HWatch: Reducing Latency in Multi-Tenant Data Centers via Cautious Congestion Watch

Ahmed M. Abdelmoniem
CSE Dept., HKUST, Hong Kong
CS Dept., Assiut University, Egypt
amas@cse.ust.hk
ahmedcd@aun.edu.eg

Brahim Bensaou and Hengky Susanto
CSE Department
HKUST
Clear Water Bay Hong Kong
{brahim, hsusanto}@cse.ust.hk

***Abstract*—Modern data centers host a plethora of interactive data-intensive applications. These are known to often generate large numbers of short-lived parallel flows that must complete their transfer quickly to meet the stringent performance requirements of interactive applications. Network resources (e.g., switch buffer space) in the data center are scarce and are easily congested, which unfortunately results in long latency. As a first solution, following a traditional design rationale, Data Center TCP (DCTCP) was proposed to speed up the completion time of short-lived flow by maintaining a low buffer occupancy in the switch. In general, DCTCP performs well in homogeneous environments, however, its performance degrades quickly in heterogeneous environments or when it uses a large initial congestion window. To resolve this problem, we propose a Hypervisor-based congestion watching mechanism (HWatch), which measures the network load in the data center via ECN and uses the resulting statistics to determine the appropriate initial congestion window size to avoid congestion. HWatch neither needs modification to the TCP stack in the VMs nor requires any specialized network hardware features to meet its targets. In our evaluation, we demonstrate the benefits of HWatch in improving the performance of TCP flows through large-scale ns2 simulation and via testbed experiments in a small data center[1].**

***Index Terms*—Congestion Control, DCN, ECN, Hypervisor.**

I. Introduction

In the recent explosive growth of cloud-based and high-performance computing data-driven deployments, applications rely on distributed frameworks, such as Hadoop, [1], or Spark [2] to process massive sets of data. To achieve excellent performance at the application level, a timely data transfer is crucial to building scalable and responsive applications, and a slight delay may lead to significant performance degradation at the application level [3]. There have been considerable efforts to address the performance of delay-sensitive applications in data centers, invoking techniques such as request scheduling, resource allocation and over-provisioning, or traffic prioritization [4], [5], [6], [7], [8], [9]. Nevertheless, these solutions fall short of their goal in the context of large-scale multi-tenant environments (with abundant computing resources) as network congestion is often the primary cause of the application performance degradation in such environments.

[1]This work has been accepted for publication in ACM International Conference on Parallel Processing (ICPP) 2020

Today, TCP is still the most widely used protocol to resolve the congestion problem in the data centers. Our preliminary investigation reveals that, with the presence of short-lived and long-lived flows competing for the limited available resources (e.g., switch buffer), TCP flows may suffer from bloated buffers or incast congestion leading to excessive timeouts. These latter result in significantly longer flow completion times (FCT) for delay-sensitive flows, which are often short-lived. The reason behind this is that in the presence of the small shared buffer, when they experience a packet loss, short-lived flows often do not have a sufficiently large pipeline of in-flight data packets to trigger TCP's duplicate ACKs loss recovery mechanism. This forces them to frequently rely on TCP retransmission timeout to detect packet losses. Such timers are hardcoded in TCP and are typically in the range of 200-300 ms. With a typical delay of less than 1ms in a large data center, such timeout delay would inflate the FCT by several orders of magnitude. DCTCP was introduced to tackle this problem by following the traditional rationale of keeping the buffer occupancy at the switches low, as a means to providing allowance in the buffer to absorb bursty packet arrivals [10], [11], [12], [13]. However, our findings show that despite the availability of the headroom in the buffer, DCTCP still suffers from packet losses due to the use of large initial congestion windows by default in most TCP implementations.

To resolve these shortcomings, we propose our practical and novel non-intrusive system to achieve performance gains while meeting the following design requirements: (R1) Our system must improve the FCTs of delay-sensitive applications; (R2) It should not degrade the performance of long-lived flows dramatically like in preemptive systems; (R3) It must comply with the VM autonomy principle. That is, modifications are only applied to the hypervisors, that are fully controlled by the data center operator, without touching the network stack of the guest VM; and finally, (R4) the solution must be practically easy to deploy without requiring changes to the switching devices nor the NICs at the hosts.

To satisfy these requirements, we address the buffer overflow problem in three phases. In phase one, we introduce an analytical framework to explore the design space for the solution, provide a holistic view of the problem, systematically investigate the complex interaction between network components

(e.g., switch and end hosts), analyze the obtainable information used for decision making, and determine the decision point. Additionally, using this framework, we model the buffer overflow problem as the classic bin packing problem [14], allowing our design to inherit wisdom from earlier studies of the same problem.

In phase two, we describe our theoretical approach to provide guidelines to our system design and implementation with insights gained from our analysis using the framework. Our scheme draws inspiration from the Next Fit algorithm, a widely used solution for the bin packing problem. However, since the buffer overflow problem is a distributed online problem, we design a distributed version of the Next Fit scheme to solve the buffer overflow problem. The key insight to our design is to perform stochastic information mining from ECN and TCP, and then use the information to determine how and how many packets should be transmitted by the senders. By doing so, we ensure that either there is sufficient buffer space along the path to accommodate the imminent incast traffic with a standard initial TCP congestion window, or the looming incast traffic does not start with a full initial TCP congestion window.

In phase three, we present HWatch, a system whose design and implementation are based on our theoretical results while taking into account realistic and practical conditions. The HWatch prototype is implemented in the hypervisor using ECN, a built-in function that is commonly available in today's commodity switches to access the network traffic condition. Here, we also address both the practical and engineering challenges in our implementation and deployment of HWatch prototype in a small data center.

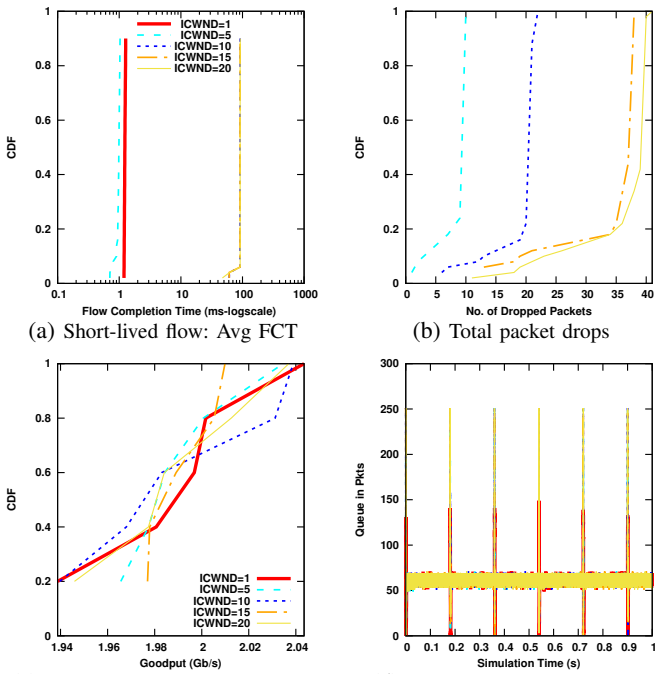
Through our experiments, we demonstrate that HWatch improves the performance up to $10\times$ on average in a large scale simulation and 100% in the testbed experiments.

II. Motivation

A. Preliminary Investigation

DCTCP [10] is one of the most widely accepted congestion control mechanisms for data center networks. It has been included in Linux kernel distributions since version 3.18. DCTCP is an ECN-based congestion controller that maintains a low queue occupancy as a means of improving flow latency. It has been analyzed rigorously in theory [15] and in practice [10], [16]. Although DCTCP achieves good performance gains, compared to TCP NewReno, it fails in many cases, and in particular those involving large initial congestion windows and those where it coexists with other TCP variants.

To demonstrate and highlight such cases, we conduct several ns2 simulations of DCTCP in a 10Gbps dumbbell network with an RTT of 100μs and a bottleneck buffer size of 250 packets (which is quite large compared to the 30-35 packets-per-port encountered in shallow buffered commodity switches). We first run an experiment of a typical scenario where a few background flows regulated by DCTCP face at certain epochs a sudden surge of short-lived (small) traffic flows. To analyze the effects of the initial sending window, we use various values of the initial windows and report the FCT of incast flows.



