Adventures in TCP/IP

Dynamic Goodput Pacing: A New Approach to Packet Pacing

BY RANDALL STEWART

The previous column in this series focused on the FreeBSD infrastructure that supports pacing for TCP stacks. This column continues exploring pacing in FreeBSD by discussing a pacing methodology that is available in the RACK stack today in the developer version of FreeBSD. This pacing methodology is called Dynamic Goodput Pacing (DGP) and represents a new form of pacing that can provide good performance and yet still be fair in the network. To understand DGP, we first will need to discuss congestion control, since DGP works by combining two forms of congestion control that traditionally have not been used together. Consequently, this column will first discuss what congestion control is as well as two kinds of congestion control that DGP combines into a seamless pacing regime.

Congestion Control

When TCP was first introduced to the budding Internet, it did not contain anything called congestion control. It had flow control, i.e., making sure that a sender did not overrun a receiver, but there was no regard at all to what TCP was doing to the network. This caused a series of outages that have since been termed "congestion collapse" and brought about changes to TCP to have a "network aware" component to try to assure that actions by TCP would not cause problems on the Internet. This "network aware" component is called congestion control.

When TCP was first introduced to the budding Internet, it did not contain anything called congestion control.

Loss Based

The very first congestion control introduced to the Internet was loss-based congestion control. Today it is one of the most widely deployed forms of congestion control though it does have its downsides. There are two main algorithms (though others do exist) used in loss-based congestion — one called New Reno[1] and the other called Cubic[2]. New Reno and Cubic both share one fundamental design, Additive Increase and Multiplicative Decrease (AIMD). We will look a bit more closely at New Reno, since it is simpler to understand.

TCP will start with a number of basic variables set to preset defaults:

- **Congestion Window (cwnd)** — How much data that can be sent into the network without causing congestion. This is initialized to the value of the Initial Window.

- **Slow Start Threshold (ssthresh)** — When the cwnd reaches this point the increase mechanism is slowed down from something called "Slow Start" to "Congestion Avoidance". The ssthresh value is ostensibly set to infinity initially but will get set to ½ the current cwnd whenever a loss is detected.
- **Flight Size (FS)** — The number of data bytes in flight to the peer that has not been acknowledged. This of course starts at zero and is incremented every time data is sent and subtracted from when data is cumulatively acknowledged.
- **Initial Window (IW)** — This is the initial value to be set into the cwnd, in most implementations it is set to 10 segments (10 x 1460), but it may be more or less (initially TCP had this value set to 1 segment).
- **Algorithm – "Slow Start" (SS)** — The slow start algorithm is one of the algorithms used for the additive increase part. In slow start every time an acknowledgment arrives the cwnd is increased by the amount of data acknowledged.
- **Algorithm – "Congestion Avoidance" (CA)** — The congestion avoidance algorithm will increase the cwnd 1 packet every time a full congestion windows worth of data has been acknowledged.

The implementation will send out the IW worth of data towards its peer moving FS to the IW size as well.

So initially the cwnd is set to the IW, the increase algorithm is set to SS, and the flight size is set to 0 bytes. The implementation, assuming an infinite amount of data to send, will send out the IW worth of data towards its peer moving FS to the IW size as well. The peer will send back acknowledgments for every other packet. (Some implementations such as macOS may change to every eighth packet or every single packet.) This means that with each arriving acknowledgement the FS will go down by two packets and the cwnd will be increased by two packets, which means we can send out four more packets (if the flight size before the acknowledgment arrived was at its maximum value i.e. the cwnd). This sequence will continue until a loss is detected.

There are two ways in which we can detect loss: via indications of loss in the returning acknowledgment (where the cumulative acknowledgment point does not advance) or via a timeout. In the former case we cut the cwnd in ½ and store this new value in the ssthresh variable. If it's via the latter, we set the cwnd to 1 packet and again set ssthresh to ½ the old cwnd value before the loss.

In either case we start retransmitting the lost data and once all the lost data has been recovered, we start sending new data with a new lower cwnd value and an updated ssthresh. Note that whenever the cwnd rises above the ssthresh point we will change the algorithm used for increasing cwnd to congestion avoidance. This means that once a whole cwnd of data has been acknowledged we increase cwnd by one packet.

