Mitigating Inter-datacenter Incast with a Proxy

The shortest path is not necessarily the fastest

Anchengcheng Zhou
Princeton University
ann.zhou@princeton.edu

Carter Costic
Princeton University
cartercostic@princeton.edu

Hongyu Hè
Princeton University
hhy@g.princeton.edu

Ahmad Ghalayini
Microsoft
aghalayini@microsoft.com

Abdul Kabbani
Microsoft
abdulkabbani@microsoft.com

Maria Apostolaki
Princeton University
apostolaki@princeton.edu

Abstract

Many-to-one communication (*i.e.,* incast) is a long-standing challenge in networking with a wide range of proposed solutions. However, as incast-inducing applications today (*e.g.,* storage, ML training) scale beyond a single datacenter, they introduce new challenges that current solutions do not handle. In particular, inter-datacenter links have orders of magnitude higher latency than intra-datacenter paths, lengthening the feedback loop that senders rely on to adjust their sending rates and drastically increasing incast completion times.

To reduce inter-datacenter incast latency, we propose adding a proxy server in the sending datacenter to relay traffic between the senders and the receiver. Surprisingly, adding this extra hop reduces incast latency! The insight is that the added hop shifts the congestion point closer to the senders, shortening the feedback loop and allowing senders to converge quickly at a rate that fully utilizes the link while avoiding severe congestion. Motivated by preliminary results, we investigate low-overhead proxy designs and explore ways to expose the proxy as a broader optimization opportunity for application developers, cloud operators, and tenants.

CCS Concepts

• **Networks** → **Network protocol design**.

Keywords

Inter-datacenter Incast, Proxy, Congestion Feedback, Routing, Datacenter Networks

ACM Reference Format:
Anchengcheng Zhou, Carter Costic, Hongyu Hè, Ahmad Ghalayini, Abdul Kabbani, and Maria Apostolaki. 2025. Mitigating Inter-datacenter Incast with a Proxy: The shortest path is not necessarily the fastest. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3772356.3772410

HotNets '25, College Park, MD, USA

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772410

1 Introduction

Incast, the communication pattern where multiple senders transmit simultaneously to a single receiver, is common in datacenters and particularly problematic because it can quickly overwhelm the network, causing congestion and severely degrading the performance of critical applications [18, 52, 57]. To address this, the research community has developed a wide range of solutions tailored to different scenarios, from absorbing incast bursts in deep buffers to refining loss detection heuristics and upgrading the RTO timer accuracy to avoid false retransmission [20, 49, 57, 70].

While these solutions help alleviate the challenges of intra-datacenter incast, they fall short in the case of inter-datacenter incast, where the senders and the receiver reside in different datacenters. Inter-datacenter communication involves significantly higher latencies—often orders of magnitude greater than intra-datacenter traffic—which in turn severely delays the feedback loop that senders rely on to adjust their sending rates to the bottleneck link. More specifically, senders lack coordination and hence must independently adjust their sending rates to the available bandwidth based on feedback from the network. The effectiveness of this adjustment depends on the round-trip time (RTT) between each sender and the receiver. The longer this feedback loop, the slower the convergence to suitable sending rates that in-aggregate match the bottleneck bandwidth, leading to prolonged incast durations in inter-datacenter settings.

Despite their detrimental impact, inter-datacenter incasts have received little attention, potentially because they were previously considered unlikely to occur in practice, as applications were typically confined to a single datacenter. However, as applications and cloud infrastructure continue to scale, it is increasingly common for workloads to span across different datacenters, inevitably creating inter-datacenter

incasts. Our collaboration with a major cloud provider confirmed that inter-datacenter incasts are not merely a theoretical concern but are actively impacting performance in production environments. Importantly, "different datacenters" does not necessarily imply large geographic separation. Cloud providers often deploy multiple datacenters connected via long-haul links within the same metropolitan area or region. This allows them to scale infrastructure without modifying the core network topology or altering the associated routing, management, and fault domains. While one might assume that providers could prevent inter-datacenter incasts simply by placing senders and receivers in the same datacenter, this is ineffective and sometimes infeasible in practice: enforcing such restrictions would further constrain the already complex job allocation process, and cloud providers often lack full visibility into the workload behavior of clients. Moreover, nodes may need to reside in different datacenters for compliance and reliability reasons.

This paper presents a somewhat surprising proposition: we can mitigate the impact of inter-datacenter incast by intentionally routing traffic through a fake receiver (or a proxy) located within the sending datacenter. At first glance, this may seem counterintuitive. Lengthening the path each packet takes should impair, not help, the completion time of the incast-inducing job. However, the proxy directly addresses the core challenge of inter-datacenter incast, namely the long feedback loop which slows down convergence.

