LCCN-W2025-Data Center Congestion Control Comparison: DCTCP Vs Swift

Project in Computer Networks - 236340 - Winter 2024-2025

Team: Yasmin Mitkal, Omar Garah, Amir Watted

Advisor: Eran Tavor

Table of Content

Abstract:

Efficient data transfer within data centers is critical to supporting modern computation and services. This project investigates and compares two contemporary congestion control protocols, SWIFT and DCTCP, designed for intra-data center communication. Using a simulation-based approach, we evaluated the protocols on key metrics such as throughput, latency, and packet loss under diverse traffic patterns. Our findings indicate that Swift provides lower packet loss, DCTCP achieves better delay under heavy load, and both perform similarly in throughput making Swift a strong alternative that requires further tuning to reach its full potential.. These insights contribute to understanding protocol suitability for specific data center workloads, guiding the design of efficient communication infrastructures.

Introduction:

In modern data centers, efficient data transfer is critical to ensuring the performance of large-scale distributed systems. Congestion control protocols play a vital role in maintaining high throughput and low latency.

Traditional congestion control mechanisms, like TCP, struggle to meet the unique demands of data center environments. This has led to the development of specialized protocols like SWIFT and DCTCP.

This project aims to explore, implement, and compare Swift and DCTCP protocols to evaluate their effectiveness in improving data center communication.

Congestion Control:

Congestion control ensures efficient data flow in a network by preventing overload, minimizing latency, and maximizing throughput. It dynamically adjusts the congestion window (cwnd) or packet transmission rate based on feedback mechanisms such as packet loss, delay, or Explicit Congestion Notification (ECN). By regulating the sender's transmission behavior, congestion control maintains network stability, prevents excessive queuing, and ensures fairness among competing flows.

NS-3:

ns-3 is a powerful and widely used discrete-event network simulator designed for research and educational purposes. It provides a comprehensive platform for modeling and analyzing network protocols, topologies, and applications in a controlled and reproducible environment.

For this project, ns-3 was used to simulate congestion control protocols, enabling a detailed examination of their behavior under various network conditions.

To structure our analysis clearly, we begin by presenting the **theoretical background** of the two congestion control algorithms—DCTCP and Swift. These sections cover their key principles, mechanisms, and algorithmic behavior, establishing the foundation for our project and guiding the implementation of both protocols in the NS-3 simulator.

DCTCP Overview:

DCTCP (Data Center TCP) is a transport protocol designed specifically for data center networks to address the limitations of traditional TCP in environments with low round-trip times (RTTs), high bandwidth, and latency-sensitive applications. By leveraging Explicit Congestion Notification (ECN) for fine-grained congestion feedback, DCTCP aims to achieve high throughput while maintaining low buffer occupancy. DCTCP introduces lightweight modifications to TCP. It uses Explicit Congestion Notification (ECN) to provide detailed congestion feedback to end hosts, enabling a more precise adjustment of the sender's congestion window. DCTCP is designed to operate with small queue occupancies, without loss of throughput. DCTCP achieves these goals primarily by reacting to congestion in proportion to the extent of congestion.

Principles of DCTCP:

- Early Congestion Detection: DCTCP uses a straightforward marking scheme at switches, setting the Congestion Experienced (CE) codepoint in packet headers when the queue occupancy goes above a predetermined threshold (K). This allows DCTCP senders to receive congestion signals sooner than traditional TCP, enabling faster reaction to prevent excessive queue buildup.

- Proportionate Reaction to Congestion**:** DCTCP senders don't just react to the *presence* of congestion, they respond *proportionally* to its *extent*. This nuanced reaction is based on the fraction of marked packets (α) received within a specific time window. A higher α indicates more severe congestion, leading to a more aggressive reduction in the sender's window size, while a lower α triggers a gentler adjustment.

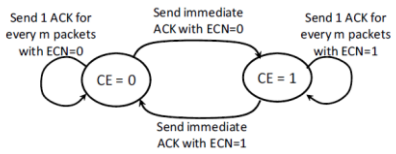
- Multi-Bit Feedback from ECN: DCTCP cleverly extracts multi-bit feedback from the inherently single-bit ECN marking mechanism. This is achieved by having the DCTCP receiver echo back the exact sequence of marked packets to the sender. The sender can then accurately calculate α, which serves as the basis for its congestion control adjustments.

- Compatibility with TCP Mechanisms: DCTCP leverages and retains many of the core mechanisms of traditional TCP, including slow start, additive increase in congestion avoidance, and recovery procedures for lost packets. The primary modification lies in how DCTCP senders respond to the ECN feedback, allowing it to integrate smoothly with existing TCP implementations.

DCTCP's Algorithm:

- Parameter K- the marking threshold : An arriving packet is marked with the CE codepoint if the queue occupancy is greater than K upon it’s arrival.
- The receiver sets the ECN-Echo flag in a series of ACK packets until it receives confirmation from the sender (through the CWR flag) that the congestion notification has been received. The receiver tries to accurately convey the exact sequence of marked packets back to the sender. Delayed ACKs are supported. To use delayed ACKs (one cumulative ACK for every m consecutively received packets), the DCTCP receiver uses the

trivial two state state-machine shown as shown to determine whether to set ECN Echo bit. The states correspond to whether the last received packet was marked with the CE codepoint or not. Since the sender knows how many packets each ACK covers, it can exactly reconstruct the runs of marks seen by the receiver.



- The sender maintains an estimate of the fraction of packets that are marked, called α, which is updated once for every window of data (roughly one RTT) as follows:

$$(1) \quad \alpha \leftarrow (1-g) \times \alpha + g \times F$$

$F = fraction\ of\ packets\ that\ were\ marked\ in\ the\ last\ window\ of\ data.$

$0 < g < 1$
$= weight\ given\ to\ new\ samples\ against\ the\ past\ in\ the\ estimation\ of\ \alpha$

- In response to a marked ACK, DCTCP uses α:

$$(2) \quad cwnd \leftarrow cwnd \times (1 - \alpha/2)$$

Note: parameters for the marking threshold (K) and the estimation gain (g) should be carefully chosen.

Logic:

- Given that the sender receives marks for every packet when the queue length is higher than K and does not receive any marks when the queue length is below K, Equation (1) implies that α estimates the probability that the queue size is greater than K. Essentially, α close to 0 indicates low, and α close to 1 indicates high levels of congestion.

- Equation (2) implies that when α is near 0 (low congestion), the window is only slightly reduced. In other words, DCTCP senders start gently reducing their window as soon as the queue exceeds K. This is how DCTCP maintains low queue length, while still ensuring high throughput. When congestion is high (α = 1), DCTCP cuts window in half, just like TCP.

Swift Overview:

Swift is a congestion control protocol designed for Google data centers to deliver high performance and simplicity in large-scale deployments. It uses delay as a congestion signal, which can be easily measured and divided into fabric and host components**.** This allows Swift to respond separately to congestion occurring in the network fabric versus at the hosts. At its core, Swift employs a straightforward AIMD (Additive-Increase Multiplicative-Decrease) controller. This controller adjusts a congestion window based on whether the measured delay exceeds a predetermined target delay.

Principles of Swift:

- Swift uses end-to-end round-trip time (RTT) measurements to adjust a congestion window with an Additive-Increase Multiplicative-Decrease (AIMD) algorithm, with the goal of maintaining a target delay.

- Delay is chosen as the primary congestion signal because it is easy to decompose into fabric and host components, can be measured accurately with modern hardware, and provides a multi-bit signal encoding the extent of congestion.

- The protocol decomposes RTT into several delay components, such as local NIC Tx delay, forward and reverse fabric delay, local and remote NIC Rx delay, and remote processing delay. These components are measured using timestamps from the NIC hardware and host software.

- Swift uses two congestion windows, fcwnd and ecwnd, to track fabric and endpoint congestion, respectively. Both windows employ the AIMD algorithm but use different delay targets. The effective congestion window is the minimum of the two.

- Swift includes mechanisms to handle large-scale incast, where the number of flows exceeds the bandwidth-delay product (BDP) of the path. In such scenarios, the congestion window can drop below one packet, and the sender paces packets to avoid overload.

- To account for longer paths and heavier loads, Swift scales the fabric target delay based on topology and load. Topology-based scaling adds a per-hop delay, and flow-based scaling adjusts the target based on the congestion window.

