2018 IEEE International Conference on Cloud Engineering

Deadline-Aware Scheduling and Routing for Inter-Datacenter Multicast Transfers

Siqi Ji, Shuhao Liu, Baochun Li
University of Toronto

Abstract—**Many applications like geo-replication need to deliver multiple copies of data from a single datacenter to multiple datacenters, which has benefits of improving fault tolerance, increasing availability and achieving high service quality. These applications usually require completing multicast transfers before certain deadlines. Some of the existing works only consider unicast transfers, which is not appropriate for the multicast transmission type. An alternative approach proposed by existing works was to find a minimum weight Steiner tree for each transfer. Instead of using only one tree for each transfer, we propose to use one or multiple trees, which increases the flexibility of routing, improves the utilization of available bandwidth, and increases the throughput for each transfer. In this paper, we focus on the multicast transmission type, propose an efficient and effective solution that maximizes throughput for all transfer requests while meeting deadlines. We also show that our solution can reduce packet reordering by selecting very few Steiner trees for each transfer. We have implemented our solution on a software-defined overlay network at the application layer, and our real-world experiments on the Google Cloud Platform have shown that our system effectively improves the network throughput performance and has a lower traffic rejection rate compared to existing related works.**

I. Introduction

Cloud computing provides users and enterprises with a massive pool of resources to store and process their data. Since the volume of data grows exponentially, data migration and processing have become more crucial than ever before. In order to increase availability and reduce latency for end users, large cloud service providers have deployed tens (or even hundreds) of datacenters around the world in a geographically distributed fashion.

To improve fault tolerance, increase availability and achieve high service quality, many applications require efficient data transfers from one datacenter to multiple datacenters, typically for data replication, database synchronization, and data backup. For example, search engines need to synchronize databases regularly for the purpose of achieving a higher quality of user experience [1]. Blocks of a file in many distributed file systems like HDFS are replicated for fault tolerance.

Inter-datacenter transfers can roughly be classified into three categories based on their delay tolerance: *interactive transfers*, *elastic transfers* and *background transfers* [2]. *Interactive transfers*, like video streams and web requests, are highly sensitive to loss and delay, so they should be delivered instantly with strictly higher priority. *Elastic transfers* are delay tolerant but still require timely delivery (before a deadline). For example, many applications need to backup data within a certain time period. *Background transfers*, such as data warehousing, do not have explicit deadlines.

Why do we need to consider transfer deadlines? When multiple inter-datacenter transfers are sharing the same links in the inter-datacenter network, the total demand for these transfers typically far exceeds the available network capacity. On the one hand, some transfers like elastic transfers need to be completed timely, which can be modeled as deadlines. On the other hand, cloud providers set deadlines for most transfers based on their delay tolerance to different customer service level agreements (SLAs). A survey of WAN customers at Microsoft [3] shows that most transfers require deadlines and they would incur penalties if deadlines are missed. Customers are willing to pay more for guaranteed deadlines. Therefore, it is an important topic to meet as many transfer deadlines as possible.

In this paper, we focus on elastic transfers and background transfers that deliver data from one datacenters to multiple datacenters. We propose an efficient solution to maximize the network throughput and consider transfer deadlines at the same time. The multicast (one-to-many) transfer type is quite representative, other transmission types like unicast (one-to-one) and broadcast (one-to-all) can be transformed into it.

Traditional wisdom used *Steiner Tree Packing* [4], [5] to maximize the flow rate from a source to multiple destinations, which is an NP-complete problem. Another approach is to treat multicast transmission as multiple unicast transfers. Existing solutions, like B4 [6], SWAN [2] and BwE [7], aimed to maximize utilization and focused on max-min fairness. Tempus [8] designed a strategy to maximize the minimum fraction of transfers finished before deadlines. Amoeba [9] guaranteed deadlines, it introduced a deadline-based network abstraction for inter-datacenter transfers. DCRoute [1] scheduled each transfer with a single path to avoid packet reordering and it also guaranteed transfer deadlines for admitted requests.

Unfortunately, these solutions were not explicitly designed for multicast transfers, which can actually waste bandwidth by finding paths from the source to each destination, result in rejecting more transfer requests with deadline requirements. DCCast [10] and DDCCast [11] proposed to use minimum weight Steiner Trees and DDCCast used As Late As Possible (ALAP) policy for rate allocation. However, DCCast and DDCCast used one minimum weight Steiner tree for a request, which reduced the flexibility of choosing routing paths. Besides, if the bandwidth required by a request with a specific deadline is higher than the maximum available bandwidth in

0-7695-6371-6/18/31.00 ©2018 IEEE
DOI 10.1109/IC2E.2018.00035

IEEE computer society

the network, the request will be rejected by only choosing one tree. However, if we can split the traffic at the source and use multiple trees for delivering data, then the request can meet its deadline with higher throughput. Moreover, in the admission control part of DDCCast, a request can be rejected although it could have been admitted by choosing other forwarding trees. DDCCast and DDCast did not aim to achieve maximized throughput.

In this paper, we design a new routing and scheduling algorithm for multiple multicast data transfers across geo-distributed datacenters to maximize network throughput, with the consideration of transfer deadlines. We have implemented our solution in an application-layer software-defined inter-datacenter network. We also evaluate the performance of our solution with real-world experiments in the Google Cloud Platform. Our contributions are the following:

First, prior works on inter-datacenter traffic engineering [1], [2], [6]–[9] focused on unicast transfers, which are not effective for multicast transfers. We propose to use Steiner trees for each multicast transfer.

Second, prior work on multicast inter-datacenter transfers [10], [11] used one tree for each transfer, which could reject some transfer requests that have early deadlines. Our solution has higher flexibility for routing and uses at least one tree for each transfer. We formulate the problem as a Linear Program (LP), which can pack multiple multicast transfers with deadlines efficiently and achieve high throughput. Besides, to reduce packet reordering overhead at the destination, we add a penalty function in the objective of LP and use a log-based heuristic [12], [13] to find sparse solutions.

Third, prior work on multicast techniques [14]–[18] used software-defined networking (SDN) at the network layer. However, hardware switches in each datacenter can only support a limited number of forwarding entries. Besides, it is complicated and costly to solve the flow table scalability problem at large scales. We have implemented our solution in an application-layer software-defined network (SDN), which does not need to modify the underlying network properties and can scale up to a large number of transfer requests.

Finally, our real-world experimental results over the Google Cloud Platform have shown that our solution performs higher throughput and accommodates more transfer requests with deadlines as compared with the existing related works that consider deadlines.

