Cutting Tail Latency in Commodity Datacenters with Cloudburst

Gaoxiong Zeng[1], Li Chen[2]*, Bairen Yi[3]*, Kai Chen[1]
[1]*iSING Lab@Hong Kong University of Science and Technology* [2]*Huawei* [3]*ByteDance*

***Abstract*—Long tail latency of short flows (or messages) greatly affects user-facing applications in datacenters. Prior solutions to the problem introduce significant implementation complexities, such as global state monitoring, complex network control, or non-trivial switch modifications. While promising superior performance, they are hard to implement in practice.**

This paper presents Cloudburst, a simple, effective yet readily deployable solution achieving similar or even better results without introducing the above complexities. At its core, Cloudburst explores forward error correction (FEC) over multipath — it proactively spreads FEC-coded packets generated from messages over multipath in parallel, and recovers them with the first few arriving ones. As a result, Cloudburst is able to obliviously exploit underutilized paths, thus achieving low tail latency. We have implemented Cloudburst as a user-space library, and deployed it on a testbed with commodity switches. Our testbed and simulation experiments show the superior performance of Cloudburst. For example, Cloudburst achieves 63.69% and 60.06% reduction in 99th percentile message/flow completion time (FCT) compared to DCTCP and PIAS, respectively.

I. Introduction

Low latency message[1] delivery is critical for many applications in datacenter networks (DCN), such as web search [1], [2], page creation [3], recommendation systems, stream processing, and online advertising [4]. These applications are usually user-facing, and even a very small delay in flow completion time (FCT) can reduce application performance, degrading user experience [1], [2] and causing financial loss [4].

However, the long tail latency problem is particularly pronounced in production DCNs for multiple reasons (§II-A): (1) applications emit synchronized high fan-in bursts (incast [5]); (2) shared-buffer switches are too shallow [6] to absorb bursts; (3) transport protocols use Automatic Repeat Request (ARQ) [7], [8] and retransmission timeouts for packet recovery; (4) coarse-grained load balancing (e.g., ECMP [9]); and (5) hardware malfunctioning (e.g., packet black-hole, silent random packet drops, etc. [10]–[12]) is unpredictable.

In order to tackle such long tail latency problem, prior works (§II-B) adopt a variety of strategies from fine-grained load balancing [3], [13]–[15], rate control [1], [4], [16]–[20], prioritization [2], [3], [21]–[23], to fast loss recovery [11], [24]. However, most of these proposals introduce non-trivial implementation difficulties, such as global state monitoring [13], [14], complex network control [14], [22], and switch modifications [13], [16], [21], [24]. While achieving superior performance, these solutions are hard to deploy in existing commodity datacenters.

*Work done while Li Chen and Bairen Yi were both at HKUST.

[1]Message and short flow are used interchangeably in this paper.

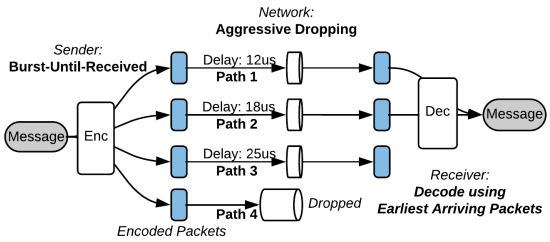

Fig. 1. Cloudburst Concept

Therefore, we ask a pragmatic question: *is there a simple scheme for commodity datacenters to cut tail latency without the above complexities, while still delivering similar or better performance?* In this paper, we answer the question affirmatively by presenting Cloudburst, a simple and deployable solution to cut tail latency of short messages in datacenters.

Cloudburst works as follows (Figure 1 & §III):

- *Encoding messages with FEC:* Forward error correction (FEC) has been deployed in many applications [25], [26], it uses proactive and redundant approach to tolerate errors. In DCN, the error tolerant feature can be adopted to handle packet losses. That is, we do not need to retransmit the lost packets while still guaranteeing the reliable transmission. Cloudburst explores the coding dimension of the transport-layer design. It employs FEC with proactive and oblivious redundant transmission. Each encoded packet contains information of multiple original packets. In this way, even if some of the encoded packets are lost, the original packets can still be reconstructed at the receiver.
- *Bursting over multiple paths:* Cloudburst spreads encoded packets over multiple paths, which obliviously exploits the rich path diversity in modern DCNs [27]–[29], as well as the temporary network under-utilization [30]. If any congestion-free path exists, Cloudburst will take advantage of them without extra signalling overhead.
- *Separated and size-limited switch queue:* We limit the buffer usage of the Cloudburst short flows to minimum, so that the per-hop queueing latency of a Cloudburst packet is deterministically low (less than a few μs). This is achieved by separating Cloudburst short flows and other traffic in different queues, and limiting the maximum depth of the Cloudburst queue to a tiny value (a few packets). Such a limitation on buffer usage of Cloudburst traffic also makes it friendly to other traffic.

Our design choices result in a very simple transport that works drastically different from TCP. Cloudburst performs neither congestion control[2] nor reactive retransmission (i.e., no per-packet ACK, no congestion feedback, no loss detection, and no retransmission timeout (RTO)). Instead, it simply transmits at line-rate and unilaterally keeps generating and sending the encoded packets until the receiver acknowledges the reception of the message. Meanwhile, the in-network requirement of Cloudburst is also simple: limiting queue size is readily configurable in commodity switches [33].

To be deployable with existing commodity datacenters, Cloudburst is now implemented as a user-space library (§IV), so that applications can voluntarily use it for latency-sensitive short flows. We built a small 2-tier leaf-spine testbed and deployed Cloudburst on it. The testbed consists of 6 Pronto-3295 (Broadcom) Ethernet switches and 8 Dell R320 servers with quad core Xeons E5-1410 CPU and 1GbE NIC. Our implementation experience shows that Cloudburst is readily-deployable with existing commodity switches.

Our testbed experiments (§V-A–§V-C) show that Cloudburst outperforms prior practical schemes. For example, it achieves more than 60.06% and 63.69% reduction in 99th percentile (p99) flow completion time compared to PIAS [2] and DCTCP [1]. Furthermore, our results also verify that it is resilient to incast, and handles failures gracefully. We complement testbed experiments with large-scale simulations (§V-D) in 10/40G environments. We find that Cloudburst achieves comparable or better performance than prior solutions. For example, it achieves 24% tail latency reduction compared to a clean-slate design, pFabric [21].

II. Achieving Low Latency

We first look at the causes for long tail latency in DCN (§II-A). Then, we review the prior solutions to cut tail latency (§II-B). Finally, we overview Cloudburst (§II-C).

A. Causes of long tail latency

The traffic characteristics of DC applications, current DCN configurations, and the usage of TCP in DCNs jointly contribute to the long tail latency for short flows.

High fan-in bursts (incast [5]): Tree-based applications emit synchronized high fan-in bursts. They are constructed with multiple layers, and each parent aggregates results from its children. The interactivity constraint (e.g. 300ms [4]) is distributed to each layer (e.g. 40ms). Hence children of a same parent respond at almost the same time, causing fan-in burst at the parent. These synchronized fan-in bursts exceed the egress port capacity of switches, causing queue to build up and may lead to congestive packet drops.

