Evaluating the Impact of SDN-Induced Frequent Route Changes on TCP Flows

Radu CÂRPA, Marcos DIAS DE ASSUNÇÃO, Olivier GLÜCK, Laurent LEFÈVRE, Jean-Christophe MIGNOT
Inria Avalon - LIP Laboratory - École Normale Supérieure of Lyon, University of Lyon, France
Email: radu.carpa@ens-lyon.fr, marcos.dias.de.assuncao@ens-lyon.fr, olivier.gluck@ens-lyon.fr,
laurent.lefevre@inria.fr, jean-christophe.mignot@ens-lyon.fr

***Abstract*—Traffic engineering technologies such as MPLS have been proposed to adjust the paths of data flows according to network availability. Although the time interval between traffic optimisations is often on the scale of hours or minutes, modern SDN techniques enable reconfiguring the network more frequently. It is argued, however, that changing the paths of TCP flows too often could severely impact their performance by incurring packet loss and reordering. This work analyses and evaluates the impact of frequent route changes on the performance of TCP flows. Experiments carried out on a network testbed show that rerouting a flow can affect its throughput when reassigning it a path either longer or shorter than the original path. Packet reordering has a negligible impact when compared to the increase of RTT. Moreover, constant rerouting influences the performance of the congestion control algorithm. Designed to assess the limits on SDN-induced reconfiguration, a scenario where the traffic is rerouted every 0.1s demonstrates that the throughput can be as low as 35% of that achieved without rerouting.**

I. INTRODUCTION

Although the "best effort" service that the Internet Protocol (IP) provides has been able to handle a wide range of workloads, current infrastructure requires intelligent traffic management to enable sharing resources between delay critical audio flows and best-effort traffic. Traffic engineering techniques are widely used by network operators to provide better Quality of Service (QoS) via fine-grained traffic management. MultiProtocol Label Switching (MPLS) is the classical protocol used to this end.

MPLS-based traffic engineering allows for dynamically configuring how the traffic flows across the network based on available network resources, and for changing the flow routes dynamically to meet QoS requirements. Unfortunately, route changes induce jitter that can perturb overlaying transport protocols. For example, the TCP congestion control algorithm may improperly assume that some data segments are lost and hence reduce its sending rate. Hence, in industrial deployments, operators usually avoid frequent route changes.

Software Defined Network (SDN), on the other hand, enables applications to reconfigure the network more frequently than previous technologies. Large service providers, including Microsoft [1] and Google [2], have deployed SDN in production environments, thus increasing the efficiency of their network. However, the frequency of reconfigurations that these organisations perform is as low as once every 5 minutes [1], and not all data flows are rerouted during these optimisations because doing so could severely hurt the performance.

The question this work aims to address is therefore: *what would be the impact of highly frequent rerouting decisions on network traffic?* To answer this question, we evaluate the impact of frequent route changes on the performance of Cubic TCP flows. We consider backbone networks with high aggregation level employing SDN to improve their energy efficiency. Previous work introduced algorithms to minimise the number of active system components to reduce its overall energy consumption [3], [4]. Flows are aggregated on a subset of links to put network elements in sleep mode, hence maximising the utilisation of active components. Frequent and fast route changes are needed for both reacting quickly to traffic bursts and waking up sleeping devices to avoid congestion. The changes are made either to provide more network capacity or to detect network over-subscription and thus turn off active components to save energy.

By evaluating whether it is efficient to change the routes of TCP flows frequently, the analysis focuses on the impact of packet reordering due to the difference in delay between two routes. The employed methodology may be relevant and applicable to any network operator willing to perform aggressive traffic engineering with frequent route changes.

The rest of the paper is structured as follows. Section II presents the context and background while Section III describes the theoretical analysis. Experimental setup and validations are presented and analysed in Section IV. Section V discusses related work and Section VI concludes the paper.

II. CONTEXT AND BACKGROUND

A. Context

We consider an operator with an energy-efficient network whose resources are underutilised at times, and the data flows are hence rerouted to release links and/or devices that can henceforth be turned them off. When an increase of network traffic is detected, the network devices are switched back on, and the flows are rerouted accordingly to avoid congestion.

The operator employs SDN, a recent trend in network management that advocates for separating the control plane from the data plane. The goal is to move towards more programmable networks, where very sophisticated optimisation can be done online by a centralised controller, which has global knowledge of the network status and can automatically

react to changes by rerouting data flows. For example, after moving to SDN, Google advertises near 100% link utilisation in their network while maintaining a very high quality of service [2].

B. Rerouting and congestion control

The TCP's congestion control algorithm, defined in the RFC 5681, is an important, and one of the most complex, features of modern TCP implementations. This algorithm tries to split the network capacity fairly among all the flows traversing it. Under such algorithm, the sender keeps a congestion window that is dynamically modified depending on the network conditions. The source cannot send more data than what fits in this window during a Round-Trip Time (RTT). The maximum instantaneous throughput, B, of the TCP connection parameters, is thus limited by the size of the congestion window: $B = W \cdot MSS/RTT$, where W is the size of the congestion window in segments, MSS is the maximum segment size in bytes.

