SQCC: Stable Queue Congestion Control

Jian Tang[1], Tingting Xu[1], Liming Wang[2], Zhenjie Lin[2], Ming Zhao[2],
Xiaoliang Wang[1], Cam-Tu Nguyen[1], Chen Tian[1], Zhuzhong Qian[1], Wenzhong Li[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, China
[2]CSG China Southern Power Grid Digital Platform Technology Company, China

2023 14th International Conference on Network of the Future (NoF) | 979-8-3503-3807-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/NoF58724.2023.10302751

***Abstract*—The advent of high-speed networks has revolutionized data center capabilities by providing low latency and high bandwidth for applications. However, traditional TCP congestion control algorithms are no longer adequate for data center networks. RTT-based congestion control leverages advanced NIC hardware to identify accumulated queuing delay of the end-to-end path. It is simple, effective, and adaptable to different environments. Nevertheless, RTT-based congestion control faces challenges related to unstable queue length and oscillation caused by RTT feedback delays. With the increase of queue length, the oscillation range also amplifies. To address these issues, we propose SQCC, which introduces two key enhancements. Firstly, it employs a novel error function to regulate the queue length within a controlled range that is proportional to the number of incast flows. Secondly, it incorporates self-adjustable parameters for rate increment and RTT threshold, effectively managing queue oscillation and ensuring a non-empty link. We evaluate the algorithm's effectiveness through NS3 network simulations, and the results demonstrate that SQCC achieves an 80% reduction in queue size upon convergence and exhibits a significantly low oscillation range (27% to 57%) in large-scale incast scenarios.**

***Index Terms*—Data center networks, Transport protocols, Congestion control**

I. Introduction

To ensure the performance, high-performance computing [1] and large-scale distributed machine learning [2]–[11] applications demand extremely low network latency. Low-latency networks require congestion control algorithms that can maintain a stable low queuing latency [12]–[14]. Traditional TCP congestion control algorithms are insufficient for data center networks because they rely on packet loss as a congestion signal, which can cause significant delays in detecting, responding to congestion, as well as converging to stable status.

As a result, numerous congestion control algorithms tailored to data center environments have been proposed [15]–[21]. Among them, RTT-based congestion control algorithms have gained increasing attention from enterprises and researchers in recent years due to their simplicity and ease of deployment. These algorithms rely on the round-trip time (RTT) between the sender and receiver as a congestion signal, which allows them to detect and react to changes in network conditions more quickly than traditional TCP algorithms.

Compared to other congestion control algorithms, the RTT-based approach does not rely on the underlying network switch features ike ECN, and is measured solely by the sender [22]. It means that RTT-based approaches are equally applicable even in network architectures without switches. Additionally, the implementation of RTT-based congestion control is relatively simple, making it easy for developers to deploy, upgrade, and maintain. However, existing RTT-based congestion control algorithms still fail to meet the requirements of ultra-low latency in high-speed data centers.

Fundamentally, the RTT-based approach itself suffers from feedback delay. As the congestion grows, it takes long time for the sender to receive acknowledgments (ACKs) and calculate the RTT information. The growing feedback delay can cause long congestion reactions, which affects the ability of the algorithm to eliminate congestion in a timely manner. Previous works such as TIMELY [17] and patched TIMELY [23] struggle to rapidly reduce the sending rate when the RTT is large. For incasts traffic, traffic exacerbates packet accumulation in the congestion point with the increase of the scale, further delays the feedback and leads to the queue oscillation [24].

Therefore, when designing and deploying RTT-based congestion controls in data centers, the following points need to be considered: 1) Designing effective rate reduction metrics that can quickly respond and reduce the rate when the RTT is large, ensuring a stable low-latency queue. 2) Addressing the issue of feedback delay in RTT measurements to maintain queue length within a small range of oscillation.

This paper analyses the existing issues with RTT-based congestion controls and introduces a novel RTT congestion control, named SQCC, that can maintain a relatively stable queue, reducing the side effect of delayed feedback and fluctuating queue caused by incast traffic. The contributions are summarized as follows:

- SQCC introduces a new rate reduction metric that allows for rapid rate reduction with the increase of RTT, effectively emptying the queue to reduce congestion. The rate reduction metric in SQCC bounds the congestion queue length, i.e., when the algorithm is stable, the queue length is confined within a limited range with the increase of incast flows. The maximum queue length is guaranteed not to exceed the theoretical upper limit.
- SQCC incorporates two self-adjustable parameters, rate increment, and RTT threshold, reacting to queue oscillations caused by incast flow size fluctuation. It effectively reduces the range of queue oscillations when the algorithm is stable while ensuring high-resource utilization, i.e., keeping non-empty link.
- Through flow modeling analysis and NS3 [25] simulation experiments, it has been verified that SQCC achieves significant improvements compared to previous RTT-based congestion control algorithms in scenarios with

large-scale incast flows. The maximum queue length during stability can be reduced by up to 80%, and the corresponding range of oscillation can be reduced by 27% to 57%, demonstrating the effectiveness of SQCC.

II. Motivation

A. TIMELY Rate Adjustment

TIMELY [17] was proposed and implemented as a congestion control mechanism in data centers by using only RTT variations for congestion control. It adjusts the sending rate based on the gradient of RTT changes. We briefly review the corresponding algorithms and apply the same notations in the literature [17].

Algorithm 1 TIMELY

$newRTTDiff = newRTT - prevRTT$
$prevRTT = newRTT$
$rttDiff = (1-\alpha) \cdot rttDiff + \alpha \cdot newRTTDiff$
$rttGradient = \frac{rttDiff}{D_{minRTT}}$
if $newRTT < T_{low}$ **then**
    $rate = rate + \delta$
else if $newRTT > T_{high}$ **then**
    $rate = rate \cdot (1 - \beta \cdot (1 - \frac{T_{high}}{newRTT}))$
else if $rttGradient \leq 0$ **then**
    $rate = rate + \delta$
else
    $rate = rate \cdot (1 - \beta \cdot rttGradient)$
end if

The rate adjustment in TIMELY (Algorithm 1) involves calculating the RTT gradient (ΔRTT, or $rttGradient$ in algorithm) and adjusting the rate accordingly. If $\Delta RTT \leq 0$, indicating an empty or decreasing congestion queue, the rate is increased by δ. If $\Delta RTT > 0$, indicating an increase in queue length and potential congestion, the rate is reduced based on the gradient. The algorithm relies on additional parameters: T_{low} and T_{high}. T_{low} ensures stability by disregarding temporary queue length increases caused by segment-sized transmissions. T_{high} enables rate reduction when the queue exceeds a threshold, preventing it from becoming unlimited.

