Inter-Datacenter Bulk Transfers with NetStitcher

Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez
Telefonica Research
Barcelona, Spain
nikos@tid.es, msirivi@tid.es, yxiao@tid.s, pablorr@tid.es

ABSTRACT

Large datacenter operators with sites at multiple locations dimension their key resources according to the peak demand of the geographic area that each site covers. The demand of specific areas follows strong diurnal patterns with high peak to valley ratios that result in poor average utilization across a day. In this paper, we show how to rescue unutilized bandwidth across multiple datacenters and backbone networks and use it for non-real-time applications, such as backups, propagation of bulky updates, and migration of data. Achieving the above is non-trivial since leftover bandwidth appears at different times, for different durations, and at different places in the world.

To this end, we have designed, implemented, and validated *NetStitcher*, a system that employs a network of storage nodes to stitch together unutilized bandwidth, whenever and wherever it exists. It gathers information about leftover resources, uses a store-and-forward algorithm to schedule data transfers, and adapts to resource fluctuations.

We have compared *NetStitcher* with other bulk transfer mechanisms using both a testbed and a live deployment on a real CDN. Our testbed evaluation shows that *NetStitcher* outperforms all other mechanisms and can rescue up to five times additional datacenter bandwidth thus making it a valuable tool for datacenter providers. Our live CDN deployment demonstrates that our solution can perform large data transfers at a substantially lower cost than naive end-to-end or store-and-forward schemes.

Categories and Subject Descriptors

C.2.5 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Design

Keywords

bulk transfers, inter-datacenter traffic, store-and-forward

SIGCOMM'11, August 15–19, 2011, Toronto, Ontario, Canada.

1. INTRODUCTION

Online service companies such as Amazon, Facebook, Google, Microsoft, and Yahoo! have made huge investments in networks of datacenters that host their online services and cloud platforms. Similarly, hosting and co-location services such as Equinix and Savvis employ distributed networks of datacenters that include tens of locations across the globe. A quick look at any datacenter directory service [2]. reveals that datacenters are popping up in large numbers everywhere. The trend is driven primarily by the need to be co-located with customers for QoS (latency directly affects the revenues of web sites [26]), energy and personnel costs, and by the need to be tolerant to catastrophic failures [5].

1.1 The problem

The dimensioning of a site in terms of key resources, such as the number of servers and the amount of transit bandwidth to the Internet, depends highly on the peak demand from the geographic area covered by the site. In addition, the demand on a datacenter exhibits strong diurnal patterns that are highly correlated with the local time [21]. The combination of peak load dimensioning with strong diurnal patterns leads to poor utilization of a site's resources.

At the same time, Greenberg et al. [21] report that networking costs amount to around 15% of a site's total worth, and are more or less equal to the power costs [21]. They also report that wide-area transit bandwidth costs more than building and maintaining the internal network of a datacenter, a topic that has recently received much attention [22]. The transit bandwidth of a site is used for reaching end customers and for server-to-server communications with remote datacenters (Fig. 1). Datacenter operators purchase transit bandwidth from Telcos and pay based on flat or 95-percentile pricing schemes, or own dedicated lines.

Peak dimensioning and diurnal patterns hinder the amortization of the investment in bandwidth and equipment. In particular, with flat rate pricing, transit bandwidth sits idle when the local demand drops, *e.g.*, during early morning hours. Similarly, under 95-percentile billing, a datacenter operator pays a charged volume [30] according to its peak demand but does not use the already paid for bandwidth during the off-peak hours. Finally, in the case of owned dedicated lines, peak dimensioning pushes for constant upgrades even if the average utilization is low.

1.2 Our approach

The aim of our work is to rescue purchased but unutilized (leftover) bandwidth and put it to good use for the benefit

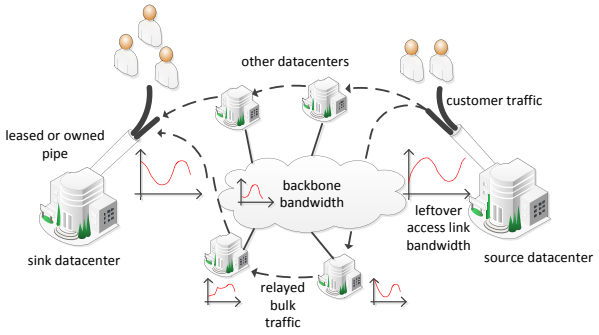

Figure 1: A source datacenter using its leftover transit bandwidth to perform a bulk backup to a remote sink datacenter. The two end points are in different time zones and thus their peak customer hours and their leftover bandwidth appear during non overlapping time intervals. Store-and-forward through intermediate nodes can solve the problem but needs to be aware of the constraints of these nodes and the backbone that connects them.

of backup, bulky updates propagation, and data migration applications running between different datacenters. Such applications can improve the fault tolerance and customer experience of a network of datacenters (Sect. 1.3). Their importance has been recently highlighted in a large scale survey of 106 large organizations that operate two or more datacenters conducted by Forrester, Inc. [10]. A key finding was that the vast majority (77%) of interviewed organizations run backup and replication applications among three or more sites. More than half of them reported having over a peta-byte of data in their primary datacenter and expect their inter-datacenter bandwidth requirements to double or triple over the next two to four years. As a general conclusion, IT managers agreed that the rate at which the price of inter-datacenter bandwidth is falling is outpaced by the growth rate of inter-datacenter traffic. Furthermore, Chen et al. [17] recently reported that background traffic is dominant in Yahoo!'s aggregate inter-datacenter traffic, and Google is deploying a large-scale inter-datecenter copy service [37].

By being elastic to delay, the aforementioned applications can postpone their transfers to non-peak hours when the interactive customer traffic is low. The challenge in efficiently rescuing such leftover bandwidth is that it can appear in multiple parts of the world at non-overlapping time windows of the day; such fluctuations in the availability of leftover bandwidth often relate to time zone differences but can also appear within the same time zone when mixing different types of traffic (*e.g.*, corporate and residential traffic). For example, a sending datacenter on the East Coast has free capacity during early morning hours (*e.g.*, from 3-6am) but cannot use it to backup data to another datacenter on the West Coast. The reason is that during that time the West Coast datacenter is still on night peak hours.

Our key insight is that we can rescue leftover bandwidth by using store-and-forward (SnF) algorithms that use relay storage points to temporarily store and re-route data. The intermediate datacenters are thus stitching together the leftover bandwidth across time and different locations. However to do so, we have to consider both the current and the future constraints of the datacenters as well as the constraints on the backbone network that connects them (Fig. 1).

We have designed, implemented, and validated *NetStitcher*, a system that employs a network of commodity storage nodes that can be used by datacenter operators to stitch together their unutilized bandwidth, whenever and wherever it exists. Such storage nodes can be co-located with datacenters or expanded to other parts of the inter-datacenter network as needed. The premise of our design is that the cost of storage has been decreasing much faster than the cost of wide area bandwidth [20].

NetStitcher employs a store-and-forward algorithm that splits bulk data into pieces which are scheduled across space and long periods of time into the future, over multiple paths and multiple hops within a path. Scheduling is driven by predictions on the availability of leftover bandwidth at access and backbone links as well as storage constraints at storage relay nodes. With this information, the system maximizes the utilization of leftover resources by deriving an optimal store-and-forward schedule. It uses novel graph time expansions techniques in which some links represent bandwidth (carrying data across space), while others represent storage (carrying data across time). The system is also equipped with mechanisms for inferring the available future bandwidth and adapts to estimation errors and failures.

By performing intelligent store-and-forward, we circumvent the problems that plague other approaches, such as direct end-to-end (E2E) transfers or multipath overlay routing. Both of these approaches have no means to bridge the timing gap between non overlapping windows of leftover bandwidth at different sites. Its advanced scheduling, also gives *NetStitcher* an edge over simpler store-and-forward algorithms that do not schedule or use information regarding future availability of leftover resources. For instance, simpler random or greedy store-and-forward algorithms, such as BitTorrent's [18], have a partial view on the availability of leftover resources. Consequently, they may forward data towards nodes that are able to receive data very fast but are unable to relay them in fast paths towards the receiver.

1.3 NetStitcher applications

We now briefly discuss two applications that exchange inter-datacenter non-interactive bulk traffic and can therefore benefit from *NetStitcher*.

