Abstract

Congestion control algorithms rely on a variety of congestion signals (packet loss, Explicit Congestion Notification, delay, etc.) to achieve fast convergence, high utilization, and fairness among flows. A key limitation of these congestion signals is that they are either late in feedback or they incur significant overheads. An ideal congestion control must discover any available bandwidth in the network, detect congestion as soon as link utilization approaches full capacity, and react timely to avoid queuing and packet drops, without significant overheads. To this end, this work proposes Scout service that leverages priority queues to infer “bandwidth availability” and “link busyness” at the host. The key observation here is that as the high priority queue (HPQ) gets busier, the low priority queue (LPQ) is served less. Therefore, the state of the link can be observed from the LPQ and any congestion can be detected several RTTs earlier than observing the HPQ. We propose a new transport protocol, Double-Window Transmission Control Protocol (DWTCP) that builds upon the Scout service to dynamically adjust its congestion window. Our testbed and simulation-based evaluation demonstrates that Scout enables a data center transport to achieve high throughput, near-zero queues, lower latency, and high fairness.

1 Introduction

Congestion Control (CC) algorithms rely on a variety of congestion signals (such as packet loss, Explicit Congestion Notification (ECN), delay, or INT) to achieve fast convergence, short flow completion times, small queues, high link utilization, and high fairness among the flows. Today’s data centers serve applications that generate highly dynamic network traffic. The performance of these applications is strictly dependent on the efficiency of the underlying CC algorithm. The efficiency of CC is tightly coupled with how quickly and accurately it can detect congestion in the network.

Existing congestion signals have limitations that impact CC solutions relying on them. Some congestion signals provide late feedback, i.e., they provide network state information when congestion has already happened in the network (e.g., loss [7] or delay [27]). The fundamental reason for the late feedback is built-in to the inherent design of these congestion signals such that all these signals rely on the behavior or state of the queues in the network. For example, packet drops happen when the queue is already full, and ECN is marked when the queue is above a certain threshold. Similarly, delay is observed when the link is already over-provisioned. Due to the limitations of these signaling mechanisms, the underlying congestion control algorithms use conservative control laws and struggle to achieve fast convergence, high link utilization and fairness. Recent solutions such as Inband Network Telemetry (INT) [1][2] and credit-based [14] provide rich signals that can alleviate this problem. However, these solutions incur overheads and require changes in the switches, which may not always be feasible.

Figure 1: Under strict priority queueing, low priority packets can only be served if the high priority queue is idle. This effect can be used as a congestion signal.

In this work, through Scout service, we demonstrate that one can get an early and amplified congestion signal from the network switches, using a basic feature (priority queues) available in commodity switches, without the need...
for switch modification or imposing any complicated algorithm at the host. Scout service is built based on a simple yet effective idea of using priority queues in the switches to get the congestion state. The key observation here is that as the high priority queue (HPQ) gets busier (higher link utilization), the low priority queue (LPQ) gets less chance to be served (Figure 1). Therefore, the state of the link can be observed from the LPQ and any congestion can be detected several RTTs before observing the HPQ building up as done by the existing congestion signals.

Scout service allows end-hosts to send Scout traffic (with low priority) to the receiver alongside the normal traffic (with high priority) and then monitors the Scout acknowledgments (ACKs) from the receiver to infer “bandwidth availability” and “link busyness” (or early indication of congestion). At the switch side, as shown in Figure 1, the normal traffic (blue packets) is served with strict high priority, while Scout traffic (red packets) is served in the low priority queue, i.e., only when the output port is idle. In this setting, Scout traffic can only fill in the gaps between normal traffic packets. Such behavior is an implicit guarantee from the network switches that it has available bandwidth equal to the number of Scout packets. In addition, the absence (loss) of, or a delay in serving Scout packets indicates a busy link which provides an early indication of congestion.

Scout service is fundamentally different from existing credit-based schemes [14], probing-based schemes [8, 9, 16, 15, 26], and INT-based schemes [23]. Compared to credit-based schemes [14], Scout service does not need to set aside any bandwidth for processing credits at the switches; instead, it uses network bandwidth only when there is no data traffic. Compared to existing probing based schemes [9, 16, 15, 26], Scout service uses low priority queues to probe the network, which not only provides significant amplification in congestion signal, but also significantly reduces the overhead of Scout traffic, as the Scout traffic is served only if there is available bandwidth along the path. Compared to INT based schemes, Scout service does not require switches to make any computations and update packet headers, and has a lower overhead.

To demonstrate the effectiveness of Scout, we present a new transport protocol, Double Window Transmission Control Protocol (DWTCP), that dynamically adjusts the congestion window size of a flow based on the Scout feedback. The key idea of DWTCP is to increase aggressively if the network has more bandwidth available and slow down as the link busyness approaches the full link utilization. Since Scout provides early and amplified congestion signal, this aggressiveness improves convergence as well as fairness.

DWTCP is a complete end-host solution and requires no changes in the switch architecture. At its core, DWTCP uses transmission control protocol (TCP) as the flow control mechanism. Bandwidth availability (from Scout) informs the sender of any available bandwidth in the network, and as a result, senders increase their window aggressively for faster convergence. Similarly, the loss and delay in Scout ACKs inform the sender about the link busyness which is used by DWTCP to adjust its transmission rate.

A Scout based transport has the following key properties:

- Scout enables a congestion control algorithm to discover any available bandwidth in the network, detect congestion significantly earlier than existing congestion signals (i.e., as soon as the link utilization approaches full capacity), and to react timely to avoid queue buildup and packet drops.

- Scout does not incur much overhead in terms of network bandwidth and switch buffer utilization. First, Scout packets are served only if there is available bandwidth. Second, Scout packets are very small (64 bytes in our implementation). Third, Scout does not require switches to reserve too much buffer capacity for LPQ. We demonstrate that even LPQ of 500 bytes (i.e., 5-10, 64-byte Scout packets) results in the best performance.

- DWTCP detects and reacts quickly to network changes and is able to maintain near-zero queues, converges fast (maintain similar throughput as DCTCP even with near-zero queues), and significantly reduces the latency of short flows (details in §2).

- DWTCP is not sensitive to parameter settings. We mathematically analyze the stability of DWTCP and deduce the stability conditions (see Appendix A).

Mathematically, we can analyze the stability of DWTCP by considering the stability of DWTCP and deduce the stability conditions (see Appendix A). Evaluation also suggests that the recommended parameters provide best performance across a range of network settings.

We evaluate DWTCP using a combination of NS3 simulations and a Linux kernel testbed implementation. For the latter, we implement a loadable kernel module that does not require any changes in Linux’s TCP stack. We compare against DCTCP [5], TCP Reno [7] and HPCC [23] using a range of workloads and network settings. For the long-lived flows, DWTCP improves convergence time by 2x, and achieves near perfect fairness. For more realistic data center settings, DWTCP maintains near-zero queue lengths without affecting throughput or incurring long flow completion times even at 80% load. Moreover, at high loads, compared to DCTCP (and HPCC), DWTCP improves the mean latency and tail latency by up to 1x (1.7x) and 3x (5x), respectively, for the datamining workload and up to 2x (3x) and 3.5x (5x), respectively, for the websearch workload. At low loads, HPCC performs 31% better than the DWTCP. We observe similar trends in our testbed results. Finally,
Scout incurs very little overhead in the network (less than 0.2% in our experiments).

2 Design Rationale

Timely detection of congestion is crucial to the transport layer design. Traditionally, CC has relied on reactive congestion signals (such as packet loss [7], delay [27, 22], or ECN [5]) that react to queue occupancy or the proactive congestion signals (such as credits or INT) that predict the link state even before congestion occurs. Reactive congestion signals provide delayed feedback (due to their reliance on the queue length) when the congestion has already happened. Proactive congestion signals provide early feedback, but current implementations incur significant overheads in terms of either probing (Credits [14]) or adding extra header fields (XCP [21], INT [1], etc.).

