Designing a New TCP Based on FAST TCP for Datacenter

Feng Zheng
Department of Electronic Engineering
Tsinghua University
Beijing, China 100084
Email: zheng-f11@mails.tsinghua.edu.cn

Yongfeng Huang
Department of Electronic Engineering
Tsinghua University
Beijing, China 100084
Email: yfhuang@.tsinghua.edu.cn

Donghong Sun
Network Research Center
Tsinghua University
Beijing, China 100084
Email: sundh@cernet.edu.cn

Abstract—TCP incast problem has become a severe problem in datacenters due to the catastrophic collapse of goodput at the receiver side. Many solutions have been proposed, however, none of them solved it fundamentally. In this paper, we take a deep dive into the causes of incast problem, and find out that the root cause of TCP incast is the droptails induced by TCP congestion control algorithm based on packet losses. According to the root cause, we design a new TCP based on FAST TCP, a delay based congestion control algorithm, to radically solve TCP incast problem. The new TCP aims at maintaining a relatively small queue length that does not exceed the switch buffer and meanwhile fully occupy the bottleneck link. The simulation results demonstrate that our method cuts off most of the timeouts and attains high goodput under various conditions.

I. INTRODUCTION

TCP incast [1]–[3] has become a significant problem in datacenters recently with the development and wide usage of cluster storage [4], [5] and cloud computing [6]. Incast is a communication pattern, termed in [4]. Figure 1 shows the typical topology of incast scenario, a client requests data blocks from many servers via a switch with small buffer size, and the senders are not permitted to send the next block until all of them finish transmitting the current data block. There are 2 essential preconditions of TCP incast:

1) Network environment: High bandwidth, low latency and small switch buffer;
2) Traffic pattern: Synchronized-block transmission pattern with relatively small block size. The client does not send new requests until all data blocks are successfully received.

The goodput suffers a drastic reduction as the number of servers increases which is called the TCP incast collapse. The incast collapse potentially arises in cluster storage, web search [7] and Map/Reduce [8] applications especially during the shuffle stage.

Many methods have been proposed to solve TCP incast problem, but none of them managed to solve TCP incast fundamentally. In this paper, we understand TCP incast thoroughly. The causes of incast and their logical relations are obtained. Based on the analysis, we figure out the dominant cause: the unavoidable droptails over the switch buffer due to the congestion control mechanism of TCP based on packet losses. We then present a new TCP based on FAST TCP [9], a delay based congestion control algorithm, to avoid the occurrence of droptails, hence radically solves TCP incast.

The remainder of the paper is organized as follows. In Section II, related works are presented and discussed. In Section III, we analyze incast problem thoroughly and present the new TCP including our rationales and designing details. In Section IV, simulations are conducted in incast scenario based on the ns-2 platform. We mainly illustrate the performance of the improved FAST TCP, TCP newreno and TCP newreno with $\text{RTO}_{\text{min}} = 200\mu s$. Finally, the paper is concluded in Section V.

II. RELATED WORK

Many methods have been proposed to solve TCP incast problem. In [2], several techniques are proposed including alternative TCP implementations, reduced duplicate ACK threshold, and disabling TCP slow start. However, most of these methods are ineffective. [10] focused on reducing the waiting time of timeouts by lessen $\text{RTO}_{\text{min}}$, which is proved to be practical and effective. But their method only mitigates the impact caused by TCP incast instead of fundamentally solving it. And meanwhile microsecond-granularity $\text{RTO}_{\text{min}}$ might lead to spurious retransmission causing potential problems.

DCTCP [11] focuses on reducing the occupation of switch buffer. It modifies the threshold of ECN and use ECN as the
signal of network congestion. DCTCP outperforms TCP when
the server number is small, but eventually converges the per-
formance to TCP when server number increases to e.g., over
35. Also, it requires modifications at both sender and receiver
side which is too costly. ICTCP [12] aims at avoiding packet
losses before incast congestion by modifying TCP. It uses
goodput increase estimation to control the receive window.
ICTCP modifies only the receiver side and outperforms TCP,
but the average goodput is restricted below 70% of the link
capacity due to its mechanism.

Solutions at application layer are also proposed, including
staggering data transfer or globally schedule the data transfer
[13]. These methods cannot be general solutions because
schemes differs in different applications. There are as well
data link layer solutions using EFC [3] or QCN [14], both of
which are difficult to implement.

