Published in the IEEE Internet of Things Journal, 2020 https://doi.org/10.1109/JIOT.2020.2996800

Efficient Multisource Data Delivery in Edge Cloud With Rateless Parallel Push

Shouxi Luo, Tie Ma, Wei Shan, Pingzhi Fan, Huanlai Xing, Hongfang Yu

***Abstract*—As the key infrastructure for emerging 5G and IoT applications, micro data centers would be widely deployed at network edges to provide high-bandwidth low-latency cloud service. In these systems, applications would deliver large-size data objects among servers for various purposes like service deployment, application scale-up, and data duplication on demand. Accordingly, reducing the delivery time is crucial for the optimization of service delay and system utilization.**

To accelerate the delivery, this paper proposes a multi-source-aware adaptive data transmission solution, Parallel Push (PPush), by leveraging the fact that data objects in cloud are generally replicated among servers by design. At the high-level, PPush achieves efficient delivery of multi-source data by launching multiple *push* flows in parallel; and at the low-level, it decouples transfers from different sources by encoding data objects with rateless RaptorQ code, and further employing novel congestion controls to prioritize the bandwidth allocation of concurrent tasks respecting their remaining sizes. Fluid model analysis along with Mininet-based test and packet-level simulation shows that, unlike DCTCP and other proposals, *push* is robust to packet loss and achieves provable prioritized bandwidth allocation. Extensive simulation results imply that, with above advantages, PPush could achieve very efficient data delivery by making use of all available data sources: for instance, compared with the straightforward design of equal-size task split and fair bandwidth allocation, its adaptive task assignment and prioritized traffic scheduling reduce the average task completion time in a tested scenario by 1.495× and 1.329×, respectively, demonstrating a total improvement of 1.586×, when enabled at the same time.

***Index Terms*—Edge cloud, data delivery, congestion control, prioritized bandwidth allocation**

I. Introduction

NOWADAYS, to provide high-throughput, low-latency, and cost-efficient cloud service for emerging 5G and Internet of Thing (IoT) applications, increasing amounts of internet service, cloud, and content providers are extending their data center infrastructures to the network edge, making the wave of edge cloud computing [1]. Essentially, these edge data centers are micro- or middle- scale clusters, which would host various distributed applications like content delivery, website hosting, computation offloads, bigdata analysis, machine learning, and so on [1, 2]. In practice, servers in these clusters commonly need to deliver large-size data objects for various purposes on demand. For example, to launch a new microservice or cloudlet, a selected set of servers would pull the involved Virtual Machine (VM) or container images from the registries [3]; likewise, to scale up model inference service, distributed Artificial Intelligence (AI) applications would duplicate the trained model (with up to thousands megabytes of parameters) [4] to newly launched servers; and to recover from server failures or to improve the availability of critical data, distributed storage applications might replicate data blocks to different carefully selected slave nodes [5, 6]. Accordingly, to reduce service delays and improve the utilization of infrastructure, resource-bounded edge data centers must achieve very efficient delivery of data objects in practice.

By looking into the design principles of these applications, we find that the involved data objects are generally replicated among hosts and racks for various purposes by design. Indeed, such a multi-source nature of data objects also brings with the new opportunity of parallel transmission for the acceleration of their deliveries [7]. For instance, to reduce service deployment time, container images for microservice and cloudlet would get hosted on multiple replicated registries [3]; consequently, by launching multiple cooperative pull requests from all the registries, it is possible to further reduce the time of image downloads in service deployment and cloudlet initialization. Likewise, to guarantee high data availability, each data chunk in distributed file systems like GFS [5] and HDFS [6] is replicated among 3 or 5 or even more carefully selected servers; accordingly, for model parameters stored in these systems, by fetching different parts of the trained model from back-end storage nodes and existing inference nodes in parallel, AI applications are able to cut down the time of model preparation greatly when scaling up.

However, such a design is not widely used in production so far, as there does not exist a practical and generic protocol that would make efficient use of the multi-source nature of the data. Although several production clusters have tried the alternative design of peer-to-peer (P2P) mechanisms—splitting the target object into pieces and transmitting them from all available source nodes collaboratively, existing solutions like BitTorrent and its variations Murder [8] and Cornet [9], are highly tailored and far from optimal on two aspects. Firstly, to download the same object from multiple sources collaboratively, P2P systems run slow and sophisticated peer protocols for the search and delivery of pieces dynamically [10],

The work of Shouxi Luo is supported in part by China Postdoctoral Science Foundation (2019M663552) and Fundamental Research Funds for the Central Universities (2682019CX61); the work of Pingzhi Fan is supported in part by NSFC Project (61731017) and National Key R&D Project (2018YFB1801104); and the work of Hongfang Yu is supported in part by National Key R&D Project (2019YFB1802803) and PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (PCL2018KP001). (Corresponding author: Shouxi Luo, Hongfang Yu.)

S. Luo, P. Fan, and H. Xing are with the School of Information Science and Technology, Southwest Jiaotong University, Chengdu 611756, China (e-mail: {sxluo, pzfan, hxx}@swjtu.edu.cn).

T. Ma and W. Shan are with Xidian University, Xi'an 710071, China (e-mail:matie_xidian@163.com, xdshanwei@126.com).

H. Yu is with University of Electronic Science and Technology of China, Chengdu 611731, China (e-mail: yuhf@uestc.edu.cn).

with the cost of non-trivial control overheads [8]. Making fixed delivery task assignment to source nodes ahead would eliminate the overheads, however, such a design could not react to network variations like source failures and bandwidth changes, resulting in inflexibility and inefficiency. Secondly and more critically, as prior study has shown [11–13], to optimize the average task completion times, applications prefer prioritized bandwidth allocations. Unfortunately, existing peer protocols are generally built upon either TCP or uTP [10, 14], which pursues fair bandwidth allocations by design and might cause incast problem [11]. Despite a lot of novel prioritized protocols have been proposed, they *i*) could not scale [15], or *ii*) are inefficient with the problem of slow start [9] or no performance guarantees [16, 17], or *iii*) are hard to deploy because of their non-trivial modifications on switch hardware [7, 12, 13, 18] or exclusive use of all the limited priority classes [19]. There also exist some proposals that require no hardware modification [20]; however, like most of aforementioned solutions [12, 13, 16–19, 21], they are unaware of the multi-source nature of data, leading to sub-optimal.

To overcome these shortcomings and provide an efficient object delivery service for cluster application, this paper proposes PPUSH, a practical multi-source aware transmission protocol. At the high level, PPUSH achieves low-overhead multi-source data delivery with the assistance of a centralized controller. To be scalable, the controller's job is minimal. For each data object, the controller maintains the set of its active source nodes and works as the front-end resource server waiting for receivers' fetch requests. On getting a fetch, it selects a group of source nodes to *push* different parts of the object via UDP in parallel. Then, for each received packet, the receiver generates an ACK to the corresponding sender for acknowledgment and periodically reports its remaining task size to the controller (we refer the packets sent by a source node along with the triggered ACKs as a *push* flow). Once the receiver has obtained enough packets to recover the original object, it terminates all *push* flows with FIN packets.

At the low level, PPUSH *i*) lets source nodes send diverse RaptorQ coded packets instead of the raw content, to avoid duplicated transmission without complex explicit collaboration [7] (§III-A), and *ii*) develops a DCTCP-alike sim-reliable congestion control algorithm to achieve configurable weighted fair sharing for concurrent *push* flows (§III-B,§III-C). RaptorQ is a systematic and rateless coding scheme widely used for forward error correction (FEC). With negligible overheads [7], it makes PPUSH robust to packet loss, thus avoiding the problem of head-of-line blocking, and more importantly, decoupling the transmission tasks of different sources. As a result, source nodes can send data as fast as they can without worrying about duplicated transmission; and also, the entire delivery task could tolerate both switch and source failures. Regarding the congestion control, the algorithm employed by *push* is distinguished from that of DCTCP with two novel designs. First, as *push* flows tolerate packet loss, they employ explicit- instead of cumulative- acknowledgments to avoid head-of-line blocking [22]. Second, to prioritize bandwidth allocations for the optimization of average task completion time, *push* flows with the lower priority are set to be more sensitive to ECN markings: consider that a *push* flow's current running average of the fraction of ECN mark is α; then, on getting an ECN mark, its sender would reduce the congestion window size by $\alpha/(2\theta)$, rather than by the original $\alpha/2$ suggested by DCTCP [22]. Here, θ denotes the level of sensitivity at which it reacts to ECN mark. As we prove in §IV, such a simple design enables concurrent *push* flows to share the bottleneck bandwidth in proportion to their θs. By dynamically updating *push* flows' θ-parameters according to their remaining task sizes via a controller, PPUSH could optimize their average completion times without upgrading switch hardware (§III-C).

We not only analyze *push* protocol with fluid models and numerical techniques, but also prototype PPUSH in Mininet and develop a packet-level event-driven simulator for detailed large-scale performance study. Extensive results show that, PPUSH converges very fast and would make provable weighted fair yet work-conserving bandwidth allocation. By leveraging the adaptive task assignment and θ-based traffic prioritization, it outperforms the straightforward solution (i.e., fixed task split and fair bandwidth allocation) up to 1.6× on tested workloads.

In summary, our contributions are:

- **PPUSH**, a multi-source aware, loss-tolerable transport protocol achieving very efficient delivery of multi-source data objects with RaptorQ-decoupled parallel pushing.
- **Weighted fair congestion control (WFCC)**, an easy-to-implement congestion control algorithm that achieves provable weighted fair sharing bandwidth allocation for concurrent flows without specific hardware support. Besides PPUSH, WFCC can be used for many other protocols including DCTCP, QUIC, as well.
- **Verified mathematical proof**, a fluid model based mathematical analysis along with Mininet-based prototype and packet-level simulation proving that WFCC enforces controllable weighted fair bandwidth allocations.
- **Extensive simulation**, detailed performance evaluations showing that PPUSH is robust to packet loss, performs graceful traffic scheduling, and achieves very efficient delivery of multi-source data objects.

The rest of this paper processes as follows. Section II briefly analyzes the design challenges, then sketches the core idea of PPUSH. Section III presents the design details. Section IV further analyzes the WFCC employed by PPUSH with fluid models, and verifies the theoretical findings with both Mininet-based test and packet-level simulation. Before reviewing the related work in Section VI, extensive performance evolution follows in Section V. Finally, Section VII concludes the paper.

