TCP Congestion Control Approaches in Modern Operating Systems

Junshuai Zhang
Aberystwyth Univeristy, UK
juz1@aber.ac.uk

***Abstract*—TCP provides reliable end-to-end transmission and congestion control for the Internet. The most well-known TCP congestion control implementation is TCP Reno. However, TCP Reno is lack of efficiency to fully utilise network capabilities. There are many variants of TCP attempting to improve it. TCP New Reno is an improvement used by Apple's Macintosh systems, while Microsoft's Window systems applied Compound TCP. We discuss how they work, analyse their performance in high latency networks, and consider the effect of increasing packet loss. Also, we concentrate on their fairness in different aspects. In the end, we introduce another modern TCP congestion avoidance algorithm CUBIC TCP and look into the future.**

***Keywords*—TCP congestion control, TCP Reno, TCP New Reno, Compound TCP, CUBIC TCP, TCP fairness.**

I. Background

TCP congestion control means that senders should control the amount of data injected into networks to avoid congestion. In TCP, a sliding window has been used to send more than one segment before receiving the acknowledgement of the last segment. It improves efficiency greatly, but does not mean senders can send data without limits. An overlarge window size will cause congestion in networks without doubt. That leads to more terrible efficiency of sending data.

TCP determines the window size by the minimum of congestion window ($cwnd$) and receiver window ($rwnd$). While $rwnd$ is used for receivers to tell senders how much data they can deal with, $cwnd$ is the crucial parameter for congestion avoidance. So congestion control algorithms are usually algorithms about how to adjust $cwnd$ to fit network capabilities.

There are various TCP congestion control algorithms. Even though the significant influence of the window management on the communication performance has been mentioned in the fundamental TCP specification document, it only provides the window management suggestion, rather than specifying an approach of TCP congestion control [2]. In fact, a perfect solution of the window management have not been figured out. It turns out, there are many TCP variants which managing to improve the congestion control strategy to utilise networks as properly as possible.

II. TCP Congestion Control

The most well-known TCP congestion control approach is TCP Reno. In TCP Reno, four congestion control algorithms have been utilised: slow start, congestion avoidance, fast retransmit, and fast recovery. The following explanation of the process of TCP congestion control is based on RFC 2001, a very original version of TCP Reno [3].

A. Slow Start and Congestion Avoidance

The process of congestion control can be separated into two phases: *slow start* and *congestion avoidance*. The current $cwnd$ and *slow start threshold* ($ssthresh$) is used to determine which phase it is. If the current $cwnd$ is not greater than $ssthresh$, it is slow start phase. Otherwise, it is congestion avoidance phase.

In the beginning, the $ssthresh$ is an extra high value so that the transmission begins with slow start mode until the first congestion is detected. The $cwnd$ is initialised to one *maximum segment size* (MSS, which is typically 536 octets or 512 octets), and increased by one MSS whenever an acknowledgement(ACK) is received. It turns out, in the slow start phase, the $cwnd$ commonly has an exponential growth per RTT if no ACK is delayed.

When congestion is observed by a *retransmission time out* (RTO) or receiving duplicate ACKs, $ssthresh$ will be set to the current window size (i.e. the minimum of $cwnd$ or $rwnd$), but at least two MSS. This results in senders enter congestion avoidance mode. Then the $cwnd$ will obey the rules: each time sender receives an ACK, the $cwnd$ increases by $MSS * MSS/cwnd$, but at most one MSS per *Round-Trip Time* (RTT). If a RTO occurs, $cwnd$ will be set to 1 MSS, entering slow start mode. The $ssthresh$ should be changed like the following:

$$ssthresh = max(FlightSize/2, 2 * MSS),$$

where $FlightSize$ is the amount of data that has been sent but not yet acknowledged. Then senders resend the unACKed segments. If the sender receives three duplicate ACKs, it will use the algorithms fast retransmit and fast recovery, which we will discuss next.

B. Fast Retransmit and Fast Recovery

A sender will resends a missing segment unless receiving a new ACK before a RTO. However, retransmission time is too long to wait. When the sender receives three duplicate ACKs, it is very possible that the first segment the sender sent is lost. Then the sender can retransmit the unACKed segment as soon as possible. This is called *fast retransmit*.

