Understanding Synchronization in TCP Cubic

Sonia Belhareth*, Dino Lopez-Pacheco†, Lucile Sassatelli†, Denis Collange*, Guillaume Urvoy-Keller†
*Orange Labs, Sophia Antipolis, France
† Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis
Emails: {sassatelli,lopezpac,urvoy}@i3s.unice.fr, {sonia.belhareth,denis.collange}@orange.com

***Abstract*—TCP Cubic is designed to better utilize high bandwidth-delay product paths in IP networks. It is currently the default TCP version in the Linux kernel. Our objective in this work is to better understand the performance of TCP Cubic for scenarios with a large number of competing long-lived TCP flows, as can be observed, e.g., in cloud environments. In such situations, Cubic connections tend to synchronize each other and this synchronization is higher than with TCP New Reno. We investigate this phenomenon in detail through experimentations in a controlled testbed, measurements with Amazon EC2's servers, located in the US and simulations.**

We demonstrate that several factors contribute to the appearance of synchronization in TCP Cubic: (i) the rate of growth of the congestion window when a Cubic source reaches the capacity of the network (that it might over- or under-estimate) and its relation to the RTT of the connection, (ii) the way the congestion Cubic tracks the ideal cubic curve in the kernel (as the congestion window grows in a discrete fashion in units of MSS while the cubic curve assumes a fluid window), (iii) the competition among the Cubic sources and the aggressiveness of the sources that did not experience losses during the last loss episode.

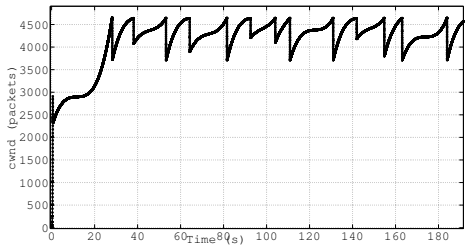
We also propose and evaluate several modifications to the TCP Cubic algorithm to alleviate the amount of packets lost during the synchronization episodes.

I. Introduction

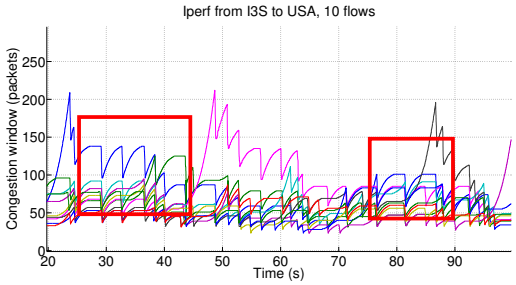
Massive data transfers are the norm in typical cloud scenarios, either within the data center itself or between the data center and the customer premise. In such a scenario, the transport layer, namely TCP, is put under pressure and might suffer performance problem, e.g., the TCP incast problem observed when a large number of storage devices simultaneously send data chunks to a single machine leading to congestion at the switch servicing the machine [1].

Cloud environments are characterized by plenty of bandwidth. Modern versions of TCP such as Cubic, are designed to work efficiently in such situations as they are able to probe for available bandwidth in a non linear fashion, unlike TCP New Reno, which inflates its windows by one MSS per RTT in stationary regime. However, there is a price to pay for being more aggressive: the fairness offered by Cubic and other high speed versions of TCP is not as high as legacy TCP versions [2]. Several studies also pointed out the appearance of synchronization among competing Cubic flows [3]. The latter means that Cubic flows, when competing for a bottleneck, tend to loose packets at the time instant and the resulting aggregated throughput time series exhibit a clear Cubic behavior as if a single flow was active.

In this work, we investigate the issue of synchronization among TCP Cubic sources in detail. As a motivating example of the problem, consider the time series of an average congestion window of 10 Cubic TCP flows established between the same pair of sender/receiver that compete for a shared bottleneck, obtained with ns-2, in Figure 1a. The clear Cubic shape that appears regularly indicates that the flows are synchronized (for more details on TCP Cubic, see Section III). Note that the code of Cubic in ns-2 is a fork of the one in the Linux kernel. One can obviously argue that the simulations set-up does not catch the complexity of a real operational IP network, and thus synchronization might be the result of idealized simulation conditions. This is why we present in Figure 1b the congestion window evolution of 10 transfers in parallel between a server in Amazon data center of Oregon and a server in our lab. We have obviously no control on the path, but we can clearly observe two periods highlighted by red rectangles where the majority of flows are synchronized.



(a) Average window size (ns-2): 10 TCP Cubic flows,sharing a common bottleneck



(b) 10 TCP Cubic transfers between I3S and Amazon EC2 data center of Oregon

Fig. 1: Synchronization in TCP Cubic

We study the extent and the root causes of synchronization using an experimental approach with a testbed hosted in our lab combined with simulations. The former enables to experiment with actual protocol implementation in a controlled environment while the latter permits to explore a wider set of network scenarios.

Our contribution to the study of the synchronization of TCP Cubic are as follows:

- We experimentally establish the relation between the existence and extent of synchronization with key parameters like RTT and buffer size. We demonstrate the resilience of synchronization to background traffic, and how the Fast Convergence option, which aims at making Cubic more fair to fresh connections, catalyzes synchronization. For this point and the subsequent ones, we use New Reno as a reference point.
- We demonstrate that several factors contribute to the appearance of synchronization in TCP Cubic: (i) the rate of growth of the congestion window when a Cubic source reaches the capacity of the network and its relation to the RTT of the connection, (ii) the way the congestion Cubic tracks the ideal cubic curve in the kernel, (iii) the competition among the Cubic sources and the aggressiveness of the sources that did not experience losses during the last loss episode.
- We propose and evaluate two approaches to reduce the level of synchronization and hence the loss rate of TCP Cubic transfers. Perhaps more importantly, we provide hints that synchronization is the price to pay to have a high-speed TCP version that needs to explore the available bandwidth in the network in a super-linear manner. It is probable that we can alleviate synchronization, as our modifications of TCP Cubic do, but eliminating it out completely will be a complex task.