However, TIMELY lacks a stable convergence length for the congestion queue due to its criteria for rate acceleration and deceleration [20], [23]. This results in oscillations between the T_{low} and T_{high} thresholds, which are amplified when dealing with multiple incast flows. Unfairness arises when flows are transmitted at varying rates or times. Additionally, the algorithm exhibits a prolonged convergence time, especially when a large number of incast flows are sent in segments.

The patched TIMELY [23] addresses the limitations of TIMELY. The algorithm (in Algorithm 2) maintains similar handling for values below the T_{low} threshold and above the T_{high} threshold as in TIMELY. It introduces two key differences in rate adjustment when the measured RTT falls between T_{low} and T_{high}:

First, in the patched TIMELY, the rate reduction is determined based on the sign of ΔRTT, replacing the gradient with the current RTT as a reference. This modification addresses fairness concerns that arise when different flows with different rates experience the same ΔRTT. By utilizing each flow's individual RTT as the indicator for rate reduction during congestion, the algorithm adapts to varying RTT values and achieves convergence towards a common target RTT_{ref} of 50us. This approach ensures fair rate reduction among flows.

Algorithm 2 Patched TIMELY

$newRTTDiff = newRTT - prevRTT$
$prevRTT = newRTT$
$rttDiff = (1-\alpha) \cdot rttDiff + \alpha \cdot newRTTDiff$
$rttGradient = \frac{rttDiff}{D_{minRTT}}$
if $newRTT < T_{low}$ **then**
    $rate = rate + \delta$
else if $newRTT > T_{high}$ **then**
    $rate = rate \cdot (1 - \beta \cdot (1 - \frac{T_{high}}{newRTT}))$
else
    if $rttGradient \leq -0.25$ **then**
        $weight = 0$
    else if $rttGradient \geq 0.25$ **then**
        $weight = 1$
    else
        $weight = 2 * rttGradient + 0.5$
    end if
    $error = \frac{newRTT - RTT_{ref}}{RTT_{ref}}$
    $rate = \delta(1 - weight) + rate \cdot (1 - \beta \cdot error \cdot weight)$
end if

Second, the algorithm incorporates a weight function to facilitate smooth rate adjustments within the range of $\Delta RTT \in [-k, k]$, where k is set to 0.25. This feature prevents oscillations caused by abrupt rate changes. Importantly, when $\Delta RTT = 0$, the weight function assigns a weight of 0.5, effectively neutralizing rate adjustments and maintaining the current rate. The patched TIMELY aims to converge to a stable point with minimal oscillations, ensuring queue length stability. However, it is important to note that the weight function does not guarantee stability in practical scenarios involving numerous incast flows, although it does help mitigate oscillations to some extent.

TABLE I: Notations

R	Rate	g	RTT Gradient
q	Queue Length	t	Time
τ^*	Rate Update Interval	τ'	Measured Delay
N	Incast Ratio	α	EMA Smoothing Factor
C	Link Capacity	β	Rate Decrease Factor
δ	Rate Increase Factor	i	Flow Number
T_{low}	Low Threshold	T_{high}	High Threshold
D_{minRTT}	Delay	D_{prop}	Propagation Delay
seg	Segment Size		

B. Analysis of Patched TIMELY

In this section, we analyze the flow model using the patched TIMELY as an example, based on paper [23]. Besides, we further examine the applicability of the patched TIMELY

in practical environments [26], [27]. Based on this analysis, we summarize the issues with RTT-based congestion control algorithms.

We differentiate the rate adjustment formula in the patched TIMELY and consider the relationship between queue length and time, along with relevant functions. This allows us to derive the differential equation model for the patched TIMELY. The differential equation model is as follows (refer to Table I for the relevant parameters).

$$\frac{\mathrm{d}R_i}{\mathrm{d}t} = \begin{cases} \frac{\delta}{\tau*}, & q(t-\tau') < C*T_{low} \\ \frac{(1-w_i)\delta}{\tau*} - \frac{w_i\beta R_i(t)}{\tau*}\frac{q(t-\tau')-q'}{q'}, & \textit{Otherwise} \\ -\frac{\beta}{\tau*}(1-\frac{C*T_{high}}{q(t-\tau')})R_i(t), & q(t-\tau') > C*T_{high} \end{cases} \tag{1}$$

$$\frac{\mathrm{d}q}{\mathrm{d}t} = \sum_i R_i(t) - C \tag{2}$$

$$\tau^* = \max\{\frac{seg}{R_i}, D_{minRTT}\} \tag{3}$$

$$\tau' = \frac{q}{C} + \frac{MTU}{C} + D_{prop} \tag{4}$$

$$w_i = \begin{cases} 0, & g_i \le -0.25 \\ 2g_i + 0.5, & -0.25 < g_i < 0.25 \\ 1, & g_i \ge 0.25 \end{cases} \tag{5}$$

When the queue is stable, i.e. the queue length remains constant and stays within the range of T_{low} and T_{high}, otherwise, according to the algorithm, there must be either rate increase or rate decrease, and stability cannot be achieved. When the queue length is stable, the RTT measured in each round also remains constant, resulting in $\Delta RTT = 0$, which implies $g_i = 0$. Substituting this into Equation (3) yields $w_i = 0.5$. At this point, the rate of each flow no longer changes, therefore, both Equation (1) and Equation (2) equal 0, and we can solve for:

$$\begin{cases} \frac{\mathrm{d}R_i}{\mathrm{d}t} = \frac{(1-w_i)\delta}{\tau*} - \frac{w_i\beta R_i(t)}{\tau*}\frac{q(t-\tau')-q'}{q'} = 0 \\ \frac{\mathrm{d}q}{\mathrm{d}t} = \sum_i R_i(t) - C = 0 \end{cases} \Rightarrow \begin{cases} R_i = \frac{C}{N} \\ q^* = \frac{N\delta q'}{\beta C} + q' \end{cases} \tag{6}$$

This indicates that the differential equation has a trivial solution, which means that the algorithm can theoretically converge to a stable point where each flow shares the bottleneck link's bandwidth equally, and the queue length on the bottleneck link is given by (6).

