Name: Calvin Deutschbein

TCP Congestion Project Report

Introduction:

The goal of this project was to design, test, and evaluate a kernel modification or extension. To that end, the TCP Congestion framework in Linux systems is well suited for modification. Historically, as the networking services were used more and more extensively leading up to the modern use case of the internet, TCP Congestion went from a near impossibility to an everyday reality of networking. There have been many proposed methods for congestion control given the different nature of various networks and the relation of different endpoints to the overall network.

One such TCP Congestion module is known as Vegas. Unlike many other protocols that increase network load until packets start dropping, Vegas rather measures the time it takes for packets to be acknowledged and throttles connection speed when times start increasing significantly as this may be a sign of heavy network load. In a perfect world, this would allow Vegas to reach maximum network load similarly to other TCP Congestion modules, but without overshooting to the point of dropping packets and therefore have overall higher performance.

In practice, Vegas does not seem to perform particularly well. It has been suggested that because Vegas throttles prior to packet drop and other methods, most notably the immensely popular Reno (currently the default in most systems), systems using Vegas continuously cede network usage to other systems more willing to cause packet loss. To that end, this project attempts to modify the Vegas protocol in order to more aggressively utilize the network, while still attempting to prevent consistent packet loss and preserving other positive features of existing TCP Congestion modules, such as responsiveness to changing network conditions.

Modules Description:

The Vegas module operates in the same paradigm as other congestion controls. It first becomes active when packets need to be sent over a network connection but there is no knowledge of the bandwidth of the connection. Like Reno and other protocols, it begins in a state ironically called slow-start which is used to quickly increase transmission speed up to a certain threshold. Slow-start begins with a certain window size in packets and increases this window size by one packet each time a packet is acknowledged, such that the window size increases exponentially until the pre-defined threshold is reached.

As the goal of the changes to this module is to increase the aggression with which the Vegas module exercises network bandwidth, slow-start, while already highly aggressive, was targeted. By default, in the Vegas module, slow-start is used up to the slow-start threshold as it would in any other module. However, in the modified Vegas module, slow-start is bypassed as aggressively as possible. While slow-start is still used while Vegas is gathering round-trip time data and the Vegas algorithm is not yet online but rather still acting as Reno, in the modified kernel as soon as Vegas comes live it exits slow-start mode and immediately sets window size to the predicted optimal window, rather than taking more time in slow-start before switching to Vegas specific implementation. This is the first major change made to the module.

After the slow-start threshold is reached, the Vegas specific aspects of the protocol come into play. Do note that the entire time the connection has been live, Vegas has been tracking round-trip times for all packets. In the Vegas congestion avoidance algorithm, this round trip time is then used to calculate the desired speed with which packets are sent. After enough (usually 2) packets have been acknowledged that the algorithm may be confident round-trip times are not the result of a delayed packet, the minimum round-trip time (selected instead of an average or some other measure of central tendency in order to again avoid undue influence of delayed packets) is taken over the past two round-trip time units of time, to get a current estimate for the fastest rate of transmission the network in its present condition may tolerate.

The Vegas protocol, as presently implemented, acts as Reno in the case that there are not enough round-trip times to evaluate, and otherwise computes a predicted optimal network transmission speed by defining a window of appropriate size. The current window size is compared against this predicted optimal size in terms of packets. Updates are then performed additively rather than exponentially (as they had been in slow-start) to dial in more precisely on an optimal window size.

In its implementation, this window size difference represents the number of packets expected to be live along the connection, with at least one packet present in the buffer on the bottle-neck router in the connection. The difference is compared against various values and processing flow branches accordingly.

If there are expected to be fewer than alpha (by default 2) packets live on the network, the current window is set to increase by one toward the predicted optimal window. In Vegas, this occurs after slow-start has completed and Vegas protocol runs the connection. In the modified module, this occurs as soon as two round-trips have occurred.

If there are fewer than beta (by default 4) packets live on the network, the current window is set to increase by one toward the predicted optimal window. This occurs occurs at the same stage in processing flow as the alpha comparison. Any additional packets remaining on network are, in Vegas, taken as a sign of imminent packet loss.

In Vegas default, if the number of packets live on the network is between alpha and beta, no changes are taken as any number of packets between these two values are taken to the target. In modified Vegas, exactly beta (rather than between alpha and beta) is targeted. Here the modified implementation takes advantage of the higher computational capabilities of modern hardware to probabilistically increase toward beta in proportion to the distance from beta (as load must increase discretely in packet sized units). This effectively allows the protocol to maintain bandwidth demand more aggressively while in its desired range without triggering a high degree of packet loss by deterministically increasing window size on a regular time interval. The original protocol did not have the luxury of such implementation because it had to more carefully consider overhead (especially compared to the even more efficient Reno) but for the sake of this project there was little compelling reason not to leverage this capability.