II. Related Work

A. TCP Cubic

Various congestion control strategies for TCP have been designed to meet the ever-changing networking requirements of the Internet, especially Cubic TCP [4], which is the default TCP version in recent Linux kernels, Fast TCP [5] or CTCP [6].

The focus of this paper is on Cubic, which is characterized by a Cubic window growth function [4]. The aim of Cubic is to achieve a more fair bandwidth allocation among flows with different RTTs (round trip times) by making the window growth independent of the actual RTT.

In order to understand how Cubic behaves in data centers, we designed a fluid model for Cubic, presented in [7], that allows to predict various metrics, such as distribution of the window sizes of N long-lived competing connections, throughput, RTT, loss rate and queue size. The model is validated against ns-2 simulations in typical cloud scenarios. The fit is very good when Cubic operates in TCP mode, while it is less satisfactory when in pure Cubic mode. The reason behind this latter observation is that our model does not capture the high synchronization among the competing flows.

B. Synchronization

In [3], Hassayoun and Ros found that high-speed versions of TCP may be prone to strong packet-loss synchronization between flows. The authors studied several high speed version of TCP and observed, through simulation, the existence of synchronization among sources for all flavors of high-speed TCP.

In [8], the same authors evaluate the potential impact of the Random Early Detection (RED) queue management algorithm on high-speed TCP versions. They study the relation between buffer size, active queue management and loss synchronization. Their study focuses on several metrics: loss synchronization, goodput, link utilization, packet loss rate, and convergence to fairness for high-speed flows. For large buffers, RED strongly reduces the synchronization rate as expected, whereas with droptail, the fraction of synchronized sources is often close to 100%. In contrast, with medium to small buffers, the loss synchronization is roughly similar with both types of queue management strategy.

III. Background on TCP Cubic

A. Window variation in TCP Cubic

When in congestion avoidance, TCP Cubic features two modes of operations, the so-called TCP and Cubic modes [4]. The TCP mode is to be used in low bandwidth delay products (BDPs), while the Cubic mode is triggered for high BDPs. Each mode corresponds to a specific way of increasing the window size and is determined by the following pair of equations:

$$w_c(t) = C(t - V_{Cubic})^3 + w_{max} \quad (1)$$

$$w_{tcp}(t) = w_{max}(1-\beta) + \frac{3\beta}{(2-\beta)}\frac{t}{R(t)} \quad (2)$$

where w_{max} is the congestion window prior to the last loss event[1], $R(t)$ is the estimated RTT of the connection, β and C are constant values usually set to 0.2 and 0.4, respectively, and $V_{Cubic} = \sqrt[3]{\frac{\beta w_{max}}{C}}$. The congestion window size $cwnd(t)$ is set to $max(w_c(t), w_{tcp}(t))$ upon each ACK reception. TCP Cubic is thus said to operate in Cubic mode (resp. TCP mode) if the maximum is $w_c(t)$ (resp. $w_{tcp(t)}$).

The equation of $w_c(t)$ is designed in such a way that when a TCP Cubic connection is operating in the cubic mode, it converges quickly to w_{max}. Then it plateaus for a while, before increasing again to probe the link to sense whether more bandwidth is available in the path (see Figure 2).

Upon detection of a loss, w_{max} is set to the last congestion window $cwnd(t)$, before the congestion window be reduced by 20%. In case the last w_{max} was larger than the congestion window when the loss is detected, and if the Fast Convergence mechanism is applied, w_{max} is set to $0.9 * cwnd$. This is what happens in Figure 1a where we observe that the plateau is sometimes at a lower level than the maximum.

Another major difference between Cubic and previous TCP versions is that the congestion window increase is not correlated to the RTT. Indeed the amount of packets by which the congestion window must be increased depends only on the time elapsed since the last congestion event. In contrast, with standard TCP, flows with very short RTTs increase their congestion windows faster than flows with longer RTTs.

Concerning the TCP mode, we can note that $w_{tcp}(t)$ depends both on the RTT of the connection and the time

[1]Note that w_{max} is varying over time but is constant between two loss events. This is also the case for V_{Cubic}.

elapsed since the last loss event. Thus, in practice, when the RTT is low, $w_{tcp}(t)$ ensures that the window increase of TCP Cubic is not slower than one of New Reno.

B. TCP Cubic mode of operation

TCP operates either in TCP or Cubic modes. The Cubic mode depends on the bandwidth delay product of the path, through the value of $w_{\max}$, while the TCP mode depends on the RTT. The net result, for a given path with a minimum latency $RTT_{\min}$, is that TCP Cubic operates either in TCP or in Cubic mode. We can observe an alternation of modes if $RTT_{\min}$ is below the threshold that triggers Cubic while it is above when the buffer starts filling up and the RTT increases. At 100 Mb/s, the latency of the path that ensures that TCP is always in Cubic mode is 39 ms, while at 1Gb/s, it is 18 ms.

To find those values, one needs to consider the difference $D(t, RTT, w_{max}) = w_c(t) - w_{tcp}(t)$. We can see in Figure 2 that this difference first increases with t, then decreases and increases again. We set RTT to $RTT_{\min}$ as $w_{tcp}(t)$ decreases with an increasing RTT. The minimum of the function is obtained for:

$$t_0(RTT, w_{max}) = \left(\frac{\beta}{C(2-\beta)RTT}\right)^{\frac{1}{2}} + V_{Cubic}\ .$$

One next finds $RTT_{\min}$ such that $D(t, RTT_{\min}, w_{max})$ is positive, which ensures that the Cubic mode dominates.