Shallow shared-buffer switch: Shared buffer is usually configured to absorb bursts, i.e. a portion of buffer is shared among multiple ports, so if one port is experiencing bursts, it can take all available shared buffer. However, shallow buffer switches cannot absorb fan-in bursts from applications at high bandwidth. The expansion of buffer (from 4MB to 12MB [6], [34]) in commodity switches is not compatible with the 10X+ growth of per port bandwidth (from 1Gbps to 10/40Gbps [29]). Additionally, throughput-intensive flows requires a certain amount of buffering ($\alpha\times$Bandwidth Delay Product, α depends on the transport protocol [1]) in the switch to achieve high throughput, requiring guaranteed buffer in each port and further reducing the shared buffer for burst tolerance. When bursts occurs, packets from latency-sensitive flows may be dropped due to lack of buffer [5].

Error handling and retransmission timeout: Applications deliver data reliably using TCP with Automatic Retransmission Request (ARQ) error handling (e.g., Go-back-N [7], Selective Repeat [8]), and timeout-based error discovery). For each error, ARQ needs at least one RTT to recover, thus it works well for long flows, because acknowledgement of received or missed packets are batched for large congestion windows. However, ARQ does not help for short flows, which may finish within the slow start phase. If a flow's initial window packets coincide with congestion and get dropped, it can only discover the loss by RTO. Therefore, setting a proper RTO_{min} [3], [5] is key to fast recovery of short flows. RTO_{min} in DCN is usually set to 5ms [5], [11], which is almost two orders of magnitude larger than the base RTT ($\sim$ 100us). A single timeout can easily lead to long latency tail. Reducing the timeout can surely benefit packet recovery, but may also increase the load on network and servers due to frequent retransmissions and require high-precision timers.

Malfunctioning hardware: Packet loss may also occur due to hardware failures, which happens even in well-engineered modern DCNs with lossless fabric [10]–[12]. Such failures may come from TCAM deficits, aging transceivers, etc. Particularly, silent packet drops or blackholes are discovered with network-wide diagnostic tools [10]. Thus, these failures are difficult for single-path transport to recover, and can take multiple RTTs for multipath transport to recover.

Imperfect flow load balancing (LB): Current load balancing in commodity DCNs usually depends on flow hashing, i.e., per-flow ECMP [9], to keep utilization of parallel paths even. However, hash collision may occur, which can temporarily overload some links and cause queue length to grow, prolonging per-packet latency and inducing packets drops. Randomized LB also cannot help with application bursts when all flows have the same destination port.

B. Prior Solutions

We overview the prior solutions to the tail latency problem:

Reducing queueing latency: Generally, solutions in this category follow a few strategies such as fine-grained load balancing [3], [13]–[15], rate control [1], [4], [16]–[20], and traffic prioritization [2], [3], [21]–[23].

- By load balancing traffic across multiple paths evenly, we can avoid excessive queue build-up that may result in queueing latency or loss. For better performance, congestion awareness is often required [3], [13]–[15]. For example, CONGA [13] measures link utilization and

[2]In fact, for erasure coded flows, dropping can be considered as a form of congestion control [31], or "decongestion control" [32].

directs flows to less congested paths, while DeTail [3] detects and avoids congestion by monitoring queue lengths. In the extreme, Fastpass [14] centrally schedules and routes every packet with complete knowledge of per-path load conditions. These solutions, however, require very complex network control or switch modifications.

- By rate-throttling flows (especially the larger ones) based on ECN [35], delay [36], or in-band network telemetry (INT) [16] signal, we can control the queue build-up so that latency-sensitive short flows see small queues at the switch. While helpful, these solutions [1], [4], [16]–[19] still rely on load-balancing, and short flows may suffer long latency when traffic is unevenly distributed. Besides, the advanced INT signal may not even be available on switches and thus require customized hardware.
- By giving latency-sensitive flows high priority, the switch dequeues them first, without regards to lower priority ones before them, thus achieving low latency [2], [3], [21]–[23]. For example, pFabric [21] assumes priority dequeueing (and dropping) to minimize flow completion time of short flows, and QJump [23] leverages prioritization to cut tail latency. However, they either assume infinite switch queues or rely on accurate configurations.

Recovering from packet losses: Fast in-network feedback for packet drops [24], [41] can accelerate the transition of TCP state machine, thus triggering retransmission earlier. As above, these solutions also require non-trivial hardware modifications. Retransmission can also be proactive: FUSO [11] augments MPTCP [37], [38] by eagerly retransmitting on less congested paths. However, it needs to keep complex states of subflows.

Another line of work [17], [18], [39] seeks help from lossless fabric [40]. However, at large scale, packet loss may still happen even on lossless fabric, due to mis-configuration or hardware failures [12]. In case of losses, they often rely on the NIC hardware for efficient recovery. This line of work is still under exploration and is beyond the scope of this paper.

Proactive transport solutions: Some recent works [41], [42], [44] use pre-allocated rate to avoid excessive packet delay and inaccurate self-adapted rate control. In these solutions, link capacities are *proactively* allocated by the receivers as "credits" to each active sender who then send "scheduled packets" at an optimal rate to ensure low queueing delay and near-zero packet loss. However, this approach requires at least one RTT to allocate credits to a new flow, which is unacceptable for short flows. While recent schemes [45], [46] enable line rate start, issues still exist for short flows. For example, it is hard to assign a right amount of credits to senders before termination in the last RTT [42], [43]—too large for short flows will lead to link under-utilization due to credit wastage, while too small will introduce large delay.

C. Cutting tail latency via FEC over multipath (Overview)

Cloudburst aims to cut the long tail latency with a simple and readily deployable protocol. It applies FEC to multiple paths (§III-A) in commodity datacenters. By proactively spreading encoded packets over multiple paths in parallel, and decoding the first few arriving ones to recover the original message, Cloudburst obliviously exploits the uncongested paths to achieve persistent low latency, while not maintaining network state or performing complex control.

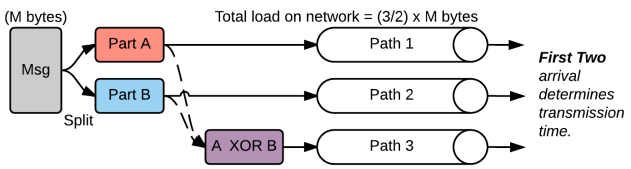

Fig. 2. Example of a 3-path FEC

At the end-host, Cloudburst performs "burst-until-received" (§III-B). The sender encodes short messages with FEC and proactively sends the encoded packets over multipaths until the messages are decoded at the receiver with the first few arriving packets. In this way, Cloudburst always exploits the best paths for low latency. If any uncongested path exists, Cloudburst uses it without signaling; if a path is experiencing congestion or failure, Cloudburst avoids it automatically by using encoded packets from other under-utilized paths.

