OTCP: SDN-Managed Congestion Control for Data Center Networks

Simon Jouet*, Colin Perkins† and Dimitrios Pezaros‡
School of Computing Science, University of Glasgow, Glasgow, G12 8QQ, Scotland
Email: *simon.jouet@glasgow.ac.uk, †csp@csperkins.org, ‡dimitrios.pezaros@glasgow.ac.uk

***Abstract*—TCP suffers from incast collapse in data center networks when used with partition aggregate workloads due to inadequate congestion control parameters. This causes poor application performance by under-utilizing the network, and can be one of the limiting factors in low-latency, high-throughput environments. To resolve this, we present Omniscient TCP (OTCP), a Software Defined Networking (SDN) approach to compute environment-specific congestion control parameters based on centrally available network properties. Through experimental evaluation in Mininet, we show up to 12× and 31× reduction in Flow Completion Time (FCT) at the mean and 95th percentile, an 8× FCT improvement on highly congested networks when combined with DCTCP [1], as well as improved fairness and reduced end-to-end latency.**

I. Introduction

Numerous warehouse-scale datacenters have been deployed in the last decade to support the emerging needs of Internet applications and services, ranging from web search to storage and general large-scale computation clusters such as Hadoop [2]. In order to support existing applications and improve the cost-efficiency of such deployments, datacenters have been built using traditional internet technologies. As such, the time-tested TCP protocol is used as transport for most applications. However, TCP was originally designed to operate over the wide area where the network characteristics are unknown and the latency, throughput, packet loss and flow characteristics are widely different from those in a datacenter environment [3]. This mismatch between the TCP congestion control parameters and the operating environment of a datacenter are responsible for a phenomenon called TCP throughput incast collapse, resulting in under-utilizing the network and delivering extremely poor performance for applications with a partition-aggregate traffic pattern [4], [5].

As datacenters are managed by single authoritative entities and centralized management protocols such as OpenFlow are becoming increasingly popular, the unknowns upon which TCP congestion control parameters are set should be revisited. In contrast to the wide-area Internet, the topology, throughput, latency, and device properties are known or discoverable which can be used instead of the conservative and inadequate default congestion control values. Therefore, in such an environment, it comes to question whether each host should still estimate its own congestion control parameters based on locally-observable network behaviour, or environment-specific parameters should be computed globally using available information and distributed to the hosts. As a consequence of managing the congestion control parameters centrally, typical misbehaviour created by the static pair-wise congestion control configuration, including long latencies, low throughput and unfairness can be alleviated. Previous work on TCP have addressed some of these issues, such as DCTCP designed specifically to reduce overall buffer occupancy of the switches in the fabric causing long packet traversal delays. Even though the performance improvements of DCTCP is significant it still suffers from poor performances under bursts of short-lived flows due to its inadequate congestion control parameters.

In this paper, we present and evaluate Omniscient TCP (OTCP), a Software Defined Networking (SDN) approach to improve the overall network performance and utilization of modern SDN-managed infrastructures using the legacy TCP protocol. OTCP is able to achieve this by centrally collecting the network infrastructure properties and subsequently compute and distribute congestion control parameters suitable for the operating environment. Through OpenFlow [6], the leading realization of SDN, OTCP can be deployed side-by-side with a controller to collect operating network metrics including topology, latency, throughput and switches' buffer sizes. Once those metrics are collected, OTCP calculates suitable TCP retransmission timers, RTOmin and RTOinit, as well as the initial and maximum congestion window size that matches the route Bandwidth Delay Product (BDP) between end-hosts. Finally, OTCP exposes a JSON/REST northbound interface, to which an end-host daemon connects to receive the congestion control parameters when a connection is established to a new host within the infrastructure. Through evaluation on Mininet, we highlight that centralized management of the congestion control parameters is not only possible, but also significantly improves the infrastructure performance. Using retransmission timers that match the network latency instead of being 2-to-3 orders of magnitude longer than the fabric latency, network utilization can be significantly improved by reducing idle transmission times waiting for the timeouts to elapse. Using a congestion window matching the BDP of the network prevents the buffers of the bottleneck switches to overflow immediately after the initial window (IW) of the synchronized flows is sent, reducing network latency and improving fairness amongst competing flows.

The remainder of this paper is structured as follows: Section II describes the typical datacenter network characteristics

978-1-5090-0223-8/16/$31.00

and the misbehavior of TCP over such low-latency, high-throughput environments. In section III, we present the details of OTCP, a SDN-based approach to congestion control parameter management. We evaluate OTCP in Section IV and show that it outperforms TCP with a flow completion time (FCT) improvement by 12× at the mean and 31× at the 95th percentile, better fairness amongst competing flows and a reduction in end-to-end delay. We also show that OTCP can be used in conjunction with DCTCP to achieve an 8× improvement in FCT in a highly congested network. Section V discusses related work, and section VI concludes the paper.