TCP was initially designed for standard IP routing assuming that all packets follow the same route towards a destination. Packet reordering and route changes were considered rare. However, in an SDN network, the controller may frequently shift a TCP flow to an alternative path to optimise the overall network throughput. The route change can impact the throughput of a TCP flow in two different ways:

- When the new route has higher RTT, the sender increases the size of the congestion window to maintain the same sending rate. For example, if the RTT doubles, the throughput will be halved and will gradually increase with the growth of the TCP congestion window; a direct application of the equation $B = W \cdot MSS/RTT$.
- The receiver will see packets arriving out of order if the new route has a lower RTT. TCP's congestion control algorithms will assume the worst-case scenario and will see this as an indication of packet loss due to congestion. Figure 1 illustrates the problem considering a sample backbone network. The link *ab* becomes available for transmission and hence packets 2 and 3 take a route shorter than packet 1. The packets will arrive out of order at the receiver who will use duplicate ACKs to notify the sender about a problem. The sender will reduce the size of the sending window to avoid congestion, hence decreasing the transmission rate.

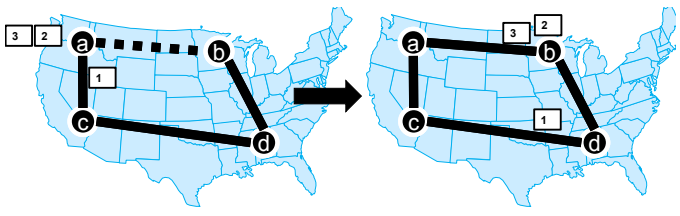

Figure 1. Rerouting towards a lower RTT ⇒ packet reordering.

Given that a large part of network traffic is often carried by a small number of large, long-lived flows [5], rerouting these flows may significantly reduce the overall network throughput. We encountered this problem in practice on a testbed set up to evaluate energy-efficient algorithms [3] [4]. These algorithms, implemented in the SDN controller ONOS [6], were reacting too fast to a reduction in the network throughput caused by a recent rerouting. While the algorithms tried to take advantage of this transitory condition to save energy, they created undesired traffic oscillations.

C. Congestion Control Algorithms

The analysis focuses on the impact of route changes on the default TCP congestion-control mechanism in the Linux kernel. As of writing, the default algorithm used in all the tested devices is Cubic, with a fallback to Reno, including devices using kernel versions 3.2, 3.13, 4.4 and 4.7.

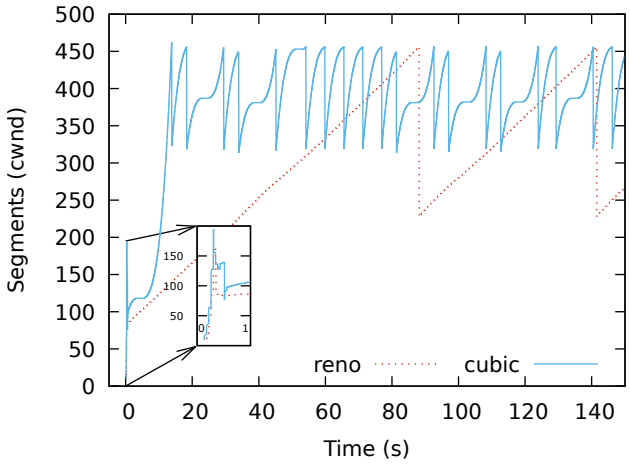

Figure 2. TCP congestion control; RTT = 50ms; 100mbps link.

Figure 2 shows how the congestion window of a single flow evolves when using the Reno and Cubic congestion control algorithms.

1) RENO: The standard TCP congestion control mechanism is Reno, which is still the fallback algorithm in the Linux kernel. This algorithm is straightforward and comprises two phases: 1) the slow start phase, where the window size doubles each RTT; and 2) the congestion control phase where the window size increases by one each RTT or is divided by two under packet loss.

The slow start phase allows for a fast increase in the window size at the beginning of communication. It corresponds to the spike at time 0 in Figure 2. In this work, we ignore the slow start.

Although TCP Reno behaves well under small Bandwidth x Delay Product (BDP), it severely underutilises the channel on long fat networks – *i.e.* networks with high data rate and high delay – because it linearly grows the windows by one every RTT when recovering from packet loss. In Figure 2, Reno takes more than a minute to grow the congestion window and fill a 100Mbps link with an RTT of 50ms.

2) CUBIC: The Cubic TCP algorithm [7] aims to avoid the shortcomings of Reno and achieve high data rates in networks with large BDP. At the same time, when Cubic detects a network with small BDP, it tries to mimic Reno's behaviour emulated by a mathematical model. This happens, particularly in local high-speed, low-delay, networks.

As its name suggests, the algorithm grows the congestion window by using a cubic function of the elapsed time from the last loss event $y = (\Delta t)^3$. More precisely, using a shifted and scaled version $y = 0.4 \cdot (\Delta t - RTT - K)^3 + W_{max}$. In this equation, Δt is the time passed from the last loss event, W_{max} is the size of the congestion window just before the loss event. $K = (W_{max}/2)^{1/3}$ is a parameter that depends on W_{max}.

Cubic is also less aggressive in reducing the congestion window at a loss event. Compared to Reno, which halves the window, Cubic reduces it by 20%.