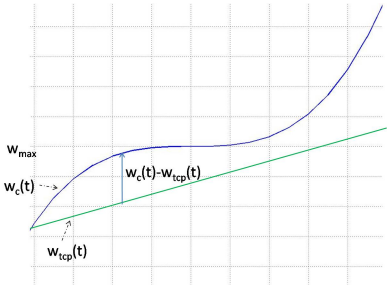


Fig. 2: Congestion window growth of TCP Cubic in Cubic and TCP modes.

IV. Experimental set-up

A. Testbed

We have created a set of experimental scenarios in our laboratory using the testbed presented in Figure 3. It consists of 3 multi-core Dell servers, 2 acting as TCP client or server and one as router. All links are 1 Gb/s links. The router uses netem [2] to control the path latency and capacity, and also the buffer size at layer 3. We use the default FIFO/droptail as server scheduling/queue management policy at the bottleneck.

Various scenarios are created by varying the latency and buffer size. We set the buffer size at the router to {10%, 30%, 50%, 100%} of the bandwidth delay product BDP (i.e., the product of the minimum latency and the capacity of the path). For each scenario, we compare the performance of Cubic with the ones of NewReno. NewReno is used here as a baseline for comparison as it is known to be less sensitive to synchronization than any high speed TCP version.

[2]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

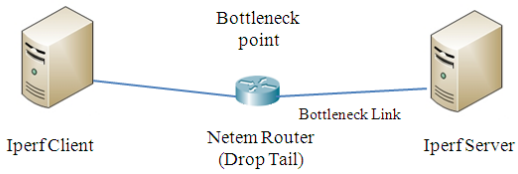


Fig. 3: Experimental network setup

B. Scenarios

We consider, similarly to [7], several typical cloud networking scenarios:

- Scenario A - Cloud-clients. We consider here a set of clients that simultaneously download content from a data center (DC). We assume that they share the 1 Gbps access link of the DC and that they have a low path latency to the DC, 20 ms (a typical latency for FTTH clients in France).
- Scenario B - Intra-DC. We consider a set of transfers within a data center (DC) where the path capacity is set to 1 Gbps while the latency is low, 1 ms, reflecting the small physical distance between the servers.
- Scenario C - Inter-DC. This scenario is similar to the previous one, except that the path latency is one order of magnitude higher. We consider 50 ms of latency.

Links between machines in our testbed are at 1Gbps. However, we cannot operate netem at such a high speed when controlling both the capacity and the buffer size. We thus constrain the capacity to 100 Mb/s and we inflate the latency of the path in such a manner that the bandwidth-delay product of the path be the same or similar to the consider scenario.

V. Synchronization in TCP Cubic

A. Cloud center scenarios

1) Scenario A (Cloud-clients): Table I contains the targeted (ideal) parameters of the scenario, as well as the ones used in our testbed due to our technical constraints. Note that we define hereafter the bandwidth-delay product of a path (BDP) as the product of the capacity of the bottleneck and the minimum latency of the path.

	Ideal parameter	Testbed parameter
Throughput	1 Gbps	100Mbps
RTT (ms)	20	200
Buffer size (packet)	50	[0.1, 0.3, 0.6, 1]* BDP
BDP (packet)	1667	1667

TABLE I: Cloud clients scenario

We vary the buffer size at the bottleneck from 10% of the BDP to 100% of the BDP for both Cubic and New Reno. We report results only for 100% BDP owing to space constraints.

Time series of the total window size of one of our experiments taken at random (which were all quantitatively and qualitatively similar), summed over all the connections, is presented in Figure 4, for both Cubic and TCP New Reno. From this figure, we note that:

- The congestion window for Cubic reaches larger values compared to New Reno. This means that the number of packets above $BDP + BS$ is larger in Cubic than in New reno, which causes more losses with Cubic.
- Cubic flows are more synchronized than New reno. This is indicated by the window reduction during loss episodes closes to 20%. Indeed, a reduction of 20% of the aggregated congestion window is only possible if all sources experience packet looses simultaneously. In contrast, in the New Reno case, flows are less synchronized giving an overall window decrease after loss clearly smaller than 50% (NewReno decreases its congestion window by 50% upon loss detection).

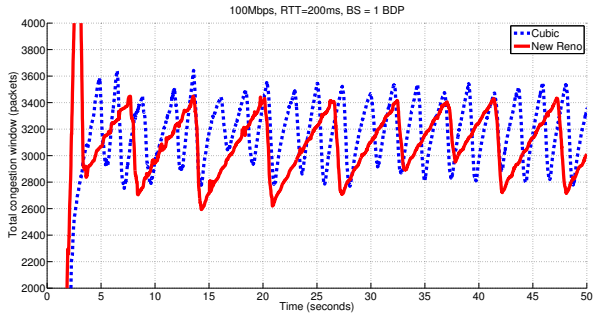


Fig. 4: Total window size

2) Scenario B (Intra-DC): Table II contains the targeted (ideal) parameters of the scenario, as well as the ones used in our testbed.

	[7] parameter	Testbed parameter
Throughput	1Gbps	100Mbp
RTT (ms)	1	10
Buffer	50	1000
BDP	84	84

TABLE II: Intra-DC scenario

The BDP for this scenario is equal to 84 packets. If one adds to it a buffer size equal to the BDP, it gives an average of 1 packet per flow which is low for our 100 flows in parallel. In such a scenario, the Linux kernel reduces automatically the MTU to values as low as 40 bytes. This phenomenon leads to different congestion window sizes to obtain a fixed bandwidth, making the analysis of results more complex. To work around this issue, we used a larger buffer of 1000 packets.