The remainder of this paper is organized as follows. In Sec. II, we discuss our work in the context of related work. In Sec. III, we present the motivation of our design by using an example, talk about our design objectives and choices. In Sec. IV, we present our solution and formulate the routing problem of multiple multicast inter-datacenter transfers with the consideration of deadlines. In Sec. V, to show the practicability, we present our real-world implementation of our design. In Sec. VI, we evaluate its validity and performance in Google Cloud Platform, and we provide our discussion about future work in Sec. VII. We conclude the paper in Sec. VIII.

II. Related Works

A large amount of related works exist in the literature on datacenter traffic engineering or deadline-aware routing. Video streaming is one type of multicast transfers, which needs to deliver video content from a single source to other users in remote regions. This kind of transfer is highly delay-sensitive. Celerity [19] packed only depth-1 and depth-2 trees; Airlift [20] maximized throughput without violating end-to-end delay constraints by using network coding; and Liu *et al.* [21] proposed a delay-optimized routing scheme by only solving linear programs. However, these works were explicitly designed for delay-sensitive video streaming. We focus on elastic and background transfers which are delay-tolerant and some of them have deadlines.

Some existing works focused on improving performance for bulk transfers. Laoutaris *et al.* [22] proposed NetStitcher which minimized the completion time of bulk transfers by stitching unutilized bandwidth and employing a store-and-forward algorithm. They extended their work in [23] by considering the time-zone difference for delay-tolerant bulk transfers. Chen *et al.* [24] considered bulk transfers with deadlines in grid networks. Store-and-forward is also used in [25], [26] to complete transfers. Wang *et al.* [25] aimed to minimize network congestion of deadline-constrained bulk transfers. Wu *et al.* [26] concentrated on a per-chunk routing scheme. Storing data at the intermediate datacenters will increase the storage cost and transfer overhead. Owan [27] jointly optimized bulk transfers in optical and network layers. These works only considered unicast bulk transfers.

Google B4 [6] and Microsoft SWAN [2] used SDN among inter-datacenters for traffic engineering to maximize network throughput. BwE [7] provided work-conserving bandwidth allocation and focused on max-min fairness. These works did not consider transfer deadlines. Tempus [8] proposed an online scheduling scheme to maximize the minimum fraction of inter-datacenter transfers finished before deadlines. Ameoba [9] and DCRoute [1] guaranteed deadlines for admitted requests but they were not explicitly designed for multicast transfers.

DCCast [10] chose minimum weight forwarding trees for transfers and it focused on multicast transfers. DDCCast [11] was based on DCCast which took transfer deadlines into consideration. Our work differs from DDCCast because our solution chooses multiple trees (and at least one) for each transfer, which can accommodate more transfer requests than using exactly one tree and achieve higher throughput.

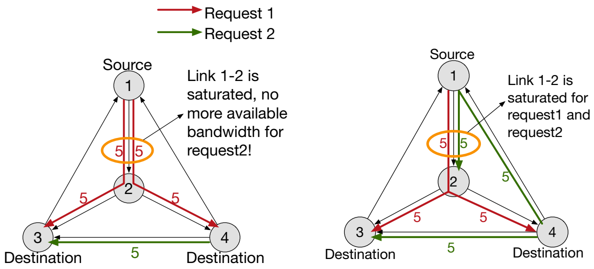
III. Background and Motivation

Definition of Meeting Deadlines. In this paper, we focus on multicast inter-datacenter transfers, which need to send multiple copies of data from a single source to multiple destinations. For each multicast transfer, we say that a transfer meets its deadline when all destinations receive the overall data before a particular time.

Scheduling of Data Transfers. Our solution aims to pack transfer requests that arrive in a small time interval optimally by taking full advantage of available inter-datacenter

Requests	Source	Destinations	Volume (MB)	Deadlines (seconds)
R_1	1	3, 4	200	40
R_2	4	2, 3	200	40

TABLE I: Request requirements.


Request 1
Request 2
Source
1
Link 1-2 is saturated, no more available bandwidth for request2!
5
5
2
5
5
3
4
Destination
5
Destination
Source
1
Link 1-2 is saturated for request1 and request2
5
5
2
5
5
5
3
4
5
Destination
Destination

(a) Using unicast transfers.

(b) Using Steiner trees.

Fig. 1: A motivation example: (a) Finding paths from the source to each destination, request R_2 will miss its deadline. (b) Using Steiner trees for transfers, both R_1 and R_2 can complete before deadlines.

capacities. If the available bandwidth can not accommodate all requests, then we reject requests with lower priorities and repack these requests when there are available capacities. We do not use the *As Late As Possible* policy because it may reduce the network capacity for future requests and achieve low throughput. We try to take full advantage of available bandwidth to pack requests at the current scheduling time slot.

Motivation Example. Considering the directed network shown in Figure 1, all link capacities are 10MB/s. There are two transfer requests R_1 and R_2. Table I shows detailed requirements for request R_1 and R_2. If we treat each multicast transfer as multiple unicast transfers, then we can find paths from the source to each of its destinations independently and assign rate for each path. Figure 1(a) illustrates this approach. However, link $1 \rightarrow 2$ becomes saturated for R_1, which results in no more bandwidth for R_2 to deliver data from 4 to 2. Therefore, R_2 will miss its deadline. Missing deadlines of requests will greatly degrade service quality and violate the application SLAs. Moreover, sometimes it will cause a great loss.

A better approach is to use Steiner trees for delivering source data to all destinations. As we can see in Figure 1(b), using trees to deliver data to destinations can save more bandwidth. Datacenter 1 sends one copy to datacenter 2; then datacenter 2 sends two copies to destinations. Request R_1 only takes 5MB/s of link $1 \rightarrow 2$, which leaves another 5MB/s for request R_2. Therefore, both R_1 and R_2 will meet their deadlines.

In this paper, we propose to use Steiner trees for multiple multicast transfers. Traditional wisdom applies *Steiner tree packing* but it is an NP-hard problem. We formulate the problem as a Linear Program (LP) and use a log-based heuristic to find sparse solutions.

Notation	Intrepretation
T^i	The set of feasible Steiner trees for request i.
S^i	Source datacenter of request i.
R^i	Destination datacenters of request i.
Q^i	Data volume in bytes of request i.
D^i	Deadline requirement of request i.
a^i	Priority of request i.
$G = (V, E, C)$	G denotes the inter-datacenter network graph, V and E are the set of vertices (datacenters) and edges (links) respectively. For each e $\in E$, $C(e)$ represents the available bandwidth capacity.

TABLE II: Mathematical notations used in this paper.

IV. Deadline-aware Routing and Scheduling

In an inter-datacenter network, given a number of transfer requests arriving within a small time interval, the key idea of our design is to determine the sending rate of each request on each Steiner tree by solving a routing problem. We aim to maximize the throughput for all requests, subject to deadline constraints. Moreover, we try to use few Steiner trees for each request in order to reduce the data splitting overhead at the source and packet reordering at destinations. Table II presents notations we used in this paper and their definitions.