III. Estimating the Behaviour of TCP Cubic under Route Changes

A. Recovering from a Loss

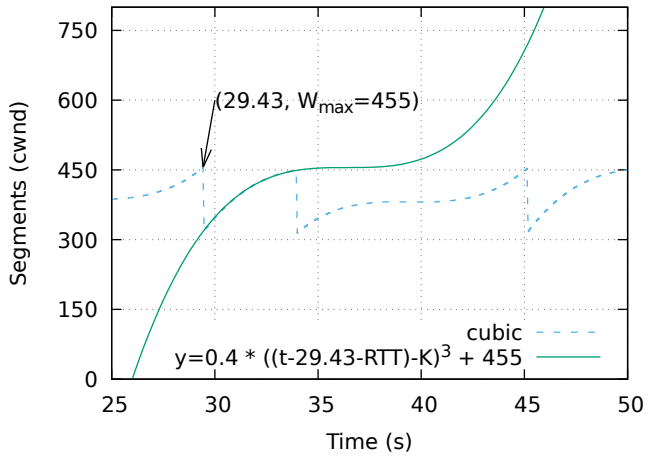

Figure 3. Function used by the Cubic algorithm; RTT = 50ms.

Figure 3 illustrates the cubic function when applied to part of the trace from Figure 2. At time $t = 29.43$, a loss is detected when reaching the bottleneck capacity of a network path. Cubic hence reduces the congestion window to 80% of W_{max}.

The growth function, y, used to increase the congestion window depends mainly on W_{max}, which makes it easy to estimate the time taken by TCP cubic to restore its congestion window after a loss. To do that, we ignore the insignificant dependency on RTT in the Cubic function and solve $W_{max} = y = 0.4 \cdot (\Delta t - K)^3 + W_{max}$. We obtain $\Delta t = K = (W_{max}/2)^{1/3}$.

Moreover, the RTT and the bottleneck bandwidth are used to determine the maximum window size. Table I estimates the size of the congestion window needed to transmit at 100Mbps with 1448 byte segments. It also uses the equation $\Delta t = K$ to assess the time taken by the Cubic algorithm to grow the window back to W_{max} after a loss.

RTT (ms)	10	50	180	500
W_{max} (segment)	87	431	1550	4316.3*
Time to recover (s)	3.5	6	9.2	13*

* the default maximum size of the kernel buffer is too small to allow full speed communications with $RTT = 500ms$. In this case, the transmission speed is limited by the size of the kernel buffer; approximately 2100 segments.

Table I
ESTIMATED W_{max} AND TIME NEEDED TO RECOVER AFTER A LOSS, 100MBPS BOTTLENECK LINK.

This estimation is valid only if no further loss is detected. Figure 3 contains a counter-example where TCP would reach W_{max} at approximately $t = 36s$, but a loss happened at ~34s and forced the congestion control algorithm to reduce the size of the congestion window.

While particularly interested in the time needed to recover from a loss, we note that route changes may generate false loss events when the TCP flow shifts towards a shorter route – shorter here meaning a route with smaller round trip time. At such time, the size of the congestion window is reduced, and cubic tries to grow it back. Intuitively, the throughput of a TCP flow may drop significantly if the SDN controller attempts a second re-optimisation in the meantime.

B. Relevant Linux Implementation Details

The TCP implementation in the Linux kernel incorporates many optimisations [8]. One of the optimisations, which is very important for our work, is the possibility to undo an adjustment to the congestion window. This implementation tries to distinguish between packet reordering and loss by using the "Timestamp" TCP option. When the sender detects that a past loss event was actually a false positive due to packet reordering, the algorithm reverts the window size to the value used before the reduction. As a result, packet reordering may have much less impact compared to standard TCP.

C. Multiple TCP Flows on a Bottleneck Link

As mentioned earlier, TCP cannot send more data than the size of the congestion window per RTT. As a result, when the sender's window W is smaller than W_{max}, TCP spends part of the time waiting for acknowledgements without sending any data.

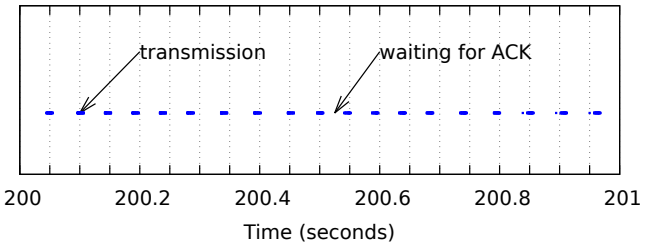

Figure 4. Alternated sending and waiting phases.

Figure 4 illustrates this case, summarising data collected using the *tcpdump* application to capture the packets of a TCP flow over a second. The figure shows the times packets were captured. Approximately 2/3 of the time, there was no packet passing through the network because the congestion window is too small (only 150 segments). Under such conditions, a congestion window of 431 segments was needed to fill the bottleneck link.

If a route change happens when a flow is waiting for acknowledgements, there is no re-ordering problem since no packet is sent. Hence, the probability that a TCP destination will receive packets out of order increases as W approaches W_{max}. Respectively, when TCP backs off and lowers its sending rate by reducing W, it also reduces the probability to be impacted by the rerouting.

If a large number of TCP flows share the same bottleneck link, the flows spend a lot of time waiting for the ACKs. Hence, rerouting a large bunch of TCP flows at the same time may have less impact on the overall network throughput than rerouting a single TCP flow.

This TCP behaviour may change in the future. The benefit of smoothing the transmission over time by employing traffic pacing was shown to have a beneficial effect on network performance [9] and state-of-art congestion control algorithms, like BBR [10], use this technique to avoid the bufferbloat problem.

IV. Experimental Evaluation

A. Experimental Setup

We consider backbone, highly over-provisioned, networks where resource capacity is not a limiting factor. The transmission speed is either bounded by a congested link in the aggregation, or by a very high propagation delay, *i.e.* by the size of the maximum allowed congestion window. No congestion ever occurs on the backbone links.