Now in briefly summarizing how loss-based congestion control works I have skipped over some finer points (more details on how to recognize loss for example) and some other nuances. But I wanted to give you an idea as to how it is working so we could then shift our attention to routers on the Internet to focus on what happens because of these loss-based mechanisms.

Routers typically have buffers associated with their links. This way, if a burst of packets

arrives (which happens often), they do not have to discard any packets but can forward the packets to the next hop through whatever link that leads there. Especially when the link speeds vary between incoming versus outgoing. Let's look at Figure 1 below.

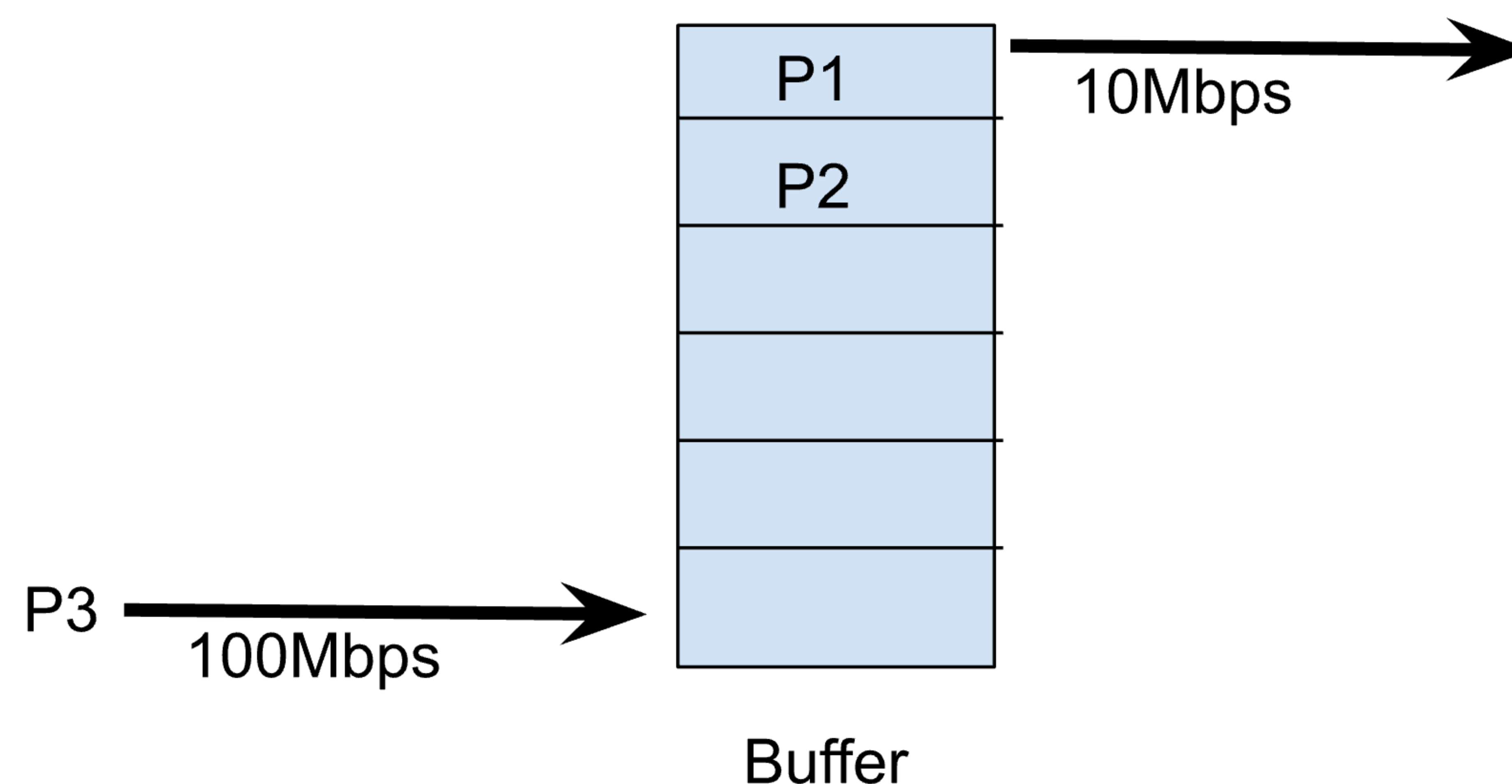

Figure 1: A bottleneck router with a buffer

Here we see P3 arriving at 100Mbps and it will get placed into the third slot in the router's buffer. P1 is currently being transmitted onto the 10Mbps link and P2 is waiting for its turn to be transmitted. Assuming a 1500-byte packet P3 will take approximately 120 microseconds to transmit across the 100Mbps network. When it is its turn to go out onto the 10Mbps destination network it will take 10 times that or 1200 microseconds to be sent. This means that every packet in the router's buffer will cause a 1200 microseconds of additional delay to be added to the packet just arriving.

Now let's step back and consider what our congestion control algorithm is going to optimize for. It will send packets as fast as it can until it loses a packet. If we are the only ones sending on the network this means that we will have to completely fill the routers buffer before a loss occurs. This means we are optimizing the router to always have a full buffer. And since memory is cheap, routers have grown quite large buffers. This means that we end up with long delays when a large transfer is happening via a loss-based congestion control algorithm. In Figure 1 we see only 6 slots for packets but in real routers there can be 100's or 1000's of packets in a routers buffer waiting to be sent. This means that the round-trip time seen by a TCP connection might vary from just a few milliseconds (when no packets are in queue) and then spike up to seconds due to buffering by the routers and TCP's AIMD congestion control algorithms always wanting to keep the buffer completely full.

If we are the only ones sending on the network this means that we will have to completely fill the routers buffer before a loss occurs.

You may have heard the term "buffer bloat" which impacts any real time applications (video calls, audio calls or games), this is directly caused by loss-based congestion control and is what we have just described.

Delay Based