A. Finding Feasible Steiner Trees

Network. We model the inter-datacenter network as a directed graph $G = (V, E, C)$. Link capacity is assumed to be stable in one time period. $C(e)$ denotes the available link capacity, which is the maximum packet sending rate on edge e $\in E$.

We use depth-first search (DFS) to find a set of feasible Steiner trees for each request. Nodes in trees that are pure relays are called *Steiner nodes*. A Steiner tree is a distribution tree that connects the sender with receivers, possibly through Steiner nodes. DFS starts at the source node, then explores as far as possible until it finds destinations, otherwise it will go backward on the same path to find nodes to traverse. It will not end until it finds all destinations. The set of feasible Steiner trees is denoted by T^i:

$$T^i = \{t \mid t \text{ is a Steiner tree (or multicast tree) from } S^i \text{ to } R^i\}.$$

When the number of datacenters and destinations increases, the number of possible Steiner trees found by DFS will be very large. In order to reduce the complexity of our solution, we add some constraints for finding feasible trees. We classify the Steiner trees into two types: only one path contains all destinations and other trees. Using one path includes all destinations can save bandwidth efficiently for multicast transfers, so we keep this kind of paths in the process of DFS. For other trees, we limit the maximum hop number to be 2, which significantly reduces the number of possible Steiner trees with negligible performance loss.

B. Linear Program Formulation

Request Completion Time. The completion time of a request is measured from the moment the source starts to send data, to the time all the data have been received by all destinations. It includes the propagation delay, queueing delay and transmission delay. Propagation delay and queueing delay are in the order of milliseconds; since delay-tolerant transfers are always large transfers, these delays are negligible. We only consider transmission delay when we calculate the transfer completion time.

A transfer request i can be specified as a tuple $\{S^i, \mathrm{R}^i, \mathrm{Q}^i, \mathrm{D}^i, \mathrm{a}^i\}$. Large a^i represents a high priority. Our objective is to maximize the network throughput for all transfers and meet transfer deadlines as many as we can. Some transfers may not have deadlines, so we use very large value of deadlines for these transfers. We formulate the problem as the following linear program:

$$\begin{aligned}
\text{maximize} \quad & \chi && (1)\\
\text{subject to} \quad & \chi \le \sum_{t\in T^i} x^i(t), \forall \mathrm{i} = 1, ..., \mathrm{n}, && (2)\\
& \sum_{i=1}^{n} \sum_{t\in T^i} x^i(t)\, \phi(t, e) \le C(e), \forall \mathrm{e} \in E, && (3)\\
& D^i \sum_{t\in T^i} x^i(t) \ge Q^i, \forall \mathrm{i} = 1, ..., \mathrm{n}, && (4)\\
& x^i(t) \ge 0, \chi \ge 0, \forall t \in T^i, \forall \mathrm{i} = 1, ..., \mathrm{n}. && (5)
\end{aligned}$$

where ϕ is defined as:

$$\phi(t, \mathrm{e}) = \begin{cases} 1, & \text{if } \mathrm{e} \in t, \\ 0, & \text{otherwise.} \end{cases}$$

The linear program we formulate above can be solved by a standard LP solver efficiently. The objective of the problem is to maximize throughput for all requests, which is the sum of flow rates in all selected Steiner trees. $x^i(t)$ represents the flow rate for a Steiner tree t. Since flow rates of different requests contend for edge capacities, for each edge e, the summation of trees' flow rates that use edge e should not exceed the edge capacity. This is reflected in constraint (3). Constraint (4) ensures that all transfers will complete prior to deadlines. The flow rate $x^i(t)$ and throughput objective χ are guaranteed to be non-negative in constraint (5).

Post-Processing. However, it is possible that meeting all transfer deadlines will exceed link capacities, so the linear program may not have feasible solutions. When the linear program does not have feasible solutions, our approach is to reject the transfer which has the lowest priority. If there are multiple requests with the same priority, then we remove the request that needs the largest bandwidth.

C. Choose Sparse Solutions

The linear program we just formulated above has a collection of feasible solutions. Since we use multiple Steiner trees to deliver source data to destinations for each request, then it is inevitable to split data at the source, which will add splitting overhead. Besides, using multiple trees will also add packet reordering overhead at destinations. In order to reduce such overhead, we prefer to use few trees for distributing data, which needs us to choose sparse solutions from the feasible solutions. Therefore, we can add a penalty function at the objective:

$$\text{maximize} \quad \chi - \mu \sum_{i=1}^{n} \sum_{t\in T^i} g\left(x^i(t)\right), \qquad (6)$$

subject to the same constraints (2)$-$(5). And $g\left(x^i(t)\right)$ is defined as:

$$g\left(x^i(t)\right) = \begin{cases} 0, & \text{if } x^i(t) = 0, \\ 1, & \text{if } x^i(t) > 0. \end{cases}$$

Problem (6) is different from Problem (1) because we changed the objective function. In order to get the optimal throughput and use fewer trees, μ should not be too large or too small. If μ is too large, the solution can be far from optimality; if it is too small, many trees may be selected. In our experimental settings, we let $\mu = 0.01$ and Problem (6) returns almost the same throughput value as Problem (1), the error is smaller than 10^{-8}, which can be ignored. We will show this in our forthcoming experimental results.

Problem (6) is a non-convex optimization problem. A log-based heuristic is widely used for finding a sparse solution, the basic idea is to replace $g\left(x^i(t)\right)$ by $\log\left(|x^i(t)| + \delta\right)$, where δ is a small positive threshold value that determines what is close to zero. Since the problem is still not convex, we can linearize the penalty function which is inspired by [13] by using a weighted l1-norm heuristic:

$$\text{maximize} \quad \chi - \mu \sum_{i=1}^{n} \sum_{t\in T^i} \left(W^i(t) \cdot x^i(t)\right), \qquad (7)$$

subject to the same constraints (2)$-$(5). In each iteration we recalculate the weight function W^i where:

$$W^i(t) = \frac{1}{\left(x^i(t)\right)^k + \delta}.$$

Then Problem (6) becomes a linear problem, and it is solved iteratively. $\left(x^i(t)\right)^k$ is obtained from the kth iteration, δ is a small positive constant. We can see that if $\left(x^i(t)\right)^k$ is smaller, then the weight function W^i becomes larger, $x^i(t)$ will be smaller. Upon convergence, $\left(x^i(t)\right)^k \approx \left(x^i(t)\right)^{k+1} = \left(x^i(t)\right)^*$, for $\mathrm{i} = 1, ..., \mathrm{n}, \mathrm{t} \in \mathrm{T}^\mathrm{i}$, then:

$$W^i(t) \cdot \left(x^i(t)\right)^* = \frac{\left(x^i(t)\right)^*}{\left(x^i(t)\right)^k + \delta} = \begin{cases} 0, & \text{if } \left(x^i(t)\right)^* = 0, \\ 1, & \text{if } \left(x^i(t)\right)^* > 0. \end{cases}$$

Eventually, the transformed Problem (7) approaches the Problem (6) and yields sparse solutions. Algorithm 1 presents a summary of our solution.