II. Problem Analysis

In this section, we first overview the practical limits and challenges of designing new transport protocols for multi-source data delivery in edge data center (§II-A), then overview how PPUSH achieves the goal with novel designs (§II-B).

A. Design Challenges

To make efficient and practical parallel *push* in edge data center, the following challenges must be addressed.

Sender node dynamic. As is known, both node and link failures are common to occur in today's data centers since *i*) servers are generally built from inexpensive commodity components for cost effectiveness and *ii*) network updates are error-prone [5, 23–25]. Thus, some source nodes of a delivery task might be unavailable because of failure, or conversely, become available during the transmission because of automated repairs [5, 24]. To achieve the best performance, the proposed solution should be able to make use of dynamically available source nodes for the delivery on a routine basis.

Low cooperation overheads. Intuitively, the straightforward design to achieve efficient object delivery upon dynamical source nodes is to directly employ off-the-shelf P2P file sharing applications (e.g., BitTorrent). However, their peer protocols are sophisticated, slow, and would introduce non-trivial overheads [8, 10]. The alternative is to build a centralized controller to control all the traffic [15]. Unfortunately, such a design is impractical as well, since it is agnostic on bursty link congestion and might underuse link capacities because of its scheduling latency. Moreover, the controller would become the bottleneck as it is too involved with the scheduling.

Unpredictable packet loss. In practice, multiple types of traffic generally coexist in the network. Some of them might trigger micro-bursts [26] (e.g., incast [11]), which would build up the shallow switch queue very quickly and cause unpredictable packet drops. Hence, on one hand, the proposed protocol should be able to tolerate these bursty packet drops; while on the other, it should keep queue utilization rate low. Furthermore, to guarantee low latency for time-sensitive traffic, the impacts of traffic triggered by the considered background data delivery tasks on the other, must be isolated (e.g., use the lowest priority for data delivery [27]).

Limited priority queue. As prior study has shown, the key to optimize the average transfer completion times is to dynamically prioritize their traffic respecting remaining sizes [12]. To be easy-to-deploy, the proposed solution should avoid switch modifications; the state-of-the-art solutions implement this type of schedule by directly employing the priority queues provided by switch hardware then updating the assignment of priorities dynamically. However, this design is impossible in our settings, since current switch hardware supports only $4 \sim 8$ queues, most of which are already reserved for other applications or services [21]. Hence, there is only the lowest queue that PPUSH can use for traffic prioritization.

B. Solution Overview

Based on above analysis, we develop PPUSH, a readily-deployable data center transport protocol, to achieve efficient delivery of multi-source data objects. Basically, as Figure 1 sketches, PPUSH can make full use of all the available source nodes with the help of a centralized controller (§III-A). To be scalable, the job of the controller is minimal: it *i*) only maintains the active set of source node for each delivery task and *ii*) updates the corresponding congestion control parameter according to remaining task sizes in period (§III-C). To remove cooperation overheads and tolerate packet drops,

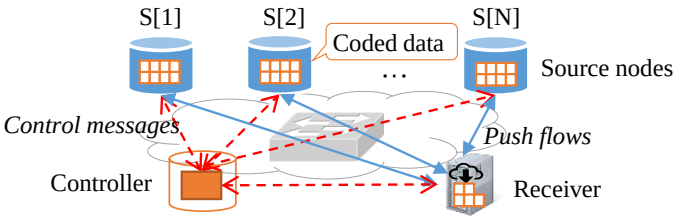


Fig. 1: PPUSH Architecture

PPUSH *i*) encodes multi-source data objects with RaptorQ code and *ii*) has source nodes partitioning the stream of coded packets to avoid duplicated transmission distributively [28]. More specifically, the source node first splits the data object into blocks, and further encodes each of them into a stream of equal-size, UDP-encapsulatable unique symbols via fountain coding technique [28]. Then, *push* flows send different parts of the coded symbols of consecutive blocks in UDP flows. After getting any sufficient subset of encoded symbols, the receiver recovers each original data block via decoding. To achieve prioritized bandwidth allocations while keeping low queue occupancy, PPUSH employs a DCTCP-alike weighted fair congestion control algorithm for each *push* flow (§III-B).

To make efficient use of the residual bandwidth without preempting other time-sensitive traffic, PPUSH tags packets with the Differentiated Services Code Point (DSCP) value bound with the lowest weight. Then, the weighted fair queuing in switch would ensure that these *push* flows would make very graceful use of the available link capacity.

In the following, we first introduce the design detail of PPUSH in §III then illuminate why the proposed congestion control yields provable weighted fairness in §IV.

III. PPUSH Protocol

In this section, we first present the detail of how PPUSH executes parallel *pushes* (§III-A), then describe the weighed fair congestion control (WFCC) algorithm that enforces controlled weighted fair bandwidth allocation for concurrent *push* flows (§III-B), and finally depict how PPUSH achieves prioritized schedule of delivery tasks for the optimization of their average completion times by updating their congestion control parameters via a logical central controller dynamically (§III-C).

A. Architecture and Workflow

As Figure 1 shows, PPUSH involves three types of nodes, namely, *receiver*, *controller*, and *source node*, respectively.

Receiver. To fetch a data object, the receiver sends a request along with the Uniform Resource Identifier (URI) of the resource, its device (or user) ID to the corresponding controller via UDP, then listens on the same UDP port for acknowledgment and task summary from the controller, and coded symbols of data blocks from source nodes. If no acknowledgment is obtained within a given time, it retries until it gets one. During the receiving, the receiver periodically reports the first un-decoded block's index along with the remaining number of needed symbols to the controller. Such a feedback message works as heartbeats indicating the receiver's activity

and progress; thus, the controller can launch new *push* flows for it in case new source nodes are online. On getting a coded data symbol from the source node, the receiver immediately generates an acknowledgment message (refer as ACK for short hereafter), indicating which symbol is received along with how many symbols are still needed from this sender for the corresponding block's decoding,[1] to the source. Moreover, this ACK would have an ECE flag enabled if the received IP packet is marked with ECT. Once the receiver has obtained whole of the original data object via decoding, it sends FINs to the controller and all its active data senders. To guarantee graceful termination, the FIN to controller would get resent if no acknowledgment is obtained.

Controller. The controller in PPUSH is essentially the manager of distributed source nodes and simultaneously acts as the front-end server of data delivery. On getting a fetch request, it records the identification as well as the IP address and UDP port of the receiver, then emits an acknowledgment. In the meantime, it dynamically selects a subset of active source nodes containing the desired data object, to dispatch coded data symbols to the receiver via *push* flows. To avoid the problem of incast [9, 11], the number of selected senders should be limited (e.g., by 3 for instance). During the delivery, based on the reported remaining task sizes, it periodically recomputes and adjusts the congestion control parameters of all *push* flows, such that concurrent *push* flows would achieve controlled weighted fair sharing on bottleneck links. In case the controller does not get any update of a receiver for a long time, it pauses all the corresponding *push* flows since the receiver is likely to be offline. Finally, on getting the FIN, it replies an acknowledgment and stops all the corresponding *push* flows immediately.

Source node. In PPUSH, source nodes are the back-end storage nodes holding the coded blocks of data objects, managed by the controller. To keep the computation overhead and latency introduced by RaptorQ decoding on the receiver small, large-size data objects in PPUSH would be split into small blocks for pipelined delivering and decoding. On getting a *push* task from the controller, the source node sends all or parts of the encoded symbols of the data object to the receiver. As the parameter settings for both the splitting and encoding of data objects keep consistent among all source nodes; these selected source nodes would have exactly the same data blocks and encoded symbols for the same data object.

Suppose that at most M source nodes would host the requested data object; and encoded symbols of block j are indexed with $0, 1, 2, \cdots$. To avoid duplicated transmission, the i-th node only sends the symbols with index $i + kM$, where $k = 0, 1, 2, \cdots$ for block j. Similar to QUIC [29], packets in each *push* flow carry with monotonically-increasing packet numbers indicating their transmission orders. As well, during the delivery, sent symbols would get acknowledged by the receiver explicitly. Thus, the sender can speculate which symbols are lost. The acknowledgment also specifies how many symbols from this sender are still required by the receiver to decode the block. For each un-decoded block, the sender would resend the lost symbols if there are, or emit new ones, until the block gets decoded successfully. However, if the sender switches to send the coded symbols of block $j+1$ only after it gets the decoded confirmation of block j, there would be about one Bandwidth Delay Product (BDP) of in-flight packets along the way belonging to block j because of the acknowledgment delay. To reduce these redundant packets, if the number of sent-yet-unacknowledged packet belonging to block j is larger than the required amount specified by the receiver, the sender would deliver the symbols of block $j + 1$ to use the remaining bandwidth gracefully.

Like QUIC [29], the emission of coded symbol packet in each *push* flow is also controlled by a congestion window. As detailed in §III-B and §III-C, based on the congestion signals implied by acknowledgments along with the parameter dynamically specified by the controller, *push* senders update their window sizes with a DCTCP-alike congestion control algorithm, achieving prioritized bandwidth allocations.

B. Weighted Fair Congestion Control

To not overload the network, source nodes employ congestion windows to control the sending of coded blocks and guarantee that the number of sent-yet-unacknowledged blocks[2] would not exceed the current congestion window size. During the delivery, PPUSH would dynamically update the size of congestion window based on the arrival of acknowledgment; the employed algorithm is quite similar to that of DCTCP [22]. In the following, we mainly describe their differences.

Firstly, as the acknowledgment sent by receiver is not cumulative, packet drops in PPUSH could not be informed from duplicated acknowledgments. Accordingly, the source node considers a packet lost after at least DUPACKNUM (typically 3) packets sent after it have been acknowledged. Secondly, to achieve prioritized bandwidth allocations among concurrent *push* flows, transfers with the lower priority in PPUSH is set to be more sensitive to ECN markings. Like DCTCP, the source node maintains a running average of fraction of ECN-marked packets (α) via Equation (1) for each window of data, where F is the fraction of packets marked in the most recent window, and $g \in (0, 1)$ is a fixed parameter. Suppose the current congestion window size is W. On getting an acknowledgment, W would be updated following Equation (2). Here, θ is a parameter configured by the controller specifying the degree of sensitiveness at which this *push* flow reacts to ECN markings.

$$\alpha \leftarrow (1 - g)\alpha + gF \tag{1}$$

$$W \leftarrow W + \begin{cases} 1/W, & \text{if ECN} = 0 \\ 1/W - \alpha/(2\theta), & \text{if ECN} = 1 \end{cases} \tag{2}$$

Indeed, as Theorem 1 clarifies, by dynamically updating θs respecting the remaining task sizes of their transfers, PPUSH

[1] For a block whose original symbol size is u, if the receiver has received v coded symbols, the amount of remaining desired coded symbols for each of its N active source, can be estimated by $\lceil \frac{u-v+2}{N} \rceil$ according to [28]. Moreover, to tolerate the possible drop of ACK packet, we can let each ACK confirm the receipts of multiple recently received symbols for redundancy.