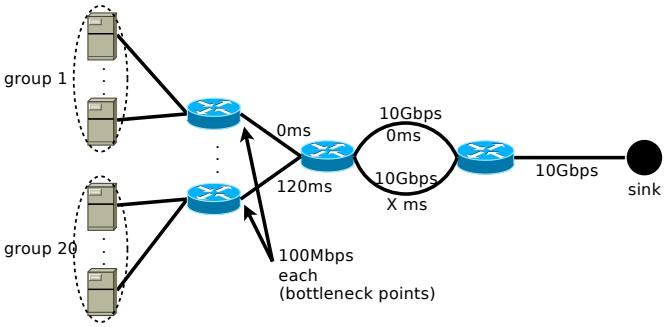

Figure 5. Testbed overview

We construct the topology presented in Figure 5. The examined scenario consists of multiple TCP sources sending data to remote clients. The transfers pass through a backbone network employing traffic engineering: the two parallel 10Gbps links. The traffic is routed through 2 alternative paths within the backbone network, emulated by these 2 links. The paths have different delays. One of the backbone paths has no additional delay (0ms), while the second path induces an additional delay of X ms to simulate a longer route. We use *tc-netem* for this purpose. Each of the two paths (links) has sufficient capacity to transfer the flows. The bottleneck capacity is outside the backbone network. Congestion never happens on these 2 links.

The sources are aggregated in groups. The flows from 2 different groups never share a bottleneck point. However, two flows of the same group compete for the bandwidth allocated to the group. Each group has a different path RTT, uniformly distributed in the interval [0ms, 120ms], which adds to the delay of the backbone paths. The goal is to recreate the scenario where flows with different end-to-end delay coexist in the backbone network.

Concerning the physical infrastructure: nodes in the topology correspond to Ubuntu 14.04 Linux servers running Open vSwitch v2.4 [11] (manually compiled for Ubuntu 14.04). The network is controlled by the ONOS SDN controller (v1.7) via an out-of-band control connection. We wrote an ONOS SDN application to reroute the flows between the 2 backbone links.

The backbone links use 10G optical Ethernet interconnects. The connections to the TCP sources use 1G Ethernet ports. In the physical topology, both 10G and 1G network interconnects pass through Dell Ethernet switches. The point-to-point links are emulated using vlans on these hardware switches. This permits to reconfigure the network topology remotely. Unfortunately, the queuing disciplines on these switches are opaque and cannot be fine tuned. To avoid perturbations, we use *tc-tbf* (Token Bucket Filter) to limit the speed of each group to 100Mbps. In such a way, the data passing through a hardware switch is always much below the link capacity. Moreover, this allows avoiding perturbations due to the Ethernet flow control mechanisms: PAUSE frame. We use a drop-tail bottleneck queue having the size 0.1 * BDP.

B. Rerouting Independent Flows

This section analyses the impact of rerouting multiple independent TCP flows and how the total backbone throughput changes as a consequence of rerouting.

We generate 20 unsynchronised flows, one flow per group, that start at random times within a 30-second interval. We run the transfer for 400 seconds to allow TCP flows to converge to a steady state. Afterwards, we reroute the traffic every 200 seconds and measure how the aggregated throughput changes at during rerouting.

Figure 6 shows how the aggregated throughput varies when the second backbone path induces an additional delay of 10ms, 30ms, 50ms, 70ms or 90ms.

At $t = 600$, the flows are rerouted towards the path with lower delay. Respectively, at $t = 400$ and $t = 800$, the flows are rerouted towards the longer path. This experiment confirms our expectations, that *rerouting the flows impacts their throughput both when moving towards a longer route and when moving towards a shorter one.*

Nevertheless, packet reordering, which happens at $t = 600$, has a less pronounced impact than expected. Theoretically, Cubic would reduce the transmission window to 80% of its

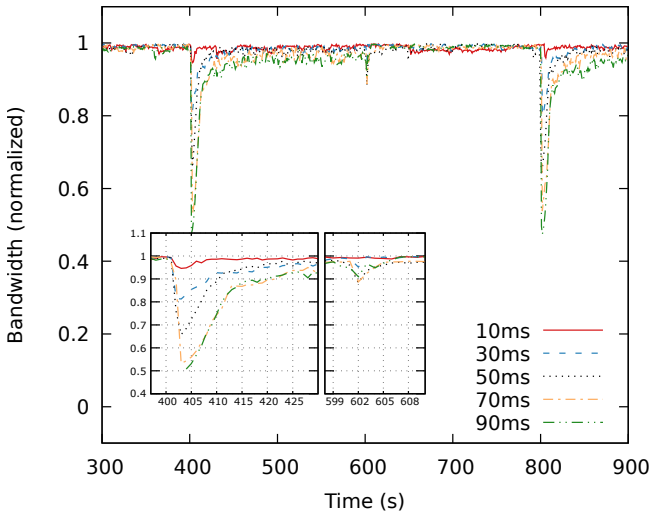

Figure 6. Throughput drop when rerouting 20 independent cubic flows

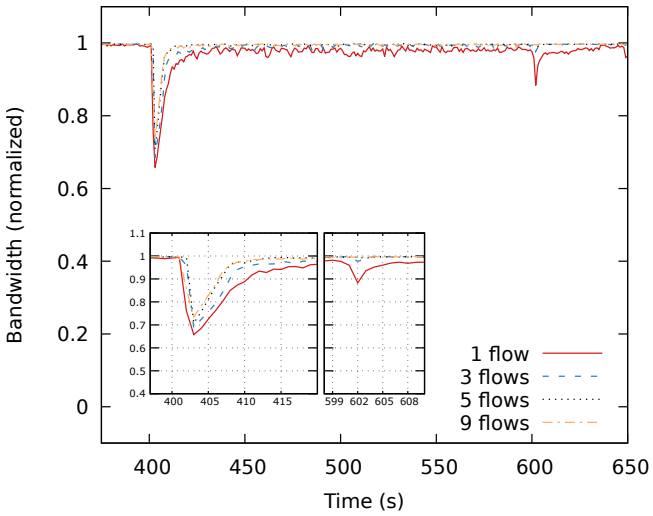

Figure 7. Impact of the number of flows in each of 20 groups.

