A Decentralized SDN Architecture for the WAN

Alexander Krentsel
Google / UC Berkeley

Nitika Saran*
Cornell

Bikash Koley
Google

Subhasree Mandal
Google

Ashok Narayanan
Google

Sylvia Ratnasamy
Google / UC Berkeley

Ali Al-Shabibi
Google

Anees Shaikh
Google

Rob Shakir
Google

Ankit Singla
Google

Hakim Weatherspoon
Cornell

dsdn-sigcomm@google.com

Abstract

Motivated by our experiences operating a global WAN, we argue that SDN's reliance on infrastructure *external* to the data plane has substantially complicated the challenge of maintaining high availability. We propose a new *decentralized SDN* (dSDN) architecture in which SDN control logic instead runs within routers, eliminating the control plane's reliance on external infrastructure and restoring fate-sharing between control and data planes. We present dSDN as a simpler approach to realizing the benefits of SDN in the WAN. Despite its much simpler design, we show that dSDN is practical from an implementation viewpoint, and outperforms centralized SDN in terms of routing convergence and SLO impact.

CCS Concepts

• **Networks → Network architectures**; **Network protocol design**; Control path algorithms; *Network reliability*;

Keywords

Wide-Area Networks, Software-defined Networking, Traffic Engineering

ACM Reference Format:
Alexander Krentsel, Nitika Saran, Bikash Koley, Subhasree Mandal, Ashok Narayanan, Sylvia Ratnasamy, Ali Al-Shabibi, Anees Shaikh, Rob Shakir, Ankit Singla, and Hakim Weatherspoon. 2024. A Decentralized SDN Architecture for the WAN. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672257

1 Introduction

"The cheapest, fastest, and most reliable components of a computer system are those that aren't there." — Gordon Bell

Availability is a network operator's highest priority and yet, despite significant effort and investment, modern WANs remain vulnerable to failure. Figure 1 shows the frequency of outages that we have experienced in operating the WAN at Google in recent years. Generally speaking, this frequency has not declined over the years despite considerable efforts to improve reliability. Moreover, while minor disruptions are to be expected due to link cuts or router crashes, more impactful medium and major disruptions are harder to rationalize. Major outages continue to be reported by virtually every WAN operator [16, 28, 37, 58, 67]. Avoiding major failures is critical since they impact customers, internal productivity, and market reputation. Ultimately, they also impact innovation as these experiences encourage an overly conservative attitude toward introducing change.

The fundamental challenge in eliminating major failures lies in their complexity: they typically involve bugs or errors in multiple components spanning diverse teams and codebases, that interact in unanticipated ways. As a result, these outages persist, despite our extensive efforts to improve testing, diagnostics, change procedures, and verification.

Given this situation, we ask: Is there more we can do to avoid complex failures? The approach we explore in this paper is to address complexity directly and focus on *simplifying* the network. More concretely, we aim to identify network components that can be removed without impacting the network's functionality or performance. Intuitively, this strategy offers: (i) fewer points of failure and failure modalities, (ii) a reduced search space for testing and verification thus improving coverage, and (iii) fewer maintenance "touchpoints" and hence fewer opportunities for errors.

But are there significant opportunities for simplification in today's WANs? In this paper, we will argue that SDN control infrastructure in the WAN introduces significant complexity and propose a new control architecture that allows us to remove the vast majority of this infrastructure without losing the benefits of SDN. This raises three key questions which we summarize here and address through this paper.

Q1) Why target SDN infrastructure for simplification? While the road to SDN varied across use cases [12], in the WAN context that we focus on, support for operator-driven innovation was a key driver of SDN adoption, enabling operators to customize and evolve network control based on their needs. When SDN was embraced in the mid-2000s, routers were typically closed platforms with no practical way of running 3rd-party code "on-box." As such, supporting operator-defined code naturally implied running the control plane on infrastructure *external* to the data plane. This external SDN control plane has typically been implemented as a logically

*Work done while at Google.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0614-1/24/08.
https://doi.org/10.1145/3651890.3672257

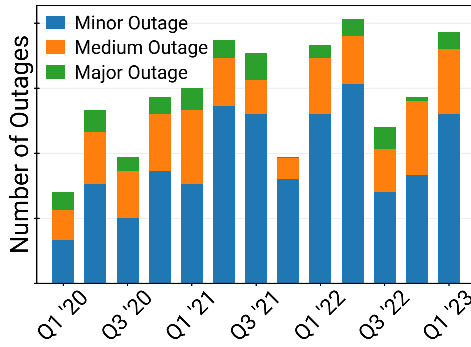


Figure 1: *Historical outage frequency in our WAN. Outages are classified internally based on several factors, including perceived user impact, duration, and blast radius (single or multi-region).*

centralized controller whose essential task is to run a traffic engineering (TE) algorithm that computes paths in a capacity-aware manner [11, 25–27].

Unfortunately, implementing the high-level SDN idea in a global WAN involves far more than simply running a TE process on a datacenter server. It typically involves multiple software services, spanning a hierarchy of controllers, and running on a global footprint of servers that connect to the data plane via a separate control plane network. We use "SDN control infrastructure" to refer to these off-box components that are introduced in addition to the data plane. We note further that this infrastructure is not a one-for-one replacement for traditional solutions because, in practice, WANs continue to run protocols such as IS-IS and BGP as a fallback in case SDN control fails. As a result, we argue that current WAN control planes are overly complex: they bring new dependencies (due to SDN), while still retaining old undesirable dependencies (on protocols and vendors).

Thus the goal of our work is to simplify the network by eliminating these external components and dependencies where possible, thereby improving overall availability.

Q2) What benefits of SDN must a simplified architecture retain? SDN WANs offer important benefits that we must retain as we consider network simplification: *operator-defined innovation*, improved network *efficiency* (because TE computations that act on a global network view have been shown to achieve higher network utilization than greedy distributed solutions such as RSVP-TE [2, 25]) and *simpler "consensus-free" route computations* (because paths are selected by the controller without complex distributed protocols across routers).

Q3) How do we retain the benefits of SDN without its control infrastructure (and without a return to traditional protocols)? We see a way forward through developments in the router industry. In early SDN implementations, separation of control between vendors and operators necessitated separation of infrastructure. However, the router ecosystem has evolved since and vendors now support running 3rd-party code on the router's control CPU and implement vendor-neutral interfaces that expose key control-plane functions to external code [8, 41, 51, 57]. This shift offers an important opportunity to rethink existing SDN designs since we can now run operator-defined code within routers and manage that code in a vendor-neutral manner.

In this paper, we use this opportunity to propose a novel decentralized SDN (dSDN) architecture in which every router runs an operator-defined "dSDN controller." Each dSDN controller constructs a global network view via a simple flooding-based dissemination protocol and then locally runs a TE algorithm to compute capacity-aware paths. Finally, for simple consensus-free path selection we use source routing: when a packet enters the network, the ingress router records the TE-computed path into the packet's header and all other routers along the path simply enforce the source route. Thus in dSDN, a router R is the sole decision maker for paths that originate at R.

