Calendaring for Wide Area Networks

Srikanth Kandula Ishai Menache Roy Schwartz Spandana Raj Babbula

Microsoft

Abstract– Datacenter WAN traffic consists of high priority transfers that have to be carried as soon as they arrive, alongside large transfers with preassigned *deadlines* on their completion. The ability to offer guarantees to large transfers is crucial for business needs and impacts overall cost-of-business. State-of-the-art traffic engineering solutions only consider the current time epoch or minimize maximum utilization and hence cannot provide pre-facto *promises* to long-lived transfers. We present TEMPUS, an online temporal planning scheme that appropriately *packs* long-running transfers across network paths and future timesteps, while leaving capacity slack for future changes. TEMPUS builds on a tailored approximate solution to a mixed packing-covering linear program, which is parallelizable and scales well in both running time and memory usage. Consequently, TEMPUS can quickly and effectively update the promised future flow allocation when new transfers arrive or unexpected changes happen. Our experiments on traces from a large production WAN show, TEMPUS can offer and keep promises to long-lived transfers well in advance of their actual deadlines; the promise on minimal transfer size is comparable with an *offline* optimal solution and outperforms state-of-the-art solutions by 2-3X.

Categories and Subject Descriptors

C.2.1 [Computer-Communication Networks]: Network Architecture and Design

Keywords

Inter-datacenter; wide area networks; software-defined networking; deadlines; mixed packing covering

1. INTRODUCTION

Calendaring, colloquially, refers to setting aside future resources so that long term objectives are met in spite of short-term demands for resources.

Several large cloud companies operate geo-distributed datacenters connected through a Wide Area Network (WAN). The WANs carry traffic on the order of terabits/sec; are expensive and business critical [11, 13].

Traffic on the inter-datacenter WAN can be roughly characterized as a mix of two types. The first, called *highpri* traffic, comprises of instantaneously arriving demands due to customer facing traffic. These demands are somewhat unpredictable but are in general much smaller than the total capacity and need to be fully met. The second comprises of large transfers between datacenters that require WAN capacity over extended periods of time (see, e.g., [11, 13]). These make up the bulk of traffic on the WAN. Examples include moving a new search index generated at one datacenter to all the other distribution datacenters or moving datasets collected at one datacenter for later analysis on a big data cluster at another datacenter.

Long-running transfers are often time-critical. Delaying them may directly impact service quality and customer revenue. For example, delays in transferring the search index reduce response quality. Delays in onboarding datasets into Azure or AWS reduce revenue. In addition, transfer delays increase cost by wasting other resources, e.g., servers and developers are idle waiting for datasets to arrive. The time-sensitivity of the long-running transfers can be modeled in the form of a *deadline*. Some deadlines are hard, i.e., transfers are useless if late; however, many are "soft"; the transfer still needs to be completed however its "value" reduces past the deadline.

Ideally, a network provider should serve both types of traffic well. That is, the network should at all times be able to support the instantaneous demands. In addition, enough resources should be assigned to the longer term traffic so that these transfers have a predictable completion time. The objective of this paper is to design a solution to the calendaring problem, with the following characteristics:

1. No delays and zero loss rate for high-priority traffic.
2. When not limited by network capacity, long-running requests are fully met before the deadline.
3. When demands exceed network capacity, continue to offer guarantees such as maximizing the minimal fraction of transfers that finish before deadline (fairness), or maximizing a specified total utility function.

Satisfying all these goals is hard. Prior TE schemes minimize the maximum link utilization; some are dynamic [14] whereas others can be oblivious to the actual demands and network topology [1, 9]. With inter datacenter WANs, maximizing return on the investment requires operating at nearly full utilization; hence prior work does not apply. More recently, TE schemes have been proposed for inter-datacenter WANs [11, 13]. While they do operate at high utilizations, for reasons that we will explain below, these schemes only plan for a single future timestep (e.g., five minutes). Any transfers that last longer receive no guarantees. Further, such one-timestep TE has poor practical performance since (a) it cannot differentiate which of the large transfers to serve before the others, and along which network paths and (b) a current high transfer rate does not guarantee high transfer rates in the future. Scheduling transfers in earliest deadline first (EDF) order is also far from optimal primarily because it does not carefully spread transfers across the network. By scheduling strictly on deadline order, EDF can miss opportunities to simultaneously schedule non overlapping transfers. Also, EDF is unfair and performs poorly when all of the deadlines cannot be met.

To understand why calendaring is hard, consider the simpler *offline* scenario in which all requests and their requirements are known ahead of time. Satisfying all of the demands before the corresponding deadlines, or getting as close as possible to that goal,

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.
Copyright 2014 ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626336.

requires optimally spreading traffic not just along all the available network paths but also along future times. This optimization problem can be formulated as a linear program (LP) but the size, as we will show, is typically huge (millions of variables and constraints).

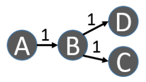
The *online* version is harder still. Clearly the problem evolves with time; new long-running requests may arrive; links may fail or the volume of high-priority traffic can change. So the (offline) optimization problem described above has to now *run repeatedly*, e.g., once every few minutes or whenever substantial changes happen. This increases the computational cost. Worse, the cumulative effect of running a sequence of optimizations, each of which has only a limited knowledge of the future, can be very poor compared to the offline oracle solution above. For example, requests arriving later will be treated unfairly if all future network capacity has been promised to prior requests. Also, by not being able to *jointly optimize* over requests that arrive at different times, the solution quality will be poor (as we show later). Reneging on promises made to prior requests does help improve solution quality, i.e., more requests can be finished before deadline, but doing so significantly reduces the value of calendaring; no promises are "for sure" and hence, programmer's and customer's overall plans are disrupted.

Calendaring requires *online temporal* planning; *temporal* implies network resource has to be allocated far into the future to offer and meet guarantees to long-running transfers; *online* implies having to do this with imperfect future knowledge. To the best of our knowledge, no prior solution exists to this problem.

We present TEMPUS, a solution to the calendaring problem that meets these goals. Flows are allocated far into the future. Traffic is spread across network paths and time so as to maximize the minimal fraction of requests served before deadline as well as to maximize a utility function overall. Current network capacity is fully allocated. Future network capacity is systematically under-allocated so as to (a) offer high promises to requests that are currently in the system and (b) leave room open for future arrivals and other changes.

We achieve all of the above with an appropriate sequence of linear programs to be solved at each timestep. TEMPUS uses our novel mixed-packing covering solvers, which have the following distinctive features: (a) they have smaller memory footprint compared to running an LP solver at each timestep, and perhaps more importantly, (b) they conserve computation. That is, since the cumulative effect of the online optimization across all timesteps allocates only as much network flow as the offline LP, the total computational cost should be similar; i.e., the running time when added up across all timesteps equals the runntime of the offline LP; and hence the per timestep running time is much smaller. Our main algorithmic contribution is in showing how the packing-covering solver at each timestep proceeds by editing the flow allocation from the previous timestep rather than having to begin from scratch. In general, this is harder to achieve in an LP solver. Finally, we also show how to parallelize our mixed packing-covering solver.

We evaluate TEMPUS by measuring its ability to schedule large transfers that resemble those observed on a production WAN across a wide range of load factors. Compared to the offline optimal scheme, we see that TEMPUS achieves nearly the same minimum guarantee on transfer satisfaction, and nearly finishes the same fraction of flows before their deadlines. Along both these metrics, TEMPUS improves on top of the best performing variants of greedy (one timestep) schemes. Further, TEMPUS offers a promise to each transfer that is nearly the best possible as soon as the transfer arrives or soon after. Greedy heuristics, in contrast, can offer no guarantees beyond the fraction that has already been served. TEMPUS's parallelized version appears comparable to top-of-the-shelf LP solver on running time.



a: arrival, D: demand, d: deadline

Request	a	D	d
R_1: A → B	1	4	6
R_2: A → C	1	2	8
R_3: A → D	1	2	7
R_4: B → C	4	5	9

	R_1	R_2	R_3	R_4
Offline	[2,5]	1,8	[6,7]	[4,7], 9
EDF	[1,4]	[7,8]	[5,6]	[4,6][9,10]
Greedy	[1,8]	[1,6]	[1,6]	[4,9]
Online	[1,4]	[7,8]	[5,6]	[4,6][9,10]
Tempus	[1,6]	[1,8]	[1,7]	[4,9]

Figure 1: An example illustrating online temporal planning. On the left are the network and request requirements. On the right are the schedules output by various schemes; some schedules have multiple requests active at the same time on the same edge, assume then that each request gets a fair share. The shares due to TEMPUS are uneven as we describe later.

2. INTER-DC CALENDARING MODEL

In this section we describe characteristics of inter-datacenter traffic, motivate the need for online temporal planning and observe some key requirements for a calendaring solution.

2.1 Inter-DC WAN Traffic

Heterogeneous traffic types/ deadlines. As described earlier, traffic on the WAN is a mix of high priority traffic and large long-running transfers. The former has to be served fully with no delays whereas the latter can be served more flexibly. We use the term *request* to denote an application-desired transfer that the logically centralized traffic engineering algorithm has to manage (as in [11, 13]). Source and destination for requests are typically groups of servers (10s-1000s) in different datacenters. Consequently, requests can achieve high cumulative rate. The request specifies a begin time, the bytes to move within a deadline, optionally a set of paths through the network that the request's traffic should be sent over and optionally a peak traffic rate. We assume that the scheduling epochs are much coarser than TCP time-scales, (e.g., minutes vs. RTT of 100ms). Typical request sizes range from tens of terabytes to petabytes; deadlines range from an hour to a couple days.

Need to highly utilize the WAN and to offer service guarantees to traffic. The WAN between datacenters is an expensive resource; one study estimates amortized annual cost of 100s of millions of dollars [11]. Hence, fully utilizing the WAN maximizes return on cost. Prior work on managing inter-DC traffic designs an application-network interface to learn application requirements and runs an optimal traffic allocation per timestep. The resultant allowed rates are conveyed to each app and desired routes programmed into the network [11, 13]. TEMPUS uses a similar architecture. However, prior work either allocates traffic one time-step at a time [11, 13] or minimizes the maximum link utilization [1, 9, 14, 22] and cannot offer deadline SLAs to transfers.