Swift's Algorithm:

Algorithm 1: SWIFT REACTION TO CONGESTION

Parameters: ai: additive increment, β: multiplicative decrease constant, max_mdf: maximum multiplicative decrease factor
$cwnd\_prev \leftarrow cwnd$
$bool\ can\_decrease \leftarrow (now - t\_last\_decrease \geq rtt)$ ▷ Enforces MD once every RTT

On Receiving ACK
  $retransmit\_cnt \leftarrow 0$
  $target\_delay \leftarrow$ TargetDelay() ▷ See S3.5
  if $delay < target\_delay$ **then** ▷ Additive Increase (AI)
    if $cwnd \geq 1$ **then**
      $cwnd \leftarrow cwnd + \frac{ai}{cwnd} \cdot num\_acked$
    else
      $cwnd \leftarrow cwnd + ai \cdot num\_acked$
  else ▷ Multiplicative Decrease (MD)
    if $can\_decrease$ **then**
      $cwnd \leftarrow \max(1 - \beta \cdot (\frac{delay - target\_delay}{delay}), 1 - max\_mdf) \cdot cwnd$

On Retransmit Timeout
  $retransmit\_cnt \leftarrow retransmit\_cnt + 1$
  if $retransmit\_cnt \geq RETX\_RESET\_THRESHOLD$ **then**
    $cwnd \leftarrow min\_cwnd$
  else
    if $can\_decrease$ **then**
      $cwnd \leftarrow (1 - max\_mdf) \cdot cwnd$

On Fast Recovery
  $retransmit\_cnt \leftarrow 0$
  if $can\_decrease$ **then**
    $cwnd \leftarrow (1 - max\_mdf) \cdot cwnd$

$cwnd \leftarrow$ clamp$(min_cwnd, cwnd, max_cwnd)$ ▷ Enforce lower/upper bounds
if $cwnd \leq cwnd\_prev$ **then**
  $t\_last\_decrease \leftarrow now$
if $cwnd < 1$ **then**
  $pacing\_delay \leftarrow \frac{rtt}{cwnd}$
else
  $pacing\_delay \leftarrow 0$;
Output: $cwnd, pacing\_delay$

Parameters:

$ai = additive\ increment$

$\beta = multiplicative\ decrease$

$\max\_mdf = maximum\ multiplicative\ decrease$

$cwnd = current\ window\ size$

$\min\_cwnd = minimum\ possible\ window\ size$

$\max\_cwnd = maximum\ possible\ window\ size$

$rtt = measured\ rtt$

$delay = measured\ rtt\ (seperated\ into\ fabric\ and\ hosts\ components)$

Target Delay Computation in Swift

$Target\ Delay$:

$$t = base\_target + \#hops \times \hbar + max(0, min(\frac{\alpha}{\sqrt{fcwnd}} + \beta, fs\_range))$$

$$\alpha = \frac{fs\_range}{\frac{1}{\sqrt{fs\_min\_cwnd}} - \frac{1}{\sqrt{fs\_max\_cwnd}}} \qquad \beta = -\frac{\alpha}{\sqrt{fs\_max\_cwnd}}$$

Swift dynamically adjusts its target delay based on network topology and traffic load to ensure fairness and performance across varying path lengths and flow intensities. The target delay is composed of three key components:

1. **Base Target Delay**
   Represents the minimum delay on a lightly loaded one-hop path. It includes propagation, serialization, minimal queuing delays, and measurement uncertainties.
2. **Topology-Based Scaling**
   Increases the target delay linearly with the number of hops (*#hops* × $\hbar$), compensating for longer paths. Hop count is inferred from TTL values.
3. **Flow-Based Scaling**
   Adjusts the target delay inversely with the square root of the congestion window (*cwnd*). Smaller *cwnd* (indicating more contention) leads to a higher delay target, enabling fairer bandwidth recovery for congested flows.

Additional parameters:

- fs_range limits scaling extent.
- fs_min_cwnd / fs_max_cwnd define the active scaling range.
- α and β control the scaling curve shape.

Logic:

- **Congestion Detection**: The algorithm primarily uses delay as a congestion signal. It compares the measured delay with a target delay. If the measured delay exceeds the target, it signals congestion.
- **Additive Increase (AI)**: When the measured delay is below the target delay, indicating no congestion, Swift increases the congestion window (cwnd) additively. This increase is calculated as ai * num_acked, where num_acked represents the number of acknowledged packets. This ensures that the congestion window grows gradually, probing for available bandwidth.
- **Multiplicative Decrease (MD)**: When congestion is detected (delay ≥ target delay), the algorithm decreases the congestion window multiplicatively. This decrease is proportional to how much the measured delay exceeds the target delay. This rapid reduction in the congestion window helps to alleviate congestion quickly. To avoid overreacting to the same congestion event, the multiplicative decrease is limited to occur only once per RTT.
- **Retransmission Timeout (RTO)**: The algorithm maintains a retransmission timeout for each flow. If an acknowledgment is not received within the RTO, the packet is considered lost and retransmitted. The congestion window is reduced aggressively (by max_mdf) upon an RTO to respond to potential severe congestion.
- **Fast Recovery**: Swift uses selective acknowledgments (SACK) to detect lost packets quickly. Upon detecting a lost packet, Swift retransmits it and reduces the congestion window using the maximum multiplicative decrease factor (max_mdf).
- **Pacing**: Swift uses pacing when the congestion window falls below one packet. The pacing delay is calculated as RTT/cwnd, ensuring packets are sent out at controlled intervals to prevent overwhelming the network during high congestion.
- **Congestion Window Bounds**: The algorithm enforces upper and lower bounds on the congestion window. The min_cwnd parameter sets the minimum congestion window size, while the max_cwnd parameter sets the maximum size. This prevents the congestion window from becoming too small or too large, ensuring stable and predictable behavior.

Goals:

1. Deep understanding of DCTCP and Swift.
2. Ramp up NS3 simulator.
3. Simulate and test DCTCP on a fat tree topology.
4. Implement Swift protocol in NS3.
5. Simulate and test Swift on a fat tree topology.
6. Compare between DCTCP and Swift : Creating a graph for each protocol showing the throughput as a function of number of connections .

Timetable:

Week 1:

- Introduction to the project.

Week 2: Research and learn:

- Learn about congestion control.
- Learn about DCTCP and Swift .
- Read the articles.
- Tutorial with NS3.

Week 3-5:

- Prepare for the CDR defining our goals .
- Experiment with NS3 and running different NS3 examples.
- Building the fat tree topology.

Week 6:

- Understand DCTCP implementation in NS3.
- Run DCTCP on our data center in NS3.

Weeks 7-10:

- Implement Swift in NS3.
- Run Swift on our data center in NS3.

Week 11:

- Compare between Swift and DCTCp : creating a graph for each protocol showing the throughput as a function of number of connections.

Week 12:

- Finalize.

Topology and Setup:

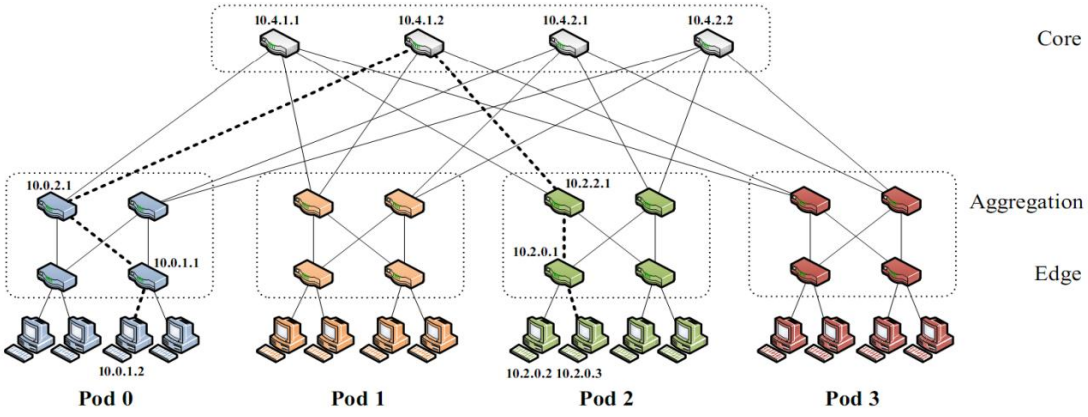
Data center networks have been extensively explored in the past decade. The Fat-Tree topology approach is one of the early approaches for data center topology.

To simulate our data center on NS3 for experimenting with DCTCP and Swift on a real life data center we used the Fat -Tree topology.

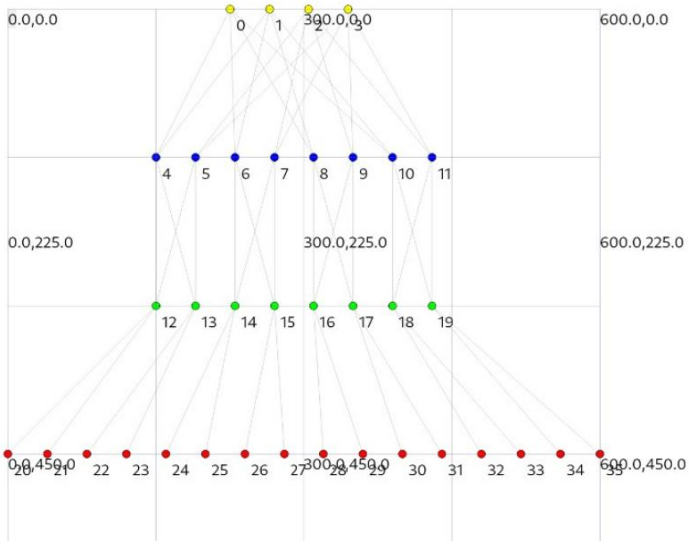
Fat -Tree Topology:

A Fat-Tree Topology is defined as a special type of network topology used in data centers, consisting of multiple pods with three layers of switches, providing multiple paths between hosts and supporting a large number of physical hosts. The topology consists of k pods with three layers of switches: edge switches, aggregation switches, and core switches. Thus, in a k pod fat-tree topology, there are k switches (each with k ports) in each pod arranged in two layers of $\frac{k}{2}$ switches, one layer for edge switches and the other for aggregation switches. Each edge switch is connected to $\frac{k}{2}$ aggregation switches. There are $\left(\frac{k}{2}\right)^2$ core switches, each of which connects to k pods. There are $\frac{k}{2}$ edge switches in each pod. Thus, for k pods, there are a total of $k^2/2$ edge switches. Altogether, $k^3/4$ physical hosts can be supported. Each aggregation switch is connected to $k/2$ edge switches and $k/2$ core switches. Thus, the total number of links, $L = 2k\left(\frac{k}{2}\right)^2 = k^3/2$.

