DCRoute: Speeding up Inter-Datacenter Traffic Allocation while Guaranteeing Deadlines

Mohammad Noormohammadpour, Cauligi S Raghavendra, Sriram Rao

DCRoute: Speeding up Inter-Datacenter Traffic Allocation while Guaranteeing Deadlines

Mohammad Noormohammadpour
University of Southern California
noormoha@usc.edu

Cauligi S. Raghavendra
University of Southern California
raghu@usc.edu

Sriram Rao
Microsoft
sriramra@microsoft.com

***Abstract*—Datacenters provide the infrastructure for cloud computing services used by millions of users everyday. Many such services are distributed over multiple datacenters at geographically distant locations possibly in different continents. These datacenters are then connected through high speed WAN links over private or public networks. To perform data backups or data synchronization operations, many transfers take place over these networks that have to be completed before a deadline in order to provide necessary service guarantees to end users. Upon arrival of a transfer request, we would like the system to be able to decide whether such a request can be guaranteed successful delivery. If yes, it should provide us with transmission schedule in the shortest time possible. In addition, we would like to avoid packet reordering at the destination as it affects TCP performance. Previous work in this area either cannot guarantee that admitted transfers actually finish before the specified deadlines or use techniques that can result in packet reordering. In this paper, we propose DCRoute, a fast and efficient routing and traffic allocation technique that guarantees transfer completion before deadlines for admitted requests. It assigns each transfer a single path to avoid packet reordering. Through simulations, we show that DCRoute is at least 200 times faster than other traffic allocation techniques based on linear programming (LP) while admitting almost the same amount of traffic to the system.**

***Index Terms*—Datacenter; Routing; Traffic Allocation; Traffic Scheduling; Deadlines; Wide Area Networks;**

I. Introduction

Cloud Computing allows customers to build online applications that can cost effectively scale as necessary [1]. It provides a massive pool of resources for online applications that can be flexibly obtained when needed and then returned back to the pool at a later time. Such resources can be provided to different applications on top of the infrastructure built and maintained by a cloud company. Examples of hosted online applications include video streaming, data storage and sharing, and big data processing. Most companies that provide cloud computing services own multiple datacenters placed in different cities and countries in order to improve availability, reduce end-to-end delays for end users, and provide customized regional services. At the time of writing this paper, Amazon has more than two dozen availability zones each consisting of one or more discrete datacenters [2], Microsoft has 22 regions with plans to build 8 more [3], and Google relies on more than a dozen datacenters [4].

Many applications hosted on these datacenters need to transfer data to their peers in other datacenters for the purpose of data replication and synchronization [5], [6]. The aim is to improve fault-tolerance and user quality of service by making multiple copies of data and getting data closer to end users. Most of these transfers have to be completed before a deadline in order to meet customer service level agreements (SLAs) and they can take hours to complete [7]. For example, search engines may have to exchange data among datacenters in order to synchronize their search databases and storage applications may need to back up user data over certain periods.

As mentioned in [8], inter-datacenter traffic can be categorized into three groups: *interactive* traffic that has to be transmitted as soon as possible since it is in the critical path of user experience, *elastic* traffic that requires timely delivery which can be modeled in the form of a deadline, and *background* traffic which is bandwidth hungry and has less priority than the other two categories of traffic. In this paper, we focus on elastic traffic with user specified deadlines.

Previous work in the context of inter-datacenter traffic scheduling either fails to consider the negative effects of packet reordering caused by multiplexing packets over different paths (AMOEBA [6]) or cannot guarantee that admitted requests will actually complete transmission before the deadlines specified by customers (B4 [9], SWAN [8], TEMPUS [7]). In addition, AMOEBA and TEMPUS which are the state-of-the-art techniques in this area, model the allocation problem as large linear programs (LP), with possibly hundreds of thousands of variables, solving which incurs large memory and CPU overhead and can take a long time.

Avoiding packet reordering allows data to be instantly delivered to applications upon arrival of packets. In addition, inter-datacenter networks have characteristics similar to WAN networks (including asymmetric link delays and large delays for links that connect distant locations) for which multiplexing packets over different paths has been shown to considerably degrade TCP performance [10]. Putting out of order packets and segments back in order can be expensive in terms of memory and CPU usage, especially when transmitting at high rates.

As explained in [11], TCP needs to buffer as much as the bandwidth-delay product (BDP) of the network in lossless and $2 \times BDP$ in lossy networks to put out of order packets back in order. For high speed inter-datacenter networks with tens of gigabits of speed and tens of milliseconds of latency considerable amount of buffering may be needed. In [12], authors show how Vanilla kernel uses 50% more CPU in

presence of severe packet reordering and present Juggler, a reordering resilient network stack designed for low latency datacenter networks which can reduce the extra CPU utilization to 10%: still a considerable amount. For higher latency networks, due to large variation of RTT over multiple paths, reordering may further increase CPU utilization.

In [13], we proposed RCD, a technique that speeds up the traffic allocation problem by scheduling transfers close to their deadlines. Through simulations, we showed that RCD speeds up the allocation process by allowing new transfers to be scheduled only considering the residual bandwidth which would result in creation of much smaller LP models. However, we did not discuss the reordering problem and only evaluated our technique for a single link scenario.