As will be shown below, we believe that the root cause
of TCP incast is the droptails over the switch buffer caused
by packet loss based congestion control algorithm. Some
solutions did not concern about the root cause, including
reducing $RTO_{min}$ and solutions at application layer or data
link layer. DCTCP and ICTCP concerned about the root
cause of incast, but their performance are restricted below
the bottleneck link capacity due to their mechanisms. The
new TCP proposed in this paper deracinates the droptails and
fundamentally solves TCP incast, and keeps a high goodput
under various conditions.

III. A New TCP For Datacenter based on FAST TCP
A. Finding The Root Cause of TCP Incast

In this section, we investigate the causes of TCP incast from
surface to depth, and figure out the root cause of TCP incast.

The direct cause of TCP incast problem is the long idle
link time resulted from the timeouts. In the synchronized-
block communication pattern, a timeout occurs in any of the
senders will lead to a universal waiting of all the senders,
thus the bottleneck link is vacant during the timeout period.
By default, TCP experiences a timeout that lasting for 200ms.
However, the transmission time of each block is no more than
10ms, much smaller than a single timeout. The relatively long
timeout induces the collapse of goodput.

After examining lots of experimental results, we figured out
that there are two types of timeouts leading to incast problem:
Head-TOs and Last-TOs.

The Head-TOs occurs at the head of a data block. We define
the time slot from the moment the senders start transmitting
a new block to the moment that all the senders finish their
current data block as one round. By the end of a round, some
of the flows finish their blocks earlier, due to the unfairness
of TCP. These flows will keep their cwnds and wait until the
next round. Then the other flows occupy the available bandwidth
of the bottleneck with a growth of cwnd. At the beginning of
the next round, the summation of the cwnd values will be very
large and exceed the switch buffer, causing a severe buffer
over flow at the next round. The drastic droptails will very
likely induce full window losses of flows with small cwnd.

This type of timeout is called Head-TOs. With the growth
of the sender number, the cwnd value of each flow decreases,
rising the probabilities of Head-TOs among all flows.

Figure 2 illustrates the cwnd values of N = 8 flows during
a particular time slot in incast scenario. The synchronized
block size $S = 256KB$, buffer size $B = 64KB$, round trip time
$D=100us$ and link capacity $C = 1Gbps$. The space between
each two adjacent curves is the cwnd value of a flow, thus
the curve on the top shows the cwnd summation of all 8
flows. Each round without timeouts lasts $SN/C = 16ms$.
We notice that every 16ms there is a burst of cwnd and
unfortunately enough, at the time of 0.385s, a Head-TO occurs,
leading to a 200ms idle link time. The simulation results attest
our analysis. Head-TOs derive from the unavoidable droptails
caused by the nature of TCP congestion control algorithm,
and deteriorate due to the unfairness and competitiveness of TCP.

The Last-TOs are caused by insufficient duplicate ACKs
before fast retransmission stage at the tail of synchronized
blocks. If a packet or more of the last 3 packets of a block are
dropped due to the buffer overflow, less than 3 packets will be
sent sequentially hence no enough duplicate ACKs will return
to trigger the data-driven recovery. Then occurs a timeout. In
figure 2, we can observe that the total congestion window
size hovers around the network capacity, CD + B = 76.5KB,
and overwhelms the capacity frequently. If unfortunately the
dropped packets fall in the last 3 packets of a block, a Last-
TO occurs. Last-TOs are also primarily caused by droptails
over the switch buffer, and the potential precondition is that
the default duplicated ACK threshold is 3.

Figure 3 shows the normalized idle link time caused by
Head-TOs and Last-TOs with TCP newreno. The switch buffer
is 64KB, the synchronized block size is 256KB and bandwidth
is 1Gbps. The two types of timeouts lead to more than 80%
idle link time when server number rises to 10. The goodput
collapse coincidents well with the rising of idle link time.