II. TCP in Data Centers

A. DC Network Characteristics

TCP has been designed to operate efficiently over WAN networks, however in DCs the network characteristics are different. The reported values from large scale providers highlight that TCP accounts for 99.9% of DC traffic [1]. Nearly 75% of the traffic stays within a rack, and most of it is kept within the data center [3]. The network can be highly oversubscribed between the different layers of the fabric. At the rack level, each machine has an associated 1 or 10 Gigabit Ethernet (GbE) link to the top of rack (ToR) switch. The aggregation layer (Agg) responsible for interconnecting ToRs together has typically a 5 - 20 oversubscription ratio, while the core of the network is typically oversubscribed by a factor of 80 - 240 [7]. With latencies varying from hundreds of microseconds within the same rack to few milliseconds inside the same DC, a single-value-fits-all for TCP congestion control in DCs is not suitable without being overly aggressive or conservative. The traffic characteristics are different to the Internet, more than 50% of the time a machine has 10 concurrent flows and at least 5% of the time it has over 80 [7]. 99% of the flows are mice flows, small in size and delay-sensitive. The remaining flows are large, representing 90% of the overall bytes transferred and are throughput instead of delay sensitive [1], [7], [3].

DCs are managed by a single authoritative entity, therefore infrastructure-wide characteristics are available. They can be used to better tune the TCP stack for a specific environment and manage the allocation of network resources. Contrary to the Internet, the topology is known or can be discovered, there are a limited number of hardware configurations available, and they have known properties such as buffer occupancy, switching speed and induced delay, dropping mechanisms, and supported active queue management (AQM). In addition, the bandwidth between nodes in the network is known or discoverable, allowing a maximum bound on the congestion window to be set based on the actual BDP of the path.

Over the last few years, SDN has become more and more prominent in DC infrastructures since the OpenFlow protocol defined a framework for network management and configuration in a centralized manner. Through SDN, a centralized controller is able to maintain, manage and control a full and up-to-date view of the networking infrastructure. Using available information, such as the network topology and latency, routing mechanisms, queue length and throughput, globally-informed decisions can be made that would not have been possible in a legacy network where the control plane is fully distributed amongst the forwarding elements [6].

B. TCP Incast Collapse

A known problem in many DC is TCP incast throughput collapse. This is caused by the typical partition-aggregate workload where a single host issues a single query to a large number of workers and waits for the response [8], [9]. The response of the multiple servers is highly synchronized, generating a large burst of traffic from the workers to the aggregator, overwhelming the buffer of the bottleneck switches and causing significant packet losses. Consequently, TCP is unable to recover using fast retransmission and relies on the retransmission timeout (RTO), bound to a minimum value (RTOmin) of 1 second by RFC2988 and no less than 200ms by commodity operating systems. This 2–3 orders magnitude difference between the network Round Trip Time (RTT) and the timeout results in bursty retransmissions phases under-utilizing the network, and resulting in low throughput and long FCT for delay sensitive flows [1].

Traditionally, TCP incast collapse and the resulting low throughput has been tackled by using deep-buffered switches, reducing packet drop during congestion events, and ensuring that the link is always utilized. However, as a side effect, these deeper buffers induce large delays, retransmission synchronization, unpredictable end-to-end RTT, and prevent the congestion control algorithms to react in a timely fashion [10]. By hiding the congestion event, the large buffers prevent the end hosts from adapting their sending rate to recover from congestion, artificially increase the BDP of the network and introduces high and variable latency for soft-realtime flows. Therefore shallow-buffers are necessary to achieve low-latency transmissions and the congestion control parameter settings must be adequately tuned in the end-hosts to reflect the properties of the fabric without the large buffers masking the congestion events and artificially increasing the BDP of the infrastructure.

Previous literature on TCP incast collapse has attributed the misbehaviour of TCP and the extreme burstiness of the transmission under partition-aggregate workloads, to the mismatch of the retransmission timeout and network characteristics [4], [11], [9]. The long retransmission timeouts, create long idle periods between short transmission periods with high throughput and high buffer occupancy and the next transmission event triggered when RTOmin elapses. In this paper, we contribute to this discussion by highlighting that the congestion window control parameters should also be considered while solving TCP incast. At the onset of congestion collapse, RTO is used as a recovery mechanism, however, by also configuring the congestion window to match the BDP of the network, the rapid buffer build-up in the bottleneck switches can be mitigated, consequently preventing numerous packet drops and throughput collapse.

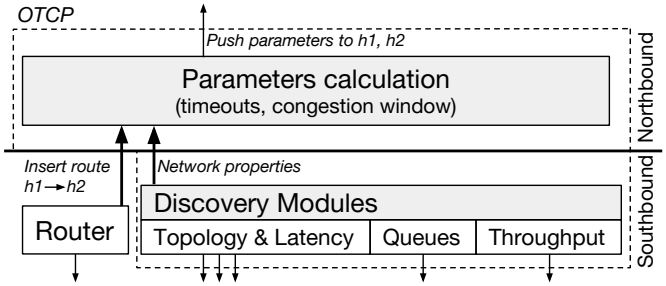


Fig. 1. Overview of OTCP architecture.

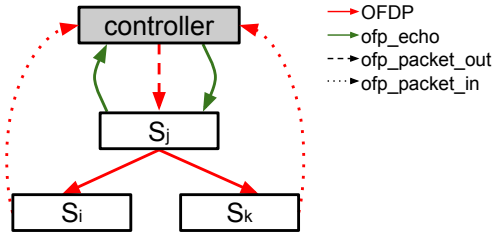


Fig. 2. Latency gathering at the controller using OpenFlow.