Fault tolerance. Improving the fault tolerance of datacenters by increasing the redundancy and security within a single site is too expensive. Quoting the VP of Amazon Web services, *"With incredible redundancy, comes incredible cost ... The only cost effective solution is to run redundantly across multiple data centers."* [5]. Instead of fortifying a single facility, a simpler solution is to periodically backup all the data to a remote location. The key challenge becomes the efficient use of WAN transit bandwidth. Hamilton points out that *"Bandwidth availability and costs is the prime reason why most customers don't run geo-diverse"* [5].

Customer experience. Since network latency impacts directly the usability and the revenues of a web site [26], firms with a global footprint replicate their information assets (such as collections of images and videos) at multiple locations around the world. For example, Facebook runs at least 3 datacenters [4] and holds more than 6 billion photographs [14]. To maintain a high quality customer experience, a new collection needs to be quickly replicated at

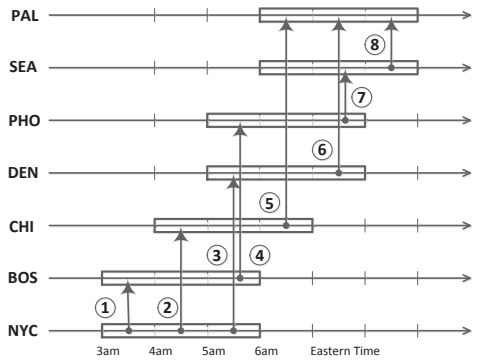

Figure 2: Timing diagram for backing up a datacenter in New York to another one in Palo Alto. The width of a rectangle indicates the window during which a datacenter can perform backups. The diagram shows the operation of a Store and Forward solution. To maximize the amount of data that can be moved between New York and Palo Alto, we need to introduce intermediate storage nodes across three time zones (Eastern, Central, Pacific): in Boston, Chicago, Phoenix, Denver and Seattle.

sites closer to the users. However, due to its "long-tail" profile, such user-generated content challenges traditional pull-based caching approaches: individual objects are accessed by a very small number of users [12]. This means that a large number (or all) of those objects needs to be proactively but cost-effectively replicated.

1.4 Our contributions and results

Our work makes the following contributions:

• The design and implementation of the first system that uses information about future bandwidth availability to provide optimally efficient bulk data transfers.

• The evaluation of *NetStitcher* on an emulation testbed showing that: a) for a large set of datacenter sender-receiver pairs in North America, *NetStitcher* doubles the median transferred volume over 24 hours. Especially when the sender is in the east and the receiver in the West Coast, *NetStitcher* can carry up five times more data than the second best policy; b) *NetStitcher* delivers tera-byte sized files in a few hours, whereas competing policies might take days; c) *NetStitcher* performs well even in international transfers in view of additional backbone network constraints.

• Our live deployment of *NetStitcher* on Telefonica's global CDN. The Points of Presence (PoP) of large CDNs are small datacenters that exchange large volumes of pushed or pulled content. We show that by using leftover CDN bandwidth when the video traffic subsides, our solution can perform bulk transfers at a very low cost per GB. Drawing from these results, *NetStitcher* is planned to augment our CDN service portfolio by allowing content owners to cheaply perform point-to-point high volume transfers. CDN services are more attractive "when viewed as as broader portfolio of services for content owners and providers" [11].

2. MOTIVATING EXAMPLE

We present a simple example to motivate the need for multipath and multi-hop store-and-forward (SnF) scheduling, and to highlight the limitations of existing techniques. The example is based on the real topology of Equinix [3], a company offering hosting services over its global network of datacenter, which includes 22 locations in North America.

Specifically, we want to maximize the volume of data that a datacenter in New York can backup to another one in Palo Alto within 24 hours, assuming that datacenters have a 3 hour time window early in the morning (*e.g.*, 3-6am) for performing such backups. Windows are depicted as rectangles in Fig. 2. Two sites can exchange data directly only if their respective rectangles overlap in time. All sites are assumed to have the same bandwidth capacity.

End-to-end: The simplest mechanism to backup data across the two locations is to use a direct End-to-End transfer. As can be seen in Fig. 2, no data can be backed up using direct E2E transfers since the windows of New York and Palo Alto do not overlap in time. This is a consequence of the window width that we assume, and the 3 hour difference between Eastern and Pacific time. However, even with a wider window – assuming that the windows begin at the same local time – the two sites would be wasting at least 3 hours of valuable bandwidth due to their time difference.

Multipath overlay routing: Overlay routing [13] is used to bypass problematic links, or even entire network segments. In our case, however, the problem is caused by the first and the last hop of the transfer. Thus, overlay redirection does not help because it still requires New York and Palo Alto to have free bandwidth at the same time. In fact, no overlay spatial redirection suffices to address the misalignment of windows. We need a combination of spatial and temporal redirection through the use of storage nodes.

Figure 2 depicts how a scheduled SnF solution can use the bandwidth and storage of 5 intermediate sites to maximize the amount of data that can be transferred between two end points. During the first hour of its window, from 3am to 4am Eastern, New York transfers data to a site in Boston (arrow (1)), where it remains stored for up to 2 hours. Between 4am and 5am, New York uploads to Chicago (2), and between 5am and 6am to Denver (3).

If the above data can be delivered to their final destination before the closing of its window, then New York will have utilized optimally its bandwidth. Indeed this can be achieved as follows. Boston relays its data to Phoenix (4) where they remain for 2 hours. Subsequently Phoenix relays the data to Seattle (7), which finally delivers them to Palo Alto one hour later (8), utilizing the last hour of Palo Alto's window. Chicago delivers its data directly to Palo Alto (5) and so does Denver one hour later (6), thus filling up the first 2 hours of Palo Alto's window.

This simple example shows that multipath and multi-hop SnF succeeds where E2E and Overlay fail. However, real SnF scheduling is substantially more involved due to the existence of varying windows and capacities, as well as additional network and storage bottlenecks. *NetStitcher* copes with this complexity and hides it from the applications. Before discussing the details of its design we will consider whether simpler SnF solutions, such as BitTorrent suffice.

Simple store-and-forward: In a nutshell, simple greedy or random SnF algorithms fail as they do not have complete information on the duration, time and location of windows. This results in data being pushed to sites that cannot deliver it forward to the destination or any other intermediate relay node (or if they do, they do it in an inefficient manner).

For example a node in New York may start greedily feeding a faster node in London, but London's window may close before it can push the data towards any useful direction. All the capacity of New York, which is committed to feeding the London site, is wasted. This is only a motivating example. More intricate inefficiencies occur in complex networks due to the myopic view of simple SnF algorithms.

3. SYSTEM OVERVIEW

NetStitcher is an overlay system comprising of a sender, intermediate storage nodes, and a receiver node. Its goal is to minimize the transfer time of a given volume given the predicted availability of leftover bandwidth resources at the access links and the backbone.

It comprises the following modules: a) *overlay management*, which is responsible for maintaining the connectivity of the overlay; b) *volume prediction*, which is responsible for measuring and predicting the available network resources; c) *scheduling*, which computes the transfer schedule that minimizes the transfer time of the given volume; and d) *transmission management*, which executes the transfer schedule specified by the scheduling module.

The scheduling module is the core of our system and we describe it in detail in Sect. 4. It schedules data transfers across multiple hops and multiple paths over the overlay. The scheduling module only specifies the rate at which each node forwards data to the other nodes. The volume to be transferred is divided in pieces and nodes follow primarily a simple piece forwarding rule: a node can forward each piece only once and to a node that has not received it. (We violate this rule in exceptional cases as discussed in Sect. 4.4 and Sect. 4.4 of [29].) By decoupling the scheduling and forwarding we simplify the design.

3.1 Design Goals

We now discuss the goals of our design and summarize how we achieve them:

Use leftover bandwidth effectively: *NetStitcher*'s multi-hop and multipath transfer scheduler uses unutilized bandwidth resources whenever and wherever they become available. Multipath data transfers enable *NetStitcher* to deal with resource constraints at the intermediate storage nodes. Multi-hop data transfers allow *NetStitcher* to use leftover bandwidth even when the window of the intermediate storage nodes in a single intermediate time zone does not overlap with the window of the sender or the receiver.

Accommodate common pricing schemes: *NetStitcher* leverages unutilized but already paid-for bandwidth, irrespective of the pricing scheme details. Such bandwidth can be manifested in two ways: a) a backbone provider has unutilized transit bandwidth during off-peak hours, which he can offer for cheap to its customers; b) a customer may not be utilizing the capacity he has reserved on a dedicated line. Also, if the customer is billed under the 95th-percentile rule, *NetStitcher* can use the difference between the 95-percentile and the customer's actual utilization.