We validate this idea in simulation and find that adding a proxy hop can potentially reduce incast completion time by 70.60% and 53.60% on average across various incast degrees and sizes. The reduction in incast completion time becomes more pronounced as the RTT difference between intra- and inter-datacenter communications increases. This reduction in incast completion time is expected: relocating the congestion point closer to the senders decreases the feedback delay and enables the senders to converge faster to the appropriate sending rates to fill (but not overwhelm) the bottleneck link. To effectively realize the benefit of the proxy hop, loss must be detected and signaled to the senders as if the proxy were the receiver (*i.e.,* we should not wait for the actual receiver to detect and signal the loss to the senders). Crucially, early loss detection at the proxy can leverage streamlined logic implemented entirely in eBPF, keeping the processing overhead low.

Our findings open up a rich research agenda that challenges how we route traffic, design applications, and allocate resources. Instead of asking how to avoid incast, we may ask how to rate-limit traffic in an application-agnostic way earlier in the path. Rather than designing application communication patterns independently of server placement, we may explicitly select a proxy to flatten or reshape traffic through the application code or in the compiler when servers are allocated across datacenters. Finally, rather than tightly colocating containers of the same application within a single datacenter and relocating them when scaling is needed, we might instead allow new containers to be placed in a different datacenter—repurposing an existing container as a proxy.

2 Motivation

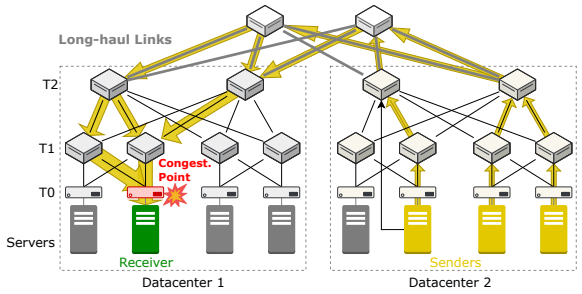
Incast emerges across datacenters as applications scale. Incast traffic is prevalent in applications ranging from traditional distributed storage [52] and real-time services *e.g.,* web search and recommendation systems [9], to modern distributed ML training. As these applications scale, inter-datacenter incast over long-haul links becomes increasingly common. For example, ***ML training*** produces challenging incast traffic. In Mixture-of-Experts models [23, 24, 43], tokens are routed by a gating function to experts sharded across devices. The ensuing dispatch and combine phases are implemented as all-to-all exchanges, so each expert simultaneously receives inputs from many senders, effectively creating multiple concurrent incasts. As large-scale training jobs grow across multiple datacenters [7, 13], inter-datacenter incast becomes an especially relevant problem. Likewise, ***storage systems*** also generate incast traffic. For example, when an erasure-coded fragment is requested by a user but unavailable due to a failure, the orchestrator needs to read other fragments from different servers to reconstruct this fragment, hence creating an incast [11, 31]. Other examples include strongly consistent geo-replicated storage systems that synchronize writes across a quorum of replicas [21, 58]. As storage systems span across datacenters [11] and regions [1–3], dealing with inter-datacenter incast is especially pertinent.

Inter-datacenter incast is uniquely challenging due to the long feedback loop. An inter-datacenter flow, where the sender and the receiver are in different datacenters, incurs millisecond-level RTTs as packets traverse the long-haul links (as opposed to microsecond-level RTTs intra-datacenter). In inter-datacenter incast, congestion builds up in the receiving datacenter where flows converge, so network feedback is delayed by several milliseconds between the congestion bottleneck and the senders. This long feedback loop is detrimental to incast latency. Being able to receive network feedback promptly is crucial for senders to adjust their sending rates to quickly mitigate congestion or link under-utilization. But a long feedback loop keeps the senders trapped at rates that are either too slow or too aggressive. Slow sending rates waste available bandwidth and make the incast completion time unnecessarily long. On the other hand, persisting in sending too aggressively, even just for a few milliseconds, can severely overload the network. Such aggressiveness is not rarely seen in incast senders that are eager to push out all traffic and thus set their initial sending

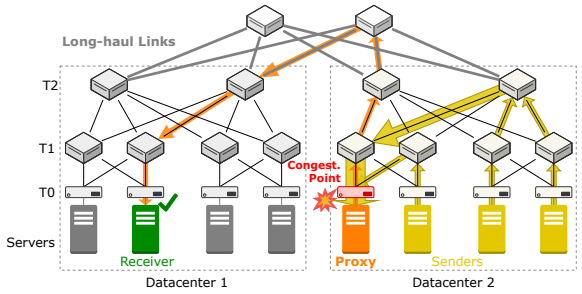
rates proportional to BDP [28, 51]. Hence, they can severely congest the network just with their first-RTT traffic.