By observing q^*, it is evident that as the number of incast flows N increases, the stable queue length increases proportionally with N. If N exceeds a certain threshold that causes the stable queue length q^* to be greater than q_{high} (where q_{high} is the queue length corresponding to the threshold T_{high}), according to the algorithm, when the measured $RTT > T_{high}$, the algorithm performs a rate reduction. However, when $RTT < T_{high}$, in order to reach a stable state, the algorithm will inevitably increase the rate, leading to a further increase in the queue length. This ultimately results in the algorithm's inability to stabilize, and the queue length oscillates around q_{high}. In Equation (6), let us assume $q^* \ge q_{high}$, then we can get:

$$\frac{N\delta q'}{\beta C} + q' \ge q_{high} \tag{7}$$

Furthermore, in a many-to-one traffic environment, the measured RTT on the bottleneck link satisfies $D + \frac{q}{C} = RTT$. Substituting this into Equation (7) and simplifying, we obtain:

$$N \ge \frac{(T_{high} - T_{low})\beta C}{\delta(T_{low} - D)} \tag{8}$$

Where D represents the link propagation delay, set to 5us in the simulation environment, and other relevant parameter values are $T_{high} = 500us$, $T_{low} = 50us$, $C = 10Gbps$, $\beta = 0.008$, and $\delta = 10M$. Substituting these values into Equation (8), we can determine that $N \ge 80$. This indicates that when $N = 80$, the congested queue length at the stable state is theoretically already reaching q_{high}. However, according to the patched TIMELY, the congested queue length cannot actually reach a stable state.

Additionally, in real-world environments, the feedback time of RTT itself is affected by the congested queue length. The longer the queue length, the more delayed the feedback time of RTT measured at the host. This causes the rate adjustment of the algorithm to always lag behind the occurrence of congestion, resulting in the congested queue length oscillating around the theoretical stable value instead of remaining consistently stable. In reality, when $N \ge 60$, the algorithm fails to reach a stable state and causes the maximum length of the oscillating queue to exceed T_{high}.

Compared to TIMELY, the patched TIMELY theoretically guarantees convergence to a stable point, where all flows fairly share the bottleneck link bandwidth and the congested queue length remains stable. However, this ideal state can only be achieved when there are fewer flows. It faces serious stability issues as the number of incast flows increases.

Based on the previous analysis, we can identify the following issues with RTT-based congestion control algorithms:1) Delayed feedback influenced by queue length: The sender may experience a significant delay in measuring RTT and adjusting the rate, potentially leading to congestion occurring in the network during this delay. Such delays contribute to oscillations in queue length. 2) Ineffective queue length control: Patched TIMELY theoretically ensures a stable queue length at convergence, in practical scenarios, the stable queue length increases proportionally with the number of incast flows. This increase in queue length exacerbates the delays in RTT feedback and intensifies queue oscillations.

The fundamental problem with RTT-based algorithms is the capability to control the congestion queue length within a low range. If the queue length can be controlled within a certain range regardless of the value of N and if RTT feedback delay does not increase with queue length, the algorithm would eventually reach a stable state.

III. DESIGN AND IMPLEMENTATION

This section proposes SQCC, an RTT based congestion control algorithm that can maintain stable congestion queue length fluctuations within a certain range. The main steps are shown in the Algorithm 3. SQCC has made two main improvements to address the issues in the patched TIMELY. First, we redesigned the speed reduction indicator function $error$, so that when a large number of flows are incast, the sender can respond quickly to speed reduction, as shown in line 18 of the Algorithm 3. Second, we designed automation parameters low and δ that vary with the number of cast flows N, significantly improving the oscillation range of the queue during stability, as shown in line 5 of the algorithm 3. At the same time, adjusting the δ parameter also reduces the length of the congestion queue when the algorithm is stable to a certain extent.

Algorithm 3 SQCC

1: $newRTTDiff = newRTT - prevRTT$
2: $prevRTT = newRTT$
3: $rttDiff = (1-\alpha)\cdot rttDiff + \alpha \cdot newRTTDiff$
4: $rttGradient = \frac{rttDiff}{D_{minRTT}}$
5: $T_{low} = \lfloor \lg N \rfloor \cdot \frac{seg}{C}, \delta = \frac{C}{N} \cdot \frac{k}{N}$
6: **if** $newRTT < T_{low}$ **then**
7: $rate = rate + \delta$
8: **else if** $newRTT > T_{high}$ **then**
9: $rate = rate \cdot (1 - \beta \cdot (1 - \frac{T_{high}}{newRTT}))$
10: **else**
11: **if** $rttGradient \leq -0.25$ **then**
12: $weight = 0$
13: **else if** $rttGradient \geq 0.25$ **then**
14: $weight = 1$
15: **else**
16: $weight = 2 * rttGradient + 0.5$
17: **end if**
18: $error = \frac{T_{high}(RTT_{cur} - T_{low})}{T_{low}(T_{high} - RTT_{cur})}$
19: $rate = \delta(1 - weight) + rate \cdot (1 - \beta \cdot error \cdot weight)$
20: **end if**

SQCC requires information on the current number of incast flows N for each adjustment. The receiver in SQCC writes the number of flows received on the current node N into the ACK each time it sends an ACK, and provides feedback to the sender. When the sender receives an ACK, they extract the N value, calculate new low and δ values, and then compare the current measured RTT value with the calculated low value to determine whether to accelerate or decelerate and use the new δ parameter and $error$ function accordingly. In practical environments, N can be estimated by the sender based on the current transmission rate, combined with the queue length reflected by the current measured RTT.