D. Proof of Convergence

In this section, we provide a brief proof of convergence for the Problem (7), which is:

Algorithm 1 Deadline-aware routing for multiple multicast transfers.

```
1: Input: Transfer requests: $\{S^i, \mathrm{R}^i, \mathrm{Q}^i, \mathrm{D}^i, \mathrm{a}^i\}$; Network Topology $G = (V, E, C)$.
2: $k := 0$. Initialize $\delta = 10^{-8}$, $\left(W^i(t)\right)^0 = 1$, sparse_flag = False.
3: update $k = k + 1$.
4: If sparse_flag == False, given the solution $\left(x^i(t)\right)^k$ from the previous iteration, get $W^i(t) = \frac{1}{(x^i(t))^k + \delta}$, solve the linear program (7) to obtain flow rates $\left(x^i(t)\right)^{k+1}$, throughput optimal value $\chi^{k+1}$ and status.
5: If status == infeasible, remove the request with the lowest priority. Solve the linear program (7) with updated inputs to obtain $\left(x^i(t)\right)^{k+1}$, $\chi^{k+1}$ and status.
6: If status == optimal: if $\left(x^i(t)\right)^{k+1} \approx \left(x^i(t)\right)^k$, return $\left(x^i(t)\right)^* \approx \left(x^i(t)\right)^{k+1}$; else go to Step 3 for another iteration. If status == infeasible, go to Step 5.
7: Output: $\{x^i(t)\}$ and corresponding Steiner trees $\{t | t \in T^i\}$.
```

Proposition 1.

$$\begin{aligned} &\textit{maximize} && \chi - \mu \sum_{i=1}^{n} \sum_{t \in T^i} \frac{x^i(t)}{(x^i(t))^k + \delta} && (8) \\ &\textit{subject to} && x = \left(x^1(t), \ldots, x^n(t)\right) \in \mathcal{C}, \forall t \in T^i, && (9) \end{aligned}$$

with $\delta > 0$ *and* $x^i(t) \geq 0$*, for* $\mathrm{i} = 1, ..., \mathrm{n}$*, where* $C \subset \mathbb{R}^n$ *is a convex, compact set. When* $\mathrm{k} \to \infty$*, we have* $\left(x^i(t)\right)^{k+1} - \left(x^i(t)\right)^k \to 0$*, for all i,* $t \in T^i$*.*

Proof. Let N_i denotes the number of Steiner trees for request i. Since Problem (8) yield $\left(x^i(t)\right)^{k+1}$, and our objective is to minimize $\sum_{i=1}^{n} \sum_{t \in T^i} \frac{x^i(t)}{(x^i(t))^k + \delta}$, thus we have that:

$$\sum_{i=1}^{n} \sum_{t \in T^i} \frac{\left(x^i(t)\right)^{k+1} + \delta}{\left(x^i(t)\right)^k + \delta} \leq \sum_{i=1}^{n} \sum_{t \in T^i} \frac{\left(x^i(t)\right)^{k} + \delta}{\left(x^i(t)\right)^k + \delta} = \sum_{i=1}^{n} N_i. \tag{10}$$

Using the inequality between the arithmetic and geometric means, we have:

$$\begin{aligned} &\left(\frac{1}{\sum\limits_{i=1}^{n} N_i}\right) \sum_{i=1}^{n} \sum_{t \in T^i} \frac{\left(x^i(t)\right)^{k+1} + \delta}{\left(x^i(t)\right)^k + \delta} \geq \\ &\prod_{i=1}^{n} \prod_{t \in T^i} \left(\frac{\left(x^i(t)\right)^{k+1} + \delta}{\left(x^i(t)\right)^k + \delta}\right)^{\frac{1}{\sum_{i=1}^{n} N_i}}. \end{aligned} \tag{11}$$

If we combine Equation (10) and (11) together, we will get:

$$\prod_{i=1}^{n} \prod_{t \in T^i} \left(\frac{\left(x^i(t)\right)^{k+1} + \delta}{\left(x^i(t)\right)^k + \delta}\right)^{\frac{1}{\sum_{i=1}^{n} N_i}} \leq 1. \tag{12}$$

We let

$$\mathrm{A}\left(\left(x^i(t)\right)^k\right) = \left(\left(x^i(t)\right)^k + \delta\right)^{\frac{1}{\sum_{i=1}^{n} N_i}}. \tag{13}$$

Since $\left(x^i(t)\right)^k \geq 0$ and $\delta > 0$, thus $\mathrm{A}\left(\left(x^i(t)\right)^k\right)$ is bounded below by $\delta^{\frac{1}{\sum_{i=1}^{n} N_i}}$, then $\mathrm{A}\left(\left(x^i(t)\right)^k\right)$ will converge to a nonzero limit as $k \to \infty$, which implies that

$$\lim_{k \to \infty} \prod_{i=1}^{n} \prod_{t \in T^i} \left(\frac{\left(x^i(t)\right)^{k+1} + \delta}{\left(x^i(t)\right)^k + \delta}\right)^{\frac{1}{\sum_{i=1}^{n} N_i}} = 1. \tag{14}$$

Now, we combine Equation (14) with Equation (10) and (11), as $k \to \infty$, we have:

$$\sum_{i=1}^{n} N_i \leq \sum_{i=1}^{n} \sum_{t \in T^i} \frac{\left(x^i(t)\right)^{k+1} + \delta}{\left(x^i(t)\right)^k + \delta} \leq \sum_{i=1}^{n} N_i, \tag{15}$$

which equals to:

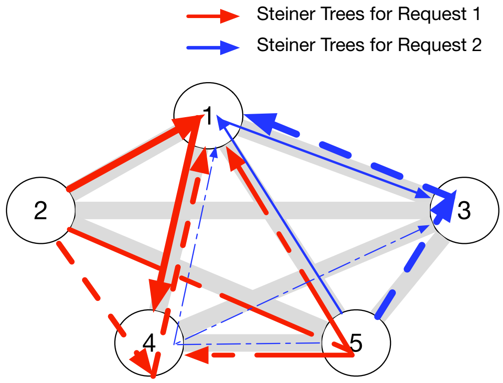
$$\lim_{k \to \infty} \sum_{i=1}^{n} \sum_{t \in T^i} \frac{\left(x^i(t)\right)^{k+1} + \delta}{\left(x^i(t)\right)^k + \delta} = \sum_{i=1}^{n} N_i. \tag{16}$$

Therefore, we have $\frac{(x^i(t))^{k+1} + \delta}{(x^i(t))^k + \delta} = 1$ when $k \to \infty$, which means that $\left(x^i(t)\right)^{k+1} \approx \left(x^i(t)\right)^k$. Convergence proved. □

E. An Example of the Optimal Solution

An example using the inter-datacenter network is shown in Figure 2. To simplify the example, we assume all link capacities are 15MB/s. Considering there are two requests R_1 and R_2. R_1 needs to send source data from datacenter 2 to datacenter 1 and 4; R_2 needs to send source data from datacenter 5 to datacenter 1 and 3. Table III gives detailed requirements of these two requests. We use this example to explain the benefit of our linear programming formulation. Our linear program tries to maximize throughput and meet deadlines for all requests, Figure 2 shows the optimal solution obtained by solving the linear program in Sec. IV-C. Our solution will split the source data at the sender based on the flow rate allocated to each tree, and send the data through different trees. We can see that both requests can meet their deadlines, R_2 can even finish the transfer before its deadline since the linear program aims at maximizing throughput.