Existing work does not address the long feedback delay. Incast literature has primarily focused on intra-datacenter incast where the feedback loop is reasonably short [17–19, 52, 57, 74]. Large switch buffers may avoid congestion from intra-datacenter incasts altogether, but are not viable in an inter-datacenter scenario that requires even larger buffers, which are expensive to build [57], difficult to design [59] and lead to low and unstable performance [5, 6, 36]. Therefore, many solutions tackle incast by modifying the retransmission timeout mechanism. They either refine loss pattern matching to avoid unnecessary timeouts [20, 25, 49] or reduce the penalty from incurring timeouts, *e.g.,* by using more fine-grained RTO [10, 18, 19, 32, 33, 55, 62, 70]. These solutions are useful for inter-datacenter incast too, but are inadequate on their own because they do not directly handle the long feedback loop, *i.e.,* the main culprit for performance degradation in inter-datacenter incast. Congestion-control-based solutions [12, 26, 71] propose more radical changes to their mechanisms such that the receiver can relay additional incast information to senders and help them adapt their sending rates more precisely after the first RTT. These approaches do not address the long feedback loop fundamentally, and consequently, severe congestion upon first-RTT traffic or changing available bandwidth persists. Floodgate [46] identifies incast earlier by keeping per-destination count on switches and mitigates by distributing credits for the windows, which require special switching hardware.

Cross-datacenter literature has given limited attention to reducing incast latency. Traffic engineering solutions do not always optimize for latency [4, 14, 16, 29, 30, 34, 35, 38, 40, 45, 63–67, 75] and when they do, they typically optimize how traffic is split across paths. For example, [42, 47, 50, 56, 72] compute path assignments based on historical traffic aggregated over a certain time interval. But path optimization does not prevent the congestion bottleneck at the receiver down-ToR, and the traffic aggregation could mask incast traffic patterns altogether. On the other hand, [22, 37, 41] make per-flow decisions on how to schedule and route the flow, but incur significant overhead on the critical path when solving the route optimization. Congestion-control-based solutions do not address the core issue of the long feedback loop either. Gemini [73] employs milder congestion window reduction for longer-RTT flows, thus avoiding link underutilization, but overlooks the more severe issue of network overload when windows are too large. Annulus [61] only handles bottlenecks that occur near the traffic source and is not applicable to inter-datacenter incast.



(a) Senders directly send to the remote receiver.



(b) Proxy relays traffic between senders and remote receiver.

Figure 1: (1a) Status quo: Senders directly send to the remote receiver. Flows are bottlenecked at the receiver down-ToR. (1b) Senders send to a proxy within their datacenter, and the proxy forwards to the remote receiver. Flows are bottlenecked at the proxy down-ToR in the sending datacenter.

3 Insights

To handle the unique challenge of the long feedback loop and mitigate inter-datacenter incast, we advocate for adding a proxy server between the senders and the receiver. Instead of directly sending to the remote receiver (shown in Figure 1a), we propose that each sender sends packets to a proxy, which then forwards them to the receiver (shown in Figure 1b). Surprisingly, the seemingly counterintuitive idea of adding an extra proxy hop reduces inter-datacenter incast latency. We explain the key insights next.

Insight #1: The proxy server shifts the bottleneck closer to the senders. The extra proxy hop lengthens the path each packet takes from sender to receiver, but shortens the feedback loop. Concretely, by placing the proxy server at the same datacenter as the senders, congestion occurs at the proxy down-ToR inq the sending datacenter which is only microseconds away from the senders. As a result, the feedback delay (between the congestion point and senders) is only several microseconds. In comparison, without a proxy, congestion occurs at the receiver down-ToR in the receiving

datacenter and the feedback delays for milliseconds. Figure 1 illustrates the comparison.

Insight #2: Early network feedback shortens the feedback loop, hence enabling faster convergence. The bottleneck shift offers senders an opportunity to quickly converge to suitable sending rates that fully utilize available bandwidth without causing severe congestion. Crucially, a proxy that simply relays packets between senders and the receiver does not accelerate convergence, because it still takes at least as long for the senders to receive network signals. Hence, to leverage this opportunity, the proxy must provide early network feedback, such as loss signals to senders, so that they can adjust their sending rates promptly to network conditions before the remote receiver even notices any issue.

Insight #3: Streamlined proxy design enables early loss detection with minimal processing overhead. A naive way to shorten the feedback loop is to set up two independent connections for each flow: one connection between the sender and the proxy, and one connection between the proxy and the receiver. This way, the sender-proxy connection is contained entirely within the same datacenter, incurring microsecond-level RTTs and allowing fast network feedback (*e.g.,* via small RTOs). However, this naive design requires the proxy to perform the entire sending and receiving logic which incurs unnecessary processing overhead. We observe that full-fledged independent connections are not necessary for shortening the feedback loop. In fact, it suffices if the proxy just keeps track of packet losses and informs the sender about them early. Leveraging switch trimming support and eBPF capabilities, we have implemented a prototype of this lightweight proxy design, which adds only microseconds of processing overhead (details in §5).

4 Early Promise

We show preliminary results based on htsim packet-level simulator [27] and demonstrate that adding a proxy hop, with both naive and streamlined proxy designs (introduced in §3, details in §5), reduces incast latency across various (1) incast degrees and (2) sizes and across different (3) intra- and inter-datacenter latency gaps.