The dSDN architecture represents a significant simplification to current WANs: it leverages two common techniques (basic control plane flooding and a TE algorithm) but otherwise eliminates any dependency on components external to the data plane *and* on legacy protocols. Despite these simplifications, dSDN retains the benefits of traditional centralized SDN (cSDN) while restoring the fate-sharing of traditional protocols that provides resiliency. The trade-off that dSDN imposes is a higher computational load on the router CPU (to run TE) and additional packet header state (to carry source-routes). Our evaluation shows that this trade-off is easily accommodated by modern routers.

We recognize that our proposal implies revisiting the SDN deployments that industry has worked on for almost 20 years. This is a radical change that will not be undertaken lightly. As such, it will likely be years before we have the deployment experience to validate our central hypothesis: namely, that the simplifications we propose will reduce complex failures and broadly improve network availability. Hence, the claims and contributions we make in this paper are more modest: (i) we articulate the case for simplification and propose a new WAN control architecture that we believe achieves this simplification while retaining the best of the SDN and traditional protocol paradigms, and (ii) we implement and evaluate dSDN, showing that it can be practically realized on current routers, and significantly outperforms cSDN. Specifically, we show that: (1) our dSDN implementation on an Arista 7808 consumes under 50% of the router's CPU resources under realistic network scenarios, (2) dSDN achieves up to 100x lower convergence times and 10-100x lower SLO impact than our current cSDN WAN, (3) the above gains hold over a range of topology, traffic, and failure scenarios.

2 The Case for Rethinking SDN Infrastructure

2.1 Context

dSDN is motivated by our experiences running two WANs. Our B4 network [27] is based on cSDN, similar to other cSDN-based networks described in the literature [11, 12, 25], while our B2 network is a traditional protocol-based WAN. For capacity-aware path selection, B4 runs centralized TE. By contrast, B2 runs RSVP-TE over IS-IS, and BGP: link state and capacity-related TE attributes are disseminated using IS-IS, based on which each ingress or "headend" router runs a constrained shortest-path computation [48] to compute the shortest path with available capacity from itself to each destination. The headend router then uses RSVP [6] to signal other routers along the selected path, reserving capacity at each. If signaling is successful, the path is installed in the network. If not, *e.g.,* because any router on the signaled path no longer has

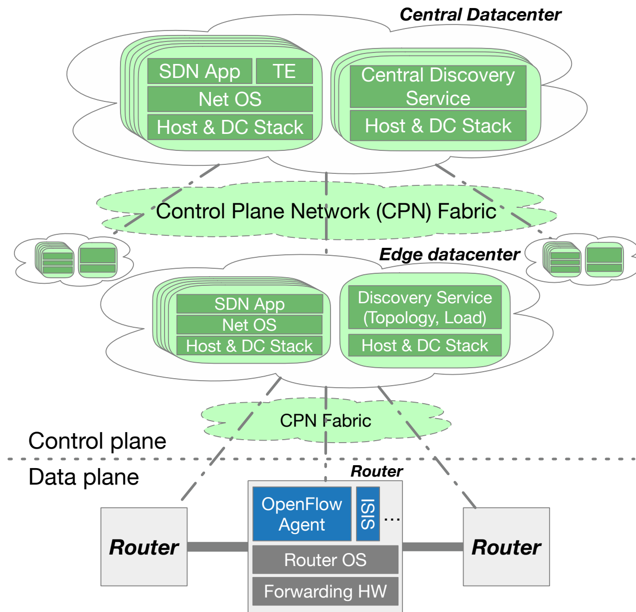


Figure 2: *Conceptual diagram of a typical large-scale WAN architecture deployment [25, 27], with external dependencies shown in green and on-box dependencies shown in blue.*

enough capacity available, then the headend router tries again using a different path. Thus each headend acts independently and makes greedy decisions based on its local view of available capacity. Compared to cSDN's centralized TE, RSVP-TE has been shown to suffer suboptimal paths and hence lower network efficiency (in the sense of lower network utilization) [25].

The dSDN architecture we develop in this paper can serve as a simpler yet equally efficient alternative to B4's cSDN architecture and as a simpler and more efficient alternative to B2's RSVP-based architecture.

2.2 SDN WAN Control Infrastructure Is Complex

Figure 2 shows the main components of a canonical cSDN implementation. The essence of a cSDN controller is its TE algorithm. However ensuring the controller is highly-available and scalable requires more than a single controller programming all switches, but rather a multi-level hierarchy of controllers deployed across multiple data centers and edge locations [25, 27]. For modularity, topology discovery and traffic demand collection run as their own services, as does switch programming, and these services run on data center infrastructure. For example, in B4, the central controller runs on standard data center servers managed by a cluster management system [24, 60, 62, 65]; our edge controllers run on special servers that are co-located with routers and run a specialized SDN platform [5, 13] that is also managed by our cluster management system for infrastructure consistency.

A separate Control Plane Network (CPN) connects controllers to the routers they control, thus avoiding a recursive dependency on network connectivity being already established. The CPN must be physically present in the same locations as all data plane devices, giving it a global footprint with several thousand devices in B4. The CPN requires its own routing for self-bootstrapping, and hence runs a minimal set of traditional routing protocols.

Crucially, each of these components — hardware and software alike — is on the critical path for high availability and thus engineered accordingly, *e.g.,* with redundancy, replication, consensus, *etc.*, becoming fairly complex systems in their own right. For example, in B4, both our edge and central SDN controllers are robustly replicated across distinct hardware and geo-diverse data centers, running Paxos for consistency and failover. In B4, these systems represent dozens of non-trivial microservices and millions of lines of code. Hardware components are likewise architected for resilience with redundant CPN switches and links.

Finally, to avoid a complete loss of connectivity in case of failure in the cSDN control stack, every SDN WAN that we are aware of continues to run IS-IS and BGP [11, 25, 27], programming forwarding entries at a lower priority to provide backup connectivity. These backup paths are insufficient for longer term operation because their placement is capacity unaware, and hence their use can lead to congestion, but they at least provide connectivity.

In summary, as shown in Figure 2, modern WAN control planes span many non-trivial components, both external (shown in green) and on the box (shown in blue).

2.3 Impact on Availability: Examples from B4

External infrastructure increases the surface area for bugs. Each component in Figure 2 introduces the possibility of additional bugs within it, which have the potential to cause outages. In B4, a bug in our topology service resulted in a partial topology (*i.e.,* missing several links) being provided to TE, which led to fewer paths being utilized resulting in severe congestion and over 50% packet loss for several minutes. Similarly, prior work has reported on bugs in the SDN programmer that installs forwarding entries [31], in the controller backup implementation [27], in the network state collection infrastructure [11], and in the CPN [19]. We have also experienced outages due to cSDN's dependency on datacenter management systems. A missing configuration flag to do with cluster management for edge controller jobs led to those jobs being incorrectly killed during routine maintenance, triggering an outage spanning a large geography for several hours.

External infrastructure introduces new failure modalities. cSDN's external infrastructure results in a loss of fate-sharing and thus an expanded space of possible failures: any subset of the control infrastructure can get disconnected from the data plane, preventing reconvergence. In such scenarios, the common strategy is to "fail static" [25, 27] but this is not a panacea since the network's forwarding state becomes increasingly stale as the problem persists. In one incident, we failed static when a misconfiguration on a CPN switch led to a portion of the CPN being disconnected. This interacted poorly with an unrelated maintenance operation that was attempting to reintroduce (previously removed) capacity back into the network. Failing static prevented the restored capacity from being used leading to severe congestion that lasted for the entire duration that it took to diagnose and fix the CPN problem.