[2] Note that, if a block is considered lost, it would not be treated as sent-yet-unacknowledged any more.

is able to perform prioritized yet starvation-free bandwidth allocation to optimize their average computing times. In the following, we describe how PPUSH achieves such optimizations with the assistance of controller, and leave the proof and verification of why such a design works in §IV.

Theorem 1. *Given a group of long-lived push flows through the same bottleneck link, in their stable states, the bandwidth obtained by the i-th flow is in proportion to its parameter* θ_i.

C. Task-level Traffic Prioritization

PPUSH is designed to minimize the average completion times of concurrent tasks. Although this problem is proven to be NP-hard in theory [12], abundant prior study has shown that it is easy to achieve near-optimal average completion time minimization by simply prioritizing concurrent flows in non-decreasing order of their remaining sizes [12, 13, 21]. Thus, to optimize the average completion times, PPUSH should configure *push* flows' θ values respecting the order of their remaining sizes and let the gap between θs as large as possible.

However, as the analysis in §IV will show, to make the bandwidth of *push* flow converge fast and to improve its stability in practice, $\{\theta_{i,j} : e \in p_{i,j}\}$, the θ-values of *push* flows $\{p_{i,j}\}$ going through the same bottleneck link e, should satisfy the requirements of $\theta^- \leq \theta_{i,j}$ and $\sum_{(i,j):e\in p_{i,j}} \theta_{i,j} \leq \theta^+$ simultaneously. Here, $p_{i,j}$ denotes the j-th *push* flow (and its path) belonging to delivery task i; θ^- and θ^+ are tunable constants that bound the choose of θs (see §IV-A for the computations of θ^- and θ^+). It is observably that, as prioritized bandwidth allocation is pursued, flows with the highest priority would take almost all of link bottleneck capacity [21]. Accordingly, on each link, we can let *push* flows with the smallest remaining task size take the most of θ^+ and all other flows use the lowest θ^-. In case the smallest task involves multiple flows in the same bottleneck; all these flows can share the same θ value fairly. Consider that there are n *push* flows through link e, among which m flows belong to the smallest task. Then, the value of these high-priority *push* flows can be set as $\frac{\theta^+-\theta^-(n-m)}{m}$. Moreover, in practice, there might be multiple bottlenecks. To avoid inconsistent θs assignment among links, PPUSH should manage θs with a global view.

Taking all these aspects into account, PPUSH updates flow θs upon every *push* task arrival and completion event as Algorithm 1 specifies. Consider that there are N active data delivery tasks, $f_1, f_2, \cdots, f_N$, in which, task f_i is with the remaining value of v_i served by a group of *push* flows: $P_i = \{p_{i,1}, p_{i,2}, \cdots, p_{i,M_i}\}$. For the sake of description, we also use the symbol of $p_{i,j}$ to denote the path and the set of links involved by this *push* flow; in case task i only involves one single *push* flow, we use f_i to indicate this path as well. Then, the assignment of their θs is conducted in non-decreasing order of their remaining task sizes (Line 5). Based on the design, if multiple tasks go through the same link, at most one of them could use the large-θ value and all other tasks should use θ^- instead. Thus, we treat the large-θ value as a specific type of exclusive link resource (i.e., *token*) and use E to denote the set of links that already allocate their large-θ token to tasks.

Algorithm 1: Consistent θ assignment

Input : Current active delivery tasks $\{f_i\}$ and their states
Output: $\{\theta_{i,j}\}$, the θ value of each *push* flow

```
1  E ← ∅ // the set of links whose large-θ tokens are already allocated
2  F ← Sort {f_i} according to {v_i} non-decreasing
3  χ[e] ← 0 for each link e // the number of flows through e
4  π[e] ← [] for each link e // high-priority flows through e
5  foreach f_i ∈ F do
6      E_p ← ∅
7      foreach p_{i,j} ∈ P_i do
8          if p_{i,j} ∩ E ≠ ∅ then
9              θ_{i,j} ← θ^-
10         else
11             θ_{i,j} ← θ^+
12             E_p ← E_p ∪ p_{i,j}
13         foreach e ∈ p_{i,j} do
14             χ[e] ← χ[e]+1
15             if θ_{i,j} = θ^+ then
16                 Append p_{i,j} to π[e]
17     E ← E ∪ E_p
18 foreach e ∈ π do
19     r ← max(1, (θ^+ - θ^-(χ[e]-len(π[e]))) / len(π[e]))
20     foreach p_{i,j} ∈ π[e] do
21         θ_{i,j} ← min(θ_{i,j}, r)
```

Then, $p_{i,j}$, the j-th *push* flow of tasks f_i, would get this large-θ token, if and only if there are enough large-θ tokens along its path (Line 8 to 12). Moreover, if two *push* flows belonging to the same task go through the same bottleneck link, they can use the same large-θ token for fair sharing. Hence, E is updated in a per-task, rather than per-flow basis (Line 17). Finally, the θ value of a *push* flow is bounded by the smallest allocated value among its path (Line 21). Regarding the value of θ^+ and θ^-, they are easy to estimate as §IV-A shows.

IV. Why It Works

In this section, we first prove Theorem 1 via fluid model based analysis (§IV-A), then show that *push* flows would converge to their weighed fair stable states very fast via both Mininet implementation and packet-level simulation (§IV-B). Table I summarizes the involved notations.

A. Fluid Model Based Analysis

Consider that there are N long-lived *push* flows going through the same bottleneck link of capacity C. With parameter θ_i, the i-th *push* flow obtains the congestion window of $W_i(t)$ at time t. At the switch side, we assume that the queue occupancy at time t is $q(t)$ and its ECN marking threshold is

TABLE I: Table of notation

Term	Meaning
N	Number of flows going through the bottleneck link
d	Round-trip propagation time
K	ECN marking threshold
C	Capacity of the bottleneck link
t	Time
$q(t)$	Queue occupancy at time t
$\bar{q}$	Average value of $q(t)$ when the network is stable
θ_i	The i-th flow's θ value
$W_i(t)$	The i-th flow's congestion window size at time t
$\bar{W}$	Average value of $W_i(t)$ when the network is stable
$\alpha(t)$	Running average of fraction of ECN-marked packets at time t
$\bar{\alpha}$	Average value of $\alpha(t)$ when the network is stable
g	The weight given to new samples in computing $\alpha(t)$
$p(t)$	1 if the current packet is ECN-marked; 0 otherwise
$\bar{p}$	Average value of $p(t)$ when the network is stable
$R(t)$	Value of the round-trip time at time t
$\bar{R}$	Average value of $R(t)$ when the network is stable
T	"Sawtooth" periodicity of flows when the network is stable

K. Let $p(t)$ denote the packet marking process at the switch; then we have

$$p(t) = \begin{cases} 1, & \text{if } q(t) > K \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

We further assume that all flows have the equal propagation delay of d and their rates are not too small. Accordingly, all flows will experience the identical round-trip time: $R(t)$, which is made of the round-trip propagation time (d), along with the queuing delay, as Equation (4) defines; and their proportions of encountered ECN markings, would be the same: $\alpha(t)$.

$$R(t) = d + q(t)/C \tag{4}$$

Essentially, *push* flows perform DCTCP-alike congestion controls. Thus, analogous to DCTCP flows [22], they would finally enter steady "sawtooth" patterns as our simulations in §IV-B show. Consider that, after time t, *push* flows are all in their stable states with the "sawtooth" periodicity of T. Then the dynamics of $W_i(t)$, $\alpha(t)$, and $q(t)$ in their stable states can be formulated as non-linear, delay-differential equations:

$$\frac{dW_i(t)}{dt} = \frac{1}{R(t)} - \frac{W_i(t)\alpha(t)}{2R(t)\theta_i} p\left(t - \bar{R}\right) \tag{5}$$

$$\frac{d\alpha(t)}{dt} = \frac{g}{R(t)}\left(p\left(t - \bar{R}\right) - \alpha(t)\right) \tag{6}$$

$$\frac{dq(t)}{dt} = \sum_{i=1}^{N} \frac{W_i(t)}{R(t)} - C \tag{7}$$

in which, $\bar{R}$ acts as the approximately fixed value for the delay of ECN notification as (8) defines.

$$\bar{R} = \frac{1}{T}\int_t^{t+T} R(t)dt = d + \frac{\int_t^{t+T} q(t)dt}{T}\frac{1}{C} \tag{8}$$

Similarly, we further define the average values of $W_i(t)$, $\alpha(t)$, and $q(t)$ by (9), (10), and (11), respectively.

$$\bar{W}_i = \frac{1}{T}\int_t^{t+T} W_i(t)dt \tag{9}$$

$$\bar{\alpha} = \frac{1}{T}\int_t^{t+T} \alpha(t)dt \tag{10}$$

$$\bar{q} = \frac{1}{T}\int_t^{t+T} q(t)dt \tag{11}$$

Then, because of the use of ECN, we would have Equation (12) and (13) for WFCC.

$$\bar{q} \approx K \tag{12}$$

$$\bar{R} = d + \frac{\int_t^{t+T} q(t)dt}{T}\frac{1}{C} \approx d + \frac{K}{C} \tag{13}$$

Moreover, recall that they are in their stable states and the periodicity is T. So, there would be

$$\begin{aligned} 0 &= \int_t^{t+T} \frac{dW_i(t)}{dt}dt \\ &= \int_t^{t+T} \frac{1}{R(t)}\left(1 - \frac{W_i(t)}{2\theta_i}\alpha(t)p\left(t - \bar{R}\right)\right)dt \\ &\approx \int_t^{t+T} \frac{1}{R(t)}\left(1 - \frac{W_i(t)}{2\theta_i}\bar{\alpha}\bar{p}\right)dt \end{aligned} \tag{14}$$

implying that

$$\bar{W}_i \approx \frac{2\theta_i}{\bar{\alpha}\bar{p}} \tag{15}$$

since $\forall t : R(t) > 0, W_i(t) \geq 0$.