Fat -Tree with k=4:



An illustration of our data center (with k = 4) on NS3 as shown in NetAnim:



Running DCTCP:

To conduct our analysis of DCTCP, we utilized its implementation in ns-3, which is an extension of the standard TCP protocol. DCTCP's implementation in ns-3 adheres closely to the logic described in the original DCTCP paper. The ns-3 extension for DCTCP integrates the algorithm into the TCP stack, introducing key functionalities to emulate DCTCP's behavior. Notable modifications include:

1. **Overriding the PktsAcked Function**: This function plays a central role in DCTCP by updating the **alpha** parameter, which represents the fraction of packets that encountered congestion. The alpha calculation is based on the ECN marks received in the acknowledgment packets. Each ECN-marked packet contributes to alpha, providing a measure of the network's congestion level.

2. **Congestion Window Adjustment**: The implementation ensures that the cwnd is reduced proportionally to the fraction of ECN-marked packets. When an ECN-marked packet is received, DCTCP interprets this as a congestion signal, triggering a Congestion Window Decrease (CWDR) event. During this event, DCTCP updates the congestion window as follows:

    1. Threshold Adjustment:
       The congestion window threshold is set to the "wanted window size", which reflects the appropriate cwnd to maintain based on the fraction of ECN-marked packets and the calculated alpha value. This ensures that

the cwnd reduction is proportional to the level of congestion in the network, rather than a fixed halving as in standard TCP.

2. Graceful Adjustment After CWDR:
   Once the CWDR event ends (when the network stabilizes and congestion subsides), the congestion window is adjusted to align with the previously determined threshold.

The extension also introduces dependencies on the configuration of Explicit Congestion Notification (ECN) and Active Queue Management (AQM) policies, such as RED (Random Early Detection).

To run DCTCP in our simulation, we used ns-3 scripts that initialize and configure the protocol within our data center. The setup involved defining key parameters like link bandwidth, delay, and queue characteristics. Specifically, the queues in the routers were modeled using a RED queue with Explicit Congestion Notification (ECN) marking enabled.

The RED queue class is central to implementing ECN in our simulation. It monitors the queue size and marks packets with an ECN bit if the queue length exceeds a specified threshold. The RED queue is governed by several key parameters such as Min and Max Thresholds: When the queue length is below the min threshold, packets are forwarded without any marking. Between the min and max thresholds, packets are marked with ECN in a probabilistic manner, with the probability increasing as the queue length approaches the max threshold. Once the queue length exceeds the max threshold, all packets are marked with ECN, signaling severe congestion to the endpoints.

To simulate real-world data centers with finite resources, the RED queue is configured with a maximum size. This ensures that queues remain bounded, and any additional packets are dropped when the queue is full.

For our simulation of DCTCP, we configured the min and max thresholds to be consistent across all routers, but most important we configured them to be equal (min = max) to apply DCTCP's logic and imitate the parameter K in the algorithm.

In the simulation, we carefully set these parameters to reflect typical data center network conditions. The min and max thresholds were tuned to balance sensitivity to congestion with stability and were chosen to maximize throughput.

How do the thresholds affect the fairness?

While tuning the min and max thresholds for the RED queues, we observed an interesting phenomenon: when the thresholds were set too low, fairness among connections was significantly reduced. In such cases, some connections achieved almost all the throughput, while others experienced very low throughput, indicating a lack of fairness in bandwidth distribution.

We hypothesize that this occurs because:

- **Early Fill of Buffers:** Connections that begin sending earlier quickly fill the buffers, utilizing the limited queue space available before any significant ECN marking starts.
- **ECN Feedback for Other Connections:** Subsequent connections attempting to send packets encounter ECN-marked responses immediately, interpreting this as an indication of severe congestion. As a result, these connections reduce their congestion window size significantly, keeping their throughput low.

This behavior suggests that with very low thresholds, the RED queue prioritizes early senders inadvertently, leading to persistent domination of the bandwidth by a subset of connections. The phenomenon could be tied to how DCTCP dynamically adjusts its congestion window based on ECN feedback, making it highly sensitive to queue management policies.

Although addressing this issue was not the primary focus of our project, it represents an intriguing direction for **future work**. Investigating the relationship between RED thresholds and fairness in detail could lead to insights into how to configure thresholds to balance throughput efficiency and fairness effectively, especially in dynamic data center environments.

Implementing SWIFT:

Unlike DCTCP, Swift does not have an official RFC or an existing implementation in ns-3. This lack of a predefined extension necessitated that we implement Swift from scratch, relying entirely on the details provided in the Swift article.

To integrate Swift into ns-3, we extended the existing **TCP stack** to incorporate its unique logic. This required overriding and modifying specific TCP functionalities to replicate Swift's behavior accurately. Our goal was to ensure a fair simulation environment for comparing Swift against DCTCP, highlighting their relative performance under identical network conditions.

Implementation Details

The integration of Swift into ns-3 involved overriding several core TCP functions to implement its congestion control mechanism.

One of our main challenges in implementing Swift was understanding the flow of operations within ns-3's TCP stack. We identified four key classes that define this flow:

1. **TcpSocketBase:**

This is the core class responsible for managing the flow of TCP, such as processing incoming ACKs, deciding which functions to call, and maintaining the state of the TCP socket. It contains an object of **TcpSocketState**, which stores critical data like the current congestion window (cwnd), and an object of **TcpCongestionOps**, which handles congestion control logic.

2. **TcpSocketState:**

This class acts as a container for the state variables of a TCP connection, such as the **cwnd**, slow start threshold, and other state-related parameters. It also includes fields such as **RTT**, which we used to track the round-trip time of the last ACKed packet, and **pacing rate**, which we leveraged to implement the pacing delay logic described in the Swift algorithm.

3. **TcpCongestionOps:**

This class encapsulates the congestion control behavior, with specific algorithms implemented in subclasses. For example, **TcpNewReno** is one such subclass, and we used it as the base for our Swift implementation.

4. **TcpRecovery:**

This class handles recovery mechanisms like fast retransmit and fast recovery, which are invoked during packet loss events.

By understanding how these components interact particularly how **TcpSocketBase** orchestrates the flow and delegates tasks to **TcpSocketState** and **TcpCongestionOps** we were able to identify where to implement Swift-specific logic.

Changes Made for Swift

After understanding the flow, we implemented Swift by making the following modifications:

1. **Swift Class Implementation:**

We created a Swift class that inherits from TcpNewReno, which in turn inherits from TcpCongestionOps. This allowed us to override methods specific to congestion control while leveraging existing TCP functionality.

We implemented Swift by creating a new Swift class that inherits from TcpNewReno, which itself extends TcpCongestionOps. This inheritance structure allowed us to reuse core TCP functionalities while overriding congestion control behavior specific to Swift.

Within the Swift class, we introduced key parameters essential to Swift's logic such as **alpha and beta** which are used to control flow-based target delay scaling based on the congestion window size.

2. **Adding Fields to TcpSocketState:**

We added fields such as **min window** and **max window** to enforce the boundaries of the congestion window wherever it is adjusted.

3. **Using Existing Fields in TcpSocketState:**

We utilized the existing **RTT** field in TcpSocketState to track the round-trip time of the last ACKed packet, which is critical for implementing Swift's logic of comparing the **target delay** with the measured delay (RTT). Similarly, we used the **pacing rate** field in TcpSocketState to implement the logic for **pacing delay**.

4. **Overriding Window Size Increase:**

We overrode the **IncreaseWindowSize** function in the Swift class. This function is called when an ACK is received, and we implemented Swift's logic to compare the **target delay** with the **measured delay (RTT)**. The congestion window is then adjusted accordingly to maintain the target delay.

5. **Handling Timeout Scenarios:**

In the **TcpSocketBase** class, we modified the **Retransmit** function to include Swift's logic for handling timeouts. Instead of reducing the congestion window to a single packet, Swift's algorithm implements different logic.

6. **Modifying Fast Recovery Logic:**

We updated the **EnterRecovery** function in TcpCongestionOps to incorporate Swift's fast recovery logic. This ensures the congestion window behavior aligns with Swift's principles during packet loss recovery events.

These changes allowed us to integrate Swift into ns-3 and faithfully implement its congestion control algorithm. By leveraging existing fields like RTT and pacing rate, along with our modifications, we ensured that Swift operates efficiently and accurately within the simulation environment.

Disclaimers:

Target Delay Configuration : Swift's target delay plays a central role in congestion control, but the original delay equation presented challenges: The article introduces parameters like **fs_range**, which are not fully explained or practically defined. To overcome this ambiguity, we opted to configure the target delay statically. This static configuration allowed us to implement the algorithm without the uncertainty of undefined parameters and still test its effectiveness.

Parameter Tuning : Swift relies on several parameters that significantly influence its behavior: Key parameters include **alpha**, **beta**, and **min/max window sizes**. These parameters dictate how the algorithm adjusts the congestion window based on measured delays and control its bounds. The original article does not provide detailed guidance on configuring these parameters. We had to manually tune them to balance throughput and responsiveness. Different parameter configurations could lead to varying performance outcomes, making this an area that warrants further study.

Considerations for Fair Comparison with DCTCP: To ensure a fair comparison between Swift and DCTCP, we took several measures:

1. **RTT Limited to Fabric Delay:** In DCTCP, ECN marks are applied only within router buffers, focusing solely on congestion within the fabric. To align with this behavior, Swift's congestion window calculation was also limited to fabric delay.
2. **Identical RED Queue Configuration:** We used the **RED queue** class for Swift's router buffers but disabled ECN marking to ensure identical setups for both protocols. This ensured fairness by maintaining the same buffer sizes and transmission rates across simulations.