At the switch, Cloudburst performs aggressive dropping (§III-C) for Cloudburst flows, We limit the buffer usage of the Cloudburst flows to minimum, i.e., separating Cloudburst traffic and other traffic in different queues, and limiting the maximum depth of the Cloudburst queue to a small value. This ensures that the per-hop queueing latency is deterministically low. Besides, it protects other flows from being affected by the burst-until-received behavior of Cloudburst flows, making it friendly to co-existing traffic.

III. Design

We proceed to describe the details of Cloudburst design.

A. Choosing FEC for encoding

Figure 2 is a simplified illustration of how FEC works. There are 3 paths from source to destination, and past experience indicates only one of them has congestion at any given time, but the sender does not know which until it sends. Suppose the sender has 2 packets to send, say A and B. A suitable FEC scheme is to create another packet $C = A \otimes B$ (bitwise exclusive-OR). After encoding, the sender sends A, B, C on three paths in parallel. The receiver can recover the message as soon as it collects any 2 out of 3 packets, thus avoiding congestion at the cost of sending 1.5x more traffic.

Fixed-rate block codes, as shown in the example in Figure 2, are simple to understand and implement, but have the fundamental difficulty of choosing a right coding rate. In Figure 2, the suitable code rate is $\frac{3}{2}$. Fixed rate codes like this may suffer if the path condition deteriorates: if two of the paths suddenly become congested instead of one in Figure 2, then the message will take much longer time to recover, as the first two arrivals determine the transmission time.

Rateless erasure codes, or fountain codes [47], are better suited for dynamic traffic characteristics in DCNs, as its code rate is adaptable to dynamic channel conditions. The key property of fountain codes is that they can generate a potentially

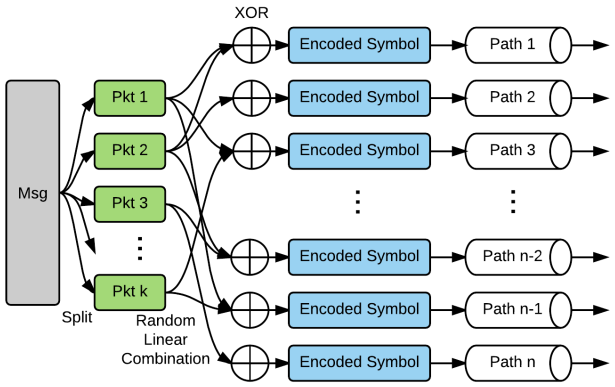

Fig. 3. Random linear encoding in one round of transmission

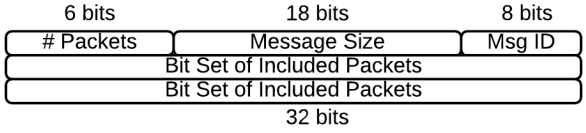

Fig. 4. Cloudburst header format

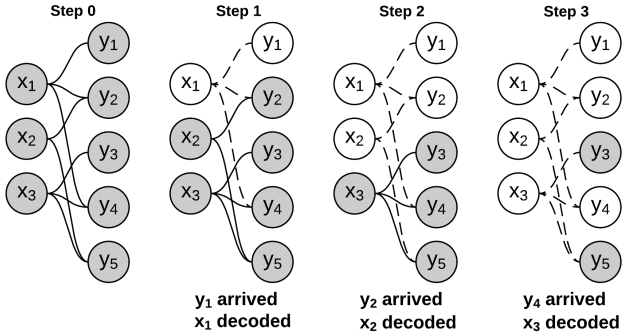

Fig. 5. Decoding example

limitless sequence of encoded symbols from a given set of source symbols (the source symbols can be decoded with a large enough subset of encoded symbols). Specifically, we adopt LT Code (LTC) [48] in our prototype due to its low complexity, where an encoded packet is generated by random linear combination of original packets.

Encoding: We assume the message is given before the transmission. As shown in Figure 3, we first packetize the original message into equal-length parts. If necessary, we pad zero bits to packets so that every part from the original message has the same size: $MTU - H_{IP} - H_{UDP} - H_{cbrst}$, where MTU is Maximum Transmission Unit (1.5KB), H_{IP} is IP header size (20B), H_{UDP} is UDP header size (8B), and H_{cbrst} is Cloudburst header size. For each encoded packet, its degree (d) is defined as the number of un-decoded original packets encoded in it. We first chose the value of d from a certain probability distribution. Then, we randomly choose d packets from the original message $\{x^{(1)}, \ldots, x^{(d)}\}$. The encoded packet in the ith iteration is then: $y_i = x_i^{(1)} \oplus x_i^{(2)} \oplus \ldots \oplus x_i^{(d)}$. The header of encoded packet (Figure 4) includes the number of packets (n) in the message, the message size in unit of bytes, an 8-bit message ID, and the list of indices $(1), \ldots, (d)$.

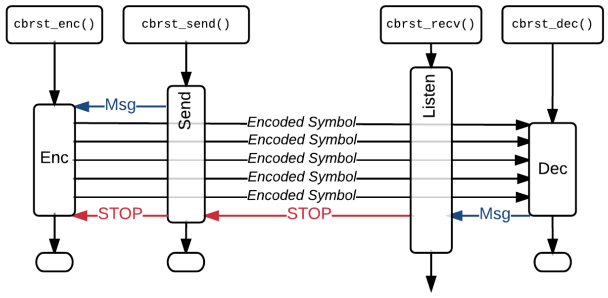

Fig. 6. Sequence Diagram

Decoding: The decoding algorithm uses standard Gaussian elimination method [49]. We illustrate the decoding with an example in Figure 5. In step 1, y_1 is received, and its degree is 1 (Degree of a symbol is defined as the number of un-decoded original packets encoded in it), so that the corresponding x_1 is decoded. In step 2, y_2 is received, and its degree is 2. But since x_1 is already decoded, y_2's degree is reduced by 1, so x_2 is decoded. By iteratively finding symbols with degree 1, the original message is recovered.

B. Burst-until-Received at End-host

The sequence diagram of the end-host operations of Cloudburst is shown in Figure 6. We elaborate on the sender and receiver operations below.

Sender operations: As a variant of fountain code, LT coding can generate endless sequence of encoded packets for a given message. LT coding enables the Cloudburst sender to continuously generate encoded packets and sending them on multiple paths, as described in Algorithm 1. It stops only after it receives a "STOP" signal from the receiver.

This burst-until-received behavior may consume all bandwidth of the network. So we add a rate control mechanism to allow users to adjust the sending rate. The user/application specifies rate r, which is the share for Cloudburst traffic. We define a round of transmission as the period of time the sender puts an encoded packet on each of the paths (Line#6-9 in Algorithm 1), and the rate is adjusted by adding delay between rounds of transmission. For example, if NIC capacity is 10Gbps, $r = 0.5$, and 5 paths between source and destination. Then the sending rate of the message is 5Gbps (1Gbps per path). Pumping 5 1.5KB encoded packets on 5 paths takes 12μs, thus, a 12μs inter-round delay is added.