That is to say, once *push* flows are in their stable states, $\bar{W}_i$, the average window size of the i-th *push* flow, would be in proportion to its parameter θ_i. Moreover, as these *push* flows are synchronized, they would share the same RTT; so, their obtained throughputs are in proportion to their parameters as well. Thus, Theorem 1 is proved, which yields a guideline for the prioritized schedule of *push* flows:

Guideline 1. *Given a set of push flows, we can control their bandwidth allocations by configuring their θ_i values.*

Indeed, by applying this technique to (6) and (7), we get

$$\bar{p} = \bar{\alpha} \tag{16}$$

$$\sum_{i=1}^{N} \bar{W}_i \approx Cd + K \tag{17}$$

and further obtain

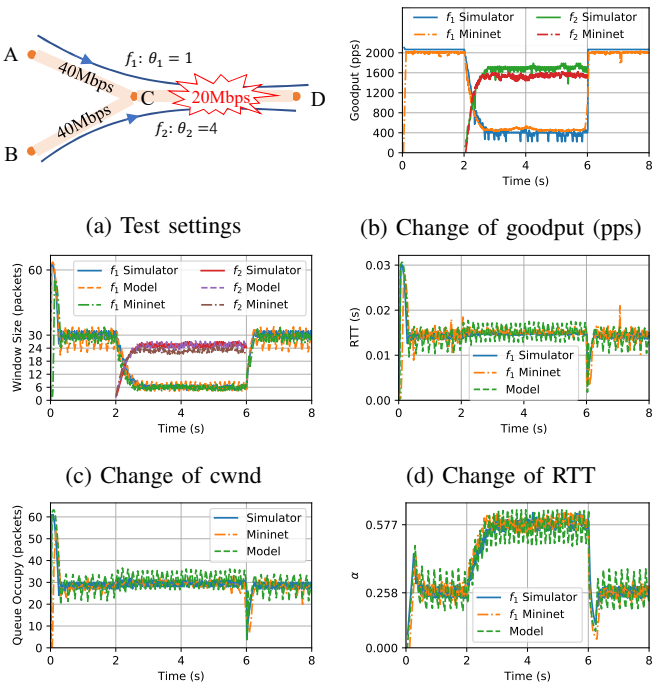
$$\bar{p} = \bar{\alpha} \approx \sqrt{\frac{2\sum_i^N \theta_i}{Cd + K}} \tag{18}$$

$$\bar{W}_i \approx \frac{\theta_i}{\sum_{i=1}^N \theta_i}(Cd + K) \tag{19}$$

by solving (15), (16), and (17) jointly.

According to the definition, α_i would never exceed 1; so, the value of θ_i should hold

$$\sum_{i=1}^{N} \theta_i \leq \frac{Cd + K}{2} \approx \frac{\bar{R}C}{2} \tag{20}$$



(a) Test settings (b) Change of goodput (pps)

(c) Change of cwnd (d) Change of RTT

(e) Change of queue occupancy (f) Change of α

Fig. 2: Flow behaviors observed in Mininet test, packet-level simulation, and fluid model analysis are consistent; when sharing the same bottleneck link, *push* flows converge to weighted bandwidth allocations respecting their θs very fast.

On the other hand, when the network is congested, the decrease of flow i's congestion window within in one RTT is about $1 - \frac{\alpha W_i(t)}{2\theta_i}$. To improve the stability of *push* flows and make efficient use of link capacities, we should let $W_i(t) + 1 - \frac{\alpha W_i(t)}{2\theta_i} > 0$, which indicates:

$$\theta_i > \frac{\alpha}{2} \frac{W_i(t)}{W_i(t)+1} \tag{21}$$

As $\alpha \leq 1$, we recommend $\theta_i \geq 0.5$ in practice, which is used as the value of θ^-. Regarding the value of θ^+, it can be approximated by $\frac{\bar{R}C}{2}$ according to (20), where $\bar{R}$ is the average round-trip time when the network is fully loaded.

B. Experimental Results

To verify that *push* flows do achieve the provable weighted fairness in practice, we prototype PPUSH in Mininet [30] and further develop a packet-level discrete-time simulator to simulate its precise behavior. We run micro-tests and compare the observed behaviors of *push* flows with those suggested by the numerical analysis of fluid model {(5),(6),(7)}. Results confirm that *push* flows converge to the provable weighted-fair yet work-conserving bandwidth allocations efficiently.

Setup. Consider that there are two *push* flows, named f_1 and f_2 belonging to two diverse tasks, going through the same bottleneck link $L_{C,D}$ as Figure 2a sketches. In tests, *push* flow f_1 starts at 0s then completes at 8s, while *push* flow f_2 appears at time 2s then terminates at time 6s. Their θ parameters, i.e., θ_1 and θ_2, are 1 and 4, respectively. We assume that the propagation delay is very small and the bottleneck link has the capacity of 20Mbps, which could transmit about 2066 *push* packets per seconds (pps), since each *push* packet is with the size of 1210 bytes. During their transmissions, f_1 and f_2 would build up the queue allocated to *push* traffic at the bottleneck link $L_{C,D}$; and when going through the link, *push* packets would get ECN markings on enqueue once the queue occupancy reaches the threshold of 30 packets. Following [22], the value of parameter g is set to $1/16$.

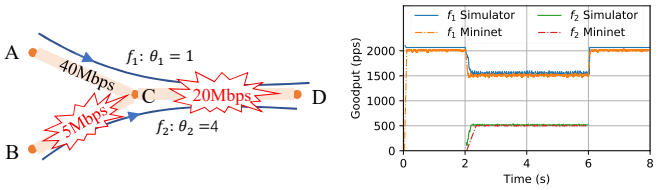
In our Mininet-based tests, the proof-of-concept implementation of *push* protocol is implemented in Python3.7. Even though *push* uses UDP as its basis, to be aware of ECN signals, both the sender and receiver should maintain the ECN field residing in IP headers directly. We implement this with *raw sockets*. Moreover, to improve the sending performance of *push* sender, we launch two loop-forever processes for the sending of data packets and receiving of ACKs, respectively. These two processes communicate with shared memory. On getting an ACK, the receiving process would update the congestion window and *sent-yet-unacknowledged* queue, according to the algorithm described in Section III-B. Because of the performance limits of Mininet, we do not use very high link capacity here [30]. In short, Mininet is built upon light-weighted virtual techniques including process-based virtualization, network namespaces, and software-based switches. During the simulation, all virtual hosts, switches, and links share the host server's CPU, memory, and I/O capacities. To reduce the impact of resource competition and to highlight the results, link capacities must be limited.

The design of our simulator follows that of [31] and is written in Python3.7 as well. Basically, we implement a discrete-event driven core, based on which, it is easy to write Python objects to emulate the packet-level behaviors of point-to-point link, queue discipline algorithms, flow senders and receivers in detail. Injected with data delivery tasks, the simulator will emulate then report how the involved packets get sent, forwarded, queued, dropped or received in detail.

As for the numerical analysis of fluid model, it is performed iteratively at the time step of one RTT.[3] We assume that, the initial congestion window of each flow is with the size of 1 and there is no queued or ECN-marked packet, i.e., $W_i(0) = 1$ and $q(0) = \alpha(0) = 0$. Regarding d, g, C, K and θ_is, they are configured respecting the parameters used in Mininet-based tests and packet-level simulations. Then, following {(3), (4), (13)} and {(5),(6),(7)}, It is easy to compute how exactly the congestion window $W_i(t)$, marked factor $\alpha(t)$, and queue occupancy $q(t)$ would change, iteratively. To stand in line with the protocol's design, during the numerical iteration, the maximum increase of w within in one RTT is bounded by $2BDP$, and the values of $W_i(t), q(t), \alpha(t)$ are amended by $W_i(t) \leftarrow \max(W_i(t), 1)$, $q(t) \leftarrow \min(q_{max}, \max(q(t), 0))$, and $\alpha(t) \leftarrow \max(\alpha(t), 0))$, respectively. As well, the code of our numerical analysis is written in Python3.7.

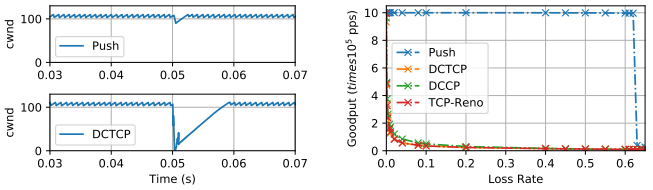
All tests and simulations are conducted on a 64-bit Ubuntu 18.04 server equipped with 16G RAM and a single Intel(R)

[3] Here, we simply use the approximated average RTT of Equation (13).



(a) Test settings (b) Change of goodput (pps)

Fig. 3: When *push* flows go through different bottlenecks, their bandwidth allocations are work-conserving.



(a) Impact of burst loss (b) Impact of random loss

Fig. 4: PPUSH achieves high goodputs upon packet loss.

Xeon(R) Silver 4210 (2.20GHz) CPU.

PPUSH converges to provable weighted fairness. As Figure 2c shows, *push* flows could make full use of the bottleneck link bandwidth; and consistent with the result of our fluid model based analysis, their stable sending rates (i.e., the observed goodputs since no packet drop occurs) are in proportion to their θs. Besides their goodputs, we also record their detailed behaviors. Recall that f_1 and f_2 go through the same bottleneck link; thus, their observed round-trip times (RTTs), fraction of marked packets (α) would be the same. For simplicity, we only plot f_1's RTTs and αs here. As Figure 2c demonstrates, similar to the behaviors of DCTCP [22], the congestion window of *push* flow converges to its steady "sawtooth" very fast; so do its RTT (Figure 2d), α (Figure 2f), and the bottleneck link's observed queue occupancy (Figure 2e). From Figure 2e, we also observe that, in their stable states, $\bar{q}$, the mean of queue occupancy, does approximate the threshold of ECN marking, i.e., 30. Thus, our conjecture of $\bar{q} = K$ specified in (12) is reasonable. Similarly, consistent with (17) and (19), the sum of active *push* flows' congestion window sizes shown in Figure 2c is approximately equal to the stable queue occupancy ($\approx$ 30). Moreover, according to (18), when only f_1 is active, the value of $\bar{\alpha}$ would approximate $\sqrt{2 \times 1/30} \approx 0.258$; and when both *push* flows are active, the value would approximate $\sqrt{2 \times (1+4)/30} \approx 0.577$. Figure 2f shows consistent results.