If we treat each multicast transfer as multiple unicast transfers, $R1$ will miss its deadline, and this approach wastes a lot of bandwidth. DDCCast [11] finds only one minimum weight Steiner tree for each request. In our example, the largest capacity for one tree is only 15MB/s, if we use only one tree to distribute data, the shortest time to finish the transfer will be 20s, which still makes both R_1 and R_2 miss their deadlines. Using multiple trees increases throughput for a transfer, which can make more transfers meet deadlines.



Request 1		Request 2	
Trees	Rate	Trees	Rate
2 → 1 → 4	15	5 → 1 → 3	4.56
2 → 4 → 1	12.06	5 → 3 → 1	15
2 → 5 ↗ 1 ↘ 4	10.44	5 → 4 ↗ 3 ↘ 1	2.94

Fig. 2: An example of the optimal solution obtained by solving the linear program in Sec. IV-C for maximizing the total throughput of all requests.

Requests	Source	Destinations	Volume (MB)	Deadline (seconds)
R_1	2	1, 4	300	8
R_2	5	1, 3	300	18

TABLE III: Request requirements for the example.

V. Implementation

We have completed a real-world implementation in a software-defined overlay network testbed at the application layer. Different from the traditional SDN techniques, our application-layer SDN does not need to cope with the complicated lower layer properties and management. Besides, our application-layer solution has higher switching capacities, which can support more forwarding rules at the datapath node and scale well to a large number of transfer requests.

Figure 3 shows the high-level architecture of our application-layer solution. After we start the testbed, the controller and datapath nodes will establish persistent TCP connections between each other; we use *iperf* to measure bandwidth availability between each node and send it to the controller, which is an important input for making routing decisions. We employ a local aggregator at each datapath node; this aggregator helps to aggregate and schedule inter-datacenter flows. In our experiments, we use six Virtual Machines (VM) instances located in six different datacenters, and one of the VMs is also launched as the central controller.

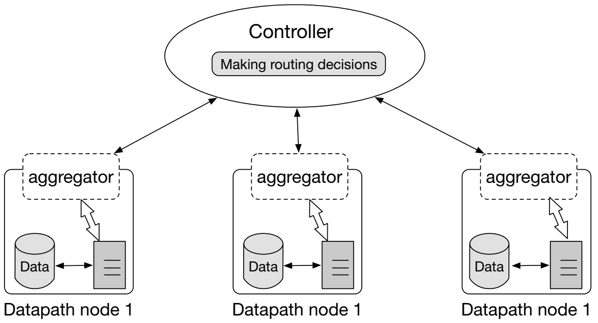


Fig. 3: Architecture of the application-layer SDN design.

Now we will explain how an inter-datacenter transfer is routed and completed through the application-layer SDN testbed. After a transfer request is submitted, the relative destination nodes will firstly subscribe to a specific channel by using a subscriber API implemented in Java, then the source node publishes its data, destinations, deadline requirement and priority information to the channel by using a publisher API. Source data will be aggregated at the local aggregator, then the aggregator consults the controller for routing rules. In the controller, our routing algorithm implemented in Python will compute routing rules by using bandwidth input and the request's information. Two types of routing rules will be published to each datapath node: one is {*'NodeID': xx, 'NextHop':xx, 'SessionId': xx*} which indicates the next-hop datacenter for the current datapath node; another is {*'NodeId': xx, 'Weight': xx, 'SessionId': xx*}, the value of *'Weight'* indicates the sending rate of the datapath node. After the aggregator gets routing rules, if we need to use multiple trees for sending data, then source data will be split at the source node. When data arrive at the aggregator of another node, the aggregator will check the rule. If *'NextHop'* is the node itself, then data will be delivered successfully and written back to the disk. If *'NextHop'* has a different node, then data will be relayed by the aggregator to another node.

In our experiments, we generate some transfer requests in a small time interval and try to send all of them before deadlines by using our routing algorithm. When a request is rejected, the controller will make a new routing decision after there are available capacities and send the decision to all datapath nodes.

VI. Performance Evaluation

We are now ready to evaluate the performance of our real-world implementation. In this section, we present our experimental settings and evaluation results. To make our evaluation more convincing, a large-scale simulation-based comparison will also be presented.

A. Experimental Setup

We have deployed our real-world implementation with the linear program routing algorithm on the Google Cloud Platform with six datacenters located geographically. In each

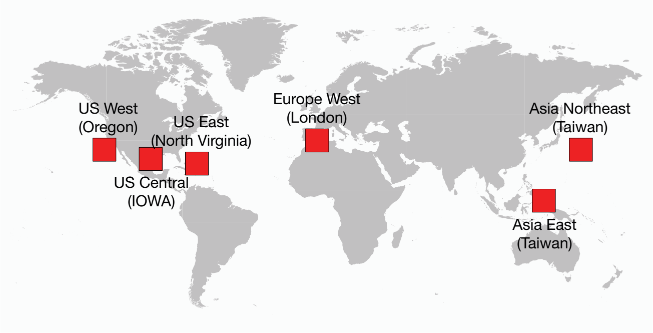


Fig. 4: The six Google Cloud datacenters used in our deployment and experiments.

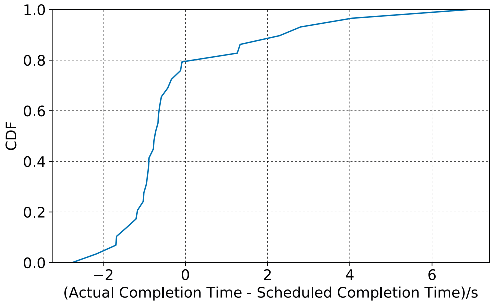


Fig. 5: Completion time deviation.

datacenter, we launch one Virtual Machine (VM). Locations of these datacenters are shown in Figure 4.