The IW, increased to 10 Maximum Segment Size (MSS) in RFC 6928 from 2–4 MSS in RFC 3390 is not suitable for high-throughput, low-latency networks. With an IW of 10 segments, a single flow can send up to 15kB of unacknowledged data after the three-way handshake. In partition-aggregate workloads, hundreds of short-lived flows can be initiated simultaneously sharing the same bottleneck path, resulting in a burst of few megabytes that will quickly overflow the bottleneck switch. The value of 10 MSS might be suitable for WAN environments with long fat pipes carrying few long-lived large flows, however it is unsuitable for a DC environment with numerous concurrent mice flows competing for high throughput over extremely low latency links. IW should be configured to match the BDP of the network as it represents the amount of unacknowledged data on the path between two endpoints. However, because of the large buffers on the switches along the path, the delay can greatly vary from an idle network to a network with all the buffers full. By configuring the congestion control parameters on the idle latency of the network, it would be possible to achieve maximum throughput at the minimum latency achievable by the fabric.

Therefore, high throughput and low latency are achievable in DC network with partition-aggregate workloads but require fine tuning of multiple parameters. The size of switch buffers is not strictly a congestion control parameter, but significantly affects the algorithm's behavior by hiding or delaying the congestion events. Using shallow-buffered switches, the latency can be kept low, however TCP's large default IW quickly overflows the buffers, resulting in low throughput. Adequately tuning the IW to reduce the initial burst of packets and subsequent overflow in the intermediate switches as well as RTOmin to prevent long off periods between transmissions the overall FCT of synchronized flows can be greatly improved.

III. Omniscient TCP

In this section, we present Omniscient TCP (OTCP), a tuning approach for TCP based on information available from the controller in SDN-based DCs. The main goal is to address the impairments of TCP under partition-aggregate workloads, to achieve high throughput, and low and stable latency, with commodity shallow-buffered switches. OTCP achieves this by configuring TCP congestion parameters on a per-route basis based on the end-to-end network characteristics including topology, latency, throughput, and buffering. The congestion window is configured to match the distinct BDP of each route, and the initial burst of traffic can be reduced, limiting the number of packet drops, increasing goodput and fairness for the flows. Using OpenFlow, static network characteristics of large-scale networks can be collected when the controller-to-switch connection is established. As the switching latency and buffer size of switches are static, this approach only requires incremental updates to the measurements on topological alterations and therefore is not necessary to be performed continuously to mitigate TCP incast collapse. An overview of OTCP architecture is shown in figure 1.

A. OpenFlow network information gathering

To calculate suitable congestion control parameter settings for the congestion window and the retransmission timeout, readily available information including latency, buffer sizes and link rate must be collected by the controller.

1) End-to-end latency: To calculate the minimum bound on the RTO as well as the BDP, the idle latency of the switching fabric must be measured. OpenFlow does not provide any direct functionality to measure latency, however the protocol is flexible enough to allow accurate measurements. A common approach for topology discovery in an OF network is to use the OpenFlow Discovery Protocol (OFDP), a variant to the Link Layer Discovery Protocol (LLDP), which allows custom Type-Length-Value (TLV) structures to be inserted in the packet payload. By storing the current timestamp in the TLV payload, OFDP can be used for both topology discovery and latency measurement. The controller generates an OFDP packet, issues a **ofp_packet_out** command to every switch to flood the packet to its neighbours, upon reception the neighbours forward the OFDP packet back to the controller using **ofp_packet_in**. As shown in figure 2 this measurement gives us the 3-hop latency from controller to S_j, S_j to S_i and from S_i back to the controller. This latency is a combination of the management and data network latency and includes the time taken by the control plane to encapsulate and decapsulate the OFDP message which can be significantly longer than data plane latencies as discussed in [12].

The controller to switch latency is measured using OpenFlow **ofpt_echo_request** and **ofpt_echo_reply** usually used by the switch as a keep-alive signal. Using this approach to measure the latency allows the control plane processing time to be included as part of the measurements. Therefore using

this measurement as well as the 3-hop latency measured during topology discovery the latency from S_i to S_j can be measured. Consequently, the latency between two arbitrary switches (S_i and S_k) through the route R is the sum of the latency of every link traversed (equation 1).

$$L_{S_i \to S_k} = \sum_{R_{i \to k}} L_{S_i \to S_{i+1}} \quad (1)$$

These measurements provide accurate latency estimate of the fabric, but do not include the latency between the ToR switches and the hosts. In our implementation we have used an approach that does not require any modification of the end-hosts. The controller generates an ARP Probe packet and sends it from the ToR switch to the host using a **ofp_packet_out** message [13]. The host's ARP response is then intercepted at the ToR and sent to the controller. Hence, the RTT between the switch and the host can be calculated based on the controller-to-switch delay from this 4 hops latency. This approach is especially suitable in environments where the controller is used as a central directory service for the ARPs such as in VL2 [7]. Once the switch-to-switch and host-to-switch latencies are measured, the minimum retransmission timer bound (RTOmin) can be set to the round-trip delay between the two hosts. It is worth noting than the previous discussion assumes that the forward and return paths are the same, hence the forward link latency and the return latency are equal. However if the links are asymmetric the calculation can be easily modified to take each latency independently.