size when a loss is detected. Then it would gradually increase the window back to the size before the loss. However, the Linux TCP can detect packet reordering in some cases. In particular, this happens when the difference of delay between two routes is small compared to the delay of the fastest route. Hence, a small variation of delay, like 10ms, has a marginal impact on the total throughput of the flows. The next section gives more insights on this optimisation.

Rerouting towards the link with higher RTT substantially reduces the aggregate throughput of the 20 flows ($t = 400$ and $t = 800$). The larger the delay difference between the two routes, the bigger the drop. In this case, rerouting is transparent to the congestion control algorithm. The drop comes from the unexpected increase of the delay. As a consequence, the size of the transmission window must be increased to allow transmission at the same speed.

C. Rerouting Flows Sharing a Bottleneck Point

To evaluate the impact of sharing a bottleneck point where flows inside a group compete for bandwidth, we fix the RTT of the second path to $50ms$ and vary the number of flows per group. With one flow per group, 20 flows traverse the network whereas, under 9 flows per group, there is a total of 180 flows.

Figure 7 shows how competing flows impact the recovering speed after rerouting. The throughput drop caused by rerouting towards a shorter route becomes quickly insignificant ($t = 600s$) in contrast to rerouting towards a longer path ($t = 400s$). With the increase in the number of flows sharing a bottleneck link, each of these flows gets a smaller proportion of the total bottleneck bandwidth. The flows hence spend more time waiting for acknowledgements than actually sending data. At rerouting, fewer flows see their packets arrive out-of-order. Moreover, the flows that do not experience out-of-order arrivals can increase their sending rate by using the capacity released by flows that have just reduced the sending window. As a consequence, the total aggregated throughput on the backbone links is almost not impacted.

When rerouting flows towards a longer path, the aggregate throughput recovered slightly quicker when the number of competing flows per group increases. It is due to non-linear growth function used by Cubic. Having more flows increases the probability that some of them will be in the "aggressive growth" phase of the cubic function.

D. One TCP Flow under Frequent Rerouting

The previous sections showed that packet reordering has negligible impact compared to the increase of RTT. In this section, we assess whether very frequent route changes may reorder enough packets to perturb the congestion control algorithm even if the delay difference between the two considered routes is small compared to the RTT.

While focusing on analysing a single TCP flow, we use *iperf* to transfer 2Gbytes of data and measure its mean throughput (on testbed from Figure 5). A big enough transfer size was chosen to reduce the impact of the slow start phase on the mean throughput. Considering a maximum bottleneck throughput of 100Mbps, each transfer takes approximately 3 minutes if transferred at full speed. The flow is frequently rerouted between the two paths. We tested with periods of rerouting going from once every 15 seconds, to as low as once every 0.1s.

Figure 8 summarises the results. Considering five experiment runs, each data point in the graph corresponds to mean throughput, *i.e.* the amount of transferred data divided by the total transmission time. The error intervals show the absolute minimum and maximum values recorded. The baseline without rerouting shows the average throughput of a TCP flow when it always passes through the same path. The baseline is the mean of 20 experiments. The inner plot zooms in on the region with frequent route changes.

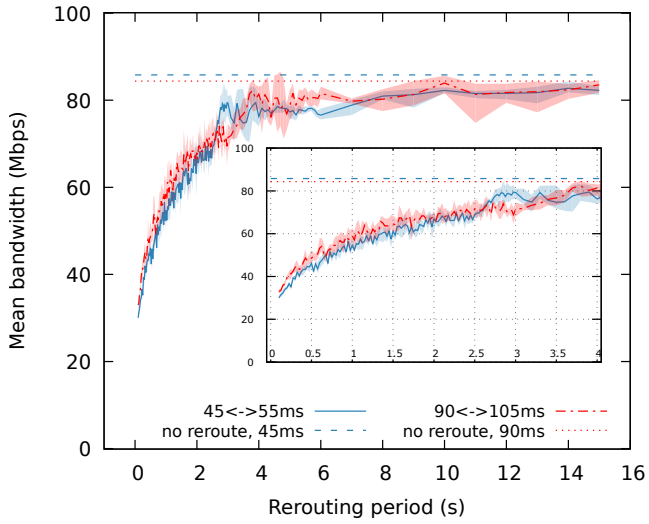

Figure 8. Throughput of a TCP flow. Periodic rerouting between 2 routes with different RTT; Low RTT.