For quite a long time, researchers and developers have known about the tendencies of loss-based congestion control to fill buffers. Long before all the talk of buffer bloat alternative congestion controls had been proposed to solve this issue. One of the first such proposals was TCP Vegas[3]. The basic idea in TCP Vegas is that the stack keeps track of the

lowest RTT it has seen, called the "Base RTT". It uses this information during Congestion Avoidance to determine an expected bandwidth i.e.:

$$Expected = cwnd / BaseRTT$$

The actual bandwidth is also calculated as well i.e.:

$$Actual = cwnd / CurrentRTT$$

Then a simple subtraction is done to determine the difference i.e.:

$$Diff = Expected - Actual$$

The difference *Diff* is then used to determine if the cwnd should be advanced or reduced based on two thresholds $\alpha < \beta$. These thresholds help to define how much data should be in the buffer of the bottleneck. If the difference is smaller than α then the cwnd is increased and when the difference is larger than β then the cwnd is decreased. Whenever the difference is between the two then no change is made to the cwnd. This clever formula with low values (usually 1 and 3) keeps the buffer at the bottleneck to a very small value optimizing the connection to keep the buffer just full enough to achieve optimal throughput for the connection.

During Slow Start, TCP Vegas modifies the way the increase works by alternating every other RTT. The first RTT Slow Start increases as New Reno or other loss-based congestion control mechanisms would. However, on the next RTT, TCP Vegas does not increase the cwnd but measures the difference using the cwnd to again calculate if the router buffer has been saturated. When the actual rate falls below the expected rate by one router buffer, slow start is exited.

Testing with TCP Vegas shows improvements to both RTT and throughput.

Perils of Mixing the Two

Testing with TCP Vegas shows improvements to both RTT and throughput. So why did we not fully deploy TCP Vegas gaining all its benefits?

The answer to that is contained within what happens when a loss-based congestion-controlled traffic competes against a delay based one. Imagine your TCP Vegas connection faithfully tuning the connection to keep only 1 or 2 packets in the bottleneck routers buffer. The RTT is low, and your throughput is at your maximum share. Then a loss-based flow begins, it will of course fill the router buffer until it experiences a loss, which is the only way it learns to slow down. To the TCP Vegas flow a signal that it is going too fast is received repeatedly, getting it to continue to cut its cwnd until it is getting almost no throughput. In the meantime, the loss-based flow gets all the bandwidth. Basically, the two types of congestion control, when mixed, always end up turning out poorly for the delay-based mechanism. Since loss-based congestion control was and is widely deployed on the Internet this then provided a huge dis-incentive for deploying a delay-based congestion control.

