Validating the ns3 implementation of TCP Vegas

Team Members:

Hardik Rana

(16CO138)

Shushant Kumar

(16CO143)

Anmol Horo

(16CO206)

TCP Vegas

TCP Vegas is a delay based congestion control algorithm which measures per packet RTT to infer the state of congestion in the network. It uses Additive Increase Additive Decrease (AIAD) algorithm to adjust its cwnd. In this project, the aim is to validate ns-3 TCP Vegas implementation by comparing the results obtained from it to those obtained by simulating Linux TCP Vegas.

Tcp Vegas is actually a modification of TCP reno [all changes are only in sending side],but it provides 37 to 71% better throughput with less losses compared to TCP reno.In TCP reno RTT is computed using coarse-grained timer but in TCP vegas RTT is computed using fine-grained timer.

Three techniques that Vegas employs to increase throughput and decrease losses

1. New retransmission Mechanism
2. Congestion Avoidance Mechanism
3. Modified slow-start Mechanism

1.New retransmission Mechanism

Reno uses two mechanisms to detect packet loss - one is Fast Retransmit and the other is timeout. According to Fast Retransmit whenever we get 3 duplicate acknowledgement then that means a packet loss, so we need to retransmit the relevant packet and enter fast recovery. While in case of timeout, in addition to retransmission of packet we reset the cwnd value and enter slow-start mechanism.

Like Reno, Vegas adapts the same behavior in case of timeout but instead of Fast Retransmit it follows a entirely different approach. Vegas can retransmit a packet after getting the first or second duplicate ACK instead of waiting for the third duplicate ACK. Vegas records the clock time when a particular packet is sent. So, whenever a duplicate ACK is received, vegas calculates the difference between the current time (the time when a duplicate ACK is received) and the time when the relevant packet is sent, and the packet is retransmitted if this difference is greater than the timeout value for that particular packet. After a retransmission whenever a first or second new ACK is received, vegas again checks if the time since the packets are sent have become greater than the timeout value then retransmit those packets.

2. Congestion Avoidance Mechanism

Let BaseRTT to be the RTT of a segment when the connection is not congested. [commonly it is the RTT of the first segment sent by the connection].If we assume that we are not overflowing the connection, then the expected throughput is given by:

Expected = WindowSize/BaseRTT ,where the windowSize is the size of current congestion window [equal to number of bytes in transmit].

Vegas also calculates the current Actual sending rate.This is done by recording the sending time for a distinguished segment, recording how many bytes are transmitted between the time that segment is sent and its acknowledgment is received, computing the RTT for the distinguished segment when its acknowledgment arrives, and dividing the number of bytes transmitted by the sample RTT. This calculation is done once per round-trip time.

By comparing Expected with Actual sending rate vegas will adjust it's window.We will have **Diff = Expected - Actual**,where the Diff is positive or zero by definition.Also we define two thresholds, $\alpha < \beta$ roughly corresponding to having too little and too much extra data in the network, respectively. When **Diff < α** , Vegas increases the congestion window linearly during the next RTT, and when **Diff > β** ,Vegas decreases the congestion window linearly during the next RTT. Vegas leaves the congestion window unchanged when $\alpha <$ Diff $< \beta$.

3.Modified Slow-Start Mechanism

In slow-start strategy the congestion window size is doubled every RTT. Vegas improves the slow-start strategy by reducing the doubling of the congestion window size to every two RTT. After every intermediate RTT, it compares expected and actual sending rates. If the rates decrease by a router buffer, then Vegas stops the exponential increase and uses linear increase in congestion window size. This strategy can still make sure that as much bandwidth as possible is used, but not too much that the network becomes congested so quickly.

Configuration

- **Configuration1 [Initial]**

  Sender to router, router to router and router to receiver bandwidth = 150Mbps

  Fack disabled

  Results : The cwnd graphs followed the same pattern for both linux and ns3 were matching in the stable state. The Queue traces generated for ns3 simulation gave zero queue size build up. For linux simulation queue traces value were obtained.

- **Configuration2**

  Sender to router and router to receiver bandwidth = 10Mbps

  Router1 to Router2 bandwidth = 1 Mbps

  Fack disabled

  Reason: 150 Mbps is a very large value and 5 senders with TCP as TCP Vegas are not enough to create congestion and also TCP Vegas will not give the expected behaviour. That configuration will work for Data Center TCP but not for TCP Vegas.

