Resolving Poor TCP Performance on High-speed Long Distance Links – Overview and Comparison of BIC, CUBIC and Hybla

Marko Šošić*, Vladimir Stojanović*
* Group for Intelligent Systems, Faculty of Mathematics, University of Belgrade, Belgrade, Serbia
sosic.marko@gmail.com, vladimir.d.stojanovic@gmail.com

Abstract—TCP is de facto standard transport protocol. It has been widely adopted and used thanks to the strong guarantees it provides, reliable transmission before all. Today, many applications require fast data transfer over high bandwidth, high latency links. Standard TCP has a huge problem in achieving optimal performance, it substantially underutilizes network bandwidth. Over time, many solutions have been proposed. This paper describes the main problems and circumstances where those arise, especially large Bandwidth-Delay Product links. An overview of modern BIC, CUBIC and Hybla algorithms is provided, and comparative real-world test results and discussion are given.

I. INTRODUCTION

Transmission Control Protocol (TCP) [1] is the protocol between the IP layer and the application layer in the TCP/IP suite. Its responsibility is to receive a request from the application, to pack the data into small packets and pass it to the IP layer. At the other end, TCP is responsible for delivering the data to the application, identical to the data sent. This is called reliable transmission. Furthermore, TCP provides error detection, flow and congestion control. A big portion of applications rely on TCP [2], [3], [4]. It is everywhere, in computers, mobile phones, watches, and almost every device that uses Internet.

Today, sending huge amount of data over fast networks (often called bulk data transfer) is widely used with many different purposes. Usually, there are no problems when the distance between the sending and the receiving side is small. But, achieving fast data transfer over long distance links becomes close to impossible with standard TCP, its congestion control algorithm is very conservative [5], [6], and it is a big limiting factor even when there is no congestion. Another problem is round trip time (RTT) fairness. It has been well documented that TCP has issues maintaining fair allocation of bandwidth when flows have different RTTs [7], [8], [9], [10]. This paper is focused on the main problems occurring on high-speed links with great latency (often referred to as Big Fat Pipes) – links with large Bandwidth-Delay Product (BDP).

This paper is organized as follows. Section II provides a background of TCP – how it works, what is congestion, how does standard TCP congestion control work, and insight on why does the performance suffer. Section III gives an overview of what can be done to resolve poor performance. Three modern congestion control algorithms – BIC, CUBIC and Hybla are described. Section IV lays out testing methodology. Section V displays comparative test results and discussion. Section VI shows the conclusion.

II. BACKGROUND AND PROBLEMS

A. Reliable Transmission and Error Detection

Every packet has a sequence number. Loss, duplication and out of order packets are detected simply by looking at those numbers, and the receiver can successfully reconstruct the data. Acknowledgments are cumulative. One acknowledgment (ACK) signifies that all the packets preceding that sequence number have been received.

For error detection, TCP uses a 16-bit checksum of the data. If checksums do not match, corrupted packet is dropped. This is quite basic error detection by today’s standards, but when it is coupled with other schemes at lower levels (Ethernet’s CRC32 for example), they manage to capture practically all of the occurring errors.

B. Flow and Congestion control

If data is transferred between two points, one can easily overwhelm the other if the other one is slower and cannot handle the sending rate. To avoid this, TCP maintains a variable called the sliding window – a value that is advertised by the receiver. The sending side can only send up to that amount of data, then it must wait for an ACK. Each time when a new ACK arrives, the window “slides” and enables more packets to be sent. At any given point, there is no more than a window of packets in flight. This is called flow control.

Even when the receiving side can handle the sending rate, it could happen that the network in-between cannot. For example, consider an office with couple of computers, all connected to a single external network link. If all of them start sending at maximum link speed, disregarding the others, everyone would suffer, as majority of packets would be dropped, and retransmitted over and over again. TCP must not be insensitive to this problem. If it is ignored, congestion collapse occurs – a state where the network is oversaturated and no progress can be made [11]. To solve this, TCP introduced a variable called the congestion window (often abbreviated cwnd). The same principle of the sliding window is used, only the cwnd tells the sender how much data network (as opposed to the receiver, with flow control) can handle in a given time. The sender then sends minimum of the congestion window and the sliding window values, ensuring that both...
the receiving side and the network are not oversaturated. This is called congestion control.