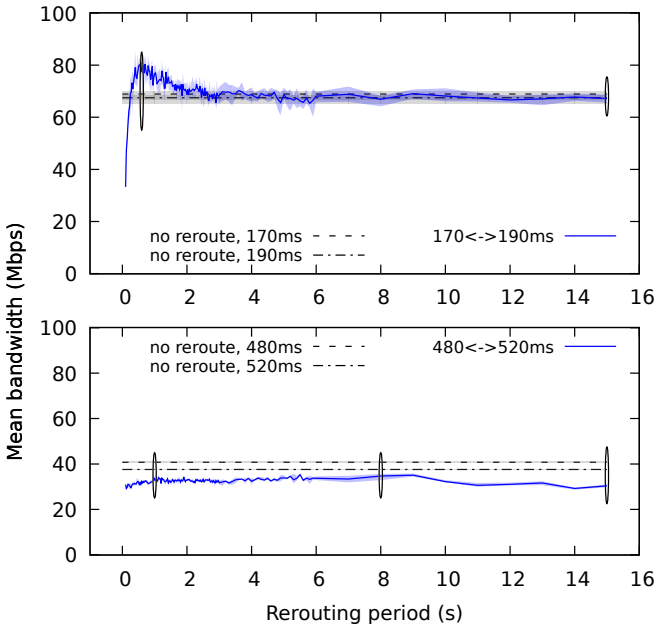

Figure 9. Throughput of a TCP flow. Periodic rerouting between 2 routes with different RTT; High RTT.

Frequent route changes have a large impact on the performance of the congestion control algorithm. When the traffic is rerouted once every 0.1s, the throughput is as low as 35% of the througtput without rerouting. We can also observe that with a route change every 2 seconds, the throughput is around 60% compared to the throughtput without rerouting. Although, from a practical point of view, we believe that these frequencies of rerouting are extreme (and not very realistic), they were included here to evaluate the limits on traffic rerouting. To re-optimise the network at such frequencies, a lot of control messages must be transmitted between the SDN controller and the switches, creating a flood of control traffic. Moreover, the network optimisation problems are usually computationally intensive and take the time to find a good solution.

It is worth noting that the biggest drop in throughput is observed when a second rerouting happens before Cubic recovers from the first rerouting. Previously, we gave an estimation of the recovery time in Table I.

Figure 9 shows two interesting results for higher RTT:

1) Frequent rerouting has a beneficial effect under an RTT of 170, observed at $period = 0.6s$ (the first ellipse) where the mean throughput of a flow rerouted every 0.6s is higher than the average throughput of a flow that is always transmitted over the same path.
2) Under RTT=480, the throughput of the flows is also less impacted when rerouted frequently.

These results are consistent among the multiple transfers. The error intervals are very tight and difficult to see in the figure. To explain this behaviour, we analyse in detail the evolution of the congestion window and packets traversing the network. We choose to concentrate on the points marked with ellipses in Figure 9.

a) Inspecting the beneficial effect at RTT=170: Figure 10 gives more insights on this case. The dashed line shows the evolution of the congestion window without rerouting. At times $t = 23$, $t = 43$, etc, the congestion control algorithm does not behave as expected. Instead of reducing the congestion window by 20% as dictated by the Cubic algorithm, the drop is much higher. Hence Cubic claims bandwidth too aggressively and huge number of packets are lost when reaching the bottleneck capacity.

Moreover, Linux TCP does not employ a *multiplicative decrease* technique of reducing the congestion window. Instead, it reduces it additively at every second duplicate ACK received. These two causes together imply a bad performance at RTT=170.

Rerouting the flow in the meantime perturbs the Cubic protocol and avoids this unwanted behaviour. It is due to the optimisation in the Linux kernel which detects out-of-order deliveries thus reducing the impact of the rerouting. The evolution of the congestion window in the case when the route changes every 0.6s can be seen on the continuous, blue, line of Figure 10. For example, at $t = 8$, a duplicate ACK is detected as being due to reordering and not to a loss, the congestion window is restored to the size before the reduction.

We mentioned in an earlier section that the TCP sender might "miss" a rerouting because the sender alternates between sending packets and waiting for ACKs. If the route change happens when the sender waits for ACKs, no packets will arrive out-of-order. The blue line illustrates this case. At small window size, between $t = 0$ and $t = 30$, the sender spends a lot of time waiting. As a result, the probability to be impacted by a rerouting is small. The bigger the congestion window, the higher the probability to be impacted by a rerouting. Starting with $t = 120$, an equilibrium is created. The sender struggles to grow the window any further: any rerouting has a high