PPUSH is work-conserving. In practice, *push* flows might have multiple bottlenecks; strictly allocating bandwidth in proportion to their weights is not work-conserving. To verify how *push* flow would behave, we reduce the capacity of link $L_{B,C}$ to 5Mbps and rerun tests as Figure 3a shows. The goodputs of *push* flows observed in Figure 3b demonstrate that the bandwidth allocations made by *push* flows are work-conserving. Thus, PPUSH achieves optimal link utilization.

V. Performance Evaluation

In this section, we make detailed evaluation of PPUSH. Extensive packet-level simulation results indicate that:

1) PPUSH is able to tolerate packet loss;
2) it performs θ-based prioritized schedule gracefully without introducing bursts;
3) it largely eliminates possible incasts;
4) by dynamically splitting task among its source nodes and prioritizing traffic respecting their remaining sizes, it delivers multi-source data objects very efficiently, outperforming existing solutions in both intra- and inter-datacenter scenarios.

Compared schemes. As verified in §IV-B, the observed behavior of our Python-based simulator is consistent with those implied by Mininet-based prototype implementation and math analysis. Thus, in the following, we use it to study the detailed performance of PPUSH. Basically, we implement the proposed *push* and several other protocols including TCP-Reno, DCTCP, and TCP-like congestion control driven DCCP (referred as DCCP for short hereafter), and SCDP as baselines.

A. Micro-benchmark Analysis

Before looking into the performance gains PPUSH could achieve on optimizing tasks' average completion times (§V-B), here, we first study its detailed behaviors on packet loss, bandwidth dynamic, θ switching, and incast impairment.

PPUSH tolerates packet loss. Different from DCTCP, *push* is designed to achieve high throughput over lossy link. To study the impacts of packet loss on *push* flows, we let a *push* flow whose θ is 1 go through a link with the capacity of 10^6pps and latency of 40us. A drop-tail queue is bounded to the link; it would mark ECT-enabled packets on enqueue once its occupancy is larger than 30 packets and would further drop packets once the occupancy reaches 100 packets. To simulate serious packet loss caused by burst congestion, we configure the link to drop 100 successive packets starting from time 0.05s. As Figure 4a shows, the bursty packet loss has little impact on the size of *push*'s congestion window, while the congestion window of DCTCP collapses, indicating the robust of *push*. However, the robustness has bounds. In this test, the congestion window of *push* flow would collapse once the number of successive drops reaches 108. Furthermore, we also test the impacts of random packet loss. As demonstrated in Figure 4b, the goodputs of TCP-Reno, DCTCP, and the semi-reliable TCP-like DCCP decrease rapidly; although DCCP is slightly better than the other two, their achieved goodputs are less than 5% of the available link capacity once the loss rate is larger than 0.1. In consistent, *push* flow achieves consistent goodputs over lossy links when the random loss rate is less than 62%. This is reasonable since *push* flow does not suffer from the problem of head-of-line blocking and it could infer lost packets for transmission efficiently.

PPUSH reacts to traffic variations very quickly. In practice, *push* flows are likely to coexist with other traffic and *push*

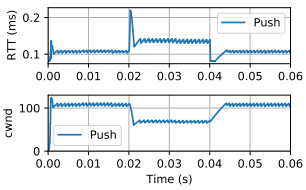


Fig. 5: *Push* flow adapts its cwnd respecting the available link capacity very quickly.

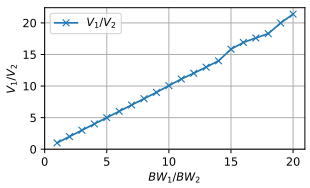
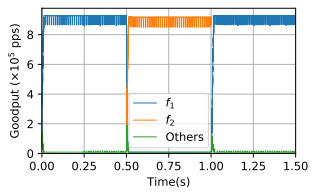
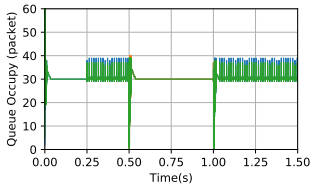


Fig. 6: PPUSH makes full use of all available bandwidth to achieve efficient data delivery.



(a) Impacts on goodput



(b) Impacts on queue occupancy

Fig. 7: PPUSH performs graceful schedule of θ without introducing burst congestion.

flows are recommended to set with low priorities. As a result, the bandwidth a *push* flow can use would vary with time, depending on the rate of high-priority traffic. In such a situation, the observed RTT of *push* flow would vary as well since its available bandwidth changes. To investigate the performance of *push* flow, we rerun the test shown in Figure 4a and let the available bandwidth get halved then doubled at time 0.2s and 0.4s, respectively. As Figure 5 shows, consistent with Equation (4), the observed RTT does change with the link bandwidth that *push* flow could use. Obviously, *push* is very efficient—it is able to adapt the congestion window size (cwnd) to match the available bandwidth in several RTTs.

PPUSH makes full use of available bandwidth. PPUSH achieves efficient data delivery by using all available bandwidth for parallel push. To check its flexibility, we setup a delivery task with the size of 0.5×10^6 packets served by two source nodes traveling diverse paths to the receiver. We let the sum of their total bandwidth be 10^6 consistently, and vary V_1/V_2, their ratio of available bandwidth, from 1 to 20. We find that, PPUSH is able to make full use of all available bandwidth as the task always completes at nearly 0.5s. Also, we find that the ratio of *push* flows' completed volumes grows linearly with the ratio of available bandwidth on their paths as Figure 6 demonstrates.

The schedule of θ is graceful. As described in §III-C, once a new task arrives or any existing task completes, PPUSH would reassign the large θ value among tasks to perform smallest-remaining-size-first scheduling. To study its impact on the network load, we let 10 *push* flows go through the same link with the capacity of 10×10^5 pps, among which f_2 starts at 0.5s then completes at 1s. Following the setting, θ^+ is with the value of 55. According to Algorithm 1, during time slots $0 \sim 0.5$s and $1 \sim 1.5$s, f_1 would obtain the large-θ value of $55 - 8 \times 0.5 = 51$ and all other active flows use the small-θ value of 0.5. Then, during $0.5 \sim 1$s, f_2 would take over the large-θ of $55 - 9 \times 0.5 = 50.5$ until it completes at time 1s. As Figure 7 shows, PPUSH performs the schedule of θ gracefully; the switch of θ would not introduce bursts. This is reasonable since the impact of θ on flow's sending rate is indirect: changing a flow's θ would not immediately impact its sending rate; instead, it controls how the *push* flow reacts to the congestion signal of ECN markings. Then, all the active flows would iteratively converge to the weighted fair bandwidth allocations in RTTs. We also observe that *i*) the link capacity is fully used; and *ii*) the ratios of the average observed goodputs of f_1 and f_2 over other flows during $0 \sim 0.5$s and $0.5 \sim 1$s, are about 102 and 101, respectively, equal to the ratios of their θ values exactly.

PPUSH largely eliminates incast. Recall that PPUSH accelerates the delivery of data objects with parallel *push* flows. As these flows are in the pattern of many-to-one aggregation, once getting synchronized, they might suffer from the incast problem: in that situation, packets may exhaust the maximum permitted queue buffer, resulting in packet losses. To study the impacts, we increase the amount of concurrent long-lived *push* flows through a same link, then observe their stable queue occupancy. Results show that, when θ-parameters are set according to Algorithm 1, *push* flows would exhaust the queue buffer of 100 packets once their amount reaches 179. By contrast, when all *push* flows share the same θ parameter of 1 like DCTCP, 90 *push* flows, about only half of that of PPUSH, would lead to full queue occupancy; as for DCTCP [22], only 61 concurrent flows would cause queue overflow. Such a result indicates that PPUSH's congestion control algorithm also largely eliminates incast problems. It is reasonable since most *push* flows in PPUSH are more sensitive to ECN markings as their θ-parameter is set to 0.5. Thus, PPUSH is able to leverage a large number of sources for data delivery at the same time.

B. Performance

Now, we evaluate the performance of PPUSH on optimizing the average task completion times via trace-based simulations.

As for the tested workloads, consistent with real distributed application designs, we consider that a fixed set of M nodes in the cluster work as the storage nodes holding all the data objects. On getting a delivery task, the PPUSH controller randomly selects R nodes to *push* the requested data in parallel. Here, we define R as the task's *fanout* and its typical value is 3. Regarding data object sizes, they are synthesized following the distribution of transfer sizes measured from a real data center [12]. In consideration of that PPUSH is not designed for the delivery of very tiny data objects, we scale the object sizes to the range of $[2, 200] \times 1024$ packets accordingly. In line with prior study [12, 13, 32], tasks are assumed to arrive in Poisson; the arrival rate is varied to obtain a desired level of network load on bottleneck links and the default average load is 0.9. Regarding the cluster's network architecture, we test the topology of One Big Switch abstraction (OBS) [12, 32], Leaf Spine [33], and Fattree [34]. We vary the simulation settings to study the robustness of PPUSH and observed consistent conclusions. By default, the tested clusters involve 100 servers

TABLE II: Compared schemes & baselines

# Scheme	Task assignment policy	Traffic prioritization	Switch modification
PPUSH(0,0)	static	no	no
PPUSH(0,1)	static	WFCC	no
PPUSH(1,0)	dynamic	no	no
PPUSH(1,1)	dynamic	WFCC	no
DCTCP [22]	static	no	no
DCCP [35]	static	no	no
TCP-Reno	static	no	no
SCDP [7]	dynamic	MLFQ	yes

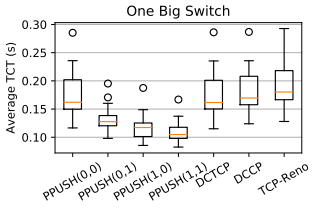
for all topologies but Fattree—due to the specific structure of Fattree, we set its k-parameter to 8, yielding a scale of 128 hosts. As for link capacities, their default values are 2×10^5 pps; while that of the fabric link of Leaf Spine topology is 8×10^5 pps. For OBS and Leaf Spine topologies, 10 hosts are randomly selected as storage nodes; and for Fattree, each of its *pod* substructure would contain exactly one storage node.