A. Design and Analysis of Speed Reduction Index Function

According to the analysis of the flow model, we found that the queue length at a stable state increases proportionally with the increase of the number of incoming flows N. Observing Equation (1), the queue length only appears as a variable in the deceleration indicator function $error$. Obviously, the queue length during stability is directly related to the setting of the $error$ function. This suggests that if we want to control the queue length, we need to redesign the $error$ function. The settings of the $error$ function in the Patched TIMELY are as follows:

$$error = \frac{newRTT - RTT_{ref}}{RTT_{ref}}, newRTT \in [T_{low}, T_{high}] \tag{9}$$

As the number of competing flows increases, the number of segments that simultaneously arrive on the switch queue at a certain time also increases, and the queue length and RTT generated by collisions also increase. The $error$ function also increases proportionally with the measured RTT, and the rate of each flow correspondingly decreases even more. When $RTT = T_{high}$, the $error$ function takes the maximum value. This means that the algorithm has an upper limit on the degree of rate reduction each time, which makes it difficult to quickly slow down to reduce queue length when there are many flows. If the $error$ function can tend towards infinity, the rate drops quickly enough, and the data packets on the congested queue are emptied quickly enough, the length of the congested queue can be controlled to fluctuate only within a certain range.

B. Derivation of Error Function

Assuming the current measured RTT value is RTT_{cur}, the corresponding queue length is q_{cur}, the current rate of each flow is R_{cur}. Correspondingly, assuming that after the algorithm adjustment, the corresponding values are $RTT_{exp}, q_{exp}, R_{exp}$, the propagation delay of the link is D, and the link capacity is C. Assuming that the packet processing delay on the host side and the packet forwarding delay on the switch port in the network are both very small, and there is no packet loss caused by packet verification errors or link disconnects, the RTT measured each time on the host side and the current queue length meet the following relationship:

$$\begin{cases} D + \frac{q_{cur}}{C} = RTT_{cur} & \text{before adjustment} \\ D + \frac{q_{exp}}{C} = RTT_{exp} & \text{after adjustment} \end{cases} \tag{10}$$

By subtracting the two equations in Equation (10), it can be obtained that

$$\Delta q = q_{cur} - q_{exp} = C(RTT_{cur} - RTT_{exp}) \tag{11}$$

Because the length of the congested queue within each round of RTT is the portion of the data sent by all flows that exceeds the link capacity, the change in length of the congested queue between each round of RTT (Δq) approximately follows the following relationship:

$$\Delta q = \sum_i R_{cur} * RTT_{cur} - \sum_i R_{exp} * RTT_{exp} - C(RTT_{cur} - RTT_{exp}) \tag{12}$$

The effective part of algorithm speed reduction is $R_{exp} = R_{cur}(1 - error)$, substitute this into Equation (12), and subtract equations (11) and (12) to obtain

$$2C(RTT_{cur} - RTT_{exp}) = \sum_i R_{cur}(RTT_{cur} - RTT_{exp}) + \sum_i R_{cur} \cdot error \cdot RTT_{exp} \quad (13)$$

$$\sum_i R_{cur} \cdot error \cdot RTT_{exp} = (RTT_{cur} - RTT_{exp})(2C - \sum_i R_{cur})$$

$$error = \frac{RTT_{cur} - RTT_{exp}}{RTT_{exp}} \left(2\frac{C}{\sum_i R_{cur}} - 1 \right)$$

When the rate of each flow oscillates near the stable point $\frac{C}{N}$, $2\frac{C}{\sum_i R_{cur}} - 1 \approx 1$, the final $error$ is

$$error = \frac{RTT_{cur} - RTT_{exp}}{RTT_{exp}} \quad (14)$$

In Equation (14), if the algorithm adjusts the expected RTT after each round of measurement to $RTT_{exp} = RTT_{ref}$, then Equation (14) is a $error$ function repaired in time. According to the analysis in the previous section, when the $error$ function works, there must be $RTT_{cur} \in [T_{low}, T_{high}]$ and timely repair is equivalent to setting a fixed RTT_{exp} to get the $error$ function with a fixed value range. If the RTT to be measured increases within the above range, the $error$ function does not take the value within the limited range of the maximum value, but can tend to infinity, as shown in the following relationship:

$$RTT_{cur} \to T_{low} \quad error \to 0 \Rightarrow RTT_{exp} \to T_{low}$$
$$RTT_{cur} \to T_{high} \quad error \to \infty \Rightarrow RTT_{exp} \to 0$$

So, when the congestion queue is very long, the algorithm can obtain a larger deceleration ratio through the $error$ function, which can quickly reduce the rate to empty the queue. Perform a simple linear fitting for RTT_{exp} and RTT_{cur} to obtain the following relationship (15).

$$RTT_{exp} = \frac{T_{low}(T_{high} - RTT_{cur})}{T_{high} - T_{low}} \quad (15)$$

Put Equation (15) into Equation (14), and we get

$$error = \frac{T_{high}(RTT_{cur} - T_{low})}{T_{low}(T_{high} - RTT_{cur})} \quad (16)$$

Equation (16) is the final $error$ function, and it meets

$$RTT_{cur} \in [T_{low}, T_{high}] \qquad error \in [0, +\infty]$$

C. Proof of Effectiveness of Error Function

By substituting Equation (16) into the flow model of the patched TIMELY, the equation for rate adjustment within the range of T_{low} and T_{high} changes. The remaining parts of the algorithm remain unchanged. By setting this equation to zero, we can determine the stable length of the congested queue.

$$\frac{\mathrm{d}R_i}{\mathrm{d}t} = \frac{(1 - w_i)\delta}{\tau*} - \frac{w_i \beta R_i(t)}{\tau*} \frac{q_{high}(q(t - \tau') - q_{low})}{q_{low}(q_{high} - q(t - \tau'))} = 0$$

$$q^* = \frac{\delta N q_{low} q_{high} + \beta C q_{low} q_{high}}{\delta N q_{low} + \beta C q_{high}} \quad (17)$$

$$= q_{high} - \frac{\beta C q_{high}(q_{high} - q_{low})}{\delta N q_{low} + \beta C q_{high}} \quad (18)$$

$$\lim_{N \to 0} q^* = q_{high} - (q_{high} - q_{low}) = q_{low} \qquad \lim_{N \to \infty} q^* = q_{high}$$