Legacy Protocols Remain. Retaining traditional protocols was perhaps not part of the original SDN vision. However, to mitigate

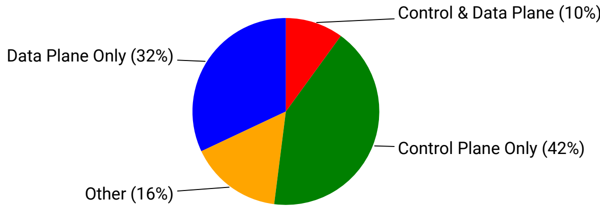


Figure 3: *Classification by root-cause for the 41 largest outages over the last 4 years in B4, with hatching on control-plane related outages.*

risk in the early days of deployment, operators retained their well-known protocol-based control plane as a backup. 15+ years later we continue to experience SDN outages because of which we – and all other WAN operators that we are aware of – continue to maintain this backup. B4 thus runs IS-IS, BGP, and a form of FRR [1] in addition to cSDN. The main simplification compared to our protocol-based network is the removal of RSVP-TE, which is a valuable simplification but far from having eliminated our dependence on vendor protocols. In fact, in addition to the challenges of testing, configuring, and operating protocols, we must now contemplate potential interactions between two control paradigms that were never designed to coexist; *e.g.,* in one published outage [31], when the SDN controller failed, the network fell back to IS-IS and BGP, but a misconfiguration of IGP weights led to severe congestion and packet loss, highlighting the burden on operators to master both TE and IS-IS configuration.

In summary, the complexity of our cSDN control plane in B4 is a dominant contributor to our most challenging outages: Figure 3 shows that 52% of our 41 largest outages have a control-related root cause. We emphasize that our point here is *not* that we must eradicate external control infrastructure. On the contrary, such infrastructure is likely still needed for configuring routers, monitoring, upgrades, *etc.* Instead, our point is that we should avoid putting these components on the critical path for network availability.

2.4 Technologies that Enable a New Architecture

The past decade has brought two key developments which, taken together, enable an alternative routing architecture. The first is that network OS developers [23, 29, 66] have adopted Linux and mainstream container technologies [42] enabling operator-defined application code to be practically deployed on routers. The second is the emergence of a new generation of control and management APIs that enable 3rd-party code to interact with router internals in a manner that is both *comprehensive* and *vendor-agnostic.* While early protocols such as OpenFlow enabled access to a router's forwarding table, other state and configuration was not easily accessible [33]. These new APIs go further in enabling general access to the router's RIB, internal counters, configuration, *etc.* [8, 38, 40, 51, 57]. They allow operator and vendor code to coexist on the same platform, and allow operator code to be portable across platforms and router OSes.

In combination, these developments mean a network operator can now deploy custom code directly on routers rather than external infrastructure.

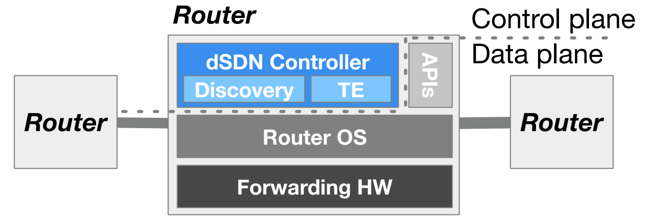


Figure 4: *Diagram of dSDN's system architecture.*

3 Design and Implementation

3.1 Goals and Approach

We design dSDN to meet the following goals: (1) on-box operation with no dependence on external infrastructure, (2) support for operator-defined control code, (3) network efficiency, matching that achieved by state-of-the-art TE solutions, and (4) simple consensus-free path computations. The first goal is a benefit of traditional protocols while the last three are realized by cSDN.

Achieving the first two goals is made possible by the ability to run custom code on routers in a vendor-agnostic manner (§2.4); we envision each router running an on-box container with our custom code that we call the *dSDN controller.* Figure 4 shows this architecture in dSDN as the counterpart to the cSDN control architecture shown in Figure 2. Achieving the latter two goals – efficiency and consensus-free computations – requires more thought.

Efficiency. cSDN achieves efficiency, *i.e.,* high network utilization, by running a TE computation that acts on a global view of topology and traffic demands. To achieve the same in dSDN, we too run TE over a global network view but do so at *every* router. Hence, each dSDN controller discovers its local state – link status, traffic matrix, *etc.* – via the APIs described in §2.4, then floods this information to all other controllers via a simple dissemination protocol (akin to IS-IS). As a result, every dSDN controller has a global network view over which it is able to run TE.

Assuming all routers have the same view, they will compute identical paths and in this sense dSDN is equivalent to having a single controller computing all routes. In practice, of course, different routers may have slightly divergent views and we evaluate the impact of this in §5.

Consensus-free route computation. cSDN enjoys the simplicity of "consensus-free" route computations. A cSDN controller is *authoritative* in that it alone computes paths and programs these as forwarding rules at every router. Each router then follows these rules no matter what its own view of network state is. This authoritative design does not require consensus across routers for path selection. The challenges of distributed consensus are well documented in BGP [21, 64] and even in relatively simple distributed protocols such as IS-IS which can suffer loops and dead-ends until all routers converge.

The key question is thus, how do we achieve consensus-free forwarding in a decentralized control plane? The answer is via source routing. When a packet enters the network, the headend router adds a source route to the packet header and all other routers blindly follow the source route. Thus the path any given packet takes is decided by a single authoritative entity — the headend router. The headend needs no programming of paths at transit

routers, and hence no agreement to establish state. Instead, all state about the path is encoded in the packet header.

Source routing has traditionally been implemented as "loose" source routing [11, 31] in which, rather than record every node along the path, the source route records only a subset of routers or special "waypoint" along the path. This introduces some complexity, requiring either inter-node signaling (to establish waypoints [2]), or an underlay routing protocol (to establish connectivity between routers). Our approach avoids this complexity by utilizing "strict" source routing, in which the complete router-level path is enumerated in the packet header, as described below.

3.2 Design Details

A dSDN controller implements three main tasks:

(1) Learning local and global network state. dSDN requires the following local state from each router: (i) link status and utilization; (ii) attached prefixes, and (iii) aggregate traffic demands to each egress router. A dSDN controller obtains this local state from its underlying router stack by subscribing to the relevant telemetry and configuration paths via the gNMI API [8] and OpenConfig [44] data models.

A controller disseminates the above local information in the form of a "Node State Update" (NSU) message that also includes the node's ID, link IDs, and a unique sequence number. NSUs are disseminated to other routers using standard flooding. By listening to NSUs from other routers, every dSDN controller reconstructs a global view of the network topology including not just standard link status but also available link capacity, which prefixes are associated with each router, and traffic demands.

(2) Computing paths. dSDN controllers compute paths using a TE algorithm based on prior work [27] which approximates max-min fair allocations and balances short paths with maintaining high network utilization, but modified with some optimizations. Most important of these is the removal of per-service utility curves, as demand is measured in-band and is thus aggregated by (destination router, priority class) tuple. In dSDN, every router R runs TE to compute the placement of all flows in the network, from which it then selects the subset of paths that start at R for programming.