Next, we show that one can get much early and amplified congestion signals, compared to packet loss and delay, by sending low rate and low priority Scout packets and observing the impact of normal traffic (high priority) on Scout packets.

![Figure 2](image.png)

(a) Low priority queue delay / high priority queue delay (b) Packet loss time difference between HPQ and LPQ

Figure 2: Amplification effect of low priority Queues. a) Packets in low priority queue (LPQ) observe significant delays compared to the packets in the high priority queue (HPQ) as link capacity approaches maximum capacity. b) Packets in LPQ observe packet drop hundreds of RTTs before packets in the HPQ, depending on the aggressiveness of ingress traffic.

2.1 Signal from Low Priority Queue

Observing low priority queue (LPQ) serves as an early (several RTTs) indication of congestion and provides significant amplification in congestion signal (such as loss and delay). To understand, let us consider a single bottleneck topology, where a fixed number of senders generate UDP traffic with gradually increasing rates. We compare the delay and packet loss experienced by the packets in the low priority queue and the high priority queue.

Figure 2a shows the amplification in one way delay (OWD) of low priority traffic compared to one way delay of high priority traffic. We observe that as the link utilization approaches 100%, the packets in low priority queue get fewer chances to be served and hence experience much higher delays. Figure 2b shows a significant difference in the time we observe packet loss from the low priority traffic compared to the high priority traffic. The magnitude of these gains can vary based on the traffic arrival patterns and queue size of the high priority queue at the switches, but the amplification (higher delay) and early notification (loss observed sooner) are properties that can be used to enhance congestion control.

2.2 Benefits for Congestion Control

Early and amplified congestion feedback from the low priority queues help congestion control in terms of high throughput, low queue sizes and small flow completion times. To show this, we present a simple experiment.

Setup. We compare Transmission Control Protocol (TCP) NewReno (loss-based), Data center Transmission Control Protocol (DCTCP) (ECN-based), and DWTCP (Scout service-based) using a single bottleneck topology (10 Gbps links, 100 µs (RTT), and 250 packets buffer). In this setup, five senders generate background traffic and 10 senders send a burst of shortflows (50KB) every 0.1 s. We evaluate this scenario using NS3 simulations.

Signals from High Priority Queue. Loss-based CC schemes rely on the state of the queues and detect congestion when the network is already congested. As a result, the senders using TCP NewReno reduce their congestion window aggressively, creating large variations in the queue occupancy. Although, this results in higher throughput for the long flows, it impacts short flow latency as they have to wait longer in the queue as shown in Figure 3a.

ECN alleviates some challenges associated with the loss-based schemes by providing early indication of congestion. However, its performance depends on the queue size. Figures 3b and 3c show that DCTCP can maintain high throughput and low queues (around its threshold) respectively. However, the completion time of short flows depends on the ECN threshold (Figure 3d).

Signals from Low Priority Queue. From Figure 3d we can observe that feedback from the low priority queue helps in detecting congestion earlier than both ECN and loss, hence maintaining small queues. This allows the sender to be aggressive in its control laws (proportional to link busyness). Note that all the other protocols observe packet drops when a burst of short flows arrives. In contrast, DWTCP maintains low queue length, therefore, it has enough room to converge without causing extra latency or packet drop. In addition, DWTCP achieves similar goodput...
9,000 | 9,200 | 9,400 | 9,600

(a) Goodput

TCP | DCTCP | DWTCP

CDF (goodput)

0 | 0.5

Goodput (Mbps)

(b) Queue length (Packet)

TCP | DCTCP | DWTCP

Queue length

0 | 100 | 200

Time (s)

(c) Flow completion time

Mean | 90th Percentile | 99th Percentile

Figure 3: Early congestion feedback helps maintain (a) high goodput, (b) low queue size, and (c) small flow completion times. DWTCP uses congestion signal from low priority queues and thus adapts to network conditions much quickly and efficiently.

as TCP NewReno and DCTCP, even though it does not rely on queue occupancy for high throughput (Figure 3a).

2.3 Design Space

A key design question is how to get feedback from the low priority queues and what congestion signals can be inferred. In this work, we propose using Scout packets to observe the state of the low priority queue, in terms of bandwidth availability and link busyness. Scout shares similarities with the probing-based schemes that observe the link state \[8,9,16,15,26\]. However, it is fundamentally different from these approaches as it uses low priority queues to probe the network. Using low priority allows Scout to operate without hurting the normal traffic, unlike existing schemes. Scout packets might cause a small delay for normal traffic if the later arrives while serving the former. This is reduced by using small packet size (64 bytes), and can be reduced further by using packet preemption (IEEE 802.1Qbu).

An orthogonal design choice could be to send actual data packets in the low priority queue as Scout. Such an approach would be useful for protocols that open multiple communication channels such as MPTCP [18], Stream Control Transmission Protocol (SCTP) [31] and HTTP 1.1 [17]. However, for single channel protocols, it may create issues with the out-of-order delivery and time-outs. We leave the evaluation of such an approach as future work.

Another design choice could be to measure the bandwidth availability and link busyness at the switch and share such information with the end-hosts using a mechanism similar to INT. However, such a design requires changes to the switch, which is orthogonal to our design objectives.

Why not ECN with Small Threshold? A simple design choice could be to use DCTCP with a very small marking threshold. However, as shown in the original DCTCP paper, small marking thresholds results in severe underutilization for the long flows [5]. To further understand this, we repeat the experiment in § 2.2 and set marking threshold (K) to seven packets (equivalent to the buffer occupancy of DWTCP). we note that although setting a smaller threshold for DCTCP improves the completion time of the short flows (see Figure 3c), it is still worse than DWTCP which takes advantage of early and amplified congestion signals and is more aggressive in its control laws. The long flows also experience under-utilization with DCTCP in this scenario. We observe a similar trend in the testbed evaluation (See § 6).

3 Scout

3.1 Overview

Scout is a probing-based congestion signaling mechanism that has three main components as depicted in Figure 4.

Scout Generator. A Scout sender sends very small (64 bytes) low priority Scout packets alongside the normal TCP traffic to observe the network state. Scout packets can be sent using either a parallel UDP socket, or on the same TCP socket after setting priority as a flag to
distinguish Scout traffic. Using UDP socket to send Scout requires updating hashing at the switches to make sure the Scout traffic takes the same path as the data packets. However, using the same TCP socket allows Scout packets to be served through the same datapath in case of using ECMP.

Scout Reflector. Scout reflector acts as a server that echoes back the ACKs for Scout packets. It can be implemented using IPTABLEs without requiring any changes in the TCP stack.

Scout Analyzer. Scout analyzer is the brain of Scout that has two main responsibilities. First, it analyzes the Scout packets and the Scout ACKs to extract congestion signals. Second, it interacts with the congestion control module to react based on the observed network state.

Scout service does not require any changes in the switches. It can be implemented in commodity switches using a strict priority queue. At the switch side, normal traffic is served with strict high priority, while Scout traffic is served with the low priority. Because Scout packets are sent with low priority, switches drop them in case of contention, which guarantees that the Scout incurs very little overhead in the network (less than 0.1% in our experiments). Moreover, Scout does not require switches to reserve too much buffer capacity for low priority queue. Even using 600 bytes (5-10, 64-byte Scout packets) for the low priority queue provides the best performance results.

3.2 Scout for Congestion Signaling

Scout provides three key benefits for the congestion control design. First, it helps in detecting available bandwidth in the network. Second, it helps in determining the amount of congestion (in the form of packet loss and delay). Third, it provides this information earlier than existing mechanisms.