Receiver operations: All encoded packets are received by the listening thread, and forwarded to their corresponding decoder (each decoder is instantiated for a message). Decoder buffers all the encoded packets that have unrecovered packets for decoding. It returns the message to the receiver listening thread after full recovery, or expires if there is no more incoming encoded packets after a pre-defined timeout. Then the receiver signals a STOP to the sender. In case that the

Algorithm 1: `cbrst_send(·)`

Input: Receiver IP & Port, Message m, Transmission timeout t, Ratio r
Output: Success/Failure

```
if sizeof(m) > MAX_MSG_SIZE then
    return Failure
Set up count down timer of t
while STOP not received do
    Wait for (1/r * NUM_PATHS * PKT_SIZE / LINK_CAP)
    foreach path p to Receiver do
        Get symbol y from encoding thread and send it on p
        if timer expires then
            return Failure
return Success
```

STOP signal is lost, the receiver will send a new STOP for each received encoded packet after the first STOP is sent.

C. Aggressive Dropping in Network

A major concern is that the open-loop rate control of Cloudburst is dangerous to other flows and may overflow switch buffers in the network. Thus, it seems that a congestion control mechanism for Cloudburst is needed. However, dropping is also a form of congestion control [32] for high-bandwidth erasure coded flows. When packets are erasure encoded, dropping some is not an issue as the message can be recovered with a subset of the packets sent from the source. Thus, for a network with purely erasure coded flows, the switches do not need deep buffers to keep network stable [50], and the sources can burst as fast as possible [32].

This is also true for TCP-like transport. For example, pFabric [21] features minimum TCP, a line rate transport, with shallow buffer and priority dropping at the switch. The flows with same priority achieves the max-min fairness despite dropping due to shallow buffer. In fact, replacing congestion control with erasure coding has been discussed for future Internet [31], but there are fundamental difficulties. First, in public Internet, many transport protocols coexist, and bursts of a few erasure coded flows may take all the buffer and bandwidth, hurting others. Second, on the path of an erasure coded flow, if a packet gets dropped after it passes a few switches, the bandwidth it consumed is wasted. These "dead packets" [32] may induce congestion collapse: packet loss requires additional packets (more redundancy) to recover, which incurs more load in the network and more packet drops, forming a positive feedback loop.

By leveraging the properties of DCNs, targeting only the short flows, and limiting buffer usage, we avoid the drawbacks of erasure coding transport.

- DCNs often have abundant multipath [27]–[29]. With encoded symbols on all paths, the network core with Cloudburst is load balanced for short flows, and the only bottleneck is the egress switch. With just one bottleneck, "dead packets" are not likely to occur.
- For long flows, using erasure coding is dangerous due to the positive feedback loop described above. However, Cloudburst serves only short flows, which do not persist: because of the small sizes and the high bandwidth network, short flows dissipates quickly.
- We enforce aggressive dropping by separating the buffer for Cloudburst and other flows, and limiting the buffer usage of the Cloudburst flows to minimum. This protects other flows from Cloudburst traffic. Thus, fan-in bursts of Cloudburst flows no longer affect others on a shared shallow-buffer switch; meanwhile, TCP flows can also safely use remaining buffer for high throughput.

D. Discussion: two potential issues

Infiniteness of flow sending: The termination condition of our algorithm is that the receiver can decode all of the original packets. However, since we do not use the ACK mechanism, the senders do not know which packets have already arrived, they just randomly chose packets, which may cause the failure of flow completion. In fact, the mathematical property of LTC [48] guarantees the flow completion. k original packets can be recovered by $k + \mathcal{O}(\sqrt{k}\ln^2 k/\delta)$ encoding packets with probability of $1-\delta$. That means, we can bound the total number of encoding packets with a certain probability. Meanwhile, we can also set a packet sending upper bound to avoid the tail latency caused by unlimited sending. When a certain number of encoding packets are sent, the sender will request the receiver for the information of received packets.

Aggravate last hop congestion: In fact, our algorithm focuses on the tail latency caused by the different congestion condition of sub-paths. In DCN, the last hop may be the bottleneck and sending FEC-coded packets only aggravates the congestion. This problem is still caused by the agnostic of packets' receiving in senders. When the last hop congestion (e.g., incast) happens, even the slow flow may not be finished in one RTT, it gives the senders the opportunity and time to respond to the packet receiving information. More specifically, when a flow does not finish within a certain time, the sender will ask the receiver for the arrival information. Even though our algorithm is not designed for solving incast problem, it eliminates the retransmission cost, thus the experiment result shows that Cloudburst still performs well in incast scenario.

IV. Implementation

In this section, we describe implementation and parameter settings of Cloudburst.

A. Enabling Cloudburst at End-host

For the end-host, we build a prototype Cloudburst with Rust 1.6 [51] as a user-space library. The implementation is multi-threaded: sending (receiving) and encoding (decoding) are handled by different threads at the sender (receiver). We discuss the settings of coding-related parameters as follows.

Choosing message size n & degree d: A larger header size supports larger message size at the cost of more header overhead and possibly higher load on the network (with the same code rate, a larger message generates more encoded symbols). We consider header overhead (percentage of packet used for header), computation overhead (number of decoding operations), and coding overhead (number of encoded symbols to reconstruct the message) when choosing n & d.

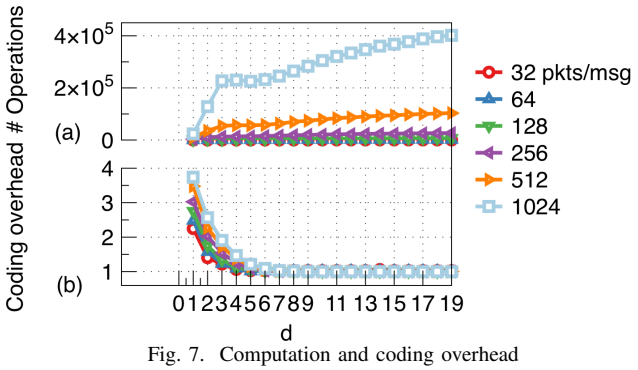

Fig. 7. Computation and coding overhead

Header overhead is as follows: x is the length of the field representing the maximum number of packets in a message ($n = 2^x$ packets). The message size is $log_2(MTU \times 2^x) = x + log_2(MTU)$ (bytes). The bit-set representing the packets included in the symbol is 2^x (bits). Thus, assuming 1.5KB MTU, the header overhead for $x = 9$ is less than 7%, which suggests that, choosing $x < 10$ is efficient.

Computation overhead for different encoding degree d and message size n is plotted in Figure 7(a). For different (d, n) pairs, we perform the en/decoding process 100 times, and calculate the average number of XOR operations necessary to reconstruct the message. We see that the number of operations increases with n. For each n, the number of operations increases with d, but slows down for $d \geq 5$.

Coding overhead is measured by the ratio between the number of packets needed to recovery the message and the number of original packets in the message. We collect this ratio by running the same experiment as above, and plot the results in Figure 7(b). The coding overhead drops for all sizes with respect to d, but beyond $d \geq 5$, increasing d does not lead to lower coding overhead.