Detecting when the congestion is taking place, and how the congestion window size should be maintained are not trivial problems. Since the Internet is based on end-to-end principle, there is no information about any node between the two endpoints. Many solutions have been developed over time, and a variety of approaches have been proposed, as shall be seen in the paper.

Most of the solutions use Additive Increase Multiplicative Decrease (AIMG) scheme [12]. The congestion window is increased by a constant on every acknowledged packet, and decreased by a multiplicative factor on sign of congestion. AIMG ensures max-min fair allocation [13]. All flows converge to their fair share of the link bandwidth.

On ACK:
\[
cwnd = cwnd + \frac{\alpha}{cwnd}
\]
(1)

On packet loss:
\[
cwnd = \frac{cwnd}{\beta}
\]
(2)

C. First Congestion Control solutions

The first congestion control algorithm was Tahoe [12]. It was implemented after TCP had already been widely used. At that point, Internet was at its knees because of ignored congestion problem.

At the connection start, the congestion window is set to a value of two (two times the segment size). For every ACK, the cwnd is increased by one. This phase is called slow start (SS), and the cwnd effectively doubles every RTT. At some point it reaches ssthresh value and switches to congestion avoidance (CA) phase. At CA, the cwnd increases by one every RTT. Tahoe assumes that if a triple duplicate ACK (often called 3dup ACK) is received, it is most likely that the packet is lost. This is the sign of congestion. Tahoe sets ssthresh to half of the current cwnd, resets the cwnd down to one, and reenters SS. However, it will wait for a timeout to expire before a retransmission. The same steps are taken if timeout expires without the ACK received.

Slow start:
\[
cwnd = cwnd + 1
\]
(3)

Congestion avoidance:
\[
cwnd = cwnd + 1/cwnd
\]
(4)

On triple duplicate ACK or timeout:
\[
ssthresh = \frac{cwnd}{2}
\]
\[
cwnd = 1
\]
(5)

Reno is an improvement upon Tahoe. It introduced fast recovery and fast retransmit [14]. On 3dup ACK, both ssthresh and the cwnd are set to cwnd/2, and there is no waiting for a timeout to expire, packets are retransmitted immediately on 3dup ACK. This increased performance.

While these solutions resolved the congestion problem, they introduced the problem of efficiency. Reno is considered obsolete, but is widely used today nonetheless.

D. Large BDP problem

BDP is a measure of how much data needs to be put on the link at one time (size of the cwnd) to fill the pipe. Suppose, for example, a cross-continental 10 Gbit/s link from Europe to USA that has 100ms RTT. BDP is 10 Gbit/s * 0.1 s = 1 Gbit (~120MB). The congestion window has to be huge (120MB). Standard TCP, by increasing the congestion window by one every RTT, would reach full sending rate approximately within one hour [15], if it is assumed none of the packets are lost (one packet loss over ~2.5*10^9 packet transmissions). This loss rate is far below what is possible today with the present technology [16], [17]. Even if it can be imagined that this low loss rate is possible, one hour is a huge amount of time to spend on reaching full rate.

Links with high bandwidth and great latency (often called Big Fat Pipes) cause a situation where TCP is mostly inefficient. Many rely on Big Fat Pipes to transfer large amounts of data, so the problem cannot be ignored. Algorithms that show great promise on resolving this are examined in later sections.

E. RTT Fairness problem

Consider general AIMD idea and its implementation used in standard TCP algorithm. The congestion window is incremented by one every RTT. This function is linear, but the rising rate depends on RTT. Bigger the RTT, the function rises slower. This means that AIMD ensures fair split only when flows have equal RTTs. If the flow has higher RTT that a competing one, it becomes penalized [7], [8], [9], [10]. For example, imagine a shared link with two flows competing. One flow has longer total end-to-end distance and latency several times the latency of the other flow. Since sending rate is directly proportional to the window size, flow with larger window will have higher sending rate. Although both flows have linear window increase, flow with smaller latency will have that value rise much faster. In a given time it will enlarge its window to a greater size comparing to the window of a flow with greater latency. It does not matter that multiplicative decrease will impact larger window more, flow with smaller latency will grab bandwidth all over again when it enters additive increase phase.
This problem arises even in most modern algorithms. The ones that do take care about this usually try to normalize RTT of the flow with the desired one.

