TCP variants: A review on congestion control algorithms

1st José Barata
Faculdade de Ciências
Universidade do Porto
Porto, Portugal
up201503334@fc.up.pt

2nd Luís Queijo
Faculdade de Ciências
Universidade do Porto
Porto, Portugal
up201503039@fc.up.pt

3rd Tiago Melo
Faculdade de Ciências
Universidade do Porto
Porto, Portugal
up201604918@fc.up.pt

***Abstract*—The Transmission Control Protocol (TCP) is a protocol that provides end-to-end connectivity in the transport layer of the Internet. Consisting of a set of mechanisms to control communication, the many variants of TCP released across the years generally differ in their congestion control approach. We begin by providing a compendium of congestion control terms and mechanisms, explaining their role and functionality. The knowledge present therein is meant to help the reader understand the following discussion. In this discussion, we examine a list of relevant TCP variants. For each variant, we explain what differentiates it from all others and consider the benefits and problems inherent to it. We specifically examine the Tahoe, Reno, New Reno, SACK, Westwood+, BIC, CUBIC, Vegas, Veno, and BBR variants.**

I. Introduction

The Transmission Control Protocol (TCP) is a connection-oriented protocol that provides logical end-to-end communication between applications through the transport layer of the Internet. TCP is generally used for services that require a reliable and ordered delivery of packets. It is equipped with several features to achieve this purpose, the most notable ones being flux control, connection management, and congestion control.

The evolution of this protocol has resulted in many TCP variants with the objective of increasing throughput and decrease congestion. Variants generally differ in the way they control the data rate, react at the lack of arrival of acknowledgments of received packets (ACKs) and deal with congestion. They are usually labeled after their underlying congestion control algorithms, as that is their main focus.

The aim of this paper is to discriminate the important differences between the most popular TCP variants implemented in the Linux kernel and some of their relevant derivatives.

II. TCP Congestion Control

In broad terms, congestion happens when the load put under a network is too large for its bandwidth. The resulting symptoms often manifest as increased delays in router queues and packet loss when these queues spill. In order to tackle this issue, considered one of the top priorities in network research, many TCP variants were developed across the years.

Although some may differ between variants, TCP congestion control commonly makes use of four different mechanisms: slow-start, congestion avoidance, fast retransmit and fast recovery. These four algorithms, when intertwined, are used to implement and manipulate the congestion window maintained for each connection.

The congestion window is a variable present in each sender host that limits the amount of data the TCP can send into the network before receiving an ACK. Controlling the window's size is useful for preventing traffic overload in links in the network. An analogous window, called the receiver window, is present at the receiver side of a given established connection to limit the amount of overdue data by advertising its information.

The slow start threshold (ssthresh) is a variable that is used to choose between the slow start and congestion avoidance algorithms for the purpose of data transmission control. The slow start algorithm is used when the congestion window size is smaller than ssthresh, while congestion avoidance is used when it is larger.

The slow start algorithm provides an exponential growth to the congestion window size (cwnd) by expanding it by a maximum of one sender maximum segment size (SMSS) per confirmation of maximum segment size received (ACK). This results in double the throughput every round trip time, which makes the slow start algorithm appropriate when it is advantageous to accelerate throughput until we hit the maximum tolerable by the network.

Congestion avoidance increments the congestion window size in a linear fashion, following an additive increase pattern that increases the size by one SMSS per round trip time until congestion is detected.

When a timeout occurs, ssthresh is updated to half the size the congestion window had before the loss. This is done to better determine if the congestion window size growth should be linear or exponential, corresponding that necessity to the appropriate algorithm.

When a sender receives 3 duplicate ACKS, the fast retransmit algorithm transmits the corresponding segment once more, as it is apparently missing. This is done without waiting for any retransmission timers to expire in order to speed up the recovery of possible losses.