In this paper, we propose DCRoute, a fast and efficient routing algorithm which eliminates the need for LP modeling and:

- **Guarantees that admitted transfers complete prior to specified deadlines.**
- **Schedules all packets of each transfer over the same path to avoid packet reordering.**
- **Works much faster than other techniques while admitting almost equal traffic to the system.**

The input to DCRoute is the network topology as well as a list of transfers including their volumes and deadlines which are submitted to DCRoute in the order of arrival. DCRoute assigns a single path to every transfer and generates a transmission schedule which specifies the rate at which every transfer should be sent over their assigned path during current timeslot. In this paper, we verify the performance of DCRoute through long running simulations. In practice, label switching (such as VLAN tagging) can be used to enforce paths and rate-limiting at the end hosts can be used to enforce transmission rates as in SWAN [8].

II. Problem Description

Figure 1 shows our problem setup which is comprised of multiple datacenters in different locations managed by a central controller. The datacenters are connected using high speed WAN links. Applications hosted on these datacenters will make transfers that may take hours to complete and have to be finished before specified deadlines. Due to large transfer time, the time it takes for the source datacenter to request a path from the central controller and the time it takes to setup such a path is considered negligible. However, the time it takes for the scheduling algorithm to prepare a traffic schedule depends on the scheduling algorithm which we aim at minimizing. In addition, in order to avoid packet reordering, we would prefer to send all packets of a transfer on a single path.

As in [6], [8], we assume the timeline is divided into properly sized timeslots over which the transmission rate is constant. Using a slotted timeline allows for schedules with variable transmission rates over time. In our model, an

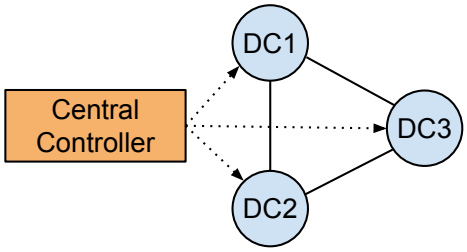

Fig. 1. Problem Setup

inter-datacenter request R is represented with four parameters $src(R)$, $dst(R)$, $vol(R)$ and $dl(R)$ which stand for source, destination, volume of data to be transferred and the deadline prior to which the transfer has to be completed.

Similar to previous work [6], we assume arriving requests are put into a queue and processed in the order of arrival and that there is no preemption: once a request is allocated, it cannot be unallocated from the system. At each moment, we have two parameters t_{now} and t_{end} which represent current timeslot and the latest deadline among all active requests, respectively. A request arriving sometime in timeslot t can be allocated starting timeslot $t+1$ since the schedule and transmission rate for current timeslot is already decided upon and broadcast into all datacenters. Also, at any moment t, t_{now} is the timeslot that includes t (current timeslot), and $t_{now}+1$ is the next available timeslot for allocation (next timeslot).

Upon arrival of a request, a central controller decides whether it is possible to allocate it considering some criteria that includes the total available bandwidth over future timeslots. If there is not enough room to allocate a request, the request is rejected and can be submitted to the system again later with a new deadline.

A request is considered **active** if it is accepted into the system and its deadline has not passed yet. Some active requests may take many timeslots to complete transmission. The total unsatisfied demand of an active request is called the residual demand of that request.

Allocation Problem: *Given active requests R_1 through R_n with residual demands D_1 to D_n ($0 \leq D_i,\ 1 \leq i \leq n$), is it possible to allocate a new request R_{n+1}? If yes, what is a possible schedule?*

An important characteristic of network traffic is that the size of smallest traffic unit (which is a packet) is significantly smaller than link capacities (which are in the range of gigabits nowadays). This allows us to solve the allocation problem by forming a linear program (LP) considering capacity constraints of the network edges as well as demand constraints of requests. The answer will give us a possible allocation if the constructed LP is feasible. Although this solution maybe straightforward, considering the number of active requests, number of links in network graph, and how far we are planning ahead into the future (t_{end}), the resulting LP could be large and may take a long time to solve. One of the ways to speed up this process is to limit the number of possible paths between every

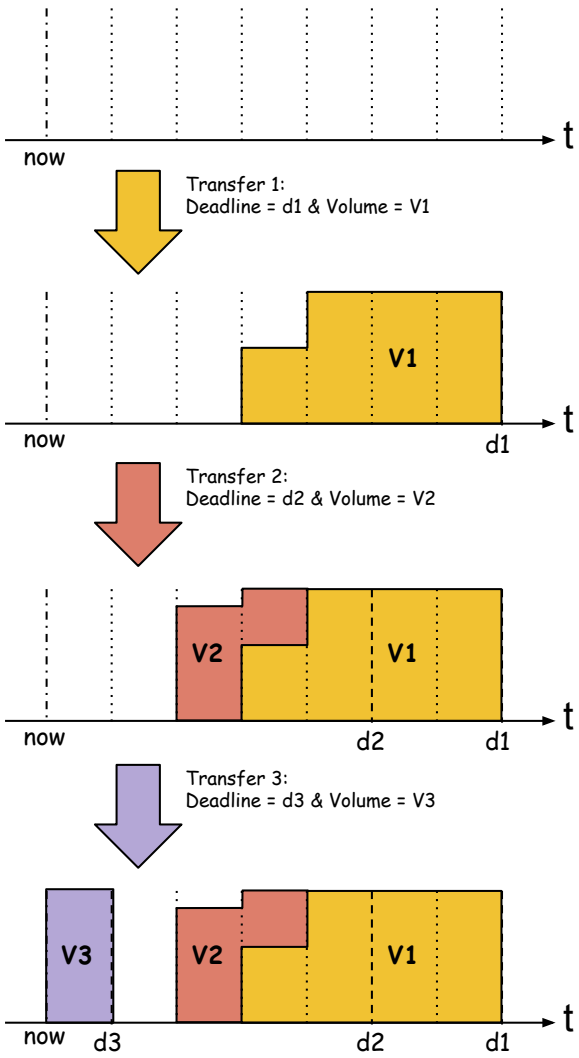

Fig. 2. An example of "As Late As Possible" allocation

pair of nodes [7], for example, using k-shortest paths [6]. While solving the LP, another speedup method is to limit the number of considered active requests based on some criterion [6] such as having a common link with the new request. It is also possible to use customized iterative methods to solve the resulting LP models faster based on the solutions of previous LP models in a way similar to the water filling process [7].