Actually, the above algorithms have been introduced by TCP Tahoe. The TCP Reno imports a new algorithm named *fast recovery*: if senders receive three duplicate ACKs, network may be not so terrible because they can still receive ACKs. So senders do not have to enter slow start mode, just do retransmit the first unACKed segment and do the following adjustment:

$$ssthresh = max(0.5 * cwnd, 2)$$

$$cwnd = ssthresh + 3 * MSS$$

If another duplicate ACK arrives, senders increase $cwnd$ by the number of ACKed octets for keeping the same amount of data in the network, sending one more packet if $cwnd$ is allowed. If the next ACK is for new data, they set the $cwnd$ to the $ssthresh$. After that, it will enter congestion avoidance mode rather than slow start mode.

In congestion avoidance phase, there is a linear growth of the $cwnd$, but multiplicative decrease of the $cwnd$ if congestion happened. So it is been called *Additive Increase Multiplicative Decrease* (AIMD) .

III. TCP Congestion Avoidance Algorithms in Modern Operating Systems

Even though TCP Reno is so widely used, there are some issues in TCP Reno. Two popular modern operating systems: Apple's Macintosh systems and Microsoft's Windows systems, both use some variants of TCP Reno [1]. Macintosh systems use TCP New Reno, while Windows use Compound TCP (CTCP). They attempt to solve the issues of TCP Reno, using different congestion avoidance algorithms.

A. Apple's Macintosh Systems: TCP New Reno

Let us talk about Apples Macintosh systems first. The TCP New Reno is a modification to solve a problem that TCP Reno does not recover efficiently from multiple packets loss. To be specific, no matter how many packets are lost, the sender will resend only the first unACKed segment when it receives three duplicate ACKs. Then the other lost segments have to wait a RTO to be resended. However, RTOs led senders not only to enter slow start mode, but also to halve the $ssthresh$ for every packet loss.

Selective Acknowledgement (SACK) is a strategy which aims to address this issue [4]. It is used by receivers to tell senders which segments have been received. However, using SACK requires supports of both sides of senders and receivers. As TCP implementation follows an robust principle: be conservative in what you do, be liberal in what you accept from others. So the TCP New Reno raises an approach for solving the problem under the circumstances that SACK is not implemented by receivers or senders [6].

The way TCP New Reno addresses this problem is that, when senders receive three duplicated ACKs, they will enter fast retransmit mode, retransmitting the first unACKed segment. If only one packet is lost, senders should receive an ACK for new data which has not been sent. If multiple packets are lost, senders should receive an ACK for data which senders have sent but has not been ACKed. And this ACK is called a partial ACK. If senders receive a partial ACK, they will retransmit the next unACKed data as soon as possible instead of waiting for three duplicate ACKs. And they will stay in fast recovery mode until not receiving partial ACKs.

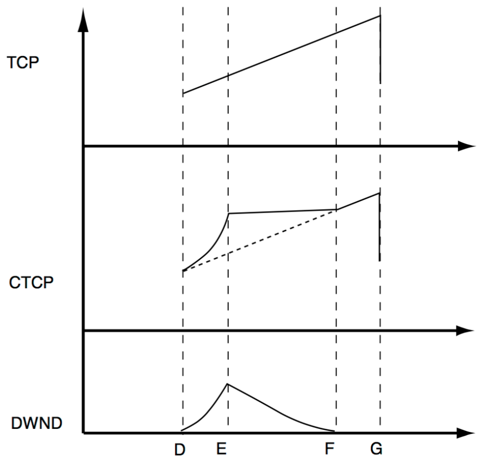

Fig. 1. Window evolution of TCP Reno, CTCP and $dwnd$ during congestion avoidance phase [9].

B. Microsoft's Windows Systems: Compound TCP

Another problem of TCP Reno is that, in a big fat pipe (i.e. a long fat network, LFN), when congestion occurs, it takes too long for the $cwnd$ to reach the bandwidth-delay product (BDP), which stands for the capability of networks. Because in TCP Reno, senders just increase the $cwnd$ by one MSS each RTT, but the BDP may be hundreds or even thousands of times as large as the $cwnd$. As the trend of the modern Internet is to become larger and faster, more and more high-speed TCP variants have been developed to replace the TCP Reno. The CTCP is one of them.

When we are talking about TCP Reno or TCP New Reno, they are loss-based algorithms, which means the variation of $cwnd$ replies to the loss of packet. The CTCP adds a delay-based component (i.e. the window size changes according to the variation of RTTs) into the standard TCP Reno. The delay-based algorithm came from TCP Vegas. By combining TCP Reno and TCP Vegas, it has been applied in several Microsoft Windows operating systems [9].