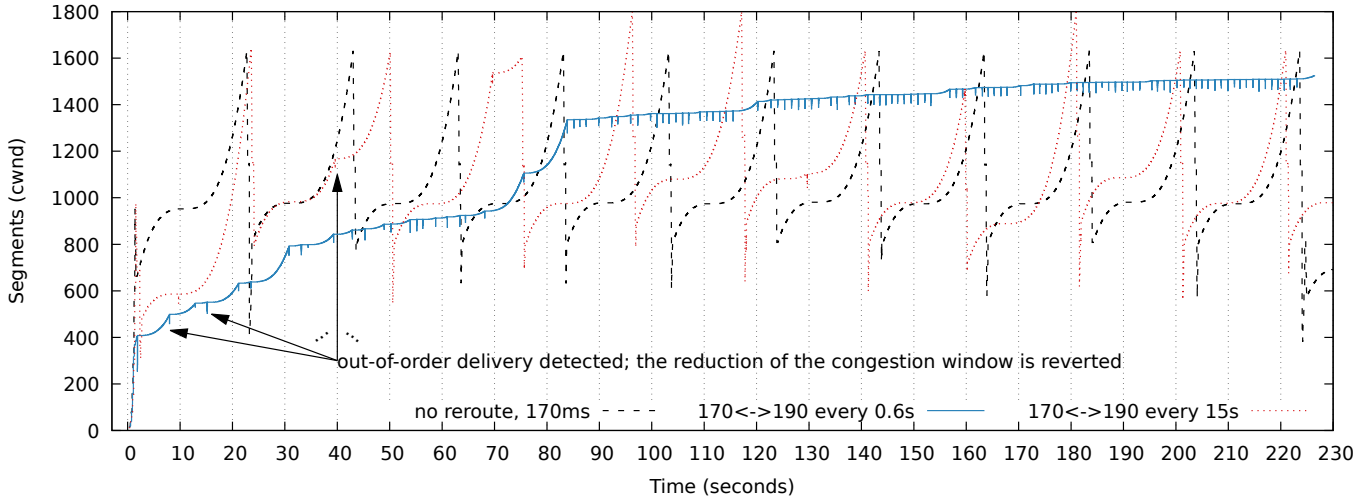

Figure 10. Evolution of the congestion window at $RTT = 170ms$.

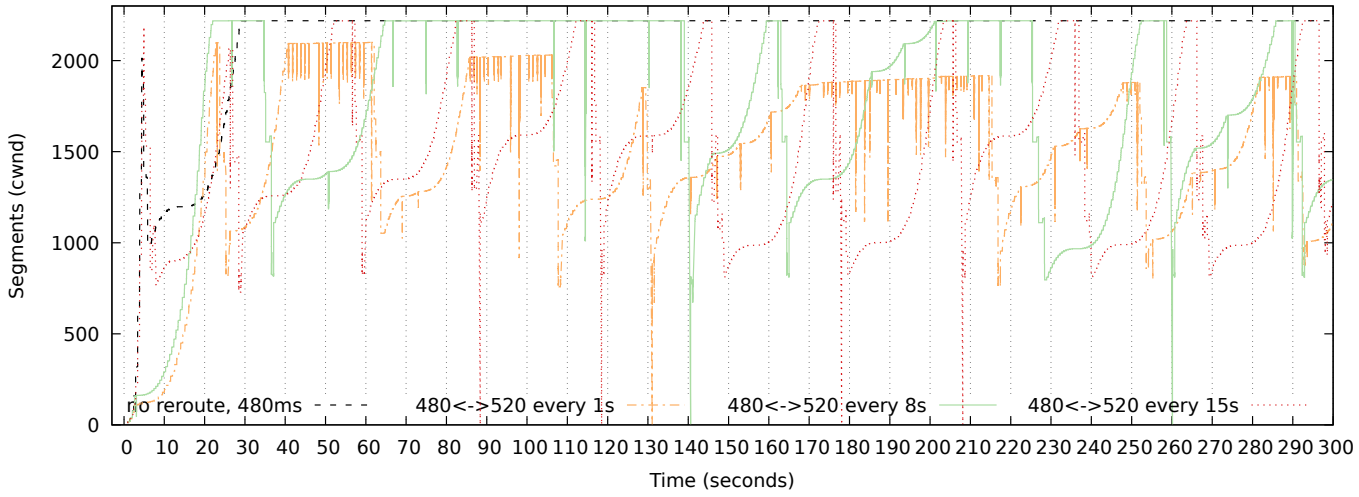

Figure 11. Evolution of the congestion window at $RTT = 480ms$.

probability of reordering packets.

b) Inspecting the unexpected behaviour at RTT=480: Figure 11, which analyses the evolution of the congestion windows, shows that the window remains constant if the routes do not change. This behaviour is expected as the transmission is limited by the size of the congestion window, which in turn is constrained by the size of the in-kernel buffer reserved for the TCP connection.

If we compare the evolution of congestion window in case of rerouting, we observe that the Linux Kernel's optimisation, which detects out-of-order packets, is more susceptible to work as expected at frequent route changes. This unexpected behaviour is consistent among multiple transfers.

However, rerouting once every 15 second allows the flow to converge towards the maximum window size. Any bigger rerouting period (20s, 30s, etc) will have less impact on the average throughput. The worst case scenario occurs at a period of 14s.

V. Related Work

Although the impact of packet reordering on the performance of TCP flows has been extensively analysed by the research community, to the best of our knowledge, previous work has not specifically evaluated the impact of frequent flow rerouting. Moreover, most of the existing work employed discrete-event simulation for evaluating the impact of link fluctuations on TCP flows [12], [13]. The present work considers a real life SDN controller and testbed.

Bennett *et al.* [14] were among the first to highlight the frequency of packet reordering and its impact on the throughput of TCP flows. Their results showed that packet reordering has a very strong effect on a network's performance. Laor and Gendel [15] have experimentally measured this impact on a testbed and also concluded that even a small rate of packet reordering could significantly affect the performance of a high

bandwidth network. As a result, a lot of work was performed to increase the tolerance of TCP to packet reordering [16] [17] [18] [19].

Finally, a solution both to detecting unneeded retransmissions caused by packet re-ordering and to undoing the congestion window was proposed and implemented. The problem of packet reordering was less considered until the introduction of Multi-Path forwarding, where packets of a single flow are split among different paths [20] [21]. It is worth noting that different seetings lead to different conclusions, for example general networks vs fat trees. Karlsson *et al.* [20], for example, concluded that multi-path forwarding reduces the throughput of TCP flows and that mitigation techniques implemented at the transport layer in the Linux kernel are not effective in reducing the impact of packet reordering. On the other hand, Dixit *et al.* [21] affirm that multi-path forwarding in fat-tree data-center networks has little impact on the throughput. In any case, modern multi-path forwarding techniques try to act on a flowlet level and avoid reordering [22].