Estimating volume of high-priority traffic: We observe that source servers can mark traffic type (e.g., using the IP ToS field). Our analysis of DC-WAN traces (taken from Microsoft's inter-DC WAN) indicates that high-priority traffic is estimatable: the per DC-pair time series of the volume of high priority traffic, over a period of a day exhibits a coefficient-of-variation, ranging from 0.06 to 0.48. A scheduling framework can plan based on the statistical averages of highpri demand but has to be robust to their variations.

2.2 Guidelines for Online Temporal Planning

In view of the above observations, there is a crucial need to spread the long-running requests over network paths and time so as to maximize deadline satisfaction as well as network utilization. However, such planning should be able to incorporate future request arrivals as well as the variability in the volume of high-priority traffic. Here, we observe some guidelines for such a solution.

The "offline" calendaring problem can be formulated as a linear program (LP). When the details of all requests and the precise volume of highpri transfers are known in advance, calendaring can be written as an LP. The details are deferred to §3. Table 1 lists the sizes of this LP. We note that a typical problem consists of networks with hundred datacenter sites, thousand links (m), tens of thousands of requests (R), i.e., a handful of requests per datacenter pair, and hundreds of time units (T) (e.g., corresponding to a five minute resolution, where the overall time horizon is a single day). The LP then has millions of variables and constraints. Such large LPs can in principle by solved by state-of-the-art LP solvers running on powerful machines but the space and running-time complexities are substantial. Here, we consider whether simpler *combinatorial* solutions exist for the core problem.

Planning in the midst of dynamically evolving conditions. Certainly, the calendaring problem evolves: new long-running requests arrive, the network can change etc. Simply re-running the above offline optimization problem, whenever a change happens or a short period later is computationally challenging. Worse, if the optimizations at each timestep run with no *context* of the others, the resultant allocations can be extremely poor.

We use a simple example to illustrate the important aspects that an online optimizer has to get right so as to approximate the offline oracle in spite of operating at each timestep without future knowledge. Figure 1 depicts a simple network and four requests. When requests are known a-priori, i.e., the offline case, observe that there exists a schedule that finishes all requests within their deadlines.

Let us first consider online strategies that look one timestep at a time. Earliest deadline first (*EDF*), at each timestep, schedules the requests in order of their deadlines. EDF does not finish R_4 before its deadline. EDF's key weakness is that picking requests strictly on deadline order prevents it from finding beneficial schedules that simultaneously schedule requests on different parts of the network (e.g., schedule R_2 early so that R_1 and R_4 can run simultaneously). Observe that in the above example, each request has exactly one usable path; EDF would have a harder time when traffic has to be spread across multiple paths. *Greedy* corresponds to schedulers proposed by prior work [11] that at each timestep allocate traffic evenly and are work-conserving; it cannot finish R_1 on time. The primary weakness here is the inability to plan into the future; fair share at each timestep does not translate to meeting deadlines, esp: if the deadlines are short. *Online* corresponds to a scheduler that does plan into the future for all the requests that it is currently aware of; it cannot finish R_4 on time. The primary weakness is that the schedule chosen at $t = 1$ offers promises to the first three requests without knowing what would arrive in the future; hence, very likely it picks a schedule that is not compatible. Re-scheduling at $t = 4$ does not help either because flow allocations in the past cannot be revisited. Reneging on promises can help the online scheduler meet deadlines of new requests but they counteract the value of calendaring. Tempus, as we show later is able to meet all deadlines. Tempus only allocates a small fraction of the capacity of edges in future timesteps. Since Tempus plans well into the future and attempts to finish as many requests as possible before their deadline, requests R_1, R_2, R_3 are "safe" with Tempus because there is enough capacity to finish them all before their deadlines even after under-allocating future edges. Further, due to the under-allocation Tempus retains enough capacity on future edges, e.g., the $B \to C$ edge for timesteps after 4, so as to finish request R_4.

Promises. As we saw above, a good calendaring solution has these properties: (a) offer to each request a promise on the fraction that will be satisfied before deadline; it is better to offer large promises early in the request lifetime; (b) do not renege on promises; (c) ensure promises are fair (or proportionally fair) in spite of online changes and (d) maintain high network utilization. In essence, the cumulative effect of the online optimizations should mimic closely the offline (impractical) scheme. Simply repeating the offline optimization at each timestep does not offer these properties.

Scheme	Num. Equations	Num. Variables
Optimal Offline (flow)	$\approx n\mathrm{RT} + m\mathrm{T} + \mathrm{R}$	$m\mathrm{RT}$
Optimal Offline (path)	$\approx m\mathrm{T} + \mathrm{R}$	PRT
Online LP	above, at each timestep	
Greedy [11, 13]	per timestep, above with T set to 1	

Table 1: Estimating the size of LPs to be solved for the calendaring problem; for a network graph with n nodes, m edges; planning T timesteps into the future; and R requests. Optionally, each request can require to be routed only along given paths (say P per request).

Conservation of computation. Notice that the online optimizations done at each timestep allocate all together a similar amount of flow as the single offline LP. Hence, it is natural to ask whether the total computation cost of the online optimizations can be about the same as the cost of running the single offline LP. For typical LP solvers, this does not hold; in fact, the cost of the optimization in each timestep can be as much as that of the offline LP.

In the rest of this paper, we will describe Tempus, an online temporal planner, that satisfies both the above requirements on solution quality and computation cost.

3. OVERVIEW OF TEMPUS

The goal of Tempus is to accommodate the long-term requests, while leaving enough capacity for ad-hoc high-priority requests. The system uses available information on long-running requests to efficiently *pack* these requests over time and network paths. Our goal here is to identify some key aspects, we defer the details to §4.

Preliminaries. The input for our problem consists of a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$, and non-negative edge capacities $c : E \to \mathcal{R}^+$. Each long-term request i with strict deadline is defined by the tuple $(a_i, b_i, d_i, D_i, s_i, t_i, \mathcal{P}_i)$ where: a_i is the *aware time* in which Tempus becomes aware of the request; b_i is the *begin time* from which the request can be scheduled; d_i is the *deadline* after which the request cannot be scheduled anymore; D_i is the *demand* of the request; s_i is the *source* node the request needs to be routed from; t_i is the *target* node the request needs to be routed to; $\mathcal{P}_i$ is the collection of admissible paths from s_i to t_i of the request. Requests with soft deadlines are defined similarly, with a value function replacing the strict deadline d_i; the details are deferred to §4. Additionally, the algorithm has an estimate $\chi_{e,t}$ for the fraction of the capacity c_e of edge e that will be needed by the high priority requests at time t; only the remaining $(1 - \chi_{e,t})$ fraction can be allocated to long-term requests.

3.1 Offline version

Formulating offline calendaring as an optimization problem: Tempus uses variables $f_{i,p,t}$ to denote the amount of flow allocated for request i on path $p \in \mathcal{P}_i$ in time t. Using these variables, linear inequalities can be formulated to assert the allocation's feasibility: in every time and for every edge the total flow allocated on it does not exceed the available capacity. Further the traffic for request i, can only be routed when it is available $t \in [b_i, d_i]$ and totals no more than demand D_i (and is at least $\alpha_i D_i$). Goals include maximizing the minimum fraction of request traffic routed before deadline ($\max \min_i \alpha_i$). The offline problem assumes perfect future knowledge of all requests and hence is not a practical solution.

Adding utility. While maximizing the smallest fraction α_i corresponds to a *fairness* objective, some parts of the network might remain under-utilized, i.e., more flow can be allocated but the request with the least α cannot be improved further. To fully use the network, we introduce a supplementary utility function, e.g., $\delta = \text{avg}_i \alpha_i$. We defer the precise definition of utility to later (§4), however, we note that several forms of utility can be specified as linear functions. After maximizing α, we execute a second optimization problem to maximize the total utility.

Note that Tempus does not directly maximize how many transfers finish before their deadline. The *all-or-nothing* nature of the metric makes it hard to optimize; e.g., it cannot be captured with a linear constraint. However, maximizing α and then utility, as Tempus does, is a close approximation. In particular, when all deadlines are satisfiable Tempus will find such a flow allocation. Further, due to utility maximization, Tempus will finish some transfers before their deadline even when it cannot finish all transfers.

3.2 Online temporal planning

The main challenge is to solve the above optimization problem in an online manner, since new long-term requests can arrive at any time and other changes may happen. As discussed before, repeatedly applying the above offline optimization problem is computationally expensive because it has to plan far into the future. Further, it can lead to a substantially worse solution. Here, we discuss how to solve the latter; speeding-up the computation is in §3.3.

Rerunning the above optimization problem at each timestep treats requests that arrive later unfairly. Maximizing the minimum α_i and then, maximizing the total utility, can use up all of the available capacity on edges causing later requests to receive a very small α. As we saw before, it also reduces how many requests finish before their deadlines because it cannot jointly optimize over all requests.

To retain fairness and improve solution quality, Tempus **systematically under-allocates future edges**. When planning at time τ, the fraction of edge capacity that is available for allocation at times $t > \tau$ is denoted by $\beta_{e,t,\tau}$. We choose β to be a function that decreases with $t - \tau$. Intuitively, the further an edge is into the future the less its capacity can be used up during the current timestep. At each consecutive timestep, more of the capacity becomes available ensuring that later arriving requests can be allocated on that {edge, time}. This helps with fairness. Solution quality also improves because Tempus allocates more capacity on timesteps in the near future for which it understands the requirements more completely. Due to the progressive release of capacity, Tempus's promise to long-term requests α_i increases with time; Tempus aims to offer large promises early in the lifetime of the request.