In the congestion avoidance phase of CTCP, the window size of senders does not only depend on $cwnd$ and $rwnd$ (or someone calls an *advertised window*, $awnd$), but also takes a variable $dwnd$ into account. It is calculated by the following formula:

$$win = min(cwnd + dwnd, rwnd).$$

When senders receive an ACK, the $cwnd$ will be changed like the following:

$$cwnd = cwnd + 1/(cwnd + dwnd).$$

So how does CTCP calculate the value of $dwnd$? It uses RTTs and win to estimate the throughout (i.e. win/RTT). If the throughout increased, $dwnd$ increase too. If throughout decreased, $dwnd$ decrease too. This considers the situation that, when routers start to queue packets, RTTs will be increased, then $dwnd$ have to go down accordingly in case to block the queue of routers.

As shown in Fig. 1, in congestion avoidance phase, $dwnd$ will grow rapidly during window growth period (from D to E). When the packets start to queue up in routers, RTTs will increase, then $dwnd$ will decrease, the window growth slows down (from E to F). Finally, the $dwnd$ is decreased to zero, the win will be the equal to the one of the normal TCP Reno (from F to G).

IV. Analyse

In this section, we talk about the performance of the above approaches in high latency networks and how increased packet loss affects them. We cite the experiment from the CTCP paper [9], which performs a good illustration of comparison between CTCP and the regular TCP (i.e. TCP Reno). The experiment is done under the environment as shown in Fig. 2.

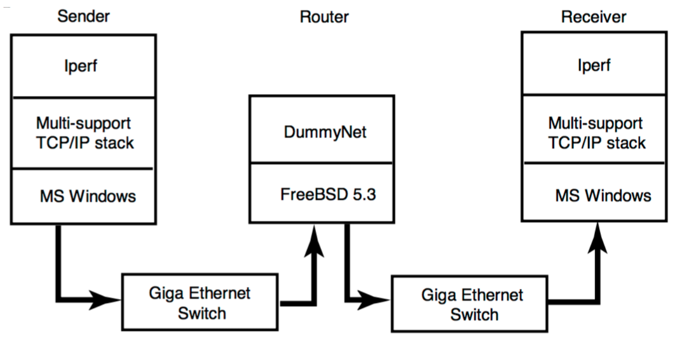

Fig. 2. A test environment for comparing the performance of CTCP, TCP Reno and HSTCP [9].

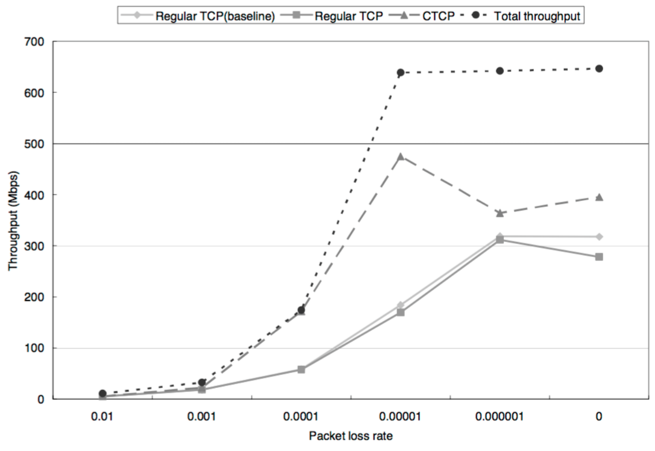

Fig. 3. Throughput of CTCP and Regular TCP flows when competing for the same bottleneck in the test environment Fig. 2 [9].

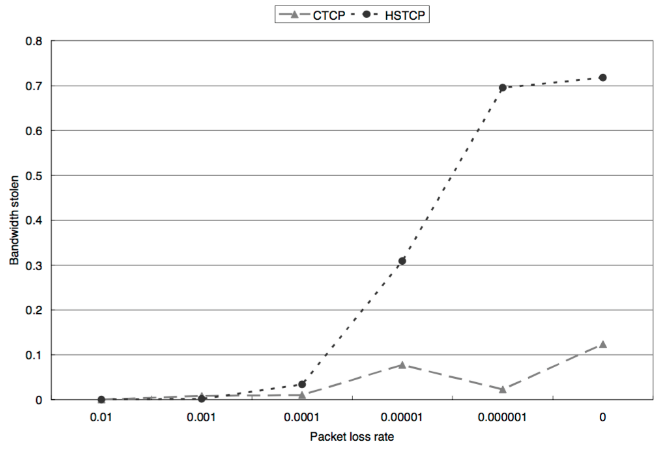

Fig. 4. Bandwidth stolen of CTCP and HSTCP for TCP Reno under various packet loss rate in the test environment Fig. 2 [9].