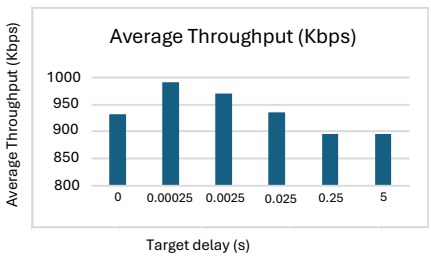
Running Swift:

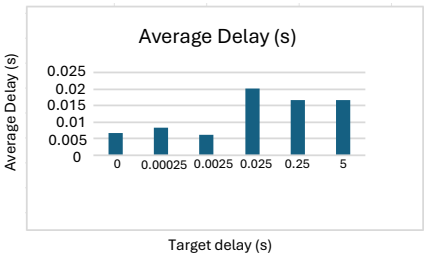
As mentioned in the previous section on implementing Swift, there was significant ambiguity surrounding many of the protocol's parameters and how they should be chosen. As a result, extensive fine-tuning was required to ensure a fair comparison between Swift and DCTCP. In this section, we present some of these parameters and discuss the results we observed. All results reported here are based on a simulated data center environment with a fat-tree topology of parameter k= 4, which corresponds to 16 hosts. The network setup involves a hotspot traffic pattern, where 650 concurrent connections are established such that all senders target a single designated receiver.

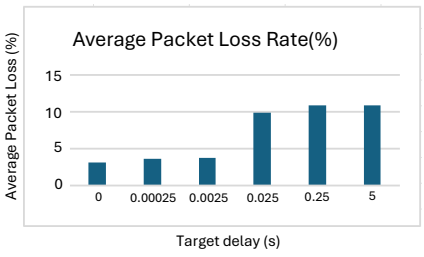
Target Delay:

One key parameter in Swift is the target delay. As discussed, we decided to configure it statically. Below, we present some results obtained using different target delays.

```
Target Delay = 0          Average Throughput = 931.568 Kbps   Average Delay = 0.00675792   Average packet loss rate = 3.13449%
Target Delay = 0.00025    Average Throughput = 991.893 Kbps   Average Delay = 0.0081513    Average packet loss rate = 3.61618%
Target Delay = 0.0025     Average Throughput = 970.33 Kbps    Average Delay = 0.00681673   Average packet loss rate = 3.66038%
Target Delay = 0.025      Average Throughput = 935.998 Kbps   Average Delay = 0.0208005    Average packet loss rate = 9.88285%
Target Delay = 0.25       Average Throughput = 895.223 Kbps   Average Delay = 0.0166009    Average packet loss rate = 10.8708%
Target Delay = 5          Average Throughput = 895.223 Kbps   Average Delay = 0.0166009    Average packet loss rate = 10.8708%
```







As the target delay decreases, we observe a corresponding reduction in both the average packet loss rate and the average RTT (latency). This behavior can be explained by the fact that a smaller target delay limits the growth of the congestion window, thereby restricting the queuing of packets in the network. Less queuing leads to lower delay and fewer dropped packets. On the other hand, when the target delay is large, the protocol allows the congestion window to expand more, resulting in increased queuing and consequently higher RTT and packet loss.

However, when it comes to throughput, the relationship is not strictly linear. Extremely small or excessively large target delays both negatively impact throughput. Very small target delays constrain the congestion window too aggressively, limiting the data rate. Conversely, very large target delays lead to excessive queuing and losses, also hurting performance. Therefore, it is crucial

to select a balanced target delay that maximizes throughput while keeping delay and packet loss within acceptable bounds.

Encouraged by these results, we conducted an additional experiment where we set the target delay to the smallest RTT observed in the session.

```
271  Target Delay = min observed rtt    Average Throughput = 1027.01 Kbps    Average Delay = 0.00928047    Average packet loss rate = 3.26633%
```

This approach allowed us to dynamically adapt to the network conditions, rather than relying on a fixed configuration. Interestingly, this strategy yielded the highest throughput (1027.01 Kbps), while maintaining a low average delay (0.00928 s) and a modest packet loss rate (3.27%).

This result suggests that using the minimum observed RTT as the target delay achieves an optimal trade-off: it avoids the drawbacks of both overly aggressive and overly relaxed congestion control. A possible explanation is that this method implicitly captures the true propagation delay of the path without inflating the window unnecessarily, which minimizes queuing and packet drops while still utilizing the available capacity efficiently.

Based on these insights, we adopt this dynamic strategy—using the minimum observed RTT as the target delay for all subsequent experiments and evaluation.

This observation suggests that further investigation is needed, as the choice of target delay has a substantial impact on Swift's performance.

Alignment with the Swift Paper

Paragraph 17 as shown in Swift's paper:

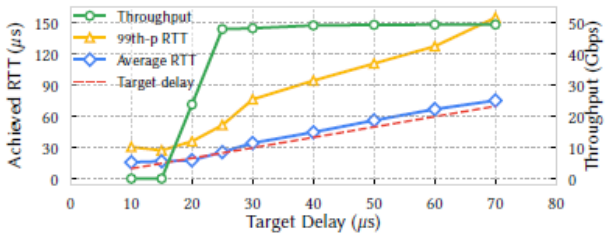


Figure 17: T_1: Achieved RTT and throughput vs. target delay, 100-flow incast.

The Swift paper highlights a key trade-off: a larger target delay can generally lead to better throughput, while a smaller target delay prioritizes lower latency. Our observations mostly align with these principles:

1. Latency Benefits of a Smaller Target Delay:
    - The paper states that setting a smaller target delay keeps the actual RTTs lower because Swift actively reduces its congestion window when delay approaches the target.

- Our results confirm this, as we observed a clear reduction in delay with smaller target delays.

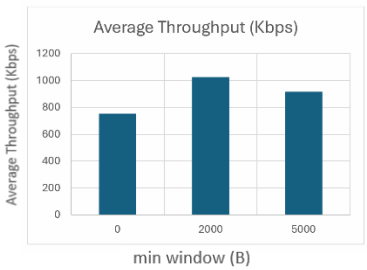
2. Throughput Benefits of a Larger Target Delay:

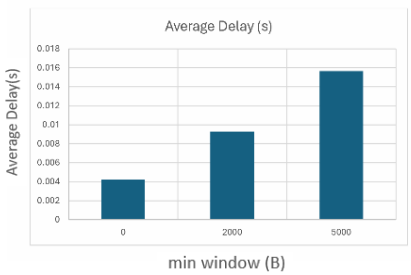
    - The paper mentions that beyond the minimum delay required for propagation and measurement inaccuracies, a higher target allows for more queuing, leading to higher throughput.
    - In our experiments, however, we found that throughput does not always increase with target delay. Initially, throughput improves as the target delay increases, but excessive queuing at large target delays results in losses and degraded performance.
    - This aligns with Figure 17 in the Swift paper, where throughput rises with increasing target delay up to a point before saturating. However, a key difference is that in our experiments, throughput actually started to decrease at some point, whereas in the paper, the throughput remains stable. This suggests that network conditions, implementation details, or specific parameters used in our setup might introduce additional constraints that were not observed in the original study.

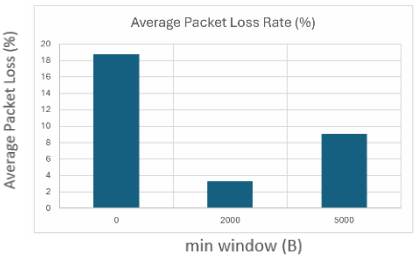
Minimum Congestion Window (min_cwnd)

Another crucial parameter in Swift is min_cwnd, which clamps the congestion window at a minimum threshold. Below, we present results from different min_cwnd settings.

```
min_cwnd = 0      Average Throughput = 753.614 Kbps   Average Delay = 0.00421358   Average packet loss rate = 18.8132%
min_cwnd = 2000   Average Throughput = 1027.01 Kbps   Average Delay = 0.00928047   Average packet loss rate = 3.26633%
min_cwnd = 5000   Average Throughput = 916.074 Kbps   Average Delay = 0.0156578    Average packet loss rate = 9.08512%
```







We hypothesize that when min_cwnd is not too small, the recovery process after a congestion event is significantly faster. Instead of restarting from an extremely

low window size and growing slowly via additive increase, the protocol can resume transmission from a more reasonable baseline, thereby reducing the time it takes to restore stable throughput.

However, if min_cwnd is set too high, the protocol risks maintaining an oversized congestion window even during periods of congestion. This can lead to excessive queue buildup, increased delay, and a higher packet loss rate—as observed in the case where min_cwnd = 5000. In such cases, the protocol does not back off sufficiently, leading to persistent congestion rather than allowing the network to stabilize.

Experiment Setup:

In our setup, we built a data center simulation to evaluate both DCTCP and Swift. As explained earlier, we used a fat-tree topology with different values of k, representing varying network scales. In all configurations, the link delay was set to 10 microseconds, and the link bandwidth was 1 Gbps. The router buffers were configured to hold 2666 packets, and the packet size was set to approximately 1500 bytes.

To assess the impact of congestion control protocols, we installed a TCP-based application on all hosts. The senders used an On-Off application, while the receivers used a Sink application to act as servers, listening for and receiving incoming messages.

The On-Off application in ns-3 models bursty traffic by alternating between two states:

- "On" state: The sender transmits data at a specified rate (in our case, 5 Gbps).
- "Off" state: The sender remains idle for a configured period.
  This pattern mimics real-world traffic bursts seen in data centers.

We tested three different traffic modes in the data center:

- Mode 0 (Hotspot Traffic): A single receiver was selected (the last host), while the other hosts sequentially became senders.
- Mode 1 (Random Hotspot Receiver Selection): A receiver was chosen randomly, and the remaining hosts were randomly selected as senders.
- Mode 2 (Random Pairing): Hosts were randomly paired, with each sender transmitting to a randomly chosen receiver.