As Table II summaries, besides the default PPUSH(1,1) scheme, in which the suffix of $(1, 1)$ indicates that both the dynamic task assignment and traffic prioritization are enabled, we also consider the variants of PPUSH(0,0), PPUSH(0,1), and PPUSH(1,0). For PPUSH(0,0) and PPUSH(1,0), their θ-parameters are set to 1 and keep consistent to disable traffic prioritization. As for the baselines, we assume that each task is equally split across all its source nodes and employs the state-of-the-art DCTCP [22], TCP-like DCCP [35], and the legacy TCP-Reno for the transmission of each subpart, respectively. In tests, we generate 200 tasks and rerun 20 instances for each parameter setting. We use the raw value of average task completion times (Average TCT or ATCT for short) as the performance metrics. And to eliminate the impact of task sizes, we also consider the normalized ATCT. Assume the average task size of the tested workload is s and the bandwidth of each end-host is c, then the ideal average task completion time used for normalization is $\frac{s}{c}$. When comparing multiple schemes, the factor of improvement made by scheme A over B is computed by $\frac{ATCT_B}{ATCT_A}$, in which $ATCT_A$ and $ATCT_B$ are the average TCTs under the schedule of A and B, respectively.

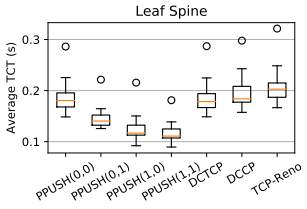
Case studies. Figure 8 shows the detailed average TCTs of different schemes over various network topologies. Recall that for each parameter settings, we rerun 20 times. Accordingly, the box of each scheme shows the lower to upper quartile values of the twenty ATCTs and the red line indicates the corresponding mean value. It is obvious that, PPUSH(1,1), i.e., PPUSH with both dynamic task assignment and θ-parameter based traffic prioritization enabled, achieves the best ATCTs. For example, its performance gains in terms of mean normalized ATCTs over PPUSH(0,0) in topologies OBS, Leaf Spine, and Fattree are about 1.586×, 1.581×, and 1.453×, respectively. We also notice that, among all topologies, the result of PPUSH(0,0) is quite close to that of DCTCP. This is reasonable since the network is not heavily loaded—in such a case, there is rarely packet drops; thus, the behavior of *push* flow is quite similar to that of DCTCP since their congestion control algorithms are exactly the same (in PPUSH(0,0), all θs are set to 1). The performances of TCP-like DCCP and TCP-Reno are much worse, since they are not designed for data center networks. Another notable finding is that, PPUSH(1,0) outperforms PPUSH(0,1), indicating dynamic task assignment achieves more performance improvement than θ-based traffic prioritization on the tested workloads. Consider the results of OBS in Figure 8a as examples, the performance gains of PPUSH(1,0) and PPUSH(0,1) over PPUSH(0,0) are about 1.495× and 1.329×, respectively. Figure 9 shows the distribution of the corresponding *push* flows' completion ratios. Results confirms that the PPUSH does dynamically adapt the task's interested data to its available source nodes. For instance, if tasks are split equally, each *push* flow would complete about 33.3% of its task since the source fanout is 3; however, under the schedule of PPUSH(1,0) about 20% of *push* flows complete about 40% of their tasks, and more than 3% of *push* flows completes 60% of their tasks. The results obtained by PPUSH(1,1) are much more obvious, in which about 8% of *push* flows completes 60% of their tasks. Such results indicate that PPUSH could make full use of all the bandwidth for data delivery, and moreover, the θ-based congestion control enables PPUSH to further prioritize tasks respecting their remaining sizes, resulting in more unbalanced completion ratios.

Impact of source fanout. By default, the task's source fanout is 3, which means each task involves 3 source nodes. In this part, we vary the source fanout from 1 to 9 and rerun the tests. To eliminate the diversity of task size, we mainly use the normalized average task completion time as the metrics hereafter; for each parameter setting, the plotted result is the average value of the twenty tries. Roughly speaking, as the results in Figure 10 show, the normalized ATCTs decrease with the increase of source fanouts. However, the achieved improvement is diminishing: 2 and 3 sources yield significant gains and once fanout > 5, there is little improvements then. Thus, to reduce the controller loads while enjoying the benefits of multiple source nodes, we recommend the use of 3 source nodes as default. The results also demonstrate that PPUSH outperforms DCTCP even when each task only involves one source node. In such cases, the benefits come from θ-based traffic prioritization; thus, PPUSH is able to optimize the completions of legacy unicast transfers as well.

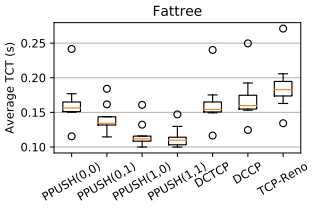
Impact of network load. To study the impacts of network load, we control the task arrival rate to vary the average network load from 0.5 to 0.99. As Figure 11a shows, for all schemes, the normalized ATCTs grow linearly with the increase of network load; and the performance gain of PPUSH(1,1) over PPUSH(0,0) would exceed 1.6×. This is reasonable, as higher network loads would cause more concurrent transfers and link congestion, resulting in larger completion times. However, the rates of increase under the schedule of PPUSH(1,1), PPUSH(1,0), and PPUSH(0,1) are less than those of DCTCP, DCCP, and TCP-Reno, indicating that the prioritized design enables PPUSH to make more efficient use of link capacities. As for PPUSH(0,0), consistent with prior findings, its results overlap with those of DCTCP. Hereafter, we only present the results obtained in the OBS topology,



(a) One Big Switch



(b) Leaf-Spine (100,25,1)



(c) Fattree(8)

Fig. 8: The detailed average task completion times (ATCTs) of PPUSH and baselines on various network architectures.

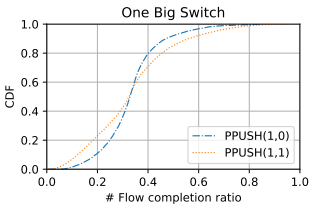
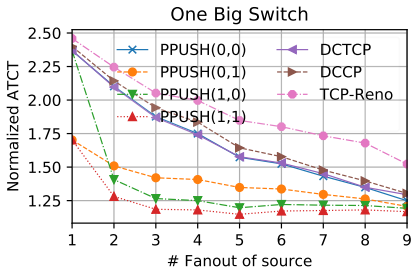
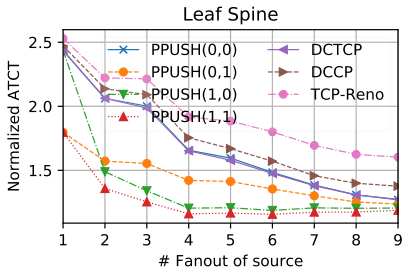


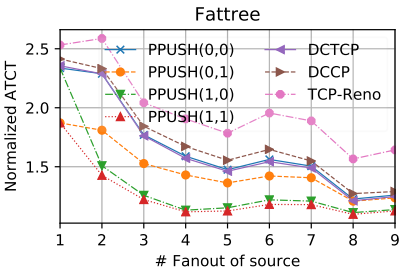
Fig. 9: Distributions of *push* flow completion ratios for test instances shown in Figure 8a.



(a) One Big Switch



(b) Leaf-Spine (100,25,1)



(c) Fattree(8)

Fig. 10: Impacts of source fanout indicates that selecting 3 source nodes for each task is good enough for PPUSH; note that the results of PPUSH(0,0) overlap with those of DCTCP.

since we observe the consistent results among all topologies like the situation shown in Figure 10.

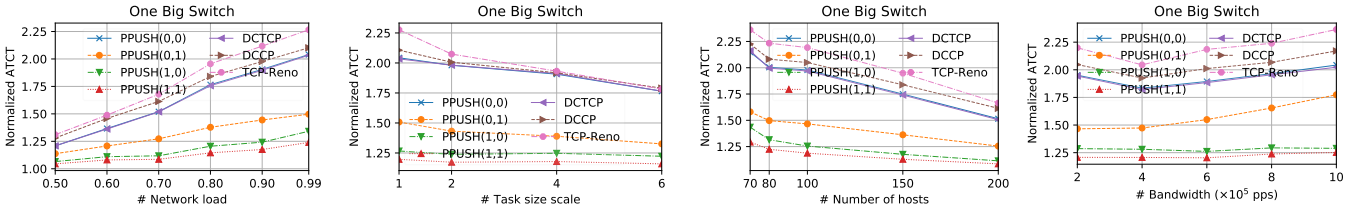
Impact of task size. Next, we re-scale each task size 2, 4, or 6 times while holding the average network load of 0.9 to study the impacts of task sizes. As Figure 11b explains, the benefits of PPUSH(1,1) and PPUSH(1,0) scheduling would obtain consistent normalized average completion time, while other schemes that split the task to source nodes equally obtain slight performance improvements. This is mainly because of the simulation setting. With the increase of task size, there would be fewer concurrent tasks in the cluster but their sizes are much larger as we let the network load keep 0.9. Accordingly, for these large-size flows, there would be enough time for their congestion control algorithms to converge, thus making fully use of link capacities.

Impact of cluster scale. Figure 11c shows the achieved normalized ATCTs would decrease with the increase of their cluster scale. Thus, PPUSH is able to provide high-performance multi-source data delivery for very large edge data centers as well. We also find that the performance improvement of PPUSH(1,1) is slightly less than that of DCTCP, DCCP, TCP-Reno; this is because its performance is quite close to the unachievable optimal of 1 already, yielding little space for improvement.

Impact of link capacity. In consideration of that clusters in practice might have various link capacities, we also test the impact by increasing the capacity of each link. As Figure 11d shows, distinguished from other schemes, PPUSH(1,1) achieves consistent normalized ATCT upon networks with larger link capacities, indicating that PPUSH(1,1) could achieve efficient data delivery on future 40G, 100G, or even 400G cluster networks as well.