As the number of incoming flows increases, we can determine the theoretical range of the stable congestion queue length by taking the limit of Equation (18), which is $q^* \in (q_{low}, q_{high})$. This means that as the number of cast flows increases, the theoretically stable congestion queue length also increases, but it should not exceed the threshold of T_{high}.

$$q^*_{patched} - q^*_{new} = \frac{q^2_{low} \delta N(\delta N + \beta C)}{\beta C(\delta N q_{low} + \beta C q_{high})} = \mathcal{O}(N)$$

The new $error$ function consistently reduces the congestion queue length compared to the patched TIMELY. As the number of incoming flows increases, this difference widens, effectively controlling queue length. The modified $error$ function ensures algorithm effectiveness by rapidly reducing the sender's rate when the RTT is large, and preventing the $error$ value from becoming infinitely large as the queue empties.

D. Analysis of Queue Oscillation Range during Stability

By modifying the $error$ function, the algorithm theoretically has the ability to control the queue length during stability between (q_{low}, q_{high}) and gradually approach q_{high} as the number of incast flows increases. However, in practice, as the number of flows increases, the queue length during stability does not remain fixed at q^* but oscillates around q^*, and the oscillation becomes larger with more flows. There are two main reasons for this:

First, as the number of incast flows N increases, during stability $RTT \to T_{high}$ and $q^* \to q_{high}$. The feedback delay of RTT at the sender is still significant due to the long congestion queue length. By the time the algorithm adjusts, congestion has already occurred for a certain period of time, during which the rate remains unchanged. Moreover, for multiple different senders, such delays can cause different flows to adjust their rates at different times, leading to oscillations in the rates of each flow and causing congestion queue oscillation.

Second, observing Equation (1), if the $error$ function is directly substituted, we can obtain the value of the $error$ function during stability.

$$\frac{\mathrm{d}R_i}{\mathrm{d}t} = \frac{(1 - w_i)\delta}{\tau*} - \frac{w_i \beta R_i(t)}{\tau*} error(RTT_t) = 0 \quad (19)$$

Equation (19) indicates that as the number of incast flows N increases, the value of the error function during stability should increase proportionally with N. This also explains why the patched TIMELY cannot tolerate a large number of incast flows. A very large error value will cause a sharp decrease in the rate of each flow. Although each flow receives a very

small bandwidth during heavy incast, the collective effect on the congestion queue will also cause a sharp decrease in queue length and RTT. This rapid reduction in error values combined with the subsequent acceleration phase of the algorithm, where many flows simultaneously accelerate by a fixed value δ, leads to multiple flow packets arriving at the queue simultaneously, causing congestion once again. This repetitive process results in oscillation of the queue length.

Based on the analysis above, the value of the $error$ function during stability increases proportionally with N at a rate of $\frac{\delta}{\beta C}$. This results in a large error value, causing the queue length to oscillate within a wide range. It is evident that reducing the value of the $error$ function is necessary to decrease the oscillation range of the queue during stability. It is important to note that lowering the error value in this context is different from increasing the error value discussed in the previous section. As RTT increases, the $error$ function should increase more rapidly to decrease the rate.

Adjustment of Rate Increment. To adjust the rate increment δ, we need to consider the following trade-off: if δ is too small, the rate will decrease gradually, resulting in a long period of empty queues and low link utilization; if δ is too large, especially when the number of incast flows N is large and each flow receives a bandwidth of $\frac{C}{N}$ (where C is the link capacity), the rate increment will exceed the rate before adjustment, causing rate oscillation and instability. For simplicity, we focus on a single oscillation near the stability point.

Let N be the number of incast flows, and each flow receives a stable bandwidth of $\frac{C}{N}$. The time interval for rate adjustment is denoted as τ. Assuming there are no packet losses due to link errors or other factors, a single rate change near the stability point should satisfy the following relationship:

$$N\left(\left(\frac{C}{N}+\delta\right)\tau+\left(\frac{C}{N}+\delta\right)(1-\beta\cdot error)\tau\right)-C\cdot 2\tau \geq C\cdot T_{low} \tag{20}$$

In this equation, $(\frac{C}{N}+\delta)$ represents the maximum rate during one queue oscillation, while $(\frac{C}{N}+\delta)(1-\beta\cdot error)$ represents the rate after rate reduction based on RTT measurement. The weight function in patched TIMELY is not considered here for simplicity. Each flow maintains its original rate for the duration of the rate update interval τ before rate adjustment occurs.

The left side of Equation (20) represents the remaining queue length after one oscillation when the rates of N flows are accelerated and decelerated. It should not be smaller than the queue length corresponding to the T_{low} parameter to prevent excessive oscillations and empty queues. The T_{low} parameter is determined by the segment size, τ, and the relationship between τ and $error$ during algorithm stability is given by:

$$error=\frac{\delta N}{\beta C},\quad \tau=\max\left(\frac{seg}{R^*}=\frac{seg}{C/N}, minRTT\right),\quad low=\frac{seg}{C}$$

By substituting these equations into Equation (20) and simplifying, we can obtain a quadratic inequality in terms of δ. Solving this inequality will yield a relationship for δ.

$$N^3\delta^2-N^2C\delta+C^2\leq 0$$
$$\frac{C}{2N}\cdot\frac{N-\sqrt{N(N-4)}}{N}\leq\delta\leq\frac{C}{2N}\cdot\frac{N+\sqrt{N(N-4)}}{N} \tag{21}$$

Furthermore, when N is large, $\sqrt{N(N-4)}\approx N-2$. Substituting this approximation into Equation (21), we get:

$$\frac{C}{N}\cdot\frac{1}{N}\leq\delta\leq\frac{C}{N}\cdot\frac{N-1}{N}\approx\frac{C}{N} \tag{22}$$

To prevent excessive rate acceleration and oscillations, it is generally recommended to set δ smaller than the per-flow fair share bandwidth $\frac{C}{N}$. Referring to Equation (18), a smaller δ corresponds to a smaller queue length in the stable state. As a result, a suitable choice for δ is $\delta=\frac{C}{N}\cdot\frac{1}{N}$.