Bandwidth Availability. Scout traffic, at the switches, can only fill in the gaps between normal traffic packets (See Figure 1). Once an ACK for the Scout packet of size $S$ is received, it provides an implicit guarantee from the network that all switches through the datapath can allocate at least $S$ bytes for that flow. We call such a signal “Bandwidth Availability”. The transport protocol can increase its window size faster, proportional to $S$, to utilize any available bandwidth. Note that, Scout is not trying to estimate the exact value of the available bandwidth, rather it tries to get implicit permission from the network to use a certain limit of the extra bandwidth through the datapath.

Figure 5: Relation between the Scout delay and link utilization. Ratio of RTT to Scout delay ($\tau/d_s$) provides a very clear indication of the congestion in the network.

Link Busyness. Link busyness indicates the amount of time that the switch is continuously busy servicing normal traffic. Increase in link busyness, i.e. smaller gaps between normal packets, is an early indication of congestion. Scout enables an end-host to detect link busyness using the delay and loss of Scout packets in the network. For example, delay ($d_s$) in a Scout packet (and corresponding ACK) can be used as an early indication of congestion. Similarly, loss of a Scout packet (or ACK) indicates severe congestion and the potential of a packet loss in the network. The sender can use these signals to reduce congestion window and prevent queue buildups and packet drops in the network.

Early Congestion Detection and Signal Amplification. Scout can detect link busyness several Round-Trip times (RTTs) (denoted by $\tau$ here) before the link reaches full utilization, which is very fundamental in designing congestion control algorithms. Figure 2a shows that the delay ($d_s$) encountered by Scout packets gets several times higher than the delay of normal traffic (5-15x) as the load increases. Furthermore, Figure 5 depicts the relation between Scout delay ($d_s$) and link utilization. For a fixed Scout packet size, Scout throughput is inversely proportional to Scout delay. As depicted with the black line, the normalized inverse Scout delay ($\tau/d_s$) provides a clear indication of congestion as the link utilization approaches 100%. Similarly, Figure 2b shows the time difference in how quickly Scout can detect the packet drop based on the aggressiveness (increase in rate). A Scout loss predicts link is full several (tens to hundreds of) RTTs earlier than similar signals in current loss-based or delay-based schemes.

3.3 Scout Optimizations

In this work, we propose the following optimizations to improve congestion signaling timeliness and accuracy, to minimize overhead and switch buffer requirements, and to increase scalability.
Scout Packets Injection Rate ($x$). The rate at which Scout packets are injected directly affects the timescale at which congestion can be detected in the network and the overhead due to Scout traffic. Too low Scout rate (one packet every few RTTs) can minimize the overhead, but it can also severely impact the timeliness of getting congestion state of the network, and ability to estimate the available bandwidth. Too high Scout rate (on sub-RTT level) can cause significant overhead, while not providing significant benefits for congestion detection. We propose to send one Scout packet, per flow, every base RTT of the network as a baseline. This also helps with requiring smaller buffers to hold Scout packets in the low priority queues. We show the impact of different injection rates in the evaluation.

Dynamic Scout Coefficient for Bandwidth Availability Estimate. Sending one Scout per RTT limits the ability to detect available bandwidth in the network. Instead of using large Scout packet size, or varying the Scout injection rate, we use a dynamic Scout coefficient $\alpha$. We send a small Scout packet (of 64 bytes) every RTT, yet dynamically scale $\alpha$ to adjust bandwidth estimate associated with an incoming Scout packet. We double the value of $\alpha$ upon receiving a Scout ACK; and reduce the $\alpha$ by half upon observing a Scout packet loss. The congestion window of the flow is increased according to the Scout coefficient, instead of the Scout packet size, upon receiving a Scout ACK. This provides significant improvement in convergence rate and helps adapt to network conditions in case of congestion.

No Scout for Flows Smaller than Initial Congestion Window Size. We do not use Scout for a flow if its size is smaller than the initial congestion window size. The intuition here is that if a flow is not going to last for more than one RTT, then getting network state feedback for such a flow is not useful. Despite not using Scout for such short flows, we can still see improvements in highly skewed workloads like datamining[19] since Scout can keep queue occupancies low and therefore short flows can complete faster. Moreover, it significantly, reduces the CPU and network overheads, due to Scout packets, for such workloads.

Per-datapath Scout. To further reduce Scout traffic and increase scalability, Scout packets can be sent between each pair of hosts (per-datapath), instead of per-flow, and the received congestion signal can be distributed among all active flows between each pair. For example, bandwidth availability signal can be distributed equally among all active flows, and link busyness can be conveyed to all active flows. Our testbed evaluation shows that this strategy does not affect the performance, yet reduces the Scout overhead significantly. We note that, our main design and evaluation does not assume this optimization.

3.4 Scout Overheads Comparison

Scout packets do not incur significant overheads in terms of CPU utilization at the end-host and/or extra traffic (Scout) in the network. To understand the overheads, let us assume that we are sending $x$ Scout packets per RTT for each connection.

For the CPU overhead, this setting will require generating $x \times 100k$ Scout packets per second per flow for a data-center with $10\mu$s RTT (as in a 100G datacenter network). If the end-host has 100 flows, then it will need to generate $\approx x \times 10$ million Scout per second, which can be generated using one core in commodity servers. Similarly, as for the network overhead, for each Scout packet of 64 bytes, the host also sends a number of packet equal to its congestion window (CWND). For a typical MTU setting of 1500 bytes, this means $CWND \times 1500$ bytes. Hence, the worst-case scenario can be calculated when $CWND = 1$ to be $64/1500 = 4.2\%$.

In practice, however, we do not observe this extreme case overhead. For example, Figure 3a shows that the DWTCP long flow has less than 0.2% overhead in terms of goodput. This is for the following reasons: First, the Scout packets are served only when there is available bandwidth in the network. The LPQ Scout packets lose to the HPQ dat packets when the link is highly utilized. Second, the average $CWND$ is much larger than 1. Lastly, the per-path implementation of the Scout scales down the overhead by the multiplexing factor per connection (e.g., in the worst case, for a load of 100 flows per server, and an average multiplexing of 10 flows per path, this leads to at most 0.5% overhead due to the Scout packets).

On the other hand, the overhead of existing schemes such as INT for a usual 5-hop topology is reported to be around 6.8% [11]. The most recent probabilistic version of the INT telemetry method [11] tries to minimize the per packet overhead by probabilistically distributing the congestion signal over the packets of a single flow. This potentially reduces the per packet overhead to be as small as 2 bytes which makes the overall overhead smaller than 0.2%. However, it should be noted that due to the distributed nature of the signal over the packets, only large flows can benefit from the PINT signal and shorter flows still encounter the normal INT overhead [11].

4 DWTCP

4.1 Overview

DWTCP uses the Scout bandwidth availability, Scout delay and Scout loss signals to control the
congestion window. At its base, DWTCP uses TCP as a transmission control mechanism.

At high level, the interaction between Scout and TCP algorithm is as follows:

- When a TCP connection is established, the Scout connection also starts sending small Scout packets periodically from the sender side towards the receiver and waits for their ACKs.
- Scout packet ACKs, from the Scout receiver, work as an indication of bandwidth availability. DWTCP increases CWND by a factor $\alpha$ of the number of received Scouts: $w \leftarrow w + \alpha \cdot S$, as shown in Figure 6 where $S$ is the Scout packet size and $\alpha$ is the Scout window size as described in §3.3.
- In addition, DWTCP also increases the congestion window by $L$ upon receiving a data packet ACK, similar to TCP.
- Scout delay $d_s$ is used as an indication of link busy- ness. If $d_s$ is larger than a target delay $d_t$ (as the maximum delay after which a Scout packet is considered lost in the network; more on this coming up), DWTCP connection makes a non-aggressive rate reduction decision. Such an early signal allows DWTCP to react quickly to congestion and maintain lower queue length as shown in Figure 6.