For each experiment, we ran the simulation separately with DCTCP and Swift, measuring the following Key Performance Indicators (KPIs):

- Average throughput
- Average packet loss rate
- Average Round Trip Time (RTT)

These metrics were analyzed as a function of the number of active connections.

To ensure accurate results, we allowed the network to stabilize before starting measurements. Traffic was initiated after an initial warm-up period, and data was collected only once the system reached a steady state.

Results and comparison: DCTCP vs. Swift:

We evaluated the performance of DCTCP and Swift under varying conditions to better understand their behavior in data center environments. Specifically, we tested the protocols across different network scales (using different K for the fat-tree) and under diverse traffic patterns (using different simulation modes). In addition, we extended our earlier results by examining multiple configurations of Swift's target delay, as this parameter significantly influences its performance.

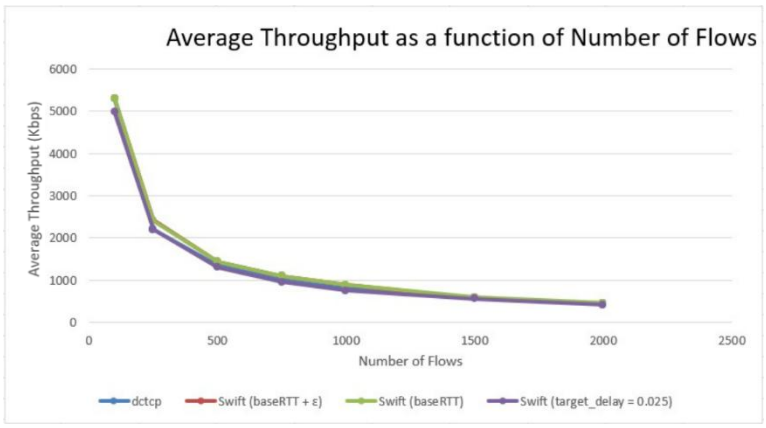
We compare three Swift configurations:

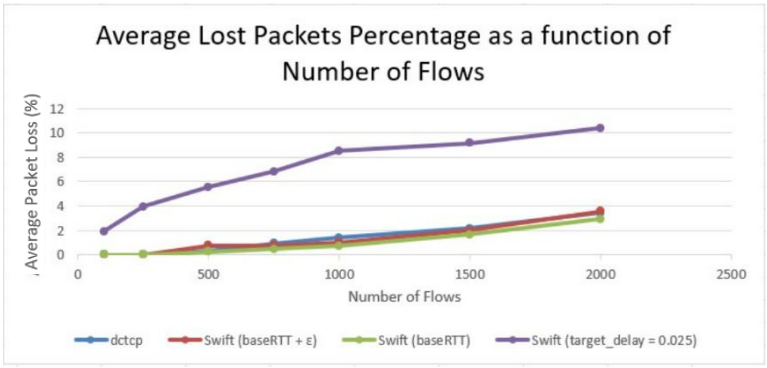
- **Swift (baseRTT)** — target delay is the minimum observed RTT.
- **Swift (baseRTT + ε)** — adds a small constant offset to baseRTT.
- **Swift (target delay = 0.025)** — uses a fixed, non-adaptive target delay.

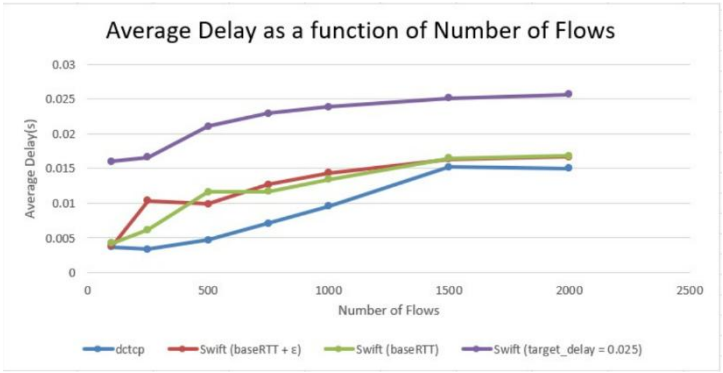
To ensure reliability, each data point shown in the following graphs represents the average of three independent simulation runs.

K=4 (16 hosts) and a 15 seconds simulation runtime:

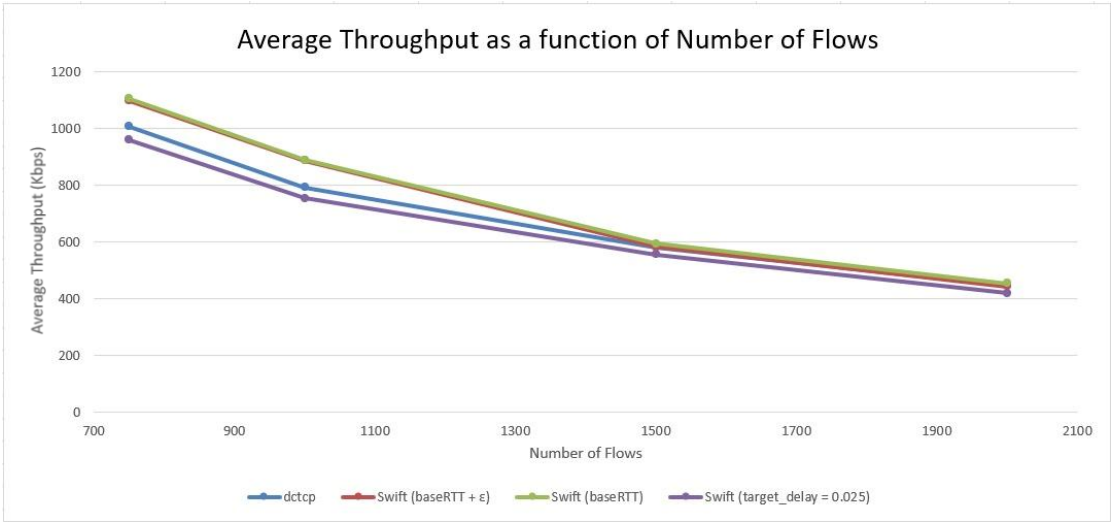
-Mode 1 (random hot spot):







-Zooming in on Throughput:



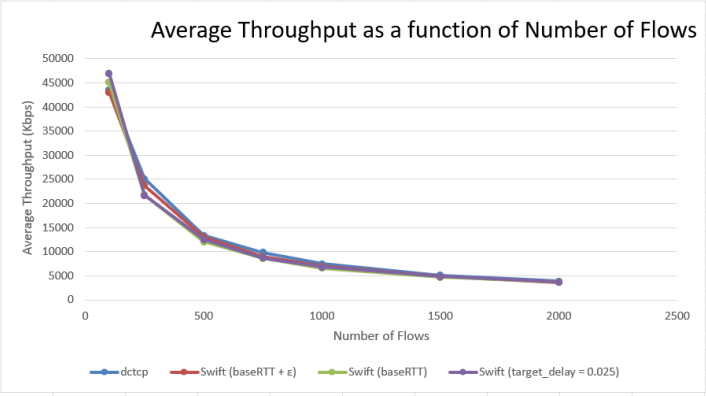
Observations:

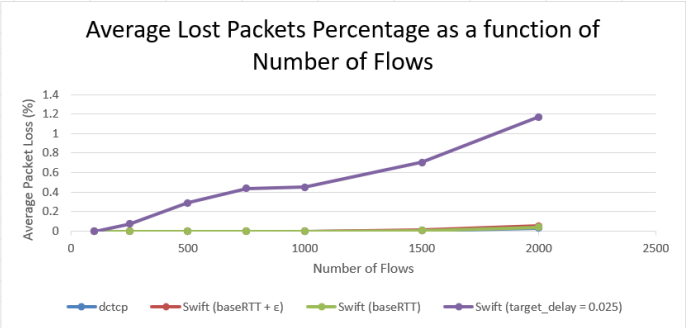
In this scenario, as the number of flows increases from 100 to 2000, we observe a clear degradation in throughput across all protocols. Initially, all protocols deliver high throughput (above 5 Mbps at 100 flows), but as the number of concurrent senders increases, throughput drops significantly. Swift variants based on baseRTT and baseRTT + ε consistently maintain higher throughput than DCTCP at all flow levels. Meanwhile, Swift with a fixed target delay = 0.025 underperforms in terms of throughput, especially at higher flow counts.

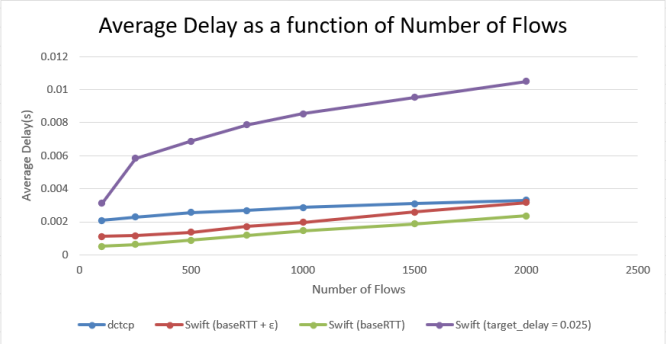
Packet loss remains negligible up to 250 flows for most protocols, except for Swift with fixed target delay, which already shows significant loss at 250 flows (~3.9%). As the number of flows increases, loss grows for all protocols, but the fixed-target Swift variant exhibits the steepest increase, reaching over 10% at 2000 flows. Swift (baseRTT) and (baseRTT + ε) maintain lower loss rates compared to DCTCP at most scales.