Control of the Acceleration Phase. To achieve a relatively stable state in the algorithm, simply adjusting the δ parameter is insufficient, especially when dealing with a large number of incast flows. In such cases, modifying T_{low} becomes necessary to differentiate between the need for a long queue length in the stable state and the rapid increase caused by a high number of flows.

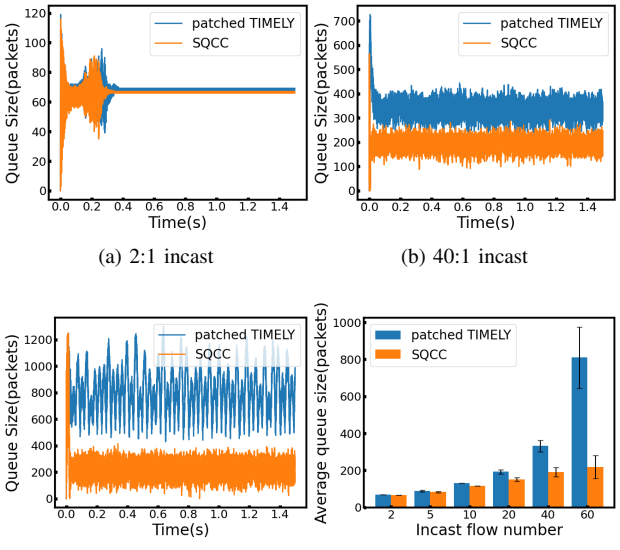
Increasing T_{low} offers several benefits. It reduces fluctuations in the sending rate by generating smaller $error$ values for larger N. It helps the algorithm reach a relatively stable state by allowing the sending rate to decrease to its minimum value and providing sufficient time for acceleration accumulation. Moreover, a single T_{low} parameter can accommodate various numbers of flows, ensuring stability in dynamic environments to some extent.

If we conduct simulation and fitting, we can find that setting

$$low=\frac{seg}{C}\cdot\lfloor\lg N\rfloor \tag{23}$$

In order to meet the requirements, we can adjust the value of δ. As N increases, the queue can tolerate more segment collisions that lead to congestion. However, since the derivation of the δ parameter involves the T_{low} parameter, the δ parameter needs to change accordingly as T_{low} changes. Let's assume that as N changes, $T_{low}=k\cdot\frac{seg}{C}$ (where $k=1,2,3$). Substituting this into Equation (20) and simplifying, we can obtain a quadratic inequality in terms of δ. Solving this inequality gives us:

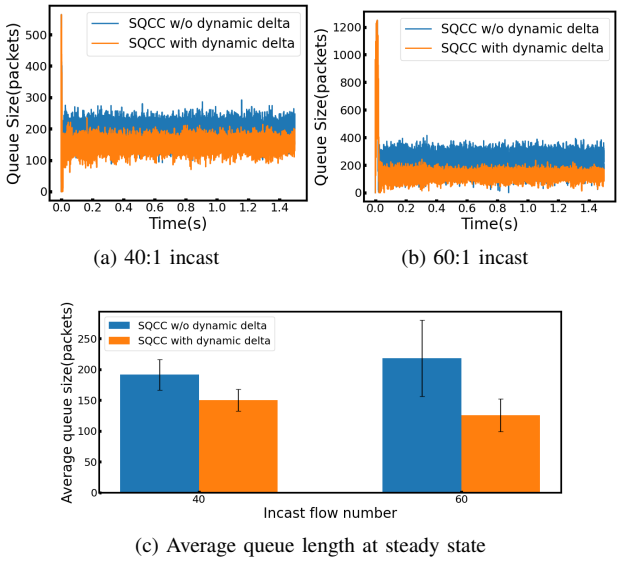
$$N^3\delta^2-N^2C\delta+C^2k\leq 0$$
$$\frac{C}{2N}\cdot\frac{N-\sqrt{N(N-4k)}}{N}\leq\delta\leq\frac{C}{2N}\cdot\frac{N+\sqrt{N(N-4k)}}{N} \tag{24}$$



(a) 2:1 incast (b) 40:1 incast

(c) 60:1 incast (d) Queue Length

Fig. 1: Comparison of the effect of modifying the error function



(a) 40:1 incast (b) 60:1 incast

(c) Average queue length at steady state

Fig. 2: Dynamically adjust the effect comparison of δ

Furthermore, when N is large, the value of k tends to be small, and we can approximate $\sqrt{N(N-4k)}$ as $N-2k$. Substituting this approximation into Equation (24), we obtain:

$$\frac{C}{N}\cdot\frac{k}{N}\leq\delta\leq\frac{C}{N}\cdot\frac{N-k}{N}\approx\frac{C}{N}$$

Similarly, the final expressions for T_{low} and the corresponding δ are given by:

$$k=\lfloor\lg N\rfloor\quad T_{low}=k\cdot\frac{seg}{C}\quad\delta=\frac{C}{N}\cdot\frac{k}{N}\tag{25}$$

IV. EVALUATION

We compared the performance of the patched TIMELY and SQCC in the NS3 simulation environment to evaluate the improvement of SQCC. Other RTT-based schemes like Swift [19] is tightly coupled with the protocol stack. The dumbbell network topology was used with 10 Gbps link capacities and 1 us propagation delay per link. Other relevant parameters were consistent with the patched TIMELY, where α, β and T_{high} is set to 0.875, 0.008 and $500\mu s$, respectively. The initialized value of T_{low} and δ is $50\mu s$ and $10M$, respectively. Metrics examined were the average congestion queue length and queue oscillation range during stability. These comparisons assessed the effectiveness of SQCC in achieving stable and efficient congestion control.