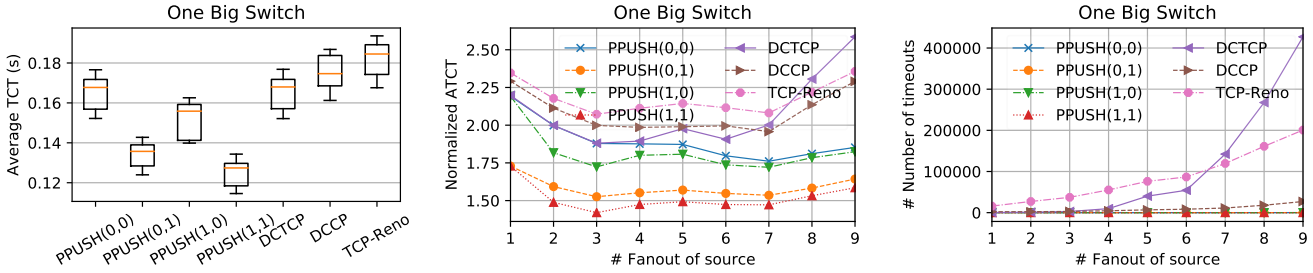
Impact of workload type. Then, we investigate the impact of workload type on the schedule performance. Here, we let each task randomly select 3 other nodes rather than from a pre-decided storage node pool as its sources, and generate 700 tasks to test. As the results in Figure 12 shows, upon this new type of workload, PPUSH(1,1) also achieves the best performance without surprise. However, a different and interesting observation is that, PPUSH(0,1) outperforms PPUSH(1,0), indicating the benefits of θ-based traffic prioritization is larger than that of dynamically task assignments. This is mainly because in this workload, there would be much more concurrent flows in the network; accordingly, the θ-based traffic prioritization is more efficient than dynamic task assignment. We also notice that, once the fanout is larger than 3, their performance would even decrease. This is mainly because, when a task arrives, it would trigger multiple concurrent flows at the same time, introducing non-trivial bursts. When the number of concurrent flows reach some threshold, these bursts would build up the queue, then lead packets drops, or even worse cause flow timeouts, resulting in legible performance degradation. We have collected the average amounts of timeout events suffered by tested schemes. As Figure 12c shows, once the source fanout is larger than 3, the amount of timeout events does increase. Even so, the performance impact on PPUSH is little. To avoid this problem, we recommend 3 source nodes for each task by default.

Compared with SCDP. Next, we compare the performance of PPUSH with the recently proposed SCDP [7]. At the high



(a) Impact of network load (b) Impact of task size (c) Impact of cluster scale (d) Impact of link capacity

Fig. 11: Extensive tests on various network loads, task sizes, cluster scales, and link capacities show that PPUSH(1,1) always obtains the best near-optimal performances; note that the results of PPUSH(0,0) overlap with those of DCTCP.



(a) ATCT details when fanout=3 (b) Normalized ATCTs vary with fanouts (c) Timeout events vary with fanouts

Fig. 12: Results indicate consistent conclusions when each task's source nodes are randomly selected from all active nodes.

level, both SCDP and PPUSH employ RaptorQ-decoupled parallel transfers to achieve efficient delivery of multi-source data objects. However, at the low level, they are totally different. Firstly, SCDP is built upon NDP, a recent protocol that achieves very low latency and high-throughput delivery of small-size flows in the context of data center networks, with the cost of non-trivial modification on switch hardware [18]. Secondly, SCDP employs Multi-Level Feedback Queuing (MLFQ) for traffic prioritization, which requires multiple queues and is sub-optimal as it does not use the exact remaining task size information available at receivers. To investigate the performance of SCDP over various task sizes, we scale the object sizes to the range of $[20, 2000] \times L$, and increase the scale factor L from 1 to 300. By default, the initial windows (IW) of SCDP is set to one BDP. Figure 13 shows the average completion times of data delivery tasks under the schedules of PPUSH and SCDP, in which the thresholds of MLFQ are $[20, 204, 2048, 20480]$ (packets), respectively. For all tests, the normalized average task completion times get better slightly with the increase of L. This is mainly because small tasks generally complete within a few RTTs; the impacts of RTT on their completions are more obvious. Basically, SCDP achieves excellent performance on delivering small data objects. However, with the increase of task size, its performance decreases; and it would underperform both PPUSH(1,0) and PPUSH(1,1) once L is larger than 10. In addition, we also rerun the tests by letting MLFQ thresholds increase with L and observe very similar phenomena. Such results imply that the performance of SCDP is very sensitive to its parameter settings: in case the thresholds of MLFQ do not match with the workload well, its performance will get degraded a lot. Indeed, this mismatching is inevasible in practice as the network loads are mutable and MLFQ thresholds

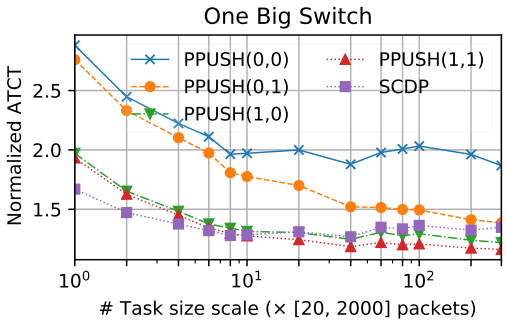
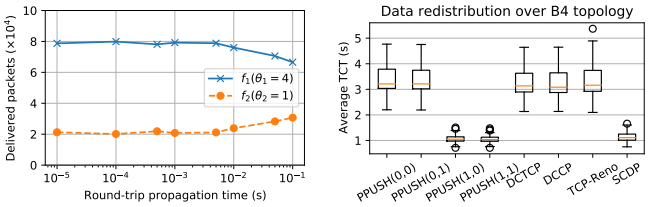


Fig. 13: The performance gap between PPUSH and SCDP is tiny; PPUSH is better unless data objects are very small.

are hard to tune. In consistent, PPUSH is able to adopt its congestion windows respecting packet drops, ECN marks, and θ parameters gracefully. The performance of PPUSH on the delivery of small objects are not as good as its performance on the delivery of large objects. This is mainly because it takes several RTTs for a *push* flow to converge and make full use of all the link capacity. A promising enhancement is to increase its initial congestion window size smartly; we leave this as future work.

PPUSH over WAN. In practice, some data objects might get replicated among geo-distributed data centers connected with wide-area network (WAN). Compared with intra-datacenter connections, WAN-based inter-datacenter paths generally have limited capacities (e.g., hundreds of Mbps) and large round-trip times (e.g., several to hundreds of milliseconds) [36]. To better understand the impact of WAN, we first consider a simple flow-level test in which two *push* flows f_1 and f_2, with



(a) Impact of propagation time (b) Data redistribution over WAN

Fig. 14: PPUSH achieves consistent performance over WAN.

the θ values of 4 and 1, go through the same WAN path/tunnel with the capacity of 10^4 pps. We vary the path's round-trip propagation time from 10us to 100ms, then observe how many packets would be delivered by f_1 and f_2 in 10 seconds. As Figure 14a shows, f_1 and f_2 can make full use of the available bandwidth, as the sum of their delivered packets approximates $10^5 = 10^4(pps) \times 10(s)$. However, with the increase of round-trip propagation time, the ratio of their allocated bandwidths would not strictly match with $\frac{4}{1}$, the ratio of their θ values, any more. This is because it takes several round-trip propagation times for *push* flows to converge. This duration increases with round-trip propagation time and does impact flows' delivered volumes.

To further investigate the performance of PPUSH, we also consider a task-level multi-source data delivery workload named *data redistribution*. Here, we consider that there are N data objects in a geo-distributed cloud, each of which is replicated among K randomly selected data centers. Then, one data center is temporarily unavailable; we i) randomly select a new data center for each impacted data object, and ii) recover it at the newly selected location with PPUSH and other protocols. Here, we use the Google's inter-datacenter WAN, B4, as the test topology [37] and scale both the number of data objects and the sizes of data objects down to accelerate the packet-level simulation. More specifically, the B4 topology involves 18 nodes and 42 links across 4 continents. We assume that each link is with the capacity of 10^4 pps, while the delays of intra- and inter- continent links are 1ms and 10ms, respectively. The size of data object is scaled down to the range of $[1, 100] \times 10^4$ packets. We consider that there are 300 data objects in total and each is replicated among 5 randomly selected nodes. The unavailable datacenter is chosen randomly and we repeat the test 50 times. Figure 14b shows the average completion times of all involved data objects under various delivery schemes. Consistent with the intra-datacenter multi-source data delivery tests shown in Figure 13. PPUSH achieves the best performances on inter-datacenter scenario as well. The small performance gaps among PPUSH(1,0), PPUSH(1,1), and SCDP imply that the performance improvements are mainly contributed by the RaptorQ-decoupled adaptive task allocation in such heterogeneous networks. In consideration of the fact that limited bandwidth and large round-trip propagation times would not impact the RaptorQ-decoupled dynamically task allocation. We argue that PPUSH would work well over WAN.

VI. Related Work

PPUSH achieves efficient deliveries of multi-source data with code-enhanced transport protocol, which implements prioritized bandwidth allocations without using priority queue. In the following, we revisit recent study on the related topics in the context of data center networks, respectively.

Multi-source data delivery. Taking advantage of the multi-source nature of data to accelerate its delivery is not new. Indeed, this is exactly the idea adopted by off-the-shelf Internet peer-to-peer (P2P) file sharing networks (e.g., BitTorrent) [10, 38], and commercial content delivery systems [39]. With the raise of cloud computing, a lot of effort has been made to deploy similar systems (e.g., a modified version of BitTorrent) in enterprise data centers like those of Twitter and Tencent, to reduce the time of package dissemination in service deployment [8, 40], or data broadcast in iterative data optimization [9]. Even though these proposals support many-to-one cooperative delivery of the same data object, the schedule results are sub-optimal as they are originally designed for one-to-many data dissemination, in which the original source sender is the scaling bottleneck. On one hand, they employ sophisticated peer protocols to select senders for each piece of the data object; such a control plane is slow and introduces non-trivial overheads [8]. On the other, their data deliveries are based on TCP or uTP. These protocols are proven to be sub-optimal for the optimization of task completion times, since they pursue max-min fairness on bandwidth allocation by design [12, 13]. Distinguished from them, PPUSH achieves efficient delivery of multi-source data object by i) decoupling the transmission of different source with RaptorQ codes and ii) developing the new transport protocol of *push* to prioritize concurrent tasks. Such a design shares the similar basic idea with the recent proposed SCDP, which also employs RaptorQ to achieve many-to-one delivery of multi-source data [7]. However, distinguished from SCDP's clean-slate, new hardware-based protocol designs [7], PPUSH requires no switch modifications thus is readily-deployable.

Code-enhanced datacenter transport protocol. In recent years, increasing attention is paid to enhancing the performance of data center transport protocols with advanced code techniques. For example, CAPS [41] employs LDPC codes, a linear error correcting code, to encode short TCP flows, based on which it spreads the coded packets among multiple available paths. Accordingly, CAPS is able to reduce the average flow completion times significantly as the problems of head-of-line blocking and out-of-order delivery are relieved with code. Likewise, DC^2-MTCP improves MPTCP by leveraging network coding to ease asynchronous packet losses [42]; LTTP [43] avoids the problem of incast for many-to-one UDP transfers with LT code; and sharing the similar idea of PPUSH, the recent proposed SCDP enables both many-to-one and one-to-many data transfers to tolerate packet loss and to achieve low flow completion times with the code of RaptorQ [7].