2) Buffer Size: Buffering in the switches impacts the latency of transmission of packets, as a packet can be delayed by the time it takes to transmit all other queued packets. A maximum bound on the latency can be calculated using the maximum buffer occupancy of each switch and the egress link rate. If queues have been assigned to the ports, the OpenFlow controller can send a **ofp_queue_get_config** packet to retrieve the queue characteristics including the length in bytes as well as the minimum and maximum data-rate (since OpenFlow 1.2). Therefore, the maximum bound on the retransmission timer can be characterized by the idle latency (RTOmin) plus the switches' queue delay. For any switch s along the route (R) from H_1 to H_2, the delay is the buffer size (Q) divided by the egress link rate (T) (equation 2).

$$RTO_{max}(H_1 \to H_2) = RTO_{min}(H_1 \to H_2) + \sum_{s \in R} \frac{Q_s}{T_s} \quad (2)$$

The initial retransmission timeout (RTOinit) can be equally set to the same value as RTOmax, however, some additional delay can be caused at the end hosts, for instance if connections have been backlogged or the machine is highly utilized. However by stressing source and destinations CPU cores at 100%, high number of IO operations and numerous memory operations, the maximum increase in latency we have observed in our experiments was 131μs. In TCP, RTOinit is one order of magnitude larger than RTOmin. In OTCP, and in order to account for the possible increase in latency on highly loaded hosts, we set RTOinit to twice RTOmax.

3) Link Rate: To associate a rate to each link of the topology, the controller listens for **ofp_port_status** asynchronous messages containing the port operating mode (10Mb, 100Mb, 1Gb, 10Gb). Knowing the latency (L) between H_1 and H_2, and the maximum sending rate between those two points (T) characterized by the lowest link rate along the route (R), the maximum value of the congestion window ($CWND_{max}$) can be calculated as the BDP (equations 3, 4).

$$BDP_{H_1 \to H_2} = RTT_{H_1 \to H_2} \times T_R \quad (3)$$

$$CWND_{max}(H_1 \to H_2) = BDP_{H_1 \to H_2} \quad (4)$$

By integrating simple measurements to the topology discovery and querying the switches configuration parameters, a controller is able to collect and manage the metrics required to calculate finely-tuned congestion control parameters for the operating environment. As described previously, these measurements are only required when the operating parameters are modified, such as the replacement or addition of links and switches and are not continuously required.

B. OTCP parameter propagation

To distribute the congestion control parameters, the controller exposes a JSON/REST northbound API to which a user-space daemon in the end-hosts connects. When the controller updates the route characteristics, the daemon is notified with the updated values, and the route entry associated is updated. In the Linux kernel, since the kernel version 2.6.23, most TCP congestion control parameters can be configured from user-space for specific destination IP addresses or subnets using netlink to configure the kernel routing table. However, the initial retransmission timeout cannot be configured without a simple kernel modification (16 lines of code in our implementation). These parameters are used by newly established flows instead of the default conservative ones. Such approach requires each end-host to execute a small custom daemon (<50 lines of code) connecting to the controller. However, with the increasing importance of network virtualization and virtual machines, end-hosts are already heavily customized, hosting hypervisors, monitoring software, and other daemons.

On port or switch failure, the controller is notified either with a **ofp_port_status** packet sent by the switches when a port changes state, or of device failure when the TCP connection is torn down. When such event is received, as typical in an OF environment, the new route is computed by the controller and the associated congestion control parameters are also recomputed and sent to the relevant end-hosts. As these events are triggered by link removal it is not required to perform new measurements as the new path will be along links and switches with already known characteristics.

Using the kernel routing table to configure congestion control parameters allows the daemon on the end-host to

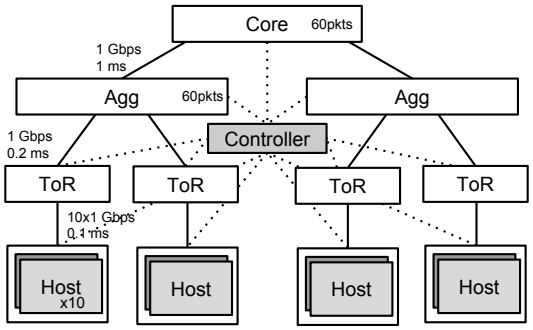


Fig. 3. OTCP Emulation Network Topology

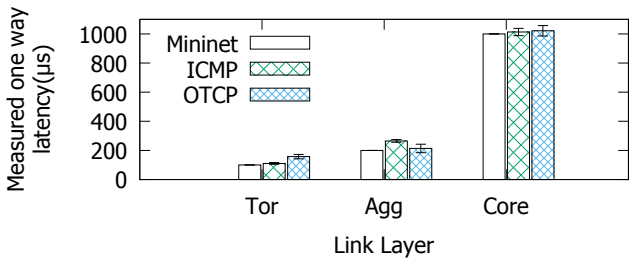


Fig. 4. Switch-to-switch and host-to-switch latency measurement comparing the topological settings, ICMP and OTCP.

be extremely lightweight. However to improve network performance, routing strategies such as Equal-Cost Multi-Path (ECMP) can be used to load-balance the traffic amongst multiple paths. In such situation the destination-based entries in the routing table might not be sufficient, and therefore the daemon should be designed to configure individual sockets when they are created. From a controller aspect the logic is similar, simply requiring parameters to be calculated for the different available routes.