We report in Figure 5 the time series of the average congestion window of both Cubic and New Reno. Note that in this case, Cubic operates in the TCP mode, and therefore, a smaller synchronization is detected. Indeed, the reduction of the total congestion window is less than 20%. Figure 6 shows clearly that the number of losses per congestion event and synchronized flows approaches the one of New Reno TCP.

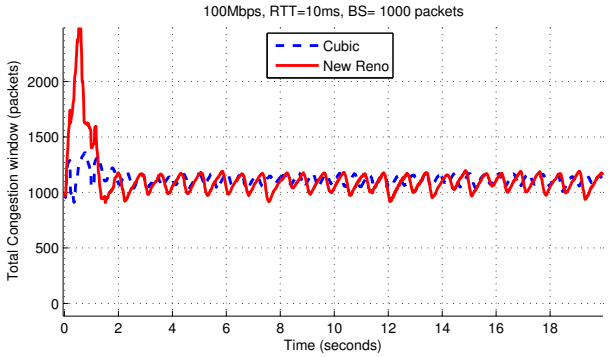
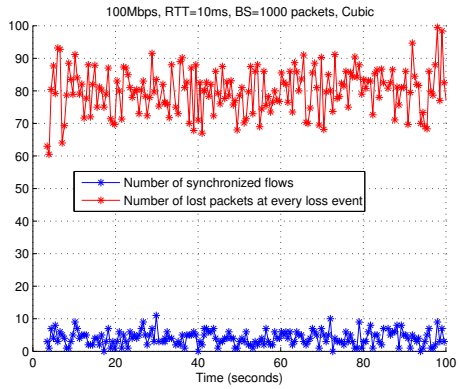
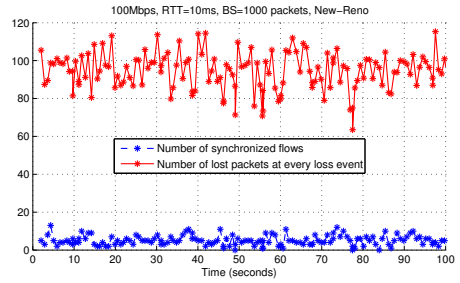


Fig. 5: Total window size



(a) Cubic, BS = 1000 packets



(b) New Reno, BS = 1000 packets

Fig. 6: Number of synchronized flow and lost packets at each congestion epoch

3) Scenario C (Inter-DC): Table III contains the targeted (ideal) parameters of the scenario, as well as the ones used in our testbed.

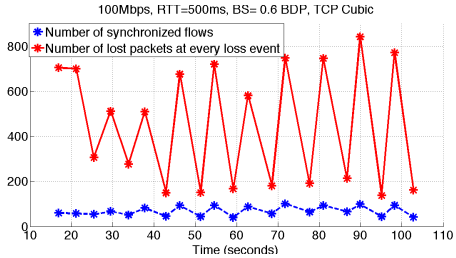
For large BDP, the congestion window growth for New Reno is much slower compared to Cubic, so we double the simulation time for New Reno to 200 seconds instead of 100. For the sake of space, we only we present the results for BS = 0.6 BDP.

With the larger BDP of this scenario, Cubic TCP operated in its cubic mode and we observe again a high synchronization

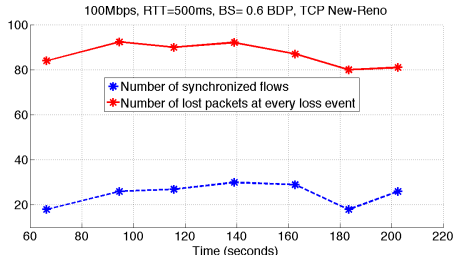
	[7] parameter	Testbed parameter
Throughput	1Gbps	100Mbp
RTT (ms)	50	500
Buffer	500	[0.1, 0.3, 0.6, 1]* BDP
BDP	4167	4167

TABLE III: Inter-DC scenario

of Cubic sources – see Figure 7, where the number of synchronized flows for Cubic is close to 100 while it is below 30 for NewReno.



(a) Cubic, BS = 0.6 BDP



(b) New Reno, BS = 0.6 BDP

Fig. 7: Number of synchronized flow and lost packets at each congestion epoch

B. Synchronization vs. background traffic

A well-known mechanism to combat synchronization consists in introducing randomness into the network. This can be done by introducing background traffic or inducing random drops through an appropriate buffer management mechanism such as RED [9].

It is known that RED can indeed break synchronization among Cubic sources [8], even though the results in [8] where obtained purely through simulation. We tested in our tesbed the resilience of synchronization to background traffic. We thus performed again experiments with Scenario C, where synchronization was highly pronounced, adding 100 short flows during the experiment. These flows are short scp transfer. They form a Poisson process with mean inter-arrival time of 1s. The files sent through scp have a size equal to 2MB.

Background traffic starts at time t = 200 seconds in Figure 8. We can notice that the overall window is reduced and reaches a value lower than 2BDP, but the Cubic shape of the average window persists, meaning that all flows are still synchronized.