Easily deployable: *NetStitcher* works at the application layer of low or high-end servers and does not require modifications on network devices.

Adapt to churn and failures: Our design periodically revises the transfer schedule taking into consideration failures, resource prediction errors, and how much data have already been transferred to the intermediate storage nodes and the receiver. By using optimal flow algorithms and resolving extreme estimation error through an end-game-mode (Sect. 4.4), we obviate the need for inefficient redundancy through re-transmissions or forward error correction codes. Due to prediction errors and unexpected events, the initially computed transfer schedule may be erroneous. Its revision may yield a longer transfer time than initially scheduled. Therefore, at the start of a transfer, *NetStitcher* can guarantee meeting a set deadline only in the absence of critical failures and unpredictable resource availability changes.

4. SCHEDULING

We now provide a detailed description of the scheduling component in three stages: a) perfect knowledge under network bottlenecks (backbone constraints) only; b) perfect knowledge under both network and edge bottlenecks; and c) imperfect knowledge under network and edge bottlenecks.

4.1 Scheduling problem statement

First, we formulate the problem that scheduling addresses. Consider a sender node v that wants to send a large file of size F to a receiver node u. The sender can utilize any leftover uplink bandwidth that cannot be saturated by its direct connection to the receiver to forward additional pieces of the file to storage nodes in the set W. Nodes in W can in turn store the pieces until they can forward them to the receiver or another storage node. We define the *Minimum Transfer Time* (MTT) problem as follows:

DEFINITION 1. *Let $MTT(F, v, u, W)$ denote the minimum transfer time to send a file of size F from v to u with the help of nodes $w \in W$ under given uplink, downlink, storage, and backbone network constraints (bottleneck) at all the nodes. The Minimum Transfer Time problem amounts to identifying a transmission schedule between nodes that yields the minimum transfer time $MTT(F, v, u, W)$.*

The task of *NetStitcher* is to achieve $MTT(F, v, u, W)$ in controlled environments where the available bandwidth is known a priori, or approximate it given some error in bandwidth prediction and occasional node churn. We also note that it is straightforward to adapt the algorithm that solves the MTT (Sect. 4.2.1) so that it maximizes the volume transferred in a given time period.

Figure 3 depicts a group of *NetStitcher* nodes and summarizes the notation for the rest of the section. $U_w(t)$ and $D_w(t)$ denote the volume of data from file F that can be sent on the *physical* uplink and downlink of node w during time slot t, which we refer to as *edge bottlenecks*. In the simplest case, edge bottlenecks are given by the nominal capacity of the access links and thus are independent of time. They become time dependent if a site's operator allocates bandwidth to the system only during specific time-of-day windows. $N_{ww'}(t)$ denotes the volume of data that can be sent on the *overlay* link from w to w' due to *network bottlenecks* (backbone constraints). Such bottlenecks capture the bandwidth allocation policies of backbone operators that might, for example, prohibit bulk transfers over links that are approaching their peak utilization. *NetStitcher* can use such information to derive backbone operator-friendly bulk transfer schedules (more in Sect. 7.3). $S_w(t)$ denotes the maximum volume of data from file F that can be stored at w during time slot t. It is dependent on the node's storage

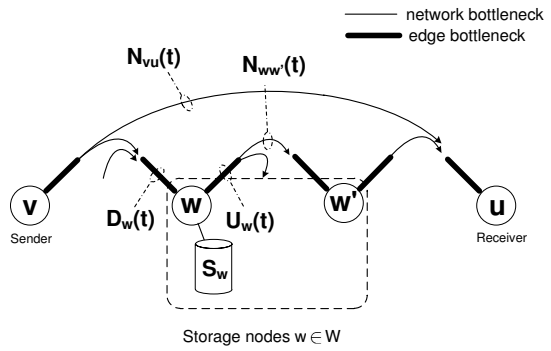

Figure 3: Model and definitions. $U_w(t)$, $D_w(t)$: uplink and downlink edge bottlenecks of w at time t. $N_{ww'}(t)$: network bottleneck of overlay connection $w \rightarrow w'$ at time slot t. $S_w(t)$: storage capacity of w at time slot t.

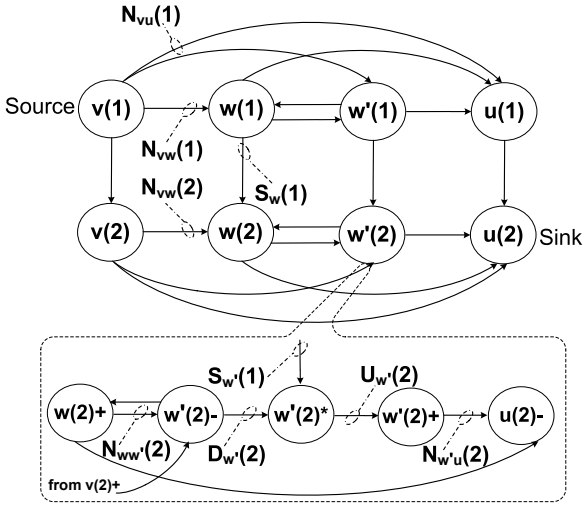

Figure 4: Reduction of the MTT problem to a variant of the maximum flow problem using time expansion. The upper portion of the figure shows the reduction when we consider only network bottlenecks. The bottom (zoomed-in) portion of the figure shows the reduction when we consider edge bottlenecks.

capacity and on the amount of storage required by other applications or *NetStitcher* sessions.

4.2 Perfect knowledge

Our approach is to reduce the MTT problem to the max-flow problem. We express the edge and network bottlenecks and the storage limits as the capacities of edges in a max-flow graph. We model the time dimension of the MTT problem through time expansion as we shortly describe.

First, we consider the case in which the bandwidth and storage resources, $U_w(t)$, $D_w(t)$, $N_{ww'}(t)$ and $S_w(t)$, are known a priori for all $w \in W$ and all time slots t that make up an entire day.

4.2.1 Network bottlenecks only

We initially consider the case when there are no edge bottlenecks. We reduce the MTT problem of minimizing the delivery time of volume F using nodes with time-varying storage and bandwidth to a maximum flow problem with constant edge capacity. The reduction is performed through the time-expansion shown in Fig. 4. Let T_{max} be an upper bound on the minimum transfer time. We construct a max-flow problem over a flow network $G(V, E)$ as follows:

• *Node set V:* For each storage node $w \in W$ (Sect. 4.1), and for $1 \leq t \leq T_{max}$, we add T_{max} virtual nodes $w(t)$. Similarly for the sender v and the receiver u.

• *Edge set E:* To model storage constraints, for $1 \leq t \leq T_{max} - 1$, we connect $w(t)$ with $w(t+1)$ with a directed edge of capacity $S_w(t)$. We repeat the same for the sender v and the receiver u. To model network constraints, for $1 \leq t \leq T_{max}$, we connect $w(t)$ with $w'(t)$, $w, w' \in W$ with a directed edge of capacity $N_{ww'}(t)$ and similarly for the sender and the receiver.

• *Single source and sink:* The source is the sender virtual node $v(1)$. The sink is the receiver virtual node $u(T_{max})$.

The maximum flow from $v(1)$ to $u(T_{max})$ equals the maximum volume that *NetStitcher* can send over T_{max} time slots. We obtain an optimal transmission schedule for the volume F by performing a binary search to find the minimum T_{max} for which the maximum flow from $v(1)$ to $u(T_{max})$ equals the volume F. We can now map the max-flow solution to a transmission schedule as follows: if the max-flow solution involves flow f crossing the edge $(w(t), w'(t))$, an optimal schedule should transmit a volume of size f from w to w' during slot t.

In most cases, the network bottlenecks observed by *NetStitcher* flows are due to backbone operator rate-limiting policies with respect to bulk flows; namely whether they admit additional bulk traffic on a link during a certain time window. Although the rate-limiting policy may depend on bandwidth availability of the physical link, we assume that in most cases the *NetStitcher* flows comprise a small portion of the link's utilization and do not compete for capacity. Therefore, we model every link that connects two nodes as a distinct edge in E. In the rare cases of multiple *NetStitcher* flows competing on a link, one can detect the shared bottleneck [25] and model it as an additional edge.