A. *High Latency Networks*

In a high latency network, RTTs are high. It is easy to understand that the loss-based approaches will not increase $cwnd$ fast enough, because they increase $cwnd$ per RTT. Therefore, Windows CTCP has a better performance than TCP Reno or TCP New Reno in this situation due to its delay-based component.

Besides, in the modern Internet, a high latency network is usually a high-speed networks (LFN), which has a high throughput. Under the circumstances, TCP Reno and TCP New Reno will have to wait for a long time to increase $cwnd$ to BDP. So they are not able to utilise high-speed network capabilities properly. But the CTCP is very scalable in this category of networks [9].

B. *The Effect of Increasing Packet Loss*

In networks, a packet loss is typically caused by congestion occurring. If congestion happened, TCP should be able to decrease the window size to slow down the sending rate. But if packet loss is caused by other than congestion, it is harmful for sending efficiency of TCP congestion control algorithms because they may recognise it as congestion then decrease the window size. In this situation, CTCP benefit from its delay-based component, and still maintains $dwnd$ to send data. Even through congestion really happened, it can be recovery faster than TCP Reno and TCP New Reno.

As shown in Fig. 3, when the packet loss rate is less than 10^{-6}, it brings no influence to whatever TCP Reno or CTCP. When the packet loss rate is between 10^{-6} and 10^{-5}, TCP Reno has underperformed, but CTCP has an excellent performance to keep the whole throughput high. When the packet loss rate is more than 10^{-5}, each algorithm does not have a good efficiency.

Another situation is, high packet loss rate highly possibly causes multiple packet losses. TCP may use SACK to solve this, and use *Forward Acknowledgement* (FACK) to retransmit the lost segments. However, if SACK is not implemented by the sender or the receiver. TCP New Reno which has a strategy to address this issue can handle this well, while CTCP has the same trouble with TCP Reno, that the window size has consecutive unnecessary reductions.

C. Fairness

The fairness of TCP congestion avoidance algorithms can be divided into two aspects: RTT fairness and TCP fairness.

1) RTT fairness: RTT fairness requires fairness when flows have different RTTs. As TCP Reno, TCP New Reno and other loss-based algorithms increase $cwnd$ based on RTTs, they have poor RTT fairness. By contrast, CTCP, which mixes a delay-based algorithm into a loss-based algorithm, will have better RTT fairness than others.

2) TCP fairness: TCP fairness requires fairness when flows are using different TCP implementations. TCP New Reno is almost the same as the old TCP Reno. So CTCP has less RTT-fairness because of $dwnd$. However, CTCP performs much better than HSTCP or other delay-based high speed TCP variants with only at most approximately 10% bandwidth stolen, as shown in Fig. 4.

V. A Modern Approach: CUBIC TCP

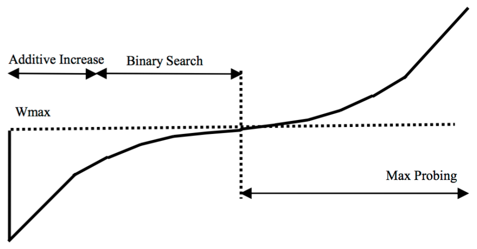

Fig. 5. The window growth function of BIC [8]

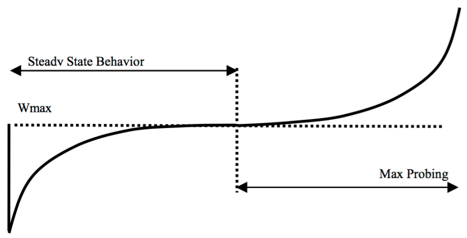

Fig. 6. The window growth function of CUBIC [8]

After much research, scientists just realised the major aim of TCP congestion avoidance algorithms is finding out an optimal size of $cwnd$. BIC TCP is an algorithm to use binary search to find out an optimal $cwnd$. And CUBIC TCP is an enhanced version of BIC. It use a cubic functions whose curve looks similar with the one of BIC to achieve better fairness. CUBIC is a high speed TCP variant like CTCP, also addresses TCP under-utilisation problems in LFNs. As the name represents, CUBIC use the following cubic function as the window growth function:

$$cwnd = C(t-K)^3 + W_{max},$$

$$K = \sqrt[3]{W_{max}\beta/C},$$

where C and β are factors for increasing speed and dropping ratio respectively, t is the elapsed time from the last window reduction, and W_{max} is the window size just before the last reduction.