In Vegas default, beta and alpha are desired to be at least two packets apart to avoid constantly having the window size in flux. In this implementation, given the introduction of a window size changes while in between alpha and beta, this range is expanded. Alpha is reduced to the minimum desirable value of one. This is the minimum because, in the event network load decreases and more bandwidth becomes available for the connection, with at least one packet on the bottleneck router's buffer there will be an immediate increase in transmission speed as the packets on network will meet reach the destination sooner. Beta is increased to six as a reasonable lower value for the minimum number of hops in a connection according to a number of networking community's forums.

While the difference is between alpha and beta, a random integer in the range [0,63] is rolled. This pseudo-random number is generated by the following code, included in tcp_vegas_cong_avoid, which is far from truly random but close enough for the purposes of this module.

```
int rand = 0;
rand = vegas→minRTT * (int)vegas * tp->snd_una;
rand = rand % 63;
```

At the very least, this should produce a different value in the range for each iteration of the module that is run, as well as for each round trip (as it is unlikely for the minRTT and snd_una to be exactly the same across multiple time frames). More over, it has a relatively low computational cost.

Then, diff is converted to a comparison value that then increases the window size at a probability that declines as it increases from alpha to beta. That is, the expected increase of the window size in packets per iteration is the value in third column. These values are not meant to be optimal, but simply chosen for testing purposes.

n	2^(6-n)	p(2^(6-n) > rand)	
	1	32	.500000
	2	16	.250000
	3	8	.125000
	4	4	.062500
	5	2	.031250
	6	1	.015625

The objective of this arrangement is that the somewhat conservative window increase will provide some degree of increased bandwidth demand to more favorably compete with Reno systems without as recklessly courting packet loss as Reno does.

Methodology:

All design, testing, and evaluation was performed on the provided virtual machine without any other user processes running. Testing was done using the iPerf network bandwidth measurement tool and network conditions were simulated using netem. The netem framework was used to provide packet loss and delay and the exact specifications of these changes will be noted in the results section.

Testing was performed in between the Vegas module as implemented in the base operating system and redesigned version based on the tcp_vegas.* files located at ~/linux-3.13.0/net/ipv4/tcp_vegas.* on the provided virtual machine, as well as the Reno implemented as system default.

As the goal of this module was to modify Vegas to be more competitive with Reno in realistic scenarios, Vegas, Reno, and modified Vegas are all tested over a variety of network conditions up to the most nearly realistic situation achievable under this controlled testing framework.

Testing was performed between two (identical) virtual machines both hosted on the same physical machine. The server used standard linux network setup and and only served to host the iPerf server process. The server was modified with netem to simulate different network conditions and then the client ran iPerf tests over the three different module types.

Each module was tested with iPerf five times each over the different emulated network conditions. Network speeds for each run as well as average speed and standard deviation were all recorded in the Results section.

The two control modules, Reno and default Vegas, were included in the kernel provided. Modified Vegas was included into the kernel with the “insmod” command.

The three testing modules were then made eligible for usage with the following command, the cubic module left valid as per linux kernel modification best practice:

```
% sudo sysctl -w net.ipv4.tcp_allowed_congestion_control="cubic reno vegas vegas2"
```

Note that modified Vegas had its name parameter set to be “vegas2” to differentiate itself from default vegas. The specific module to use for a series of tests was then set with the following command:

```
% sudo sysctl -w net.ipv4.tcp_congestion_control="<name>"
```

On the server, iPerf was run with no modifications as “iperf -s” and left running in the background for the duration of testing.

On the client, iPerf was run with no modifications as “iperf -c 192.168.188.130” which gives the servers ip address on the local network.

Netem was set using the following command on the server, where <netem options> is given in the table in results:

```
% sudo tc qdisc <add/change/delete> dev eth0 root netem <netem options>
```

Testing began with existing network conditions and then proceeded to attempt to simulate more realistic network conditions than two virtual machines on the same physical machine.

Results:

Before presenting tables, a quick note on the netem options shown in Trial 6. Trial 6 is the closest possible approximation of standard realistic networking conditions.

Its options were as follows:

```
netem delay 100ms 10ms 25% loss 0.1%
```

This means that netem is emulating a delay of 100 milliseconds but deviating from it by up to 10 milliseconds, with 25% of the delay deviation determined by the delay of the previous packet. The idea behind this specification is that this more accurately simulates realistic network conditions in which the network state is conditional on previous network state.

Netem also silently drops 0.1% of packets. The other netem options are similar but less complicated and will be included without further discussion. For more information on netem and its options, please consult
http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

The following tables present the bandwidth speeds for recorded for five tests of each module for each trial, with trials differentiated by changed network conditions from netem.

The bandwidth speeds are as reported by the iPerf tool. The tool reports three significant figures, so take only the first three digits to be significant.

Values were aligned with trailing zeros for greater visual clarity.