With the above results, we recommend to set $d = 5, n = 64 = 2^6$ for the current prototype. It thus supports message size of at most 93.44KB (i.e., $2^6 \times 1.46KB$). We note that typical latency-sensitive applications in DCNs exhibit multi-tier partition/aggregation patterns, and the queries are often less than 10KB in size [1], [4], which suggests the current header design should work well in practice.

Choosing r: r is the expected throughput of Cloudburst flows in the DCN. For oversubscribed DCN, we can adjust the parameter r in Algorithm 1 to the oversubscription ratio. For network with full bisection bandwidth, $r = 1$. In the experiments and simulations, we choose r to be exactly the oversubscription ratio of the network, so that Cloudburst flows will not overload the network.

B. Enabling Cloudburst in Network

Configuring switch buffer: To enforce aggressive dropping, we set the packets of Cloudburst flows to the same DSCP value, so that they are carried in the same switch queue, separated from other non-Cloudburst traffic. We then set the dedicated buffer for the Cloudburst queue to a small value.

On our testbed, we set it to be 1% of total buffer size, and we disable the shared buffer of this queue to avoid affecting other flows. This is done by setting "`buffer queue-limit`" in our Pronto-3295 switches [52]. Other switches also support such configurations. For example, in Cisco switches, we can set the depth of a traffic class queue [53].

Multipath routing A key issue for Cloudburst is how to spread packets on different paths. The implementation is dependent on how multipath is supported in the network.

Sub-optimal multipath: The network may use multipath implicitly, so that load balancing over multipath is transparent to the applications. ECMP [9] is an good example: for each flow, an ECMP-enabled switch picks an outgoing port at random based on the hash of the flow's source and destination IPs and ports. To use Cloudburst on ECMP, the sender and receiver need to maintain a pool of ports, and each encoded symbol will be given a header with a random combination of sender ports and receiver ports, which implicitly asks ECMP to hash packets on different paths. While this may not fully utilize all paths, we show that Cloudburst still works well in §V-C.

Explicit multipath routing: Multiprotocol label switching [54] (MPLS) can provide explicit routing for each packet, but this requires support from the network fabric. Also, setting up multiple path labels for each short message requires signaling the switches on multipath, which is undesirable for low latency delivery. To attain the same efficiency as implicit multipath with ECMP, we turn to a DCN routing scheme—XPath [55], which enables explicit path-based routing in DCNs. It compresses and pre-installs end-to-end paths into forwarding tables of commodity switches, and packets are routed based on the path ID in their headers. With XPath's explicit path control, Cloudburst adds path IDs to the headers of the encoded symbols, which will place them on different paths.

V. Evaluation

In this section, we evaluate Cloudburst with testbed experiments (§V-A–§V-C), complemented by large-scale simulations (§V-D). We summarize the results below:

- §V-A: we inspect Cloudburst's design choices and quantify their benefits. With all choices combined, Cloudburst reduces the tail latency by 75.32% compared to DCTCP.
- §V-B: we compare Cloudburst with the prior practical schemes, and find that it achieves more than 60.06% reduction in p99 flow completion time compared to DCTCP+ECMP [1], [9] or PIAS [2].
- §V-C: we dive into Cloudburst and find that it is resilient to many critical cases including incast and failures.
- §V-D: we use large-scale simulations to show that Cloudburst achieves 24% tail latency reduction compared to a near-optimal clean-slate design, pFabric [21].

Traffic patterns: Following related works [3], we emulate traffic of latency-sensitive applications: data retrieval (request/response) and page generation. Each response message is triggered by a 1.5K-byte (MTU) request from the receiver to the senders. The senders reply with a message with variable size uniformly chosen from $\{5, 10, 20, 50, 93\}$KB. The inter-arrival time of initiating requests follows exponential distribution with mean $4, 5, 10$ms (A Poisson random process

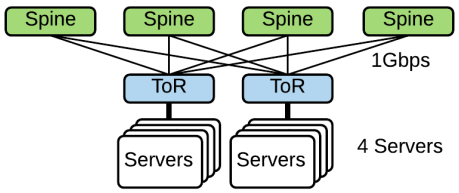

Fig. 8. Testbed setup

with arrival rate $250, 200, 100$ requests per server per second). Each server randomly picks another server to send request.

Testbed: We built a spine-leaf (8 servers, 2 leaf or Top-of-rack & 4 spine switches) testbed to create 4 paths between any pair of servers from 2 racks (Figure 8). Each path corresponds to a spine switch. We use Pronto-3295 switches and Dell PowerEdge R320 servers, each with a quad core Xeons E5-1410 CPU and 1GbE NIC, and with Debian 6.0 (kernel 2.6.32-5) installed. XPath [55] is enabled by default.

We generate background flows to create network congestion. For the 4 servers on one rack, each randomly chooses another server in the other rack, and sends a flow of 10MB using DCTCP on a random path. When a flow finishes, the server will start another one. In this way, each path has the same probability for different degrees of congestion ($1/256, 3/64, 27/128, 27/64, 81/256$ chance to have $0, 1, 2, 3, 4$ flows, respectively). Unless specified otherwise, background flows are in a separate queue (totally 2 queues are used), and switches use WRR for these 2 queues.

A. Inspecting Design Choices

Cloudburst incorporates three design choices: 1) Cloudburst uses LTC to encode the message, and runs the "burst-until-received" protocol; 2) Cloudburst spreads the encoded packets on multiple paths, obliviously taking advantage of uncongested paths; 3) the switches perform aggressive dropping with tiny queues for Cloudburst flows. We now study the impact of each of these decisions progressively.

We run the all-to-all pattern with varying request rates (each for 10 minutes), and plot the p99 completion times for $\{5, 20, 93\}$KB flows in Figure 9,10&11, respectively. We take DCTCP as a reference (parameter setting follows §V-B). We compare the following schemes:

- **A**: FEC (Design Choice 1). We encode the message into encoded packets, and send the packets at line rate using UDP on a randomly chosen single path.
- **B**: **A** + Multipath (Design Choice 1&2). Senders spread the encoded packets of each flow on multiple paths.
- **C**: **A** + Aggressive Dropping (Design Choice 1&3). Senders send encoded packets on a single path, and switches aggressively drop packets by limiting buffering.
- **D**: Cloudburst (Design Choice 1,2,&3). Cloudburst combines all three design choices.