Another concern is that whereas capacity can be set aside on future edges for highpri traffic ($\chi_{e,t}$), the actual highpri traffic may require more or less capacity. If more is needed, some flow that has been promised to long-term requests has to be lowered causing promises to be reneged. Tempus **corrects the solution for actual usages of high-priority requests.** Edges that are under-used by highpri traffic need no special consideration since Tempus uses the spare capacity to increase the flow for requests that are not fully satisfied. To correct over-usage, Tempus reduces the pre-promised flows $f_{i,p,\tau}$ of a carefully chosen set of long-term requests such that (a) the total flow on every edge remains below capacity, while (b) having minimal effect on promises α_i as well as the total utility. We use an LP for this correction; the LP is small since it only considers the current timestep τ. Intuitively, the LP chooses requests that have later deadlines or other less constrained paths, and hence decreases in $f_{i,p,\tau}$ can be made up for at other times or along other paths.

Observe that link and switch failures can also be handled similarly because a link failure can be mimicked as if a highpri request with demand equal to link capacity appears for that timestep.

A side-effect of the above is that Tempus can renege upon a promise. In practice, however, we see that this occurs rarely; because Tempus preferentially lowers the flows of requests that can be made up for later. Further, when Tempus reneges, we find that the fraction served by Tempus is close to the fraction served by the optimal offline scheme (which has perfect future knowledge).

Lengthy requests, i.e., with a large $d_i - a_i$, will slow down any temporal planner by forcing each timestep to plan far into the future (up to max d_i time). To ensure a fast solution, Tempus optionally employs a **sliding window** approach; i.e., it only plans for a fixed number of time steps into the future. Requests whose deadline lies beyond the sliding window's length are *broken* into a smaller request and their demand is scaled accordingly.

3.3 Approximate packing-covering solvers

We have a **choice of solvers**; whereas the overall framework described so far leads to a novel solution for the online temporal planning problem, how the optimization at each timestep is solved crucially impacts the practicality of the solution. We offer a brief background on iterative solvers for mixed packing covering problems and then comment on how Tempus extends them for calendaring.

Background on Packing-Covering Solvers: Given a variable vector $\mathbf{x}$, a *packing* matrix $\mathbf{P}$ and a *covering* matrix $\mathbf{C}$, a mixed packing covering problem finds a feasible solution for these inequalities:

$$\{\mathbf{Px} \leq 1, \;\; \mathbf{Cx} \geq 1, \;\; \mathbf{x} \geq 0\}, \tag{1}$$

where $\mathbf{x} \in \mathcal{R}_n$, $\mathbf{P} \in \mathcal{R}^+_{m \times n}$, $\mathbf{C} \in \mathcal{R}^+_{\ell \times n}$; $\mathcal{R}^+$ indicates that all entries in $\mathbf{P}$ and $\mathbf{C}$ are non-negative.

It is easy to see that any mixed packing covering problem is a linear program (LP). While it can be solved using typical LP solvers, its special structure allows for iterative solutions that have some useful properties. Intuitively, a common approach in the design of iterative solvers is the following. First, let $\mathbf{x} = 0$ be the starting zero solution; all packing constraints are trivially satisfied but none of the covering constraints are met. Second, inductively, suppose there exists an increment $\Delta\mathbf{x}$ to the variables that satisfies:

$$\max \mathbf{P}(\mathbf{x} + \Delta\mathbf{x}) - \max \mathbf{Px} \leq \min \mathbf{C}(\mathbf{x} + \Delta\mathbf{x}) - \min \mathbf{Cx}. \tag{2}$$

Here, max (min) is taken over the m (l) packing (covering) constraints. In words, the above equation ensures that the smallest covering constraint improves by more than the largest packing constraint. Incrementing the variables by such a $\Delta\mathbf{x}$ is helpful since it moves the solution closer towards satisfying all covering constraints while giving up a smaller amount on the packing constraints. Finally, observe that repeating the above step until all covering constraints are satisfied would yield a feasible solution.

Translating the above intuition into a solution is challenging because (a) it is not apriori clear how to search for such a $\Delta\mathbf{x}$ and (b) more importantly, it is unclear that a solution for every feasible mixed packing-covering problem could be found by summing up increments that satisfy (2).

Young's method [24], among several other methods, adopts the above approach. First, since min and max are not smooth functions, Young replaces them with:

$$\min_{1 \leq i \leq n} \{\mathbf{x}_i\} \geq lmin(\mathbf{x}) \triangleq -\ln\left(\sum_{i=1}^{n} e^{-\mathbf{x}_i}\right),$$

$$\max_{1 \leq i \leq n} \{\mathbf{x}_i\} \leq lmax(\mathbf{x}) \triangleq \ln\left(\sum_{i=1}^{n} e^{\mathbf{x}_i}\right).$$

Second, Young restricts the increment per iteration to a small amount and to one variable $\mathbf{x}_i$ in $\mathbf{x}$. Then, condition (2) becomes:

$$\frac{\partial lmax\,(\mathbf{Px})}{\partial \mathbf{x}_i} \leq \frac{\partial lmin\,(\mathbf{Cx})}{\partial \mathbf{x}_i}\,. \tag{3}$$

In words, for variable $\mathbf{x}_i$, the rate of change of the maximum over packing constraints should be smaller than the rate of change of the minimum over covering constraints. Equipped with this new condition, Young [24] proved that if the packing-covering problem is feasible, then for any current value assignment to variables $\mathbf{x}$ there is a variable $\mathbf{x}_i$ that satisfies (3). The solver starts with $\mathbf{x} = 0$, repeatedly makes increments satisfying (3) and terminates when no such increment can be found or when all covering constraints are met.

Extending mixed packing covering solvers to online temporal planning: Here, we mention how iterative solvers for mixed packing covering problems are more suitable for the calendaring problem and show how Tempus extends the theoretical work.

For the online step in the calendaring problem, we observe that the solution to the mixed packing-covering problem in a timestep can be *edited* into a feasible starting point for the problem of the next timestep. Observe that edge capacity translates to a packing constraint, request satisfaction translates to a covering constraint and that the variable vector $\mathbf{x}$ contains flow allocations. To wit, in the online step, the constraints change as follows (precise details are in §4.2): for each newly arriving request and {edge, time} that is to be newly considered, new constraints on their respective deadline satisfaction and capacity are added. Some existing constraints are edited, for example, to release more capacity for allocation. The solution to the previous packing-covering problem is expanded by adding flow variables for newly arriving requests and on new edges and setting them to zero. Crucially, observe that none of the packing constraints are violated (new edges have flow 0 and every old edge has strictly larger capacity to allocate). The old covering constraints are unchanged whereas the new covering constraints are unmet (flow for new requests is 0). Hence, the old solution is a feasible searching point for the new problem. Further, as we show later by extending Young's proof that if the new problem is feasible, increments that satisfy condition (3) will continue to exist.

This is incredibly useful because: (a) all of the previously promised allocations are retained and (b) none of the work that has been done by the previous steps, in terms of allocating flow, needs to be redone which speeds-up the computation as we prove later.

We also observe that solving calendaring with a mixed packing-covering solver requires less memory. Table 1 describes the problem size for different approaches. Notice that in the calendaring problem, the most numerous are the flow variables; per request, per time and per path (or per edge) they record how much flow has been allocated. We assert that the iterative solver above need not maintain all of the flow variables. In fact it suffices to keep per-packing and covering constraint (edge and request), the current state of $\mathbf{Px}$ or $\mathbf{Cx}$ respectively, i.e., available capacity and unmet demand. Upon allocating some flow along a path to a request, the effect to these constraints is recorded. The precise flow variables are needed only for the current time instance, so as to push appropriate routes into the network, and can be computed by solving a small LP for the current timestep that takes as input the total that has been allocated to requests in that timestep. Consequently, the memory overhead reduces by at least one order of magnitude (see §5.1 and §6.3).

Similarly to [24], we note that the iterative solver can be parallelized since typically many feasible increments $\Delta\mathbf{x}$ exist. Young describes how to parallelize the search process. However, we observe some additional aspects of our problem that make searching for a feasible increment much faster (§5.3). Consequently, the bottleneck shifts to applying the increment for which we design a novel parallelization technique (§5.2).

Finally, we note that a packing-covering solver only outputs an *approximate* solution to a *feasibility* problem. They guarantee that: (a) if the packing-covering equations are infeasible, the solver would report infeasibility; (b) if the packing-covering equations are feasible, the solver reports feasibility but the solution can violate the packing constraints by a small amount. We extend Tempus to cope with the imprecision and yet obtain the desired objective by solving a small number of feasibility problems (§4.2).

3.4 Takeaways

We have described how to design appropriate optimization problems to solve at each timestep such that their cumulative effect avoids unfairness and provides a good quality solution; close to that from the offline oracle (§3.2). The per-timestep optimization can also be posed as a packing-covering problem. And, we develop a new *online* version of Young's algorithm to solve sequences of packing-covering problems (§3.3). This solution has some nice properties compared to a general LP solver. We emphasize that unlike greedy approaches (e.g., [11]), our online algorithm schedules flows far in the future, and can thus provide *promises* α_i on the fraction of the demand that each request i would be able to transfer. Since inter-DC WANs operate at nearly full utilization, prior efforts to minimize network congestion do not apply [1, 9, 14, 22]. Further, inter-DC WANs have less statistical multiplexing than ISP networks – few large requests contribute most of the bytes – and hence, deadline SLAs to large requests is an important part of the overall traffic management problem. We proceed to the details.

4. DESIGN OF TEMPUS

In this section, we present details of Tempus including provable guarantees on its runtime. For clarity, we first examine the offline case (§4.1) and then build our novel formulation for the online case (§4.2). In either case, we first describe the optimization problem, which can be solved using an LP, and then present the Tempus algorithm that builds on Young's method [24].

Recall from §3 that the input for our problem is a graph $G = (V, E)$ with edge capacities c_e; requests defined by their parameter tuple $(a_i, b_i, d_i, D_i, s_i, t_i, \mathcal{P}_i)$ (see §3 for definitions of parameters). Additionally, unless otherwise specified, we assume that there is an absolute size T that upper bounds all request deadlines.