Values as sorted by ascended bandwidth speed such that the first value is the minimum, the fifth value is the maximum, and third value is the median in all cases.

Trial 1		Reno	Vegas	mVegas
	1	288.00	285.00	91.00
	2	290.00	287.00	92.30
	3	291.00	289.00	95.60
	4	291.00	291.00	96.40
	5	291.00	291.00	97.20
ave		290.00	289.00	94.50
stddev		1.300	2.610	2.700
netem options	no netem			

Trial 2		Reno	Vegas	mVegas
	1	296.00	297.00	35.30
	2	299.00	298.00	37.50
	3	300.00	300.00	38.20
	4	302.00	302.00	44.10
	5	302.00	304.00	47.00
ave		300.00	300.00	40.40
stddev		2.490	2.860	4.910
netem options	delay 10ms			

Trial 3	Reno	Vegas	mVegas	
	1	11.20	7.24	7.10
	2	11.20	7.42	7.35
	3	11.40	7.43	7.46
	4	11.40	7.45	7.46
	5	11.70	7.53	7.49
ave		11.40	7.41	7.37
stddev		0.204	0.106	0.161
netem options	delay 100ms			

Trial 4	Reno	Vegas	mVegas	
	1	286.00	286.00	90.30
	2	288.00	287.00	91.90
	3	288.00	288.00	91.90
	4	289.00	289.00	92.40
	5	290.00	291.00	102.00
ave		288.00	288.00	93.70
stddev		1.480	1.920	4.710
netem options	loss 0.1%			

Trial 5	Reno	Vegas	mVegas	
	1	286.00	87.60	90.00
	2	286.00	90.30	90.30
	3	288.00	93.40	91.50
	4	290.00	95.90	91.70
	5	290.00	96.50	103.00
ave		288.00	92.70	93.30
stddev		2.000	3.770	5.470
netem options	loss 1.0%			

Trial 6	Reno	Vegas	mVegas	
	1	10.10	7.06	7.62
	2	10.70	7.55	7.80
	3	10.80	8.08	7.98
	4	10.80	9.04	9.86
	5	10.90	10.50	18.10
ave		10.70	8.450	10.30
stddev		0.321	1.360	4.470
netem options	delay 100ms 10ms 25% loss 0.1%			

Discussion:

In Trials 1, 2, and 4 performance from modified Vegas is significantly worse than that of Vegas. These three trials also all happen to be the least realistic, displaying unrealistically low delay as well as either no packet loss or very low packet loss coupled with no delay. For a performance delta this great the only reasonable conclusion seems to be that skipping the slow start stage and jumping directly to predicted optimal window size has resulted in a severely inaccurate estimation that either resulted in massive packet loss (less likely) or a catastrophically undersized window (more likely) that could only be improved additively rather than exponentially. While it is impossible to say with a high degree of confidence without further research, there are no other obvious explanations for performance this poor.

Somewhat curiously, modified Vegas performed relatively more poorly to both its controls in Trial 2 with a shorter network delay, suggesting that it performs poorly under delay conditions, but when the delay was increased in Trial 3 modified Vegas became much more competitive with no statistically significant difference from default Vegas and moving within a factor of two of Reno. While Reno still competes very favorably under these conditions, this can likely be attributed more to Reno's ability to increase the window size more aggressively coming into play over a testing of relatively small duration compared to network delay. This also more closely approximates everyday internet usage (especially outside of urban areas) and forms a promising sign for the competitive nature of modified Vegas.

In both Trial 5 and 6, modified Vegas outperforms default Vegas, though only statistically significantly in Trial 6 and at that only if the (most extreme of the entire dataset) outlier is included. As both Vegas protocols are much worse than Reno in Trial 5, this test demonstrates that under at least one major case (high packet loss) of network conditions, modified Vegas fails to address the problems in Vegas that lead to its unfavorable comparison with Reno. However, Trial 6 shows that in realistic conditions (Author's note: These the log order approximations of network performance I experience on a standard daily work load) modified Vegas is highly competitive with Reno and surpasses default Vegas, though Reno boasts a much lower standard deviation. In these conditions, the randomized nature of modified Vegas is likely coming into play, given the relatively lower number of packets being transmitted and the relatively high portion of packet drops. Nevertheless, this trial makes a reasonably strong case for modified Vegas as a credible alternative to Reno under specific network conditions, especially if the ideas of this module may be carried further by more experienced and capable network scientists.

Contributions and Conclusion:

While modified Vegas far from exceeds Vegas in performance, much less challenging the well established Reno protocol, this project does successfully demonstrate that modifications to the Vegas protocol can produce favorable results. Specifically, it demonstrates that the randomized windows incrementation scheme is at least functional under a close approximation of a real network load, opening up a variety of options for applying non-discrete size changes to window size in TCP Congestion Control modules, as well as possibly other parts of the kernel. It also provides further evidence to the well established fact that slow-start is a very powerful and important part of the TCP protocol.