A. Comparison of Error Functions

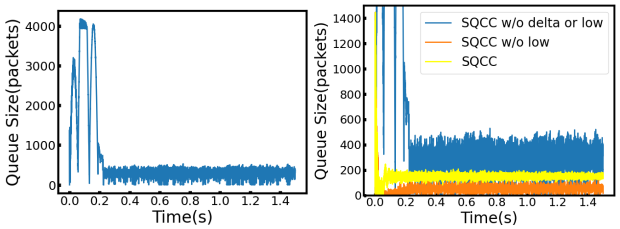
Figure 1 compares the variation of queue length over time for 2/40/60:1 incast scenarios using the patched TIMELY and SQCC with only the new $error$ function without any other optimizations. The blue lines or bars represent the patched TIMELY, while the orange lines represent SQCC. It can be observed that as the number of incast flows N increases, SQCC consistently maintains lower queue lengths compared to the patched TIMELY. As shown in Figure 1d, the average queue length during stability can be reduced by nearly 80%, achieved when $N=60$. Additionally, the queue oscillation range during stability is also lower for SQCC compared to the patched TIMELY. Figure 1c illustrates that when the number of incast flows is 60, the patched TIMELY fails to maintain a stable queue, resulting in significant oscillations around the $T_{high}=500us$ threshold. The key factor contributing to these improvements is the new $error$ function.

B. Comparison of Oscillation Range

Figure 2 compares the queue lengths with and without adjusting the δ parameter using the redesigned $error$ function. The blue line/bar represents the queue length with only the modified error function, while the orange line/bar represents the effect after incorporating the δ adjustment. For $N=40$, $\delta=6.25M$, and for $N=60$, $\delta=2.7M$ according to (21).

Figures 2a and 2b display the queue length variation over time for 40:1 and 60:1 incast scenarios. SQCC with adjusted δ parameter further reduces the queue length and oscillation range in the stable state. Figure 2c compares the average queue lengths in the stable state, with vertical lines indicating the range of queue length oscillation. In the 40/60:1 incast scenarios, compared to SQCC without modifying δ, SQCC with adjusted δ reduces average queue length by 21.5% and 42% respectively, and decreases oscillation range by 27% and 57%.

Adjusting the parameter of δ has a greater impact with a higher number of incast flows. When more flows arrive simultaneously, SQCC's error function engages more flows in acceleration and deceleration, leading to better results. By setting (22) to 10M and solving for N, we find that adjusting the δ function significantly improves performance when $N\geq 40$. However, for smaller N values, SQCC's error function already maintains low queue lengths effectively.



(a) 500:1 incast effect when only modifying the error function

(b) 500:1 incast parameter optimization effect comparison

Fig. 3: Comparison of dynamic adjustment parameters

TABLE II: 500:1 incast Average queue length at steady state

Scheme	$T_{low}(us)$	$\delta(M)$	Queue Length(pkt)	Standard Deviation(pkt)
W/O δ & T_{low}	50	10	272.77	122.38
W/O T_{low}	50	0.04	53.96	20.55
SQCC	100	0.08	149.3	13.73

C. Comparison of the Effectiveness of the T_{low} Parameter

Figure 3 illustrates the effects of various optimizations in SQCC using the scenario of 500:1 incast. Figure 3a represents the result of modifying only the $error$ function, corresponding to the blue line in Figure 3b.

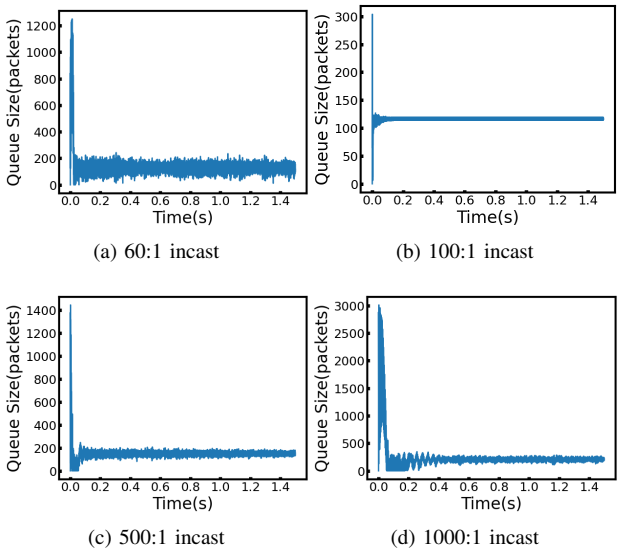
It can be observed that when there are multiple incast flows, modifying only the $error$ function is not sufficient to effectively reduce the queue length. The congestion queue length oscillates between 0 and 400 packets, leading to frequent empty queues and low link utilization. Additionally, modifying only the $error$ function does not reduce the convergence time of the algorithm since the value of $\delta = 10M$ is relatively large, requiring multiple RTTs before the rate decreases.

In Figure 3b, the orange line represents the queue behavior after adjusting $\delta = 0.04M$ in addition to modifying the error function. However, although the algorithm maintains a low queue length, the link utilization is not high, and the queue frequently becomes empty. The yellow line, on the other hand, represents the queue behavior after adjusting $T_{low} = 100us$ and $\delta = 0.08M$ based on modifying the error function. This combined adjustment of T_{low} and δ ensures a stable state with a low queue length and reduced oscillation compared to the previous optimizations.

Table II summarizes the average queue length and standard deviation during the stable state for the three optimization methods. It shows that adjusting T_{low} and δ leads to a stable and minimally oscillating queue.

D. Large Scale Incast Experiment

Figure 4 shows the variation of queue length over time for large flow incasts with different ratios (60/100/500/1000:1). SQCC is applied with the error function defined in Equation (16) and the values of T_{low} and δ set according to Equation (25). It reveals that regardless of the value of N, the queue length during the stable phase remains within the range of 100-200 packets and exhibits only small oscillations within a range of 1-25 packets. For N is less than 1000, SQCC achieves a balance between queue length and link utilization by adjusting T_{low} and δ.