Our proposition is to avoid building an LP model in the first place by trying to allocate new requests only knowing the residual bandwidth on the edges for different timeslots.

III. DCRoute

DCRoute relies on the following three techniques:

- **Requests are initially allocated as late as possible (ALAP) [14]**
- **Utilization is maximized by pulling traffic from closest future timeslots into the upcoming timeslot**
- **A variant of BFS search is used for path selection**

Figure 2 provides an example of the ALAP allocation technique. As can be seen, when the first transfer is received the timeline is empty and therefore it is allocated adjacent to its deadline. The second transfer is allocated as close as possible to its deadline. The benefit of this type of scheduling is that requests do not use resources until it is absolutely necessary. This means resources will be available to other requests that currently demand them. Now when the third transfer is submitted, the resources are free and it just grabs as much bandwidth as needed. If we had allocated the first two requests closer to current time we may have had to either reject the third transfer or move the first two transfers ahead freeing resources for the third transfer.

Assume requests R_1 through R_n are current active requests and we would like to allocate R_{n+1}. For every request R_i, $1 \leq i \leq n$, we either have $dl(R_i) \leq dl(R_{n+1})$ or $dl(R_i) > dl(R_{n+1})$. In the former case, since there is no preemption, there is no way to increase the chance of new request being accepted by shifting the traffic allocation of request i away as it has to be completed before $dl(R_{n+1})$. For the latter case, since we allocated all requests ALAP, the traffic is already shifted out of R_{n+1}'s window as much as possible. As a result, it is possible to decide on admission of new request by just looking at the residual bandwidth on the links. For a single link, since all requests use a single shared resource (link capacity) this technique allows us to optimally decide whether a new request can be allocated. For a network, each request is routed on multiple links and there are many ways to schedule requests ALAP. If some link is used by multiple requests that are routed on different edges, how traffic is allocated on the common link can affect multiple other links which will affect the requests that use those links later on. Despite this uncertainty, we will show that using this feature, we can greatly speed up the allocation process.

Using ALAP alone can result in poor utilization as we always push traffic towards future timeslots and leave the current timeslot underutilized. In order to maximize utilization, upon beginning of a new timeslot, the scheduler looks at future timeslots and pulls as much traffic as possible from the closest timeslots in the future to the upcoming timeslot. Pulling from closest timeslot allows the ALAP characteristic of allocation to hold true afterwards: all residual demands will still be allocated as close to their deadlines as possible.

While pulling traffic from future timeslots, a request can span over multiple links and if there is some traffic fully occupying the next timeslot on one of those links, it is impossible to pull traffic back for that request from the future timeslots to the next timeslot. In such cases, we may need to pull traffic from requests that may not be the closest to current timeslot. This can result in an allocation in which some requests are not scheduled ALAP because they can be pushed further into the future. To address this problem, after pulling traffic from future timeslots, we sweep requests allocated on future timeslots and push them forward as much as possible.

Figure 3 shows an example of this process. There are three different requests all of which having the same deadline. It

is not possible to pull back the green request as the link $E1$ is already occupied. Therefore, we have to pull the orange request (PullBack phase). Afterwards, the allocation is not ALAP anymore, so we push the green request towards its deadline (PushForward phase). The final allocation is ALAP and the utilization of upcoming timeslot is maximum.

A. DCRoute Algorithms

We assume a graph $G(V, E)$ connecting datacenters with M bidirectional links. For simplicity, we also assume all links/edges have equal capacity of 1.0. Every edge e has a boolean *use* property which identifies whether that edge can be used in the course of routing. Also each node v has 4 parameters *v*, *r*, *rv* and *rs*. If p is the path on BFS tree from src(R) ending at v, these parameters represent whether v has been visited before, the number of hops to v from source, the load of bottleneck link, and the sum of loads on all edges from source to v, respectively. Moreover, we have variables $S_{t,m}$ that represent the total sum of traffic over link m from time $t_{now}+1$ to t. Every time a new request is submitted to the system, t_{end} is updated so that it covers all active requests. Finally, we define the **active window** as the set of all timeslots over all edges from time $t_{now}+1$ to t_{end}. Our algorithms only operate on the active window.

Allocate(R): Algorithm 1 is executed upon arrival of a new request R and performs multiple BFS searches. In every search, we calculate multiple costs, remove edges with highest costs, and compare the result with previous steps. While examining different paths, our heuristic finds the path with most preferred characteristics: the total sum of traffic before dl(R) over the chosen path will be minimal compared to other paths when R gets allocated on that path.

In each round, the path assignment algorithm starts by choosing the path with the least number of hops and calculates the total cost of sending the request on that path. Also, the bottleneck link on that path is identified. If the total cost is less than the best path found in previous steps, this path replaces the best path. Next, all edges with an equal or higher cost than the bottleneck edge are removed from the network. This process continues until there is no path from source to destination. Now, if it is possible to allocate the request on the selected path, we apply the allocation, otherwise, the request is rejected since admitting it might result in many more future requests to be rejected.