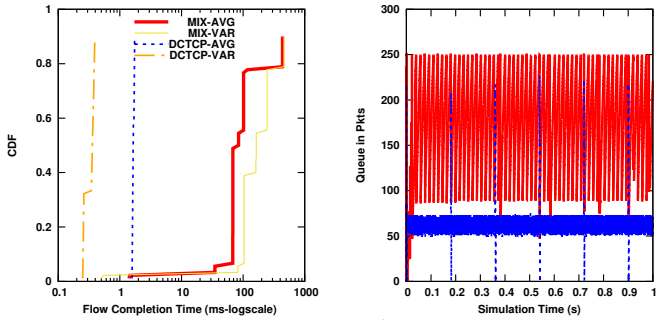
(a) Short-lived flow: Avg FCT (b) Total packet drops

(c) Long-lived flow: Avg Goodput (d) Queue occupancy over time

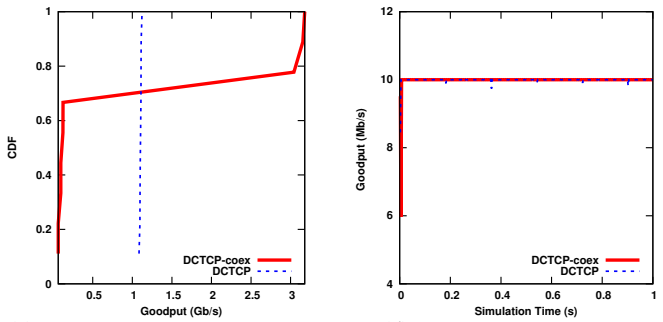
Figure 1: Performance of DCTCP with various initial congestion window values in the range of [1,20].

Figure 1 shows the average FCT for small flows, the goodput for long-lived (large) flows, the total packet drops experienced, and the persistent queue length. The average FCT for small flows and the number of drops for all flows shown in Figures 1a and 1b, respectively, indicate that short-lived flows are quite sensitive to the choice of the value of initial sending window (ICWND in the figures). The FCT increases by two orders of magnitude with a window increase from the initial window of 1-5 to 10 and above (noting that 10 MSS is the preset default value in Linux distributions). The average goodput and queue size, depicted in Figures 1d and 2a respectively, demonstrate that DCTCP is good enough for large flows to maintain a low queue occupancy except at the epoch when small flows surge.

DCTCP employs a non-conventional way of handling the ECN marking, where the congestion window is reduced proportionally to the number of marks observed, unlike regular TCP NewReno or Cubic TCP, which respond by cutting the window by half once per RTT. The proportional DCTCP marking results in the more aggressive acquisition of the available bandwidth compared to regular TCP. Hence, the coexistence of such TCP flows with various responses to ECN signaling in the same data center would result in unfairness [16], [17]. These phenomena are prevalent in current shared data centers (or clusters) with thousands of tenants, each employing its preferred version of TCP congestion control. To illustrate this, we rerun a similar simulation experiment but now with DCTCP coexisting with other TCP variants (two TCP NewReno flavors, a responsive one, and a non-responsive one to ECN marks). Figure 2 shows the average and variance of FCT for small flows, the goodput for large flows, the total number of packet drops, and the persistent queue length.



(a) Short-lived flows: Avg FCT (b) Queue occupancy over time



(c) Long-lived flows: Avg Goodput (d) Link utilization over time

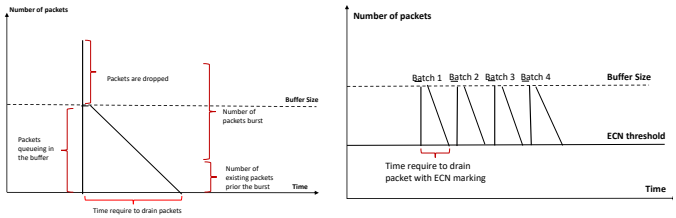
Figure 2: Performance of DCTCP w/wo the existence of other congestion control flavours within the same network.

Figure 2a indicates that when different congestion controllers are used, unfairness is observed in the performance. The FCT varies within a range of two orders of magnitude, taking values from a few milliseconds to a few hundred milliseconds. The queue size, shown in Figure 2b, implies that DCTCP is no longer able to regulate the queue or maintain it at the pre-set target threshold, which explains the unfairness and poor performance caused for the majority of both small and long-lived flows. Figure 2d shows that in both cases, the link is fully utilized, yet the performance of some flows in the co-existence case is significantly worse than the other case where DCTCP operates alone in the network.

B. The root of the problem

Based upon our preliminary investigation discussed above, we make the following observations:

- ***Observation 1***: Senders do not receive three DUPACKs to trigger TCP fast re-transmission scheme when the tail-end packet(s) of the corresponding flow are dropped. Thus, the sender must rely on TCP Retransmission Timeout (RTO) to detect packet drops;
- ***Observation 2***: During incast traffic surges, bursts of packets are dropped because a large subset of the incast group of flows experiences buffer overflow simultaneously due to the large initial sending window of TCP. In this case, the sender may lose the entire window or a large portion of it. Similar to the previous case, the sender must rely on RTO to detect packet loss especially if the sender loses a window-full of packets, leading to a larger FCT;
- ***Observation 3***: This buffer overflow problem primarily affects small flows as the added TCP timeout (hundreds to thousands of RTTs) is in practice orders of magnitude larger than the flow's nominal lifetime (a few RTTs).



(a) Burst packets (b) Batches of packet burst

Figure 3: Observation in the switch: Packet bursts and batching

In a nutshell, during bursty packet arrivals at the buffer, small flows suffer more from buffer overflow as they would most likely not have enough in-flight packets to perform TCP fast recovery, and must rely on RTO. The minRTO in most TCP implementations is set to around 200ms, which increases the latency by 2 to 4 orders of magnitude compared to the usual RTT in data center networks. The system performance degradation in the data center becomes more visible when the network traffic is dominated by small flows (e.g., 80% to 95%) [10], [18], [19], [6]. In other words, small flows are highly affected by packet dropped. The degradation is further amplified when one of the flows experiences delay, and the application relies on a parallel set of short-lived flows to finish together for the job to complete [19], [20], [21]. This delay leads to longer job completion times despite other flows from the set complete timely, which results in performance degradation at the application level.

To resolve these problems, we propose a solution that consists of the following properties:

1) To avoid having an insufficient number of packets to trigger three DUPACKs, the source sending rates are adjusted whenever there is a feedback from the network that congestion (i.e., a pre-configured buffer threshold is exceeded); and
2) To avoid a sender losing a window-full of packets or a large proportion of it, the sources' start-up speeds should be determined according to the current approximated congestion status in the network.

III. Analytical Framework

In this section, we explore analytically the design space for the solution. Specifically, we first provide a holistic perspective of the problem then systematically evaluate the complex interactions between network components (switch, sender, and receiver), then analyze the available information used for decision making by mining information from ECN and TCP.

A. Switch Buffer

To understand the packet drop phenomenon in DCTCP, we first investigate how packets are managed in the switch during an occurrence of incast or packets burst.

Generally, with respect to ECN-based congestion control schemes, packets can be grouped into the following categories:

1) Packets with ECN marking. 2) Packets without ECN marking. 3) Packets dropped due to buffer overflow.

Let us consider the scenario of a switch buffer of size B packets, where X packets for $X > B$ arrive at approximately the same time t. Denote by $Q(t)$ the queue length at time t. Then we have, the first $B - Q(t)$ packets will be queued in the buffer, whereas the next $X - B + Q(t)$ packets are dropped due to buffer overflow. Notice that as time proceeds, more packets are processed, and there is more space made available in the buffer, as illustrated in Figure 3a. In other words, buffer can absorb more packets as space becomes available. The time required to drain the entire packets that are currently queued in the buffer (including $Q(t)$) is $\frac{B}{C}$, where C denotes the packet processing capacity at the switch. Thus, the switch can absorb another B packets after $\frac{B}{C}$ time unit.