(3) Programming strict source routes. We encode source routes as stacks of labels enumerating each link to be traversed using its unique link ID learned from NSUs. We use MPLS to encode labels in the packet header, similarly to the adjacency-SID-based MPLS-SR data plane design [3] which is commonly supported today by WAN vendor hardware.

When a packet first enters the network, the headend router performs a two-stage lookup that maps from the packet's destination IP address to a source route. The first lookup table maps from the destination IP address and priority class to a unique egress router; this (prefix→egress) table is constructed using the prefix information carried in NSUs. The second lookup table maps from the egress router to a set of weighted source routes computed by TE, picking one by hashing a portion of the packet header. This two-stage lookup is standard and is supported without additional latency by the forwarding ASIC; see [11, 27] for a detailed explanation.

At intermediate routers, encapsulated packets are forwarded based on their outer label using a third MPLS forwarding table that contains static forwarding entries for the link IDs that the router

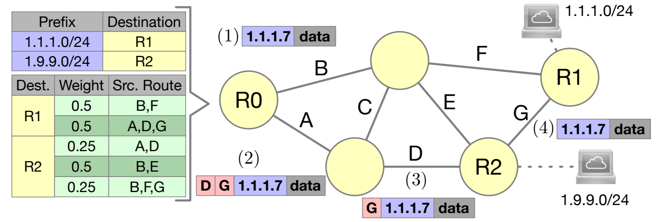


Figure 5: *The dSDN data plane from the perspective of the experience of a single packet in the network.*

advertises. This table is programmed when the dSDN controller comes up. Each router pops the outer label before forwarding the packet on. In case a failure renders a static label invalid, we use local repair paths that take the packet around the failure similar to the FRR mechanism [1] used in B4 and B2 today[1]; the invalid label is popped and a bypass source route is prepended, taking the packet to its original next hop to continue onwards as intended by the headend. This only lasts until the headend router learns of the failure and recomputes paths to avoid the failure, after which the packet will take a path that avoids that failure entirely.

Figure 5 shows a simple example of the life of a packet. When a packet destined for a host attached to $R1$ enters the network at $R0$, the forwarding hardware first maps 1.1.1.7 to $R1$ using the top table, and then picks the source route ADG by looking up $R1$ in the second table and hashing to select between the source routes to $R1$. The selected route is placed as a stack of labels A, D, and G in the packet header, then the packet is forwarded along this path with the outer label popped at each transit.

A core potential challenge of this design is the number of labels that must go in the header of the packet. This is a challenge on two operational fronts: for a path of length n, (1) the headend router must *push* n labels onto the packet, and (2) transit routers must be able to *read past* up to n labels to reach inner headers that provide entropy for effective load-sharing across multiple paths [68]. Modern routers support up to 12 labels [47] for both of these operations, which is sufficient to encode the path lengths in our current WANs. For networks with longer paths, or older hardware, we propose a *sublabel* encoding, described in Appendix §A, that compresses the source route by encoding multiple hops in a single label in a consensus-free manner.

Fault Tolerance. dSDN uses standard techniques developed in the context of IS-IS implementations to address controller crashes, bugs, failure detection, and so forth [7, 55]. This includes techniques such as: loading network state from an immediate neighbor (after a controller crash/restart), rollback (in the event of a bug), invariant checks (for malformed NSUs), preconfigured backup paths (for immediate link failure), and so forth. As with existing centralized systems, dSDN relies on the measurements that routers report to accurately reflect the true state of the system, as these observations are used as input to the TE algorithm. Increasing tolerance to byzantine failures remains an open research problem.

[1] dSDN's on-box control enables more sophisticated capacity-aware backup path selection at the affected router (*vs.* at headends as in recent work [32]): we present results in §A3 but omit them here due to space constraints.

Incremental Deployment. While we have described dSDN as a standalone or clean-slate design, we see a natural path for incremental deployment which is that we initially deploy dSDN as an alternative to existing "underlay" protocols such as IS-IS. In this model, we retain cSDN as our primary controller while dSDN acts as the backup. This requires only a software upgrade to existing routers and can be initially deployed in a single, *e.g.,* edge, region of the network. The benefit of even this first step is a better-performing underlay (since TE implements capacity-aware path selection while IS-IS does not) and a coherent architecture (since both cSDN and dSDN use similar operator-defined logic). In the next stage of deployment, one might reverse the role of cSDN and dSDN, with cSDN-programmed routes only used as backup and ultimately leveraging a streamlined form of cSDN infrastructure primarily for monitoring and management purposes, rather than on the critical path for control decisions.

Upgrades. dSDN assumes each controller solves the global TE problem consistently, but since the controller code is operator-defined, we also expect that deployed dSDN code will be updated more frequently than vendor code. This raises the question of how different versions of dSDN's TE algorithm can coexist in the network, for example during rollouts of software updates. In our existing cSDN infrastructure we find that updates qualitatively changing the TE algorithm are far less common than those that change other internals of the controller (*e.g.,* improving efficiency, adding new functionality, *etc.*). To support arbitrarily extending dSDN functionality, exchanging additional information can be done by opaquely extending the NSU with additional custom fields, similarly to how IS-IS is extended to carry arbitrary additional information via TLVs [39].

When updates to the algorithm do need to occur, we note that source routing ensures forwarding *correctness* is maintained regardless – packets will always take the path decided by the headend, and thus be loop-free. The principal concern is rather congestion during the upgrade process due to the upgraded and old controllers "mispredicting" each other's traffic placement. We anticipate that algorithm designers will evaluate this via simulation or emulation before deployment of changes. If such congestion is of concern, network operators can allow controllers to account for what algorithm each other controller is using in their solver. For example, in a network with three routers, if router A places traffic using capacity-oblivious shortest-path while B and C use a TE algorithm, routers B and C can first compute the paths A will place its traffic on, then run their TE algorithm for the remaining traffic placement. The information of which algorithm each router is using can be included in the flooded NSUs. Alternatively, existing techniques such as carefully ordering upgrades or leaving scratch space to absorb such congestion [25, 31] can be used. We leave an in-depth evaluation of this to future work.

3.3 Implementation

We prototyped the above design in approximately 22,000 lines of Go for the controller itself and 6,800 lines of C++ for the TE algorithm implementation, and have been running this prototype on production-grade Arista routers in our test lab. Figure 6 shows the prototype's system architecture.

System Modularity. We split the TE solver and controller into independent containers, with the former exposing a Solve API that

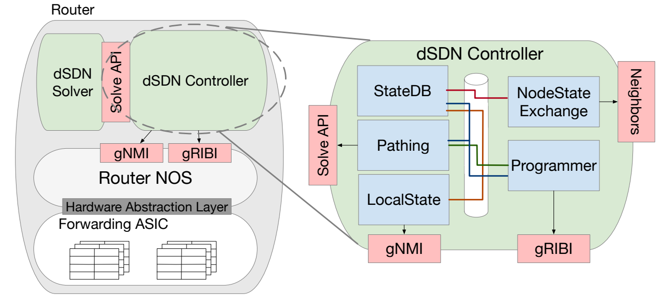


Figure 6: *dSDN implementation architecture.*

takes the network state and demands and returns paths. This separation allows the TE algorithm to be easily replaced, implemented in a different language, or even migrated off-box onto an adjacent server if further computational resources are required.