What differentiates our case from the multi-path forwarding in data-center networks is the fact that packet reordering does not arrive on a per-packet base. Rerouting a flow induces bursts of packets arriving out of order. As a result, our work shows that transport level mitigation techniques are effective as long as route changes do not happen too often.

VI. Conclusion

This work analysed the impact of frequent route changes on Cubic TCP flows. Although, to the best of our knowledge, the literature does not confirm that packet reordering caused by route changes has a noticeable impact on the network performance, the networking community usually considers it as an implicit truth.

By attempting to quantify this impact on a real testbed, experiments showed that packet reordering incurred by rerouting towards a route with lower delay has actually marginal impact on the throughput of rerouted flows. This negligible impact is partially due to optimisations introduced by Linux kernel developers. As a result, the sender can distinguish between a packet loss and reordering and recovers efficiently with negligible drops in transmission speed. Only when rerouting happens unrealistically frequently, we observe a degradation of the throughput of individual flows, but even so, the effect decreases as the flow aggregation level increases.

We also showed that shifting the traffic towards a path with longer RTT has, however, a negative influence on the throughput of TCP flows. SDN-based traffic engineering techniques must be constrained to avoid large variations in the end-to-end delay when rerouting the flows.

In any case, the traffic engineering SDN applications must be made aware of the short drop of network throughput following a rerouting and must limit the frequency of network re-optimisations. Otherwise, this throughput drop caused by the reaction of the TCP congestion control mechanisms may be incorrectly interpreted as a need for further optimisation of network flows.

Acknowledgements

This work is supported by the ELCI project, a French FSN ("Fonds pour la Société Numérique") project that associates academic and industrial partners to design and provide software environment for very high performance computing. This work is also supported by the CHIST-ERA STAR [23] project. We also thank Daniel Valentin for implementing the ONOS application used for traffic rerouting.

References

[1] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486012

[2] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486019

[3] R. Carpa *et al.*, "Improving the energy efficiency of software-defined backbone networks," *Photonic Network Communications*, pp. 1–11, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11107-015-0552-9

[4] R. Cârpa *et al.*, "Responsive algorithms for handling load surges and switching links on in green networks," in *2016 IEEE International Conference on Communications (ICC)*, May 2016, pp. 1–7.

[5] L. Qian and B. E. Carpenter, "A flow-based performance analysis of tcp and tcp applications," in *2012 18th IEEE International Conference on Networks (ICON)*, Dec 2012, pp. 41–45.

[6] "Introducing ONOS: A SDN network operating system for service providers," Open Networking Lab ON.Lab, Whitepaper, Nov. 2014. [Online]. Available: http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf

[7] S. Ha *et al.*, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[8] P. Sarolahti and A. Kuznetsov, "Congestion control in linux tcp," in *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 49–62. [Online]. Available: http://dl.acm.org/citation.cfm?id=647056.715932

[9] H. H. Gharakheili *et al.*, "Edge versus host pacing of tcp traffic in small buffer networks," in *2013 IFIP Networking Conference*, May 2013, pp. 1–9.

[10] N. Cardwell *et al.*, "Bbr: Congestion-based congestion control," *ACM Queue*, vol. 14, September-October, pp. 20 – 53, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=3022184

[11] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*, 2015.

[12] U. Ranadive and D. Medhi, "Some observations on the effect of route fluctuation and network link failure on tcp," in *10th International Conference on Computer Communications and Networks*. IEEE, 2001, pp. 460–467.

[13] F. Baccelli and D. Hong, "Interaction of tcp flows as billiards," in *22nd Annual Joint Conference of the IEEE Computer and Communications*, vol. 2. IEEE, 2003, pp. 895–905.

[14] J. C. Bennett *et al.*, "Packet reordering is not pathological network behavior," *IEEE/ACM Transactions on Networking (TON)*, vol. 7, no. 6, pp. 789–798, 1999.

[15] M. Laor and L. Gendel, "The effect of packet reordering in a backbone link on application throughput," *IEEE Network*, vol. 16, no. 5, pp. 28–36, Sep 2002.

[16] K. c. Leung *et al.*, "An overview of packet reordering in transmission control protocol (tcp): Problems, solutions, and challenges," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 4, pp. 522–535, April 2007.

[17] J. Feng *et al.*, "Packet reordering in high-speed networks and its impact on high-speed {TCP} variants," *Computer Communications*, vol. 32, no. 1, pp. 62 – 68, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366408005100

[18] E. Blanton and M. Allman, "On making tcp more robust to packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 1, pp. 20–30, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/510726.510728

[19] R. Ludwig and R. H. Katz, "The eifel algorithm: Making tcp robust against spurious retransmissions," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 1, pp. 30–36, Jan. 2000. [Online]. Available: http://doi.acm.org/10.1145/505688.505692

[20] J. Karlsson *et al.*, "Impact of multi-path routing on tcp performance," in *2012 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, June 2012, pp. 1–3.

[21] A. Dixit *et al.*, "On the impact of packet spraying in data center networks," in *2013 Proceedings IEEE INFOCOM*, April 2013, pp. 2130–2138.

[22] N. Katta *et al.*, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: ACM, 2016, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/2890955.2890968

[23] "chist-era STAR(SwiTching And tRansmission) project," http://www.chistera.eu/projects/star.