To mitigate buffer overflow, X packets can be broken into smaller batches of packets and each batch is transmitted at the time whenever buffer space becomes available. For the sake of clarity, we use a simple example as an illustration: assume there are only X packets traversing to a switch with size B. Then, one way to mitigate buffer overflow is to break X into $\lceil \frac{X-(B-Q(t))}{B} \rceil + 1$ batches and each batch arrives at the switch after the earlier batch is drained. By using ECN and other traffic statistics collected at the sender and receiver, we can estimate the level of congestion and distribute the excess traffic over several batches to avoid packet losses as depicted in Figure 3b.

We further observe resemblance between the concept of distributing packets into multiple batches to be transmitted at different times to avoid buffer overflow with the classic bin-packing problem. It is a problem of packing a set of items into a finite number of containers (bins) with a fixed volume to minimize the number of containers used. The concept of items and containers in the bin-packing problem can be visualized as the problem of packing packets into buffer at different time. For example, the t^{th} container can be visualized as the buffer space at time t. Additionally, the objective of the bin-packing problem can be perceived as minimizing the number of batches required to mitigate buffer overflow. Thus, the key intuition behind the modeling of buffer overflow problem is to visualize the buffer in a timeline allowing the scheme to treat the buffer state (e.g., buffer occupancy) at different times as containers in the bin packing problem. By doing so, our scheme draws inspiration from one of the classic solutions for bin-packing problems, such as the Next Fit algorithm.

The Next Fit algorithm evaluates whether the current item (or packet) fits the current bin (buffer stage). If so, then the algorithm places the item in the current bin. Otherwise, the algorithm puts the item in a new bin (i.e., send the packet later). One of its desired traits is that it can operate with incomplete information, which is vital in solving an online distributed problem like congestion control. Additionally, it has a linear running time, which is suitable for delay-sensitive applications. Essentially, the algorithm only concerns itself with the current packet and buffer stage, allowing for shorter FCTs.

B. Receiver

A receiver has a naturally appropriate position to evaluate information carried by ECN packets from inbound traffic. For example, by checking the Differentiated Services Code Point (DSCP) field in the TCP header, the receiver can count the number of packets with and without ECN marking. Other information available at the receiver is the inter-arrival time between two packets and the time required to receive a specific number of packets (e.g., 10 packets). By analyzing this information, the receiver can determine its sender's maximum throughput.

C. Sender

There is a wealth of information available at the sender. The sender is aware of the number of transmitted packets, the inter-departure time, congestion window size, RTT, transmission time of a set of packets, number of re-transmitted packets, and so on.

In addition, the sender can also assess the flows' contribution to the current and future congestion in the network. This assessment is achieved by counting the number of packets that have been and are ready to be transmitted. For these reasons, the sender makes a good candidate for decision point allocation. Moreover, this allocation also ensures the compatibility of our scheme with the existing commodity switches, which makes our scheme more practical and deployment friendly.

The combination of information gathered from both the sender and receiver provides a sender with a richer and more holistic view of the network condition. For instance, the number of packets dropped can be approximated by subtracting the number of packets received by the receiver from the total number of transmitted packets by the sender. Another example is the congestion severity can be estimated by comparing the time required by the sender to transmit the entire packets in the congestion window and the time taken to receive these packets at the receiver's end.

In our implementation, the receiver conveys information to its sender by inserting information observed locally into several fields in TCP header of the ACK packet. For example, TCP receiver window (*Rwnd*) is utilized for communicating the number of packets received without ECN marking. And, the 16-bits Urgent Pointer field (UPF) could potentially be used to convey the number of packets received at the receiver within a specific interval. The UPF is usually unused and URG flag for UPF is set to zero in ACK packets.

D. Further Observation

To consider more realistic conditions, we assume sender and receiver have no information on switch buffer size and capacity. Moreover, we also do not assume that switches at different layer of network topology used in data center (e.g. FatTree or Clos [22]) have uniform processing capacity.

Let C denote a switch processing capacity. When two packets, i and $i + 1$, of the same flow queue in an empty buffer, the time interval between these two packets leaving the switch is determined by the time required to process packet

$i+1$, which is $\frac{1}{C}$. In contrast, when the network is congested, there is a higher probability that there are other packets from different flows queued between these two packets. If so, then the time interval increases. This is because packet $i+1$ can only be processed after all packets ahead of it in the queue (including packet i) are processed. For these reasons, packets of the same flow can become distributed in the queue. Moreover, same situation arises when there are other packets from different flows queueing between the packets of a certain flow. For instance, there are three packets of a flow queueing in a same buffer, packet i, $i+1$ and $i+2$. When incast occurs, packet i is at the head, while packet $i+1$ and $i+2$ are in the middle and the back of the queue respectively. Therefore, the congestion severity in the network can be stochastically approximated by analyzing the time interval of packets of a certain flow from the sender and the time interval of the same packets arriving at the receiver. Intuitively, if the inter-arrival times are longer than usual, then congestion may have occurred.

IV. The Proposed Methodology

In this section, we present our theoretical approach to guide our system design, which incorporates findings from our preliminary analysis and the analytical framework.

A. Design Setup

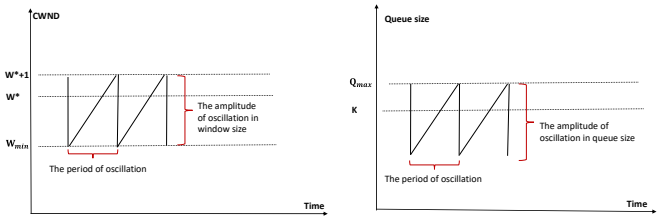
Before we present our proposed solution, we first discuss the buffer size of a switch, which is modeled as the container size in the bin packing problem. In practice, the buffer size B is usually determined according to an established general rule of thumb $B = RTT \times C$ [23], which is the bandwidth-delay product. RTT denotes the average round trip time. Although the rule of thumb is established for the Internet and the recommended size for data center network is $3 \times RTT \times C$, the typical buffer size used in commodity switches that are deployed in a production data center usually follows the general rule of thumb [10], [24]. This rule also provides the sender with an important clue of how much data can be transmitted into the network without resulting in packet drop.

Traffic in the data center generally follows a certain pattern, which is usually determined from the application level. Therefore, to ensure the robustness of our design, we consider traffic patterns that are common in the data center network. Studies in [18], [25] show that the traffic in data centers has ON-OFF patterns. That is, many links run hot for certain periods but are idle at other times. And, in other cases, the data center traffic is continuous [26].

B. Theoretical Solution

Here, we present our theoretical approach, which lays its bases on the Next Fit algorithm, such that each sender determines its transmission rate to avoid buffer overflow.

From the bin-packing perspective, the minimization of the number of "bins" can be interpreted as minimizing the number of rounds that a buffer is filled up with packets and drained. Our scheme assumes the sender and receiver do not have all the information about switch state (e.g., buffer size, queue length, etc.). Therefore, ECN is used to cue the senders on the buffer state (e.g., queue length). Hence, the sender attains crucial information on the number of packets marked and unmarked by ECN. Consequently, we consider the following two scenarios.



(a) Congestion window size (b) Queue size in buffer

Figure 4: Observation in the switch: Window and queue size

Scenario 1. The receiver provides its sender with information on the numbers of packets without ECN marking. One way to accomplish this is to convey the information through TCP receiver window (TCP *Rwnd*) in the ACK packet. Let Q, K, and B denote the queue length, ECN threshold, and buffer size, respectively, such that $0 \leq Q \leq B$. We use $K = \frac{1}{7} \times RTT \times C$ for $0 < K < B$, which is the recommended threshold by DCTCP [10]. Let X^f_{UM} denote the number of packets of flow f unmarked by ECN and assume there are n flows sharing the same congestion point (CoP) in DCTCP, for $f \in n$.

Theorem IV.1. *Buffer overflow caused by incast can be mitigated when each flow f, that shares the same CoP, transmits at most X^f_{UM}.*

Proof. We first consider different cases of traffic with the ON-OFF mode pattern.