Mixing Loss and Delay Based Approaches with DGP

DGP attempts to integrate both loss-based and delay-based approaches in choosing its pacing rate. For the delay-based component Timely[4] was chosen (with some adaptation for the Internet) though arguably any delay-based approach (including TCP Vegas) could have been adapted for this purpose. Timely uses a delay gradient to calculate a multiplier which is combined with the current loss-based congestion controls calculations (either New Reno or Cubic) to derive an overall pacing rate using the following formula:

*Bw = max(GPest, LTbw) * TimelyMultiplier*
FillCwBw = cwnd / CurrentRTT
PaceRate = max (Bw, ((FCC == 0) ? FillCwBw : min(FillCwBw, FCC)))

We will discuss each part of the above formula in the following subsections to give you an idea of how DGP works. For the deep details on Timely we recommend you read the paper[4].

Goodput (GPest)

One of the foundational measurements that DGP keeps track of is the goodput. This is like BBR's[5] delivery rate but different in a subtle way. The delivery rate calculates the arriving rate of all data at a TCP receiver. When there is no loss the delivery rate and the DGP goodput are identical. But in cases of loss, the DGP rate lessens. This is because the goodput is measured strictly on advances to the cumulative acknowledgment (cum-ack), when a loss happens the cum-ack stops advancing. All the time it takes to recover a lost packet is thus folded into the goodput estimate lowering the GPest value.

To measure the goodput initially the IW is allowed to be sent in a burst, this starts the very first measurement window. The goodput is usually measured over 1 - 2 round trips worth of data and is calculated based on the advancement of the cum-ack over that period. During the measurement period a separate RTT is also calculated over that period i.e. the curGpRTT (which will be used later as input to Timely). Once the IW is acknowledged we have a seed of the first measurement. For the next three measurements the estimate is averaged. Once a fourth measurement is made future estimates use an apportioned weighted moving average to update the current GPest. Every time a new GPest is started the curGpRTT is saved into the prevGpRTT and a new weighted moving average of RTT is also begun which will become our new curGpRTT (note this RTT is a separate measurement from the smoothed round trip that TCP continues to make as well). The GPest measurement is continually made by the sender when data is in transit to the receiver. Any time that the sender becomes application limited the current measurement is ended. Note that an implementation becoming congestion window limited does not stop the current measurement. This description has been rather brief and may warrant a future article on how the RACK stack measures the goodput.

To measure the goodput initially the IW is allowed to be sent in a burst, this starts the very first measurement window.

Long Term Bandwidth (LTbw)

DGP also tracks another bandwidth measurement termed the LTbw. The LTbw is the total sum of all bytes cumulatively acknowledged divided by the total time that the data was outstanding. This value is almost always lesser than the current goodput value but in cases of sharp decline in the bandwidth measurement it can provide a stability to the current bandwidth estimate.

Delay Gradient with Timely (TimelyMultiplier)

Timely provides a multiplier that generally ranges somewhere between 50% - 130% of the estimated bandwidth. Timely uses the following formula (from the paper):

Algorithm 1: TIMELY congestion control.

Data: new_rtt
Result: Enforced rate
new_rtt_diff = new_rtt - prev_rtt ;
prev_rtt = new_rtt ;
rtt_diff = (1 - α) · rtt_diff + α · new_rtt_diff ;
▷ α: EWMA weight parameter
normalized_gradient = rtt_diff / minRTT ;
if *new_rtt* < T_{low} **then**
  rate ← rate + δ ;
  ▷ δ: additive increment step
  return;
if *new_rtt* > T_{high} **then**
  rate ← rate · $\left(1 - \beta \cdot \left(1 - \frac{T_{high}}{new_rtt}\right)\right)$;
  ▷ β: multiplicative decrement factor
  return;
if *normalized_gradient* $\leq$ *0* **then**
  rate ← rate + N · δ ;
  ▷ N = 5 if gradient<0 for five completion events (HAI mode); otherwise N = 1
else
  rate ← rate · (1 - β · normalized_gradient)