Impact of FEC: Cloudburst uses FEC to perform loss recovery. Consider a simple mathematical model: message size is M packets, and link capacity is C packet/s. Assume the packet drop probability is p_d on a path. If a packet

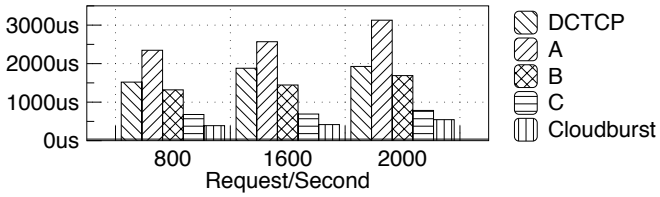

Fig. 9. p99 Completion Time (5KB)

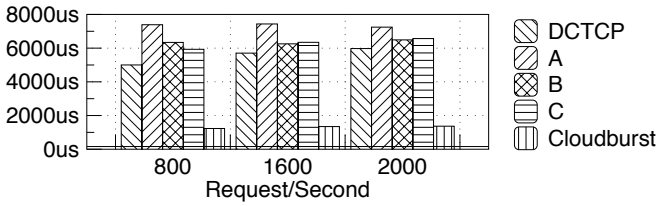

Fig. 10. p99 Completion Time (20KB)

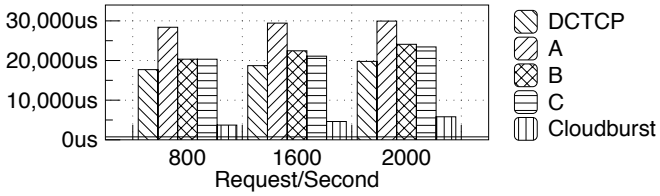

Fig. 11. p99 Completion Time (93KB)

is lost, a sender transmitting at link capacity without coding takes $\frac{M}{C}$s to retransmit it, and the expected latency is $E_D = (1-p_d)\frac{M}{C}\sum_{i=0}^{\infty}(i+1)p_d^i = \frac{M}{(1-p_d)C}$. Thus, with a larger p_d (aggressive dropping), D takes more time to recover a lost packet. Encoding essentially sends multiple packets' information at the same time. For degree d, d original packets are XOR'd for each encoded packet. In this way, a packet do not have to wait for $\frac{M}{C}$ to be retransmitted. For each encoded packet, the sender randomly chooses d packets, thus it takes $\frac{M}{dC}$ to deliver all the packet's information. Therefore, the time to receive information of all packets becomes $\frac{M}{d(1-p_d)C}$, d times smaller than E_D. However, FEC alone cannot work. Scheme **A** keeps generating encoded packets, and sends them on a single path. We see that, the tail latency is 51.12% longer compared to DCTCP across all message sizes at 2000 Request per second (rps). This is caused by the queueing created by the aggressive sending of encoded packets. In contrast, DCTCP controls the queueing by enforcing a small queue at the switch, which allows the end-hosts to react to congestion quickly.

Impact of Multipath: Queueing is created when background flows choose the same path due to imperfect load balancing. Good load balancing is difficult to achieve for TCP flows, because in-order packet delivery is expected. However, the same is not true for erasure-coded flows, as there is no order between encoded packets and the original message is reconstructed as a whole. This allows for easy implementation of load balancing: the sender can simply spray encoded packets on multiple paths. Scheme **B** does exactly this. We observe that, with evenly load balanced FEC flows, the tail latency is improved by 44.52% for 5KB messages across all loads. This improvement is smaller with increasing message size (23.88% for 93KB message), because larger messages

take longer to decode (Figure 7). Compared to DCTCP, we see the same trend: **B** outperforms DCTCP for short messages (tail latency is 16.24% shorter for 5KB) and sightly worse for long messages (tail latency is 5.89% longer for 93KB). This is because Scheme **B** spreads traffic to all available paths evenly, while DCTCP is vulnerable to imperfect load balancing.

Impact of Aggressive Dropping: To counter **A**'s queueing, we can also limit the queue depth. Scheme **C** uses FEC and aggressive dropping on a single path. In Figure 10&11, **C** is similar to DCTCP for 20KB and 93KB messages. For 5KB short messages, **C**'s tail latency is 51.42% shorter than that of DCTCP, because DCTCP relies on timeouts to discover packet loss for short messages. In contrast, **C** proactively retransmits encoded packets despite dropping.

Summary: With all three design points, we have Cloudburst. With erasure coding (Design Choice 1), each encoded packet can help recover any of the d original packets. With multipath forwarding (Design Choice 2), packets are spread on all paths evenly, thus exploiting uncongested paths obliviously. Finally, aggressive dropping (Design Choice 3) ensures that packets arriving at the receiver experience deterministically low queueing delay, because packets that encounter any queue build-up are dropped. Compared to DCTCP, Cloudburst reduces the tail latency by 75.32% (averaged over all sizes).

B. Comparing with prior schemes

We proceed to compare Cloudburst with existing schemes that are implementable in commodity DCNs:

- **DCTCP [1] + ECMP:** RTO_{min} is set to 10ms. ECN marking threshold is set to 65 packets. All flows share the same switch queue.
- **PIAS [2]:** We implemented PIAS with 2-level feedback queue and set the first threshold to be 95KB, so that all short flows or messages have highest priority.
- **Replicated DCTCP:** Transmitting flows with same content using DCTCP on 2-4 paths [56] (paths are randomly chosen). All flows share the same switch queue.
- **MPTCP [37], [38]:** Using tc in Linux, MPTCP flows are tagged with DSCP value for the short flow queue.

We run the all-to-all request/response traffic pattern, and collect message completion time (MCT) for each scheme.

Average Latency: In Figure 12, despite the en/decoding overheads, Cloudburst performs similarly to PIAS for different request arrival rates in terms of average MCT. MPTCP shows the worst performance, because if any congested path exists, MPTCP is bound to experience congestion, prolonging MCT. Among the DCTCP-based schemes, for low request rate (800 r/s), DCTCP with the most duplicated flows (4) achieves the best performance, as it transmits on all paths, thus can always avoid congestion. However, as the request rate increases, the performance of Replicated DCTCP starts to degrade, because the replication essentially multiplies the load, leading to congestion and packet drops.

Tail Latency: In Figure 13, for p99 MCT, Cloudburst outperforms all the other schemes. The p99 MCT captures the tail latency events (e.g. long queueing, packet loss). At 2000 r/s, its tail latency is over 60.06% (63.69%) less than that of PIAS (DCTCP+ECMP). This is because Cloudburst packets are encoded with redundancy, thus flows need not wait for retransmission timeout, unlike TCP variants.

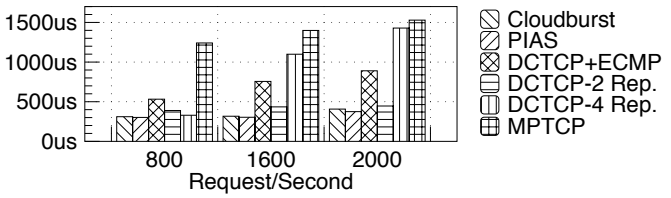

Fig. 12. Average Message Completion Time

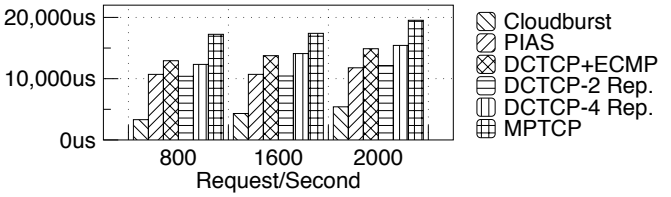

Fig. 13. p99 Message Completion Time

	800 r/s	1600 r/s	2000 r/s	Average
5KB	1.221	1.241	1.381	1.281
10KB	1.478	1.133	1.367	1.331
20KB	1.398	1.972	1.875	1.748
50KB	1.167	1.311	1.643	1.373
93KB	1.542	1.928	1.676	1.715
Average	1.361	1.517	1.591	1.490

Fig. 14. Coding rates of experiments in Figure 12,13

Coding Overhead: In Table 14, we list the coding rates in Figure 12&13, which is the ratio of the number of sent packets over the number of packets in the original message. The average coding rate is 1.49, i.e. Cloudburst adds $\sim 49\%$ more traffic load in the experiments.