4.1 (Impractical) Offline Case

In the offline case, note that the algorithm is aware of all (future) requests. Given the input graph G and all k requests, we first wish to find the largest α, $0 \leq \alpha \leq 1$, such that a fraction of at least α of the demand of each request can be routed before its deadline while respecting the capacity constraints. Maximizing α corresponds to a *fairness* objective. This is the well-known *Concurrent Multi-commodity Flow* problem. Suppose $f_{i,p,t}$ denotes the flow variable associated with request i at time t on path $p \in \mathcal{P}_i$ and $\mathbf{f}$ denotes the vector of all flow variables. Consider these quantities,

$$h_{e,t}(\mathbf{f}) \triangleq \sum_{i:b_i \leq t \leq d_i} \sum_{p \in \mathcal{P}_i : e \in p} f_{i,p,t} \qquad \text{(capacity usage)}$$

$$\ell_i(\mathbf{f}) \triangleq \sum_{t=b_i}^{d_i} \sum_{p \in \mathcal{P}_i} f_{i,p,t} \qquad \text{(satisfied demand)}$$

$$g(\mathbf{f}) \triangleq \sum_{i=1}^{k} \sum_{p \in \mathcal{P}_i} \sum_{t=b_i}^{d_i} u_{i,p,t} \cdot f_{i,p,t} \qquad \text{(utility)},$$

where $h_{e,t}(\mathbf{f})$ corresponds to the total flow on edge e at time t, $\ell_i(\mathbf{f})$ corresponds to the demand of request i satisfied by its deadline, and $g(\mathbf{f})$ corresponds to the value of the secondary utility function. Recall that we add a secondary *utility function* such that maximizing utility would increase the network utilization even when the request with the worst α cannot be further improved. We restrict utility to functions having $u_{i,p,t} \geq 0$ so that $g(\mathbf{f})$ would correspond to a covering constraint. We emphasize that $u_{i,p,t}$ are inputs to the solver and are *not* variables for the optimization problem. Note that the δ = average α mentioned earlier is a special case with $u_{i,p,t} = \frac{1}{k \cdot D_i}$.

We seek to maximize utility $g(\mathbf{f})$ among all the optimal solutions of the fairness objective (maximizing $\min_i \alpha_i$). If α and U are the target values for each of these goals, two LPs could be formulated to maximize each of them in sequence with the following constraints:

$$
\begin{aligned}
LP(\alpha, U) = \big\{ h_{e,t}(\mathbf{f}) &\leq c_e & \forall e \in E, \forall t \in [0, T], && \text{(capacity)}\\
\ell_i(\mathbf{f}) &\leq D_i & \forall 1 \leq i \leq k, && \text{(demand)}\\
\ell_i(\mathbf{f}) &\geq \alpha D_i & \forall 1 \leq i \leq k, && \text{(fairness)}\\
g(\mathbf{f}) &\geq U, & && \text{(utility)}\\
\mathbf{f} &\geq 0 \big\}
\end{aligned}
$$

4.1.1 Applying mixed packing-covering solvers

Observe that $LP(\alpha, U)$ is exactly in the form of (1), as we can normalize the constraints so that their right hand side is all 1s. Directly maximizing α and then U is not possible however since the iterative solver only checks the feasibility for given values of each. Hence, we start with $U = 0$ and conduct a search to find the largest α for which $LP(\alpha, 0)$ is feasible. The resulting α is then kept fixed and we search for the largest U for which $LP(\alpha, U)$ is feasible.

Next, we describe how to use Young's algorithm to determine given some values for α and U whether a flow vector $\mathbf{f}$ exists that satisfies $LP(\alpha, U)$? Figure 2 shows the pseudo-code for the algorithm. It follows the intuition described in §3.3. We use these internal variables, each corresponding to a constraint of $LP(\alpha, U)$:

$$
\begin{aligned}
y_{e,t} &\triangleq e^{\frac{N}{\varepsilon \cdot c_e} \cdot h_{e,t}(\mathbf{f})} && \text{(capacity)}\\
q_i &\triangleq e^{\frac{N}{\varepsilon \cdot D_i} \cdot \ell_i(\mathbf{f})} && \text{(demand - packing)}\\
z_i &\triangleq e^{-\frac{N}{\varepsilon \cdot \alpha \cdot D_i} \cdot \ell_i(\mathbf{f})} && \text{(fairness)}\\
r &\triangleq e^{-\frac{N}{\varepsilon \cdot U} \cdot g(\mathbf{f})} && \text{(utility),}
\end{aligned}
$$

where ε is an accuracy parameter and N is a scaling factor to be specified later. The algorithm begins with an all 0s solution. In every iteration a specific variable f_{i^*,p^*,t^*} in $\mathbf{f}$ is increased by a small amount. This variable is chosen according to condition $(*)$, which is equivalent to condition (3) from Section 3.3. However, we simplify by including only the unsatisfied covering constraints in $(*)$, i.e., requests i for which $\ell_i(\mathbf{f}) < \alpha D_i$ and possibly the utility constraint if $g(\mathbf{f}) < U$. One can verify the equivalence between $(*)$ and (3) by calculating the appropriate partial derivatives of the following two functions that correspond to $lmax(\mathbf{Px})$, $lmin(\mathbf{Cx})$ respectively:

$$
\ln\left(\sum_{e \in E}\sum_{t=0}^{T} y_{e,t} + \sum_{i=1}^{k} q_i\right), -\ln\left(\sum_{i=1}^{k} \mathbf{1}_{\{\ell_i(\mathbf{f}) < \alpha D_i\}} z_i + \mathbf{1}_{\{g(\mathbf{f}) < U\}} r\right).
$$

Here, 1_A refers to the indicator function of the event A, i.e., $1_A = 1$ if A is true and 0 otherwise.

4.1.2 Analysis

The following two theorems summarize the correctness of the algorithm and bound its runtime (proofs are deferred to [15]).

```
initialization
    $y_{e,t} \leftarrow 1$        $\forall e \in E,\ \forall 0 \leq t \leq T$
    $q_i \leftarrow 1$        $\forall 1 \leq i \leq k$
    $z_i \leftarrow 1$        $\forall 1 \leq i \leq k$
    $r \leftarrow 1, \mathbf{f} \leftarrow 0$
while $\exists 1 \leq i \leq k$ s.t. $\ell_i(\mathbf{f}) < \alpha D_i$ or $g(\mathbf{f}) < U$ do
    Find $1 \leq i^* \leq k$, $b_{i^*} \leq t^* \leq d_{i^*}$, and $p^* \in \mathcal{P}_{i^*}$ s.t.:
        $\frac{\sum_{e \in p^*} \frac{y_{e,t^*}}{c_e} + \frac{q_{i^*}}{D_{i^*}}}{\sum_{e \in E}\sum_{t=0}^{T} y_{e,t} + \sum_{i=1}^{k} q_i} \leq \frac{\mathbf{1}_{\{\ell_{i^*}(\mathbf{f}) < \alpha D_{i^*}\}} \frac{z_{i^*}}{\alpha D_{i^*}} + \mathbf{1}_{\{g(\mathbf{f}) < U\}} \frac{u_{i^*,p^*,t^*}}{U} r}{\sum_{i=1}^{k} \mathbf{1}_{\{\ell_i(\mathbf{f}) < \alpha D_i\}} z_i + \mathbf{1}_{\{g(\mathbf{f}) < U\}} r}$   $(*)$
    if there is no such $i^*$, $t^*$ and $p^*$ then
        abort and return there is no feasible solution;
    else
        // choosing step size parameter $\gamma$
        $\gamma \leftarrow \varepsilon \cdot \min\{D_{i^*}, \min_{e \in p^*}\{c_e\}\}$;
        if $\ell_{i^*}(\mathbf{f}) < \alpha D_{i^*}$ then
            $\gamma \leftarrow \min\{\gamma, \varepsilon \alpha D_{i^*}\}$;
        if $g(\mathbf{f}) < U$ then
            $\gamma \leftarrow \min\left\{\gamma, \varepsilon \frac{U}{u_{i^*,p^*,t^*}}\right\}$;
        // updating internal variables
        $y_{e,t^*} \leftarrow y_{e,t^*} \cdot e^{\frac{\gamma}{c_e}}, \forall e \in p^*$;
        $q_{i^*} \leftarrow q_{i^*} \cdot e^{\frac{\gamma}{D_{i^*}}}$;
        $z_{i^*} \leftarrow z_{i^*} \cdot e^{-\frac{\gamma}{\alpha \cdot D_{i^*}}}$;
        $r \leftarrow r \cdot e^{-\frac{u_{i^*,p^*,t^*} \cdot \gamma}{U}}$;
        // updating $\mathbf{f}$ [a]
        $f_{i^*,p^*,t^*} \leftarrow f_{i^*,p^*,t^*} + \frac{\varepsilon}{N} \cdot \gamma$;
return $\mathbf{f}$;
```

Figure 2: Pseudo-Code for Feasibility Check of $LP(\alpha, U)$

[a] If the increase in f_{i^*,p^*,t^*} causes a covering constraint to change from being violated to satisfied, the algorithm chooses the smallest γ value such that this covering constraint is satisfied with equality.

Theorem 1. *For any $0 \leq \alpha \leq 1$ and $U \geq 0$, if $LP(\alpha, U)$ is feasible then for every $0 < \varepsilon \leq 1/2$ and*

$$
N \geq \ln\left((m(T+1) + k) \cdot (k+1)^{\frac{1+\varepsilon}{1-\varepsilon/2}}\right)
$$

the output $\mathbf{f}$ of Algorithm 2 satisfies:

$$
\begin{aligned}
h_{e,t}(\mathbf{f}) &\leq (1+3\varepsilon)c_e & \forall e \in E, \forall 0 \leq t \leq T\\
\ell_i(\mathbf{f}) &\leq (1+3\varepsilon)D_i & \forall 1 \leq i \leq k\\
\ell_i(\mathbf{f}) &\geq \alpha D_i & \forall 1 \leq i \leq k\\
g(\mathbf{f}) &\geq U.
\end{aligned}
$$

Theorem 2. *For every $0 < \varepsilon \leq 1/2$, Algorithm 2 terminates within $\frac{N}{\varepsilon^2}\left[(1+3\varepsilon)(m(T+1)+k) + (k+1)\right] + (k+2)$ iterations.*

Note that Theorem 1 proves that for any feasible set of packing-covering constraints, the pseudocode shown will find a feasible solution with the caveat that the packing constraints can be violated by a small amount of $(1+3\varepsilon)$. Smaller the value of ε, the less the violation. Note also that $N \approx \ln(mTk)$. From Theorem 2, we see that the number of iterations is at most $O\left(\varepsilon^{-2}(mT+k)\ln(mTk)\right)$; so smaller ε increases running time. We use $\varepsilon = .1$ in TEMPUS.