Regarding delay, DCTCP shows the lowest average RTT for small flow counts but converges toward Swift values as load increases. Swift (target delay = 0.025) again displays the highest delays, even at lower flow levels. Swift (baseRTT) and (baseRTT + ε) maintain moderate and relatively stable delays throughout the flow range.

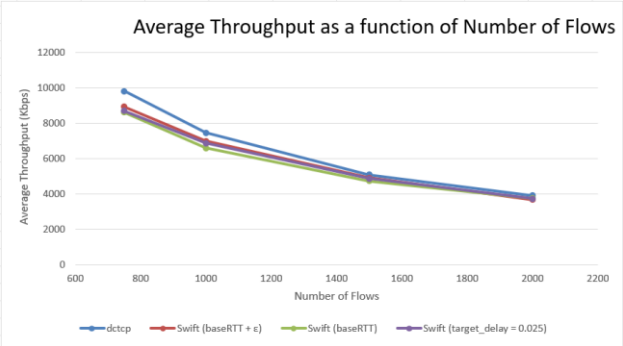
Mode 2 (random pairing):







-Zooming in on Throughput:



Observations:

Under random sender-receiver pairing (mode 2), we observe a steady decline in throughput across all protocols as the number of flows increases. At lower flow counts (e.g., 100), all Swift variants and DCTCP achieve very high throughput values, exceeding 43 Mbps, with Swift (target delay = 0.025) slightly outperforming all others. However, from 250 flows onward, DCTCP consistently outperforms all Swift variants, while Swift (target delay = 0.025) generally lags behind despite its early advantage.

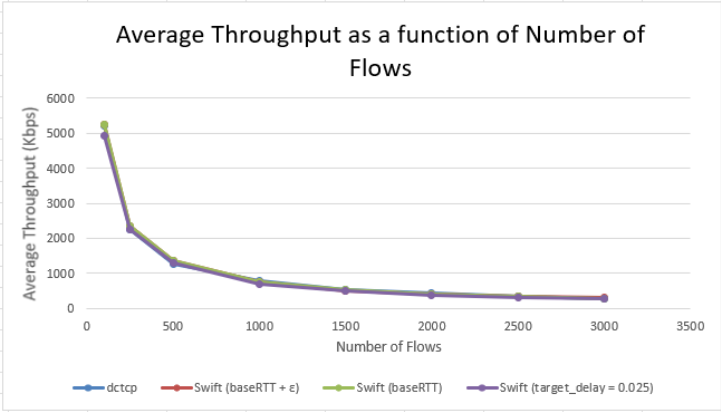
Packet loss is negligible for all protocols up to 750 flows, with the exception of Swift (target delay = 0.025), which begins showing measurable loss as early as 250 flows and rises rapidly to over 1.1% at 2000 flows. In contrast, DCTCP and the baseRTT-based Swift variants maintain near-zero loss until higher flow counts, with DCTCP demonstrating the lowest loss at scale.

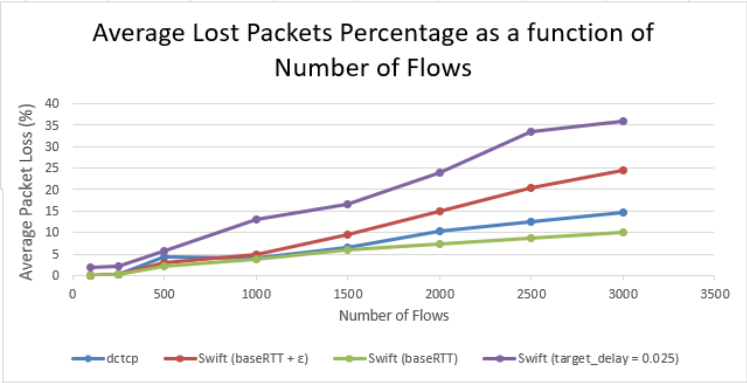
In terms of delay, all Swift variants — especially Swift (baseRTT) — consistently maintain lower average delay than DCTCP. DCTCP's delay increases steadily as load rises, while Swift (target delay = 0.025) exhibits the highest delay, exceeding 10 ms at 2000 flows. Notably, Swift (baseRTT) achieves the lowest latency across the entire range of flow counts.

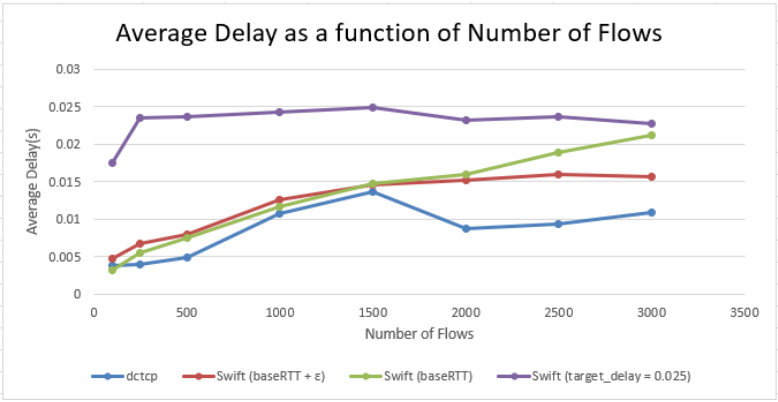
We also observe that in low-congestion scenarios (evident from near-zero packet loss), Swift (baseRTT) is outperformed in throughput by all other protocols, including DCTCP. However, it still achieves the best delay performance.

K=12 (432 hosts) and a 7 seconds simulation runtime:

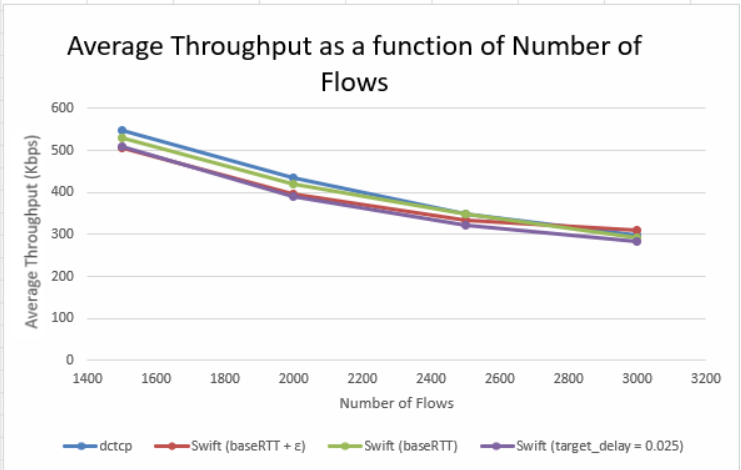
Mode 1 (random hot spot):







Zooming in on Throughput:



Observations:

In this higher-scale FatTree setup with k=12, we observe a similar trend to the previous setup: throughput generally decreases as the number of flows increases. At lower flow counts (100–500), Swift (baseRTT) and Swift (baseRTT + ε) provide slightly higher throughput than DCTCP, with Swift (target delay = 0.025) trailing behind. As the number of concurrent flows increases beyond 1000, the gap between protocols narrows, and throughput degrades significantly for all, with Swift (baseRTT + ε) and DCTCP occasionally trading places in performance. Interestingly, at 3000 flows, Swift (baseRTT + ε) slightly surpasses DCTCP.

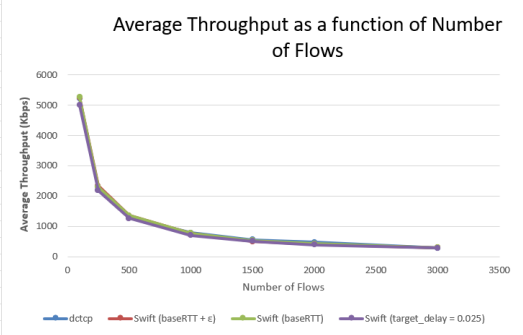
Packet loss begins to appear earlier and more sharply than in the k=4 scenario. DCTCP shows low loss up to 250 flows, but from 500 flows onward, loss grows quickly, exceeding 14% at 3000 flows. Swift (baseRTT) and (baseRTT + ε) also exhibit increasing packet loss, but at a slower rate. Swift (target delay = 0.025), however, demonstrates severe packet loss — starting as early as 100 flows (~2%) and escalating dramatically to over 35% at 3000 flows.

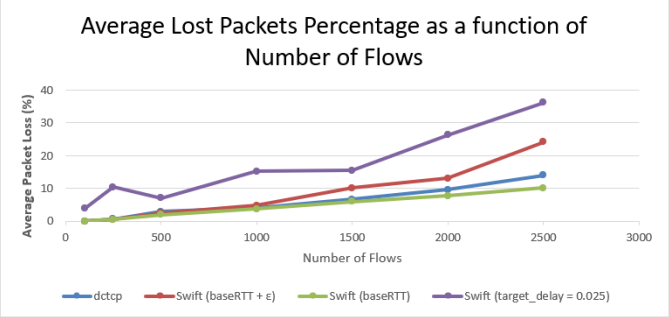
In terms of delay, Swift (baseRTT) and (baseRTT + ε) maintain moderately low and stable average delays as flows increase. DCTCP starts with the lowest delay at small flow counts but begins to converge toward Swift's delay values as the load increases. Again, Swift with a fixed target delay consistently incurs the highest delay, even at the lowest load levels.