The controller itself is modular, with standalone components that communicate via pub-sub bus as shown in Figure 6. Communication with other dSDN nodes, including neighbor discovery and flooding NSUs, is handled by the NodeStateExchange module. The StateDB module combines this stream of external updates with local system readings taken by the LocalState module to produce a global network view in what we call the NodeStateDB. The Pathing module uses this view to compute a solution by calling the TE Solver container, and the Programmer uses this solution to program paths into the hardware's forwarding tables. Additional supporting modules provide interfaces for monitoring internal state, debugging, and configuration purposes.

gRPC Communication. The dSDN controller uses gRPC for all external communication; gRPC entirely abstracts the data layer (*e.g.,* packet size management, data chunking) and ensures reliable transfer [61]. To avoid requiring an address-discovery system, dSDN further relies on IPv6 link local addressing and establishes a well-known dSDN port. In contrast to protocols like IS-IS, this design allows dSDN to cleanly isolate routing from communication details.

4 Consensus-Free Convergence

dSDN combines well-known techniques (flooding, source routing, TE, *etc.*) into a novel synthesis. As a result, convergence in dSDN plays out differently *vs.* both cSDN and traditional protocol-based networks. The process by which a network converges after an event — *e.g.,* failure, change in link capacity or traffic demand — impacts its performance and hence, as a precursor to our evaluation in §5, we briefly review cSDN and dSDN's convergence behaviors.

A network's convergence time, T_{conv}, consists of three components that manifest differently in cSDN *vs.* dSDN:

(1) Propagation Time (T_{prop}) is the time from when an event occurs to when the TE controller in question learns about the event. In cSDN, link state traverses the cSDN control infrastructure, consisting of both hardware (CPN, servers) and software (operating system components, and services such as topology discovery, *etc.*), up to a single centralized controller over a time period T_{prop}. In dSDN, NSUs traverse the data plane, with a different $T_{prop}(i)$ for when each router R_i learns of the changed state.

(2) Computation Time (T_{comp}) is the time it takes a controller to run a TE computation with the new event information and generate updated paths. In cSDN, a single central controller runs this

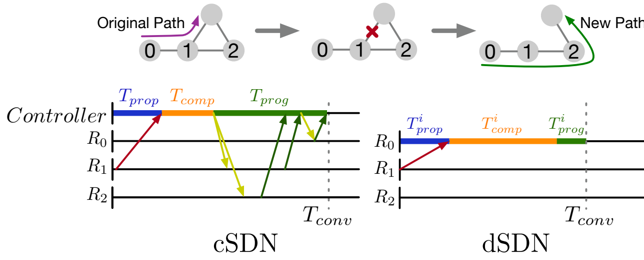


Figure 7: *Convergence behavior for a single path for which R_0 is the headend when a network event occurs, such as a link down reported by R_1. Arrows represent communication: state notification (red), programming messages (yellow), and acks (green).*

computation over time T_{comp}. In dSDN, each router runs the TE computation over time $T_{comp}(i)$ per router R_i, with a start time dependent on $T_{prop}(i)$. We generally expect similar $T_{comp}(i)$ across routers.

(3) Programming Time (T_{prog}) is the time to install computed paths at all routers. cSDN implementations typically implement this by programming forwarding rules at each router [11, 25, 27], which requires care in programming order as naively one can end up with loops or dead ends when some routers in the path are updated while others are not. The commonly deployed solution for this is a two-phase programming process to properly make-before-break paths. All paths are programmed in parallel. For each path of length n, first (a) all $n-1$ transit routers for the path are programmed in parallel with their next hop. As they finish programming, they (b) send acknowledgements back to the cSDN server. Upon receiving all $n-1$ acknowledgements, (c) the cSDN server sends a command to enable the new path and disable the old one at the headend router. As mentioned in §2.2, each step in this process often requires going through a hierarchy of layers [13]. By contrast, in dSDN, programming is an entirely local process as the controller locally programs only the paths it originates, and $T_{prog}(i)$ is when all paths originating from router R_i are established.

We see the combined effect in Figure 7, which shows the sequence of messages sent and processed for a single path during convergence after a network event in cSDN *vs.* dSDN. The full view of convergence would show the controller programming all paths in cSDN, and every router in dSDN receiving the event notification and recomputing and reprogramming the paths it originates accordingly.

Some final observations: first, since we run the same TE algorithm in both cSDN and dSDN, their routes after convergence are identical. Second, both cSDN and dSDN experience incremental convergence across paths; that is, neither cSDN nor dSDN enjoy simultaneous convergence. Rather, in both, different routers learn their new paths at different times. The reason for this "drift" is different for cSDN vs dSDN; in cSDN, it is due to distributed programming, as the controller programs all paths in parallel. In dSDN, it is because headends (and hence the paths they compute) converge independently. In this sense, dSDN's architecture does not introduce a fundamentally different convergence behavior.

5 Evaluation

The primary benefit of dSDN is simplification. Quantifying system simplicity/complexity is notoriously difficult [4, 10, 43] hence, for now, we offer only anecdotal evidence: from Figure 3, dSDN eliminates the vast majority of the components shown in green and blue, retaining only: (i) a form of IS-IS for NSU flooding, (ii) the TE component though moved to run on-box, (iii) a container management module that runs on-box. In terms of code size, dSDN represents well over an order of magnitude reduction in lines of code for equivalent functionality.[2] Nonetheless, we recognize that the true value of dSDN's simplification will only be apparent if, after years of deployment experience, we see a reduction in major outages. We hope to report on this in future work.

While we cannot assess dSDN's impact on complex failures, we can evaluate it using the standard metrics from SIGCOMM papers: scalability, repair time after link/node failures (corresponding to the "minor" outages in Figure 1), packet loss, and so forth. This is our focus in this section. In other words, while dSDN is designed to improve hard-to-quantify goals such as simplicity, we aim to show that dSDN also fares well on standard metrics.

Thus our evaluation in this section focuses on two high level questions. First, how does routing in dSDN compare to cSDN? To a first order, the quality of a routing solution is determined by the paths it selects. Picking low-cost paths that avoid congestion and packet loss is the purview of the TE algorithm and, since cSDN and dSDN run the same TE algorithm, their performance in this regard will be identical. Beyond path selection, a routing solution's performance is determined by its convergence behavior: when a network event occurs, a good routing solution must update its paths quickly and with minimal packet loss. We thus evaluate the time it takes a solution to converge after a network event (*convergence time*) and the impact in packet loss during this convergence period (*transient impact*).

The second high level question we evaluate is whether router CPUs can handle the computation load that dSDN places on them. This is important as dSDN moves TE computation from resource-rich datacenter machines to resource constrained routers, and hence we must show that this trade-off is practical.

5.1 Convergence Time

Metric. We define convergence time as the time from when a network event occurs to when new routes reflecting the event are installed at all routers. This is a standard metric used in prior work and captures tail behavior experienced by traffic [18, 20, 35].