The basic signal in DWTCP is the Scout delay, which helps the source decide when to increase or decrease congestion window. Unlike delay-based congestion control algorithms [22], Scout does not need special hardware at the host to measure accurate time stamps for $d_s$ calculation.

### 4.2 DWTCP Control Algorithm

**DWTCP Control Laws.** DWTCP uses the following control laws to manage the congestion window (CWND) of its flows.

$$w \leftarrow \begin{cases} w + L + \alpha S & \text{if } d_s \leq d_t \\ w \cdot (1 - \beta \sqrt{\frac{d_s - d_t}{d_s}}) & \text{Otherwise} \end{cases}$$

(1)

Here, $d_s$ is the Scout delay, $d_t$ is the target Scout delay, $w$ is the current congestion window, $L$ is the data packet size, $S$ is the Scout packet size, $\beta$ is a constant used to limit the maximum window reduction to 1/2 (similar to TCP) and $\alpha$ is the dynamic Scout coefficient that controls the aggressiveness of DWTCP (as described in §3.3). In addition, similar to [22], we multiply the decreasing signal by the square root of the current congestion window. Such an approach enhances fairness between flows as it forces the flows with larger $w$ to reduce their congestion window faster.

**DWTCP Control Parameters.** DWTCP performance is controlled by the target Scout delay $d_t$, Scout coefficient $\alpha$, and the constant $\beta$.

First, $d_t$ along with the $d_s$ helps decide when to switch between the increasing phase and the decreasing phase. Second, their ratio $\left(\frac{d_s - d_t}{d_s}\right)$ is used to dynamically decrease the congestion window proportional to the network state. Higher the delay, higher the decrement value is.

In DWTCP, we define Scout delay $d_s$ as follows:

$$d_s = \max(scout.sendTime - scout.rcvTime, \text{Now} - \text{unackedScout.sendTime})$$

where $\text{scoutSendTime}$ and $\text{scoutRecvTime}$ are the transmission and receiving timestamps of the last Scout respectively. The intuition here is to use the Scout throughput as an indication of link busy-ness. In other words, if the queue is busy serving normal traffic and the Scout packets are congested, it is a clear indication that flows must reduce their congestion window.

DWTCP uses target delay $d_t$ as the maximum delay after which a Scout packet is considered lost in the network. $d_t$ depends on the network RTT ($\tau$) and the link capacity. DWTCP sets $d_t$ as:

$$d_t \leftarrow \tau + K \cdot \frac{L}{C},$$

(2)

where $L$ is the packet length, $C$ is the line rate, and $K$ indicates how long the high priority queue was busy before allowing Scout to be served in the low priority queue in data packet time unit. In our experiments, we set $K$ to 100. The intuition here is that by allowing the switch to serve
one Scout packet every $K$ data packets, we limit the link utilization to $K/(K + S) \geq 99\%$.

We suggest using $\beta \approx \frac{1}{2\sqrt{BDP/N}}$, where $BDP$ is the bandwidth-delay product of the network and $N$ is an estimation of the average flow number. However, such an estimation does not need to be accurate as shown by the stability condition deduced in Appendix [A].

**DWTCP Stability Analysis.** We mathematically analyze the stability of DWTCP and demonstrate that (proof in Appendix [A]):

**Theorem 1.** DWTCP is spiral stable and has one stable point (at $Q = 0, w = BDP/N$) as long as the number of processed packets in the high priority queue before serving any Scout $(k)$ is less than $2BDP/\sqrt{\beta}$.

**DWTCP Algorithm.** The working mechanism of DWTCP is depicted in Algorithm 1. We start by initializing some common variables in lines 1 - 5. The increasing phase, defined by AckRcvd function, is triggered when data or Scout acknowledgments are received (Lines 8 - 15). In this function, $l$ and $s$ are initialized to $L$ and $S$ respectively depending on the acknowledge type (Lines 9 and 10). In addition, the CWND is to be increased according to TCP state as indicated by lines 12 and 14 for the slow start and congestion avoidance states, respectively. The decreasing phase, defined by ScoutDelayed function, is triggered by not receiving Scout for time larger than $dt$. Thereafter, we calculate the new congestion window based on Equation (5) as shown in Line 22. We also define a new variable called DET to limit the congestion window reduction to once per RTT (see condition in line 20).

**Using Scout Inter-packet Time to Signal Link Busy-ness.** Similar to Scout delay, we can use Scout inter-packet gap ($ipg_s$) to enhance the stability and fairness of the system. Hence, we can modify Equation (1) as follows:

$$w \left\{ \begin{array}{ll}
 w \cdot (1 - 2\beta \sqrt{w} \frac{(d_s - d_t)(ipg_s - ipg_t)}{d_s \cdot ipg_s}) & \text{if} (d_s > d_t) \& (ipg_s > ipg_t) \\
 w \cdot (1 - 2\beta \sqrt{w} \frac{(ipg_s - ipg_t)^2}{ipg_s^2}) & \text{if} (ipg_s > ipg_t) \\
 w \cdot (1 - 2\beta \sqrt{w} \frac{(d_s - d_t)^2}{d_s^2}) & \text{if} (d_s > d_t) \\
 w + L + \alpha S & \text{otherwise}
\end{array} \right.$$  

where $ipg_s$ is the target inter-packet gap which is calculated by dividing $d_t$ by Scout window size. Because congestion window reduction is limited by 1/2, we need to accommodate for the square of such a metric by multiplying by 2 to limit the new metric withing the same range. Our simulation and testbed tests lead to similar results.

**Algorithm 1: CWND management using Scout**

**Data:** Data ACK, Scout ACK, Scout delay ($d_s$), $\alpha$, $\beta$

**Result:** New Congestion window $w$

1. $w \leftarrow w_{init}$
2. $T = \text{physical hardware RTT}$
3. $d_t \leftarrow \tau + K \cdot \frac{d_t}{\tau}$
4. $DET = \text{Now}()$ // Decrease Execution Time
5. $\beta = 1/16$
6. $\alpha = 1$

**Function AckRcvd($w$, ACK.TYPE):**
   /* Increasing phase */
   l = (ACK.TYPE == Data)? L : 0
   s = (ACK.TYPE == Scout)? S : 0
   else if TCP.state == Slow start then
   $w \leftarrow 2 \cdot w + \alpha \cdot s$;
   else
   $w \leftarrow w + l + \alpha \cdot s$;
   return $w$

**Function ScoutDelayed($w$, scout.sendTime, scout.rcvTime, unackedScout.sendTime):**

1. $d_s = \max(scout.sendTime - scout.rcvTime, \text{Now} - \text{unackedScout.sendTime})$
2. if $d_s > d_t$, then
   if $\text{Now}() \geq DET + T$ then
   /* Decreasing phase */
   TCP.ssthresh = $w$
   $w \leftarrow w \cdot \max((1 - \beta \sqrt{w} \frac{(d_s - d_t)}{d_s}), 0.5)$
3. $DET = \text{Now}()$
   return $w$

**5 Evaluation**

We evaluate DWTCP using a combination of small scale real testbed experiments and large scale NS3 [3] simulations. We take a closer look at DWTCP performance in the following aspects.

**How does DWTCP perform in practice?** We use a simple experiment with the combination of short and long flows to demonstrate that Scout can help DWTCP achieve: i) near-zero queues, ii) small flow completion times (improvement by up to 2x compared to DCTCP, 4x compared to TCP), and iii) high throughput comparable to existing protocols (1 - 3% reduction) [5,7]. Maintaining high throughput with near-zero queues is a challenging task for TCP al-
algorithms. DWTCP, despite the Scout overhead and near-zero queues, is able to achieve similar throughput as TCP and DCTCP. We observe similar behaviors in testbed and NS3 simulations.