Timely was designed for the data center environment where the RTT's and bandwidths at various points are known entities. For use in DGP this is not the case, so we substitute the new_rtt and prev_rtt in the above formulas with the curGpRTT and the prevGpRTT respectively. We only do a Timely calculation at the end of making a goodput estimate. The multiplier calculated then stays with the connection as is until the next goodput estimate is complete and the multiplier is again updated along with any update to the goodput. Note also that timely uses a minRTT i.e. the minimum expected RTT. Again, this is not something known on the Internet as compared to the data center where the RTT at any point is a known quantity, and so it is derived as the lowest RTT seen in the last 10 seconds, the same as BBR. Also, just like BBR, to reestablish the minimum RTT periodically DGP will go into a "probeRTT" mode where the cwnd is reduced to 4 segments for a short period of time so that a "new" low RTT can be found. Note that the addition of a BBR style probe-RTT phase also helps DGP to become more compatible with BBR flows it is competing with.

With these tweaks, the Timely algorithm is adapted into DGP. For the deeper details on either probeRTT or Timely I suggest reading the papers.

Loss Based Pacing or Filling the Congestion Window (FillCwBw)

To pace out packets for loss-based congestion control a simple method exists. Take the currentRTT (kept in any stack doing Recent Acknowledgement[6]) and divide that into the congestion window. This tells the pacing mechanism what rate to pace at that will spread the current congestion window over the current RTT. Any loss-based congestion control, New Reno or Cubic, can be used with this method to simply deduce a pacing rate that would be dictated by the congestion control algorithm. We call this rate the Fill Congestion Window rate (FillCWBw) since it is designed to fill the congestion window over an RTT.

It should be noted that by pacing packets out over the entire congestion window it is highly likely that the sender will have less loss. This is due to less pressure on the bottleneck

by allowing some time between each microburst of packets sent. This time allows the bottleneck to drain some before the next microburst of packets arrives. Having less loss will naturally mean that the congestion window will gain a higher value since loss is the only thing that causes the cwnd to be reduced.

The Fulcrum Point: Fill Congestion Window Cap (FCC)

So far, DGP has calculated a bandwidth based on the goodput estimate in combination with Timely to increase or decrease that rate based on the RTT gradient (a delay-based component). We have also calculated a bandwidth for pacing based on the value of the congestion window (via whatever congestion control is in play) and the current RTT (the loss-based component). This gives us two distinct bandwidths we could pace at.

So, this is where the Fill Congestion window Cap (FCC) comes into play. If one is set (you can set it to zero to always get the fastest bandwidth), it becomes the limit of how much we will allow the loss-based rate to apply. The current default in FreeBSD is set to 30Mbps. So, for example if the FillCwBw calculated out to 50Mbps and the Bw, factoring in the Timely value on top of the estimate bandwidth came out to 20Mbps, then we would pace at the limit of the FCC i.e. 30Mbps in a default setting. If the Timely calculated Bw was 80Mbps then we would pace at 80Mbps.

What happens here is that the FCC serves as a Fulcrum point and limit to how much the nominal loss-based congestion control algorithm will influence the pacing rate. The FCC declares that your connection will push against other loss-based flows to maintain a rate of at least FCC, if possible, based on the congestion control value. If neither value meets the FCC limit, then the larger of the two will be dominant.

General Performance While Testing on the Internet

In a past large-scale experimentation at my previous company, DGP running with an FCC limit of 30Mbps (now the default) reduced RTT by up to over 100ms with no real degradation in Quality of Experience (QoE) metrics. If the FCC point was raised to 50Mbps QoE metrics improved i.e. things like Play Delay and Rebuffers improved with a sacrifice of little to no reduction of the RTT. The 30Mbps setting was adopted as a default in response to this testing, valuing the reduction in RTT (indicating much better router buffer behavior) than the corresponding gain in QoE metrics.

Enabling DGP in FreeBSD

There are at least two ways of enabling DGP on a FreeBSD system that has the RACK stack loaded and set as the default stack. If the source code of the application is available, you can add to the source code the setting of the socket option `TCP_RACK_PROFILE` to a value of '1' as follows:

```
socklen_t slen;
int profileno, err, sd;

….
profileno = 1;
slen = sizeof(profileno);
err = setsockopt(sd, IPPROTO_TCP, TCP_RACK_PROFILE, &profileno, slen);
```

The above code snippet will enable DGP on the socket associated with `sd`.