C. Initial Congestion Window

The size of the initial congestion window is dependent on the maximum congestion window as well as the number of active flows in the network during the slow-start phase. Setting the congestion window too high with respect to the BDP will lead to queue build-up on the traversed switches, and consequently to higher latencies and unfairness in flow throughput. Setting the congestion window too low will require more RTTs to reach the bottleneck capacity of the link, lengthening the slow-start phase and therefore increasing FCT. In OTCP, we set *CWNDinit* (IW) to a fraction of the *CWNDmax*, since *CWNDmax* represents the maximum congestion window assuming a single flow along the path. When the path is shared between multiple flows, the maximum congestion window of each flow should be *CWNDmax* divided by the number of active flows (α) in the link (equation 5). In low latency, high-throughput environments *CWNDmax* is small, thus as number of flows α increases the *CWNDinit* will quickly reach its minimum value of 1 MSS.

$$CWND_{init} = min(1, \frac{CWND_{max}}{\alpha}) \tag{5}$$

In the current implementation α can be configured either manually to a value matching the maximum number of flows that will fan-in from the workers or updated at run-time by polling the flow statistics of switches using **ofp_flow_stats_request** similarly to the approach used by Hedera [14]. Using the flow statistics require every TCP flow to have a matching flow table entry in the switches which might be impractical in large networks where edge switches can have fewer table entries available than the number of concurrent TCP flows. Future work will investigate the use of OpenFlow TCP flag matching (OpenFlow 1.5) as well as the upcoming research on data-plane programmability [15], [16] to monitor the number of active flows in the network and tune α accordingly.

These improvements solve TCP Incast collapse by allowing multiple mice flows to highly (throughput) and efficiently (goodput) use the network fabric without suffering from long latencies which are problematic for soft-realtime traffic.

IV. Evaluation

In this section, we demonstrate how OTCP can successfully mitigate TCP incast collapse in DC environment by improving **latency**, **throughput**, **goodput**, and **fairness**. The performance evaluation is performed on a Mininet HiFi 2.2 emulation network for large scale experimentation.

A. Experimental Setup

The experimental topology for OTCP as illustrated in Figure 3[1], has been set up as a typical three-layer tree with an increasing over-subscription ratio and latency at the higher layers. This setup has been used to simulate the bandwidth, latency and oversubscription expected in a production grade DC [7], [1]. To achieve a 10:1 oversubscription at gigabit rate, each rack contains 10 hosts connected with a gigabit link to the ToR and an egress link from ToR to Agg also at 1Gbps. To match with the latencies varying from hundreds of microseconds within the same rack, to few milliseconds for traffic traversing the aggregation and core layers, the latency of the link from host-to-Tor is set to 100μs. This value results in a host-to-host RTT of 0.4ms, matching published values [4]. To achieve millisecond latency at the Agg layer, ToR-to-Agg latency has been set to 0.2ms resulting in a 1.2ms latency and finally Agg-to-Core to 1ms to get cross DC latency of 5.2ms.

To match the shallow buffers of commodity switches, all switches have egress queues of 60 packets regardless of the layer of the topology, with a drop-tail mechanism and without any AQM for early congestion notification or drop. Finally, the Linux kernel for the Mininet host is 3.19.4 with the stock TCP parameters using TCP CUBIC, a 10 MSS IW, a RTOmin of 200ms, and an RTOinit of 1s (decreased from 3s since version 3.2 of the kernel). Both the northbound (REST) and southbound (OpenFlow 1.3) interfaces have been

[1]https://bitbucket.org/sjouet/otcp

TABLE I
OTCP CALCULATED ROUTE PARAMETERS

	RTT (μs)	RTOmin (ms)	RTOmax (ms)	RTOinit (ms)	CWNDmax (MSS)	IW (MSS)
ToR	629	1	2.069	4	49	1
Agg	1485	2	5.805	12	127	2
Core	5571	6	12.771	25	476	5

implemented in a single Go controller designed for the purpose of OTCP managing OpenvSwitch (OvS) software switches in mininet. To prevent the management traffic from impacting the production traffic and vice-versa, the traffic to the controller is on a separate out-of-band network.

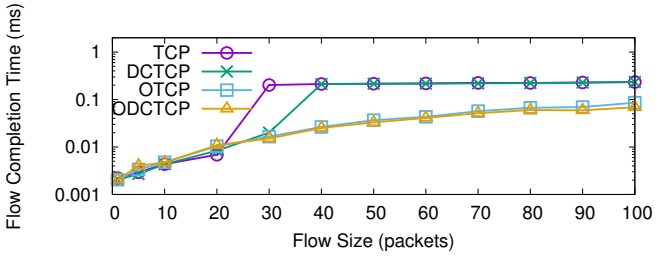
B. OTCP measurements

We first evaluate the ability for the controller to collect the measurements defined in section III-A from the network infrastructure using OpenFlow. These measurements should be conducted when the network is idle, when the buffers are empty, as we are interested in bounding RTOmin to the fabric latency, and setting the congestion window to a value achieving maximum throughput without requiring buffering in the intermediary switches. In Figure 4 we present the mean latency of 100 independent measurements comparing the latency configured in the topology (Mininet, Figure 3), ICMP ping, and OTCP latency measurements. We can see from these measurements that the latency of each link can be measured accurately even in low latency environments with OTCP and ICMP returning similar metrics close to the baseline.