In our deployment, we use all VM instances located in different datacenters as datapath nodes, and VM instance in IOWA (US-central1-a) has been used as the controller of our application-layer testbed. All VM instances are of type n1-standard-4, each has 4 vCPUs, 15GB memory and 10GB Solid-State Drive. In each VM instance, we run Ubuntu 14.04 LTS system. In our experiment, we aim at showing the benefit of using multiple Steiner trees for transfers, so we use the Linux Traffic Control (TC) to make sure that each inter-datacenter link has uniform 120Mbps bandwidth.

We use the Linux command `truncate` to generate input files with a fixed size for each request. In our experiment, when a request is submitted, destinations of requests will first launch the Java API `subscriber()` to subscribe the request. After that, the VM instance with source data will launch the Java API `publisher()` to read blocks of the file, each block is 4MB, and publish blocks of data to the aggregator.

B. Evaluation Methodology

Workload. We use file replication as inter-datacenter traffic. For each transfer, we generate the source from six datacenters randomly and increase the number of destinations from 1 to 5. The volume of each file is set to be 300MB. For the deadline-constrained transfers, we choose deadlines from a uniform distribution between $[T, \alpha \mathrm{T}]$ as OWAN [27], α represents the tightness of deadlines. When α is small, then transfers have very close deadlines. And T is the most urgent deadline of all requests, which is related to the volume of transfer data and the number of transfers. The priority value is generated randomly for each transfer. We run our experiments in multiple time slots, at each time slot, six transfer requests will be generated at the beginning of the slot within a small time interval.

Performance Metrics. We measure two metrics: the inter-datacenter throughput and the percentage of requests that meet deadlines. The inter-datacenter throughput is obtained as the total size of all files transferred divided by the total transfer time to finish requests, the unit is Mbps.

We compare our solution with two solutions DDCCast [11] and Amoeba [9]. DDCCast finds only one tree for each request and schedules requests as late as possible. To maximize utilization at the current time slot, it pulls some traffic to the current slot and pushes forward other traffic close to deadlines. Since DDCCast only uses one tree for each request, so it can not accommodate some requests with early deadlines. Amoeba considers unicast transfers; it finds k-shortest paths for each source and destination pair.

C. Real-World Evaluation Results

Sparse Solution Performance. To show that our sparse solution does not affect the optimality, we compare the sparse solution with the original linear program without penalty in Table IV. From the table, we can see that the sparse routing approach has the same optimal value as the original linear program, and it uses much fewer trees than the original linear program.

Completion Time Deviation. We run the experiment in 10 time slots and get the completion time for each transfer request. The completion time is the time from destinations subscribe requests to all destinations receive the source data. In order to show our solution performs effectively in scheduling requests with deadline constraints, we plot the CDF figure of the difference between the actual completion time and scheduled completion time in Figure 5. From the figure, we observe that 80% of requests finished before the scheduled time, the possible reason is that we use TC to set the largest bandwidth as 120Mbps, in our routing decision, we use the same value. However, it is possible that sometimes the bandwidth used by flows can not reach 120Mbps. So we set the link bandwidth a little larger than 120Mbps in later experiments, which is 130Mbps.

Early Deadline Requests and Tightness Deadline Factor: Some requests may have early deadlines, our solution has better performance for these requests. When the bandwidth required by a request is more than each link capacity in the network, one routing tree is not enough for the request to meet its deadline. We generate the value of deadlines from a uniform distribution $[T, \alpha \mathrm{T}]$. In order to show the benefit of our solution for requests with early deadlines, we let $T = 10s$ and increase α from 1.2 to 4 to see the effect of the tightness factor.

Figure 6 presents the comparison of different solutions, and the number of destinations is 2. The x-axis is the tightness

Requests	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Optimal Value**
Workload 1	2/15	5/15	2/18	2/15	2/15	3/6	1/18	3/15	1/15	5/15	5.558/5.558
Workload 2	3/18	2/18	2/18	2/11	1/11	2/18	1/11	3/18	1/11	2/18	7.901/7.901

TABLE IV: Comparison of the sparse routing approach and original linear program, the left side represents the number of trees for sparse solution, the right side represents the number of trees for original linear program.

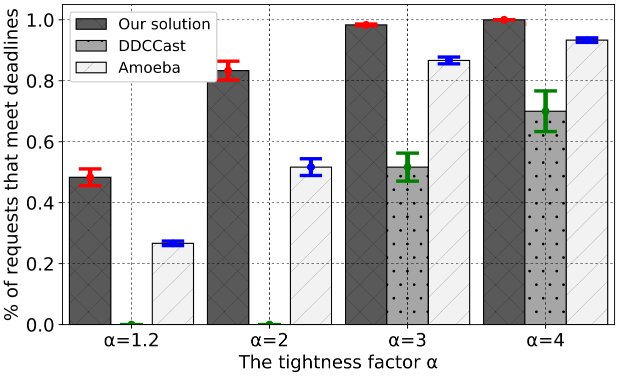


Fig. 6: Comparison of different solutions for different tightness factor.

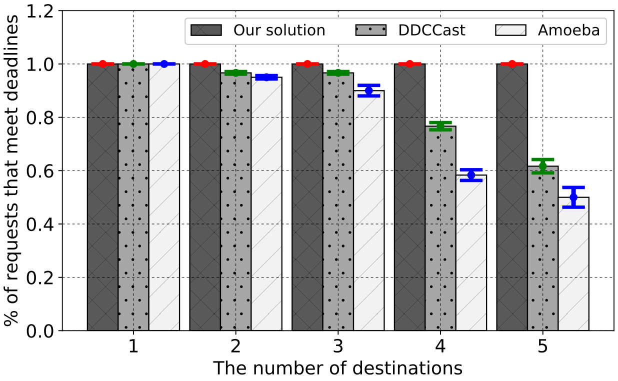


Fig. 7: Comparison of different solutions as the number of destinations increases.

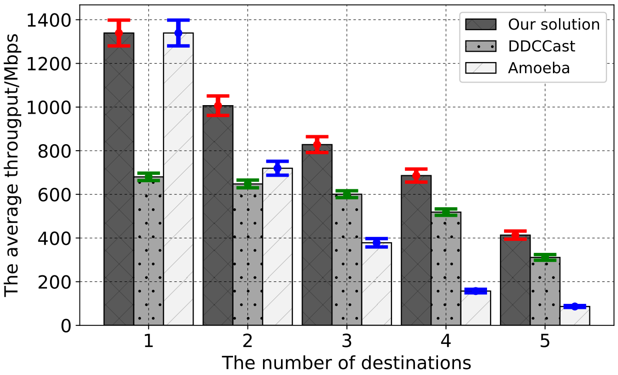


Fig. 8: Throughput comparison of different solutions.