C. Cloudburst Deep Dive

In this section, we use a series of targeted experiments to further understand Cloudburst.

Impact of incast: We first examine the incast scenario. We have 4 senders on one rack sending 90KB (60 pkts) messages to a receiver on the other rack, and set receiver's ToR switch buffer size to 100KB (with no traffic to other ports, the total switch buffer size for this port is the sum of shared and dedicated buffer). The flows start at the same time and are evenly distributed among the senders. We increase the number of concurrent flows, N, and measure the time from the start to the last flow. Figure 15 shows the average flow completion times (FCT) with the increase of N. We find that, as N grows larger, the performances of different schemes start to diverge, and DCTCP-based schemes (including PIAS) start to have increasingly longer FCT. In contrast, Cloudburst shows consistently low FCT under incast, and its FCT grows almost linearly with N. The key reason is that, unlike TCP, Cloudburst's aggressive burst-until-received protocol does not require a timeout to discover packet loss, which is bound to happen in incast. A Cloudburst flow in an incast proactively retransmits without need to discover a packet loss.

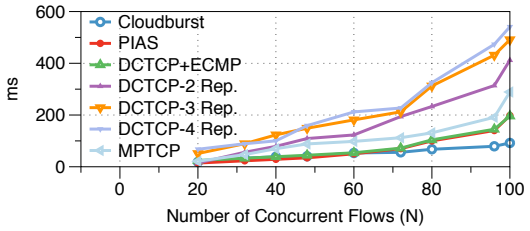

Fig. 15. Incast: Average Completion Time

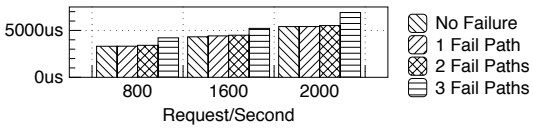

Fig. 16. Failure: p99 Completion Time

Impact of failures: When link/switch failures happen, some paths may become unavailable. We evaluate how Cloudburst handles such scenario. We vary the number of failed paths, and compare p99 FCT. As shown in Figure 16, with increasing number of failed paths, the p99 FCT increases gradually: for 2000rps, from 5412μs to 6907μs. As expected, the scenario with the most failed paths performs the worst. This is because, when there is only one path, the tiny buffer on the switch may drop many packets on this path, as Cloudburst senders continuously resend encoded packets on this path to recover the drops. This shows Cloudburst is sensitive to the existence of multipath, but not the number of available paths.

D. Large-scale Simulations

We complement the testbed experiments by simulating Cloudburst in a large DCN. We use a leaf-spine topology with 144 hosts, 9 leaf (ToR) switches, and 4 spine (Core) switches. Each leaf switch has $16\times$ 10Gbps downlinks and $4\times$ 40Gbps uplinks to the spine. The base RTT across the spine (4 hops) is 20μs. We generate the all-to-all traffic workload as above, and fix the traffic load to 2000rps. We implemented Cloudburst based on ns-2 simulation of QJump [57]. The en/decoding times of Cloudburst are obtained from experiments (not shown due to space limitation), and are added to Cloudburst's FCT measurement. We compare Cloudburst with the following:

- **DCTCP [1]:** We configure DCTCP with the recommended ECN marking threshold of 65 packets.
- **pFabric [21]:** Queue size is 34 packets ($2\times$BDP), initial window is 17 packets, and RTO$_{min}$ is 1ms.
- **QJump [23]:** Based on topology, we configure the minimum bandwidth $R = 10$Gbps, cumulative switching delay $\epsilon = 4$us, $P = MTU = 1.5$KB. For messages, we set the throughput factor $f = 1$ (guaranteed latency); for background flows, we set $f = n$ (maximum throughput. $n = 144$, the total number of end-hosts).
- **Expresspass [42]:** For credit packet, the packet size is 84 bytes and the queue size is 10 packets; for data packet, the packet size is 1538 bytes and the queue size is 100 packets. The credit rate is 500Mbps.

Tail latency: We first compare the tail latency. Figure 17 shows p99 FCT. For small message size (5KB), the difference

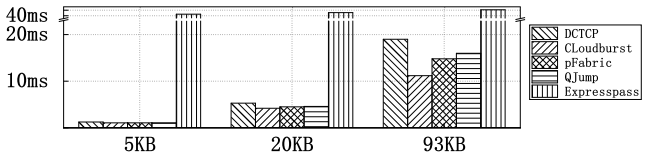

Fig. 17. p99 Completion Time

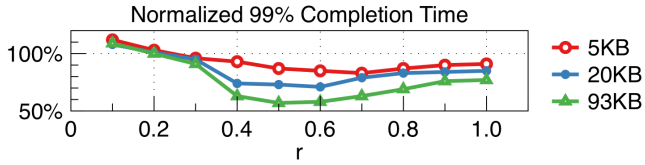

Fig. 18. Choosing r in 2:1 over-subscribed network

is insignificant. As size grows larger, Cloudburst begins to show its advantage over the others. The main reason is that TCP-like schemes takes at least RTO$_{min}$ to recover loss. Since packet loss is captured by the tail latency, Cloudburst outperforms DCTCP, pFabric, and QJump by 40.12%, 24%, and 29.63%, respectively. To our surprise, our result shows that Expresspass performs the worst among all the schemes. We imagine there are two main reasons: 1) it requires one additional RTT for credit allocation, and 2) such credit-based algorithm is fundamentally not suitable for mice flows in a dynamically changing environment, because it is very hard to set up an appropriate amount of credits for short-live flows.

Impact of over-subscription: We further examine how Cloudburst performs in over-subscribed networks. We reduce all ToR-to-Core links' capacity to 20Gbps, creating a network with 2:1 over-subscription. We run this experiment for DCTCP and Cloudburst. For Cloudburst, we vary r, the rate limit in Algorithm 1, from 0.1 (sending at 1Gbps) to 1 (10Gbps), and collect MCTs. We summarize the results in Figure 18 normalized to DCTCP. We find that, for varying sizes, choosing r close to the over-subscription ratio results in higher tail latency reduction. If r is too small, the sender is not sending enough to compensate for the aggressive dropping, and the message can take longer to finish. If r is too large, the senders may overload the network and cause more frequent packet drops, which is bad for longer messages. Overall, when r is chosen appropriately (~ 0.5), the MCT reduction is $> 12.88\%$ for all message sizes. Therefore, we suggest setting the sending rate r (in Algorithm 1) to the over-subscription ratio.