**How does DWTCP perform in large scale datacenters?** Using large scale NS3 simulations, we demonstrate that DWTCP, compared to DCTCP [5] and HPCC [11], improves latency by up to 2x at the mean and up to 3.5x at the tail with the websearch workload [5] and up to 1.5x at the mean and up to 3x at the tail with the datamining workload. HPCC performs better (up to 31%) than DWTCP at low loads.

**What is the impact of individual parameters on DWTCP?** We demonstrate that DWTCP is not very sensitive to control parameters (α and β), and it can work across a range of scenarios such as varying number of senders and long haul networks.

### 5.1 Simulation Setup

We start with NS3 [3] simulations for evaluating DWTCP. We present testbed experiments in the following section.

**Topology.** We consider two different topologies: a 144-node real data center topology and a single bottleneck topology. For the leaf-spine topology, edge links are 10 Gbps, core links are 40 Gbps, RTT is 100 µs and switch buffer size is set to 250 packets. Also, we consider a single bottleneck topology with RTT of 100 µs to keep it consistent with our real testbed setup.

**Traffic Patterns.** We generate traffic using the websearch [5] and datamining [19] workloads. All-to-all traffic pattern with poisson arrival process and an average load of 20% — 80% of link capacity is used.

**Comparisons and Metrics.** We evaluate and compare DWTCP to DCTCP, TCP NewReno (noted TCP in the figures), HPCC-PINT (noted HPCC in the figures). We report flow completion times (slowdown), fairness (Jains Index [20]), throughput and queue size. Using the publicly available NS3 simulator for HPCC, we also evaluate Timely [27] and DCQCN [32], and our results confirm what shown in [23], that is, HPCC significantly outperforms both Timely and DCQCN. As a result, in the interest of space, we only include the results of HPCC in our evaluation figures. While the implementation of SWIFT [22] is not publicly available for us to experimentally compare with DWTCP, we believe the amplification in delay and early notification through Scout can lead to better or comparable results without requiring SWIFT’s NIC hardware timestamps.

**DWTCP Settings.** For DWTCP, we set the low priority queue size in switches as 10 Scout packets (≈ 600 bytes). DWTCP sends 1 Scout packet of size 64B each RTT, unless specified otherwise. It uses β of 0.001, K = 100 and initial value of α = 20 (equivalent of 1MTU). We also evaluate the impact of varying these parameters in §5.4.

### 5.2 Performance Overview

Figure 3 shows a glimpse of DWTCP in maintaining small queues, achieving fast convergence, high throughput and small flow completion times. To highlight the performance of DWTCP in more dynamic scenarios, we consider a single bottleneck topology and generate five flows between different senders and receivers. Flows start 1s apart, and once we have five active flows, they leave 1s apart.

**DWTCP is Stable and Achieves Fairness.** Figure 4 shows that DWTCP is very stable in maintaining small (near-zero) queues (Figure 7a) and achieves high fairness (Figure 7b) compared to TCP (NewReno), and DCTCP. DWTCP reacts quickly to congestion and adapts its rate dynamically to maintain stable queue sizes. DCTCP is also very stable in maintaining queue sizes. However, it maintains queue levels around the threshold (65 in this case),
which can impact the latency of short flows. This stable behavior of DWTCP helps achieve high fairness among the flows (Figure 7b). As shown, the fairness index for DWTCP is very close to 1 (i.e., optimal fairness index). When a new flow arrives (or leaves), other flows quickly give up (or utilize) the link capacity.

**DWTCP Reacts Rapidly to Congestion.** Figures 7c and 7d depicts the throughput of TCP, DCTCP and DWTCP respectively. We notice that DWTCP reacts quickly to the congestion from the new flows and converges to fair share without any peak oscillations, unlike other protocols. Furthermore, due to dynamic bandwidth availability estimation, it quickly uses up any spare bandwidth when the flows leave. Note that, although TCP and DCTCP maintain higher queues, DWTCP is able to maintain similar throughput even with near-zero queues as shown in Figure 7d due to its fast convergence properties.

### 5.3 Benchmark Workloads

Next, we evaluate DWTCP in more realistic datacenter settings using all-to-all traffic pattern for the websearch and datamining workloads in a typical leaf-spine topology and compare with DCTCP and HPCC.

#### Datamining Workload

Figures 8a, 8b, 8c, and 8d show the percentile, tail slowdown across small flows (0, 100KB], medium flows (100KB, 1MB], and large flows (1MB, inf], respectively for the datamining workload. DWTCP outperforms DCTCP in both the average and 99th percentile slowdown. It improves the mean by up to 97%, and tail up to 3x. Moreover, DWTCP improves the tail slowdown by 1.5x-2.8x for the small flows, up to 2x-3.2x for the medium flows, and up to 3x-100x for the large flows. DWTCP improves the tail slowdown by up to 100x as it does not experience any timeouts at higher loads. HPCC performs better (up to 31%) at low loads due to flows starting at the line rate and faster convergence. However, DWTCP performs significantly better at higher loads and improves mean and tail slowdowns by up to 2x. DWTCP maintains smaller queues, even at very high loads, as a result the tail completion times of the short flows improve by up to 1.7x, whereas the medium and long flows achieve similar slowdown as in HPCC. Note that, more than 80% of the flows in datamining workload belong to small flows.

**Websearch Workload.** We observe similar performance for the websearch workload. Figure 14 (see Appendix B) shows that compared to DCTCP (and HPCC), DWTCP improves up to 2x (3x) mean slowdown and up to 3.5x (5x) for the tail slowdown. (b-d) DWTCP significantly improves tail slowdown for small by 8x (3.7x), medium by 3.5x (10x) and large flows by 3.5x (15x). Note that, compared to HPCC DWTCP performs better for both the short and long flows at high loads.

### 5.4 DWTCP Deep-dive

In this section, we study the impact of DWTCP parameters and varying network conditions, using a dumbbell topology.

**Effect of Scout Coefficient (α).** In this setup, we generate two long-lived flows. Flow 1 starts at time 0, and Flow 2 starts at time 0.5 seconds. We compare the dynamic Scout coefficient α with the fixed static value of 10. Figure 9a shows that dynamic α helps converge faster and improves fairness compared to a static α.

**Effect of Scout Injection Rate (x).** In the above experiment, for the static α, we vary the Scout injection rate instead of changing the Scout coefficient. This mimics having a sub-RTT Scout injection and also the static α.
Scout convergence without requiring larger flows, DWTCP can achieve high throughput and faster convergence. Therefore, the optimization of injecting one Scout per RTT provides a very good guarantee that as we increase the number of flows, DWTCP can achieve high throughput and faster convergence without requiring larger Scout buffers in the switches.

**Effect of β.** We repeat the above experiment settings with dynamic Scout window and varying β. β factor controls the aggressiveness of the decrease, and setting it correctly greatly improves the convergence and fairness. Figure 9c shows that using a very aggressive β results in severe underutilization. Similarly, setting it too small can increase the convergence time as the flows take longer to free up the bandwidth. This also results in larger queue sizes.

**Effect of Varying Number of Senders.** In this experiment, we increase the number of senders (receivers) to 10, 20, and 30 using default DWTCP parameters. We can verify that DWTCP can adapt to different number of flows and can maintain smaller queues (Figure 15b), achieve higher throughput and fairness (see Appendix B, Figure 15). We observe an increase in the queue size in the beginning, which is the result of synchronization among the flows.

**Effect of Varying Network Conditions.** To observe the adaptability of Scout to varying network conditions, we vary the link bandwidth every 0.1s as [0.4, 0.7, 1, 10, 0.8] Gbps. We run five long flows to observe the queue size, throughput, and fairness. We observe that DWTCP quickly grabs any available bandwidth when the available bandwidth changes from 1 Gbps to 10 Gbps and it quickly converges to full link utilization, maintains small queues and achieves fairness. See Appendix B, Figure 15 for more details.