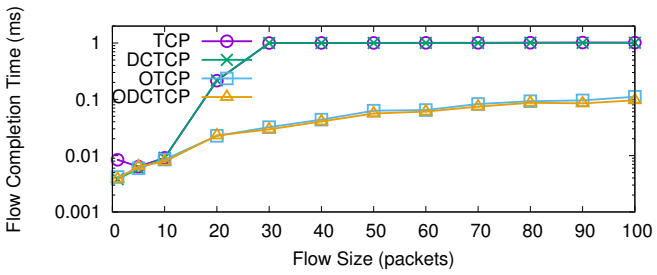
Table I shows the route parameters, grouped by layer, calculated by OTCP based on the latencies shown in Figure 4, and the buffer size as well as throughput of the network. Due to the time granularity of the Linux kernel, RTOmin has a minimum integer value of 1ms and we have therefore rounded the RTT to the highest integer in order to calculate RTOmin. RTOmax represent the maximum delay the network fabric can have considering the egress queue of each switch traversed is fully occupied. In the topology, the maximum queue length (Qs) regardless of the switch layer is 60 packets, in the worst case scenario each packet is of maximum size (MTU) totalling to 90000 bytes. The egress link has a throughput of 1Gbps (Ts) and therefore the maximum delay each buffer traversed is 720μs. As stated in Section III-A2, in OTCP RTOinit is set to twice the RTOmax and must be expressed in milliseconds similarly to RTOmin. CWNDmax is set to match the BDP of the network, therefore it is the RTT times the bottleneck throughput divided by MSS. Dividing the BDP by MSS is required as the Linux kernel expresses the congestion window as a multiple of the MSS. Finally, IW is set to match the number of expected 100 synchronized flows (α).

C. Flow Completion Time

We evaluate TCP, OTCP and DCTCP perfomance under TCP incast with respect to the mean overall FCT to show overall behaviour and at the 95th percentile to highlight unfairness between competing flows. DCTCP is a variant of TCP designed to reduce buffer build-up through Explicit Congestion Notification (ECN). Instead of the common approach of treating ECN markings as a packet loss and react by halving the congestion window, DCTCP uses multiple ECN marks to adapt the sending rate [1]. This approach requires support from both source and destination hosts as well as ECN support in intermediary switches but can reduce buffer utilization by 90% resulting in lower latencies. We also have defined ODCTCP, a combination of OTCP and DCTCP to finely tune the congestion window and retransmission timeouts (OTCP) while preventing queue build-up in the intermediary switches (DCTCP). OTCP uses the parameters shown in Table I.



(a) Mean FCT



(b) 95th percentile FCT

Fig. 5. Mean and 95th percentile Flow Completion Time for mice flows from 1 to 100 MSS sized packets. **Note the logarithmic y axis.**

We compare different variants of TCP with respect to the FCT of flow experiencing incast collapse. A single host is set as the aggregator and the remaining nodes initiate a short-lived burst of traffic to the aggregator simulating a partition-aggregate situation. In this scenario, 1 host is used as the aggregator and the remaining 9 hosts as the workers leading to a 9:1 oversubscription. For each round of the experiment, the worker nodes each initiate 10 flows to the aggregator with a size varying from 1 to 100 MSS-sized packets, totalling to 90 mice flows received at the aggregator.

In Figure 5a we observe that the mean FCT for both TCP and DCTCP is dominated by RTOmin when the flow size is 20 and 40 packets respectively. This is explained by IW being too large, quickly filling the buffers dropping more packet than required for F-RTO to be triggered and therefore relying on RTOmin to retransmit lost packets. DCTCP does not solve the problem of incast collapse on short-lived flows, because DCTCP relies on ECN markings to adapt the sending rate. This approach requires 1 RTT to be effective due to ECN, once the buffer has already been overflowed by each flow sending

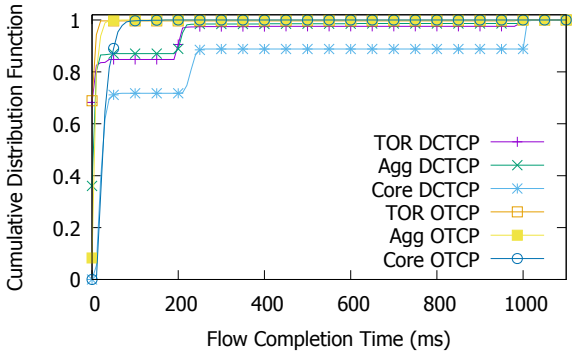


Fig. 6. CDF of Flow Completion Time (FCT) comparing OTCP with default TCP in a three-layer topology

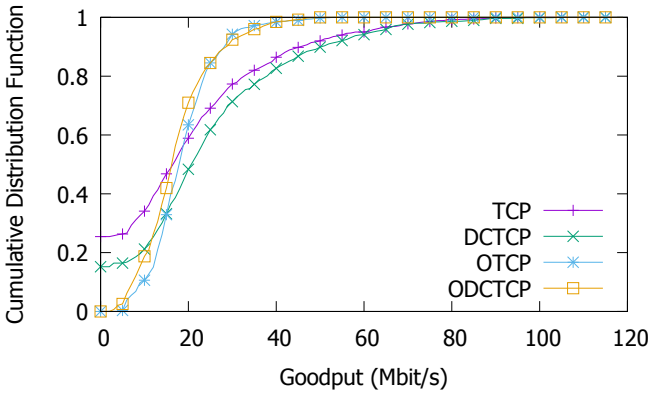


Fig. 7. CDF of flow goodput experiencing incast collapse.

their initial window. Therefore it reacts to congestion only after the congestion event has passed. OTCP and subsequently ODCTCP, using finely tuned parameters avoid overflowing the buffers from the initial burst and recover quickly on packet drops resulting in an improvement of up to 12× in mean FCT.