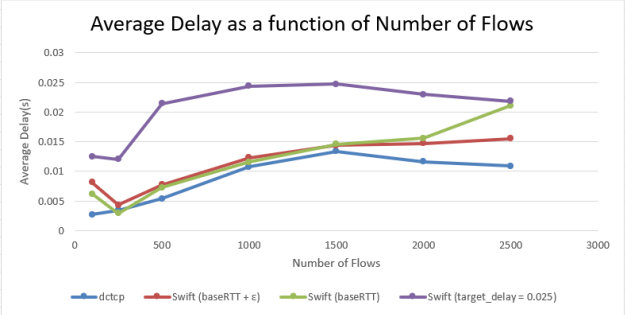
Lastly, we note that in each run, there were noticeable variations in throughput, delay, and packet loss for the same number of flows and protocol. Since each run selects a different receiver host at random, we hypothesize that the specific placement of the receiver within the FatTree topology affects congestion dynamics and therefore impacts the results. This variation suggests that host location can influence performance in incast scenarios.

To eliminate this source of non-determinism and enable a more controlled and fair comparison across protocols, we conducted a separate set of simulations in **mode 0**, where the hotspot is fixed to the last host in the topology. This setup ensures consistent traffic patterns and isolates protocol behavior from topological randomness.

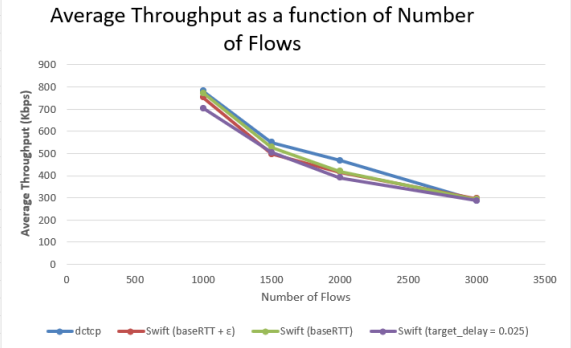
Mode 0 (last host is hot spot):







-Zooming in on Throughput:



Observations:

We observed that the performance trends in the fixed hotspot configuration (mode 0) closely resemble those in the random hotspot configuration (mode 1), across all key metrics — throughput, packet loss, and delay. The consistency of these patterns suggests that, in a FatTree topology with k=12, the specific placement of the hotspot host does not significantly impact the overall behavior of the protocols under study. This reinforces the validity of using mode 1 (random hotspot) as a representative and fair configuration for comparing the performance of congestion control protocols in large-scale data center networks.

Discussion on the results and alignment with SWIFT's article:

- Key Findings from Our Evaluation:

Throughput:

Throughput consistently decreased with increasing flow count due to rising congestion and queue pressure. In low-congestion scenarios, Swift (target delay = 0.025) often achieved the highest throughput. This is expected: because it uses a static and relatively high delay target, it allows the congestion window to grow aggressively fully utilizing available bandwidth when queues are still short and packet loss is negligible.

However, as flow counts increase and congestion emerges, Swift (target delay) suffers the most. Its static delay target becomes too aggressive, resulting in significant packet loss and degraded throughput.

When comparing DCTCP and Swift (baseRTT) specifically, results vary slightly by topology. In smaller-scale scenarios (k=4, e.g., mode 1), Swift (baseRTT) consistently outperforms DCTCP in throughput across all flow counts.

However, in larger-scale configurations (k=12), DCTCP generally performs slightly better in high congestion. In flow counts ranging from 2000 to 3000, DCTCP consistently edges out Swift (baseRTT). Despite this, the differences are very small:

- The maximum throughput gap observed between DCTCP and Swift (baseRTT) was ~13 Kbps (at 2000 flows),
- The average difference over 2000, 2500, and 3000 flows was only ~6.38 Kbps.

These results suggest that Swift (baseRTT) and DCTCP are nearly equivalent in throughput under high congestion, with DCTCP having a marginal advantage in large-scale topologies.

Packet loss:

Among all protocols, Swift (target delay = 0.025) suffered the most, frequently reaching over 30% loss in high-load scenarios such as k=12, due to its static and overly aggressive congestion response.

In contrast, Swift (baseRTT) consistently demonstrated the lowest packet loss across all topologies and flow counts. It maintained lower loss than both DCTCP and Swift (baseRTT + ε), especially as congestion intensified.

This behavior stems from the fact that Swift (baseRTT) uses the minimum observed RTT as its target delay, which encourages the sender to maintain the

congestion window small enough to avoid queue buildup altogether. As a result, the window size is continually adjusted to keep the flow operating close to the propagation delay, minimizing queuing and preventing packet loss even under increasing load.

Swift (baseRTT + ε) showed slightly higher loss than baseRTT but still performed better than the fixed-target variant, and in some cases matched or slightly underperformed DCTCP depending on the load and topology.

Delay:

Swift (baseRTT) consistently achieved the lowest delay in scenarios with moderate to low congestion, such as mode 2 (random pairing) and early stages of mode 0/1. This aligns with its design: by using the minimum RTT observed as its target delay, Swift (baseRTT) maintains the congestion window small and avoids queuing, keeping delay close to the bare propagation latency.

However, under heavier congestion, particularly in k=12 topologies at high flow counts, DCTCP sometimes achieved lower average delays than Swift (baseRTT). This is likely due to DCTCP's ECN-based backoff reacting more promptly to queue buildup in certain incast-heavy scenarios.

An unexpected behavior was also observed in DCTCP at 2500–3000 flows in k=12 (both mode 0 and mode 1), where the average delay decreased compared to 1500–2000 flows. This drop may be explained by a shift in behavior where losses or ECN-triggered reductions lead to more aggressive throttling, effectively lowering the queue occupancy though further investigation would be required to confirm this hypothesis.

Swift (target delay = 0.025) consistently showed the highest delay in all scenarios, due to its static and relatively high queuing tolerance. This makes it unsuitable for latency-sensitive traffic, especially under heavy load.

Alignment with swift paper

The Swift paper evaluates Swift by comparing it against GCN, a production-grade variant of DCTCP deployed at scale within Google's data centers. GCN includes carefully tuned ECN marking thresholds, hardware-based pacing, and NIC-level congestion handling. The authors describe it as the best-performing DCTCP implementation available in their environment. This makes Swift's reported gains even more significant, as they are measured against a highly optimized baseline.

The Swift paper makes several core claims:

- Swift achieves significantly lower delay than GCN, especially under incast or high-fan-in workloads.
- Swift matches or approaches GCN in throughput, while maintaining tighter control over queues.
- Swift consistently outperforms GCN in packet loss, particularly in tail scenarios.
- Static target delay configurations are unstable and perform poorly under dynamic congestion, reinforcing the need for adaptive delay targets.

Experimental Differences

While our evaluation is based on the same protocol logic, our setup differs substantially from that of the Swift paper. We conduct experiments in ns-3, using software-based congestion control and standard fat-tree topologies. We compare Swift against the default DCTCP implementation in ns-3, not GCN, and simulate traffic patterns such as fixed hotspot incast, random hotspot, and random sender-receiver pairing. As such, we do not expect to match the Swift paper's exact numbers — but instead aim to evaluate whether the same qualitative trends hold.

Comparative Results

Despite the differences in environment and baseline tuning, our results closely align with the Swift paper's findings in terms of throughput and packet loss, while diverging slightly in delay behavior.

✅ Throughput

Like the Swift paper, we find that Swift (baseRTT) and DCTCP achieve comparable throughput under most conditions.

- In low-congestion scenarios, Swift (target delay = 0.025) achieves the highest throughput due to its aggressive window growth.
- In smaller topologies (e.g., k=4), Swift (baseRTT) consistently outperforms DCTCP across all flow counts.
- In larger setups (k=12), DCTCP slightly outperforms Swift (baseRTT) under high congestion, but the margin is small.

These findings match the Swift paper's claim that Swift offers line rate or near line rate throughput while improving other metrics.

✅ Packet Loss

Our results also reinforce Swift's strength in controlling packet loss.

- Swift (baseRTT) consistently exhibited lower packet loss than DCTCP, especially as congestion intensified.
- Swift (target delay = 0.025) suffered significant packet loss — often exceeding 30% under high load — confirming the Swift paper's warning that static target delays are ill-suited for dynamic congestion.

⚠️ Delay

In contrast to the Swift paper, which shows Swift achieving substantially lower delay than GCN in all cases, our results show more variation:

- Swift (baseRTT) achieved the lowest delay under moderate to light congestion.
- However, in heavily congested scenarios DCTCP sometimes had lower delay than Swift (baseRTT).

Conclusions:

- Swift and DCTCP showed similar throughput behavior, especially under high congestion. Swift (baseRTT) consistently achieved lower packet loss, demonstrating better stability under load. However, DCTCP generally maintained lower delay in our simulations. This suggests that further tuning of Swift's target delay or pacing behavior may be needed to fully realize its intended low-latency advantage.
- Our findings align with the original Swift paper in terms of throughput and packet loss, where Swift (baseRTT) performs competitively and maintains low loss under congestion. However, we did not observe the same delay improvements reported in the Swift paper.
- Parameter tuning is critical. We found that values such as min_cwnd, max_cwnd, and especially the choice of target delay can significantly influence protocol performance. In particular, using overly aggressive or static values without adapting to network conditions can lead to queuing, loss, and unstable behavior.
- Setting Swift's target delay involves a clear trade-off: smaller values reduce delay and packet loss by limiting window growth, while larger values initially improve throughput but eventually cause excessive queuing, loss, and degraded performance. Using the minimum observed RTT as the target strikes an effective balance, highlighting the importance of carefully tuning this parameter for optimal results.
- The simulation results suggest that Swift's delay-based design offers a promising alternative to ECN-based schemes, but achieving its full

potential in delay performance likely requires more precise calibration of target delay and pacing behavior.

