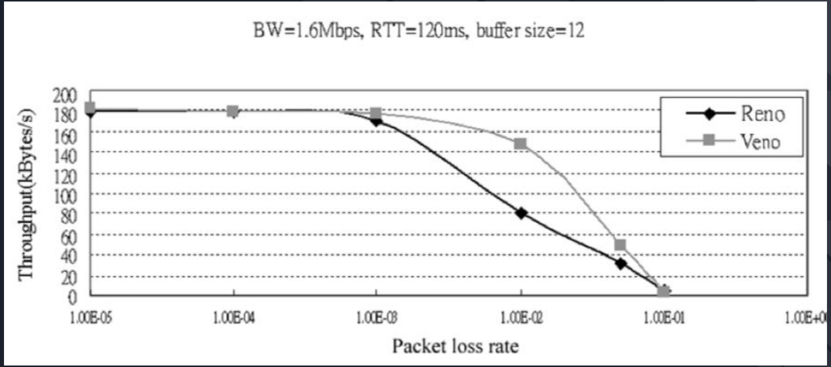
Choosing a TCP Implementation to Modify

Our emulated link, with high delay and high loss, in large part resembles a weak wireless link. According to our research, TCP veno has been optimized for this purpose.

Veno builds on two previous popular implementations: Reno and Vegas, and combines elements from the congestion control mechanisms from both, in particular:

- Slow start from Reno
- Additive Increase from both
- Additive decrease from Vegas

Simply out, Veno is not overreacting to and takes the possibility of packet loss into account; given that it's optimized for wireless access networks.



How TCP Veno Code Works

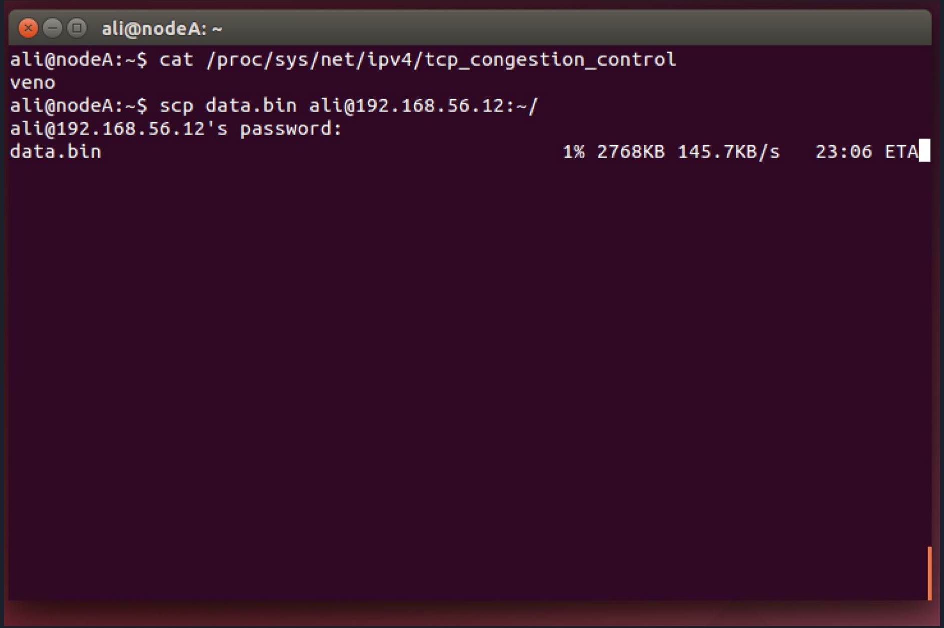
The TCP Veno algorithm was developed for wireless networks where random packet loss due is pervasive.

Monitors network to determine if packets going to be lost due to congestion or bit errors

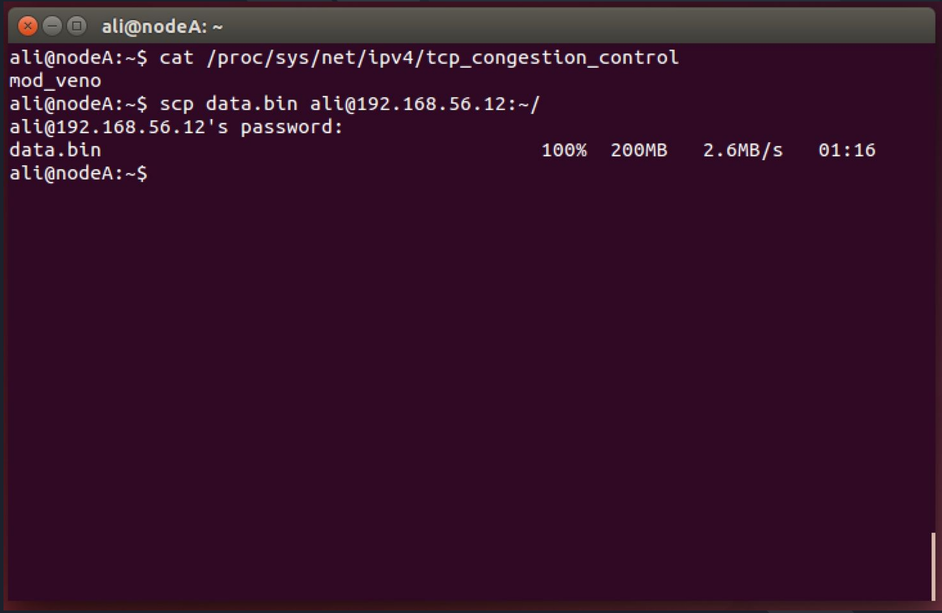
1. Refines multiplicative decrease algorithm in Reno by adjusting slow-start threshold according to perceived network congestions rather than a fixed drop factor.
2. Refines linear increase algorithm to maximize bandwidth utilization