6 Testbed based Evaluation

6.1 Implementation

The Linux implementation of DWTCP consists of three main components:

**Scout Generator.** In order to inject the Scout packets, we initiate a User Datagram Protocol (UDP) stream from each sender towards its destination using the default installed packet generator on Linux (pktgen V2.75). This packet generator is able to generate packets with inter-departure times as small as 7µs. Even though it sometimes experiences some jitters (with delays as large as 3µs), this accuracy has shown to be enough for the range of Round Trip Time (RTT) that we obtain in our testbed (80µs). Before sending each packet, in addition to marking the IP packets with low priority DSCP values, it also fills the packet payload with their send time (note that there is a gap between the marking time and the time that the packet is being placed on the wire). In our setup we use microsecond clocks and a per datapath stream of Scout packets rather than a per flow connection for the testbed experiments. Later, we show the difference of having a single stream of Scout packets per datapath with the case that they are being initiated per flow.

**Scout Reflector.** The reflector is a set of simple iptables rules installed on the hosts which result in forwarding the incoming Scout packets to their origin. This design in combination with using pktgen allows us to implement a prototype without touching the kernel code and also without compromising the performance.

**Congestion Algorithm.** We implemented the DWTCP congestion algorithm in Linux (V5.4.0) as a loadable kernel module. The Scout analyzer is currently embedded in this module via an added Netfilter hook that listens for UDP packets. Upon receiving any relevant packet, we decode its sending timestamp to perform our delay calculation.
6.2 Testbed Description

Topology Setup. We use a dumbbell topology in our testbed. There are 5 servers connected to each side of this topology (i.e., 10 servers in total) with an access link of 10Gbps each. The bottleneck link is limited to 10Gbps and 300KB switch buffers. Each server is configured with a 64-core Intel(R) Xeon(R) Gold 5218 CPU 2.30GHz, and two arrays of 32GB DRAM, and for the internal queueing discipline they use Linux’s `pfifo_qdisc` (packet FIFO queueing discipline). The bottleneck endpoint switches are set to use a strict priority for routing the traffic.

DWTCP Settings. We impose priorities by setting the IPTOS field of each Scout packet to a lower priority than the normal TCP traffic. We use a fixed Scout window size per datapath. Each UDP Scout packet is of size 64B, and they leave their source every 7µs. The lower priority queue of the switch was also set to 640B (i.e., 10 packets). Similar to simulation setup, we set β to 0.001, and K to 100.

Performance Metrics. We use `CWND` and `RTT` goodput of long flows and flow completion times (FCT) of the short flows as performance metrics. We assume that there is enough traffic being generated to congest the bottleneck (unless stated otherwise). Throughout each experiment, we periodically poll (average over buckets of 20ms) all the connection metrics of interest using a dedicated core on each machine.

6.3 Performance Overview

Steady State. In this experiment we use five hosts as the senders, each with two flows that start simultaneously (i.e., 10 flows). For DCTCP we consider an additional setting than the originally suggested parameters for its ECN marking thresholds. In this setting, we set the ECN threshold marking to be 7 packets rather than the suggested 65 packets for 10Gbps bottlenecks. This threshold is equivalent with DWTCP’s queue sensitivity parameter K which is set to 7 packets. We observe significant underutilization (around 15%) in this case as one could expect from DCTCP’s dynamic. Figure 10 compares the RTT and CWND of one of these flows from each scheme. One can see that DWTCP achieves minimum RTT and zero queue loss while maintaining link utilization close to the maximum. This is while DCTCP has 0.01% packet retransmission on average in the steady case.

Synchronized Flows. In order to measure the responsiveness of DWTCP in sudden network changes, we initiate three batches of five simultaneous flows with the contribution of one flow from each sender in a batch. Each batch starts and stops at chunks of five seconds. The small amplitude of the changes in the CWND and RTT (Figure 11a) for the flows in DWTCP compared to DCTCP explains the near-perfect fairness that DWTCP achieves. Figure 11b shows that while DWTCP maintains 2 – 3X faster RTTs, it also reaches close to the maximum throughput as depicted by the goodput CDF in Figure 11c. The responsiveness power of Scout becomes more apparent by investigating the reason for the throughput loss in DCTCP. Figure 11c depicts that DWTCP maintains on average higher goodput compared to DCTCP because DWTCP doesn’t experience packet retransmission. In addition, DWTCP keeps the link utilization below the maximum link capacity. Such behavior is intended by design to enable DWTCP’s smooth transition during sudden changes.

Flow Completion Times. In this experiment, we measure the impact of bursts. We consider two background flows from two different senders to one client. And six other flows send sudden bursts of fixed size towards the same client. We repeat bursts every 500 milliseconds and vary their size from 0.5MB to 2MB per connection. Figure 12 shows the average flow completion times of these bursts in comparison with the goodput of the background flows. We observe that DWTCP consistently obtains higher goodput than DCTCP at different burst sizes, while achieving smaller latency of 81µs (163µs for DCTCP) In addition, DWTCP improved the FCT of short flows by at least 15%.

7 Related Work and Discussion

Comparison with Delay-based CC. Delay based mechanisms, such as Timely [27] or SWIFT [22], experience stability issues or need to apply moving average to mitigate noise effect while calculating the delay at the host. On the other hand, the arrival of ScoutACK is a
Figure 11: Long flows joining (and leaving) in sequence. DWTCP maintains small queues and achieves high fairness. Besides, DWTCP converges faster, maintains fairness, and gives away bandwidth to new flows quickly.

Figure 12: FCT of short flows and goodput of long flows.

clear indication of bandwidth availability. has a very robust signal for bandwidth availability as the arrival of the Scout. Moreover, Scout reacts based on the delay or loss of the Scout traffic in low priority queue, which means it has a much faster response time in detecting congestion in the high priority queue. As a result, Scout is more stable and does not require hardware timers (or averaging) to compute delays, as in SWIFT.

Comparison with Credit-based CC. Credit-based congestion control algorithms often provide better performance than traditional congestion signals [14, 24]. However, hop-by-hop credit-based mechanisms (such as Infiniband [24]) are expensive, difficult to manage, and have a high overhead. Some well known challenges of these mechanisms are: i) Interaction with congestion control protocol and ii) Deadlock issue. Similarly, path-based credit-based mechanisms need switch modifications, or reserve a fraction of bandwidth to manage and throttle credits [14]. Compared to these mechanisms, Scout is easier to implement, does not require any switch modifications, and only requires small buffers (less than 1MTU) for the Scout traffic.

Comparison with Out-of-band OAM. Prior art on out-of-band OAM, such as PathChirp [15] and BART [16], aim to estimate available bandwidth per data path. However, these mechanisms rely on sending probe packets with the same priority as the normal traffic, which incurs significant overhead, and can impact the ongoing network traffic. Moreover, for delay and available bandwidth measurements, such approaches are very noisy and depend on the state of the main data queue. On the other hand, Scout uses low priority traffic which incurs negligible overhead, does not impact the main traffic, and provides a much amplified and stable signal.

Interaction with Multipath Routing. Scout packets need to have the same hash value as the normal traffic to follow the same path, when ECMP [4] is enabled. To achieve this, in our current implementation, we modify the hashing algorithm at the switch based on the ToS field. An alternative approach is to use a zero payload packet, from the same connection, or add an extra flag to the TCP header to denote Scout packet.

Comparison with Flow Scheduling-based Schemes. In the recent past, many scheduling based transport schemes have been proposed for data center networks [6, 23, 29, 10, 30, 28]. These schemes aim at minimizing the FCTs of latency sensitive flows and achieve near optimal performance. DWTCP is not a scheduling-based scheme, however, it maintains very small queues at the switches, which is very useful for the latency sensitive flows. In this paper, we demonstrate that DWTCP provides significant improvement, compared to DCTCP [5], in terms of FCTs. It will be interesting to evaluate how it compares to deployment friendly scheduling-based schemes [30, 29, 10]. We leave this as a future work.