As a summary, unavoidable droptails fundamentally engen-
under the two types of timeouts and the unfairness and competitiveness of TCP worsen one of them. Increasing number of timeouts together with the relatively large $RTO_{\text{min}}$ leads to a large amount of idle link time which finally brings TCP incast goodput collapse. Figure 4 presents the dependencies among these causes. The theoretical basis of most former solutions goodput collapse. TCP newreno, sychronized block size $S = 256$KB, buffer size $B = 64$KB, link capacity $C = 1$Gbps 

### FAST TCP

FAST TCP is a TCP congestion control algorithm developed by Netlab, CalTech targeted at large bandwidth, high latency networks, e.g., the backbone. It uses queueing delay instead of packet loss as a congestion signal. FAST TCP periodically updates the congestion window based on the average RTT and average queueing delay according to:

$$w \leftarrow \min \left\{ w, \frac{w C}{2} \right\} \left( 1 - \gamma \right) w + \left( \frac{\text{baseRTT}}{\text{avgRTT}} w + \alpha(w, qdelay) \right)$$  \hspace{1cm} (1)

$$\gamma \in (0, 1], \text{baseRTT}$$ is the minimum RTT observed so far, and $\text{avgRTT}$ is the average RTT since the first packet of the session. $\alpha(w, qdelay)$ is the expected number of packets a flow maintains in the bottleneck buffer. In current implementations, $\alpha$ is a constant at all times. The goal of FAST TCP is to keep a fixed queue length in the bottleneck buffer for each flow.

Our design contains three parts: slow start, congestion avoidance and retransmission mechanism, constituting a congestion control algorithm as a whole. As for the retransmission mechanism, we follow the design of TCP newreno, including the fast retransmission and recovery mechanisms, since the retransmission strategy of newreno is proved very effective and robust. In the following, we mainly present our design for the slow start stage and congestion avoidance stage of the new TCP.

1) Slow Start: In the slow start stage, $cwnd$ increases by 1 for every successfully transmitted packet, hence increases exponentially per window. The key parameter of the slow start stage is the slow start threshold $ssthreshold$. In TCP, $ssthreshold$ is a certain $cwnd$ value. As for our design, the metric is queueing delay. The sender enters congestion avoidance stage while the queueing delay detected exceeds the slow start threshold:

$$w_{\text{new}} = \begin{cases} \frac{w_{\text{old}} + 1}{f(w_{\text{old}}, qdelay)}, & qdelay \leq ssthreshold \\ f(w_{\text{old}}, qdelay), & qdelay > ssthreshold \end{cases}$$  \hspace{1cm} (2)

$f(w_{\text{old}}, qdelay)$ is the function of the new $cwnd$ in congestion avoidance stage. It may vary in different protocols. The physical meaning of $ssthreshold$ is the point we think the network is fully occupied and is starting to be congested. In the case of backbone, an RTT is usually about 200ms, so for FAST TCP the default value of $ssthreshold$ is 1ms or larger. In datacenters, an RTT only lasts 10us for gigabit network and 10us for 10 gigabit network. We need to figure out a suitable $ssthreshold$ for datacenter. As all senders are detecting the same queueing delay dominant by the switch, two conditions should be satisfied: First, before the slow start ends, there must be a positive queue length to ensure that the bottleneck link is at its capacity. Second, before the slow start ends, no buffer overflows should emerge. Then $ssthreshold$ should meet the following condition:

$$1 \text{pkt}/C < ssthreshold < B/C$$  \hspace{1cm} (3)

In gigabit network, $C = 1$Gbps, usually $1 \text{pkt} = 1$KB, if the buffer size $B = 64$KB, $ssthreshold$ should fall in $[8us, 512us]$. In fact, we prefer a small $ssthreshold$ rather than a large one.
Because if the switch buffer already holds a long queue, the network becomes vulnerable for the impending congestion avoidance period. A small \( ssthreshold \) below \( B/2C \) will be much safer. And our experiment results showed that the goodput is not very sensitive to \( ssthreshold \) when it is set in the recommended range. In our scheme, we set

\[
ssthreshold = 20us
\]  

(4)

This value meets Formula (3) even when the buffer size is as small as \( 4KB \), hence will be suitable for almost any network conditions.

2) Congestion Avoidance: In the congestion avoidance stage, we periodically updates the congestion window based on the queueing delay. The topologies of the backbone and datacenter are similar, merely the routers are replaced by switches in datacenters. So we do the same math at both the flow level and packet level as FAST did [9]. Thus in the new TCP, we keep the form of the congestion window formula unchanged, as shown in Formula (1).

To satisfy the small buffer size in datacenter, the core factor \( \alpha(w,qdelay) \) is redesigned. \( \alpha \) is a coefficient of correction in the congestion window formula, it is a function of congestion window size and queueing delay. The physical meaning of \( \alpha \) is the queue length the flow expects to maintain. Currently in FAST TCP, \( \alpha \) is set to a constant in backbone usage and the default value is \( 200pkts \). To obtain the appropriate function for \( \alpha(w,qdelay) \) in datacenters, we first set \( \alpha \) to a constant, and then explore the optimized function of \( \alpha(w,qdelay) \) via theoretical reasoning and experimental observations.