4.2.2 Edge and network bottlenecks

We now incorporate the edge node uplink and downlink bottlenecks in the MTT problem. We split each virtual node $w(t)$ in three parts, as shown in the bottom of Fig. 4: the front part $w(t)-$ is used for modeling the downlink bottleneck $D_w(t)$, the middle part $w(t)\star$ models the storage capacity $S_w(t)$, whereas the back part $w(t)+$ models the uplink bottleneck $U_w(t)$. The sender node has only a "+" part and the receiver has only a "–" part. The complete reduction is:

• *Node set V:* For each storage node $w \in W$ (Sect. 4.1) and for $1 \leq t \leq T_{max}$ we add virtual nodes $w(t)-$, $w(t)\star$, $w(t)+$. Similarly for the sender v and the receiver u.

• *Edge set E:* For $1 \leq t \leq T_{max} - 1$, we connect $w(t)\star$ with $w(t+1)\star$ with a directed edge of capacity $S_w(t)$. We repeat the same for the sender v and the receiver u. For $1 \leq t \leq T_{max}$, we connect $w(t)-$ with $w(t)\star$ and $w(t)\star$ with $w(t)+$ with a directed edge of capacity $D_w(t)$ and $U_w(t)$, respectively. Also, we connect $v(t)\star$ with $v(t)+$ and $u(t)-$ with $u(t)\star$ with a directed edge of capacity $U_v(t)$ and $D_u(t)$, respectively. In addition, for $1 \leq t \leq T_{max}$, we connect $w(t)+$ with $w'(t)-$, $w, w' \in W$ with a directed edge of capacity $N_{ww'}(t)$ and similarly for the sender and the receiver.

• *Single source and sink:* The source is the sender virtual node $v(1)\star$. The sink is the receiver virtual node $u(T_{max})\star$.

As before, we obtain the optimal MTT transmission schedule by finding the max-flow for the smallest T_{max} that equals the volume to be transferred F.

4.3 Imperfect knowledge

We have so far assumed perfect prior knowledge of bandwidth bottlenecks which is exactly the case when the datacenter operator regulates explicitly the amount of bandwidth given to the system. The system is also capable of operating in environments with imperfect yet predictable periodic patterns and adapt gracefully to estimation error end node failure. We describe how next.

We periodically recompute the transmission schedule based on revised bottleneck predictions provided by the prediction module (Sect. 5). We also recompute the transmission schedule when we detect an unexpected component failure. The first computation of the transmission schedule is as described above. However, for the subsequent computations, apart from the updated bottlenecks, we need to take into account that the sender may have already delivered some part of the file to intermediate storage nodes and the final receiver. We capture this by augmenting our basic time-expansion: we assign a new demand at the source $F_v \leq F$ and assign a new demand F_w at each intermediate storage node $w \in W$. F_w is equal to the volume of file data w currently stores. This casts the MTT problem into a multiple source maximum flow problem. The details are as follows:

• *Node and Edge sets:* The node and edge sets between the sender, the receiver and the intermediate storage nodes are obtained as described in Sect. 4.2.2.

• *Multiple sources and single sink:* We represent the volume that has yet to be transferred to the receiver as a flow from *multiple* sources to the sink node $u(T_{max})\star$. We then cast the multiple source max-flow problem to a single source max-flow as follows [19]. We create a virtual super-source node S. We connect S to the sender virtual node $v(1)$ with a directed edge of capacity equal to the demand F_v, which is equal to the volume of the file the sender has not yet transmitted to any storage node or the receiver. We also connect S to each storage virtual node $w(1)$ with a directed edge of capacity F_w equal to the volume of file data it currently stores.

An optimal transmission schedule is obtained by finding the minimum T_{max} for which the total flow from the super-source S to $u(T_{max})\star$ equals the remaining undelivered volume: $F_v + \sum_{w \in W} F_w$. The mapping from the resulting flow into a transmission schedule is as before.

We observe the following about our choice to address imperfect prediction with periodic recomputation:

Stateless. Each successive recomputation depends only implicitly on previous ones through the amount of data that has already been delivered to the storage nodes and the receiver. We do not need to consider past schedules.

Simplifies the admission of multiple jobs. The ability to revise the schedule of yet undelivered parts of a job can be used to handle multiple sender-receiver pairs. For example, a new job can run with the storage and bandwidth resources that are available given the already accepted jobs, or it can be allowed to "steal" resources from them. In that case, already accepted jobs will perceive this as estimation error and adjust. If the workload is known a priori, one can perform a joint optimization of all jobs in parallel instead of the simpler online approach mentioned above by reducing it to a maximum concurrent flow problem [35].

Gracefully falls back to multipath overlay routing. Although it is not our intended deployment scenario, *NetStitcher* can be used for settings in which only short term predictions are available. In this case, we make the re-optimization intervals short. As a result, *NetStitcher* behaves similar to overlay routing – only the schedule for the first few time slots which is based on the available information executes. The schedule for later slots for which reliable information is not available will not execute, as it will be preceded by a re-optimization.

4.4 End-game mode

The above algorithm adapts to prediction errors by periodically revising the transmission schedule. In extreme situations however, even periodic revision does not suffice. For example, a node can go off-line, or its uplink can be substantially reduced, unexpectedly holding back the entire transfer due to pieces that get stuck at the node. Since we consider this to be an extreme case rather than the norm, we address it through a simple *end-game-mode* (EGM) [18] approach as follows. Due to the low cost of storage, we can keep at some storage nodes "inactive" replicas of already forwarded pieces. If the piece is deemed delayed by the time that the EGM kicks in, a replica can switch to "active" and be pulled directly by the receiver. Similar to BitTorrent, we further subdivide a piece in subpieces. The EGM starts after a large portion (*e.g.*, 95%) of the file has been delivered, and it pulls subpieces of the delayed piece simultaneously from the source and the storage nodes that store active replicas.

4.5 Computational cost

In both the perfect and imperfect knowledge case, if the demands and capacities are encoded as integer values, we can solve the MTT in $O(F|W|^2 T_{max} \log(T_{max}))$ using the Ford-Fulkerson (FF) algorithm. A typical scenario in our experiments (Sect. 8.1) involves $|W|$=14 intermediate storage nodes, $T_{max} = 100$ and $F = 100000$. Our Python FF implementation can solve this instance in less than 25 seconds on an Intel Duo 2.40GHz, 3MB CPU, with 4GB RAM. For further speedup we can use one of the many heuristic or approximation algorithms for max flow problems [31].

5. IMPLEMENTATION

NetStitcher is implemented as a library that exposes a simple API with the following calls: `join(v)`, `leave(v)`, and `send(v,u,F)`. The first two are for joining and leaving the *NetStitcher* overlay and the third for initiating the transfer of a file F from v to u. It is implemented in *Python* and C++ and is approximately 10K lines. The system comprises of the following modules:

Overlay management: This module runs at the *broker process* and is responsible for adding a new node to the overlay and removing it.

Volume prediction: It runs at the broker process and the *peer processes* and its task is to maintain a time series with the maximum volume of data that can be forwarded to each neighbor during the slots that constitute an entire day. As it is usually the case, we assume that the bandwidth resources exhibit periodic behavior. Thus, the module uses the history of the time series to predict the future volumes. It obtains this history using the bandwidth monitor-

ing tools of our backbone provider, Telefonica International WholeSale Services (TIWS). It uses the Sparse Periodic Auto-Regression (SPAR) estimator [16] to derive a prediction for the next 24 hours. It assumes a period of resource availability $T_p = 24\ hours$, and divides time in 1-hour-long time slots t. SPAR performs short-term forecasting of the volume at time t by considering the observed volumes at $t - kT_p$ and the differences between the volumes at $t - j$ and $t - j - kT_p$, where $t, j, k \in \mathbb{N}$. We derive a long-term forecasting by deriving a prediction for $t+1$ considering the prediction for t, and continuing recursively until we predict the volume for $t + 24$.

Scheduling: This module is invoked at the *scheduler process* by peer v when it wishes to send a file to another receiver peer u. It is responsible for scheduling all data transfers between the sender v, the storage nodes W and the receiver node u. It uses volume predictions to calculate an initial transfer schedule and keeps updating it periodically as it receives updated predictions from the nodes (details in Sect. 4). The transfer schedules are computed by solving a maximum flow optimization problem using the GLPK simplex-based solver of the PuLP Python package [32].

Transmission management: This module runs at the peer processes, getting scheduling commands from the scheduler process. For each scheduling slot, the peers translate the scheduled amount volume to a set of pieces that are transferred during the slot. To avoid creating unnecessary spikes, transfers occur using CBR over the duration of the slot.