The fast recovery algorithm follows fast retransmissions by taking control of the data transmission rates. In case of a timeout, it reduces cwnd to 1 SMSS and shifts growth to

slow start. After 3 duplicate ACKs, however, cwnd is only halved and the window growth becomes linear (hence the fast recovery). This is motivated by the fact that while a timeout is usually a sign of relevant congestion, three duplicate ACKs often signify that following segments might still be able to successfully transit.

Understanding the general structure of the aforementioned techniques will accelerate any further discussion about them and help understand how these differ between the different variants across the continuation of this article.

III. Loss-based Congestion Control Algorithms

Loss-based congestion control algorithms are built upon the premise of needing packet loss as a signal to begin controlling congestion. In general, these algorithms take a more reactive approach instead of proactively seeking to avoid congestion in the first place. In the following subsections we'll go over some examples of algorithms of this nature.

A. TCP Tahoe.

Before TCP Tahoe, TCP did little to control congestion. Its implementation simply used a go-back-n model that relied on cumulative ACKs and expired retransmission timers in the hopes of providing reliability. TCP Tahoe then added the mechanisms of slow start, congestion avoidance and fast retransmission. It also enhanced several previously established features.

The TCP Tahoe algorithm is based on the principle of "conservation of packets". This principle states that if a certain connection is alread running at the maximum available bandwidth capacity, then the packet should not be injected into the network. This principle has manifested in the mechanisms listed in the previous paragraphs.

TCP Tahoe's fast retransmission algorithm performs at its best when dealing with packets that were lost due to congestion. Fast retransmiting can save a considerable amount of time whenever a loss happens, therefore improving throughput.

The main issue with TCP Tahoe is that it takes a whole timeout interval to detect packet losses and react properly. It also performs less than optimally on high delay bandwidth connections because of its mandatory initiation of slow start after a loss occurs.

B. TCP Reno.

TCP Reno improves on TCP Tahoe by detecting lost packets earlier. This is done by making it a requirement that the sender receives an immediate ACK when a segment is received. With this implementation, duplicate ACKs signify that a segment has either been delayed in the network and delivered out of order or has simply been lost. To discern between the two former cases, TCP Reno introduced the fast recovery phase after fast retransmissions.

Fast recovery differentiates between delayed/out-of-order delivery (3 duplicate ACKs) and packet loss (timeout). It also combats Tahoe's slow start issues in high delay connections by taking control of the data transmission rates after fast retransmits. Instead of simply going back to slow start, Reno's fast recovery mode chooses how to act depending on the two previously mentioned scenarios that it is meant to differentiate. When multiple packets are lost in the same window, however, TCP Reno's performance becomes similar to TCP Tahoe as it can only detect single packet losses. This happens because information about any second packet that is lost will only come after the ACK for the first retransmitted lost segment reaches the sender.

This shortcoming might result in cwnd being reduced twice during one window, which in turn might make it too small to adequately serve the algorithm. If the window doesn't have enough size to accommodate losses when there is need to receive the adequate amount of duplicate ACKs in order to fast retransmit, we would then be in a situation where we're forced to wait for a coarse-grained retransmission timer to expire.

C. TCP New Reno.

TCP New Reno is an improved version of TCP Reno that mainly aims to be more competent when it comes to multiple packet losses. It addresses this issue by not exiting fast recovery mode until all the data which was outstanding when fast retransmission was triggered is acknowledged.

When a new ACK is received, TCP New-Reno's fast recovery mode proceeds according to two cases:

- If it acknowledges all segments that were outstanding when we entered fast recovery, then it exits fast recovery while setting cwnd to ssthresh and, like Tahoe, resuming in congestion avoidance mode. This combats the issue of cwnd becoming too small, as the window size won't change more than once per window.
- If it is partial then the last segment in line was lost and is retransmitted. The number of duplicate ACKs received are then set to zero. Fast recovery ends as soon as all data in the window receives an ACK.