4.2 Online Temporal Planning

Here, we present a novel formulation that copes with online arrival of requests and the TEMPUS algorithm. The algorithm has two main properties: it sets the α_i in a fair manner; and it makes only *incremental* changes to the solution at each subsequent time-step.

We index the online model with the current time-step τ. Let $R(\tau)$ denote the requests that are relevant at time τ, i.e., requests that the algorithm is aware of (those with $a_i \le \tau$) and whose deadline did not expire yet (those with $d_i \ge \tau$). Analogous to §4.1, for a time-step τ, we consider the following quantities:

$$h^{\tau}_{e,t}(\mathbf{f}) \triangleq \sum_{i\in R(\tau):\max\{\tau,b_i\}\le t\le d_i} \ \sum_{p\in\mathcal{P}_i:e\in p} f_{i,p,t}$$

$$\ell^{\tau}_{i}(\mathbf{f}) \triangleq \sum_{t=\max\{\tau,b_i\}}^{d_i} \ \sum_{p\in\mathcal{P}_i} f_{i,p,t}$$

$$g^{\tau}(\mathbf{f}) \triangleq \sum_{i\in R(\tau)} \ \sum_{p\in\mathcal{P}_i} \ \sum_{t=\max\{\tau,b_i\}}^{d_i} u_{i,p,t}\cdot f_{i,p,t}\ .$$

As before, the online problem can be formulated as two linear optimization problems at each time-step τ: first maximize $\min_i \alpha_i$ and then maximize U, with the following constraints:

$$\begin{aligned} LP^{\tau}(\boldsymbol{\alpha},U) = \{ & h^{\tau}_{e,t}(\mathbf{f}) \le c_e && \forall e\in E, \forall \tau\le t\le T,\\ & \ell^{\tau}_{i}(\mathbf{f}) \le D_i - F_{i,\tau-1} && \forall i\in R(\tau),\\ & \ell^{\tau}_{i}(\mathbf{f}) \ge \alpha_i D_i - F_{i,\tau-1} && \forall i\in R(\tau),\\ & g^{\tau}(\mathbf{f}) \ge U - U_{\tau-1},\\ & \mathbf{f}\ge 0\}. \end{aligned}$$

Here $F_{i,\tau-1}$ denotes the total flow of request i routed *prior* to time step τ, i.e., up to time step $\tau-1$ including. Similarly, $U_{\tau-1}$ denotes the total utility obtained *prior* to time step τ. Formally,

$$F_{i,\tau-1} \triangleq \sum_{t=b_i}^{\tau-1} \sum_{p\in\mathcal{P}_i} f_{i,p,t}$$

$$U_{\tau-1} \triangleq \sum_{i=1}^{k} \sum_{p\in\mathcal{P}_i} \sum_{t=b_i}^{\tau-1} u_{i,p,t}\cdot f_{i,p,t}\ .$$

It is important to notice the following. First, $\boldsymbol{\alpha}$ given as input to $LP^{\tau}(\boldsymbol{\alpha},U)$ is a *vector* since different requests can have different promises based in part on when they are made aware to the optimization problem. Second, coordinates $f_{i,p,t}$ of $\mathbf{f}$ are variables of $LP^{\tau}(\boldsymbol{\alpha},U)$ only if the request is relevant and the time has not yet passed, i.e., $t\ge\tau$ and $i\in R(\tau)$. Flow variables from the past are fixed constants and cannot be changed by the algorithm.

4.2.1 *The basic* TEMPUS *algorithm*

First, we describe how to check whether a feasible flow assignment $\mathbf{f}$ exists that satisfies $LP^{\tau}(\boldsymbol{\alpha},U)$ given some values for $\boldsymbol{\alpha}$ and U. Observe that, to check feasibility, one can apply Algorithm 2 with syntactic changes that follow from the definition of $LP^{\tau}(\boldsymbol{\alpha},U)$. Specifically, replace D_i by $(D_i - F_{i,\tau-1})$, replace αD_i by $(\alpha_i D_i - F_{i,\tau-1})$ and replace U by $(U - U_{\tau-1})$. Further, the range of search for a satisfactory increment i^*, t^* should be restricted to flows that are relevant and time that is in the future, i.e., $i\in R(\tau), t\ge\tau$. Similarly, the denominators of condition $(*)$ are changed to include only the relevant edges $(t\ge\tau)$ and requests $(i\in R(\tau))$.

Second, it is more interesting to examine how the solution from the current time-step $\mathbf{f}$ is usable as a starting point in the subsequent time-step.[1] All requests i that are not new, i.e., $a_i<\tau$, already have a promised α_i value from the previous time step. Newly aware requests are initialized to a promise of 0; all their flow variables are also set to 0. All flow variables on edges in time-steps that have not been allocated before are also set to 0. Additionally let U be the guarantee on the secondary utility from the previous time step. Our online algorithm conducts a *water filling* process, in which the lowest promise values are increased as long as $LP^{\tau}(\boldsymbol{\alpha},U)$ is feasible, while keeping U from the previous time step fixed. Once the water filling process cannot proceed anymore, the resulting $\boldsymbol{\alpha}$ is fixed and a search for the largest possible U value is conducted.

We note that by using the previous solution $\mathbf{f}$ as a starting point in each time-step, the *total* network capacity that TEMPUS allocates across all steps is the same as the offline optimization; bar a few cases where pre-promised flows are moved elsewhere. Hence, intuitively, the work done by the algorithm and its running time are amortized over time-steps; i.e., the per time-step running time is very small. General LP solvers do not behave similarly.

4.2.2 *Analysis*

Similar to Section 4.1.2, one can prove that if $LP^{\tau}(\boldsymbol{\alpha},U)$ is feasible, then the online algorithm, regardless of the $\mathbf{f}$, will find an $i^*\in R(\tau)$, $t^*\in[\max\{\tau,b_{i^*}\},d_{i^*}]$ and $p^*\in\mathcal{P}_{i^*}$ that satisfy $(*)$.

We will focus though on bounding the number of iterations that the online algorithm executes per time-step. Let K be an upper bound on the number of relevant requests in any time step τ, namely $|R(\tau)|\le K$ for all $0\le\tau\le T$.

Theorem 3. *The number of iterations in the rth* successful *execution of the algorithm that decides whether $LP^{\tau}(\boldsymbol{\alpha},U)$ is feasible, is upper bounded by*

$$K+1+\frac{N}{\varepsilon^2}\left[X_r+(K+1)\right]\ ,$$

where the summation of X_r over all successful executions is at most $(1+3\varepsilon)(m(T+1)+K)$.[2]

Note that as before $N\approx\ln(mTK)$ and the number of iterations per timestep is at most $O\left(\varepsilon^{-2}(X_r+K)\ln(mTK)\right)$, where $X_r = O(mT+K)$. Comparing this with Theorem 2 shows that the total number of iterations used by TEMPUS are similar in the offline and online cases; the proof can be found in [15].

4.2.3 *Additional Features in* TEMPUS

We now discuss additional aspects in TEMPUS that build upon the above basic algorithm (§4.2.1). Throughout this discussion, τ refers to the current time-step while t refers to a general time-step.

Traffic Smoothening: Recall that to retain fairness when requests arrive online, TEMPUS systematically under-allocates future edges. Otherwise, early arrivals can use up all of the capacity resulting in poor promises to later arrivals and low fairness.

Intuitively, our goal with *traffic smoothening* is to limit the capacity that is allocated at any one time-step to leave room for future arrivals. Formally, we denote by $\beta_{e,t,\tau}$ the fraction of the capacity of edge e in future time t that can be allocated by online TEMPUS at time step τ. This changes the capacity constraint of $LP^{\tau}(\boldsymbol{\alpha},U)$ as follows: $h^{\tau}_{e,t}(\mathbf{f})\le\beta_{e,t,\tau}c_e$. Given an edge $e\in E$ and a time t, $\beta_{e,t,\tau}$ is a function defined for $t\ge\tau$ with a non-decreasing value in $[0,1]$. We require that $\beta_{e,\tau,\tau}=1$, since the entire capacity of an edge that belongs to the current time-step can be safely used. In our experiments, we use $\beta_{e,t,\tau} = exp\left(-(t-\tau)/c\right)$ for some constant c.

[1]Note that $\mathbf{f}$ also determines the initial values of $y_{e,t}$, q_i, z_i and r as per their definition in §4.1.

[2]Choose $N\ge\ln\left((m(T+1)+K)K^{(1+\varepsilon)/(1-\varepsilon/2)}\right)$.

Note that this function depends only on $t - \tau$, i.e., how much further into the future the edge is. An important side-effect is that the promises offered by the online allocation rely more on network capacity in the immediate future than on network capacity that is far into the future; this reduces the risk of reneging on promises since unpredictability increases the further one looks into the future.

Bounded Horizon: In order to speed up the running time, Tempus optionally uses a *sliding window* approach – given window size W, Tempus only plans up to W time steps into the future. Accordingly, $LP^{\tau}(\boldsymbol{\alpha}, U)$ can be edited to include only time-steps $t \in [\tau, \tau + W]$. We replace active requests whose deadline falls outside the window, with equivalent smaller ones that fits within the window. That is, the smaller request has deadline $\tau + W$ and has demand proportional (according to time) to its current unmet demand. When W is large, the fraction of total demand from such requests is small and there is little net impact on solution quality.