Another mechanism if you do not have access to the source code is to use **`sysctl`** to set the default profile for all TCP connections using the RACK stack to the value of '1'. You do this as follows:

```
sysctl net.inet.tcp.rack.misc.defprofile = 1
```

Note that once this value is set, all TCP connections using the RACK stack will use DGP with a default FCC value of 30Mbps. You can change that default (the FCC) as well to better match your network conditions with **`sysctl`** as well. The **`sysctl`**-variable **`net.inet.tcp.rack.pacing.fillcw_cap`** holds the FCC in bytes per second. For example, if I want to set the value to 50Mbps the following command can be used:

```
sysctl net.inet.tcp.rack.pacing.fillcw_cap = 6250000
```

The default value is 3750000 i.e. 30Mbps, you take the value you would like set in bits per second and divide by 8. So, 50,000,000 / 8 = 6,250,000.

You can also use the **`TCP_FILLCW_RATE_CAP`** socket option if you have access to the source code as follows:

```
socklen_t slen;
int err, sd;
uint64_t fcc;
….
fcc = (50000000 / 8);
slen = sizeof(fcc);
err = setsockopt(sd, IPPROTO_TCP, TCP_RACK_PROFILE, &fcc, slen);
```

Note that this will change the FCC value for just the specified connection and not the entire system.

You can also turn the FCC feature off and pace at always the maximum allowed by either Timely or the congestion control by setting the FCC value to 0. This will likely give you the best performance but will not reduce router buffer usage and thus buffer bloat.

How to Set Parameters?

So what settings are right for your network? In most cases the bottleneck is in your home gateway so knowing the bandwidth of your Internet connection can give you a good idea on what the FCC value should be set to for your connection. For example, I have two sites I administer, one is a symmetric 1Gbps connection, my FCC value for that machine I leave at the default of 30Mbps. This of course only affects outbound TCP connections using the RACK stack where the server is sending data. Leaving the default implies that for the most part delay-based performance will be coming out of my server and each connection will only push to maintain 3% of the network uplink capacity with loss-based mechanisms.

In my second system it has an asymmetric cable modem and only has 40Mbps up. In such a situation I have my FCC point set to 5Mbps. If I get more than 7 connections, they will start to push against each other using the loss-based mechanisms all attempting to get at least 5Mbps.

Future Work

Currently the FCC point is set in a static fashion on the entire system. This means that often the value is suboptimal, and a better value could possibly be selected (possibly gain-

ing both performance and reductions in RTT). The author is currently working on a more dynamic mechanism for setting the FCC point. The basic idea is that the connection would measure, over some time, the actual path capacity. Then once a value is available for the "Path Capacity Measurement" (PCM) a set percentage of that would be dedicated as the FCC point. This would then in theory make DGP more dynamic in tuning to the network path being used while reserving and pushing for some portion of the available bandwidth specific to each network type. Hopefully the work will be completed in 2025. Once completed, the RACK stack will change its default to enable DGP.

References

1. S. Floyd, T. Henderson: "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 1999.
2. S. Ha, I. Rhee, L. Xu: "Cubic: A New TCP-Friendly High-Speed TCP Variant", in: ACM SIGOPS Operating Systems Review, Volume 42, Issue 5, July 2008.
3. L. Brakmo, L. Peterson: "TCP Vegas: End to End Congestion Avoidance on a Global Internet", in: IEEE Journal on Selected Areas in Communications, Volume 13, No. 8, October 1995.
4. R. Mittal, V. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Gohabdi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats: "TIMELY: RTT-based Congestion Control for the Datacenter", in: ACM SIGCOMM Computer Communication Review, Volume 45, Issue 4, August 2015.
5. N. Cardwell, Y. Cheng, C. Gunn, S. Yeganeh, V. Jacobson: "BBR: Congestion-Based Congestion Control", in: Queue, Volume 14, Issue 5, December 2016.
6. Y. Cheng, N. Cardwell, N. Dukkipati, P. Jah: "The RACK-TLP Loss Detection Algorithm for TCP", RFC 8985, February 2021.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.