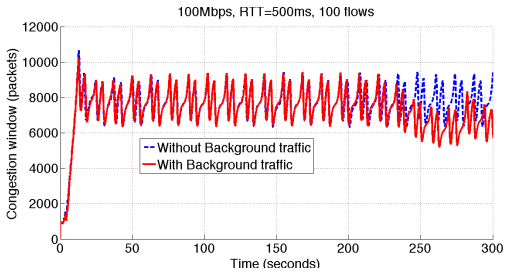


Fig. 8: Time series of window size (packets) with and without background traffic, BS= 1 BDP

C. The impact of Fast Convergence

Fast convergence (FC) is designed to make Cubic more fair as it leaves a chance to fresh flows to grab some bandwidth. It is thus not advisable to unset this option in the general case. Still, when focusing on the issue of synchronization, FC becomes a potential suspect of synchronization. Indeed, when performing FC, a source sets its $w_{\max}$ to a value lower than the estimated available bandwidth (the congestion window at the moment where loss occurs). As a consequence, when the number of flows is constant, as it is the case in our experiments, when a source performs FC, it will reach the available throughout (its share of $BS+BDP$) in an aggressive manner –see for instance Figure 1a. This aggressive behavior around the equilibrium point can make all sources (even the ones that would plateau at this level) loose some packets and thus enforce their synchronization.

To test the relation between FC and synchronization, we performed again experiments with Scenario C with an without FC for a typical run. We report in Figure 9 the average window time series and its distribution, with and without FC. As the extent of window oscillations remains similar, we can conclude that FC is not the only factor behind synchronization.

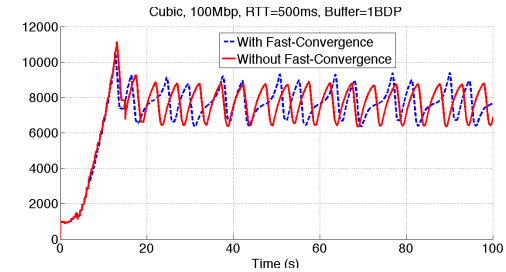


Fig. 9: Time series of total window size (packets) with and without FC, BS= 1 BDP

VI. Root cause of synchronization

As observed in Section V, TCP Cubic experiences more losses than standard TCP per congestion event. Therefore, Cubic senders have a higher probability to be synchronized.

Intuitively, high speed TCP variants are more aggressive and therefore, lead to a higher drop rate as compared to the legacy New Reno approach, where the congestion window grows linearly. While this is true for other high speed TCP protocols, like High Speed TCP, in the case of Cubic, if the

flat part of the cubic function matches the optimal network capacity, then we can expect to have (at least for this optimal scenario) a low drop rate. Indeed, Cubic is supposed to slowly enter and exit the flat part.

In fact, several key reasons explain why TCP cubic flows synchronize each other:

- First, the way TCP cubic reaches the capacity of the network, which might correspond to its equilibrium point (when the cubic curve becomes flat) or not, depending on the accuracy of the estimate made.
- Second, the way the congestion window actually tracks the cubic curve in the actual implementation can worsen the synchronization phenomenon by letting the source remains a smaller amount of time on its plateau.
- Third, the competition among TCP Cubic flows where the aggressive nature of their probing process far from the equilibrium point can lead to losses for all competing flows.

We discuss each of these points in details in the remainder of this section.

A. Behavior of TCP Cubic around the equilibrium point

Let $epochstart$ be the time right after a congestion event (i.e. $t_0 = epochstart$). Hence, at t_0, the Cubic window will be equal to $0.8 * last\_cwnd$. Using Eq. (1), we can see that theoretically, whatever the value of w_{max} and the experienced RTT are, $w_c(t)$ will reach w_{max} at $t_{max} = epochstart + V_{Cubic}$. Furthermore $w_c(t)$ will reach $w_{max}+1$, $w_{max}+2$, $w_{max}+3$ and $w_{max}+4$ at $t_{max}+1.35s$, $t_{max}+1.7s$, $t_{max}+1.95$ and $t_{max}+2.15$. Therefore, while there is 0.35s between $w_{max}+1$ and $w_{max}+2$, there is only 0.2s between $w_{max}+3$ and $w_{max}+4$. Indeed, as $w_c(t)$ moves aways from w_{max}, it increases faster. Figure 10 provides a graphical description of the period length between 2 successive expected increases of the congestion window.

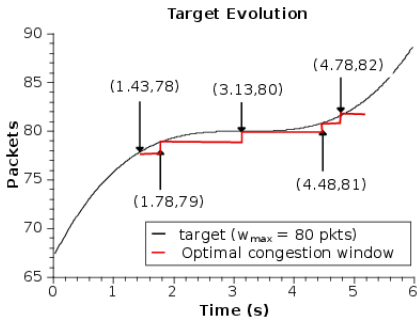


Fig. 10: Target Evolution

Consequently, three different scenarios can be drawn, based on the relative positions of the flat region and the total network available capacity, i.e. BDP+BS. We seek to understand when a source is going to send more than one packet in an RTT when reaching the network capacity. Indeed, if each source adds a single packet, like in New Reno, synchronization should be mild. If they send two packets or more, synchronization will be high.

First scenario: $w_{max} = BDP + BS$. If $cwnd = w_{max} + 1$ leads to a congestion, since between $w_{max}+1$ and $w_{max}+2$ there is a period equal to 0.35s, flows with a total RTT (i.e. propagation delay plus buffering time) smaller than 0.35s will detect the congestion at $w_{max}+1$. Flows with RTTs larger than 0.35s can potentially detect the congestion only when at $w_{max}+2$ (i.e., in a single RTT, such a Cubic flow will increase twice its congestion window). Note that whatever the RTT experienced by New RenoTCP, this last protocol is able to detect a congestion when the congestion window exceeds the total network capacity by only 1 packet, since it increases its window by at most one MSS per RTT.