Capacity Grace: As stated in Theorem 1 and §3 when discussing their applicability, iterative packing covering solvers can produce solutions that violate the packing constraints (esp: edge capacities) by a small multiplicative factor of at most $(1+3\epsilon)$. Normally, this is not an issue due to the above under-allocation, i.e., $(1+3\epsilon)\beta_{e,t,\tau} \cdot c_e$ can be smaller than c_e. Hence, then, we set the β value in the next time step ($\beta_{e,t,\tau+1}$) to be the maximum between the planned β value and the edge capacity fraction that has already been allocated ($\frac{h^{\tau}_{e,t}(\mathbf{f})}{c_e}$). This approach is the common case and allows us to cope with capacity violations due to the approximate nature of the solver.

Post-Processing LP: However, it may be possible that the capacity violations due to the solver truly exceed edge capacity. This is more likely for the current time-step, wherein all of the edge capacities are available for allocation ($\beta_{e,t,t} = 1$) as well as for time-steps in the near future that have high β values. One possibility is to try and repair the over-allocation by steering traffic from the edges where capacity is violated on to other edges and times. However, since the capacity violation can happen on any future edge, the optimization problem to repair would also have time as a dimension and can be large. Another alternative is to reject solutions that violate edge capacity, i.e., consider a given $(\boldsymbol{\alpha}, U)$ feasibility question to be infeasible even though the solver reports feasibility if it offers a flow assignment $\mathbf{f}$ that violates edge capacity. Tempus chooses this option. Due in part to such rejections the approximate solver can fail to fully allocate the edges in the current time-step even though flows that could use the capacity exist. Hence, after the searches for maximum $\boldsymbol{\alpha}$ and then maximum U have concluded, Tempus uses a post-processing LP that aims to maximize the utility by allocating all of the unused capacity in the current time-step. Since this LP only relates to the current time-step, it is considerably smaller than the calendaring LP and can be solved quickly.

High-Priority Requests: As already mentioned in §3, high priority requests arrive in an online manner and *must* be satisfied. To incorporate highpri requests in $LP^{\tau}(\boldsymbol{\alpha}, U)$, Tempus adds a new request that arrives and ends in the current time-step, has demand equal to the cumulative demand of highpri requests, and requires full service, i.e., promise = 1. Further, for future time-steps ($t > \tau$), recall that Tempus sets aside some capacity for highpri requests ($\chi_{e,t}$). This can be done by replacing the corresponding c_e with $(1-\chi_{e,t})\cdot c_e$ in the above description of Tempus. Tempus infers χ_{e_t} based on historical usages of highpri traffic per-edge, per-time-of-day.

Since the χs are only estimations, it might be that when the true demands of highpri requests for the current time-step become known, they need more than the fraction set aside for them. As a consequence, satisfying the highpri requests fully may require lowering previously promised flow. As before, note that Tempus's under-allocation of future edges helps: in particular, the allocation in the previous time-step leaves some capacity unallocated on the edges in the current time-step ($1 - \beta_{e,\tau,\tau-1}$ fraction). If the over-use is smaller than this amount, nothing more needs to be done. If not, Tempus uses a small pre-process LP that considers only the current time-step and lowers pre-scheduled flow by the amount needed to accommodate the highpri requests while minimizing the effect on the promises already offered. As the name denotes, the pre-process LP runs first, at each time-step. Since the pre-process LP only considers the current time-step, it is small and quick. Further, when Tempus subsequently searches for best $\boldsymbol{\alpha}$ satisfying $LP^{\tau}(\boldsymbol{\alpha}, U)$, any requests that were impacted by the pre-process LP, and hence had their promises reduce, would be preferentially served, i.e., offered more flow at some future time-step. Finally, we observe that link and switch failures can also be modeled as if an additional highpri request arrives on each of the failed edges with demand equal to link capacity. Intuitively, this structure – ensure current time-step is feasible and then mitigate the impact on requests by giving them more flow on other paths or at other times – allows Tempus to accommodate apriori unknown amounts of highpri traffic with a small impact on promises to long-running transfers.

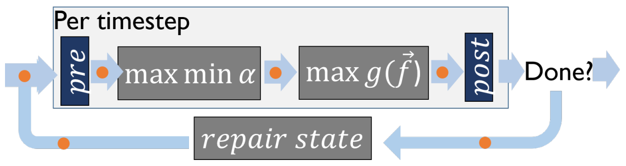


Figure 3: Tempus's workflow

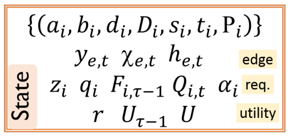

$\{(a_i, b_i, d_i, D_i, s_i, t_i, \mathrm{P}_i)\}$
State
$y_{e,t}$ $\chi_{e,t}$ $h_{e,t}$
edge
z_i q_i $F_{i,\tau-1}$ $Q_{i,t}$ α_i
req.
r $U_{\tau-1}$ U
utility

Figure 4: State kept by Tempus; at time-step τ, for currently active requests $i \in R(\tau)$ and future time-steps $t \geq \tau$.

5. IMPLEMENTATION OF Tempus

So far, we have described the theoretical framework behind Tempus. In this section, we focus on some more practical aspects.

5.1 System overview

Architecture. Figure 3 describes how Tempus operates. At each time-step τ, Tempus first attempts for fair allocation, i.e., maximize the minimum promise to all currently active requests (max min α_i, as in §4.2). Then, to allocate any remaining bandwidth Tempus attempts to maximize total utility (as described in §4.2). Around these core pieces, are a few additional pieces: a *pre*-process LP that at each time step reduces pre-scheduled flows on edges where a higher-than expected volume of high-priority traffic causes allocation to exceed capacity (§4.2.3); a *post*-processing LP that allocates any further remaining bandwidth on the edges at time-step τ so as to improve total utility (as described in §4.2.3); and finally, a state *repair* step that changes the internal state variables from being a feasible solution to the current set of packing-covering constraints towards being a feasible solution to the new set of packing-covering constraints for the next time-step.

State maintenance. Figure 4 depicts the state maintained by Tempus. The first row is the input, the per-request tuple. The next three rows are Tempus's state per edge, per request and for total utility.

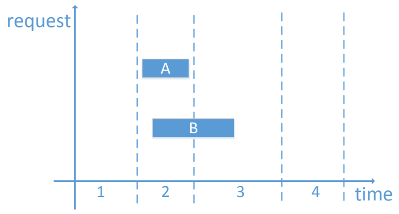


Figure 5: An illustration of how TEMPUS partitions requests to threads. There is one thread per interval. R_A is always assigned to thread 2 since it is only active in that interval. R_B is assigned to either thread 2 or 3 with equal probability since it lies half in each of those intervals.

Per edge, per time, TEMPUS retains the internal variable y (defined in §4.1.1), the fraction of an edge's capacity that has been set aside for highpri traffic χ and the amount that has already been allocated h. Per request, TEMPUS retains internal variables related to its packing and covering constraints (q, z respectively), the amount of flow already routed $F_{i,\tau-1}$, the promised fraction α_i, and the amount of flow to be routed at each future time-step t in order to achieve this promise $Q_{i,t}$. Similarly for utility, TEMPUS retains internal variable r, utility achieved so far $U_{i,\tau-1}$ and promised utility U. In particular, note that TEMPUS does not retain the flow assignment vector $\mathbf{f}$, which has a value per edge, per time, per request; the above state represents everything needed for planning. *Repairing* state, refers to editing state at the end of current time-step to be appropriate for the next time-step. In particular, the internal variables y, z, q, r, are edited as per their definition in §4.1.1 to remove the effect of flow that has been routed in the current time-step. Newly arriving edges for time-step $\tau + W + 1$ are initialized with $y = 1$ indicating zero flow; similarly, newly arriving requests receive $z, q = 1$. Finally, we note that the state maintained by TEMPUS is $O(mW + W|R(\tau)|)$, where m is the number of edges, W is the window size that TEMPUS plans over and $R(\tau)$ is the set of requests that are active at time τ. Of these, we believe $R(\tau)$ to be the largest; several hundred requests between each pair of datacenters. This is significantly smaller than the state used by a typical LP solver: per-edge (or per-path), per-request and per-time, namely $O(mW|R(\tau)|)$.

5.2 Parallelizing Young's algorithm

From Figure 4, observe that some of the state that TEMPUS has to keep can be partitioned by time (y, χ, h) and the rest can be partitioned by request (z, q, F, Q). Further observe from Algorithm 2 that except for the values of $\sum y, \sum z, \sum q$ in the denominators of $(*)$, the remaining computation– finding a tuple (i^*, p^*, t^*), routing additional flow for that request, editing the state etc.– can be done in parallel. Since the calendaring problem becomes harder the more time and more requests that one has to plan jointly, it would be very satisfying to parallelize the computation per-time and per-request; such a parallel version, if possible, could finish in constant time given enough threads.

We are able to achieve this goal partially; the overlap between requests' active times and the sums in the denominators of $(*)$ require synchronization. Step 1: we divide the total time into intervals and assign each to a different thread. The time-intervals are chosen so as to equalize the "load" (request count); (hence, intervals can have different lengths). As described in Fig. 5, requests that are active in multiple intervals are alternatively assigned to either range, with probability equal to the relative duration of the request in that time-interval. Step 2: we ask each thread, given a time interval and some requests, to make forward progress, e.g., 10^4 iterations. While most threads make forward progress, we repeat these steps. Towards the end of the execution, where the number of unsatisfied requests is small, we switch to single-thread execution (the parallelization overhead dominates otherwise). Finally, to avoid contention between threads on the shared variables, we have the threads search for a feasible tuple (i^*, t^*, p^*) and prepare a speculative update to the state variables; the actual update is performed quickly under an exclusive lock. We find speculation to succeed over 99.9% of the time and as we show in our evaluation on a single server, we find a roughly linear speedup up to 30 threads.

5.3 Efficient flow assignment

Searching for a feasible tuple. In principle, finding an (i^*, t^*, p^*) tuple that satisfies $(*)$ (see Algorithm 2) can be hard. A simple random search over each of these values (request, time, path) can take a long time since not all tuples will satisfy $(*)$ esp: in later iterations of the algorithm. Instead, we observe that for each request, its *shortest path* has the smallest value of the left-hand-side (LHS) (path length is measured as $\sum_{e\in p} \frac{y_{e,t}}{c_e}$). Hence, if there exists a tuple that satisfies $(*)$, at least one of the shortest paths must satisfy $(*)$. Using this observation, we maintain a sorted list of the shortest paths through the network for each request, time-step. We then iterate over requests and check if their shortest path (overall) satisfies $(*)$. Note that the running complexity to find a feasible tuple is now the order of number of active requests $O(|R(\tau)|)$, whereas before it was much larger $O(|P_i|WR(\tau))$.