Case 1. Each flow $f \in n$ transmits X^f_{UM} amount of packets to an empty buffer (OFF mode pattern). The queue length formed by n flows is upper bounded by

$$Q = \sum_{f=1}^{n} X^f_{UM} = \frac{1}{7} \times RTT \times C,$$

which is also K. Hence, the queue length $Q \leq K < B$.

Case 2. The initial traffic at CoP buffer is *exactly* K (ON mode pattern) before the arrival of traffic from n flows. Then, when the traffic from n flow arrives, each flow $f \in n$ transmits X^f_{UM}. Thus, $Q = 2 \times K = \frac{2}{7} \times RTT \times C$, which is $Q < B$.

Case 3. The initial queue size Q_{init} at CoP buffer is *larger* than K (ON mode pattern) before the arrival of traffic from n flows. We have $Q_{init} = K + \sum_{e=1}^{n} h_e$, where $h_e > 0$ denotes the adaptive congestion window increase step of a long-lived flow e. Let W^*_e be the critical window size of the long-lived flow at which the queue size reaches K, as illustrated in Figures 4a. We have $W^*_e = \frac{K}{N}$, where N is the number of long-lived flows. Then, $W^*_e = (\frac{1}{7} \times C \times RTT)/N$.

Let's assume $W^*_e = 1$, which is also the minimum window size. In other words, this can be interpreted as there are many flows sharing the same link. Let the increase step $h_e = 1$

(Figures 4a), which is also the same increase step used in the original liturature [10]. When W_e^* is increased by h_e, we have $W_e^* = W_e^* + h_e$ of which now $W_e^* = 2$. Therefore, the additional traffic generated by the increase step is $\sum_{e=1}^{N} h_e \approx \frac{1}{7} \times C \times RTT$, which is also an approximation of K. Then, $Q_{init} \approx \frac{2}{7} \times C \times RTT$, which is approximately equivalent to $2 \times K$. As we have demonstrated in case 1 that the total traffic from n flows transmitting at X_{UM}^f is K, the maximum queue length with additional traffic from n flows is

$$Q_{max} = Q_{init} + K \approx 3 \times K = \frac{3}{7} \times C \times RTT,$$

which is $3K$ (Figures 4b). Hence, $Q = Q_{max} \leq B$.

Next, lets consider the scenario when $W_e^* > 1$. This can be interpreted as there are less number of flows sharing CoP such that N is smaller than in the previous scenario with $W_e^* = 1$. Then, $\sum_{e=1}^{N} h_e$ decreases as N decreases, for $h_e = 1$. Hence, $Q < 3 \times K$, which is also less than B.

The analysis of the case when the traffic pattern is continuous is similar to cases 2 and 3. □

We have shown that n flows transmitting packets at X_{UM}^f does not lead to a buffer overflow.

Scenario 2. Let X_M^f be the number of packets with ECN marking associated with flow f. Assume there are n flows sharing the same CoP, where each flow f is transmitting at most at X_M^f. Also, let X_M denotes the sum of packets with ECN marking from n flows sharing the same CoP.

Theorem IV.2. *To avoid Buffer overflow, X_M^f should be divided and transmitted in two batches at different times.*

Proof. The total number of packets with ECN marking X_M transmitted by n flows sharing the same CoP is

$$X_M = \sum_{f=1}^{n} X_M^f \leq B - K.$$

In other words, $B \geq X_M + K$. Next, we consider the following cases.

Case 1. When the buffer is empty (OFF mode pattern), the queue length Q after the arrival of traffic from n flows, with each flow $f \in n$ transmitting at X_M^f, is $Q = X_M$, which is $Q < B$.

Case 2:. When the initial queue size is exactly K (On mode pattern), then Q after the arrival of traffic from n flows, with each flow $f \in n$ transmitting at X_M^f, is $Q = X_M + K$, which means $Q = B$.

Case 3. Let us consider when the initial queue size exceeds K (On mode pattern). As demonstrated in Theorem IV.1, the maximum queue length before the arrival of traffic from n flows is approximately $2 \times K$. So when each flow f is transmitting X_M^f packets, the queue length becomes $X_M + 2 \times K > B$. In other words, the traffic increases by K, which results in packet drop due to a shortage of buffer space.

However, if each flow reduces its number of transmitted packets by $\frac{K}{n}$, such that the total number of transmitted packets per flow is reduced to $X_M^f - \frac{K}{n}$, then the queue length is reduced to $X_M - K + 2K$, which is B. Thus, to avoid buffer overflow, a volume of traffic of size K must be transmitted at a different time. In other words, X_M^f must be transmitted in two batches (or two rounds), each batch of size $\frac{1}{2} \times X_M^f$. In a special case when $X_M^f = 1$, the packet is randomly placed in one of the batches with probability $\frac{1}{2}$.

The analysis for traffic with a continuous pattern is similar to case 2 and 3. □

Corollary IV.2.1. *The packets must be transmitted in three batches (rounds) to mitigate buffer overflow caused by incast,*

Remark. Here, corollary IV.2.1 suggests transmitting a batch of packets based on the number of packets without ECN marking and two other batches of packets based on the number of packets with ECN marking.

Corollary IV.2.2. *The completion time can be shortened by transmitting the first and second batch together.*

Proof. As shown in Theorem IV.1, the queue length Q when traffic pattern is in ON mode and part of the buffer is occupied by $2K$ packets. Together with traffic from the first and second batch, we have $Q = 2K + K + (B - K)/2$. That is $Q = \frac{6}{7} \times C \times RTT$. Hence, $Q < B$. □

Lemma IV.3. *Three batches of packets can be delivered within 2 RTTs.*

Proof. Consider a network topology, which only consists of a pair of sender and receiver connected by a single switch. For simplicity, we disregard the propagation delay. Let T be the time required to drain a full buffer. So the time required to drain the first batch in the switch is at most T, which is also the time when the first batch reaches the receiver. That is $\frac{1}{2}RTT$. At one RTT, which is equivalent to $2 \times T$, ACKs from the first batch arrive at the sender. It is also the time when the second batch reaches the receiver. At the same time, the sender begins transmitting the third batch. Afterward, the ACKs from the second and third batch arrive at the sender at time $3 \times T$ and $4 \times T$ after the first batch is transmitted, respectively. Since $T \approx \frac{1}{2}RTT$, the delivery requires $4 \times \frac{1}{2}RTT$, which is $2\ RTTs$. □

Corollary IV.3.1. *When a path connecting a pair of sender-receiver with at least three hops, the three batches can be completed in at most $RTT + 2T$.*

Proof. It takes a single RTT for the ACKs of the first batch of packets to reach the sender. Since the interval between two batches is at most T, the ACKs from the second and third batch arrive at the sender at T and $2 \times T$ after ACKs of the first batch arrives. Thus, it is at most $RTT + 2 \times T$. □

Remark. The critical insight to our methodology is that our scheme utilizes ECN signal to implicitly inform its senders on how many and how packets should be transmitted. Different

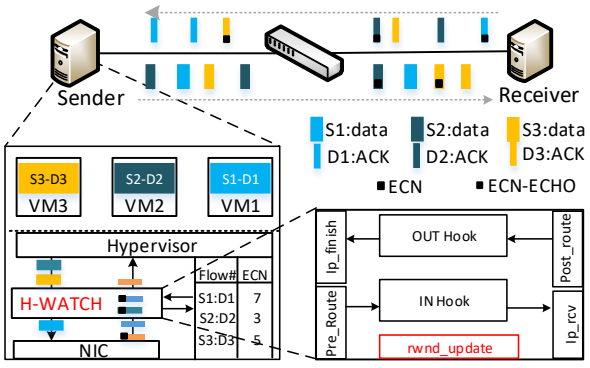


Figure 5: HWatch system is an end-host module that controls TCP receive window values.

from other ECN based techniques (e.g., DCTCP and ECN-RED), the switch only provides binary signal whether the network is congested or not.