Methodology. Convergence time consists of 3 periods — T_{prop}, T_{comp}, and T_{prog} — which we measure in this section. For cSDN, we derive these measurements from our production network B4, with TE running on datacenter servers. While this network is not exactly identical to a dSDN deployment as its individual routers are themselves made up of a Clos topology of switches, it provides a real production convergence time comparison. As dSDN is not deployed in production, we approximate its equivalent performance as follows. For dSDN's T_{prop}, we measure the propagation time of IS-IS link-state update messages in B4. We believe this is a good

[2] The above discussion ignores the many systems used for monitoring, configuration, and maintenance which we (perhaps conservatively) assume will remain similar in complexity across cSDN and dSDN.

approximation since dSDN's NSU are propagated inline in a manner similar to IS-IS updates.[3]

For T_{comp} and T_{prop}, as computation and programming operations in dSDN are entirely *local* to a router, we evaluate these times by running our dSDN implementation on an Arista 7808 router. We view this as a reasonable estimate of performance in production because: (i) our prototype is production grade, (ii) our test router is identical to routers deployed in our WAN, and (iii) the test workloads (topologies, network events, traffic demands, *etc.*) are drawn from B4, reflecting our cSDN measurements.

For completeness, we also compare dSDN's performance to RSVP-TE since the latter is how capacity-aware routing is typically implemented without SDN, including in our B2 network and elsewhere.

Data. The T_{prop} distribution for cSDN reflects 4,758 link failure events over a period of 18 months; for dSDN it reflects 33,000 IS-IS link-state events over a period of 27 months. The T_{comp} distribution is over 1,000 TE computations on a server-class machine for cSDN and an Arista router for dSDN. Finally, T_{prog} in cSDN reflects 26,723 programming events from a 10 day period, and in dSDN it reflects the 1,000 paths computed when measuring T_{comp}.

5.1.1 Results: dSDN *vs.* cSDN in B4.

Figure 8 shows T_{prop}, T_{comp}, and T_{prog} for cSDN and dSDN for our B4 network which has O(100) nodes and O(10k) demands. Noting the varying Y-axis scales for each figure and, in particular, the log scale Y-axis for T_{prop} and T_{prog}, we make the following observations from these results:

(1) Average T_{prop} is ~20x lower in dSDN than in cSDN. This difference arises because updates in cSDN traverse a hierarchy of software services across the router, edge controller, and collection infrastructure from edge to top-level datacenters, while NSUs propagate inband through routers' forwarding planes. Though the cSDN time could likely be optimized, we speculate that dSDN's in-band forwarding via router hardware is close to the minimal T_{prop} that can be achieved.

(2) Average T_{comp} is ~35% higher in dSDN than in cSDN. Since cSDN and dSDN run the same TE algorithm on the same inputs, this gap stems primarily from the slower CPUs on the Arista router: our dSDN controller runs on three 1.9GHz cores, while the cSDN controller runs on forty 2.8GHz cores. We evaluate this further in §5.3.

(3) Average T_{prog} is ~1000x lower in dSDN than in cSDN. This difference is to be expected since programming in dSDN is a router-local and single-step operation, while cSDN requires a two-phase network-wide programming process across all routers (§4). The *magnitude* of the difference is accounted for by tail effects in cSDN; completing the programming of a path is gated by the slowest response from any router on that path, and network-wide convergence is gated by the slowest path. (See Appendix §B for a breakdown of cSDN T_{prog}.)

These results raise an orthogonal question: can we optimize cSDN's convergence time by applying dSDN's source routing to cSDN? To our knowledge, no existing WAN uses strict source routing in this manner but we believe this is a promising optimization. Nonetheless, even with this optimization, T_{prog} in cSDN remains a non-local operation and hence higher than in dSDN.

(4) Overall convergence time: summing across components we see that dSDN's convergence time is 120-150x faster than cSDN. Our goal with dSDN was to achieve simplification without cost to routing performance. Our results show that not only is dSDN's convergence time well within current targets, it significantly outperforms cSDN.

[3] NSUs are larger that IS-IS messages as they contain additional demand information however this adds little to T_{prop}; for a 200-node network with 5 traffic classes, demand adds 4KB per router in the worst case (all-pairs demand), which adds less than 4 μs of transmission time on a 10 Gbps link.

5.1.2 Results: dSDN *vs.* RSVP-TE in B2.

Figure 9 shows production measurements of the convergence time in B2, our protocol-based WAN running RSVP-TE for capacity-aware routing (§2.1). We compare this to dSDN's convergence time computed as in §5.1.1 but now with B2's demand and topology as input.

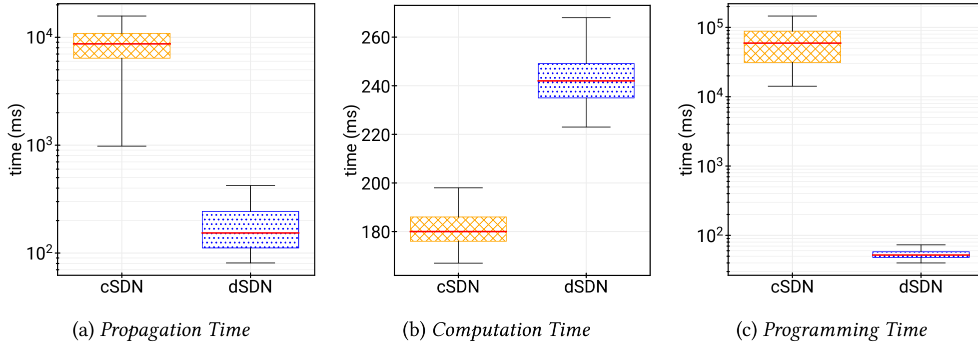
We observe that RSVP-TE achieves a median convergence time of 45.5s and particularly poor tail behavior. This is because when a link cut occurs, all the routers that act as the headend for a path traversing the failed link simultaneously (but independently) race to restore the affected paths, leading to a signaling "stampede." In our experience, and matching prior reports [11], RSVP-TE can take 10+ minutes to reconverge from the largest outages. In comparison, we see that dSDN achieves a median convergence time of approximately 29.8s seconds with much lower variance.

We make two additional observations. First, while Figure 9 focuses on convergence time, it is important to remember that dSDN (and indeed, cSDN) significantly outperforms RSVP-TE in the quality of paths it computes and hence the network utilization it achieves. Prior work reports up to a 60% increase in network utilization by replacing RSVP-TE with centralized TE [25]. Second, we note that dSDN's convergence time on B2 is significantly higher than dSDN's convergence time on B4, which is almost entirely due to a longer T_{comp} because of B2's larger topology and demands; we explore scalability further in §5.3.

5.2 Transient Impact

Metric. Convergence time alone does not capture the performance impact on traffic carried by the network during the convergence period. Consider two alternate scenarios for a flow F during convergence. In one, F experiences 1% packet loss for 100 seconds, and in the other, F experiences 10% packet loss for 10s. The preferred scenario depends on the operator's service-level objective (SLO) threshold; with the SLO set at 5% loss, the first scenario is better as there is no SLO violation, but set at 0.5% the second scenario is preferable as the SLO is violated for a shorter period. A further consideration is priority or class of traffic; SLO violations for high priority traffic would be deemed worse than for lower priorities.