Efficient updates to shortest path. The shortest path data-structure has to be updated when path lengths change; for example, when new flow is allocated on an edge, the edge length $y_{e,t}$ changes. Observe that if some flow was allocated for (i^*, t^*, p^*), the shortest-paths can change only for time t^*. Even better, since the edge lengths only increase, the shortest paths could only have changed for requests whose shortest path at t^* has common edges with the path p^* (whose flow has been augmented). By keeping appropriate data structures (e.g., for each (e, t), the list of requests whose shortest path at time t passes through e), this observation quickly updates the shortest paths that change in each iteration.

6. EVALUATION

6.1 Methodology

Datasets: We evaluate TEMPUS on the topology and traffic matrices from one of Microsoft's production wide area networks. Nodes in the graph are PoPs and edges are bundles of optical links. The network has 40 nodes and 280 edges. We also evaluate TEMPUS on larger synthetic topologies.

We collected sampled NetFlow data at every ingress into the WAN to obtain five-minute averages of traffic sent between PoPs (one day's worth of data overall). From discussions with cluster operators, we identified the portion of this traffic that is considered highpri. Recall that we model highpri traffic as an unpredictable amount of demand that arrives in each time step and has to be completely satisfied. The remaining traffic portion, which is the bulk of the bytes on the WAN, are attributed to large requests. While NetFlow traces do have TCP flow level information, we could not identify which of the flows belonged to a single request (e.g., moving a petabyte between datacenters). Instead, we interviewed cluster operators for typical distributions of the sizes of requests and the durations between their arrival and hard deadline. Given a traffic demand between a pair of PoPs, we repeatedly sample from these distributions to generate large transfer requests until their cumulative volume equals the observed traffic demand.

Load factor: To evaluate calendaring solutions under a wide range of network load, we scale the actual traffic demand matrix by a constant factor, that we call load factor.

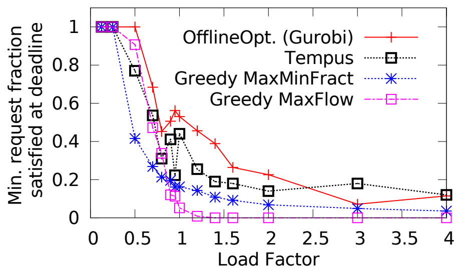


Figure 6: Minimum request fraction satisfied post-facto.

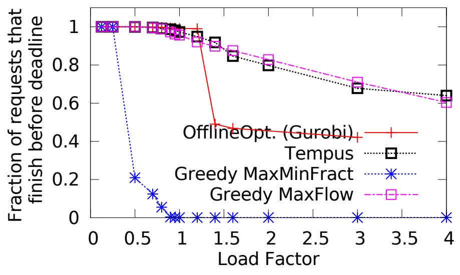


Figure 7: Fraction of transfers that finish before deadline.

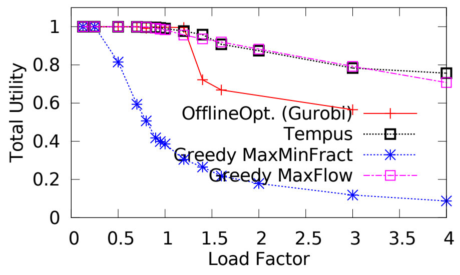


Figure 8: Total utility.

Metrics: We compare solutions on these metrics:

• **MinFract.** (or α) This metric is the minimum of the fraction of request that is transferred before deadline. It bounds the worst-case behavior of a calendaring solution and also measures sensitivity towards unpredictable arrivals or load changes. An ideal calendaring scheme would offer the largest possible value of α.

• **Requests finishing before deadline.** This metric measures how many requests finish on time. When the network load is low, carrying as much flow as possible will satisfy all (or most) deadlines. However, as the network load increases, finishing more requests on time requires carefully spreading traffic across available paths and time. Also, there is a tension between finishing more requests vs. offering a minimum guarantee to every request.

• **Total utility or δ.** Whereas the above measure is binary, 1 when a request meets its deadline and 0 otherwise, this metric is a smoother measure of the total utility from carrying traffic. Recall that our design can handle arbitrary measures of utility, including the case where each request specifies, a non-negative number per timestep to depict the value per unit demand that is satisfied at that time step. Here, we use a simpler variant: the average fraction satisfied over all large transfers. A calendaring solution should maximize utility. However, there is a similar tension between maximizing α (the min. share of a request) versus maximizing δ.

• **Promise.** A guarantee is only useful if provided pre-facto. Whereas α above measures the minimum post-facto fraction that each request gets served, with promise, we measure what the calendaring solution can guarantee ahead of time to each request. This is not relevant for offline solutions; their promise equals the best achievable fraction. For online schemes, however, where future arrivals of large requests and the remaining capacity after serving highpri transfers is unknown, it is challenging to specify a high promise. An ideal calendaring solution should offer as high a promise as possible, as early in a request's lifetime as possible and ensure that the promise does not decrease over request lifetime.

• **Running time, memory usage.** We are also interested in how long the different solutions take to run; how much memory they use; and how well these scale as the size of the problem increases.

Compared schemes:

• Offline Optimal (Gurobi): We formulate the calendaring solution as a sequence of two LPs over all requests and over all time-epochs. The first LP maximizes α, and then the second LP maximizes δ, where the optimal α obtained in the first LP is incorporated as a constraint (α, δ defined above). Gurobi was the best performing of the commercially available LP solvers that we experimented with.

• Tempus: We built an incomplete implementation of Tempus. In particular, bounded time horizon has not been implemented. The *offline* variant does the same as above except by invoking Young's instead of an LP solver. The *online* variant is as described in §4.2. Also, we have implemented parallel versions as described.

• Greedy MaxFlow: This is an online solution. At each time step, this scheme first routes highpri traffic. Then, given all the un-satisfied demands, it uses an LP solver to maximize the total flow that can be served in the current time step. It does not plan into the future.

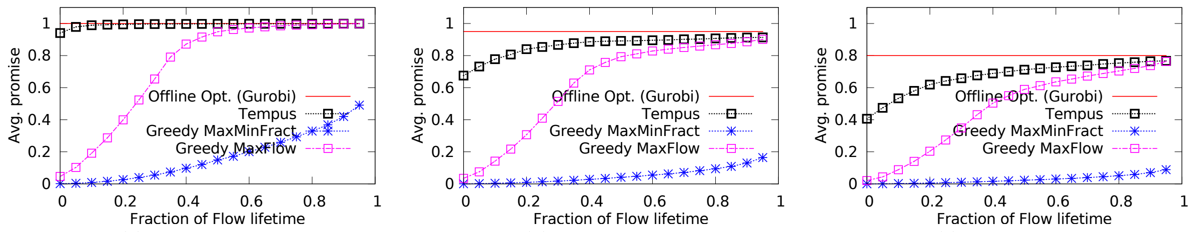
• Greedy MaxMinFract: This is an online solution similar to the above; the only difference is that the LP solver's goal is to maximize the minimum fraction served for all pending requests.

6.2 Value of calendaring

We compare the performance of online version of Tempus with online alternatives and the offline optimal. Figure 6 shows that at high load both the Greedy schemes have very small minimum offered service (α). In fact, the α quickly reaches zero for Greedy MaxFlow; that is unlucky requests can receive no service at all. This is because both the Greedy schemes plan one timestep at a time. By planning into the future, we see that the online variant of Tempus continues to offer a high minimum service, closely approximating that achieved by the offline optimal.

Note that Gurobi could not compute the offline optimal for load factors above 1 because it could not finish within the time-limits imposed by Gurobi software. Sometimes looking for the best α would finish but not the second LP of maximizing δ given the best α. The main issue is the degree of precision that the LP solvers operate at; lower precision values let the solver finish quickly but lead to poor answer quality. E.g., Gurobi has a parameter called optimalitytol; with the default sensitivity value of 10^{-6}, maximizing α yields an α that is 2X smaller than the best possible value; setting sensitivity to 10^{-7} yields the best α but takes 5X more time. Throughout our evaluation, whenever Gurobi solver times-out, we re-run with progressively smaller sensitivity parameters. We believe that doing so is okay because running time is a challenge that cannot be neglected when looking at calendaring solutions. As we show later, Tempus finishes well within that time.

Figure 7 shows that even at high load factors, there is enough residual capacity after serving highpri traffic to finish most requests before their deadline. Between these two figures, it is interesting to note that the Greedy, single timestep planners, have a hard choice. Maximizing the minimum guarantee, such as Greedy MaxMinFract (shown with star markers) does, finishes very few flows before their deadline. This is because whenever some request has a large pending demand, MaxMinFract has to allocate more capacity



(a) Load factor 0.7 (b) Load factor 1.6 (c) Load factor 3

Figure 9: How the promised fraction evolves over lifetime of requests? We show the average promise over all requests.

Instance.	# nodes (n)	# edges (m)	# transfers (R)	# timesteps (T)
prod. dataset	40	280	10^4	288
E1	200	1000	10^4	2000
E2	200	1000	$2 * 10^4$	2000
E3	200	1000	$2 * 10^4$	5000
E4	200	1000	$2.5 * 10^4$	5000
E5	250	1250	$3 * 10^4$	8000

Table 2: Comparing Tempus with Gurobi on larger instances. For all instances, we choose 15 paths per src-dest pair; the transfer sizes and durations were scaled proportional to the number of timesteps, transfers such that the load factor remains around 0.4.