Figure 5 illustrates the goodput of the new TCP with different \( \alpha \). We notice that, when \( \alpha \) is relatively small, the new TCP attains a good performance at large server numbers, but suffers a goodput penalty when server number is small. On contrary, when \( \alpha \) is large, our method performs well with small server numbers and suffers incast problem as the server number rises.

Then we consider the function of \( \alpha \). Firstly, Though \( \alpha \) is the expected queue length a flow tends to maintain, a small \( \alpha \) might slow down the convergence of the congestion window and lead to very small \( cwnd \) values. As a consequence, the total \( cwnd \) fail to fully occupy the bottleneck link at a small server number causing a goodput penalty. Figure 5(a)(b) show the situation, when \( \alpha = 0, 1 \), the network suffers a goodput penalty when server number is lower than 7. We examined the trace files and found out that \( cwnd \) values of most flows are less than 2 and are forced to be 2 because it is the minimum \( cwnd \) for FAST TCP. In fact, the total congestion window size \( Nw \) should be larger than the bottleneck link capacity in one RTT to cram the bottleneck, that is \( Nw > CD = 12.5pkt \). If \( w \) is the minimum, \( N \geq 7 \), which is identically the threshold shown in Figure 5.

Secondly, when \( \alpha \) is relatively large, FAST performs well with small server numbers. But as the server number \( N \) rises, the total queue length \( N\alpha \) rises as well. When \( N\alpha \) exceeds the buffer size, or \( N\alpha > B \), inevitable incast problem takes place, as shown in Figure 6(c)(d).

As a conclusion, for small server numbers, we apply a large \( \alpha \), and for large server numbers, we apply a small \( \alpha \). We adjust \( \alpha \) according to:

\[
\alpha = \begin{cases} 
\alpha_0 & N \leq N^* \\
\alpha_1 & N > N^* 
\end{cases}
\]  

(5)

1) \( N^* \), the threshold identifying small server numbers versus large ones. The lower bound of \( N^* \) is \( N_{min}^* = \min\{N : Nw_{min} \geq CD = 12.5pkt \} = 7 \), where \( w_{min} = 2 \) for FAST TCP. We should notice that \( CD \) is always 12.5pkt because for gigabit ethernet, \( C = 1Gbps \) and \( D = 100us \), and for 10 gigabit ethernet, \( C = 10Gbps \) and \( D = 10us \). The upper bound for \( N^* \) is \( N_{max}^* = B/\alpha_0 \), \( \alpha_0 = 2 \), as shown below, and for most switches, \( B > 32 \), thus there is a large "safe zone" for \( N^* \): \( N^* \in [7, B/2] \), \( B > 32 \);

2) \( \alpha = \alpha_0 \) when \( N < N^* \), \( \alpha_0 \) should be large enough to pull the congestion window to the equilibrium point and fully engage the bottleneck link. According to our experimental results, any \( \alpha_0 \in [2, 5] \) performs well. But we suggest \( \alpha_0 \) to be as small as possible. Because the capacity of the bottleneck seldom changes, hence \( N_{min}^* \) stays the same. But the buffer size might change if the switch is substituted, in case a smaller buffer is utilized, a large \( \alpha_0 \) might lessen \( N_{max}^* \) and even cause incast collapse before \( N^* \). In our scheme, \( \alpha_0 = 2 \);

3) \( \alpha = \alpha_1 \) when \( N > N^* \). \( \alpha_1 \) should meet \( N\alpha_1 < B \). \( \alpha_1 = 0 \) seems to be a easy choice, but we prefer \( \alpha_1 = 1 \) because we believe that maintaining a queue length of 0 is unsafe. On the other hand, the minimum congestion window of FAST TCP is 2pkt, making the upper bond of server number \( (CD+B)/2 \). Usually, buffer size \( B \) is much larger than \( CD \), the packets transmitted in a RTT, so the upper bond lies below the incast threshold \( C/\alpha_1 \) when \( \alpha_1 = 1 \).