Results for Veno and our Modified Veno

With our knowledge that our our link is lossy and congestion from other nodes will not happen, we modified veno to achieve optimal results for our link



```
ali@nodeA:~$ cat /proc/sys/net/ipv4/tcp_congestion_control
veno
ali@nodeA:~$ scp data.bin ali@192.168.56.12:~/
ali@192.168.56.12's password:
data.bin                                        1% 2768KB 145.7KB/s   23:06 ETA
```



```
ali@nodeA:~$ cat /proc/sys/net/ipv4/tcp_congestion_control
mod_veno
ali@nodeA:~$ scp data.bin ali@192.168.56.12:~/
ali@192.168.56.12's password:
data.bin                                              100%  200MB   2.6MB/s   01:16
ali@nodeA:~$
```

tcp_mod_veno.c

We originally started by attempting to modify the tcp veno algorithm for higher throughput, but we found an aggressive additive increase proved to be more reliable. We maintained the veno modification, as this was our basis, even though the code does not resemble veno in the slightest

We build a simple state machine to change our send window size based on the link state:

1. If the link is open, do nothing
2. If the link is congested, lossy, or disordered, we decrease our window size and jump to the recovery state.
3. We rebuild the window size incrementally in the recovery state

tcp_mod_veno.c

We interpret different tcp congestion avoidance events to jump to our recovery state

1. CA_EVENT_FAST_ACK → Do nothing
2. CA_EVENT_SLOW_ACK → Recovery State
3. CA_EVENT_TX_START → Do nothing
4. CA_EVENT_CWND_RESTART → Recovery State
5. CA_EVENT_COMPLETE_CWR → Recovery State
6. CA_EVENT_LOSS → Recovery State
7. CA_EVENT_FRTO → Recovery State

tcp_mod_veno.c

We dumbed down the congestion avoidance function to provide only an additive increase whenever called.

In conclusion, rather than sample rtt and model our send window, we aggressively recover our window size given almost any TCP Congestion Avoidance event. In addition, by increasing our window size when the congestion avoidance function is called, we are assuming that our link has no congestion and that congestion is being confused for a lost packet. We thus increase our window size until we receive a NACK, at which point we jump down to our base window size value of 5500B.

tcp.h

Modifications to RTO and TIMEOUT Definitions:

Line 137: TCP_RTO_MAX: ((unsigned)(120*HZ)) → ((unsigned)(3*HZ/5))

Line 138: TCP_RTO_MIN: ((unsigned)(HZ/5)) → ((unsigned)(HZ/10))

Line 139: TCP_TIMEOUT_INIT: ((unsigned)(1*HZ)) → ((unsigned)(HZ/5))

Line 225: TCP_INIT_CWND: 10 → 1000

Add new definition:

Line 64: #define TCP_DEFAULT_INIT_RCVWND 1000

tcp_timer.c

Line 517: icsk->icsk_backoff ++; → icsk->icsk_backoff = 0;

This disables exponential backoff.

tcp_output.c

Line 3564, 3574: Added 2nd ack buffer so we can send twice the amount of data in one go

```
          struct sk_buff *s_buff;

          buff = alloc_skb(MAX_TCP_HEADER,sk_gfp_mask(sk, GFP_ATOMIC |
__GFP_NOWARN));

          s_buff = alloc_skb(MAX_TCP_HEADER,sk_gfp_mask(sk, GFP_ATOMIC |
__GFP_NOWARN));
```

tcp_output.c

We repeat this process (inserting a 2nd ack buffer) for header space reservation, control bit preparation, and actual transmission.

Lines 3574 - 3599:

```
skb_reserve(s_buff, MAX_TCP_HEADER);

 tcp_init_nondata_skb(s_buff, tcp_acceptable_seq(sk), TCPHDR_ACK);

skb_set_tcp_pure_ack(s_buff);
/* Send it off, this clears delayed acks for us. */

__tcp_transmit_skb(sk, s_buff, 0, (__force gfp_t)0, rcv_nxt)
```