TCP New Reno still suffers from taking one round trip time to detect each packet loss. We can only detect more than one segment was lost when an ACK for the first retransmitted segment is received.

D. TCP SACK

TCP with Selective Acknowledgements (SACK) is another expansion of TCP Reno that enables multiple packet loss detection and retransmission of multiple lost packets per round trip time. This variant requires segments to not be acknowledged cumulatively but selectively. This means that each ACK has a section describing which segments are being acknowledged rather than it acknowledging all previous segments. This allows the sender to better know which packets are still outstanding. A new pipe tracks the number of packets that are currently in transit.

SACK'S fast retransmit functions like New Reno's. It enters this phase when a loss has occured and exits when all of the data that was outstanding when fast retransmit began is acknowledged. This variant doesn't really differ from the others

in terms of congestion control other than the aforementioned issues that it aims to fix with selective ACKs.

E. TCP BIC

The TCP BIC variant was used by default in Linux kernel versions 2.6.8 and above, being changed to CUBIC in version 2.6.19. It is used predominantly in networks with high speed and high latency (long fat networks) because it is more efficient than others at correcting underutilized bandwidth.

The way BIC's congestion control algorithm adjusts the congestion window size can be divided into three parts: binary search increase, additive increase and slow start. In case of network failure, multiplicative decrease is used to correct cwnd.

After a packet loss, BIC reduces its cwnd using multiplicative decrease. The congestion window size before that reduction is saved as cwndmax (maximum congestion window size) and the window size after the reduction is saved as cwndmin (minimum congestion window size). Binary search is then performed using these two values. In broad terms, this search translates into "jumping to the midpoint" between cwndmax and cwndmin. The motivation behind this is that since a loss occured at cwndmax, the appropriate cwnd for the network should be inbetween these numbers.

To prevent excessive jumps in window size within a single round trip time, a fixed constant smax is set. If the distance between the midpoint and cwndmin is larger than smax, BIC increments cwnd by smax. If no packet losses occur at the updated cwnd, that becomes the new minimum. This process continues until cwnd is set to the current maximum. This means that after a reduction, the window will likely grow in a linear fashion (additive increase) followed by a logarithmic one (binary search).

A new maximum must be found when cwnd grows past it. In this case, BIC enters the max probing phase. Window growth here is exactly symmetric to those used in additive increase and binary search (concave). This makes it exponential (convex) in nature.

F. TCP CUBIC

CUBIC is a derivative of BIC that substantially simplifies the somewhat convoluted window adjustment of BIC by replacing it by a cubic growth function. Based on the time since the last congestion event, this function's inflection point is set to the window size prior to that same event.

This cubic function's shape can accommodate both the concave and convex window growth portions present in TCP BIC. During its concave portion, window size rapidly grows towards the cwnd before the last congestion event. Hitting that inflection point, CUBIC begins probing for more bandwith in exponential fashion. The time spent at the plateau between concave and convex is usually enough for the network to stabilize before probbing begins.

CUBIC is generally a tamer and more systematic approach than BIC TCP. Another substantial difference between CUBIC and other TCP variants is that it does not rely on round trip times to make cwnd adjustments. Window size is solely dependent on the last congestion event.

IV. Delay-based Congestion Control Algorithms

As opposed to reactive algorithms, delay-based congestion control algorithms generally try to avoid loss before it happens instead of waiting for congestion to happen and then control it. As such, they don't usually focus on packet loss as an indicator of congestion but rather on packet delay. This usually proactive approach to congestion control is the premise of the delay-based congestion control algorithms we will examine in the following subsections.

A. TCP Vegas.

TCP Vegas is an innovative modification of TCP Reno that implements several new mechanisms for congestion detection and avoidance. It makes use of the RTT as a main parameter to monitor traffic. The proactive approach this algorithm takes has historic relevance as it became an inspiration for many other preemptive congestion control algorithms later on.