In the next discussion, we address the practical challenges in designing and implementing our theoretical solution in a deployable system. Also, we explain how to determine both X_M^f and X_{UM}^f in practice.

C. System Design

Here, we describe our system design of HWatch based upon the theoretical results. HWatch is an end-to-end solution implemented in the hypervisors that measures the congestion level using configurable network congestion signaling; and controls the transmission speed by adjusting the TCP receiver window of the ACK segment headers according to the measured congestion level. HWatch employs ECN, a built-in function readily supported by existing commodity switch hardware, to detect and measure congestion level in the network. Thus, HWatch is also a readily deployable scheme that is compatible with legacy TCP because it is a hypervisor-based solution that relies only on built-in functions in existing hardware. HWatch imposes the following simple mechanisms: *1)* Throughout data transfer, HWatch observes the network congestion level and adjusts the sending speed accordingly. *2)* On Connection start-up, HWatch measures congestion status in the network to fix the appropriate start-up sending rate.

To realize these, HWatch respects the following control rules: *Rule 1*: To avoid the bloated queue in the switches, ECN and flow throttling are used to handle the continuous flow of ECN marks received by the long-lived steady-state flow, leading the associated sender's hypervisor to adjust the receiver flow control window in the ACKs. This reduces the long-lived flow's sending speed. *Rule 2*: For the short-lived flows to avoid incast, the senders' hypervisor infers the congestion level during connection setup from the SYN/SYN-ACK packets and if available from any other packets flowing between the sender/receiver. Then, the receiver determines its receiver window according to the number of probing packets marked by ECN, and the receiver communicates its receiver window size to the sender through the returning SYNACK packet.

HWatch us deployed at both the sender's and the receiver's hypervisor as shown in Figure 5. Generally, to tackle the problem when flow does not receive three DUPACKs to trigger TCP fast re-transmission, the sender transmits standard TCP packets into the network. However, in the presence of congestion, the receiver determines the size of its receiver window (*Rwnd*) in proportion to the number of packets unmarked and marked with ECN marking. After that, the receiver communicates the size of *Rwnd* to the sender via the returned ACK packets.

HWatch tackles the issues when packets are dropped due to buffer overflow triggered by the large initial sending window of TCP. It infers network congestion from ECN marks in SYN/SYN-ACK packet and any other packets already flowing between the source-destination pairs. Also, it may inject raw IP dummy packets from hypervisor-to-hypervisor before the SYN packet to have a more accurate measure of the congestion level in the network. The dummy (probe) packets will carry the ECN marks to the receiver in the case of congestion. Then, the receiver applies the window re-sizing scheme directly in proportion to the number of packets untagged and tagged with ECN congestion marks, and then conveys it to the sender through the ACK packets. In other words, HWatch compensates short-lived flows having insufficient in-flight packets to probe the network and trigger duplicate ACKs to signal packet loss (e.g., losing the entire or large proportion of the window).

However, without careful consideration of what the probe packet size should be, the probing mechanism may unnecessarily overload the network with the probe packets. To resolve these issues, we use the following probe packets:

1) $Probe_1$: Probe packet with at least zero Bytes of payload (or raw IP packets) of size less than or equal to 38 bytes (i.e., ETH (18 Byte) + IP (20 Byte) headers + Payload $\geq$ 0), and;
2) $Probe_2$: The data packets, generally filled with payload of size 1500 bytes, are used as the probes for later rounds.

$Probe_1$ is to ensure the probing mechanism has minimal impact on the network performance during the connection establishment. However, to have a continuous flow of probe packets for after connection setup phase, HWatch utilizes both $Probe_1$ and $Probe_2$ packets together to measure the network condition throughout the flow life-time.

We note that the accuracy of the queue occupancy estimation during the connection setup might be dependent on the number and inter-departure times of the initial probes ($Probe_1$). HWatch scheme seeks to balance the probing accuracy while minimizing the unwanted impact on network performance. This can be achieved by using the small 38-byte raw IP packets and adding non-uniform delay among the probes so that their inter-departure times are not zero nor uniform.

The number of probing packets used is determined by the operators according to the definition of short-lived flows or the default initial window size. In our experiment, the Linux default initial window size is set to 10. Thus, we same value for the probes at the connection setup and so each flow transmits 10 probing packets of type $Probe_1$. The inter-departure times for probe packets can be pre-set by the operator or drawn from a random non-uniform number. The total transmission

time for all probes need to be within a bounded range so as not cause severe delay for connection setup (or hand-shake) process (e.g., a reasonable value would be RTT/2 worth of delay before sending out the SYN).

D. System Implementation

In our implementation, HWatch scheme is realized in the Hypervisor-level shim-layer data processing path of data center end-hosts (servers) of both the sender and receiver. The realization is accomplished in the following two ways:

1) by implemented a Kernel Module that utilizes the NetFilter framework [27] to intercept and process the arriving and outgoing packets as depicted in Figure 6; or
2) by expanding the data-path of the virtual switch (e.g., Open vSwitch [28]) with HWatch modules as shown in Figure 7.

In both approaches, HWatch modules and the shim-layer implementation have sender and receiver version. One module processes incoming and outgoing packets, including handling the events of SYN, SYN-ACK, and ACK packets, as well as timer expiry. In HWatch, all traffic from and to the guest VMs goes through HWatch modules for further inspection and processing. The *Rwnd* and checksum field are updated with new information according to the conditions observed through SYN-ACK at the receiver or the ACKs at the sender. Then, HWatch forwards the new ACK or SYN-ACK. HWatch utilizes token buckets to pace between batches of SYN-ACK packets.

At the connection-setup stage, HWatch module hashes the flow, retrieves the relevant information, and stores it into a flow-table indexed by the flow's 4-tuples (i.e., source IP, dest. IP, source Port, and dest. Port). Also, HWatch stores other various information, including the number of non-ECN, ECN marks, the window scale factor, and so on. Flow table entries are cleared when the connection is terminated (i.e., FIN packet is received or sent by a guest VM). Information in the receive window field is updated when the end-hosts have a new TCP checksum value. The shim-layer (or module) is allocated directly above the NIC driver for a non-virtualized setup and directly below the hypervisor to provide effective supports to VMs in the data center.

In both cases, the module or the shim-layer is implemented at both the sender and receiver. That is the module would implement one function for incoming and outgoing processing which handles the events of SYN, SYN-ACK and ACK packets arrival and timer expiry. In HWatch, all incoming and outgoing traffic to the guest VMs pass through HWatch module for further processing. When the conditions for modifying either the SYN-ACK at the receiver or the ACKs at the sender are met, the receive window and checksum field are updated and the new ACK or SYN-ACK are forwarded. The HWatch module, at connection-setup, hashes the flow, extracts the relevant information and stores it into a flow-table indexed by the 4-tuples of the flow (i.e., source IP, dest. IP, source Port and dest. Port). It stores various state information including the window scale factor, number of non-ECN, ECN marks, and so on. Flow entries are cleared from the table when the connection

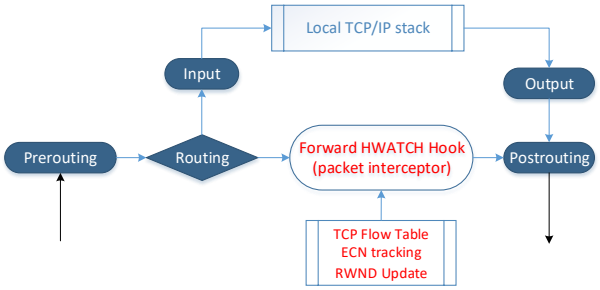


Figure 6: HWatch is implemented via netfilters by inserting hooks into the forward processing path of incoming and outgoing packets

is closed (i.e., FIN is sent by a guest VM). TCP check-sum value is recalculated the at the end-hosts whenever the receive window field is updated. The shim-layer (or module) resides right above the NIC driver for a non-virtualized setup and right below the hypervisor to support VMs in cloud data centers.