Second scenario: $w_{max} = BDP + BS - 1$. When $w_c(t) = w_{max} + 2$, congestion occurs but since between $w_{max}+2$ and $w_{max}+3$ there is a period equal to 0.25s, if the total RTT is larger than 0.25s, the connection will potentially increase its congestion window twice (or more depending on the experienced RTT) ending with a congestion window equal to $w_{max}+2$ or more.

Third scenario: $w_{max} = BDP+BS+1$. If $w_c(t) = w_{max}$ already exceeds the total network capacity by one packet, since between $w_{max}-1$ and w_{max} there is a period equal to 1.35s, theoretically, only flows with an RTT larger than 1.35s will increase twice their congestion windows before detecting a congestion. Hence, after a congestion event, w_{max} will be set again to $w_{max} = BDP + BS + 1$ and the number of losses will be small. We want to highlight that the theoretical Cubic target is able to converge to a $w_{max} = BDP + BS + 1$ from any w_{max} value, like shown in Figure 11.

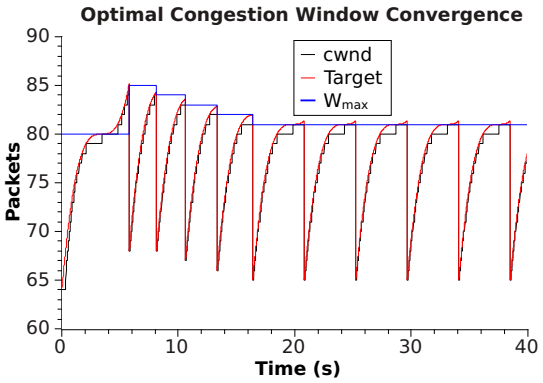


Fig. 11: Converge properties of the optimal congestion window ($BDP + BS = 80$).

To sum up the three above scenarios: (i) overestimating the bottleneck is not a big issue as there is little chance that the sources increases several times its congestion window when entering the flat region (it should be larger than 1.35s); (ii) precisely estimating the bottleneck precisely means that the source will be too aggressive if the RTT is larger than 0.35 s and (iii) if the source underestimates the capacity, the RTT for which it becomes too aggressive is 0.25s. The latter scenario is thus the more dramatic one. We can note that Fast Convergence, that forces to set its $w_{\max}$ equal to $0.9 * w_c(t)$

upon a loss leads exactly to the latter scenario. FC is thus a net contributor to the too high aggressiveness of a TCP Cubic source.

As an illustrative example, the Amazon EC2 experiment presented in Figure 1b was a case where the base RTT (measured by ping) was 190 ms. Hence, when adding the buffer size along the path (which we do not know), being above 250 ms is clearly an option. This RTT combined with the use of Fast Convergence explains why we observe episodes of synchronization.

The above analysis assumes a perfect source in isolation. In practice, the actual implementation as well as the competition among Cubic flows worsen the situation as we discuss below.

B. Tracking of cubic function in the actual implementation of TCP Cubic

In the real life, the tracking of the target window is not perfect. We have extracted the algorithm used by TCP Linux from ns-2, which is supposed to be the same as the one in some Linux kernels, to build our own simulator and be able to trace the several variables used inside. We have found that, assuming a constant reception of ACKs and a total RTT of one second, when the congestion window reaches w_{max}, it will stay in the flat region for period shorter to 1.35s (around 0.8s as we can see in Figure 12). Such a result was confirmed by ns-2 assuming the same RTT. Staying a shorter period on the plateau can lead to have too many losses when getting above the network capacity.

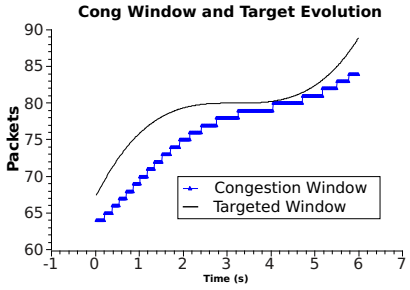


Fig. 12: More real Cubic congestion window evolution.

C. Competition around the equilibrium point

Let us suppose that during a given congestion event, the total capacity was exceeded by n packets only (where n is equal to the number of flows) as the legacy New Reno version of TCP does, and that the congestion window of each Cubic flows was equal to the actual share that each connection deserves. In this scenario, it is highly likely that not every flow would experience a packet loss. Put differently, the synchronization between flows would be low. However, those Cubic flows that did not experience losses will enter their convex region, and thus their congestion window will grow faster and during the next congestion event, the number of dropped packets will increase. This will finally lead to a high synchronization between flows. Figure 13 illustrates graphically our arguments provided in this paragraph by zooming on a specific moment in time of one simulation we performed. We observed a first loss event where only two flows are affected. We next observe that the flows that experience losses will soon again plateau around the equilibrium point. In contrast, the ones that did not loose enter the aggressive probing part of the cubic curve. Even, if they are just leaving their plateau as it is the case here, the number of losses that they induce in the buffer is such that all four sources losse packets at the same time instant, i.e., they become synchronized.