4.1 Methodology

Simulation setup. We simulate two datacenters, each using a leaf-spine topology [8] with 8 spine switches and 8 leaf switches. Each leaf switch is connected to 8 servers. Spine and leaf switches, and leaf switches and servers, are connected with 100Gbps links of 1us propagation delay. The two datacenters are connected via 64 backbone routers. Each spine switch is connected to 8 backbone routers with 100Gbps links of 1ms propagation delay. Senders follow a DCTCP-like congestion control where the sender resets its congestion window upon timeout, decreases the window upon receiving marked ACK packet or NACK packet and increases the window upon receiving unmarked ACK packet. Initial window is set to be 1BDP, following [51]. We use packet spraying. Spine and leaf switches have 17.015MB buffer and the lower and higher marking thresholds are 33.2KB and 136.95KB, respectively, following [9]. Backbone routers have a deeper 49.8MB buffer and the marking thresholds are 9.96MB and 39.84MB. We run each setup 5 times and report the average, minimum and maximum incast completion time.

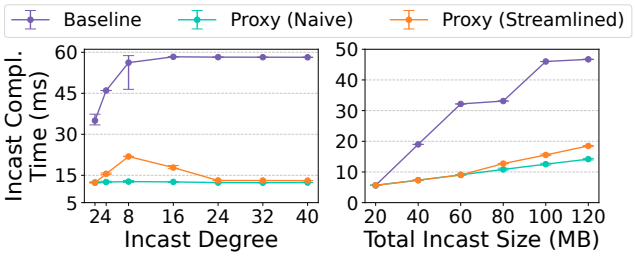


Figure 2: Left: Both proxy schemes significantly reduce the incast completion time across all incast degrees compared to the baseline. The reduction is more pronounced at larger degrees.
Right: Both proxy schemes significantly outperform the baseline for any incast that is large enough to induce packet loss at the first RTT.

Schemes. We compare three schemes below.

- **Baseline**: No proxy is used. Senders directly send to the receiver.
- **Proxy (Naive)**: We designate a proxy server in the sending datacenter. For each flow, we set up two connections: one from sender to proxy (proxy_R), and the other from proxy (proxy_S) to receiver. Proxy_R keeps a queue and enqueues packets when receiving from the sender. Proxy_S sends a packet onto the wire as long as the queue at proxy_R is non-empty and there is bandwidth available.
- **Proxy (Streamlined):** We designate a proxy server in the sending datacenter. Each flow from the sender to the receiver is routed via the proxy. We enable packet trimming support on switches (*e.g.,* used in NDP [27], EQDS [54] and Ultra Ethernet [68, 69]). Upon receiving a packet from the sender, the proxy checks whether it is a header-only packet. If so, it sends a NACK back to the sender; otherwise, it forwards the packet to the receiver. Upon receiving a packet from the receiver, the proxy simply forwards it to the sender.

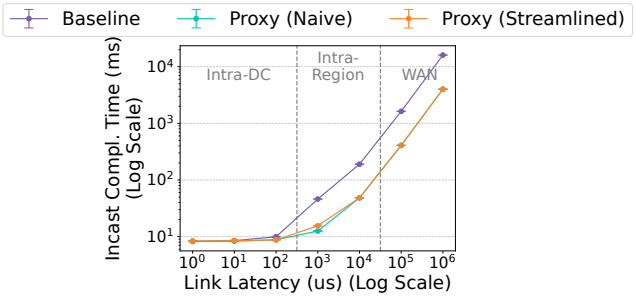


Figure 3: Both proxy schemes significantly reduce the incast completion time at either the region-level or the WAN-level. The reduction becomes more pronounced with higher link latency, since the feedback loop shortens more substantially. Both x- and y-axes are log scale.

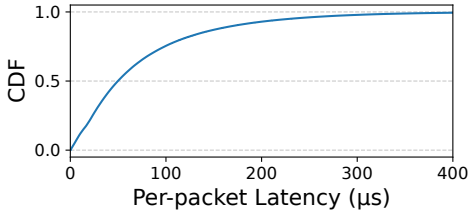


Figure 4: A naive user-space proxy implementation incurs prohibitively expensive kernel overhead and user-kernel context switches.

4.2 Results

Incast degree. In Figure 2 (Left), we fix the total incast size to 100MB and vary the number of incast senders. The total traffic is split equally among all senders. Both Naive and Streamlined speed up incast completion compared to the baseline across all incast degrees by a staggering 40.43ms (75.67%) and 37.63ms (70.60%) on average, respectively. The latency benefit gets more pronounced as the incast degree gets higher. At larger incast degrees, the baseline sees higher aggregate sending rates from more senders initially, and thus takes more time to decrease to match the available bandwidth. Moreover, Figure 2 (Left) validates the potential of Streamlined. At larger incast degrees, the performances of the two proxies are almost equivalent. At smaller incast degrees, Streamlined incurs larger incast latency compared to Naive (but still much smaller compared to baseline), because the fast feedback loop decreases the sending rates too aggressively, leading to link under-utilization.

Incast size. In Figure 2 (Right), we fix the incast degree to 4 and vary the total amount of incast traffic. Both proxy schemes demonstrate significant incast latency reduction compared to the baseline for any incast larger than 20MB, achieving 57.08% and 53.60% reduction on average, respectively. In the case of the 20MB-incast, it starts with a reasonable collective sending rate, sees no packet loss, and thus the feedback delay is not as important; all three schemes are on par and there is no benefit using a proxy.