PullBack(): Algorithm 2 looks at the timeslots starting $t_{now}+2$ to t_{end} and pulls back traffic to $t_{now}+1$ (next timeslot to be scheduled). When pulling back traffic, all edges on a request's path have to be checked for unused capacity and updated together as we pull traffic back.

PushForward(): After pulling some traffic back, it may be possible for some other traffic to be pushed ahead even further. Algorithm 3 scans all future timeslots starting $t_{now}+2$ and makes sure that all demands are allocated ALAP. If not, it moves as much traffic as possible to the future timeslots until all residual demands are ALAP.

Algorithm 1

```
1: procedure ALLOCATE(R)
2:     for t = t_end + 1 to R.dl do
3:         for e ∈ E do
4:             S_{t,e} ← S_{t−1,e}
5:     t_end ← max(R.dl, t_end)
6:     for e ∈ E do
7:         e.use ← True
8:     for v ∈ V do
9:         {v.v, v.r, v.rv, v.rs} ← {False, 0, 0, 0}
10:    SE ← array of edges sorted by S_{R.dl,e} descending
11:    i ← 0, j ← 1, flag ← True, r_b ← BFS_test(R)
12:    while r_b.r is not −1 do
13:        if (r_b.rv > 0) and flag then
14:            flag ← False
15:            while S_{R.dl,SE[i]} > r_b.rv do
16:                SE[i].use ← False
17:                i ← i + 1
18:        SE[i].use ← False, i ← i + 1
19:        r_n ← BFS_test(R)
20:        if r_n.r == −1 then break
21:        α ← r_b.r × R.vol + r_b.rs
22:        β ← r_n.r × R.vol + r_n.rs
23:        if (α > β) or ((α == β) and (r_b.rv > r_n.rv)) then
24:            j ← i + 1, r_b ← r_n, flag ← True
25:    while i ≥ j do
26:        i ← i − 1, SE[i].use ← True
27:    path ← edges of the path ending at R.dst on the BFS
28:            tree starting at R.src
29:    if PathAllocate(path, R, False) then
30:        return PathAllocate(path, R, True)
31:    else
32:        return False

33: procedure BFS_TEST(R)
34:    for v ∈ V do
35:        {v.v, v.r, v.rv, v.rs} ← {False, 0, 0, 0}
36:    Q ← a queue having R.src as first element
37:    R.src.v ← True
38:    while Q is not empty do
39:        node ← head of the Q removed
40:        if node == R.dst then
41:            return {node.r, node.rv, node.rs}
42:        for next ∈ all neighbors of node do
43:            e ← edge connecting node to next
44:            if e.use == True and next.v == False then
45:                add next to Q
46:                next.v ← True
47:                next.r ← next.r + 1
48:                next.rv ← max(node.rv, S_{R.dl,e})
49:                next.rs ← node.rv + S_{R.dl,e}
50:    return {−1, −1, −1}

51: procedure PATHALLOCATE(path, R, apply)
52:    vol ← R.vol
53:    for t = R.dl to t_now + 2 step −1 do
54:        space ← vol
55:        for e ∈ path do
56:            space ← min(space, 1 − S_{t_now+1,e})
57:        if space > 0 then
58:            vol ← vol − space
59:            if apply then
60:                for e ∈ path do
61:                    add space of R to edge e at time t
62:                    for t' = t to t_end do
63:                        S_{t',e} ← S_{t',e} + space
64:    return vol == 0
```

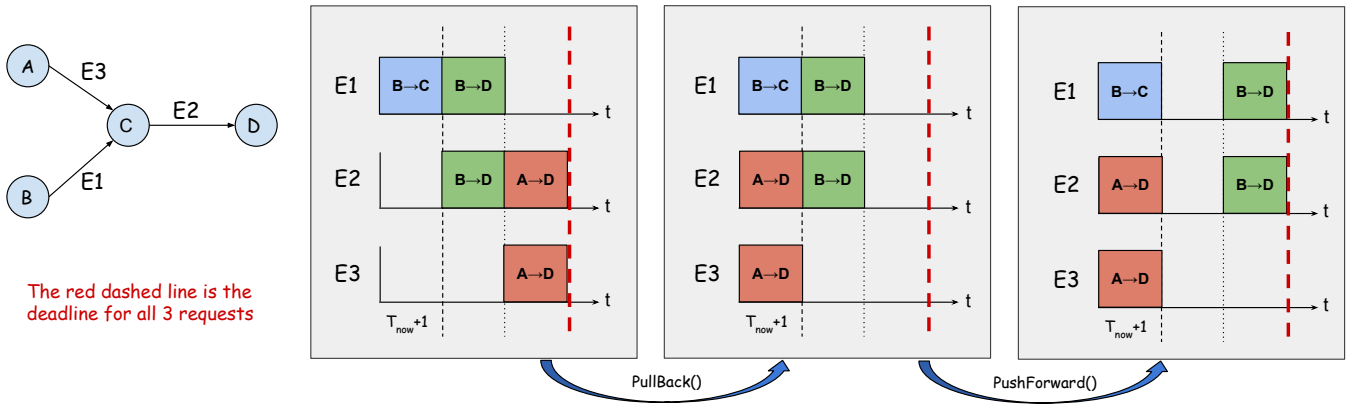

Fig. 3. An example of improving utilization while keeping the final allocation ALAP