Reflecting these considerations, we measure transient impact with *bad seconds* which captures both input duration *and* magnitude, which we obtain as follows. We group flows by their priority classes, source metro, and destination metro area. Each of these flow groups has an SLO threshold based on priority class: 99.99% (*i.e.,* four 9s, <0.01% loss) for the highest, and one "9" less for each subsequent lower priority class. We say that a flow group violates its SLO if more than 5% of flows suffer traffic loss beyond their threshold. At



(a) *Propagation Time* (b) *Computation Time* (c) *Programming Time*

Figure 8: *Individual components of convergence time in B4 for cSDN vs. dSDN.*

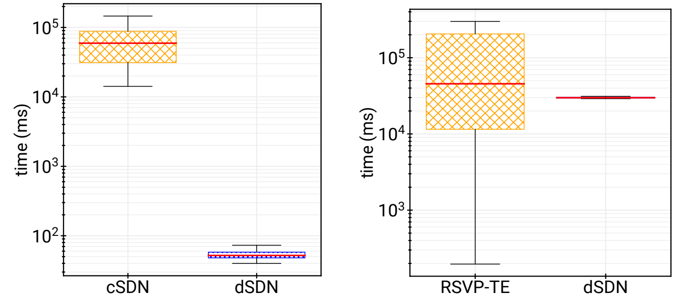


Figure 9: *Total convergence time in B2 for RSVP-TE vs. dSDN.*

any point in time, we define the *blast radius* as the fraction of all flow groups that violate their SLO, and integrate blast radius over the convergence period (from t_s to t_e in Equation 2) to obtain *bad seconds* as our metric for a network event's performance impact. As an example, if for 100 flow groups converging over a 10 second period, 50 violate their SLO for 5 seconds, then 10 of these continue to violate their SLO for another 5 seconds, this totals $\frac{50}{100}*5+\frac{10}{100}*5 =$ 3 bad seconds. This approach is summarized in Equations 1 and 2 and reflects internal best-practice for how we monitor WAN performance.

$$\text{blast radius} = \frac{\text{\# of flows violating SLO}}{\text{total \# of flows}} \quad (1)$$

$$\text{bad seconds} = \int_{t=t_s}^{t_e} (\text{blast radius})\, dt \quad (2)$$

Note that this implicitly captures the impact of differing convergence times, as an equally "bad" outage that lasts longer will incur a higher number of bad seconds.

Methodology. As transient impact depends on the detailed timing of control events and traffic arrivals at each router, we turn to simulation to measure bad seconds since we do not have a large-scale dSDN deployment. For this, we extend an in-house event-driven network simulator used for capacity planning in our WAN to capture transient convergence behavior. The simulator ingests topology, traffic demand, and failure and repair information logged from our production WAN and replays the same within the simulation, modeling hop-by-hop message propagation over a data plane network at the granularity of packets or flows. We run all simulations for 1,000 days of events.

Our simulator models the distributed behavior of dSDN as follows. Given a failure event, NSUs are propagated with a per-hop propagation time determined by each link's IGP metric.[4] Upon receipt, a router runs TE for a duration we obtain by sampling from the distribution of T_{comp} from §5.1.1. Programming time at that router is similarly obtained by sampling from the distribution of our prototype measurements of T_{prog}. Throughout this process, data traffic continues to flow through the simulated network and we measure the per-flow loss at each router over time, from which we calculate bad seconds.

[4] We observe this to be consistent with measured IS-IS propagation times.

Since our simulator does not model individual components of the cSDN control infrastructure, we simulate cSDN as follows. Corresponding to each link failure event, we model T_{prop} and T_{comp} by sampling from the measured distributions of these times in production (§5.1.1). We simulate each path's programming time by sampling from two separate distributions corresponding to the two phases of programming: one for the time to program all transit routers, and another for subsequently enabling the path at the headend router.

For calibration, we also simulate a hypothetical "omniscient" protocol that converges instantly: *i.e.,* new paths are installed with zero delay after a network event. Any packet loss in this case is because the network has insufficient capacity to handle the offered demand. Thus, comparing the bad seconds in cSDN/dSDN *vs.* in the omniscient protocol allows us to clearly identify the portion of bad seconds that is due to convergence effects *vs.* the fundamental decrease in network capacity due to failure.

Finally, we note that in practice cSDN uses predefined bypass paths [1], and dSDN will do the same. Here, we show results without bypass paths in effect, since bypass paths can mask the impact of poor convergence behavior. We show results with bypass paths in Appendix §D, and confirm that these remain qualitatively unchanged.

Results. Figure 10 shows the bad seconds in cSDN, dSDN, and our omniscient protocol for our highest, intermediate, and lowest priority traffic classes. We see that, as we move from higher to lower priority classes, bad seconds increases for both cSDN and dSDN. This is to be expected because TE prioritizes allocating paths to high priority flows and hence the impact of capacity loss is greater on lower priority traffic. We also see that the bad seconds in our omniscient protocol range from zero (at our highest priority) to low (under 1.98 at the lowest priority) showing that the vast majority of the bad seconds in cSDN and dSDN are due to their convergence behavior rather than any fundamental loss in network capacity.

Finally, we observe that in all cases, the number of bad seconds is significantly lower in dSDN *vs.* cSDN. For example, at the 98%ile, high priority traffic in cSDN experiences 146.9 bad seconds compared to 2.23 in dSDN, while for lower priority traffic the bad seconds are 1122.9 vs 57.3 respectively. This is due to dSDN's faster convergence time: dSDN more quickly moves traffic from a failed/congested path to a better path, and hence experiences loss for a shorter period of time.

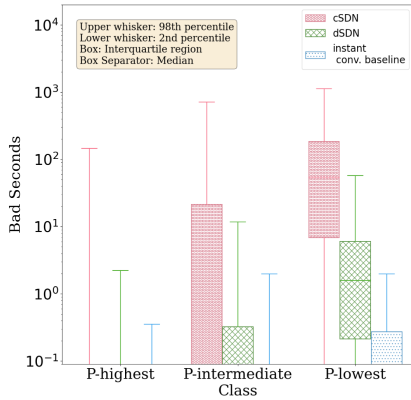


Figure 10: *Bad Seconds distribution for cSDN and dSDN, for different traffic classes.*

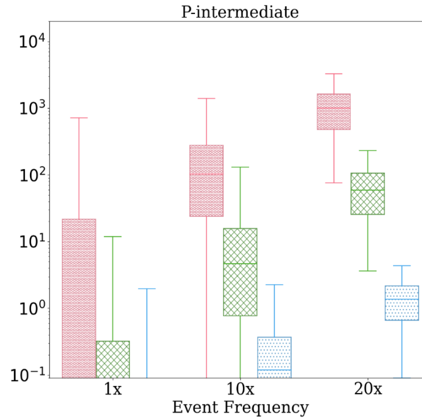


Figure 11: *For traffic of intermediate priority, bad seconds distribution with churn rate (frequency of events) 10x and 20x.*

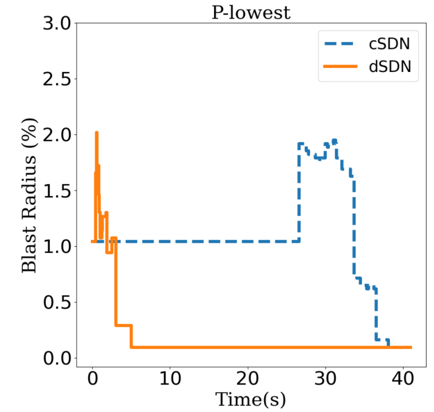


Figure 12: *Timeline of blast radius (% of impacted flows) in the lowest priority class, for a selected failure event.*

Figure 12 shows the impact timeline of a single failure within our simulator. In dSDN, each headend converges independently and we see this in the timeline: spikes in loss occur intermittently during convergence due to congestion as some traffic moves onto links on which other traffic still remains, falling as more and more headends reprogram their paths. A similar pattern is seen in cSDN, albeit with an increased repair timeline.