Figure 5b shows the FCT at the 95th percentile. We can observe a similar pattern as for the mean with TCP and DCTCP performing similarly and the FCT being dominated by RTOinit instead of RTOmin. In this case, due to the large IW, the SYN packet of some flows is dropped by the switch requiring RTOinit to elapse before the packet is resent. These long delays at the 95th percentile result in long tails in the overall FCT. Using OTCP the FCT of the long tail flows can be improved by 31×. Overall, at the mean and 95th percentile, OTCP performs similarly ensuring fairness amongst flows and no long delays caused by a minority of the flows.

Finally in Figure 6, we show the Cumulative Distribution Function (CDF) of the FCT for OTCP and DCTCP across the three layers of the topology. For clarity, we have omitted TCP and ODCTCP from this experiment. In the three scenarios, OTCP outperforms DCTCP. RTOmin and RTOinit for DCTCP are clearly visible on the FCT. At the three layers, we can observe a plateau of 200ms as well as a long tail at 1s generated by the loss of the initial SYN packet. When traffic is confined in the same rack, OTCP is able to complete all of the flows in less than 30ms compared to DCTCP completing 82% of the flow in the same interval, 97% after 250ms and every flow after 1 second. Through the agg layer, using OTCP, all the flows are transmitted in 40ms while DCTCP completes 86% in the same time and 98% after 220ms. Finally, at the core, OTCP completes transmission after 80ms while DCTCP completes 72% in the same interval and 89% after 250ms.

D. Goodput

In order to show that the link is fairly shared amongst the competing flows, we compare the goodput distribution of the different congestion control algorithms. In figure 7 we highlight that the goodput distribution for TCP and DCTCP is unfair with few flows reaching extremely high throughput and most flows performing poorly. Consistent with the FCT evaluation for TCP and DCTCP, a large number of flows achieve a goodput of less than a 1Mbit/s due to long off periods and few lucky flows are able to achieve high goodput. This long tail in goodput represents the few flows that were able to send the full IW of 10 MSS without retransmissions, and therefore completed early their transmission.

O(DC)TCP does not suffer from this unfairness in goodput, with every flow transmitting from 10 to 30 Mbit/s while for TCP and DCTCP it ranges from lower than 1Mbit/s to 80Mbit/s. This fairness improvement is achieved by reducing the IW and therefore better interleaving the packets when the synchronized flows send their IW and resulting in each flow observing a similar level of network congestion. These observations highlight that in OTCP the bandwidth is fairly divided amongst the competing flows, reducing the impact of incast collapse and improving the end-to-end goodput necessary for partition-aggregate workloads to operate properly.

E. Background traffic

The previous experiments have shown that OTCP can prevent the overflow of the traversed buffers by reducing the initial window to a value close to the BDP as well as recover quickly from packet loss using tuned retransmission timers. In OTCP, long-lived background flows can increase their congestion window up to the clamp value CWNDmax (Table I) that matches the BDP of the path, however, when multiple concurrent long-lived flows share the same bottleneck path, the total number of unacknowledged packets can increase above the BDP and therefore create queue build-up. Most flows in DC are mice flows, however, most of the traffic is carried by a minority of background flows that are throughput instead of delay sensitive called elephant flows. These background flows used for data consistency, migration or replication are long lived and cause queue build-up in the switches.

We have further evaluated the FCT of flows experiencing incast collapse when the traffic traverse the core layer of the network and every layer is heavily utilised by background flows. Figure 8 shows the mean FCT of the four different congestion control algorithms. For this evaluation both mice and elephant flows are managed by the same congestion

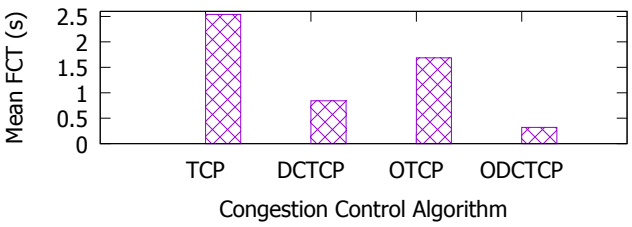


Fig. 8. Mean FCT of flows under incast with an heavily utilized network.

control algorithm. At every layer, the background traffic is generated from multiple hosts resulting in queue build-up increasing end-to-end latency and resulting in larger number of packet drops during the incast event as no buffer space is available. In this scenario TCP is known to perform poorly as it does not prevent queue build-up and the 10 segments from the IW as well as long RTO worsen the problem [10]. DCTCP was specifically designed to address the queue build-up from the background flows but still suffers from large IW and long RTO. OTCP suffers from the queue build-up from the background flows but reduces incast collapse by only sending few packets in the initial window and recovering quickly from dropped packets. Finally, we show that OTCP over DCTCP (ODCTCP) complement each other: OTCP allows finely-tuned parameters to be used adjusting the initial window to match the BDP of the network and the retransmission timers to match the latency of the network while DCTCP prevents queue build-up at the switches improving mean FCT by 8×.