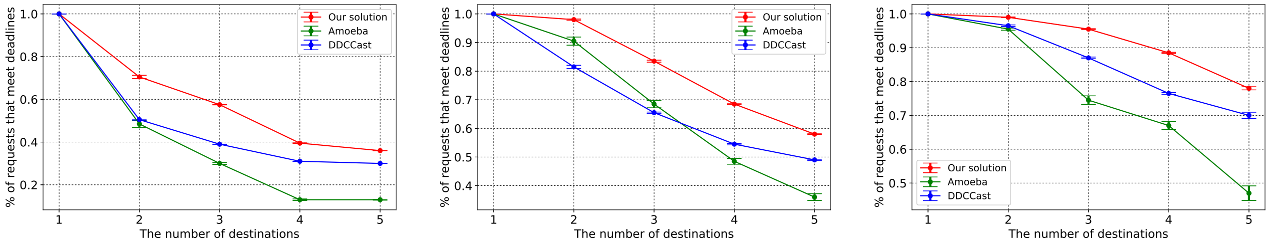
factor α; the y-axis represents the percentage of requests that meet deadlines. We can see that when the deadline ranges from 10s to 20s, DDCCast can not accommodate such request because the largest capacity for one tree is 120Mbps. As α increases, more requests can meet their deadlines because the range of deadlines becomes larger. Amoeba achieves lower percentage of requests that meet their deadlines since the unicast way uses more bandwidth for each transfer than our solution. DDCCast performs worse than Amoeba because it can not accommodate transfers that have deadlines earlier than 20s. The comparison result shows that our solution admits more early deadline requests than DDCCast and Amoeba.

Effect of the Number of Destinations. We increase the number of destinations from 1 to 5, and we set $\alpha = 2$, $T = 20s$. Figure 7 shows the percentage of requests that meet their deadlines as the number of destinations increases. As a consequence, our solution admits more requests than the other two solutions since our solution can highly utilize bandwidth by using multiple multicast trees for each transfer request. When the number of destinations increases, Amoeba does not have enough bandwidth to allocate for all source and destination pairs; DDCCast finds a minimum weight tree for each transfer request, which has less flexibility in routing, some requests may not have enough bandwidth to be scheduled.

Throughput. To demonstrate the throughput improvement of our solution, we plot the throughput performance in Figure 8. The average throughput is calculated as the total file size of all requests that meet their deadlines divided by the total transfer time. We only consider the throughput for requests that meet deadlines. Our solution has the maximum utilization of network bandwidth and admits more transfers than the other two solutions, so the throughput is also the highest. We can see that, when the number of destinations is 1 or 2, Amoeba has higher throughput than DDCCast. The possible reason is DDCCast always tries to push some transfers close to deadlines, which can make the transfer time longer than Amoeba.

D. Simulation-Based Comparison

We further present our comparison over large-scale simulations. In each scenario, we repeated 20 times and the average was measured and plotted in the figure. Figure 9 gives the comparison of different solutions as the number of destinations increases when the tightness factor is fixed. 20 requests are generated in each run. In Figure 10, the comparison is



(a)

(b)

(c)

Fig. 9: Comparison of different solutions as the number of destinations increases when the tightness factor : (a) $\alpha = 1$. (b) $\alpha = 2$. (c) $\alpha = 3$.

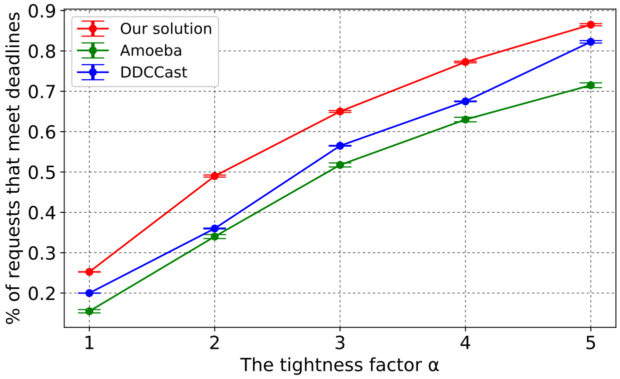


Fig. 10: Comparison of different solutions as the tightness factor increases.

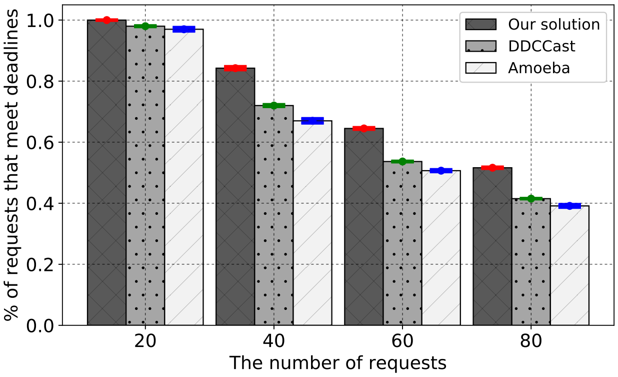


Fig. 11: Comparison of different solutions as the number of requests increases.

presented when the tightness factor increases. 40 requests are generated in each run and the number of destinations is set to be 3. Clearly, our solution outperforms DDCCast and Amoeba, admits more requests than the other two solutions, which is consistent with the real-world experiment.

We also check the benefits of our solution when the number of requests increases. Figure 11 shows the result. When there are 20 requests, our solution admits 3% more requests than DDCCast and Amoeba. When the number of requests increases to 80, our solution admits 10%-12% more requests than the other two baselines. It is obvious that our solution has a greater ability to accommodate more requests than other approaches.

Scalability. To show the scalability of our linear program with sparse solutions, we record the running time of the LP with different number of requests, which is shown in Figure 12. Our algorithm packs these requests at the same time. The running time is the average time of multiple runs. When the number of requests is 80, the running time is less than 4s, which is acceptable when compared with the transfer time of requests since these requests always need to transfer large files. The result proves that our solution is efficient and converges very fast. Moreover, the number of datacenters in practice is always small. Thus our solution is scalable.

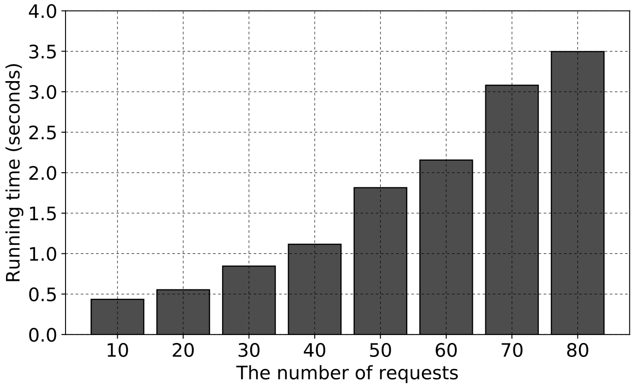


Fig. 12: The computation time of our approach.