  Results: The queue traces were zero in both but cwnd for ns3 became constant within 1 second whereas in linux it reached the stable state only after 6 seconds.

- **Configuration3**

  Router1 to Router2: Bandwidth = 1Mbps

  Delay = 50ms

  End Hosts to Router: Bandwidth = 10Mbps

Delay = Same as before

Fack disabled

Results: In this configuration the queue length for both the stacks is almost 0 and initial behavior of congestion window is also overlapping.

- **Configuration4**

  Router1 to Router2: Bandwidth = 200Kbps

  Delay = 50ms

  End Hosts to Router: Bandwidth = 10Mbps

  Delay = Same as before

  Fack disabled

  Results: The value of queue traces were generated in linux but in ns3 it still remained zero. The graphs also did not overlap with each other and were too different in terms of pattern and magnitude.

- **Configuration5**

  Router1 to Router2: Bandwidth = 200Kbps

  Delay = 50ms

  End Hosts to Router: Bandwidth = 1Mbps

  Delay = Same as before

  DSACK,Fack disabled

  Results: The value of queue traces were generated in linux at later stage stable to 70.

- **Configuration6**

Here we changed value of diff in ns3 implementation of TCP Vegas by referring to the linux implementation of TCP Vegas.

Sender to router, router to router and router to receiver bandwidth = 150Mbps

Fack disabled

Results : The cwnd graphs followed the same pattern for both linux and ns3 were matching in the stable state. The Queue traces generated for ns3 simulation gave zero queue size build up. For linux simulation queue traces value were obtained. The results did not refer much compared to previous results in configuration 1.

- **Configuration7**

Here also we changed value of diff in ns3 implementation of TCP Vegas by referring to the linux implementation of TCP Vegas.

Router1 to Router2: Bandwidth = 1Mbps

Delay = 50ms

End Hosts to Router: Bandwidth = 10Mbps

Delay = Same as before

Fack disabled

Results: In this configuration the queue length for both the stacks is almost 0 and initial behavior of congestion window is also overlapping.

Differences Found

1. Modified Slow Start of the TCP Vegas not implemented both in linux and ns3

   According to the TCP vegas, in slow start the congestion window increases every other RTT but it is not followed in either linux or ns3 implementation of TCP Vegas. In both case, the congestion window increases in every RTT just like Reno.

2. The the way value of diff is calculated in Linux implementation of TCP Vegas is a little different than the ns3.

   Linux code for diff calculation :-

```
 rtt = vegas->minRTT;
 target_cwnd = (u64)tp->snd_cwnd * vegas->baseRTT;
 do_div(target_cwnd, rtt);
diff = tp->snd_cwnd * (rtt - vegas->baseRTT) / vegas->baseRTT;
```

   ns3 code for diff calculation :-

```
double tmp = m_baseRtt.GetSeconds () / m_minRtt.GetSeconds ();

 targetCwnd = static_cast<uint32_t> (segCwnd * tmp);

 diff = segCwnd - targetCwnd;
```

In case of linux implementation of TCP Vegas, the value of diff by referring to the linux code can be written as ,

diff = tp->snd_cwnd * (vegas->minRTT - vegas->baseRTT) / vegas->baseRTT;

While in ns3 implementation of TCP Vegas, the value of diff can be inferred as,

diff= segCwnd * (m_minRtt.GetSeconds () - m_baseRtt.GetSeconds ()) / m_minRtt.GetSeconds ()

The denominator value of diff in both linux and ns3 are different in one case it is baseRTT while in other it is minRTT.

3. In order to apply vegas congestion control algorithm the condition checked in linux and ns3 are as given below :

   ns3 => if (tcb->m_lastAckedSeq >= m_begSndNxt)

   Linux => if (after(ack, vegas->beg_snd_nxt))

   after function in linux reduces to vegas->beg_snd_nxt - ack < 0

   While in ns3 if has condition m_begSndNxt - tcb->m_lastAckedSeq <= 0

4. Linux implementation contained given below chunk of code whose equivalent code was not found in ns3 implementation of TCP Vegas.

   The line 271 of the Linux implementation of TCP Vegas contains the following lines of code:

```
if (tp->snd_cwnd < 2)
        tp->snd_cwnd = 2;
else if (tp->snd_cwnd > tp->snd_cwnd_clamp)
        tp->snd_cwnd = tp->snd_cwnd_clamp;
```

   The equivalent of the above mentioned piece of code in ns3 implementation of TCP vegas was missing.