V. Related Work

Significant work has been done on TCP to optimize its performance in specific environments. Variants such as the Rate Control Protocol (RCP) and the Variable-Structure congestion Control Protocol (VCP) estimate link congestion to avoid queue build-up but require custom support on the switch and end-hosts [17]. Fast TCP and XCP have been proposed for environments opposite to DC using long-distance, high-latency links [18], [19]. These protocols have been optimized to achieve high throughput for long-lived flows over long fat pipes (LFP), opposite to what OTCP aims to achieve. TCP NewReno and TCP SACK modify the congestion mechanisms to slightly increase the throughput, however do not prevent TCP Incast collapse. TCP pacing has been proposed to tackle low throughput, however it does not prevent high latencies due to queue build-up when the number of concurrent flows is high, and it is not effective in low-latency networks [20].

In order to recover from incast collapse and prevent the ripple effect of synchronous retransmission after backoff, adding jitter to the backoff timeout has shown an improvement on the throughput, but a degradation on the mean FCT. Facebook had the more radical approach to drop TCP and implement a congestion algorithm on top of UDP with a transmission window size based on the number of concurrent flows in the system, supposedly halving the overall request time of their memcache cluster [4]. Zhang et al. propose to retransmit the last packet of the window multiple time to forcefully generate ACK for F-RTO [9]. Another significant contribution to this problem has been the evaluation of fine-grained retransmission timeouts in a datacenter environment [4]. Allowing the minimum retransmission timeout to match the round trip time of the environment instead of using an overly conservative value as well as providing high resolution timers for the TCP stack shows that the throughput can be significantly improved and long idle times between retransmissions can be prevented.

Fastpass proposes to disable congestion control in the end-host and delegate individual packet scheduling decisions to a controller [21] with a timeslot allocation by a controller for every packet. This approach allows buffer to be kept at low utilization and limit TCP retransmissions, at the cost of increasing the mean time of each packet by the RTT between the host and controller. PCC and Remy use machine learning techniques to discover suitable congestion control parameters for the network [22], [23]. PCC relies on online micro-experiments performed by the end-hosts while Remy generates parameters based on a priori knowledge of the network.

OTCP addresses the problem of Incast Collapse by centrally calculating per-route congestion control parameters and distributing these values to the end-hosts. In addition to the minimum retransmission timeout covered by Vasudevan et al. OTCP calculates the initial and maximum retransmission timeouts as well as congestion windows bounds based on idle network latency, throughput and buffer sizes.

VI. Conclusion

We have implemented Omniscient TCP (OTCP), a centralized controller-based protocol to calculate and manage congestion control parameters for the low-latency, high-throughput environment of DC networks. Using a SDN controller to obtain topological and operational information about the infrastructure, route-specific congestion control parameters can be calculated based on the complete view of the network characteristics. By distributing those parameters to the end-hosts, we show that TCP incast throughput collapse can be mitigated, improving the performance of soft-real-time, partition-aggregate applications.

Our results show that under TCP incast, OTCP significantly outperforms TCP for short-lived, soft-real-time flows, with a 12× improvement in Flow Completion Time at the mean and 31× at the 95th percentile, a low and stable end-to-end latency, as well as higher and fairer goodput. We also show that OTCP can be used jointly with DCTCP to prevent queue build-up and Incast collapse under bursts of traffic, reducing FCT in highly congested networks by 8×. This is showing that fine-tuned congestion control parameters can significantly improve application performance experiencing TCP incast in a DC environment.

Acknowledgments

The work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) projects EP/L026015/1 and EP/L005255/1.

REFERENCES

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM 2010*.

[2] L. A. Barroso and U. Holzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2009.

[3] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *SIGCOMM IMC 2010*.

[4] 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," in *SIGCOMM 2009*.

[5] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *SIGCOMM 2009*.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM 2009*.

[7] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," in *SIGCOMM 2009*.

[8] A. Phanishayee, E. Krevat, V. Vasudevan, D. Andersen, G. Ganger, G. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," in *FAST 2008*.

[9] J. Zhang, F. Ren, L. Tang, and C. Lin, "Taming TCP incast throughput collapse in data center networks," in *ICNP*, 2013, pp. 1–10.

[10] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the Internet," *Queue*, vol. 9, no. 11, Nov. 2011.

[11] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Workshop on Research on Enterprise Networking 2009*.

[12] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for openFlow switch evaluation," in *PAM 2012*.

[13] S. Cheshire, "Ipv4 address conflict detection," 7 2008, RFC 5227.

[14] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *NSDI 2010*.

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM 2013*.

[16] S. Jouet, R. Cziva, and P. Dimitrios P., "Arbitrary packet matching in openflow," in *HPSR 2015*.

[17] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *SIGCOMM*, vol. 36, no. 1, Jan. 2006.

[18] Y. Zhang and T. R. Henderson, "An implementation and experimental study of the explicit control protocol (XCP)." in *INFOCOM 2006*.

[19] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "Fast tcp: Motivation, architecture, algorithms, performance," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006.

[20] M. Podlesny and C. Williamson, "Solving the tcp-incast problem with application-level scheduling," in *MASCOTS 2012*, 2012.

[21] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A Centralized Zero-Queue Datacenter Network," in *ACM SIGCOMM 2014*, Chicago, IL, August 2014.

[22] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance," in *NSDI 15*, May 2015.

[23] K. Winstein and H. Balakrishnan, "TCP Ex Machina: Computer-generated congestion control," in *SIGCOMM '13*.