VI. Conclusion

We present the design, implementation, and evaluation of Cloudburst — a simple scheme to cut long tail latency of message delivery by proactively sending FEC-coded packets generated from the messages on multipath in parallel, thus avoiding complexities like prior solutions. Cloudburst is readily deployable in today's commodity DCNs. We implemented a Cloudburst prototype, and validated its performance with extensive testbed experiments as well as large-scale simulations.

Acknowledgement: This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF-16215119 and GRF-16213621. We thank the feedback from the anonymous reviewers. Kai Chen is the corresponding author of this paper.

REFERENCES

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," In ACM SIGCOMM 2010.

[2] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-Agnostic Flow Scheduling for Commodity Data Centers," In NSDI 2015.

[3] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: reducing the flow completion time tail in datacenter networks," In ACM SIGCOMM 2012.

[4] B. Vamanan, J. Hasan, and T.N. Vijaykumar. "Deadline-aware datacenter tcp (D2TCP)," In ACM SIGCOMM 2012.

[5] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication," In ACM SIGCOMM 2009.

[6] W. Bai, S. Hu, K. Chen, K. Tan, Y. Xiong, "One More Config is Enough: Saving (DC)TCP for High-speed Extremely Shallow-buffered Datacenters," In IEEE INFOCOM 2020.

[7] Y. Yao, "An effective go-back-N ARQ scheme for variable-error-rate channels," In IEEE Transactions on Communications 43, 1 (1995).

[8] E. Blanton, K. Fall, and M. Allman, "A conservative selective acknowledgment (SACK)-based loss recovery algorithm for TCP," 2003.

[9] C. E. Hopps, "Analysis of an equal-cost multi-path algorithm," 2000.

[10] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, "Pingmesh: A large-scale system for data center network latency measurement and analysis," In ACM SIGCOMM 2015.

[11] G. Chen, Y. Lu,YuanMeng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang, "Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers," In USENIX ATC 2016.

[12] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over Commodity Ethernet at Scale," In ACM SIGCOMM 2016.

[13] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, "CONGA: Distributed congestion-aware load balancing for datacenters," In ACM SIGCOMM 2014.

[14] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," In ACM SIGCOMM 2014.

[15] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient Datacenter Load Balancing in the Wild," In ACM SIGCOMM 2017.

[16] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: high precision congestion control," In ACM SIGCOMM 2019.

[17] 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 ACM SIGCOMM 2015.

[18] R. Mittal, 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 ACM SIGCOMM 2015.

[19] G. Kumar, N. Dukkipati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat, "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter," In ACM SIGCOMM 2020.

[20] G. Zeng, S. Hu, J. Zhang, and K. Chen, "Transport Protocols for Data Center Networks: A Survey," In Journal of Computer Research and Development 2020.

[21] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-optimal Datacenter Transport," In SIGCOMM 2013.

[22] A. Munir, G. Baig, S. Irteza, I. Qazi, I. Liu, and F. Dogar, "Friends, not foes: synthesizing existing transport strategies for data center networks," In ACM SIGCOMM 2014.

[23] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues Don't Matter When You Can JUMP Them!" In USENIX NSDI 2015.

[24] P. Cheng, F. Ren, R¿ Shu, and C. Lin, "Catch the whole lot in an action: rapid precise packet loss notification in data centers," In USENIX NSDI 2014.

[25] R. Mahajan, J. Padhye, S. Agarwal, and B. Zill, "High performance vehicular connectivity with opportunistic erasure coding," In USENIX ATC 2012.

[26] H. Harry Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," In ACM SIGCOMM 2014.

[27] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," In ACM SIGCOMM 2008.

[28] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C Snoeren, "Inside the Social Network's (Datacenter) Network," In ACM SIGCOMM 2015.

[29] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," In ACM SIGCOMM 2015.

[30] T. Benson, A. Akella, and D. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," In IMC 2010.

[31] D. Clark, S. Shenker, and A. Falk, "GENI Research Plan (Version 4.5)," GENI Research Coordination Working Group and GENI Planning Group. GDD (2007).

[32] B. Raghavan, and A. C. Snoeren, "Decongestion control," In ACM HotNets 2006.

[33] Pica8 PICOS Configuration Guide. https://docs.pica8.com. Accessed on July 2021.

[34] G. Zeng, W. Bai, G. Chen, K. Chen, D. Han, Y. Zhu, and L. Cui, "Congestion Control for Cross-Datacenter Networks," In IEEE ICNP 2019.

[35] S. Floyd, "TCP and explicit congestion notification," In ACM SIGCOMM 1994.

[36] G. Zeng, W. Bai, G. Chen, K. Chen, D. Han, and Y. Zhu, "Combining ECN and RTT for Datacenter Transport," In ACM APNet 2017.

[37] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP," In USENIX NSDI 2011.

[38] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and ro- bustness with multipath tcp," In ACM SIGCOMM 2011.

[39] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren, "Re-architecting Congestion Management in Lossless Ethernet," In USENIX NSDI 2020.

[40] IEEE. 802.11Qbb, "Priority based flow control," 2011.

[41] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," In ACM SIGCOMM 2017.

[42] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," In ACM SIGCOMM 2017.

[43] G. Zeng, J. Qiu, Y. Yuan, H. Liu, and K. Chen, "FlashPass: Proactive Congestion Control for Shallow-buffered WAN," In IEEE ICNP 2021.

[44] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," In ACM SIGCOMM 2018.

[45] S. Hu, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang, "Aeolus: A Building Block for Proactive Transport in Datacenters," in ACM SIGCOMM 2020.

[46] S. Hu, G. Zeng, W. Bai, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang, "Aeolus: A Building Block for Proactive Transport in Datacenter Networks," in IEEE/ACM ToN 2021.

[47] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," In ACM SIGCOMM 1998.

[48] M. Luby, "LT codes," In Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002.

[49] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ Forward Error Correction Scheme for Object Delivery," In RFC 6330, 2011.

[50] T. Bonald, M. Feuillet, and A. Proutiere, "Is the 'Law of the Jungle' Sustainable for the Internet?" In IEEE INFOCOM 2009.

[51] Rust Programming Language. https://www.rust-lang.org.

[52] PicOS Routing & Switching Command Ref. https://shorturl.at/eixFG. Accessed on July 2021.

[53] QoS: Congestion Management Configuration Guide, Cisco IOS XE Release 3S. https://shorturl.at/svAIX. Accessed on July 2021.

[54] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," In RFC 3031, 2001.

[55] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo, "Explicit Path Control in Commodity Data Centers: Design and Applications," In USENIX NSDI 2015.

[56] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," In ACM CoNEXT 2013.

[57] QJUMP ns-2. https://github.com/camsas/qjump-ns2. Accessed on July 2021.