HWatch is implemented as a loadable Linux kernel module via the NetFilter packet processing mechanisms [27]. The system adds hooks that attach to the forward processing path of the TCP/IP stack in the Linux kernel. Since the built module is loadable, deploying HWatch into the host operating system means the TCP/IP implementation of the guest operating system remains intact. Figure 6 shows that the NetFilter hooks are inserted into the forwarding stage of packet processing. The hook intercepts the entire forwarded (incoming/outgoing)/ TCP packets not destined to the host machine. When TCP packets are intercepted, their headers are checked, and their intended processing is determined based on its type (i.e., whether it is SYN-ACK, FIN, and ACK packet).

HWatch implementation can also be incorporated into virtual switches employed by most hypervisors for the purpose of inter-connecting the guest VMs to the physical network. As shown in Figure 7, OpenvSwitch (OvS) can be patched to modify its Kernel data-path modules with the mechanisms of HWatch described in Section IV. For OvS implementation, there are no NetFilter hooks uses. However, HWatch's flow table, ECN tracking, and window update functions are added to the packet processing logic of the kernel data-path module of OvS. In virtualized data centers, OvSes patched with HWatch can process the traffic for inter-VM, Intra-Host, and Inter-Host communications. The method of deploying HWatch in production data centers can be done efficiently by applying the HWatch patch and then recompiling the OvS Linux kernel module.

E. Practical Challenges

In the following discussion, we demonstrate that HWatch is a relatively simple solution that is capable of resolving non-recoverable packet losses and achieve satisfying performance in various simulation scenarios. For instance, HWatch reduces the average and variance of FCT among small flows compared to existing proposals. Since HWatch is a hypervisor-based solution, it does not require any modification of the network stack in the guest OS, allowing our solution to be more readily

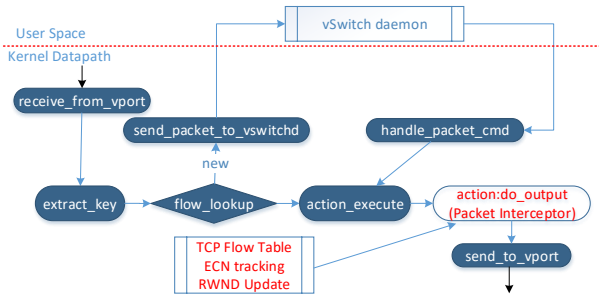


Figure 7: HWatch can be realized by modifying the OpenvSwitch (OvS) kernel datapath module (i.e., adjust the flow table processing)

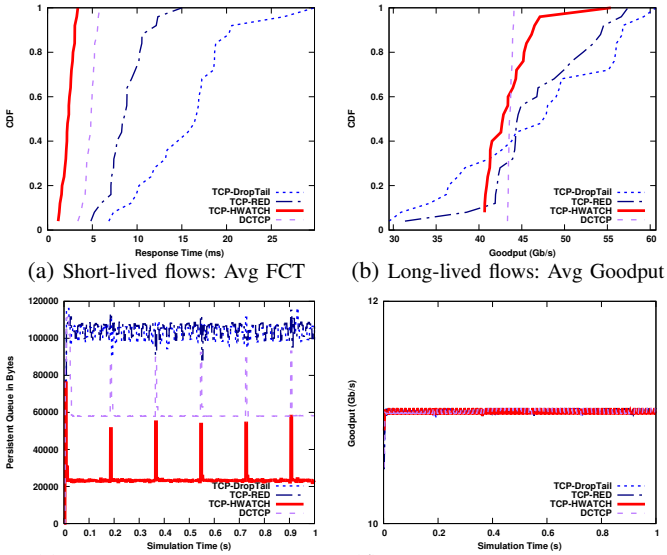
deploy-able in the current production data centers without major interruption to the system.

One of the practical challenges to overcome is the use of window scaling TCP. As described in TCP specifications [29], the scaling is achieved via means adding a three-byte scale option field to the TCP header in all segments. Otherwise, the scale value is exchanged at the stage of connection-setup during the SYN segments exchange. For reducing overhead reasons, most OS implementations, including Linux adopt the latter approach. Practically, scaling the window may be unnecessary for networks with Bandwidth-Delay Product (BDP) less than 31.25 KByte (i.e., 1 Gb/s for an RTT of 250μs). However, when links operate at higher speeds of 40 Gb/s (i.e., BDP=1.25 Mbyte) or 100 Gb/s (i.e., BDP=3.125 Mbyte), scaling the window becomes essential for efficient bandwidth utilization. Hence, in HWatch design, the flow table also keeps track of the scale factor exchanged during the connection setup of TCP flows. Then, the sender and receiver could use the stored scaling factor to re-scale the incoming and outgoing receive window field respectively before and after updating the TCP receive window field.

HWatch relies on congestion feedback from the network to infer the congestion level (e.g., ECN marks). Hence the efficiency of it relies on the appropriate setting of the marking thresholds in the routing devices [30]. RED is the AQM used for enabling ECN marking in the switches and is commonly available in the entry-level data center switches. However, RED is known to be sensitive to the parameter settings [31], [32]. [31] gives a guideline on the right RED parameter setting. However, the choice of these settings depends on the characteristics of the underlying network (e.g., link speeds, buffer size, and the RTT). HWatch also relies on Weighted RED AQM for ECN marking, and its settings are inherited from recommended DCTCP [10]. In this setting, the low and high marking thresholds are set so that packets are ECN marked if the instantaneous queue occupancy exceeds the quarter of the buffer size. We follow the recommended DCTCP settings as it has been shown to perform well in data centers analytically [33] and practically [10].

F. Mutipath TCP (MPTCP)

HWatch can be extended to improve the performance of MPTCP [34], which is a variant of TCP that allows a single connection to use multiple paths simultaneously. MPTCP opens



(a) Short-lived flows: Avg FCT (b) Long-lived flows: Avg Goodput

(c) Persistent queue over time (d) Bottleneck utilization over time

Figure 8: Performance of short-lived and long-lived flows in 50 sources scenario

multiple TCP connections between a sender and a receiver. The first connection in MPTCP is performed according to a regular TCP connection establishment scheme. After that, the additional TCP connections are established one by one with an extra step to bind with the first connection. Thus, since every connection establishment in MPTCP relies on TCP, HWatch logic can be directly applied to MPTCP. We leave the extension of this study for future work.

V. Simulation Analysis

In this section, we evaluate the performance of HWatch in a large-scale data center network. We conduct simulations in ns2 and compare HWatch with the state-of-the-art schemes.

The number of probes used in HWatch is set to 10, which is chosen so that the overhead level can be tolerated. The ECN marking threshold on the switches is set to 20% of the buffer size. The parameters of DCTCP and TCP-RED are set according to the recommended settings by their authors [10], [31]. We use ns2 version 2.35 [35], which we have extended with the HWatch module, and it is installed as a new processing layer (like hypervisor) on the end-hosts. We compare TCP NewReno with Droptail, TCP-RED, DCTCP, and TCP with HWatch module. For DCTCP, we use a patch for ns2.35 available from the authors [36]. ECN-bit capability is enabled on the switches as well as both the TCP sender and receiver. Here, our experiments utilize high-speed links of 10 Gb/s for sending stations, a bottleneck link of 10 Gb/s, low RTT of 100 μs, and RTO_{min} of 200 ms. We had to modify the TCP implementation in ns2 because there is no flow-control (or receiver window processing), which is different from the standard TCP implementation.

Simulation Results: We simulate various scenarios with 50 sources, which consists of 1:1 long-lived to short-lived TCP flows ratio. These scenarios combine the effects of incast

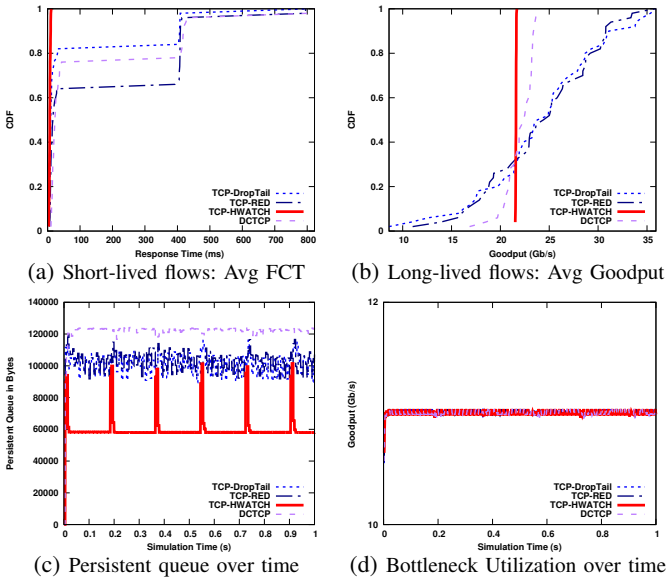
congestion resulting from the short-lived flows and buffer-bloating resulting from the pressure of the long-lived flows.