Algorithm 2

```
1: procedure PULLBACK()
2:     for t = t_now + 1 to t = t_end do
3:         for e ∈ E do
4:             reqs ← all requests allocated on e at t
5:             for R ∈ reqs do
6:                 vol ← how much of R allocated on e at t
7:                 path ← path assigned to R upon allocation
8:                 for e' ∈ path do
9:                     vol ← min(vol, 1 − S_{t_now+1,e'})
10:                if vol > 0 then
11:                    for e' ∈ path do
12:                        move vol of R from edge e' at t
13:                        to edge e' at t_now + 1
14:                        for t' = t_now + 1 to t' = t do
15:                            S_{t',e'} ← S_{t',e'} + vol
```

Algorithm 3

```
1: procedure PUSHFORWARD()
2:     for t = t_now + 2 to t = t_end do
3:         for e ∈ E do
4:             reqs ← all requests allocated on e at t
5:             for R ∈ reqs do
6:                 vol ← how much of R allocated on e at t
7:                 path ← path assigned to R upon allocation
8:                 for t_2 = R.dl to t_2 = t + 1 step −1 do
9:                     free ← min(vol, free space on e at t_2)
10:                    for e' ∈ path do
11:                        free ← min(free, free space on e' at t_2)
12:                    if free > 0 then
13:                        for e' ∈ path do
14:                            remove free of R from edge e' at t
15:                            add free of R to edge e' at t_2
16:                            for t' = t to t' = t_2 do
17:                                S_{t',e'} ← S_{t',e'} − free
```

Walk(): Algorithm 4 is executed when the allocation for next timeslot is final. This algorithm tells each datacenter of the decided allocation and adjusts request demands accordingly by deducting what is scheduled to be sent from the total demand.

B. DCRoute and Multi-Path Routing

As mentioned earlier, to avoid packet reordering, DCRoute maps every transfer to exactly one path and if there is no single path that can allocate a transfer, it will be rejected. Although this sounds too restrictive, we show in the next section that

Algorithm 4

```
1: procedure WALK()
2:     Broadcast the schedule for t_now + 1 to all datacenters
3:     for e ∈ E do
4:         D ← S_{t_now+1}
5:         for t = t_now + 1 to t = t_end do
6:             S_{t,e} ← S_{t,e} − D
7:     t_now ← t_now + 1
8:     t_end ← max(t_end, t_now + 1)
```

limiting every transfer to a single path results in 2% less utilization in the worst case.

To use the aggregate bandwidth on multiple paths, one can use Multi-Path TCP (MPTCP) [15] which allows sending traffic from multiple interfaces of a single host. MPTCP is based on multiple sub-flows that behave similar to single TCP flows. While using MPTCP, the overall CPU and memory footprint can increase significantly compared to TCP which is the cost paid for increased bandwidth. However, by carefully choosing which parts of the data goes over what path, it is possible to improve the CPU footprint [11].

To increase network utilization, it is possible to use MPTCP and apply DCRoute to all sub-flows created. To do that, one can divide the total demand of a request over multiple sub-flows and assign the same deadline as the deadline of the original request to all of them. If all sub-flows can be allocated with DCRoute, then the request is accepted.

Deciding on the number of sub-flows and the portion of traffic that goes over each one of them is not a trivial problem. It may be possible to extend DCRoute by ranking the paths found in **Allocate** procedure and choosing a subset of the best paths. Further discussion of this subject is out of the scope of this paper.

IV. Simulation Results

In this section, we perform simulations to evaluate the performance of DCRoute. We generate synthetic traffic requests with Poisson arrival and input the traffic to both DCRoute and a few other techniques that can be used for traffic allocation. Two metrics are being measured and compared: **allocation**

time and **portion of rejected traffic** both of which are desired to be small.

Simulation Parameters: We used the same traffic distributions as described in [6]. Requests arrive with Poisson distribution of rate λ. Also, total demand of each request R is distributed exponentially with mean $\frac{1}{8}$ proportional to the maximum transmission volume possible prior to $dl(R)$. In addition, the length of requests is exponentially distributed for which we assumed a mean of 10 timeslots. We performed the simulations over 500 timeslots.

All simulations were performed on a machine with an Intel Core i7-6700T CPU and 24 GBs of memory and the algorithms are coded in Java. To solve linear programs faster, we used Gurobi Optimizer [16] with a free academic license. Gurobi Optimizer can speedup the LP solving process using several techniques including parallel processing. All simulations presented here are performed 3 times and the average is reported. We compare DCRoute with the following allocation schemes for all of which we used the same objective function as [6]:

Global LP: This technique is the most general and flexible way of allocation which routes traffic over all possible edges. All active requests are considered for all timeslots on all edges creating a potentially large linear program. The solution here gives us a lower bound on traffic rejection rate.

K-Shortest Paths: Same as Global LP, however, only the K-Shortest Paths between each pair of nodes are considered in routing. The traffic is allocated using a linear program over such paths. We simulated four cases of $K \in \{1, 3, 5, 7\}$. It is obvious that as K increases, the overall rejection rate will decrease as we have higher flexibility for choosing paths and multiplexing traffic.

Pseudo-Integer Programming (PIP): In terms of traffic rejection rate, comparing DCRoute with the previous two techniques is not fair as they allow multiplexing packets on multiple paths. **The aim of this technique is to find a lower bound on traffic rejection rate when all packets of each request are sent over a single path.** To do so, the general way is to create an integer program involving a list of possible paths (maybe all paths) for the new request and fixed paths for requests already allocated. The resulting model would be a non-linear integer program which cannot be solved using standard optimization libraries available. We instead created a number of linear programs each assigning one of the possible K-Shortest Paths for the newly arriving request. We then compare the objective values manually and choose the best possible path. In our implementation, we chose $K = 20$. This K seems to be more than necessary as we saw negligible improvement in traffic rejection rate even when increasing K from 5 to 7. Using PIP, the path over which a request is transferred is decided upon admission and does not change afterwards. We implemented two versions of this scheme:

- **Pure Minimum Cost (PMC):** We choose the path that results in smallest objective value.
- **Shortest Path, Minimum Cost (SPMC):** Amongst all shortest paths that result in a feasible solution and have the least number of hops, we choose the one with smallest objective value.