Now we get the mapping between \( \alpha \) and server number. But

Fig. 5. The new TCP with static \( \alpha \) utilized in incast scenario
transmission control protocols are not able to detect the server number. Fortunately current server number can be estimated by queueing delay, because \( \alpha \) is the expectation of queue length of each flow, dividing queue length by \( \alpha \) should generate current server number:

\[
\text{Server\#} = qC/\alpha
\]

Now the metric of small-large threshold becomes \( q \text{delay} \). And the threshold \( q^* = N^*\bar{\alpha}/C \). Notice that we apply \( \bar{\alpha} = (\alpha_0 + \alpha_1)/2 \) as a approximation of \( \alpha \) in case of gigabit ethernet that resolves the contradiction at the inflection point as well as keeps \( N^* \) in the "safe zone". The function of \( \alpha(w, q - \text{delay}) \) is as follows:

\[
\alpha = \begin{cases} 
  2 & q \leq N^*\bar{\alpha}/C \\
  1 & q > N^*\bar{\alpha}/C 
\end{cases}
\]

and in gigabit ethernet, \( \bar{\alpha} = 1.5 \).

Figure 6 shows the total cwnd of the new TCP in incast scenario when 8 flows inject their data through a switch with 64KB buffer. Comparing with the situation of TCP newreno showed in Figure 2, the total cwnd value is much smaller than the network capacity. No buffer overflow occurs, and the positive queueing delay guarantees the full utilization of the bottleneck link. The new TCP resolves the root cause of incast.

### IV. PERFORMANCE EVALUATION

#### A. Simulation Environment

We use ns-2 to do the simulation. ns-2 is an event-driven network simulator and models the applications at packet level. We simulate one client and multiple servers connecting to the same switch, as is shown in Figure 1. The round-trip-times without queueing delay is 100\(\mu\)s. Each link has a capacity of 1Gbps. Default \( RTO_{min} \) for the new TCP and TCP newreno is 200\(\mu\)s. Our simulation application performs the incast communication pattern and we implement the new TCP in this system. The sender number varies from 1 to 31. Each measurement runs for 10 second of simulated time.

We choose TCP newreno, which is currently widely used, as our benchmark. And compared the performance of our new TCP with TCP newreno under different traffic and network conditions.

#### B. Performance Analysis

Figure 7 shows the normalized waiting time of TOs and the normalized numbers of timeouts per block. The TO time of TCP newreno with \( RTO_{min} = 200\mu\)s is less than that of TCP newreno, but the number of timeout events per block is much larger. Because reducing \( RTO_{min} \) do reduce the waiting penalty of timeouts, but a large amount of spurious timeout emerge. And this is why reducing \( RTO_{min} \) to less than 1\(\mu\)s is regarded as unsafe. We can see that the the new TCP has both the least idle link time and timeout events per block, indicating that our method radically reduces probability of
timeout occurrence.

Figure 8 presents the goodput of the new TCP, TCP newreno and TCP newreno with $RTO_{\text{min}} = 200\mu s$. The switch buffer is 64KB. We can observe that the new TCP outperforms the other two mechanisms with different block sizes. The goodput collapse when the number of sender increases for TCP newreno. As for TCP newreno with $RTO_{\text{min}} = 200\mu s$, incast problem is mitigated, but the network capacity is not fully utilized and the percent of usage decreases as the sever number increases. The new TCP, however, maintains high goodput. Figure 8 indicates that, the new TCP has a good performance under different traffic conditions.

Figure 9 exhibits the goodput of the New TCP with different buffer size, comparing with TCP newreno. TCP newreno shifts right its occurrence point of incast as the buffer size grows, but no essential changes are made, while the new TCP keeps a high goodput all the time. This experiment exhibits that the new TCP performs well under different network conditions.

V. Conclusion

In this paper, we analyze TCP incast in depth, and find out the root cause of incast. Based on the analysis, we design a new TCP based on FAST TCP. The new TCP updates its congestion window based on the queueing delay. It maintains a certain queue length at the switch buffer for each flow and keeps the total queue length below the buffer size. With this mechanism, the new TCP avoids the droptails over switch buffer, hence fundamentally solves TCP incast. The simulation results validate that the new TCP cut off most of the timeouts and outperforms TCP newreno and TCP newreno with $RTO_{\text{min}} = 200\mu s$, under different traffic conditions and network configurations.

Acknowledgment

This work was supported in part by the National High Technology Research and Development Program of China (863 Program)(NO.2011AA010704) and the Use Case Study for Solving Incast Problem in Typical DC Application.

References