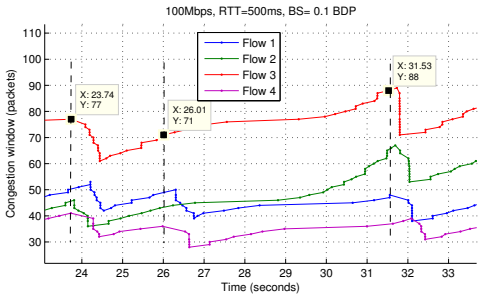


Fig. 13: Cubic leading a to a high synchronization

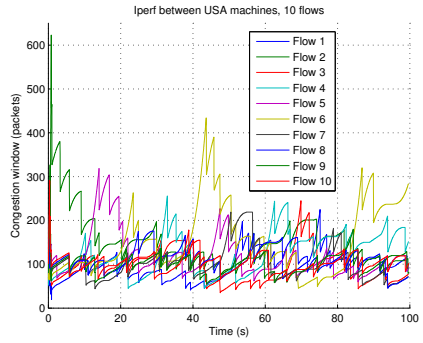
D. Discussion

From the analysis presented above, it is clear that the RTT of the connection plays a key role to determine the level of synchronization we might expect. Referring back to the methodology presented in Section V, it becomes clear, in light of what we discussed in this section, that increasing the RTT to obtain the same BDP as in the ideal cloud scenarios that we devised, was introducing a bias towards more synchronization. For the intra data-center scenario (scenario B) where the ideal RTT was 1ms, synchronization is likely not to occur. This is confirmed by our experimental results (see Figure 6a) because the RTT in the experimental testbed is still low (10ms). It should be the same in the inter data-center case (scenario C) where the ideal RTT is 50 ms, while we observed synchronization by working at 500 ms. It is even highly possible that Cubic operates in the TCP mode and not the Cubic mode in such a scenario, in which case the means-field model that we proposed in [7] demonstrated that no synchronization should be present.

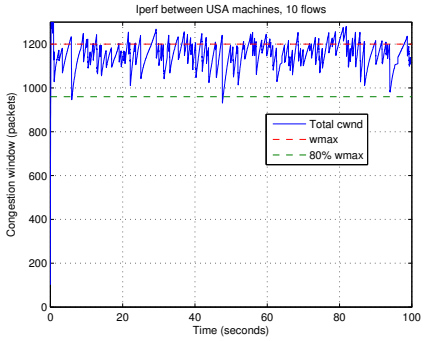
As illustrative examples of the above points, we report in Figure 14a a typical experiment made between a pair of servers in the Oregon data center of Amazon where the RTT was in the order of a ms. We never observed any synchronization in this case (out of the numerous trials we made). While Figure 14a reports the congestion window of each individual flow, Figure 14b reports the aggregate congestion window and we can observe that it never decreases by 20% (as 80% of 1200 is 960 and we are always above this line).

The previous experiment was obtained with 10 flows. With 100 flows between the pair of servers, we observe in Figure 15 that the flows now operate in the TCP mode of Cubic with no synchronization.

However for the case of a remote client or distant data centers transfers, synchronization is likely to pop up. The Amazon EC2 experiment in Figure 1b is a good illustration of this point. Additionally, since flow synchronization leads to a reduction of around 20% of the total traffic, buffer sizes



(a) Individual Congestion Window



(b) Aggregate Congestion Window

Fig. 14: Intra data center transfers - 10 flows

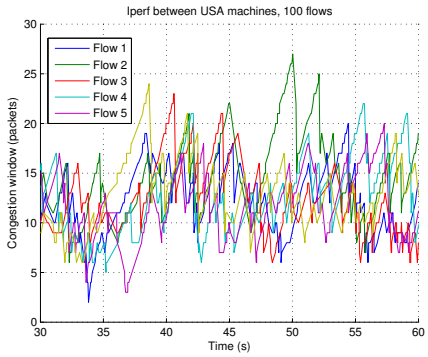


Fig. 15: Individual Congestion Window, Intra data center transfers - 100 flows

smaller than 20% of the expected average BDP can lead to an under utilization of the available bandwidth, specially if the maximum experienced RTT of the traffic exceeds 250ms.

VII. Alleviating Synchronization

In this section, we aim at investigating solutions to work around the problem of synchronization faced by TCP Cubic. As the root of the problem lies in behavior of TCP around the equilibrium point, we investigated the two following approaches:

- First, we linearize TCP Cubic when it operates close to its plateau. More precisely, we enforce TCP to increase by one MSS per RTT in the range $[w_{\max} - 2, w_{\max} + 2]$. We call this modification LinCubic.
- Second, as we observed that the actual implementation was not accurately tracking the cubic curve, we devised a version that fulfills this goal. We call this modification AccuCubic.

To evaluate the impact of those different modifications, we implemented them in ns2 and started observing their behavior in the case of a single flow. We consider a link capacity equal to 1Mbps, a latency equal to 500ms and a buffer size equal to one BDP (41 packets). The network capacity is thus $w_{max_{ideal}} = BDP + BS = 82$ packets. In Figures 16a and 16b, we report the evolution of the congestion window.

We can observe that FC indeed plays a significant role. It globally worsens the situation for Cubic and LinCubic, but not for AccuCubic. We observe that LinCubic performs very well by precisely tracking the network capacity with or without FC. We have no clear explanation why AccuCubic prefers that FC be turned off.

We further tested the potential benefit of those modifications in the case of 100 flows competing for the bottleneck. We consider various scenarios by varying the RTT from 100 to 500 ms and considering different buffer sizes from $0.1\times$BDP to 1 BDP. For each scenario, we performed 10 runs. We report the number of synchronized flows in the case of 500ms and a buffer size equal to one BDP in Figure 17 for a typical run. Results are consistent with the case of a single flow: LinCubic noticeably decreases the number of synchronized flows as well as AccuCubic when FC is turned on.