Difficulties:

1. Setting up the simulator. NS3 has many versions and so many different sources for handling with it that in some point it felt like a never ending loop while trying to set it up.

2. We found difficulties in tuning the some parameters for each of the protocols in the simulation, for example : in DCTCP the min, max thresholds. In Swift the min, max cwnd. It is not explained in the articles how are these parameters chosen so we had to experiment and fine tune them.

3. The fact that swift doesn’t have an official RFC and we had to rely in our implementation only on the paper where there were some missing details so we had to make some assumptions and try to complete it according to our understanding.

4. While implementing Swift we had to dig up the existing files of tcp and understand NS3 mechanisms to integrate Swift's logic. We found some difficulty as this is a very big system of files with many dependencies where each minor change could lead to major bugs.

Future Work:

While our study provides valuable insights into the behavior of Swift and DCTCP under various traffic patterns and congestion levels, several promising directions remain open for exploration:

1. **Dynamic Target Delay Configuration**
   One of the key challenges we encountered with Swift was the configuration of the target delay. Although the original paper suggests a mathematically grounded method for deriving this delay based on several parameters, many of those parameters were either unspecified or impractical to determine in our simulation context. Consequently, we experimented with two extremes: a relatively large static delay (0.025s) which performed well under light congestion, and a baseRTT approach which selects the smallest RTT as the target offering robust performance under high congestion. Future work could explore mechanisms for

dynamically switching between these configurations depending on the congestion level in the network, potentially combining the benefits of both.

2. **Hybrid Congestion Signals: Delay + ECN**
   Swift exclusively relies on delay as its congestion signal, while DCTCP depends solely on ECN markings. A hybrid approach leveraging both ECN and delay signals might offer more accurate or responsive control under diverse workloads. Future efforts could experiment with integrating ECN into Swift's congestion control loop, potentially enhancing both accuracy and responsiveness.

3. **Fairness Evaluation**
   Although our study briefly examined fairness (specifically for DCTCP), we did not thoroughly evaluate this metric across both protocols. Fairness is a crucial attribute for congestion control algorithms in datacenters. A detailed analysis comparing Swift and DCTCP in terms of fairness could offer important insights and optimizations.

4. **Parameter Tuning and Automation**
   Due to a lack of definitive guidance in the literature, many of Swift's parameters had to be manually set. This tuning process was both time consuming and prone to bias. Automating parameter tuning using adaptive algorithms or learning based techniques could significantly improve the usability and performance of Swift in real world deployments. Additionally, enabling runtime dynamic adaptation of these parameters could further enhance performance under shifting network conditions.

5. **Scaling to Higher k in Fat-Tree Topologies**
   Simulations with a fat-tree topology of k=12 were extremely resource intensive, with some runs taking several days to complete. Due to these constraints, we were limited in the scales we could reasonably explore. With access to more powerful computational resources or optimized simulation techniques, future studies could investigate the behavior of Swift and DCTCP at larger k values and broader datacenter scales, revealing new scalability insights.

6. **In-depth Analysis of Congestion Window Behavior**
   During the course of this work, we occasionally examined the behavior of the congestion window to confirm expected behavior. However, a deeper analysis of the window dynamics across both DCTCP and Swift could yield deeper insights into the protocols responsiveness, stability, and efficiency. Such an analysis could help improve the protocol by identifying issues and guiding better congestion control.

7. **Adaptive Recalibration of baseRTT**
   As an extension of the baseRTT approach we previously suggested, another idea we intended to explore is to dynamically update baseRTT over time. In our current method, baseRTT is fixed as the lowest RTT seen during a flow's lifetime. While this works under consistent congestion, it may become inaccurate if conditions change. Periodically re-sampling the minimum RTT within defined intervals could allow the target delay to better reflect current congestion, improving Swift's adaptability.

References:

1. Kumar, Gautam et al. "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter" Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication . New York, New York : Association for Computing Machinery. 514–528. Web. (https://dl.acm.org/doi/pdf/10.1145/3387514.3406591/)
2. Data Center TCP (DCTCP) Mohammad Alizadeh, Albert Greenberg , David A. Maltz, Jitendra Padhye† , Parveen Patel , Balaji Prabhakar , Sudipta Sengupta , Murari Sridharan https://people.csail.mit.edu/alizadeh/papers/dctcp-sigcomm10.pdf
3. https://www.sciencedirect.com/topics/computer-science/fat-tree-topology
4. https://www.nsnam.org/releases/ns-3-38/documentation/
5. https://www.nsnam.org/docs/release/3.44/doxygen/index.html

Appendix : NS-3 + Swift/DCTCP Data Center Simulation Setup

A1. Download and Build NS-3 (v3.43)

1. **Download NS-3**
2. wget https://www.nsnam.org/release/ns-allinone-3.43.tar.bz2
3. tar -xvf ns-allinone-3.43.tar.bz2
4. cd ns-allinone-3.43
5. **Build NS-3**
6. ./build.py --enable-examples --enable-tests
7. ./test.py

A2. Integrating the Custom Simulator

Replace/Add These Files from the GitHub Repo

Clone the repo:

git clone https://github.com/omarghara/DCTCP-SWIFT-YASOMAM

cd DCTCP-SWIFT-YASOMAM

Now apply the following updates to NS-3:

Target Directory	Action	Files
src/internet/model/	✅ **Add**	tcp-swift.cc, tcp-swift.h
src/internet/model/	✅ **Replace**	tcp-congestion-ops.cc/.h
		tcp-recovery-ops.cc/.h
		tcp-socket-base.cc/.h
		tcp-socket-state.cc/.h
src/internet/	✅ **Update**	CMakeLists.txt – Add the new Swift source files
scratch/	✅ **Add**	swift_cmd.cc, run_swift.sh

Use cp to move the files into their corresponding NS-3 folders.

A3. Rebuild NS-3 with Swift

From the ns-3.43 root directory:

```
./ns3 configure
./ns3 build
```

A4. Running the Simulation

Single Command via CLI

Example Run

```
./ns3 run "scratch/swift_cmd \
 --protocol=swift \
 --mode=2 \
 --k=4 \
 --p2p_DataRate=10Gbps \
 --p2p_Delay=10us \
 --num_of_flows=50 \
 --app_packet_size=1448 \
 --app_data_rate=100Mbps \
 --flowmonitor_start_time=0.1 \
 --simulation_stop_time=10 \
 --swift_mode=1"
```

Parameters Explained

Parameter	Description
--protocol	Choose congestion control protocol: swift or dctcp
--mode	Defines flow-to-host behavior: **0-** All to last host **1-** All to one random host **2-** Each flow has random sender/receiver
--k	Fat-tree switch port count (determines topology size)
--p2p_DataRate	Bandwidth of point-to-point links (e.g., 10Gbps)
--p2p_Delay	Delay of point-to-point links (e.g., 10us)
--num_of_flows	Number of simultaneous flows to generate
--app_packet_size	Size of each application packet (in bytes)
--app_data_rate	Rate at which the application generates packets
--flowmonitor_start_time	Time to start collecting FlowMonitor statistics
--simulation_stop_time	Total simulation runtime (in seconds)
--swift_mode	Sets the target delay logic for Swift: **0-**Use baseRTT **1-** Use baseRTT + ε **2-** Use static delay value

---

Batch Testing via Script

Run:

./run_swift.sh

This script automates multiple simulation runs with various configs and logs results for comparison.

A5. Visualizing Results

The simulation outputs:

Metric	**Where to Find It**
Throughput	Console output / .csv file
Packet Loss	Console output / .csv file
NetAnim Trace	scratch/swift-animation.xml

View the Simulation in NetAnim

We provide a NetAnim-compatible version of the fat-tree topology implemented in tcp-fat-tree-netanim. Use this file to visualize your simulation in NetAnim.

Steps:

1. Build NetAnim:
2. cd ../netanim
3. qmake NetAnim.pro
4. make
5. Run NetAnim:
6. ./NetAnim
7. Open the Simulation Output:
8. ./NetAnim ../ns-3.43/scratch/swift-animation.xml

The animation file is generated when using tcp-fat-tree-netanim. This version is a simplified version of the full fat-tree and is tailored for use with NetAnim to help visualize flow paths, queue buildup, and link usage**.**

Behind the Scenes

Component	Description
swift_cmd.cc	Main runner: builds fat-tree, launches OnOff apps, logs stats
tcp-swift.cc/h	Swift congestion control module (delay-based feedback)
CLI Parameters	Allow full customization without touching code
Output Metrics	Throughput, packet loss, delay via FlowMonitor

---

🔧 **What You Can Modify**

- **Topology**: k value to scale the fat-tree
- **Congestion control**: choose DCTCP or Swift via CLI
- **Delay Targets**: change swift_mode to simulate different delay sensitivities
- **Application Layer**: modify OnOff app packet size and rate
- **Duration**: extend simulation_stop_time

⚠️ **Important Note on DCTCP & Swift Coexistence**

⚠️ **Please Read Before Running Simulations**

This simulation setup uses the **DCTCP implementation from standard NS-3**, alongside a **custom Swift implementation**.

To avoid conflicts between **Swift** and **other TCP protocols** (including DCTCP), we recommend:

- **Maintain Two NS-3 Versions**:
    - One for **standard protocols** like DCTCP (ns-3.43/)
    - One for **Swift** with custom TCP modifications (ns-3.43-swift/ or similar)

Although Swift includes a feature to **toggle itself on/off at runtime**, in practice:

- Some file-level or class-level **conflicts may still occur**
- These conflicts need manual review and validation

✅ Keeping separate NS-3 directories avoids protocol contamination and ensures reproducibility.