Vegas implements three mechanisms that differ from TCP Reno:

- A new retransmission mechanism using a factor of 3/4 on duplicate ACKs (instead of Reno's 2/4 and triple ACK respectively).
- A modified slow start which tries to find the right cwnd without incurring a loss.
- Most importantly, a new congestion avoidance algorithm.

This congestion avoidance algorithm's main idea is avoiding packet losses. This is achieved by calculating the difference (DIFF) of an expected throughput with the measured throughput and trying to predict when congestion will occur based on it, adjusting accordingly. If DIFF*BaseRTT (BaseRTT usually being the RTT of the first segment sent) is smaller than a preset value A, it means that the connection is in danger of not utilizing the available bandwidth so the cwnd is increased linearly. On the other hand if the value is higher than another preset value B (B¿A) then the actual throughput is too far from the expected throughput and it means there is some congestion and the cwnd needs to be decreased. If the value is somewhere in between then the cwnd stays the same.

Although Vegas has a bigger throughput, less packet loss and faster retransmission than TCP Reno and New Reno, it wasn't widely deployed because when put together with congestion control algorithms (which "rush" to a packet loss in slow start and also rely on packet loss meaning they always try to grab more bandwidth) it is always having few access to bandwidth compared to these algorithms. Along with that, a wrong calculation of BaseRTT can mean constant congestion because Vegas relies on this value being accurate (which may very well not be). There is also a rerouting problem with Vegas where it will function poorly when new routing paths with bigger propagation delay appear.

B. TCP Veno.

As the name suggests, Veno is a combination of TCP Vegas and TCP Reno, using packet delay to monitor traffic, but mainly using Reno's already established mechanisms adding simply a new AIMD and implementing Vegas congestion avoidance mechanic. It has been included in the Linux kernel since version 2.6.18.

The motivation to the development of Veno was the emergence of heterogeneous networks (such as LANs and wireless), where different protocols may be in use. The problem is that algorithms like Reno, New Reno and SACK design solutions to deal with packet loss alone which may affect the throughput when packet loss was actually a random packet loss (noise, link errors...) which in networks like wireless is very common. Likewise, delay based algorithms like Vegas assumes all packet loss is due to random loss, because it mainly focus on avoiding congestion. This can cause throughput to not be the most efficient.

Veno distinguishes between congestive loss and non congestive loss and sets the ssthresh value accordingly. Vegas mechanism can help distinguish between these two: when a packet loss is detected by fast retransmit if the value was below the threshold B (meaning it's within non congestive boundaries) then the loss was probably due to random loss so the ssthresh can be set high, if not then most likely it was a congestive loss and ssthresh should be set lower. When packet loss is detected by retransmit timeout timer it functions as same as Reno.

Although Veno is fair with algorithms like Reno, New Reno and Sack, and improves efficiency when compared to these algorithms, it still has a major problem: not good enough for high speed networks.

C. TCP BBR.

First published by Google in 2016, implemented on Linux TCP since Linux 4.9 and achieving an average of 4 % higher network throughput and up to 14 % in some countries when implemented on Youtube, Bottleneck Bandwidth and Round-trip propagation time (BBR) is a delay/congestion based algorithm that aims to create no packet loss or buffer filling while achieving full bandwidth throughput.

The states of a queue and link with a single data flow can be described as the following:

- State 1: Send rate is lower than the capacity of the link, thus being under-utilised;
- State 2: Send rate is greater than the link capacity and the queue in the receiver starts filling up;
- State 3: Send rate is still greater than the link capacity and queue is full, resulting in packet losses.

Whilst loss based algorithms like CUBIC hover between state 2 and 3 and can achieve pretty good performance, it is still sub-optimal as the best point to achieve full bandwidth capacity with low losses is between state 1 and 2. TCP Vegas already tried to perform this but had many packet losses as well as not reacting well to bandwidth changes due to its linear increase of cwnd.