6. NORTH AMERICAN TRANSFERS

Since most datacenters are located in North America we begin our evaluation from there. In a subsequent section we look at international transfers. We build our case study around Equinix [3] which operates 62 datacenters in 22 locations across the four time zones that cover North America. The exact breakdown of locations and datacenters to time zones is: a) 16 datacenters in 3 locations in Pacific; b) 2 datacenters in 2 locations in Mountain; c) 12 datacenters in 4 locations in Central; and d) 32 datacenters in 13 locations in Eastern (see [3] for the exact list of locations).

6.1 Experimental setting

Sender-receiver pairs: We assume that only one datacenter in each location participates in the *NetStitcher* network. We consider all sender-receiver pairs of such datacenters that are located in different time zones.

Datacenter characteristics: We assume that each site has 1Gbps uplink and downlink transit bandwidth that is made available for inter-datacenter bulk transfer during early morning hours: 3am to 6am. This access link configuration is chosen based on [10] which reports that most firms had between 1 and 10 Gbps of transit bandwidth in each datacenter where they have a point of presence. The duration of the window was based on discussions with datacenter operators [6]. We vary it in Sect. 6.4 to show its effect. In this first set of results we do not consider backbone constraints.

Transfer policies: We consider the following policies: End-to-End (E2E), Overlay, Random store-and-forward, BitTorrent (BT), and *NetStitcher*. Since in this setting there are no bottlenecks in the backbone and the senders have the same capacity as the receivers, E2E and Overlay perform equally. Consequently, we refer to them as E2E/Overlay. Overlay, Random SnF, BitTorrent, and *NetStitcher* are allowed to use any intermediate Equinix datacenter other than the sender and the receiver. Random SnF is the simplest store-and-forward policy in which the file is cut in pieces and forwarding decisions are completely random. It introduces no redundancy by relaying a single copy of each file part. An alternative approach is to use multiple copies of file parts to deal with the lack of informed scheduling. A simple way to do this with an existing technology is to set up a BT swarm with the initial seeder as the sender, a receiver, and a number of intermediate BT clients acting as relays.

Methodology: We run all the above policies on a dedicated server cluster using our *NetStitcher* implementation and the μTorrent client. Overlay is obtained from *NetStitcher* by setting the available storage of intermediate nodes to a very low value. Random SnF is obtained from *NetStitcher* by not imposing any rate limits specified by the scheduler. E2E is obtained from *NetStitcher* by removing intermediate storage nodes. We repeat each experiment 5 times and we report means and confidence intervals. In all experiments we attempt to sent a 1223GB file. 1223GB is actually the maximum that can be transferred due to access link capacities, independently of timing issues (1Gbps $\times$ 3 hours considering header overheads and TCP inefficiencies).

Metrics: For each pair of datacenters we attempt to send the above mentioned file and report the transferred volume to the receiver within 24 hours starting at the beginning of the window of the sender. We also look at the transfer completion time for a given file size.

6.2 Transferred volume

Figure 5(a) depicts the complimentary cumulative distribution function (CCDF) of transferred volume over 24 hours between all 286 pairs of locations in which Equinix operates at least one datacenter. We make the following observations:

- *NetStitcher* is always better than all other competing policies. E2E/Overlay is better than BT and Random SnF in most pairs. And BT is generally better than Random SnF.
- *NetStitcher* transfers up to 1223GB in one day whereas E2E/Overlay goes up to 833GB, BT up to 573GB and Random SnF up to 448GB.
- The median volume *NetStitcher* transfers (783GB) is almost double or more than that of all other policies .

To interpret the above results we start with the factor that has the most profound effect on performance – the amount of misalignment between windows of leftover bandwidth at the sender and the receiver. In this particular example, the misalignment is directly related to the local time. In Fig. 5(b) we plot the transferred volume of different policies against the time zone difference between the sender and the receiver, which ranges from -3 hours, in the direction from Eastern to Pacific time, and up to to +3 hours, in the opposite direction. To explain better the situation we also include the performance when both ends are on the same time zone (not included in the previous figure). We conclude:

- **When there is no time difference**, E2E/Overlay matches the optimal performance of *NetStitcher* which depends only on the capacity of the two end points. Random SnF suffers by permitting file parts to do random walks over intermediate datacenters. In some cases, they will be delivered through longer suboptimal paths, whereas in other cases they will get stuck in intermediate nodes and never be delivered. By creating multiple copies of the same file part

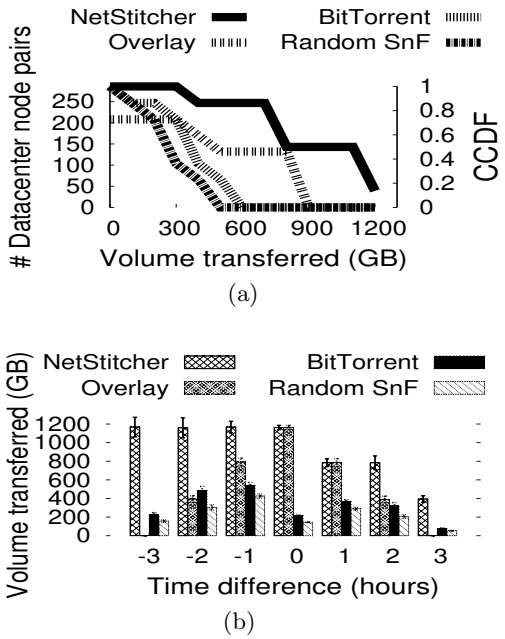

Figure 5: (a) CCDF of the volumes transferred between a *NetStitcher* sender and receiver among all possible sender/destination pairs in the North American Equinix topology. The y axis indicates the number of pairs that have transfered more than the specified volume during the first 24h of the transfer; (b) Mean volumes transferred in 24h and 95% confidence intervals. The x axis indicates the time difference between the receiver and the sender, e.g., -3 means that the receiver is 3h behind the sender.

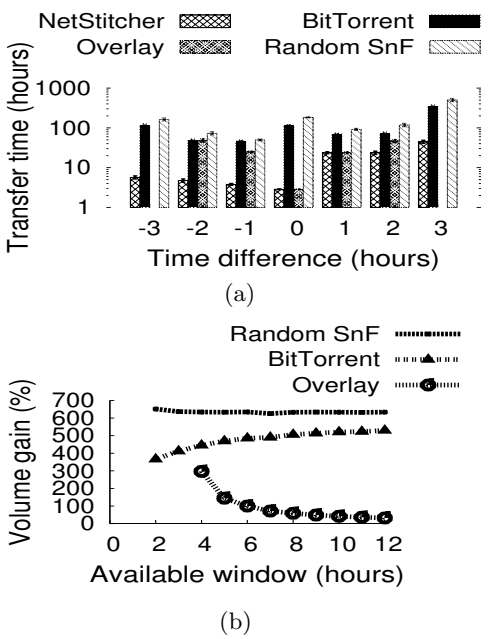

Figure 6: (a) Mean transfer times for 1150GB volume and 95% confidence intervals. The x axis indicates the time difference between the receiver and the sender. For -3h and +3h, difference overlay has infinite transfer time and is not depicted. (b) Normalized transfered volume gains of *NetStitcher* over another policy as a function of the available window.

and relaying them through all intermediate nodes BT pays a penalty in cases when Random SnF succeeds in delivering a file part to the final receiver without using redundant copies. When however Random SnF gets stuck, BT has a better chance to deliver due to the multiple copies it relays. The poor performance of the two extreme cases of simple store-and-forward, full (BT) and no (Random SnF) redundancy, highlights the value of *NetStitcher* scheduling.

• **When the time difference is negative,** *i.e.*, when transferring from east to west, *NetStitcher* remains optimal in terms of absolute transfer volume . In the same direction, E2E/Overlay, BT, and Random SnF decline progressively, suffering by the increasing misalignment of windows. E2E/Overlay however, is more severely penalized due to its lack of store-and-forward capabilities and eventually ends up carrying no data. BT is consistently better than Random SnF but is always at least 50% worse than *NetStitcher*.

• **With positive time difference,** even *NetStitcher* cannot avoid losing capacity. This is because when the window of a sender opens at 3am local time, the window of the receiver at a +1h time zone has already been open for one hour and has been effectively lost. The same applies to E2E/Overlay. With a +2h time zone difference between sender and receiver, E2E/Overlay loses two hours of available bandwidth but *NetStitcher* loses only one. This might seem odd but it is in fact expected. A +2h receiver will be opening its window for the second time 22 hours after the opening of the window of the sender. This hour can be used, not by the sender, but by a +1h intermediate storage node that received the data at a prior time and stored them until that point. This explains why the performance of *NetStitcher* deteriorates slower than E2E/Overlay when going from west to east.