The flows start at the beginning and keep sending at full speed during the whole simulation period. The small flows initiate 6 epochs of data transfer during the whole simulation. In each epoch, the short-lived flows each flow transmits 10 KBytes of data in random order. The inter-arrival time between two consecutive short-lived flows is randomly selected with an average equal to the transmission time of a single segment. By doing so, the simulator generates the incast problem where short-lived flows starting time are correlated. We analyze the CDF of the average flow completion time (FCT) of short-lived flows over the incast rounds, the persistent queue size, the goodput of long-lived flows, and the link utilization.

Figure 8 shows the results of this experiment. Figure 8a shows that HWatch improves the average FCT for short-lived flows compared to other schemes. That is $3\times, 5\times$, and $10\times$ improvement compares to DCTCP, TCP-RED, and DropTail, respectively. The results indicate they can avoid losses and hence expensive RTO. Figure 8b demonstrates that the goodput of long-lived flows achieves almost the same results as DCTCP, which is a result of regulating receive window for these long-lived flows. Figure 8c shows that HWatch can control queue occupancies at the targeted low threshold levels, and Figure 8d shows that the bottleneck link is fully utilized with HWatch like other schemes.

To study the scalability of HWatch, we repeat the same experiment while doubling the number of sources to 100. Figure 8 depicts the outcome of our experiments. Figure 9a further illustrates that HWatch can significantly improve the average FCT for all short-lived flows and avoid the consequences of timeout (i.e., no FCT exceeds tens of milliseconds). This is because HWatch is capable of mitigating packet drops caused by buffer overflow and delivers packets in fewer RTTs (e.g., avoid re-transmission). Moreover, Figure 9b, 8d and 9d match the findings of previous scenario regarding long-lived flows' goodput, queue occupancy and link utilization.

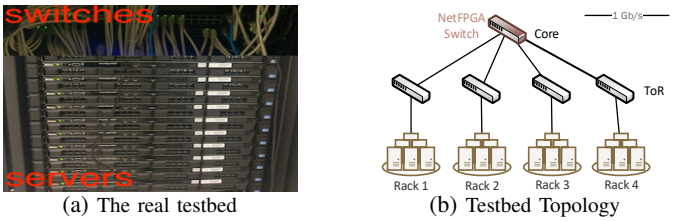
Discussion: The favorable HWatch's performance is due to the following reasons. First, dispersing packets transmission at different creates in smaller size of incast allowing the buffer to observed the incoming packets and mitigate packet drop problem, which results in faster FCT, especially the short-lived flows. Secondly, another insight is that packet(s) of the smaller short-lived flow (e.g., a flow with one or two packets) can be transmitted and completed in the first batch. In contrast, some of packets belongs to larger short-lived flows are transmitted later in batches two and three so that these flow consume less resources. By doing so, HWatch stochastically prioritizes the available buffer space to smaller short-lived flows. Thirdly, HWatch's probing mechanism functions as incast early warning system allowing long-lived flows that are actively sending data to scale back and reduce the transmission rate to release some buffer space This is achieved when some packets of the active flows are marked by ECN during the probing. By doing so, HWatch strikes the balance between improving the performance of short-lived flows while keeping minimum impact to the long-lived flows.



(a) Short-lived flows: Avg FCT (b) Long-lived flows: Avg Goodput

(c) Persistent queue over time (d) Bottleneck Utilization over time

Figure 9: Performance of short-lived and long-lived flows in 100 source scenario



(a) The real testbed (b) Testbed Topology

Figure 10: Real testbed used in the experiments

VI. Testbed Experiment Results

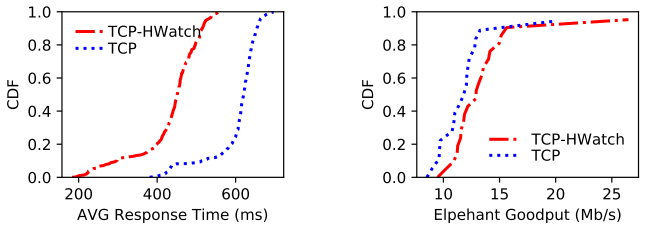
A. Testbed Setup

To put HWatch to the test, we utilize a testbed consisting of 84 virtual servers interconnected via 4 non-blocking leaf switches and 1 spine switch in our mini data center.

Our testbed, as shown in Figure 10, has 4 racks (rack 1, 2, 3 and 4). Each server per rack is connected to a leaf switch via 1 Gbps link. The spine switch is realized by running a "reference_switch" image on a 4-port NetFPGA card [37], which is installed on a desktop machine. The network's base RTT is around 200 microseconds. The servers run Ubuntu Server 14.04 LTS with Linux kernel v3.18.

The HWatch end-host module is invoked and installed on the host OS whenever necessary only. HWatch runs with the default settings (i.e., RTO of 4ms). A traffic generator is employed to run the experiments which generate short-lived traffic (i.e., parallel web-page requests from Apache web servers hosting an object of size 11.5KB). In addition, we utilize the iperf program to generate long-lived traffic (e.g., VM migrations, backups).

We set up a scenario to produce both a collision between suddenly arriving short-lived and long-lived flows situation with bottleneck link in the network being at the network core. The traffic is generated from and to iperf and/or Apache client or



(a) Short-lived flows: Avg FCT (b) Long-lived flows: Avg Goodput

Figure 11: Experimental results of the real testbed

server processes. Each process is created and associated with its unique virtual ports on the virtual switch (i.e., OpenvSwitch). By doing so, we can generate a large number of flows (in thousands). This set-up ultimately mimics a data center with a large number of flows.

Experiment Results In our evaluation, we consider a scenario in which TCP short-lived flows start at different times while long-lived flows are taking over the network bandwidth. Specifically, a burst of short-lived TCP flows is introduced to compete for bandwidth in a short period of time. We first generate 7 TCP iperf flows from each sending rack for 30 secs resulting in 42 ($2 \times 7 \times 3 = 42$) long-lived flows at the bottleneck. At the 10^{th} sec, we use traffic generators on receiving rack to request the webpage (1000 times) from each of the 7 web servers on 3 racks. We use 10 parallel connections to request the web pages ($7 \times 6 \times 3 \times 10 = 1260$ flows in total) and repeat the ice epoch 5 times ($1260 \times 5 = 6300$ flows in total). This results in high traffic pressure as many flows come in and out in a short period.

Based on the outcome from the experiments, we make the following observations. Figure 11a suggests that short-lived flows still benefit from HWatch by achieving comparably shorter flow completion time on average (up to 100% improvement). At the same time, Figure 11b demonstrates that the TCP long-lived flows are not affected by the significant disruption caused by the intermittently arriving short-lived flows. This indicates that HWatch is very useful in apportioning the link capacity. In summary, our experiments confirm that HWatch mitigates packets drop and delivers packets in less RTT. Thus, HWatch tackles congestion and allocates the capacity among various flow types well. The testbed experiments confirm the findings in the simulation experiments of why HWatch performs well. Since the probing scheme functions as early warning system, the advantages of HWatch lay in its speed and effectiveness of responding to the congestion warnings in timely manner.

VII. Related Work

The current existing work point to TCP timeouts as the culprit for low throughput and high latency problems in data centers [38], [39], which often leads to performance degradation of latency-sensitive applications [40], [41]. There are several solutions to this problem.