(a) 60:1 incast

(b) 100:1 incast

(c) 500:1 incast

(d) 1000:1 incast

Fig. 4: Queue length changes in a large number of flow incast environments

V. RELATED WORK

Traditional TCP congestion control algorithms are not well-suited for data centers due to their reliance on packet loss and slow convergence. New congestion control algorithms have been proposed, categorized into special switch-based, receiver-based, and sender-side feedback approaches. Special switch-based algorithms like pFabric [28], PDQ [29], and D3 [30] prioritize traffic for rate allocation but require costly customizations and lack scalability. Floodgate [31] and BFC [32] design a switch-based per-hop per-flow flow control to absorb incasts, revising much logic of switches. HPCC [33] leverages INT to measure the precise link load, requiring the middle node of network (switches, routers, etc.) to support the INT technology.

ECN-based algorithms use explicit congestion notification markings to control congestion, such as DCTCP [15] and D2TCP [16]. However, ECN-based approaches face challenges in parameter tuning, consistency, and scalability [34], [35]. RTT-based algorithms like TIMELY, Patched TIMELY, and Swift [19] use RTT information to detect congestion. Swift relies on a custom network protocol stack to achieve fine-grained control over packet transmission and reception, enabling precise RTT measurements and dynamic adjustment of the RTT threshold to reduce queue length. However, this approach is often challenging to deploy on a large scale in different network environments [20]. Patched TIMELY and Swift improve stability and achieve low queue lengths, but Swift is tightly coupled with the protocol stack, making deployment difficult. Inaccurate RTT measurements impact Swift's performance. PowerTCP [21] utilizes both ECN and delay to detect congestion.

VI. CONCLUSION

In response to the issue of unstable queue length and limited congestion control capability of the patched TIMELY in the scenario of large-scale traffic incast, we have designed a new $error$ function and adjusted the parameters T_{low} and δ, proposing SQCC. Extensive and in-depth simulation experiments conducted on the analysis of the patched TIMELY and the implementation of SQCC have demonstrated that the congestion queue length remains stable within the range of $[q_{low}, q_{high}]$. Additionally, when the number of flows N is less than 1000, we are able to maintain the queue length within a very narrow range, resulting in minimal oscillation in the stable state.

VII. ACKNOWLEDGEMENT

We gratefully appreciate the feedback from anonymous reviewers. This work was partially supported by National Natural Science Foundation of China under Grant Nos. 62172204. Collaborative Innovation Center of Novel Software Technology and Industrialization.

REFERENCES

[1] V. Mauch, M. Kunze, and M. Hillenbrand, "High performance cloud computing," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1408–1416, 2013.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI)*, 2016.

[3] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *Statistics*, 2015.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[5] C. Galakatos and T. K. E. Zamanian, "The End of Slow Networks: It's Time for a Redesign," *Proceedings of the VLDB Endowment*, vol. 9, no. 7, 2016.

[6] A. Dragojević, D. Narayanan *et al.*, "FaRM: Fast Remote Memory," in *Proceedings of USENIX NSDI*, 2014.

[7] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *Proceedings of USENIX ATC*, 2013.

[8] Y. Taleb, R. Stutsman *et al.*, "Tailwind: Fast and Atomic RDMA-Based Replication," in *Proceedings of Annual Technical Conference (ATC)*, USA, 2018.

[9] M. Wu, F. Yang, J. Xue *et al.*, "Gram: Scaling graph computation to the trillions," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[10] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.

[11] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang *et al.*, "From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud," in *Proceedings of SIGCOMM*, 2022.

[12] N. Dukkipati, "Rate control protocol (RCP): congestion control to make flows complete quickly," 2008.

[13] ——, "RCP: Congestion control to make flows complete quickly," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, 2006.

[14] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, 2002.

[15] 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 SIGCOMM*. ACM, 2010.

[16] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 115–126, 2012.

[17] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "TIMELY: RTT-based Congestion Control for the Datacenter," in *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

[18] Y. Zhu, H. Eran, D. Firestone *et al.*, "Congestion control for large-scale RDMA deployments," *ACM SIGCOMM Computer Communication Review*, 2015.

[19] G. Kumar, N. Dukkipati *et al.*, "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter," in *Proceedings of ACM SIGCOMM*. New York, NY, USA: Association for Computing Machinery, 2020.

[20] J. Tang, T. Xu, C. Nguyen, X. Wang, S. Lu, and B. Ye, "Tuning Target Delay for RTT-based Congestion Control," in *30th International Conference on Network Protocols (ICNP)*. IEEE, 2022.

[21] V. Addanki, O. Michel, and S. Schmid, "PowerTCP: Pushing the performance limits of datacenter networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[22] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized" zero-queue" datacenter network," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 307–318.

[23] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye, "ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2016.

[24] J. Gettys, "Bufferbloat: Dark buffers in the internet," *IEEE Internet Computing*, vol. 15, no. 3, pp. 96–96, 2011.

[25] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[26] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: stability, convergence, and fairness," *ACM SIGMETRICS Performance Evaluation Review*, vol. 39, no. 1, pp. 73–84, 2011.

[27] F. Golnaraghi and B. C. Kuo, *Automatic control systems*. McGraw-Hill Education, 2017.

[28] 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*, vol. 43, no. 4, pp. 435–446, 2013.

[29] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 127–138, 2012.

[30] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 50–61, 2011.

[31] K. Liu, C. Tian, Q. Wang, H. Zheng, P. Yu, W. Sun, Y. Xu, K. Meng, L. Han, J. Fu *et al.*, "Floodgate: Taming incast in datacenter networks," in *Proceedings of the 17th International Conference on emerging Networking Experiments and Technologies*, 2021, pp. 30–44.

[32] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson, "Backpressure flow control," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA, Apr. 2022.

[33] Y. Li, R. Miao, H. H. Liu *et al.*, "HPCC: High Precision Congestion Control," in *Proceedings of SIGCOMM*. New York, NY, USA: ACM, 2019.

[34] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *Proceedings of SIGCOMM*. ACM, 2015.

[35] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "ACC: Automatic ECN tuning for high-speed datacenter networks," in *Proceedings of ACM SIGCOMM*, 2021.