Traffic prioritization without priority queue. To achieve weighted-fair bandwidth allocation among tenants in public cloud, Seawall transmits their flows with separate TCP-

like tunnels and controls each tunnel's aggregated sending rate with weighted additive increase, multiplicative decrease (AIMD) [44]. Differently, WFA prioritizes traffic using the proportional fairness of TCP connection: it splits each transmission task into a controlled amount of concurrent flows based on its weight [9]. Such a design is sub-optimal since it would launch a large number of small flows in batches, introducing bursts, incasts, and unstable inefficient bandwidth utilization because of slow-start. Quite similar to the design of *push*, D2TCP [45] and L2DCT [17] prioritize TCP flows for the optimization of missed deadlines and average completion times by dynamically updating how senders react to ECN markings. However, their heuristic algorithm designs lack strong theoretical analysis. To support strict prioritized bandwidth allocation, PDQ lets switches compute flows' sending rates respecting to the tagged priorities explicitly. However, it is hard to deploy as the required switch operation is not supported by off-the-shelf hardware [18]. Motivated by PDQ, PAM [46] simplifies the involved switch operations with hardware-supported approximate designs, and designs novel rate control algorithms to achieve priority-based rate schedules for data center multicasting [33], using the emerged dataplane-programmable switch [47]. Systems like Varys [48], Fastpass [15] and Flowtune [49] achieve traffic prioritization by explicitly controlling the sending rate, or the emitting of packet or flowlet, from a central controller; however, they are hard to scale because the central schedulers are too involved in the scheduling. Distinguished from all above solutions, PPUSH employs provable yet easy-to-deploy weighed-fair congestion control for traffic prioritization; it is easy to scale up as the controller only needs to update the θ-parameters of current flows on the event of task arrival and completion.

VII. Conclusion

In this post-cloud era, micro data centers would be widely deployed at the network edge to build up low-latency, high-throughput, and cost-efficient cloud service for emerged 5G and IoT applications. On the cluster side, the critical data objects involved by edge computing are generally replicated among carefully selected servers and racks for various purposes like load balancing and high availability; and the associated cluster applications generally need to deliver large-size data objects among servers on demand.

To achieve efficient delivery of these multi-source data objects in edge cloud, this paper proposes a novel generic solution named PPUSH. At the high-level, PPUSH launches multiple parallel transfers with the assistance of controller for efficient and collaborative delivery based on the multi-source nature of data objects. At the low-level, it decouples associated transfers by encoding the payload data with RaptorQ codes and develops a new loss-tolerable, readily-deployable, prioritized transport protocol named *push* for the delivery from each source. To study the detailed behavior of PPUSH, we *i*) analyze its congestion control algorithm with fluid model, *ii*) prototype PPUSH in Mininet, and *iii*) further develop a packet-level network simulator. Extensive results obtained from Mininet-based micro benchmarks and packet-level large scale simulations indicate that, PPUSH is robust to packet loss and achieves provable prioritized bandwidth allocations; moreover, it achieves very efficient data delivery by making full use of all available source nodes, in both intra- and inter-datacenter scenarios. For instance, in a tested intra-datacenter instance, compared with the straightforward design of splitting tasks among sources equally and letting concurrent flows share bandwidth fairly, the design of adaptive task allocation and prioritized bandwidth allocations acquire the improvements of 1.495× and 1.329×, respectively, yielding a total improvement of 1.586×, when enabled at the same time.

References

[1] F. Liu, G. Tang *et al.*, "A survey on edge computing systems and tools," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1537–1562, Aug 2019.

[2] N. Abbas, Y. Zhang *et al.*, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, Feb 2018.

[3] "Geo-replication in azure container registry," https://docs.microsoft.com/en-us/azure/container-registry/container-registry-geo-replication, 2019.

[4] M. Li, D. G. Andersen *et al.*, "Scaling distributed machine learning with the parameter server," in *OSDI*, Oct. 2014, pp. 583–598.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*, 2003, pp. 20–43.

[6] K. Shvachko, H. Kuang *et al.*, "The hadoop distributed file system," in *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[7] M. Alasmar, G. Parisis, and J. Crowcroft, "Scdp: Systematic rateless coding for efficient data transport in data centres," *CoRR*, vol. abs/1909.08928, 2019.

[8] S. James and P. Crowley, "Experimental analyses of data distribution on data center networks," in *IEEE P2P 2013 Proceedings*, Sep. 2013, pp. 1–10.

[9] M. Chowdhury, M. Zaharia *et al.*, "Managing data transfers in computer clusters with orchestra," in *SIGCOMM*, 2011, pp. 98–109.

[10] B. Cohen, "The bittorrent protocol specification," http://www.bittorrent.org/beps/bep_0003.html, Jan 2008.

[11] M. Alizadeh, A. Greenberg *et al.*, "Data center tcp (dctcp)," in *SIGCOMM*, 2010, pp. 63–74.

[12] M. Alizadeh, S. Yang *et al.*, "pfabric: Minimal near-optimal datacenter transport," in *SIGCOMM*, 2013, pp. 435–446.

[13] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *SIGCOMM*, 2012, pp. 127–138.

[14] A. Norberg, "utorrent transport protocol," http://www.bittorrent.org/beps/bep_0029.html, Jun 2009.

[15] J. Perry, A. Ousterhout *et al.*, "Fastpass: A centralized "zero-queue" datacenter network," in *SIGCOMM*, 2014, pp. 307–318.

[16] C. Wilson, H. Ballani *et al.*, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM*, 2011, pp. 50–61.

[17] A. Munir, I. A. Qazi *et al.*, "Minimizing flow completion times in data centers," in *IEEE INFOCOM*, April 2013, pp. 2157–2165.
[18] C. Raiciu and G. Antichi, "Ndp: Rethinking datacenter networks and stacks two years after," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 5, pp. 112–114, Nov. 2019.
[19] W. Bai, L. Chen *et al.*, "Information-agnostic flow scheduling for commodity data centers," in *NSDI*, 2015, pp. 455–468.
[20] B. Montazeri, Y. Li *et al.*, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *SIGCOMM*, 2018, pp. 221–235.
[21] Y. Lu, G. Chen *et al.*, "One more queue is enough: Minimizing flow completion time with explicit priority notification," in *IEEE INFOCOM*, May 2017, pp. 1–9.
[22] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of dctcp: Stability, convergence, and fairness," in *SIGMETRICS*, 2011, pp. 73–84.
[23] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *SIGCOMM*, 2011, pp. 350–361.
[24] J. Meza, T. Xu *et al.*, "A large scale study of data center network reliability," in *ACM IMC*, 2018, pp. 393–407.
[25] J. Meza, "Large scale studies of memory, storage, and network failures in a modern data center," *CoRR*, vol. abs/1901.03401, 2019.
[26] R. Joshi, T. Qu *et al.*, "Burstradar: Practical real-time microburst monitoring for datacenter networks," in *9th Asia-Pacific Workshop on Systems*, 2018, pp. 8:1–8:8.
[27] W. Bai, L. Chen *et al.*, "Enabling ecn in multi-service multi-queue data centers," in *NSDI*, 2016, pp. 537–549.
[28] L. Minder, A. Shokrollahi *et al.*, "RaptorQ Forward Error Correction Scheme for Object Delivery," RFC 6330, Aug. 2011.
[29] A. Langley, A. Riddoch *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *SIGCOMM*, 2017, pp. 183–196.
[30] N. Handigol, B. Heller *et al.*, "Reproducible network experiments using container-based emulation," in *8th CoNEXT*, 2012, pp. 253–264.
[31] N. Jay, N. Rotman *et al.*, "A deep reinforcement learning perspective on internet congestion control," in *36th International Conference on Machine Learning*, vol. 97, 09–15 Jun 2019, pp. 3050–3059.
[32] S. Luo, H. Yu *et al.*, "Towards practical and near-optimal coflow scheduling for data center networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3366–3380, Nov 2016.
[33] S. Luo, H. Xing, and K. Li, "Near-optimal multicast tree construction in leaf-spine data center networks," *IEEE Systems Journal*, pp. 1–4, 2019.
[34] S. Luo, H. Yu *et al.*, "Traffic-aware vdc embedding in data center: A case study of fattree," *China Communications*, vol. 11, no. 7, pp. 142–152, July 2014.
[35] S. Floyd and E. Kohler, "Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control," RFC 4341, Mar. 2006.
[36] F. Lai, M. Chowdhury, and H. Madhyastha, "To relay or not to relay for inter-cloud transfers?" in *HotCloud*, Jul. 2018.
[37] C.-Y. Hong, S. Mandal *et al.*, "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan," in *SIGCOMM*, 2018, pp. 74–87.
[38] Wikipedia contributors, "Peer-to-peer file sharing — Wikipedia, the free encyclopedia," 2020, [Online; accessed 07-Jan-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Peer-to-peer_file_sharing
[39] X. Chen, M. Zhao *et al.*, "The cask effect of multi-source content delivery: Measurement and mitigation," in *39th IEEE ICDCS*, July 2019, pp. 261–270.
[40] W. Kangjin, Y. Yong *et al.*, "Fid: A faster image distribution system for docker platform," in *IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Sep. 2017, pp. 191–198.
[41] J. Hu, J. Huang *et al.*, "Caps: Coding-based adaptive packet spraying to reduce flow completion time in data center," *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, pp. 2338–2353, Dec 2019.
[42] J. Sun, Y. Zhang *et al.*, "Dc2-mtcp: Light-weight coding for efficient multi-path transmission in data center network," in *IEEE International Parallel and Distributed Processing Symposium*, May 2017, pp. 419–428.
[43] C. Jiang, D. Li, and M. Xu, "Lttp: An lt-code based transport protocol for many-to-one communication in data centers," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 1, pp. 52–64, January 2014.
[44] A. Shieh, S. Kandula *et al.*, "Sharing the data center network," in *NSDI*, 2011, pp. 309–322.
[45] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *SIGCOMM*, 2012, pp. 115–126.
[46] S. Luo, H. Yu *et al.*, "Efficient file dissemination in data center networks with priority-based adaptive multicast," *IEEE Journal on Selected Areas in Communications*, 2020.
[47] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *Symposium on SDN Research (SOSR)*, 2017, pp. 115–121.
[48] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *SIGCOMM*, 2014, pp. 443–454.
[49] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet control for datacenter networks," in *NSDI*, Mar. 2017, pp. 421–435.