The simple method to resolve the problem of the outstanding waiting time of RTO is to lower the default MinRTO value [40], [39]. This approach is proven to be practical and can be quickly implemented. However, when the right minRTO is not set, TCP congestion window is frequently backs off to 1 MSS. Consequently, this results in low TCP transmission rate and hence low bandwidth utilization (or throughput collapse). Moreover, reliance on modification of static MinRTO is not scalable in large-scale heterogeneous networks. This approach also requires modification to the TCP stack of the tenant's VM, which makes deployment more difficult due to lack of access for operators to the guest OS.

Another approach is via controlling the queue build-up at the switches using feedback mechanisms (E.g., explicit congestion control or receiver window). This limits how much TCP packets can be transmitted into network [10], [42], [5], [12], [13], [43], [44], [45], [46], [47]. By doing so, it improves the performance of short-lived flows lowering their flow completion time while at the same time allowing long-lived flows to obtain higher link utilization. However, similar to solutions discussed above, this approach also requires modification at the TCP stack, completely new switch design, or prone to fine-tuning of various parameters in the operating system or in the applications, which also make the solution more difficult to deploy.

VIII. Conclusion

In this paper, we proposed HWatch to improve the performance of flows in data centers and address the buffer overflow problem by measuring the congestion level during connection establishment. Our scheme, in particular, allows short-lived flows to mitigate early packet drops in a highly congested network in the data center, which results in a faster FCT for short-lived flows. We demonstrate the benefits and effectiveness of HWatch through large scale simulation and testbed experiments in a data center.

References

[1] J Dean and S Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Communications of the ACM (CACM)*, page 107–113, 2008.

[2] Apache. Spark: Lightning-fast cluster computing.

[3] Todd Hoff. Google: Taming The Long Latency Tail - When More Machines Equals Worse Results.

[4] M. Mattess, R. N. Calheiros, and R. Buyya. Scaling MapReduce Applications Across Hybrid Clouds to Meet Soft Deadlines. In *Proceedings of IEEE AINA*, 2013.

[5] Ahmed M. Abdelmoniem and Brahim Bensaou. Reconciling Mice and Elephants in Data Center Networks. In *Proceedings of IEEE CloudNet*, 2015.

[6] Ahmed M. Abdelmoniem and Brahim Bensaou. Incast-Aware Switch-Assisted TCP Congestion Control for Data Centers. In *IEEE Global Communications Conference (GlobeCom)*, 2015.

[7] Jiawei Huang, Tian He, Yi Huang, and Jianxin Wang. ARS: Cross-layer adaptive request scheduling to mitigate TCP incast in data center networks. In *Proceedings of IEEE INFOCOM*, 2016.

[8] Ke Wu, Dezun Dong, Cunlu Li, Shan Huang, and Yi Dai. Network congestion avoidance through packet-chaining reservation. In *Proceedings of ACM ICPP*, 2019.

[9] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of EuroSys*, 2019.

[10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, 2010.

[11] Ahmed M. Abdelmoniem, Brahim Bensaou, and Amuda James Abu. HyGenICC: Hypervisor-based Generic IP Congestion Control for Virtualized Data Centers. In *Proceedings of IEEE ICC*, 2016.

[12] Amuda James Abu, Brahim Bensaou, and Ahmed M. Abdelmoniem. A Markov Model of CCN Pending Interest Table Occupancy with Interest Timeout and Retries. In *IEEE International Confereence on Communications (ICC)*, 2016.

[13] Amuda James Abu, Brahim Bensaou, and Ahmed M. Abdelmoniem. Leveraging the Pending Interest Table Occupancy for Congestion Control in CCN. In *Proceedings of IEEE LCN*, 2016.

[14] C. C. Lee and D. T. Lee. A simple on-line bin-packing algorithm. *Joural of the ACM*, 32(3), July 1985.

[15] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: stability, convergence, and fairness. In *Proceedings of ACM SIGMETRICS*, 2011.

[16] Glenn Judd. Attaining the promise and avoiding the pitfalls of TCP in the datacenter. In *Proceedings of NSDI*, 2015.

[17] Ahmed M. Abdelmoniem and Brahim Bensaou. Enforcing Transport-Agnostic Congestion Control via SDN in Data Centers. In *Proceedings of IEEE LCN*, 2017.

[18] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of ACM IMC*, 2010.

[19] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of ACM SIGCOMM*, 2014.

[20] Hengky Susanto, Hao Jin, and Kai Chen. Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks. In *Proceedings of IEEE ICNP*, 2016.

[21] Hengky Susanto, Ahmed M. Abdelmoniem, Honggang Zhang, Benyuan Liu, and Don Towsley. A near optimal multi-faced job scheduler for datacenter workloads. In *Proceedings of IEEE ICDCS*, 2019.

[22] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM*, 2008.

[23] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. In *Proceedings of ACM SIGCOMM*, 2004.

[24] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ecn for data center networks. In *Proceedings of ACM CoNEXT*, 2012.

[25] Srikanth Kandula, Sudipta Sengupta, Albert G. Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of ACM IMC*, 2009.

[26] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (Datacenter) network. In *Proceedings of ACM SIGCOMM*, 2015.

[27] NetFilter. NetFilter Packet Filtering Framework for linux.

[28] OpenvSwitch.org. Open Virtual Switch project.

[29] J Postel. RFC 793 - Transmission Control Protocol, 1981.

[30] Pica8. PICOS documentation.

[31] Sally Floyd. Red parameters setting.

[32] C.V. Hollot, V. Misra, D. Towsley, and Wei-Bo Gong. A control theoretic analysis of RED. In *Proceedings of IEEE INFOCOM*, 2001.

[33] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability analysis of QCN. *ACM SIGMETRICS Perf. Eval. Review*, 39, 2011.

[34] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proceedings of USENIX NSDI*, 2011.

[35] NS2. The network simulator ns-2 project.

[36] Mohammad Alizadeh. Data Center TCP (DCTCP).

[37] netfpga.org. NetFPGA 1G Specifications. http://netfpga.org/1G_specs.html.

[38] Jiao Zhang, Fengyuan Ren, Li Tang, and Chuang Lin. Modeling and Solving TCP Incast Problem in Data Center Networks. *IEEE Transcations of Parallel and Distributed Systems (TPDS)*, 26, 2015.

[39] Ahmed M. Abdelmoniem and Brahim Bensaou. Curbing Timeouts for TCP-Incast in Data Centers via A Cross-Layer Faster Recovery Mechanism. In *Proceedings of IEEE INFOCOM*, 2017.

[40] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proceedings of ACM SIGCOMM*, 2009.

[41] Ahmed M. Abdelmoniem and Brahim Bensaou. Hysteresis-based Active Queue Management for TCP Traffic in Data Centers. In *Proceedings of IEEE INFOCOM*, 2019.

[42] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM Transactions on Networking*, 21, 2013.

[43] Ahmed M. Abdelmoniem, Brahim Bensaou, and Amuda James Abu. SICC: SDN-based Incast Congestion Control for Data Centers. In *Proceedings of IEEE ICC*, 2017.

[44] Ahmed M. Abdelmoniem, Brahim Bensaou, and Amuda James Abu. Mitigating TCP-Incast Congestion in Data Centers with SDN. *Special issue on Cloud Communications and Networking, Annals of Telecommunications*, 2017.

[45] A. S. Sabyasachi, H. M. D. Kabir, A. M. Abdelmoniem, and S. K. Mondal. A resilient auction framework for deadline-aware jobs in cloud spot market. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 247–249, 2017.

[46] Ahmed M. Abdelmoniem, Brahim Bensaou, and Victor Barsoum. IncastGuard: An Efficient TCP-Incast Congestion Effects Mitigation Scheme for Data Center Network. In *Proceedings of IEEE GlobeCom*, 2018.

[47] Ahmed M. Abdelmoniem, Yomna M. Abdelmoniem, and Brahim Bensaou. On Network Systems Design: Pushing the Performance Envelope via FPGA Prototyping. In *IEEE international Conference on Recent Trends in Computer Engineering (IEEE ITCE)*, 2019.