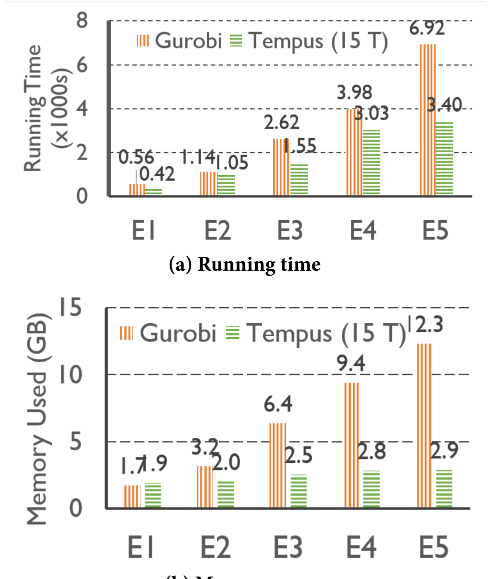
to that request. Maximizing total flow, as Greedy MaxFlow (shown with light squares) does, finishes more flows before deadline but at the cost of significant unfairness (α =0 for load factors $\geq$ 1).

Figure 8 shows a similar trade-off. Whereas the Greedy schemes can only do one of these two well: finish more flows before deadline or offer a high minimum guarantee post facto, the online variant of Tempus does both well and is nearly the same as optimal.

Figure 9 shows how the promise evolves over the lifetime of a request. We see results at three different load factors; the promise value is averaged over all requests; the x axes plots the fraction of request lifetime. First, we see that Tempus begins with a high promise when the requests arrive, much higher than either version of Greedy. Second, due to the online nature of Tempus and the network smoothening, the promise gradually increases with time, as Tempus becomes more confident of how much capacity will be available in the future. The Greedy schemes can promise no more than the fraction that has already been served. Finally, we see that the best possible promise from the offline oracle reduces as the load factor increases. In all cases, Tempus reaches a promise that is over 80% of the best possible promise when the request is a fifth of its way into its lifetime (x=0.2). At high load factors, even when requests come close to their end, neither Greedy scheme offers as large an average promise as Tempus.

6.3 Computational costs, Scalability

Recall that one of our reasons behind using mixed packing covering solvers in Tempus is to scale to larger problem sizes. Figure 10 compares the running time and memory used by Gurobi with an offline variant of Tempus for larger instance sizes. Table 2 describes the instances. Notice that as proved in Theorem 3, the runtime of the online variant of Tempus is $O(\ln(mTK)(X_r+K)/\varepsilon^2)$ per timestep, the $\sum X_r$ over all timesteps is $O(mT+K)$ and the runtime of the offline variant is $O(\ln(mTk)(mT+k)/\varepsilon^2)$. Here, k is the total number of requests whereas K is the number that are active at each timestep; hence $K \leq k$. Hence, loosely speaking, the per timestep runtime of online Tempus is $\frac{1}{T}$ that of the offline variant. Gurobi ran on a 4 core 3.3GHz Intel Xeon E31240 with 16GB of memory. Tempus with 15 threads ran on an 8 core 2.5GHz Intel Xeon L5420 with 16 GB of memory. The latter is several years older; the disparity is because our Gurobi license was for a specific server. Our implementation of



(a) Running time

(b) Memory usage

Figure 10: Comparing the scalability of Tempus vs. Gurobi on the large instances shown in Table 2.

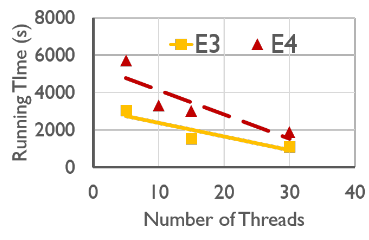


Figure 11: Tempus's running time vs. number of threads.

Tempus is in a high-level language that has garbage collection and is not optimized.

Regardless, we find that Tempus obtains comparable if not better running times than Gurobi. Further, Tempus uses significantly less memory. These gains are because our implementation has no variable that is indexed per time, per request, per path. Though such a variable exists in our formulation, we realize that the state needed for the packing covering problem is smaller (see §5.1 for details).

Finally, Figure 11 shows how the running time of Tempus changes when different numbers of threads are used. We see a sub-linear speedup. At 30 threads, the machine's CPU was consistently pegged at 100%. However, servers with up to 32 or more cores are available at modest prices hinting that Tempus can be sped up further. We take care to point out that our parallelization of Tempus is not the best possible; we attempt to but do not always ensure that the degree of parallelism is as much as the number of threads.

7. RELATED WORK

Traffic Engineering (TE) has been a long standing problem in networking. Early work focused on minimizing the maximum link utilization in order to avoid congestion. Some were adaptive [14]; some were oblivious to demands and network failures [1, 9]; some distributed traffic across paths [1, 14] while others computed suitable routing parameters (e.g., edge weights for OSPF equal cost multipath routing [9]). Recently [5, 6] consider the problem of sustaining different fairness criteria in a shared network. We note that to maximize return on investment inter-DC WANs run at nearly full utilization and hence, minimizing congestion does not apply. Further, prior work does not consider deadline SLAs for large transfers.

Traffic engineering for datacenter WANs has been a recent topic of research. NetStitcher [17] uses information about future bandwidth availability to move data between datacenters at low cost; it does not support deadlines for requests. B4 [13] and SWAN [11], designed by Google and Microsoft, respectively, focus on improving the utilization of the inter-DC WAN. The resource allocation policies offered therein can be classified as greedy/ one timestep, namely they do not compute long-term allocation schedules which are needed to offer deadline SLAs for long-running requests.

Deadlines have been considered in different subareas. Some prior work supports deadlines for flows within a datacenter by suitably modifying TCP or designing new transport protocols [4, 21, 23]. The calendaring problem is fundamentally different, as it operates at a much longer time scale and schedules requests which are aggregations of many flows. Other recent work considers deadlines for batch jobs in large clusters [7, 12, 19]. The algorithmic tools developed therein do not apply to the calendaring context mainly because we need to consider network routing in addition to scheduling.

Packing and covering problems have been considered extensively. Some algorithmic work adapts techniques from multi-commodity flow to general mixed packing and covering systems of inequalities [20] and proves that the running time of the algorithm depends only on the number of variables and constraints [10, 8]. Young [24] presented a further improvement. We extended Young to the online case, observed ways to speed-up the search for a feasible increment, to reduce memory footprint and also a novel parallelization method.

Finally, we note several fast algorithms for solving packing/covering linear programs, and specifically the Concurrent Multicommodity Flow problem. Some of these works deal only with *pure* packing or *pure* covering constraints, e.g., [2, 18], whereas calendaring needs a *mixed* packing/covering LP. Some others [2, 3] take longer to converge when approximation error has to be small. Recall that ε appears in bounds on the amount of violation in constraints; whereas the running time of Young's depends on ε^{-2}, those other works have running time proportional to ε^{-5} and ε^{-6}, respectively. Even faster probabilistic algorithms exist [16], however they are more complex and do not always find a feasible solution.

8. CONCLUDING REMARKS

In this paper, we design Tempus, a novel WAN transfer framework, which schedules long-running transfers in space and time. This allows the network provider to accommodate both high priority traffic and long-running requests. Tempus uses an efficient online algorithm which incrementally utilizes a quick mixed packing-covering subroutine. Our simulation results demonstrate that Tempus clearly outperforms greedy state-of-the-art heuristics in terms of both transfer ratio guarantees and promise quality. Further, Tempus is efficient in terms of running time, memory usage, and update overhead, which we believe would make it an attractive framework for future traffic engineering systems.

While we focus here on making intra-org WAN transfers more efficient, we believe that the ideas in Tempus can be used in other contexts. For example, in the *public* cloud setting, where customers pay the cloud provider for data transfers, Tempus could be used to support new pricing frameworks where users specify data size, deadline and their willingness to pay. Customers can receive discounts for willingness to defer transfers, and providers can exploit price differentiation for more profits. More generally, Tempus can be reused whenever sequences of optimization problems, which change based on some underlying structure, have to be solved efficiently.

Acknowledgements

We thank our shepherd Vyas Sekar and the SIGCOMM reviewers for useful comments.

References

[1] D. Applegate and E. Cohen. Making Intra-Domain Routing Robust to Changing and Uncertain Traffic Demands. In *SIGCOMM*, 2003.

[2] B. Awerbuch and R. Khandekar. Stateless distributed gradient descent for positive linear programs. In *STOC*, 2008.

[3] B. Awerbuch and R. Khandekar. Greedy distributed optimization of multi-commodity flows. *Distributed Computing*, 2009.

[4] L. Chen et al. Towards minimal-delay deadline-driven data center tcp. In *HotNets*, 2013.

[5] E. Danna, A. Hassidim, H. Kaplan, A. Kumar, Y. Mansour, D. Raz, and M. Segalov. Upward max min fairness. In *INFOCOM*, 2012.

[6] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *INFOCOM*, 2012.

[7] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.

[8] L. K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discret. Math.*, 2000.

[9] B. Fortz and M. Thorup. Internet Traffic Engineering by Optimizing OSPF Weights in a Changing World. In *INFOCOM*, 2000.

[10] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and fractional packing problems. *SIAM J. Comput.*, 2007.

[11] C.-Y. Hong et al. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

[12] N. Jain, I. Menache, J. Naor, and J. Yaniv. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. In *SPAA*, 2012.

[13] S. Jain et al. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[14] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *SIGCOMM*, 2005.

[15] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for Wide Area Networks (Extended Version), 2014.

[16] C. Koufogiannakis and N. E. Young. Beating simplex for fractional packing and covering linear programs. In *FOCS*, 2007.

[17] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *SIGCOMM*, 2011.

[18] M. Luby and N. Nisan. A parallel approximation algorithm for positive linear programming. In *STOC*, 1993.

[19] V. Nagarajan, J. L. Wolf, A. Balmin, and K. Hildrum. Flowflex: Malleable scheduling for flows of mapreduce jobs. In *Middleware*, 2013.

[20] S. Plotkin, D. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 1995.

[21] B. Vamanan et al. Deadline-Aware Datacenter TCP (D2TCP). In *ACM SIGCOMM*, 2012.

[22] H. Wang et al. COPE: Traffic Engineering in Dynamic Networks. In *ACM SIGCOMM*, 2006.

[23] C. Wilson et al. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[24] N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *FOCS*, 2001.