A. Google's GScale Network

GScale network [9] comprised of 12 nodes and 19 links (at 2013, they have 15 datacenters as of 2016 [4]) is a private network that connects Google data centers. We used the same topology to evaluate DCRoute as well as other allocation schemes. Figure 4 shows the rejection rate of different techniques for different arrival rates from low load ($\lambda = 1$) to high load ($\lambda = 15$). We have included the schemes that potentially multiplex traffic over multiple paths just to provide a lower bound. Comparing with PMC and SPMC schemes over all arrival rates, DCRoute performs $< 2\%$ worse than the one with minimum rejection rate. Also, compared to all schemes, DCRoute rejects at most 4% more traffic.

Figure 4 shows the relative time to process a request using different schemes. This time is calculated dividing the total time to allocate/adjust all requests over all timeslots by the total number of requests. DCRoute is about 3 orders of magnitude faster than either PMC or SPMC. It should be noted that the rate at which time complexity grows drops as we move towards higher arrival rates since there is less capacity available for new requests and many arriving requests get rejected by failing simple capacity constraint checks.

B. Variable Network Size

We simulated different methods against four networks from 5 to 20 nodes:

$(N, M) \in \{(5, 7), (10, 17), (15, 27), (20, 37)\}$

In our topology, each node was connected to 3 or 4 other nodes at most 2 hops away. The arrival rate was kept constant at $\lambda = 6.0$ for all cases.

Figure 5 shows the rejection rate of different schemes for different network sizes. As network size increases, since λ is kept constant, the total cpacity of network increases compared to the total demand of requests. As a result, for a scheme that multiplexes request traffic over different paths, we expect to see decrease in rejection rate. For the K-Shortest Paths case with $K \in \{1, 3\}$ we see increase in rejection rate which we think is because these schemes cannot multiplex packets that much. Increasing the network size for these cases can cause more requests to have common links as the network is sparsely connected and create more bottlenecks resulting in a higher rejection rate.

PMC has a high rejection rate for small networks since choosing the minimum cost path might result in selecting longer (more hops) paths that create larger number of bottlenecks due to collision with other requests. Increasing network size, there are more paths to choose from and that results in less bottlenecks and therefore less rejection rate. In contrast, SPMC enforces the selection of paths with smaller number of hops resulting in lower rejection rates for small networks (due

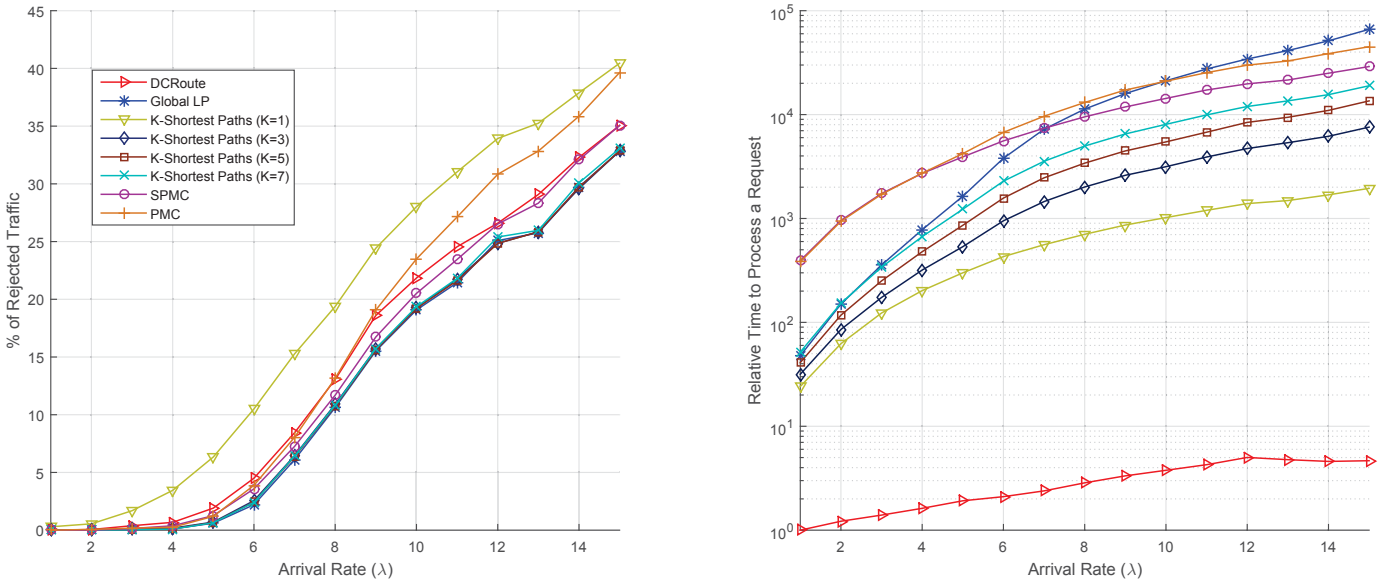

Fig. 4. Total % of rejected traffic and relative request processing time for GScale network with 12 nodes and 19 links

to request paths colliding less) and more rejections as network grows due to less diversity of chosen paths.