8 Conclusion and future work

In this paper, we presented Scout service, an early indicator of congestion and bandwidth availability based on sending probe traffic with low priority. We also presented DWTCP as a Scout-based congestion control mechanism. Our extensive simulations and testbed experiments show that DWTCP outperforms TCP NewReno, and DCTCP in
maintaining close to zero queue length, and reduces network latency to near optimal. In addition, DWTCP outperforms HPCC for high load scenarios and achieves comparable results at low loads without the need for switch modifications. The experimental results also show significant reductions in flow completion times (mean and tail) compared to DCTCP. We have mathematically proved that DWTCP is stable (and the stability conditions are deduced), and has one stable equilibrium point.

The Scout has a very nice property that as it provides the signal several RTTs earlier, it makes a congestion control protocol less sensitive to the timestamping delays and therefore it eliminates the need for hardware timestamping even for small values of RTTs. Even though our experiments show Scout has reasonable overhead, one can imagine the overhead can be further reduced by using a data packet, or by carrying some data over Scout probes. Furthermore, one can envision enriching other congestion control protocols using Scout service. We leave these as future work.

References

[1] In-band network telemetry in barefoot tofino. https://tinyurl.com/2p8jya2a, 2021.
[2] In-band network telemetry in broadcom tomahawk 3. https://tinyurl.com/2p8kyddf, 2021.
[3] Network simulator 3. https://www.nsnam.org/, 2021.
[4] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In Proceedings of the 2014 ACM conference on SIGCOMM, pages 503–514, 2014.
[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In Proceedings of the ACM SIGCOMM 2010 Conference, pages 63–74, 2010.
[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Pfabric: Minimal near-optimal datacenter transport. ACM SIGCOMM Computer Communication Review, 43(4):435–446, 2013.
[7] M. Allman, V. Paxson, W. Stevens, et al. Tcp congestion control. 1999.
[8] T. E. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan. Pcep: Efficient endpoint congestion control. In NSDI, 2006.
[9] M. M. Bahnasy, A. Beliveau, B. Alleyne, B. Boughzala, C. Padala, K. Idoudi, and H. Elbiaze. Using ethernet commodity switches to build a switch fabric in routers. In 2015 24th International Conference on Computer Communication and Networks (ICCCN), pages 1–8, 2015.
[10] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), pages 455–468, 2015.
[11] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. Pint: Probabilistic in-band network telemetry. In Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 662–680, 2020.
[12] L. S. Brakmo, S. W. O’Malley, and L. L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. SIGCOMM Comput. Commun. Rev., 24(4):24–35, Oct. 1994.
[13] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. Queue, 14(5):20–53, Oct. 2016.
[14] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pages 239–252, 2017.
[15] L. Cottrell. Patchchirp: Efficient available bandwidth estimation for network paths. 4 2003.
[16] S. Ekelin, M. Nilsson, E. Hartikainen, A. Johnsson, J.-E. Mangs, B. Melander, and M. Bjorkman. Real-time measurement of end-to-end available bandwidth using kalman filtering. In 2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006, pages 73–84, 2006.
[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.
[18] A. Ford, C. Raiciu, M. Handley, S. Barre, J. Iyengar, et al. Architectural guidelines for multipath tcp development. IETF, Informational RFC, 6182:2070–1721, 2011.
To achieve low latency at the queues within the network, DWTCP controls the congestion windows of sources to keep the utilization of resources close to the maximum link capacity. Such behavior is fulfilled as follows:

\[
\begin{align*}
\frac{dw}{dt} &\leq \begin{cases} 
L + \bar{D} & d_s \leq d_t \\
\frac{L}{\tau} & d_s > d_t 
\end{cases} \\
\frac{dS}{dt} &\leq \begin{cases} 
L + \bar{D} & t \leq d_t \\
-\beta \frac{dS}{dt} & t > d_t 
\end{cases}
\end{align*}
\]

Using fluid model, we can formulate this equation as a differential equation as follows:

\[
\frac{dw}{dt} = \begin{cases} 
\frac{L + S}{\tau} & t \leq d_t \\
-\beta \frac{dS}{dt} & t > d_t 
\end{cases}
\]

\[d_s = \tau + k \frac{L}{C} \quad d_t = \tau + k \frac{L}{C} \tag{4}\]

Where \(\bar{k} > k\).
For the sake of simplicity, let’s redefine the decreasing part of the equation to become:

\[ w' = \frac{(L + S)/\tau}{w \cdot (d_s - d_t) / d_t}, \quad t \leq d_t \]

(5)

One can notice that we removed \( \sqrt{w} \) as it is originally used to enhance the fairness of the system. In addition, we replaced \( d_s \) in the denominator by \( d_t \) to simplify the equation. Because \( d_s > d_t \) in the decrease phase, using a lower value in the denominator forces the system to be more aggressive in the decrease. Hence, if the system is stable in such a state, it is expected to be also stable if the changes are less aggressive.

To study the stability of such a system we tackle the two phases separately. First, the decreasing phase defined as:

\[ w' = -\frac{w \cdot \beta (d_s - d_t)}{d_t} \]

By substituting \( d_s \) and \( d_t \) using (4), we get

\[
\begin{align*}
w' &= -w \cdot \beta \left( \frac{d_s - d_t}{d_t} \right) \\
&= -w \cdot \beta \left( \frac{\tau + \beta L \bar{k} - \tau - \beta L \bar{k}}{\tau + \beta L \bar{k}} \right) \\
&= -w \cdot \beta \frac{\beta L \bar{k}}{\tau \beta L \bar{k}} \\
&= -w \cdot \beta \frac{\bar{k} - k}{\tau + \frac{\beta L \bar{k}}{\tau}} \\
&= -w \cdot \beta \frac{\bar{k} - k}{\tau + \frac{\beta L \bar{k}}{\tau}} \\
&= \frac{w' + \beta \bar{k} - k}{\tau BDP + k}
\end{align*}
\]

Such an Equation expresses the relationship between the derivative of the congestion window \( w' \) and bandwidth-delay product \( BDP \) where \( BDP = \frac{\tau C}{\bar{k}} \).

For \( k = 0 \); i.e., \( d_t = \tau \) where \( \tau \) is RTT, the decreasing function becomes:

\[ w' = -w \cdot \beta \frac{\bar{k}}{BDP} \]

By adding the queue changing rate, one can summarize the \textbf{CWND} decreasing phase as follows:

\[
\begin{align*}
\text{CWND decrease} &= \frac{L + S}{\tau} \\
\text{queue} &= \frac{Nw}{\tau} - C
\end{align*}
\]

(6)

On the other hand, the \textbf{CWND} increase phase can also be summarized as follows:

\[
\begin{align*}
\text{CWND increase} &= \frac{L + S}{\tau} \\
\text{queue} &= \frac{Nw}{\tau} - C
\end{align*}
\]

(7)

Figure [13] depicts the phase trajectory of DWTCP dynamics after numerically solving Equations [6] and [7]. One can notice that such a system is a stable spiral system and, it has one stable equilibrium point at \( Q = 0, w = BDP/N \). For instance, if such a system starts at a state with high \( w \), it reacts by reducing \( w \) while increasing queue size for a while (depicted by the red path in Figure [13]). Thereafter, it surpasses the fair-share value of \( w \) to be able to reduce queue size. Finally, it recovers by increasing \( w \) again to reach the fair share.