Latency gap. In Figure 3, we fix the incast degree to 4 and the total incast size to 100MB. The intra-datacenter link latency is 1us. We vary the latency of the long-haul links connecting the two datacenters. Latency under a few hundred microseconds mimics intra-datacenter; tens of milliseconds, intra-region; and over a few hundred milliseconds, WAN. Both proxy schemes outperform the baseline for any link latency larger than or equal to 100us, illustrating the great promise of our approach in alleviating inter-datacenter incast at both the region-level and the WAN-level. The incast latency savings are more pronounced with larger link latencies. Naive reduces incast completion time by 1.16ms (11.70%) with 100us link and by 12.02s (75.00%) with 1ms link. The reductions are 1.22ms (12.29%) and 12.02s (75.00%) for Streamlined. The latency saving increases with larger link latency because the feedback loop shortens more substantially.

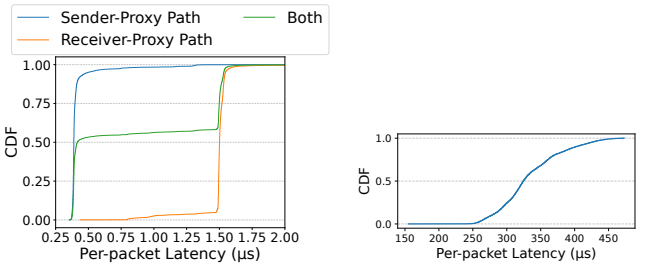
5 Design Considerations

We reduce inter-datacenter incast latency by leveraging a proxy server placed strategically in the sending datacenter. We argue that an effective proxy design must: (1) shorten the feedback loop from the congestion point to the senders with minimal processing overhead; and (2) orchestrate the proxy server selection across multiple incasts. In this section, we share our preliminary investigation towards shortening the feedback loop with minimal overhead and discuss a few unresolved questions that require further investigation.

Testbed setup. Our testbed consists of two x86_64 Ubuntu 24.04 servers running kernel version 6.11.0 with Mellanox ConnectX-5 NICs. The proxy is loaded onto one server, with the sender and receiver programs colocated on the other[1]. Each server utilizes a single NIC link connected to a switch. We analyze two proxy implementations: (1) a user-space program instrumented at the TC layer, where a sender's packet is intercepted by the proxy and forwarded to its socket mirror; and (2) an eBPF program loaded on a TC qdisc. To generate test load between sender and receiver, we use the iperf utility. A single test run consists of a 10Gbps line rate for 30 seconds. We report CDFs of per-packet latency measured using a combination of eBPF instrumentation and tcpdump.

Using independent connections (*i.e.,* Naive). One naive proxy design involves running a sender program and a receiver program at the proxy. For each flow, two independent connections are set up between the sender and the proxy,

[1]By colocating sender and receiver, we avoid having to synchronize distributed traces to capture measurements.



(a) Lower bound **(b) Upper bound**

Figure 5: (a): The lower-bound overhead is small, highlighting the potential of having an eBPF-based proxy on the critical path. (b): In contrast, the upper-bound overhead (including networking stack overhead) is disproportionally large, highlighting the minute impact of the proxy and the importance of hooking lower in the stack.

and the proxy and the receiver, respectively. The feedback loop is fast in this case because the sender-proxy connection is contained entirely in the same datacenter, *e.g.,* supporting only microsecond-level timeout for loss detection in the worst case. Figure 4 shows the per-packet latency of our naive proxy design implemented in user space, which captures the packet transmission time from the TC hook to user space, user-space processing latency, and back. The 99th percentile latency gets as high as 359.17us, demonstrating a high processing overhead that may defeat the purpose of using a proxy for reducing incast latency. The excessive processing overhead can be attributed to two sources. First, the naive design requires the proxy to perform the entire sending and receiving logic which incurs additional processing overhead. Second, the implementation in user space incurs additional overhead *e.g.,* from context switches and interrupts when a packet traverses to and from the user space. While moving the implementation to kernel space could save some overhead, the extra sender and receiver logic still incurs additional processing, and maintaining independent connections relies on socket-level semantics and naturally precludes more efficient implementation lower in the kernel.

Tracking packet loss at the proxy (*i.e.,* Streamlined). We argue that it is not necessary to incur the full sending and receiving logic on the proxy and set up full-fledged connections for shortening the feedback loop. Instead, it is sufficient to just have the proxy keep track of packet loss and signal the sender when loss happens. This way, senders learn about network congestion with only microsecond-level delay and can promptly decrease their sending rates. To validate, we have implemented a proxy prototype using Linux TC, demonstrating that we can avoid unnecessary processing and user-space networking overhead by pushing codes down into kernel via eBPF. In Figure 5, we measure the lower bound (including runtime of eBPF bytecode without kernel overhead from NIC to TC) and upper bound (including proxy processing and forwarding in addition to packet-to-wire, physical transmission, packet reception) of the processing overhead [2]. The median lower-bound overhead of merely 0.42us highlights the potential of having an eBPF-based proxy on critical path. In Figure 5a, distributions of the two paths differ as a result of different per-flow state management. The disproportionally large upper-bound overhead, with a median of 325.92us, highlights the minute impact of the proxy logic itself compared to networking stack overhead and underscores the importance of hooking the proxy lower in the host stack.