In a nutshell, from the evaluation results, our solution maximizes the network throughput and admits more transfer requests than DDCCast and Amoeba. Compared with DDCCast, our solution can admit some requests that have early deadlines, which demand more bandwidth than each link capacity.

VII. Discussion

We now give some discussions about our future work.

Dynamic Resources. Our work assumes network resources are stable as previous related works. However, the network resources can change dynamically over the time. In the future work, we may consider dynamic resources in making routing decisions. The controller will measure bandwidth information at each time slot and repack the remaining requests under the current network resources.

Different Request Arrival Rates. Our work does not explore the effect of the request arrival rate. Since our objective is to maximize throughput and accommodate a maximal number of transfer requests with deadline requirements, we assume requests arrive in a small time interval at the beginning of each time slot. The results show that our solution has a good performance in routing requests that arrive closely. In the future work, we may add the time dimension to our formulation and explore the effect of different request arrival rates.

VIII. Conclusion

In this paper, we have presented our design of an efficient solution for multicast inter-datacenter transfers, which aims at maximizing the network throughput and meeting as many transfer deadlines as possible. Traditionally, initializing the multicast transfer as multiple independent unicast transfers can waste more bandwidth and let some other requests miss their deadlines. Thus we propose to use multiple Steiner trees for each multicast transfer. We formulate the problem as a linear program (LP) and find sparse solutions by using a weighted l1-norm heuristic. To prove the practicality and efficiency of our solution, we have implemented our idea in a software-defined overlay network testbed at the application layer. Google Cloud Platform is used for our real-world experiments with six Virtual Machine instances in six different datacenters. Experimental results and large-scale simulations have clearly shown that our design performs substantially better in maximizing throughput and meeting transfer deadlines than closely related existing work.

References

[1] M. Noormohammadpour, C. S. Raghavendra, and S. Rao, "DCRoute: Speeding up Inter-Datacenter Traffic Allocation While Guaranteeing Deadlines," in *Proc. IEEE International Conference on High Performance Computing (HiPC)*, 2016.

[2] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-Driven WAN," in *Proc. ACM SIGCOMM*, 2013.

[3] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache, "Dynamic Pricing and Traffic Engineering for Timely Inter-Datacenter Transfers," in *Proc. ACM SIGCOMM*, 2016.

[4] K. Jain, M. Mahdian, and M. R. Salavatipour, "Packing Steiner Trees," in *Proc. ACM-SIAM symposium on Discrete algorithms*, 2003.

[5] Y. Wu, P. A. Chou, and K. Jain, "A Comparison of Network Coding and Tree Packing," in *Proc. International Symposium on Information Theory (ISIT)*, 2004.

[6] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a Globally-Deployed Software Defined WAN," in *Proc. ACM SIGCOMM*, 2013.

[7] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila *et al.*, "BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing," in *Proc. ACM SIGCOMM*, 2015.

[8] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula, "Calendaring for Wide Area Networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 515–526, 2015.

[9] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang, "Guaranteeing Deadlines for Inter-Data Center Transfers," *IEEE/ACM Transactions on Networking*, 2016.

[10] M. Noormohammadpour, C. S. Raghavendra, S. Rao, and S. Kandula, "DCCast: Efficient Point to Multipoint Transfers Across Datacenters," in *Proc. USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2017.

[11] M. Noormohammadpour and C. S. Raghavendra, "DDCCast: Meeting Point to Multipoint Transfer Deadlines Across Datacenters Using ALAP Scheduling Policy," *arXiv preprint arXiv:1707.02027*, 2017.

[12] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge university press, 2004.

[13] M. S. Lobo, M. Fazel, and S. Boyd, "Portfolio Optimization with Linear and Fixed Transaction Costs," *Annals of Operations Research*, vol. 152, no. 1, pp. 341–365, 2007.

[14] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing Tables in Software-Defined Networks," in *Proc. IEEE INFOCOM*, 2013.

[15] Yu, Minlan and Rexford, Jennifer and Freedman, Michael J and Wang, Jia, "Scalable Flow-Based Networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[16] B. Leng, L. Huang, X. Wang, H. Xu, and Y. Zhang, "A Mechanism for Reducing Flow Tables in Software Defined Network," in *Proc. IEEE International Conference on Communications (ICC)*, 2015.

[17] L.-H. Huang, H.-J. Hung, C.-C. Lin, and D.-N. Yang, "Scalable and Bandwidth-Efficient Multicast for Software-Defined Networks," in *Proc. IEEE Global Communications Conference (GLOBECOM)*, 2014.

[18] S.-H. Shen, L.-H. Huang, D.-N. Yang, and W.-T. Chen, "Reliable Multicast Routing for Software-Defined Networks," in *Proc. IEEE INFOCOM*, 2015.

[19] X. Chen, M. Chen, B. Li, Y. Zhao, Y. Wu, and J. Li, "Celerity: A Low-Delay Multi-Party Conferencing Solution," in *Proc. ACM international conference on Multimedia*, 2011.

[20] Y. Feng, B. Li, and B. Li, "Airlift: Video Conferencing as a Cloud Service Using Inter-Datacenter Networks," in *Proc. IEEE International Conference on Network Protocols (ICNP)*, 2012.

[21] Y. Liu, D. Niu, and B. Li, "Delay-Optimized Video Traffic Routing in Software-Defined Interdatacenter Networks," *IEEE Transactions on Multimedia*, vol. 18, no. 5, pp. 865–878, 2016.

[22] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez, "Inter-Datacenter Bulk Transfers with Netstitcher," in *Proc. ACM SIGCOMM*, 2011.

[23] N. Laoutaris, G. Smaragdakis, R. Stanojevic, P. Rodriguez, and R. Sundaram, "Delay-Tolerant Bulk Data Transfers on the Internet," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 6, pp. 1852–1865, 2013.

[24] B. B. Chen and P. V.-B. Primet, "Scheduling Deadline-Constrained Bulk Data Transfers to Minimize Network Congestion," in *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2007.

[25] Y. Wang, S. Su, A. X. Liu, and Z. Zhang, "Multiple Bulk Data Transfers Scheduling Among Datacenters," *Computer Networks*, vol. 68, pp. 123–137, 2014.

[26] Y. Wu, Z. Zhang, C. Wu, C. Guo, Z. Li, and F. C. Lau, "Orchestrating Bulk Data Transfers Across Geo-Distributed Datacenters," *IEEE Transactions on Cloud Computing*, vol. 5, no. 1, pp. 112–125, 2015.

[27] X. Jin, Y. Li, D. Wei, S. Li, J. Gao, L. Xu, G. Li, W. Xu, and J. Rexford, "Optimizing Bulk Transfers with Software-Defined Optical WAN," in *Proc. ACM SIGCOMM*, 2016.