![Figure 13: Phase space trajectory of the DWTCP. One can notice that DWTCP is a stable spiral system and, it has one stable equilibrium point at \( Q = 0, w = BDP/N \).](image)

**Theorem 2.** DWTCP is spiral stable and has one stable equilibrium point as long as the number of packet processed in the high priority queue before serving any Scout packets \( \bar{k} \) is less than \( \frac{2BDP}{\sqrt{\tau}} \).

**Proof.** Lyapunov has shown that an Ordinary Differential Equation (ODE) of a nonlinear dynamical system \( x' = f(x(t)) \) is said to be stable if all of its solutions start sufficiently close to the equilibrium point \( x_e \) and remains close to it as \( t \) increases [25]. I.e., \( x(t) \) is Lyapunov stable if for every \( \epsilon > 0 \), there exists a \( \delta > 0 \) such that, if \( \|x(0) - x_e\| < \delta \), then for every \( t \geq 0 \) we have \( \|x(t) - x_e\| < \epsilon \). More strongly, for a Lyapunov stable function \( x(t) \), it is said to be
asymptotically stable if \( \lim_{t \to \infty} ||x(t) - x_c|| = 0 \). To that end, we prove that all solutions of DWTCP ODE stay close to the equilibrium point.

For the decreasing phase depicted by Equation (6), we deduce the solutions of this differential equation by extracting the Jacobian matrix as follows:

\[
J = \begin{pmatrix}
\frac{\partial w'}{\partial w} & \frac{\partial w'}{\partial q} \\
\frac{\partial w}{\partial w} & \frac{\partial w}{\partial q}
\end{pmatrix} = \begin{pmatrix}
-\frac{\beta \cdot \bar{k}}{\tau} & -\frac{\beta \cdot w}{\tau} \\
\frac{\beta \cdot \bar{k}}{\tau} & \frac{\beta \cdot w}{\tau}
\end{pmatrix}
\tag{8}
\]

The eigenvalues of a \( 2 \times 2 \) matrix are the solutions of the determinant equation \( |\lambda I - j| = 0 \)

\[
\lambda + \frac{\beta \cdot \bar{k}}{\tau \cdot BDP} \quad \frac{\beta \cdot w}{\lambda} = 0
\tag{9}
\]

\[
\lambda^2 + \lambda - \frac{\beta \cdot \bar{k}}{\tau \cdot BDP} + \frac{\beta \cdot w \cdot N}{\tau^2 \cdot BDP} = 0
\tag{10}
\]

The roots of the previous equation can be calculated as follows:

\[
r_\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

\[
= \frac{-\frac{\beta \cdot \bar{k}}{\tau \cdot BDP} \pm \sqrt{\left(\frac{\beta \cdot \bar{k}}{\tau \cdot BDP}\right)^2 - 4\left(\frac{\beta \cdot w \cdot N}{\tau^2 \cdot BDP}\right)}}{2}
\tag{11}
\]

For a stable spiral system, the eigenvalues must be complex numbers with a negative real part. To satisfy such a requirement, Inequality (12) must hold to guarantee that the eigenvalues have a negative real part. In addition, Inequality (13) must hold to guarantee that eigenvalues are complex numbers.

\[
-\frac{\beta \cdot \bar{k}}{\tau \cdot BDP} < 0
\tag{12}
\]

\[
\frac{(\frac{\beta \cdot \bar{k}}{\tau \cdot BDP})^2 - 4\frac{\beta \cdot w \cdot N}{\tau^2 \cdot BDP}}{\beta^2 \cdot BDP - 4w \cdot N \cdot BDP} < 0
\tag{13}
\]

\[
\frac{\beta^2 \cdot BDP^2 - 4\frac{\beta \cdot w \cdot N}{\tau^2 \cdot BDP}}{\beta \cdot \bar{k}^2 - 4w \cdot N \cdot BDP} < 0
\]

At the equilibrium point, \( BDP = w \cdot N \). Therefore, we can reduce Equation (13) to become

\[
\beta \cdot \bar{k}^2 < 4BDP^2
\]

\[
\bar{k} < 2 \frac{BDP}{\sqrt{\beta}}
\tag{14}
\]

Hence, one can conclude that the DWTCP decreasing phase is stable as long as the number of processed packet in the high priority queue before serving any Scout \( \bar{k} \) is less than \( 2 \frac{BDP}{\sqrt{\beta}} \); i.e., Inequality (14) must be satisfied.

For the increasing phase stated in Equation (7), we can represent Jacobian matrix as follows:

\[
J = \begin{pmatrix}
0 & 0 \\
0 & \frac{\beta}{\tau}
\end{pmatrix}
\tag{15}
\]

The eigenvalues of this matrix are the solutions of the determinant equation \( |\lambda I - j| = 0 \).

\[
\lambda + \frac{\beta}{\tau \cdot BDP} = 0
\]

\[
\lambda^2 + \lambda = 0
\tag{16}
\]

\[
(\lambda = 0) \text{ and } (\lambda = \frac{N}{\tau})
\]

Thus, the increasing phase is not stable for DWTCP. However, the decreasing phase is triggered when \( \bar{k} > \tau \). Hence, we conclude that the stability of DWTCP is controlled by the decreasing phase.

\[
\square
\]

\section*{B Extra Results}

Here we provide additional results for understanding DWTCP performance.

\textbf{Websearch workload} Here, we present the slowdown in flow completion time of the websearch workload in a leaf-spine topology as explained in §5.3. As shown in Figure 14, DWTCP reduces the mean slowdown by 2x and the 99\textsuperscript{th} percentile by 3.5x (for the 60\% load). Also, the improvement in the tail slowdown is much more significant for smaller flows (up to 8x as shown in Figure 14).

\textbf{Effect of varying number of senders} In this experiment, we set the number of senders (receivers) to 10, 20, and 30 and we the default DWTCP parameters. We can verify that DWTCP adapts to different number of flows and maintains small queues (Figure 15b). It also achieves high throughput and better fairness as depicted in Figures 15a and 15c, respectively. We observe an increase in the queue size in the beginning, which is the result of synchronization among the flows.
Effect of varying network conditions  To observe the adaptability of Scout to varying network conditions, we vary the link bandwidth every 0.1 seconds as [0.4, 0.7, 1, 10, 0.8] Gbps. We run five long flows to observe the queue size, throughput, and fairness. We observe that DWTCP quickly grabs any available bandwidth when the available bandwidth changes from 1Gbps to 10Gbps and it quickly converges to full link utilization as shown in Figure 16a. There are two important observations: First, as the link capacity grows and there is less room, DWTCP adapts its aggressiveness to avoid any queue buildups (Figure 16b). Scout incurs very little overhead and Scout rate adjusts with the variation in link rate ensuring fairness among the flows (Figure 16c). Second, when the available bandwidth decreases, Scout detects it quickly and flows give up the extra bandwidth without causing any queue buildups and losing on fairness as is the case with TCP.

Performance in large BDP networks. In this experiment, we evaluate the performance of DWTCP for long haul such as intra-data center networks. Here, the challenge is that the Bandwidth-Delay Product (BDP) is very high, and the buffer sizes are much lower than the BDP. Moreover, the TCP feedback loop is very long, resulting in a very long convergence time for TCP's congestion window to reach its fair share of the high BDP. In this experiment, We generate
DWTCP converges to the optimal throughput in high BDP faster than other protocols with no queue buildups.

5 long flows, set the link capacity to 1Gbps and set RTT to 6ms.

Figure 17a shows the throughput of DWTCP, TCP and DCTCP over time. As shown, TCP takes around 1 second and DCTCP around 0.4 seconds to converge to the optimal throughput due to the high BDP. On the other hand, the congestion window in DWTCP reaches its optimal value quickly showing orders of magnitude faster convergence time as compared to DCTCP and TCP. Also, note that the Scout rate at the switch adapts quickly based on the available bandwidth and DWTCP converges quickly without any queue buildups as shown in Figure 17b which is not the case for CUBIC.