Future work #1: Tracking packet loss at the proxy without router support. A generalizable proxy design needs to keep track of packet loss without special router support, *e.g.,* packet trimming. The challenge lies in disambiguating reordered packets from lost packets within eBPF's constrained memory and limited primitives. For instance, given the resource constraint, which packets are more important to keep track of? How much error can the proxy tolerate with its loss detection? Are false positives or false negatives more fatal? The answers to these questions are intertwined with routing (*e.g.,* packet spraying causes more reordered packets), topology (*e.g.,* unstructured topology can cause more reordered packets with varied-length paths) and congestion control (*e.g.,* BBR [15] is more resilient to loss), warranting further investigation.

Future work #2: More efficient proxy implementation. As the proxy processing is on the critical path, the per-packet processing overhead must be kept low for the proxy to run at line rate and also for incast completion time to stay low. While our initial implementation leverages TC's flexibility, there is still room to improve performance. For example, the proxy program has the potential of being offloaded to the NIC directly, and moving to the eXpress Data Path (XDP) hook can further reduce kernel overhead. The implementation decision could also depend on characteristics of the deployment target.

Future work #3: Orchestrating proxy selection across incasts. Effective orchestration among the senders and proxy servers in an incast and across multiple incasts in the same datacenter is critical to the performance in a real-world deployment. But effective orchestration is non-trivial due to several challenges. First, the proxy needs to be selected quickly and avoid contention with other incasts. It can be

[2]Lower bound was taken via eBPF instrumentation, while the upper bound was taken via tcpdump. tcpdump was utilized for its flexibility, but we found measurements to encapsulate additional host latency reflecting prior work [39].

selected either by a global orchestrator, which requires frequent updates on proxy status, or in a decentralized manner with repeated trials by individual incast, which can lead to communication overhead. Second, further research is needed to determine which server can act as a proxy, which server can act as an orchestrator and whether there is any opportunity of leveraging application-layer orchestration. Third, as shown in Figure 2 (Right), not all incasts benefit from using a proxy and future work needs to understand how to identify incasts that should be routed through a proxy.

6 Research agenda

Beyond the proxy mechanics discussed in §5, this paper opens up a rich research agenda related to the interface between application development and resource allocation.

Proxying incast through programming abstraction: While beneficial, integrating a proxy into datacenter operations is not straightforward. It is unrealistic to assume that application developers or cloud customers will proactively choose to use a proxy instead of sending traffic directly across datacenters. Therefore, we need a programming abstraction that allows developers to declare when their application creates incast-like communication across components that could be remote. At deployment time, the cloud provider can use this information to convert an inter-datacenter incast into a proxy-assisted one, without requiring any changes or permission from the application. Doing so is not trivial. A programming abstraction must strike a balance between expressiveness—so it can capture meaningful information about incast behavior—and usability—so that developers are willing and able to adopt it. Worse yet, a poorly designed abstraction may introduce new semantic violations or failure modes, for example, if the specification is ambiguous or misinterpreted at deployment time. This raises a key research question: *How can we design a programming abstraction that enables developers to communicate potential incast patterns clearly and concisely, without introducing ambiguity or imposing excessive overhead.*

Proxying incast through pattern-aware rerouting: An alternative approach to leveraging proxies is to identify incast-inducing jobs based on the traffic patterns they create. This is particularly useful in scenarios where the cloud runs third-party applications, and explicit developer annotations are unavailable. Importantly, some applications exhibit periodic behavior, providing an opportunity to predict when an incast is about to occur. ML training is one such example, where synchronization phases follow regular patterns [44, 48, 53, 60]. In these cases, the cloud operator can proactively detect incast and route traffic through a local proxy, naturally throttling it before it traverses long-haul links. However, this is extremely challenging, as it demands highly accurate, low-latency detection and near-instantaneous intervention. This raises an interesting research question: *How can we detect an incast and route traffic through a proxy fast enough to tame it without disrupting application semantics?*

7 Conclusion

Inter-datacenter incast is an emerging yet challenging problem. This paper posits that adding a proxy hop in the sending datacenter accelerates incast completion by allowing senders to converge faster to an appropriate aggregate sending rate. Motivated by initial results, we present a rich research agenda ranging from designing a low-overhead proxy to rethinking datacenter resource allocations.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. This work was supported by the National Science Foundation (NSF) through Grants CNS-2442625, and CNS-231944.