6.3 Transfer time

Another way to observe the performance benefits of *NetStitcher* is to look at the transfer completion time of fixed size files. For this purpose, we set the volume equal to 1150GB, which is close to the maximum that can be shipped by *NetStitcher* in the 3 hours that the window of the sender opens. We measure the transfer time for all policies. The results are depicted in Fig. 6(a):

• For the -3h pairs (Eastern and Pacific), we see that *NetStitcher* delivers the volume on average in 5 hours 45 minutes whereas E2E/Overlay can never deliver it (infinite time), BT needs around 5 days, and Random SnF around 7 days.

• In the absence of time zone difference, *NetStitcher* and E2E/Overlay achieve the absolute minimum of 3 hours, as derived by the access link capacities.

• In the opposite direction, for the +3h pairs (Pacific, Eastern), we see that *NetStitcher* incurs its highest delay of approximately 1 day plus 21 hours approximately. This is approximately 7.5 times smaller than the second best BT.

6.4 Effect of the window

Next we look at the effect of variable available window on the volume transferred over 24. Specifically, we compute the normalized gain of *NetStitcher* against any of the competing policies (*i.e.*, (volume-of-*NetStitcher* − volume-of-

policy)/volume-of-policy), for all sender and receiver pairs. As can be seen in Fig. 6(b), even with a 12 hour window, the gain of *NetStitcher* over E2E/Overlay is approximately 25%. BT and Random SnF are always substantially worse. We have experimented with several other parameters of the system such as the access bandwidth capacities, amount of storage, number of relay nodes, and the results are consistent and supportive of our above main observations.

6.5 North American transfers conclusions

The important lessons from our evaluation are:

• *NetStitcher* always outperforms store and forward policies that do not consider future resource availability, and instead redundantly and/or randomly forward data.

• The benefits of *NetStitcher* over end-to-end or multipath overlay policies, which do not utilize storage, increase with the decrease of the duration of the window and the increase of the time difference between the sender and the receiver.

• Store and forward policies are more effective when the time difference between the receiver and the sender is positive, *i.e.*, the receiver resides west of the sender.

7. INTERNATIONAL TRANSFERS

In this section we turn our attention to international transfers between continents. We use this setting to study two issues: a) the effect of a larger time zone difference between the sender and receiver; and b) the effect of additional constraints appearing in the backbone network. Long haul links (*e.g.*, transatlantic) have more intricate peak and valley behavior than access links since they carry traffic from multiple time zones. Thus they can create additional constraints, e.g., by peaking at hours that do not coincide with the peak hours of their end points. *NetStitcher* can circumvent around such combined constraints due to its advanced network and edge constraint scheduling.

7.1 Experimental setting

Topology: We now present a case study in which Equinix uses the international backbone of TIWS to perform bulk transfers between its sites. TIWS is a transit provider [9] with PoPs in more than 30 countries and peering points with more than 200 other large networks. WS peers with Equinix datacenters in 7 locations in Europe, 1 in Asia, and 4 in US. We emulate a scenario in which *NetStitcher* nodes with 1Gbps uplink/downlink are collocated with Equinix datacenters in Frankfurt, Dusseldorf, Zurich, New York and Palo Alto as shown in Fig. 7. The figure shows the actual WS backbone links that connect the above cities Next to the name of each city we state the real number of Equinix datacenters in that city. For each setting we repeat the experiment 5 times and report mean values in Table 1.

Backbone constraints: We assume that the backbone operator permits bulk traffic to cross only links that are on a valley, *i.e.*, are close to their minimum utilization. Backbone operators naturally prefer to carry delay tolerant traffic during the off peak hours of their links thus improving their overall utilization. This can benefit both backbone operators who can postpone costly upgrades and datacenter operators who can be offered better pricing schemes/QoS if they stir their bulk traffic in an a backbone-friendly manner. *NetStitcher* achieves this objective through its ability to schedule around combined edge and network bottlenecks (described in Sect. 4.2.2). To showcase the above, we used real 5-minute aggregate utilization data from TIWS and integrated these constraints to our emulation environment.

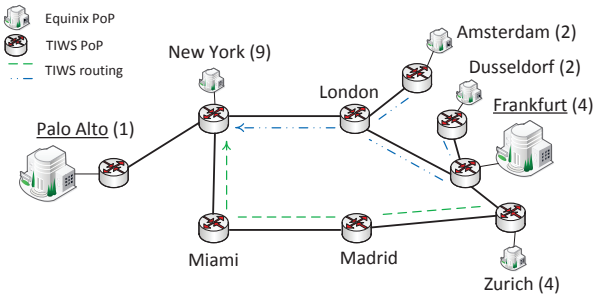

Figure 7: The joint Equinix/TIWS topology used in our transfer from Frankfurt to Palo Alto. In parenthesis we list the number of datacenters operated by Equinix in each city.

7.2 Crossing the Atlantic

We now zoom on the details of a transatlantic transfer between an Equinix datacenter in Frankfurt and one in Palo Alto. This example highlights the additional measures that need to be taken in terms of scheduling and provisioning of storage nodes when transfers are subjected to combined backbone and access link bottlenecks. The Atlantic presents one of the toughest barriers for inter datacenter bulk transfers since there are no major relay nodes in between to assist in the transfer. Therefore we study it to determine what can be achieved in this difficult but important case.

Starting with the sending and receiving windows of the datacenters we notice that these need to be at least 7 hours long. This is because the windows need to bridge the 6 hour gap between central Europe (GMT+1) and the East Coast of the US (GMT-5), and leave at least one hour of overlap for performing the transfer. We assume that the available window starts at 00:00 am local time and ends at 7:00 am local time at each datacenter. Assuming that there are no backbone constraints, the Frankfurt datacenter can upload for 2 hours to 2 datacenters in Amsterdam, another 2 hours to 2 datacenters in Dusseldorf, and another 2 hours to 2 datacenters in Zurich. Then during the last hour of its window (5am-6am GMT), which overlaps with the first hour of the window of New York, Frankfurt, and the other 6 datacenters in Amsterdam, Dusseldorf, and Zurich can push in parallel all their data to 7 datacenters in New York. The New York datacenters can eventually deliver them to Palo Alto, 16 hours after Frankfurt initiated the transfer. Effectively, Frankfurt can deliver to Palo Alto as much data as its window permits within a day (first row of Table 1).

7.3 Putting the backbone in the picture

The previous example shows that large enough windows at the end-points combined with sufficient intermediate storage nodes can bypass the Atlantic barrier. In reality, however, all the European datacenters of Equinix have to cross the Atlantic using one of TIWS's transatlantic links. In particular, due to the topology and routing of TIWS, Frankfurt, Amsterdam, and Dusseldorf cross the Atlantic through the London-New York link, whereas Zurich crosses through the Madrid-Miami link (Fig. 7). However, these links become available for bulk transfers at different times of the day and

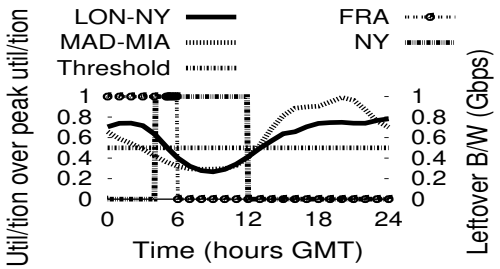

Figure 8: Diurnal utilizations of the London-New York (LON-NY) and Madrid-Miami (MIA-MAD) transatlantic TIWS backbone links. We also depict the cutoff threshold (0.5) and the available windows of the New York (GMT-5) and Central Europe, e.g., FRA, datacenters (GMT+1). Due to confidentiality reasons we report the utilization over peak utilization ratio, instead of the actual utilization.

for different durations. We assume that bulk traffic is allowed only as long as it does not exceed the cutoff threshold, which we set equal to 50% of the peak actual link utilization. Consequently based on our utilization traces, the London-New York link becomes available from 7am GMT to 2pm GMT, whereas the Madrid-Miami link becomes available from 3am GMT to 1pm GMT (see Fig. 8). Thus, the Madrid-Miami link is available for bulk transfers during a bigger part of the day than the London-New York link.