III. MODERN SOLUTIONS AND ALGORITHMS

A. SACK and FACK options

It is worth mentioning that pure sliding window approach coupled with cumulative acknowledgment scheme is very conservative. Consider a situation where one packet is lost, and all other are reaching the receiver without problems. When a window of packets following the lost one is sent, TCP would stop and wait for the ACK of that one packet, it would not know about all the other received successfully, thanks to the cumulative acknowledgment. This is obviously a waste of time. To overcome that, TCP introduced SACK [18] (Selective Acknowledgement) and FACK [19] (Forward Acknowledgement) options. The two enable more information about lost packets to be sent to the receiver and provide the window inflation – a technique that allows the window to rise when small number of packets is lost until that hole is filled, thus enabling continuous flow. Both are very important in efficient network transfers, especially at high latency links, where a large amount of time is required to retransmit lost packets.

B. BIC

BIC (Binary Increase Control) [20] is TCP congestion control algorithm with primary focus on performance and great stability in high-speed networks. That resulted in an aggressive and efficient algorithm, but also in extreme unfairness to other TCP flows.

BIC has a unique algorithm for calculating its cwnd that uses a binary search algorithm.

This is presented in pseudo code:

```plaintext
BIC_cwnd(Smax, Smin)
begin
/* Smax: predefined maximal window increase */
/* Smin: predefined minimal distance to Wmax */
Wc = 1; /*current cwnd size */
Wt; /*target cwnd size */
Wmax; /* size of cwnd prior to congestion event */
MAX_PROBING:
if("ACK") then Wc = Wc + 1; /*exponential increase*/
else if("packet loss") then goto PACKET_LOSS;
else goto MAX_PROBING; /* repeat */
PACKET_LOSS:
/* enter additive increase phase */
Wmax = Wc;
Wc = Wmax / 2;
goto ADDITIVE_INCREASE;
ADDITIVE_INCREASE:
if("ACK") then
    /* if Wmax is reached, return to probing phase */
    if(Wmax - Wc < Smin) then goto MAX_PROBING;
    Wt = (Wc + Wmax) / 2;
    /* withholding the maximal increase parameter */
    if(Wt - Wc > Smax) then
        Wc = Wc + Smax;
        goto ADDITIVE_INCREASE; /* repeat */
    else goto BINARY_SEARCH;
else if("packet loss") then goto PACKET_LOSS;
else goto ADDITIVE_INCREASE; /* repeat */
BINARY_SEARCH:
if("ACK") then
    Wc = Wt;
    if(Wmax - Wc < Smin) then goto MAX_PROBING;
else
    Wt = (Wc + Wmax) / 2;
    goto BINARY_SEARCH; /* repeat */
else if("packet loss") goto PACKET_LOSS;
else goto BINARY_SEARCH; /* repeat */
end.
```

Figure 2. Two flows competing; congestion avoidance phase is shown. Flow in blue has four times larger latency than the other one shown in red. Although multiplicative decrease has a greater impact on red flow (as intended to ensure max-min fair allocation), additive increase will always allow the red flow to quickly climb back up and allocate most of the bandwidth.

This problem arises even in most modern algorithms. The ones that do take care about this usually try to normalize RTT of the flow with the desired one.

III. MODERN SOLUTIONS AND ALGORITHMS

A. SACK and FACK options

It is worth mentioning that pure sliding window approach coupled with cumulative acknowledgment scheme is very conservative. Consider a situation where one packet is lost, and all other are reaching the receiver without problems. When a window of packets following the lost one is sent, TCP would stop and wait for the ACK of that one packet, it would not know about all the other received successfully, thanks to the cumulative acknowledgment. This is obviously a waste of time. To overcome that, TCP introduced SACK [18] (Selective Acknowledgement) and FACK [19] (Forward Acknowledgement) options. The two enable more information about lost packets to be sent to the receiver and provide the window inflation – a technique that allows the window to rise when small number of packets is lost until that hole is filled, thus enabling continuous flow. Both are very important in efficient network transfers, especially at high latency links, where a large amount of time is required to retransmit lost packets.