References

[1] [n. d.]. Distributed Data Center Architecture: Ensuring Scalability. https://www.gable.ai/blog/distributed-data-center-architecture
[2] [n. d.]. Regional, dual-region, and multi-region configurations. https://cloud.google.com/spanner/docs/instance-configurations
[3] [n. d.]. StorageGRID architecture and network topology. https://docs.netapp.com/us-en/storagegrid-116/primer/storagegrid-architecture-and-network-topology.html
[4] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. 2021. Contracting wide-area network topologies to solve flow problems quickly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 175–200.
[5] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. 2022. ABM: Active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 36–52.
[6] Vamsi Addanki, Wei Bai, Stefan Schmid, and Maria Apostolaki. 2024. Reverie: Low pass {Filter-Based} switch buffer sharing for datacenters with {RDMA} and {TCP} traffic. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 651–668.
[7] Adi Gangidi. 2023. Scaling RoCE Networks for AI Training. https://atscaleconference.com/videos/scaling-roce-networks-for-ai-training/.https://atscaleconference.com/videos/scaling-roce-networks-for-ai-training/
[8] Mohammad Alizadeh and Tom Edsall. 2013. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st annual symposium on high-performance interconnects*. IEEE, 71–74.
[9] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.
[10] M Allman, H Balakrishnan, and S Floyd. 2001. RFC3042: Enhancing TCP's Loss Recovery Using Limited Transmit.
[11] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. 2023. Empowering azure storage with

{RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 49–67.

[12] Wei Bai, Kai Chen, Haitao Wu, Wuwei Lan, and Yangming Zhao. 2014. PAC: Taming TCP incast congestion using proactive ACK control. In *2014 IEEE 22nd International Conference on Network Protocols*. IEEE, 385–396.

[13] Bloomberg. [n. d.]. Inside the First Stargate AI Data Center. https://www.bloomberg.com/news/features/2025-05-20/inside-stargate-ai-data-center-from-openai-and-softbank

[14] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. 2019. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication*. 29–43.

[15] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue* 14, 5 (2016), 20–53.

[16] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. 2019. Lancet: Better network resilience by designing for pruned failure sets. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (2019), 1–26.

[17] Wen Chen, Fengyuan Ren, Jing Xie, Chuang Lin, Kevin Yin, and Fred Baker. 2015. Comprehensive understanding of TCP Incast problem. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 1688–1696.

[18] Yanpei Chen, Rean Griffit, David Zats, and Randy H Katz. 2012. Understanding TCP incast and its implications for big data workloads. *University of California at Berkeley, Tech. Rep* (2012).

[19] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. 2009. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*. 73–82.

[20] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. The RACK-TLP Loss Detection Algorithm for TCP. RFC 8985. doi:10.17487/RFC8985

[21] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[22] Marek Denis, Yuanjun Yao, Ashley Hatch, Qin Zhang, Chiun Lin Lim, Shuqiang Zhang, Kyle Sugrue, Henry Kwok, Mikel Jimenez Fernandez, Petr Lapukhov, et al. 2023. Ebb: Reliable and evolvable express backbone network in meta. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 346–359.

[23] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. 2022. Glam: Efficient scaling of language models with mixture-of-experts. In *International conference on machine learning*. PMLR, 5547–5569.

[24] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[25] Sally Floyd, Tom Henderson, and Andrei Gurtov. 2004. Rfc3782: The newreno modification to tcp's fast recovery algorithm.

[26] Yixiao Gao, Yuchen Yang, Tian Chen, Jiaqi Zheng, Bing Mao, and Guihai Chen. 2018. Dcqcn+: Taming large-scale incast congestion in rdma over ethernet networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 110–120.

[27] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 29–42.

[28] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Abdul Kabbani, et al. 2023. Datacenter ethernet and rdma: Issues at hyperscale. *arXiv preprint arXiv:2302.03337* (2023).

[29] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. 15–26.

[30] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. 2018. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 74–87.

[31] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 15–26.

[32] Van Jacobson and Robert Braden. 1988. RFC1072: TCP Extensions for Long-Delay Paths.

[33] Van Jacobson, Robert Braden, and Dave Borman. 1992. RFC1323: TCP extensions for high performance.

[34] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.

[35] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. 2016. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 73–86.

[36] Steven H Low Fernando Paganini Jiantao and Wang Sachin Adlakha John C Doyle. 2002. Dynamics of TCP/RED and a Scalable Control. In *IEEE INFOCOM*. Citeseer, 1.

[37] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. 2016. Optimizing bulk transfers with software-defined optical WAN. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 87–100.

[38] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. 2014. Calendaring for wide area networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 515–526.

[39] Aqsa Kashaf, Aidan Walsh, Maria Apostolaki, Vyas Sekar, and Yuvraj Agarwal. 2024. Network Function Capacity Reconnaissance by Remote Adversaries.

[40] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. 2022. Decentralized cloud wide-area network traffic engineering with {BLASTSHIELD}. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 325–338.

[41] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, et al. 2023. {OneWAN} is better than two: Unifying a split {WAN} architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 515–529.

[42] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. {Semi-oblivious} traffic engineering: The road not taken. In *15th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 18). 157–170.