CUBIC have a better RTT-fairness than any other TCP variants we mentioned, because its $cwnd$ grows independently of RTT. It may not so TCP-friendly as CTCP because it shows more aggressiveness. But it can utilise network capabilities much more than CTCP [10].

CUBIC is important because there are most web services running on Linux nowadays. For example, when people surfing on the Internet, it is very possible that the HTTP servers may use CUBIC TCP to transmit data of web pages. Meanwhile, the current Apple's Mac OS X systems use CUBIC TCP as TCP implementation by default.

VI. Discussion

Since the first version of TCP Reno was implemented, a lot of improved TCP variants includes TCP New Reno, Compound TCP and CUBIC TCP have been published. All TCP variants have some characteristics according to its implementation: loss-based algorithms are aggressive to satisfy bandwidth requirement but cause TCP unfairness and RTT unfairness; delay-based approaches provide RTT fairness but they are difficult to meet TCP fairness and it is lack of aggressiveness.

Since a long time ago, the industry seems to prefer aggressive loss-based algorithms to friendly delay-based algorithms (even the mixed loss-based algorithm CTCP). As I can check out, TCP Reno and TCP New Reno are classic implementations in old operating systems; BIC is used by default in Linux kernels 2.6.8 through 2.6.18; CUBIC TCP is implemented and used by default in Linux kernels from 2.6.19 to 3.1; Apple's Macintosh systems, as we mentioned, changed from New Reno to CUBIC; even through CTCP is used on Microsoft Windows Vista and Windows Server 2008 (not sure whether it is by default), it is not the default setting any more since Windows 7.

The reason of this, in my opinion, is that the complete fairness is impossible. As we know, TCP Reno, the most well-known variant of TCP, itself has own issues which causes under-utilisation problems of networks. Hence, there are many updates of TCP Reno. For example, *initial window* (IW) have been changed a couple of times [7]. If we compare the original version and the newest version of TCP Reno, they will not be completely fair.

Besides, we have no way to know whether there is a over-aggressive TCP which could make us slow down, so we have to use the strongest strategy we know. The TCP protocol wars are like competitive markets, we do not and are not able to limit the people's choice. Operating system manufactures would not like to let their products show any weakness to others. Meanwhile, if we assume everyone choose the newest best option, this brings another form of fairness.

The only thing worthy to consider may be that we cannot shut down the old TCP algorithms, because there are always some services using out-of-update TCP implementation. And a congestion avoidance algorithm should ensure they will not make more congestion in the networks, causing the whole network slow. In this context, as I think, the future of TCP protocol wars will be gorier and gorier.

References

[1] Huston, G., "TCP Protocol Wars," *Internet Protocol Journal*, 18(2), June 2015.

[2] Postel, J., "Transmission Control Protocol," *RFC 793*, 1981.

[3] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", *RFC 2001*, January 1997.

[4] Mathis, M., Mahdavi, J., Floyd, S. and A. Romanow, "TCP Selective Acknowledgement Options," *RFC 2018*, October 1996.

[5] W. Richard Stevents, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," *RFC 2581*, 1997.

[6] S. Floyd and T. Henderson, "The NewReno modification to TCPs fast recovery algorithm," *RFC 2582*, April 1999.

[7] Allman, M., Floyd, S., and C. Partridge, "Increasing TCP's Initial Window", *RFC 3390*, October 2002.

[8] Injong Rhee, and Lisong Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," In Proc. Workshop on Protocols for Fast Long Distance Networks, 2005.

[9] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan, "A Compound TCP Approach for High-speed and Long Distance Networks," Proc.of IEEE INFOCOM'05, July 2005.

[10] Oura, Ryo, and Saneyasu Yamaguchi. "Fairness Comparisons among Modern TCP Implementations." *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*. IEEE, 2012.