Backbone-constraint-unaware scheduler: A scheduler that is unaware of backbone constraints, such as the above, can push data to datacenters whose exit link over the Atlantic is unavailable when their available windows overlap with the available windows of their intended receiver datacenters in the East Coast. In particular, because of the unavailability of the London-New York link at 5am GMT, all the data at Frankfurt, Amsterdam, and Dusseldorf cannot cross the Atlantic. At the same time, however, the Madrid-Miami link is available, thus the 2 Zurich datacenters can push their data to New York. This way however, only 2 hours worth of data using Frankfurt's uplink capacity can be delivered to Palo Alto during a day.

Backbone-constraint-aware scheduler: The scheduler that is aware of both edge and backbone constraints (Sect. 4.2.2), can deliver up to 4 hours worth of data. This is done by sending 4 hours instead of 2 hours worth of data to Zurich, and having the 4 datacenters in Zurich forward their data to New York through the Madrid-Miami link. Note that the Madrid-Miami link is available between 5am and 6am GMT when the available window of the Central Europe datacenters overlaps with the window of New York. Approximately 4 hours worth of data is the best attainable performance given the number and locations of Equinix datacenters, and the topology and routing rules of TIWS.

To increase the performance to the maximum 7 hours worth of data, Zurich needs to have 7 datacenters. Alternatively, TIWS can install a dedicated *NetStitcher* node in its PoP in Madrid (last row of Table 1).

7.4 International transfers conclusions

The conclusions from this evaluation are:

• Backbone bottlenecks may appear on transatlantic links.

Conditions	Volume Transfered	Transfer Time
No backbone constraints	2720GB	16h
Backbone constraints and constraint-aware scheduler	1553GB	36.4h
Backbone constraints and constraint-unaware scheduler	799GB	∞
Backbone constraints, constraint-aware scheduler and *NetStitcher* node in Madrid	2720GB	16h

Table 1: Volumes transferred during 24h and the total time it takes to transfer 2720GB between the Frankfurt and Palo Alto Equinix datacenters. We repeat the experiment 5 times and report mean values. We consider as the start time of a transfer the moment the window of Frankfurt becomes available.

These bottlenecks render transfer schedules that do not consider them ineffective, and raise the need for a backbone-constraint-aware transfer scheduler.

• The introduction of *NetStitcher* storage nodes in strategically selected locations is needed to bypass network bottlenecks and maximize efficiency.

8. LIVE CDN DEPLOYMENT

In this section we present a real deployment of *NetStitcher* aiming to: a) assess the forecasting accuracy of the SPAR-based predictor; and b) quantify cost savings based on a simple, but realistic pricing scheme.

The system has been installed on a subset of nodes operated by a global CDN platform. The CDN serves video and other content and is deployed in 19 Points of Presence (PoP) in Europe, North and South America. For our experiments we had access to 49 CDN servers, each with direct Gbps access to the backbone and abundant storage. *NetStitcher* is planned to run on top of this infrastructure and take advantage of its resources when the critical operational traffic (video or file-sharing) to local customers subsides.

8.1 Experimental setting

We evaluate *NetStitcher* in the presence of emulated CDN operational traffic. Our purpose is to show that our scheduling algorithm can substantially reduce costs, even when the nodes have irregular but mostly predictable capacity.

Hardware and software: Each CDN server has a quad Intel Xeon 2GHz/4MB CPU, 4GB RAM, 2TB storage, and runs Red Hat 4.1.2-46 with a 2.6.18-164.el5 kernel. Each server has 1Gbps uplink/downlink capacity.

Sender-receiver pairs: We use 3 CDN servers in Spain (Madrid and Barcelona; GMT+1), 2 servers in the UK (London; GMT+0), 3 servers in Argentina (Buenos Aires; GMT-3), 3 servers in Chile (Valparaiso; GMT-4), 3 servers in the East Coast (Miami and New York; GMT-5), 1 server in the central US (Dallas; GMT-6), and 1 server in the West Coast (Palo Alto; GMT-8). We consider all the sender-receiver pairs between servers residing in distinct timezones.

Methodology: We use a 4-day trace of the CDN operational load of the node in Madrid. Unlike other nodes used in our experiment, Madrid's is currently part of our commercial deployment with a load largely induced by a TV broadcaster. Thus, as expected it exhibits a periodic diurnal behavior, allowing us to use it for a more faithful evaluation.

We depict the last 2 days of the trace in Fig. 9(a). For confidentiality reasons we report the ratio of the utilization over the 95th-percentile utilization. We normalize this ratio to correspond to the utilization of the server's 1Gbps capacity. We assign this diurnal load to all CDN servers for the time zone they reside in. We assign the remaining capacity as available resources for *NetStitcher*, E2E/Overlay and BT.

The transfers start on the 4th day of Madrid CDN trace at 0am GMT+1. *NetStitcher* knows the operational traffic load of the first 3 days. It uses it to predict the bandwidth availability during the 4th day with the autoregressive volume predictor. The predictor revises the prediction as described in Sect. 5. Considering the revisions, we recompute the transfer schedule (Sect. 4.3) every 1 hour.

According to this time series, the uplink capacity of the CDN servers suffices to upload at most 4260GB over a day. We therefore arrange for *NetStitcher* and BT to transfer a volume of this size. For each sender-receiver pair we repeat the experiment 5 times and we compute the cost savings of using *NetStitcher*, E2E/Overlay and BT.

Pricing scheme: The cost of leftover bandwidth used by *NetStitcher* and the other policies is zero. The transit bandwidth for our CDN costs $7/Mbps/month. In this deployment, the customer wants to transfer a 4260GB file in 24 hours. If the policy is unable to deliver all the file, the customer sends the remainder using a CBR end-to-end transfer over 24 hours, and pays for the excess transit bandwidth. This excess bandwidth is paid at both the sender and the receiver under the 95th-percentile billing scheme. *NetStitcher* and BT incur the additional cost of intermediate node storage. To compute this cost we consider the storage occupied on an hourly basis. We approximate the storage cost at the intermediate nodes by considering the per-hour normalized cost of Amazon S3 [1]: $0.000076 per GB per hour.

8.2 Live deployment results and conclusions

Performance of the autoregressive predictor: In Fig. 9(a) we depict the forecast of the autoregressive predictor as derived at the end of the 3rd day. As can be seen, the predictor approximates satisfactorily the actual CDN utilization, with a 0.18 maximum error.

Cost savings: Figure 9(b) shows the monthly cost for transferring 4260GB every day with *NetStitcher*, E2E/Overlay and BT against the time zone difference between the sender and the receiver. We observe that:

• *NetStitcher* yields up to 86% less cost compared to the E2E/Overlay when the sender is in GMT+1 and the receiver in GMT-6. It yields up to 90% less cost compared to BT when the sender is in GMT+0 and the receiver in GMT+3.

• *NetStitcher* yields the greatest cost savings when the time difference is negative, *i.e.*, when going from east to west. As the time difference between sender and receiver increases, the costs for all policies increase. BT however is more severely penalized due to its excessive replication, which forces the customer to send up to ∼85% of its data using additional transit bandwidth, when the time difference is +9h. For +9h time difference, E2E/Overlay costs almost as much as *NetStitcher*. In this case, the extra paid bandwidth for *NetStitcher* is slightly lower than E2E/Overlay and do not compensate for *NetStitcher*'s storage requirements. We note that the reported monetary values would be proportionally higher if the deployment involved a larger file, and thereby more servers and more transit bandwidth.

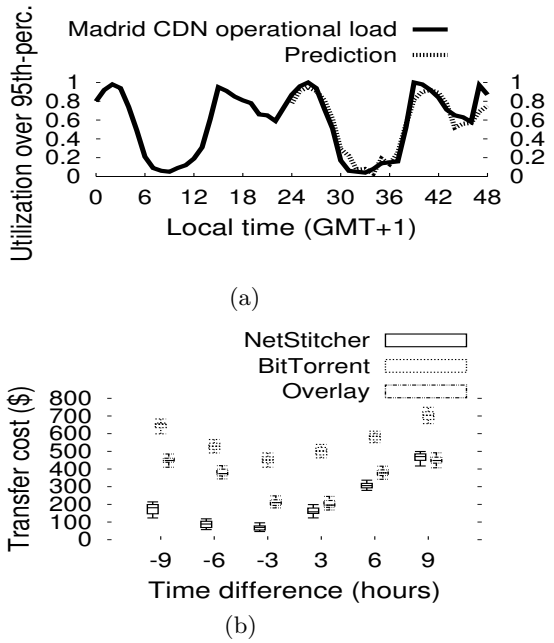

Figure 9: (a) Diurnal operational traffic bandwidth utilization of the Madrid CDN node; (b) Percentile plot of USD monthly cost for transferring 4260GB every day. The x axis indicates the time difference between the receiver and the sender. For each time difference, the boxes are shown in the horizontal order NetStitcher, BT and Overlay.