B. BIC

BIC (Binary Increase Control) [20] is TCP congestion control algorithm with primary focus on performance and great stability in high-speed networks. That resulted in an aggressive and efficient algorithm, but also in extreme unfairness to other TCP flows.

BIC has a unique algorithm for calculating its cwnd that uses a binary search algorithm.

This is presented in pseudo code:

```plaintext
BIC_cwnd(Smax, Smin)
begin
/* Smax: predefined maximal window increase */
/* Smin: predefined minimal distance to Wmax */
Wc = 1; /*current cwnd size */
Wt; /*target cwnd size */
Wmax; /* size of cwnd prior to congestion event */
MAX_PROBING:
if("ACK") then Wc = Wc + 1; /*exponential increase*/
else if("packet loss") then goto PACKET_LOSS;
else goto MAX_PROBING; /* repeat */
PACKET_LOSS:
/* enter additive increase phase */
Wmax = Wc;
Wc = Wmax / 2;
goto ADDITIVE_INCREASE;
ADDITIVE_INCREASE:
if("ACK") then
    /* if Wmax is reached, return to probing phase */
    if(Wmax - Wc < Smin) then goto MAX_PROBING;
    Wt = (Wc + Wmax) / 2;
    /* withholding the maximal increase parameter */
    if(Wt - Wc > Smax) then
        Wc = Wc + Smax;
        goto ADDITIVE_INCREASE; /* repeat */
    else goto BINARY_SEARCH;
else if("packet loss") then goto PACKET_LOSS;
else goto ADDITIVE_INCREASE; /* repeat */
BINARY_SEARCH:
if("ACK") then
    Wc = Wt;
    if(Wmax - Wc < Smin) then goto MAX_PROBING;
else
    Wt = (Wc + Wmax) / 2;
    goto BINARY_SEARCH; /* repeat */
else if("packet loss") goto PACKET_LOSS;
else goto BINARY_SEARCH; /* repeat */
end.
```

Figure 3. BIC congestion window growth function. After packet loss it is in additive increase phase, followed by binary search phase and finally it is in max probing phase. The y axis represents the congestion window size, whereas x axis is time.
C. CUBIC

The window growth function of BIC achieves good scalability and stability in high-speed networks while in low-speed networks the window growth function is still too aggressive. Also, several phases (additive increase, binary search increase and max probing) of window control add complexity in implementing the protocol and analyzing its performance.

Window growth function of CUBIC [21] is not dependent of RTT, but rather on the time between two consecutive congestion events, which defines the cwnd size in real time. This property enables CUBIC to be efficient, while also being TCP friendly to other flows in heterogeneous networks [22].

The congestion window of CUBIC is calculated in the following manner:

\[ W_{\text{CUBIC}} = C(t - K)^3 + W_{\text{max}} \]  

where \( C \) is a scaling factor, \( t \) is the elapsed time from the last window reduction, \( W_{\text{max}} \) is the window size just before the last window reduction, and \( K = \sqrt[3]{W_{\text{min}} / C} \), where \( W_{\text{min}} \) is the reduced window size just after the last congestion event. After a congestion event, window growth function is in steady state where it grows concavely up to \( W_{\text{max}} \), after which it enters probing state and grows convexly until the next congestion event, in order to determine currently available network bandwidth.

![CUBIC congestion window growth function](image)

Figure 4. CUBIC congestion window growth function. After packet loss it is in steady state phase, and afterwards it is in max probing phase. The y axis represents the congestion window size, whereas x axis is time.

D. Hybla

Hybla [23] is designed mainly for satellite connections where large RTT and packet loss are present [24]. Basic idea is to remove the effect of large RTT by using its normalized value:

\[ \rho = \frac{\text{RTT}}{\text{RTT}_0} \]  

where \( \text{RTT}_0 \) is round trip time of the reference connection to with the desired performance.