BBR tries to achieve a constant use of all available bandwidth without filling the buffer, which results in no congestion. By doing precise measures of every packet RTT and keeping the bottleneck of the bandwidth, it can maintain itself in a point where it doesn't cause the queues to fill up and still uses all the available bandwidth. At the same time it also uses probes like CUBIC to find out if it has more bandwidth to claim. The problem that TCP Vegas had when used concurrently with loss based algorithms is solved in BBR as it has a startup that doubles the sending rate at each RTT, making a much better job at getting bandwidth. It also adapts exponentially to bandwidth changes thus being able to keep it's already acquired bandwidth and decreasing/increasing it when needed.

BBR is very efficient, fast, and makes minimal demands on network buffer capacity. It is, however, still in development and therefore possesses some flaws such as starving CUBIC in some networks, still letting some packet losses occur as well as not being scalable. BBRv2 is already tackling most of these problems.

V. Hybrid Congestion Control Algorithms

As the name implies, hybrid algorithms use both packet loss and packet delay as means to control congestion to improve throughput. The following TCP variants can be considered both loss and delay-based.

A. TCP Westwood / Westwood+.

TCP Westwood+, being very similar to Westwood, is a modification of TCP Reno based on a end-to-end bandwidth estimate that optimizes performance in wireless and cable networks. It was first implemented on the Linux kernel in version 2.2.

The main goal of Westwood/Westwood+ is to improve performance in heterogeneous networks, and it has some similarities to Veno in that is uses an algorithm to adaptitively set sstresh and cwnd. It uses equal slow-start and congestion avoidance phases to that of Reno. However, instead of blindly halving the cwnd on packet loss, it uses an algorithm to estimate the real end-to-end bandwidth and sets sstresh and cwnd based on the bandwidth value when congestion occurred. This exact mechanic is also what was improved on TCP Westwood because soon after it's release, the algorithm used to estimate bandwidth was found to be flawed.

Westwood+ successfully improved on Reno's and New Reno's fairness, also being friendlier and achieving better results on heterogeneous networks comparatively to the previous mentioned algorithms, but still underperforms in high-speed networks.

References

[1] C. Callegari, S. Giordano, M. Pagano, T. Pepe. *Behaviour and analysis of TCP Linux variants*. University of Pisa, Via Caruso 16, 56122 Pisa, Italy, 2011.

[2] B. Qureshi, M. Othman. *Progress in Various TCP Variants*. February 2009.

[3] Pooja Chaudhary, Sachin Kumar. *A Review of Comparative Analysis of TCP Variants for Congestion Control in Network.* International Journal of Computer Applications (0975 – 8887) Volume 160 – No 8, February 2017.

[4] Sangtae Ha, Injong Rhee, Lisong Xu. *CUBIC: A new TCP-Friendly High-Speed TCP Variant*. North Carolina State University, Raleigh, NC 27695.
[5] U. Hengartner1, J. Bolliger1, Th. Gross1. TCP Vegas Revisited. Departement Informatik,ETH Zurich CH, 8092 Zurich.
[6] Fu Chegpeng. TCP veno: end-to-end congestion control over heterogeneous networks. Chinese university of Hong Kong 2001.
[7] Larry Bruce. Computer Networks: A Systems Approach 6th Edition. August 2019.
[8] By Geoff Huston. BBR, the new kid on the TCP block. 9 May 2017.
[9] Geoff Huston. TCP and BBR. 2018-05-15.
[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Van Jacobson. BBR Congestion Control. IETF 97: Seoul, Nov 2016.
[11] Luigi A. Grieco and Saverio Mascolo. Performance Evaluation and Comparison of estwood+, New Reno, and Vegas TCP Congestion Control. Dipartimento di Elettrotecnica ed Elettronica, Politecnico di Bari, Italy.
[12] Moritz Geist, Benedikt Jaeger. Overview of TCP Congestion Control Algorithms. Technical University of Munich, Germany.