• The maximum storage required by *NetStitcher* is 619GB per 24 hours, when the sender resides in GMT-8 and the receiver in GMT+1. This entails a monthly cost of ∼$33, which is substantially lower than the median bandwidth cost for this case: $438. On average, storage costs are only 14% of the total cost. This is because an intermediate storage node can delete a piece after forwarding it. On the other hand BT, which maintains copies of forwarded data, needs a maximum storage of 1872GB per 24 hours, yielding a $101 monthly cost.

9. RELATED WORK

Differentiated traffic classes: There have been proposals for bulk transfers at different layers of the protocol stack via differentiated treatment of traffic classes. For instance, the Scavenger service of Qbone [36] tags delay tolerant bulk traffic so that routers can service it with lower priority. However, such approaches use end-to-end flows, and thus suffer from the shortcomings of the E2E policy discussed before. That is, they allow a single bottleneck to block the utilization of bandwidth that gets freed up elsewhere in the path.

Delay tolerant networks: Store-and-forward has been used in proposals for wireless intermittently connected Delay Tolerant Networks (DTN) [24]. Mobile devices forward and store messages with the aim of eventually delivering them to the intended final recipient whose location is unknown and changing. Such systems have to operate in highly unpredictable environments and suffer from disconnections at both the control (routing) and the data plane (transmission). *NetStitcher* is designed to tap on periodic phenomena appearing in wireline networks while having at its disposal

an always connected control plane that allows it to schedule and route efficiently. On a tangential but still related area, [23] shows that mobility combined with storage increases the capacity of wireless DTNs. *NetStitcher* does the same by using storage to stitch together leftover capacity that would otherwise be wasted due to time misalignments.

Breitgand et al. [15] used store and forward to deliver individual low priority messages of a network monitoring system over a single path. They proposed an online algorithm to deal with unpredictable changes in resource availability. We target multipath bulk transfers in mostly predictable environments, which dictates the reduction of our problem to an efficiently computable maximum flow formulation. We address unpredictability by recomputing the schedule sufficiently often and when component failure is detected.

Point-to-point multipath bulk transfers: Pucha et al.'s dsync [34] is a file transfer system that adapts to network conditions by determining which of its available resources (e.g. network, disk, network peers) is the best to use at any given time and by exploiting file similarity [33]. *NetStitcher* does not rely on network peers that store content of interest in advance, and it has different design goals. Signiant [8] and Riverbed [7] offer point-to-point intra-business content transfer and application acceleration platforms, which utilize WAN optimization technologies. One could view *NetStitcher* as an additional WAN optimization technology. To the best of our knowledge, Signiant and RiverBed do not employ store and forward techniques.

Kokku et al. [27] focus on the TCP implications of multipath background transfers. They do not consider store and forward, but instead aim at making bulk background transfers more TCP-friendly, while maintaining efficiency.

Single-hop and single-path SnF bulk transfers: The closest work to *NetStitcher* is [30, 28], which developed analytical models for transferring bulk data through single-hop and single-path transfers while minimizing 95th-percentile transit costs for Telcos. It quantified the cost for transferring bulk scientific data across time zones and compared it to the use of postal service. That work did not address design and implementation issues, whereas *NetStitcher* is a system deployed in a production setting for SnF bulk transfers. In this work, we extend SnF scheduling beyond the simple case of single-path and single-hop over a single unconstrained storage node presented in [30].

10. CONCLUSIONS AND FUTURE WORK

We have presented the design, implementation, and validation of *NetStitcher*. It is a system for stitching together unutilized bandwidth across different datacenters, and using it to carry inter-datacenter bulk traffic for backup, replication, or data migration applications. *NetStitcher* bypasses the problem of misaligned leftover bandwidth by using scheduled multipath and multi-hop store-and-forward through intermediate storage nodes. As future work, we consider results on bootstrapping deployments and on security. An additional interesting topic is to adapt our system for operations in a P2P setting (higher churn, free-riding, etc).

11. REFERENCES

[1] Amazon Simple Storage Service. aws.amazon.com/s3/.
[2] Datacenter map. www.datacentermap.com/datacenters.html.
[3] Equinix datacenter map. www.equinix.com/data-center-locations/map/.
[4] Facebook Statistics. www.facebook.com/press/info.php?statistics.
[5] James Hamilton's blog: Inter-datacenter replication & geo-redundancy. perspectives.mvdirona.com/2010/05/10/InterDatacenterReplicationGeoRedundancy.aspx.
[6] Personal communication.
[7] Riverbed Products&Services. http://www.riverbed.com/us/products/index.php.
[8] Signiant Products. www.signiant.com/Products-content-distribution-management/.
[9] Telefonica International Wholesale Services Network Map. http://www.e-mergia.com/en/mapaRed.html.
[10] Forrester Research. The Future of Data Center Wide-Area Networking. info.infineta.com/l/5622/2011-01-27/Y26, 2010.
[11] Ovum. CDNs The Carrier Opportunity. reports.innovaro.com/reports/cdns-the-carrier-opportunity, 2010.
[12] B. Ager, F. Schneider, J. Kim, and A. Feldmann. Revisiting Cacheability in Times of User Generated Content. In *IEEE Global Internet'10*.
[13] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *ACM SOSP'01*.
[14] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *USENIX OSDI'10*.
[15] D. Breitgand, D. Raz, and Y. Shavitt. The Traveling Miser Problem. In *IEEE/ACM Transactions on Networking'06*.
[16] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *USENIX NSDI'08*.
[17] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu. A First Look at Inter-Data Center Traffic Characteristics via Yahoo! Datasets. In *IEEE INFOCOM'11*.
[18] B. Cohen. Incentives Build Robustness in BitTorrent. In *P2P Econ'03*.
[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. MIT Press'01.
[20] J. Gray. Long Term Storage Trends and You, 2006. http://research.microsoft.com/∼Gray/talks/IO_talk_2006.ppt.
[21] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The Cost of a Aloud: Research Problems in Data Center Networks. *ACM SIGCOMM Comput. Commun. Rev.*, 39(1), '09.
[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a Scalable and Flexible Data Center Network. In *ACM SIGCOMM'09*.
[23] M. Grossglauser and D. Tse. Mobility Increases the Capacity of Ad-hoc Wireless Networks. In *IEEE/ACM Transactions on Networking'01*.
[24] S. Jain, K. Fall, and R. Patra. Routing in a Delay Tolerant Network. In *ACM SIGCOMM'04*.
[25] D. Katabi, I. Bazzi, and X. Yang. A Passive Approach for Detecting Shared Bottlenecks. In *IEEE ICCN'02*.
[26] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers not to the HiPPO. In *ACM KDD'07*.
[27] R. Kokku, A. Bohra, S. Ganguly, and A. Venkataramani. A Multipath Background Network Architecture. In *IEEE INFOCOM'07*.
[28] N. Laoutaris and P. Rodriguez. Good Things Come to Those Who (Can) Wait or How to Handle Delay Tolerant Traffic and Make Peace on the Internet. In *ACM HotNets-VII'08*.
[29] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-Datacenter Bulk Transfers with NetSticher. Technical report, 2011. Available at the authors' website.
[30] N. Laoutaris, G. Smaragdakis, P. Rodriguez, and R. Sundaram. Delay Tolerant Bulk Data Transfers on the Internet. In *ACM SIGMETRICS'09*.
[31] T. Leighton, C. Stein, F. Makedon, E. Tardos, S. Plotkin, and S. Tragoudas. Fast Approximation Algorithms for Multicommodity Flow Problems. In *ACM STOC'91*.
[32] S. Mitchell and J. Roy. Pulp Python LP Modeler. code.google.com/p/pulp-or/.
[33] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting Similarity for Multi-Source Downloads using File Handprints. In *USENIX NSDI'07*.
[34] H. Pucha, M. Kaminsky, D. Andersen, and M. Kozuch. Adaptive file transfers for diverse environments. In *USENIX ATC'08*.
[35] F. Shahrokhi and D. Matula. The Maximum Concurrent Flow Problem. In *Journal of the ACM'90*.
[36] S. Shalunov and B. Teitelbaum. Qbone Scavenger Service (QBSS) Definition. Internet2 Technical Report'01.
[37] D. Ziegler. Distributed Peta-Scale Data Transfer. www.cs.huji.ac.il/~dhay/IND2011.html.