Compared to these two approaches, DCRoute balances the choice between smaller and longer paths. The assigned path has the least sum of load on the entire path and the least bottleneck load among all such paths. Paths with heavily loaded links and unnecessarily larger number of hops are avoided. As a result, rejection rate compared to *min(PMC, SPMC)* is relatively small ($< 3\%$) for all network sizes. Also, as Figure 5 shows, similar to previous simulation, DCRoute is almost three orders of magnitude faster than PIP schemes and more than 200 times faster than all considered schemes.

C. Effect of Timeslot Length

As mentioned earlier, the timeline is divided into timeslots. In this paper, we do not discuss the exact duration of timeslots, however we would like to see how smaller timeslots affect the amount of rejected traffic as well as the speed of different methods. The transmission rate of transfers only changes when moving from one timeslot to the next. Having timeslots that last longer than most requests would essentially result in a fixed transmission rate for such transfers. This causes less critical transfers (with a later deadline) to have to share the bandwidth with more critical transfers (with a closer deadline), reducing the flexibility of the system and increasing the number of rejected transfers. Having very short timeslots on the other hand can result in unnecessary scheduling overhead and practical complications. For example, rate-limiters at the end hosts may not be able to converge to specified rate if they have to change rate too often. Therefore, a proper timeslot length could be a few times smaller than most requests but large enough to avoid unnecessary processing overhead.

As shown in Figure 6, we divide every timeslot into 2 to 5 smaller timeslots. We again use GScale topology and consider $(\lambda = 6)$. We only considered the K-Shortest Paths techniques with $K \in \{3, 7\}$ since during previous tests they provided the best utilization at the least processing cost. It appears that if the timeslots are short enough, making them shorter does not improve the load accepted into the system: DCRoute rejects 2% more traffic compared to K-Shortest Paths schemes for all timeslot resolutions. Although DCRoute is always more than 2 orders of magnitude faster than K-Shortest Paths schemes, it does have a slightly higher growth rate in processing time as the timeslot resolution increases.

D. Discussion

We do not directly compare our scheme with any of the previous schemes, such as AMOEBA or TEMPUS, since we pursue the following two objectives together which is not the case for previous work:

- Avoiding any packet reordering
- Guaranteeing deadlines for admitted transfers

However, since AMOEBA is the most similar to DCRoute, we would like to provide an approximate comparison. We argue that AMOEBA is essentially an extension to the K-Shortest Paths LP technique $(K = 10)$ introduced here. By intelligently avoiding LP modeling for some special cases, AMOEBA provides a speedup of 30 times compared to 10-Shortest Paths LP for $(\lambda = 8)$ [6]. For the same arrival rate and request volume/deadline distribution, DCRoute performs more than 1000 times faster than 10-Shortest Paths LP technique. Also in a previous work called RCD [13], we showed how the close to deadline scheduling technique can speed up the traffic allocation by up to 15 times compared to AMOEBA while resulting in same utilization over a single link. DCRoute is based on RCD coupled with a path selection heuristic that eliminates the need for LP modeling.

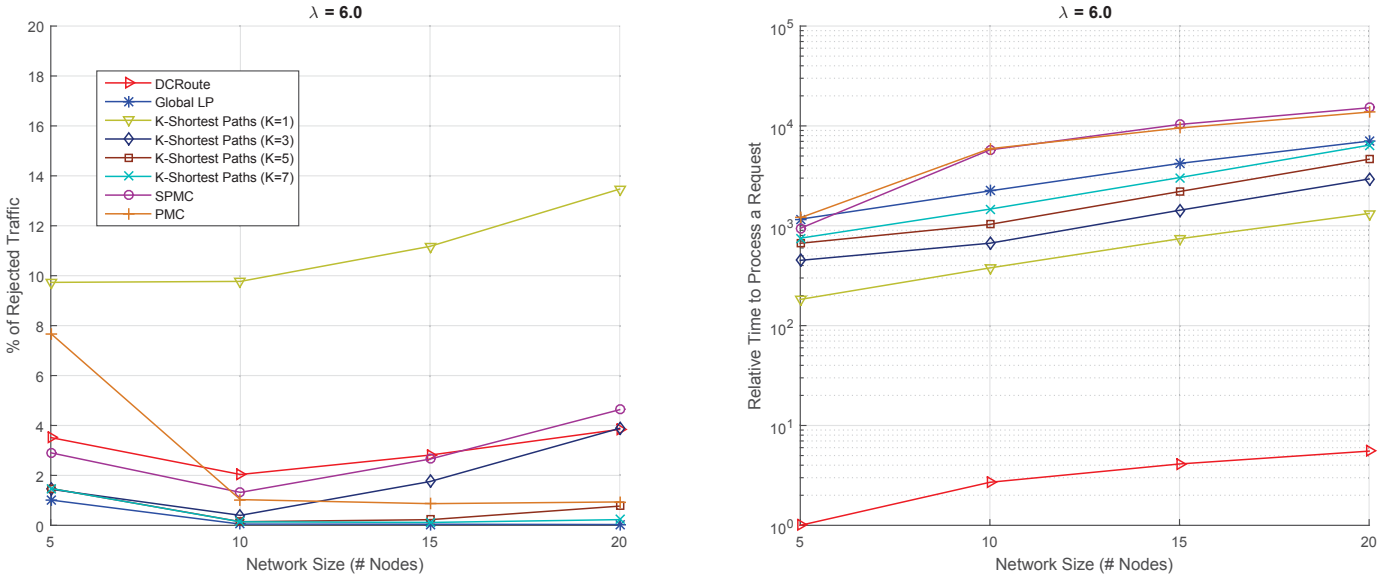

Fig. 5. Total % of rejected traffic and relative request processing time for different network sizes