\( \rho \) is used to calculate the cwnd of Hybla - \( W^{H} \):

\[ W^{H}(t) = \begin{cases} \rho \frac{2^{\rho t / \text{RTT}} - 1}{\ln(2)}, & 0 \leq t \leq t_{\gamma,0} \\ \rho \frac{t - t_{\gamma,0}}{\text{RTT}_0} + \gamma, & t \geq t_{\gamma,0} \end{cases} \]

This is especially useful for connections with large RTT, whereas for connection with \( \text{RTT} \leq \text{RTT}_0 \), Hybla behaves as standard TCP.

Hybla recommends use of TCP options including SACK and FACK to improve overall sending rate and also timestamps and packet spacing to improve estimation accuracy of retransmission timeout and burstiness of traffic, respectively.

When it comes to fairness, more simultaneous Hybla connections are fair to each other. And because of normalized RTT, other types of connections do not suffer if their RTT is close to \( \text{RTT}_0 \).

IV. METHODOLOGY

The testing environment consisted of two computers – one in Belgrade, SRB, and the other one – a CLOUD server in Oregon, USA. The computer in Belgrade works under Linux Ubuntu 12.04 (kernel version 3.2), with Intel Pentium IV processor @1.8GHz and 2GB of RAM. CLOUD server works under Linux Red Hat 6.3 operating system (kernel version 2.6.32), with Intel Xeon processor @2.0GHz, and 4GB of RAM. On the link between those two computers, the network bandwidth was 100Mb/s, with average RTT of 185ms.

The following TCP algorithms were tested: Reno, BIC, CUBIC and Hybla. As the most important characteristic of data transfer, the overall achieved throughput was measured. Each test consisted of 150 seconds long data transfer, after which the average throughput from that period was measured. For these tests, iperf [25] – a specialized measurement tool was used.

Large BDP on the tested link required enhancements to the operating system TCP variables in order to test the algorithms without operating system limitations. Default values of those TCP variables yielded performance limited to just 10Mb/s.

Tests were performed 500 times for each of the tested algorithm at different times of the day, with minimal interval between successive tests of 10 seconds. This was done in order to more realistically demonstrate algorithmic behavior on a highly unpredictable, public network such as the Internet. In the end, distributions of achieved throughputs are illustrated for each of the four tested algorithms, along with minimum, maximum and average value of those 500 test results.

V. RESULTS AND DISCUSSION

In order to examine with great detail the behavior of various TCP algorithms under the influence of large BDP
in real world, CLOUD server, located in Oregon, USA, was chosen.

Reno performed poorly, which was expected, considering that it was not designed for networks with large BDP. In conducted tests, Reno averaged 15.8Mb/s, which is severe underutilization of available network bandwidth.

BIC averaged 60.9Mb/s, with great stability. Its aggressiveness yielded the best overall performance along with highest maximum speed recorded. This was done at the cost of being unfriendly to other TCP flows, which is an important factor in public networks.

CUBIC, a modification of BIC, designed with the purpose of enhancing TCP friendliness, expressed lower aggressiveness than its predecessor, which deteriorated its overall performance. But, its average of 55.6Mb/s is rather good considering large dispersion of tests results.

Hybla averaged 60.3Mb/s, with the smallest exhibited variance in test results. Even though it is designed primarily for satellite connections it performed well in these tests. It is extraordinary that it can perform aggressively, and with high stability while being reasonably TCP friendly.

VI. CONCLUSION

It is obvious that the Reno is deprecated. Current network conditions, primarily large BDP, demand changes in dealing with congestion, most of all, more aggressiveness in TCP behavior when the link is non-congested. Some of the new solutions were presented and all three of them showed great promise on resolving the problem. However, Hybla produced the best overall results.

Our current work focuses on creating an adaptive solution, which can even more reduce the effects of large BDP. Current version of this algorithm has a different approach to congestion, and it is still in development.
phase, but preliminary results show 15% better performance than that of Hybla.

**ACKNOWLEDGMENT**

We would like to thank Prof. Dr. Aleksandar Jovanović, coordinator of our Group for Intelligent Systems, for useful discussions on the topics, and all his help.

**REFERENCES**