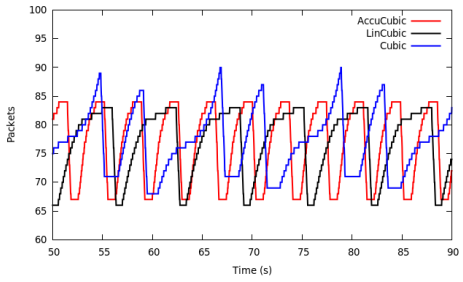
At this stage, we believe that even if the behavior of TCP Cubic can be improved, as exemplified by LinCubic and AccuCubic, the solution to combat synchronization might not be only sought in the TCP implementation itself. Indeed, those improvements might always be partly mitigated by the competition among Cubic flows outlined in Section VI-C. Solutions to the problem of synchronization should thus also be looked for outside TCP itself, e.g., through the use of buffer management mechanisms like RED or Codel [10].

VIII. Conclusion

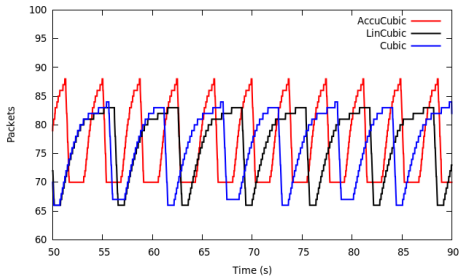
In this work, we have explored in detail the root causes behind the synchronization of TCP Cubic flows that can be easily observed through simulations for instance. We made use of a combination of experiments in a testbed, simulations and some experiments in the wild to analyze the extent of the phenomenon.

The controlled nature of our testbed enabled us to precisely analyze the phenomenon of synchronization and discover its root causes. Simple experiments in the wild (with a distant EC2 datacenter) confirmed that the phenomenon can affect real world transfers.

We discovered that while TCP cubic is known to provide a form of fairness by making the window growth independent of the RTT of the connection (which TCP NewReno is unable to do as the window growth is tightly coupled to the RTT of each connection), synchronization is a subtle result of the interaction

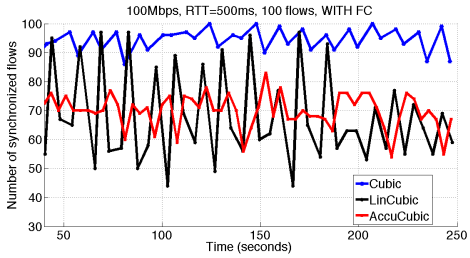


(a) With FC

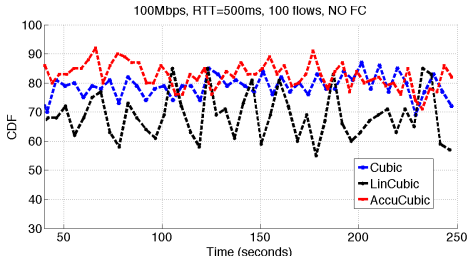


(b) Without FC

Fig. 16: Ns2 Simulations - A single flow



(a) With FC



(b) Without FC

Fig. 17: RTT=500ms

between: (i) the way TCP Cubic reaches the capacity of the network, (ii) the relation between the RTT of the connection and the window growth of the cubic function that occurs at specific time instant. In addition, Fast Convergence, that biases the estimate of the capacity made by TCP Cubic to give a chance to other connections to grab some bandwidth, significantly increases the synchronization phenomenon. Last but not least, even with a perfect estimation of the bottleneck capacity, synchronization can occur starting from an unsynchronized situation where some flows loose while some others do not. Indeed, the sources that did not loose are likely to start probing aggressively (due to the shape of the cubic function) which can result in massive losses for all flows later on. This can be observed especially if the RTT is large. When the RTT is low for all connection, TCP Cubic is quite immune to synchronization.

We proposed and evaluated two modifications to the TCP Cubic algorithm that aim at combating synchronization. They improved noticeably the situation and we intend to explore how they can combined with advanced queuing mechanisms like CoDel, to further reduce synchronization.

We also want to explore data center scenarios with a high dynamics in the number of flows and especially a competition between short and long flows. Due to the noise induced by short flows, long flows are likely to underestimate the network capacity, which, as we have seen, can lead to too many packets sent when reaching the actual capacity, and thus possibly, synchronization.

ACKNOWLEDGMENT

This work was parly supported by AWS in Education Grant award.

REFERENCES

[1] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained tcp retransmissions for datacenter communication," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 303–314, Aug. 2009.

[2] Y.-T. Li, D. Leith, and R. Shorten, "Experimental evaluation of tcp protocols for high-speed networks," *Networking, IEEE/ACM Transactions on*, vol. 15, no. 5, pp. 1109–1122, Oct 2007.

[3] S. Hassayoun and D. Ros, "Loss synchronization and router buffer sizing with high-speed versions of TCP," in *IEEE Infocom Workshops*, 2008, pp. 1–6.

[4] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant." *Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[5] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance," *IEEE/ACM Trans. on Netw.*, vol. 16, no. 6, pp. 1246–1259, 2006.

[6] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound tcp approach for high-speed and long distance networks," in *IEEE Infocom*, 2006.

[7] S. Belhareth, L. Sassatelli, D. Collange, D. L. Pacheco, and G. Urvoy-Keller, "Understanding tcp cubic performance in the cloud: A mean-field approach," in *CLOUDNET*, 2013, pp. 190–194.

[8] S. Hassayoun and D. Ros, "Loss synchronization, router buffer sizing and high-speed tcp versions: Adding red to the mix," in *IEEE LCN*, 2009, pp. 569–576.

[9] J. Lee, J. P. Hespanha, and S. Bohacek, "A study of tcp fairness in high-speed networks," Tech. Rep., 2005.

[10] K. M. Nichols and V. Jacobson, "Controlling queue delay," *Commun. ACM*, vol. 55, no. 7, pp. 42–50, 2012.