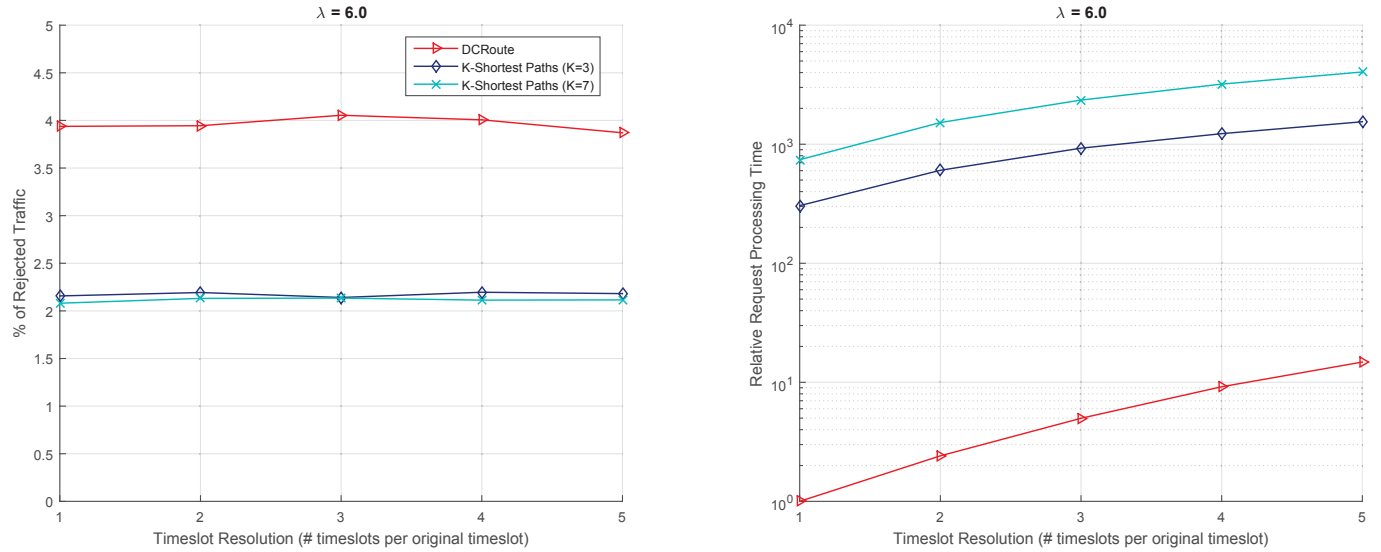

Fig. 6. Total % of rejected traffic and relative request processing time as timeslot resolution increases

V. Conclusions and Future Work

In this paper, we proposed DCRoute, a routing algorithm for Inter-Datacenter networks which guarantees that transfers complete before their deadlines and to avoid reordering, schedules all packets of a request on the same path. Inspired by the fact that allocating ALAP allows for new requests to be scheduled considering only residual bandwidth, DCRoute performs much faster than schemes based on LP modeling. It is more than 2 orders of magnitude faster than all simulated schemes and 3 orders of magnitude faster than schemes that do not multiplex transfer packets on multiple paths.

We showed that DCRoute admits at most 4% less traffic compared to schemes that multiplex packets on multiple paths (which provide an estimate of lower bound on rejected traffic) and 2% less traffic compared to schemes that schedule all packets of each transfer on the same path.

Finally, we studied the effect of timeslot resolution and found out that making timeslots smaller than necessary does not increase the admitted load and only incurs extra processing costs. In this paper, we evaluated DCRoute with synthetic traffic. Evaluation of DCRoute using real inter-datacenter traffic is suggested as a future work. In addition, studying the effects of link failures and developing methods to properly handle them can be a subject for future work.

Acknowledgement

We would like to thank the anonymous reviewers of HiPC whose suggestions and comments helped us improve the quality of this paper.

REFERENCES

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[2] Amazon web services (aws) - cloud computing services. https://aws.amazon.com/.

[3] Microsoft azure: Cloud computing platform & services. https://azure.microsoft.com/.

[4] Compute engine - iaas - google cloud platform. https://cloud.google.com/compute/.

[5] Ravi Kiran R Poluri, Samir V Shah, Rui Chen, and Lin Huang. Datacenter synchronization, October 16 2012. US Patent 8,291,036.

[6] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing deadlines for inter-datacenter transfers. In *Proceedings of the Tenth European Conference on Computer Systems*, page 20. ACM, 2015.

[7] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. *ACM SIGCOMM Computer Communication Review*, 44(4):515–526, 2015.

[8] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 15–26, New York, NY, USA, 2013. ACM.

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

[10] M. Laor and L. Gendel. The effect of packet reordering in a backbone link on application throughput. *IEEE Network*, 16(5):28–36, Sep 2002.

[11] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 29–29. USENIX Association, 2012.

[12] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. Juggler: a practical reordering resilient network stack for datacenters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 20. ACM, 2016.

[13] M. Noormohammadpour, C. S. Raghavendra, S. Rao, and A. M. Madni. Rcd: Rapid close to deadline scheduling for datacenter networks. In *World Automation Congress (WAC)*, pages 1–6. IEEE, 2016.

[14] K.Y. Li and R.J. Willis. An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research*, 56(3):370 – 379, 1992.

[15] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. Tcp extensions for multipath operation with multiple addresses, 2013.

[16] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016.