[43] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[44] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. 2023. Accelerating distributed {MoE} training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 945–959.

[45] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 527–538.

[46] Kexin Liu, Chen Tian, Qingyue Wang, Hao Zheng, Peiwen Yu, Wenhao Sun, Yonghui Xu, Ke Meng, Lei Han, Jie Fu, et al. 2021. Floodgate: Taming incast in datacenter networks. In *Proceedings of the 17th International Conference on emerging Networking Experiments and Technologies*. 30–44.

[47] Ximeng Liu, Shizhen Zhao, Yong Cui, and Xinbing Wang. 2024. FIGRET: Fine-Grained Robustness-Enhanced Traffic Engineering. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 117–135.

[48] Natchanon Luangsomboon, Fahimeh Fazel, Jörg Liebeherr, Ashkan Sobhani, Shichao Guan, and Xingjun Chu. 2023. On the burstiness of distributed machine learning traffic. *arXiv preprint arXiv:2401.00329* (2023).

[49] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. 1996. RFC2018: TCP selective acknowledgement options.

[50] Congcong Miao, Zhizhen Zhong, Yunming Xiao, Feng Yang, Senkuo Zhang, Yinan Jiang, Zizhuo Bai, Chaodong Lu, Jingyi Geng, Zekun He, et al. 2024. MegaTE: Extending WAN Traffic Engineering to Millions of Endpoints in Virtualized Cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 103–116.

[51] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 221–235.

[52] David Nagle, Denis Serenyi, and Abbie Matthews. 2004. The Panasas ActiveScale storage cluster-delivering scalable high bandwidth storage. In *SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. IEEE, 53–53.

[53] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[54] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. 2022. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 761–777.

[55] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. 2011. RFC 6298: Computing TCP's retransmission timer.

[56] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. 2023. {DOTE}: Rethinking (Predictive){WAN} Traffic Engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1557–1581.

[57] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G Andersen, Gregory R Ganger, Garth A Gibson, and Srinivasan Seshan. 2008. Measurement and analysis of TCP throughput collapse in cluster-based storage systems.. In *FAST*, Vol. 8. 1–14.

[58] Jun Rao, Eugene J Shekita, and Sandeep Tata. 2011. Using paxos to build a scalable, consistent, and highly available datastore. *arXiv preprint arXiv:1103.2408* (2011).

[59] Hamed Rezaei and Balajee Vamanan. 2021. Superways: A datacenter topology for incast-heavy workloads. In *Proceedings of the Web Conference 2021*. 317–328.

[60] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. 2022. Accelerating collective communication in data parallel training across deep learning frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1027–1040.

[61] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, et al. 2020. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 735–749.

[62] P Sarolahti, M Kojo, K Yamamoto, and M Hata. 2009. RFC 5682: Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP.

[63] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. 2021. Cost-effective cloud edge traffic engineering with cascara. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 201–216.

[64] Rachee Singh, Nikolaj Bjørner, and Umesh Krishnaswamy. 2022. Traffic engineering: from isp to cloud wide area networks. In *Proceedings of the Symposium on SDN Research*. 50–58.

[65] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. 2021. Cost-effective capacity provisioning in wide area networks with shoofly. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 534–546.

[66] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. 2018. RADWAN: rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 547–560.

[67] Shih-Hao Tseng, Saksham Agarwal, Rachit Agarwal, Hitesh Ballani, and Ao Tang. 2021. {CodedBulk}:{Inter-Datacenter} Bulk Transfers using Network Coding. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 15–28.

[68] Ultra Ethernet Consortium. 2025. UEC Progresses Towards v1.0 Set of Specifications. https://ultraethernet.org/uec-progresses-towards-v1-0-set-of-specifications/. Accessed: 2025-07-10.

[69] Ultra Ethernet Consortium. 2025. Ultra Ethernet Specification Update. https://ultraethernet.org/ultra-ethernet-specification-update/. Accessed: 2025-07-10.

[70] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. 2009. Safe and effective fine-grained TCP retransmissions for datacenter communication. *ACM SIGCOMM computer communication review* 39, 4 (2009), 303–314.

[71] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. 2010. ICTCP: Incast congestion control for TCP in data center networks. In *Proceedings of the 6th International COnference*. 1–12.

[72] Zhiying Xu, Francis Y Yan, Rachee Singh, Justin T Chiu, Alexander M Rush, and Minlan Yu. 2023. Teal: Learning-accelerated optimization of wan traffic engineering. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 378–393.

[73] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. 2022. Congestion control for cross-datacenter networks. *IEEE/ACM Transactions on Networking* 30, 5 (2022), 2074–2089.

[74] Jiao Zhang, Fengyuan Ren, and Chuang Lin. 2011. Modeling and understanding TCP incast in data center networks. In *2011 Proceedings*

IEEE INFOCOM. IEEE, 1377–1385.

[75] Gongming Zhao, Jingzhou Wang, Hongli Xu, Zhuolong Yu, and Chunming Qiao. 2023. COIN: Cost-efficient traffic engineering with various pricing schemes in clouds. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 1–10.