Increasing Churn. While the above results show that dSDN fares well under realistic failure scenarios, we want to verify that it continues to do so under more pessimistic assumptions. We thus repeat our simulations while artificially increasing the rate of failures, which we call the churn rate. Figure 11 shows the bad seconds for the intermediate priority class when increasing churn by a factor of 10x and 20x from the baseline failure rate. At these higher churn rates, it is often the case that one event occurs while the network is still converging after a previous event. We see that, as expected, with a higher churn rate, impact per event grows but dSDN continues to perform better than cSDN. For example, in the P-intermediate class, for a 10x (20x) higher churn rate, the median bad seconds for cSDN is about 22x (17x) that of dSDN.

In summary, we conclude that dSDN's decentralized approach to convergence does not lead to worse performance. On the contrary, dSDN typically achieves lower convergence time and incurs fewer bad seconds relative to our current implementation of cSDN.

5.3 Scalability

Our results above show dSDN's simplifications do not impact — and in fact, improve on — the performance of our current network. However, they do reveal a trade-off: on B4, while dSDN's T_{prop} and T_{prog} are considerably faster than cSDN, the T_{comp} portion is 35% longer because of the fundamentally more constrained CPU resources on the router. While this trade-off works in our current network, we must evaluate whether it might become problematic as the network evolves.

Methodology To answer this question, we focus on our larger (less abstracted) B2 topology as it is the more computationally challenging case for TE: relative to B4, our B2 network has 30x more flows across approximately 6x more nodes and 10x more links. As our performance target for dSDN, we pick the average convergence time that B2 achieves today in production using RSVP-TE since this is clearly an acceptable level of performance in practice: 106.6s as reported in §5.1.2. Figure 9 showed that dSDN's average convergence time on B2's topology and demand was 29.8s, well below our target of 106.6s. On B2 data, dSDN's T_{prop} and T_{prog} are O(100ms) and hence T_{comp} is by far the largest contributor to this overall convergence time. As such, going forward, we ignore T_{prop} and T_{prog} and focus on T_{comp}. While this section focuses primarily on B2, we also consider additional internal and external topologies, including B4, later in this section.

We evaluate dSDN router performance by running our dSDN prototype on our Arista test router (with six 1.9GHz cores), and for comparison with cSDN, running TE on a data center server (with forty 2.8GHz cores).

5.3.1 Results

Impact of available CPU capacity on T_{comp}. Figure 13 plots how T_{comp} is affected as we increase the number of cores available to the dSDN controller and the datacenter solver. We make a few important observations. First, T_{comp} even with just a single core is well under our threshold of 106 seconds. Second, comparing T_{comp} on the router *vs.* server, we see that faster cores improve T_{comp} by as much as 41%. Third, while we see an initial improvement in T_{comp} by adding more cores, this improvement flattens out at approximately 5 cores on both the router and server. This flattening is because of the limits to parallelization of our current TE algorithm that serializes on the final step in flow assignment.[5] This tells us that selecting routers with higher speed CPUs, without necessarily changing the number of CPUs per router, will improve dSDN's performance. Since a router's control CPU is a very small portion of the overall power and cost budget of the router, we view this as a modest and practical change that yields valuable improvements in T_{comp}.

Impact of network evolution on T_{comp}. Next, we examine how T_{comp} scales as the network evolves. In general, TE runtime increases with network size, the number of flow demands, and the difficulty of allocating flows [11, 34, 49]: with decreasing available

[5] To our knowledge, this is true of current TE algorithms in the literature [11, 25, 27]. An open question is whether there are better parallel algorithms for TE that one might leverage.

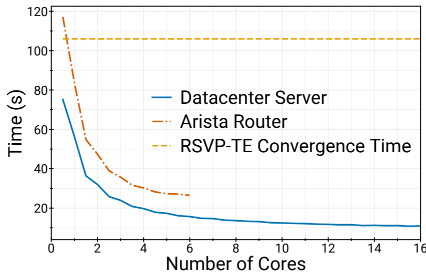


Figure 13: *T_{comp} under varying numbers of cores running TE for B2, truncating at 16 cores for legibility as the graph remains flat up to the max 40 cores.*

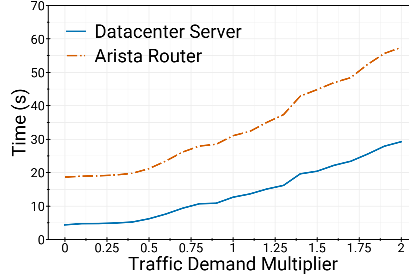


Figure 14: *T_{comp} for B2 with increasing traffic demand sizes on a static topology.*

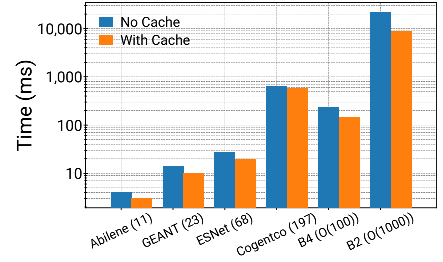


Figure 15: *dSDN T_{comp} on different networks, with and without caching. The number of nodes is in parentheses. Note the log y axis.*

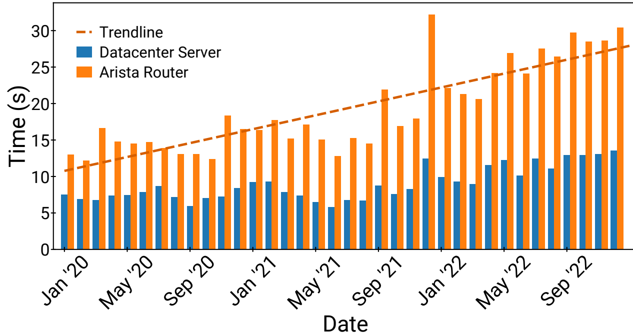


Figure 16: *dSDN Runtime on B2 as it has grown in size over a period of three years, approaching 1000 nodes at the rightmost bar in the figure.*

capacity, the TE algorithm requires additional iterations to find viable paths.

Thus, for our first "stress test", we repeat the above experiments using our B2 topology while artificially scaling the traffic demand by a constant multiplier, representing demand growth in a constant-sized network. We set the number of router cores available to dSDN to 4 in these experiments, guaranteeing at least 2 cores are available for other router control plane use. Figure 14 shows the resultant T_{comp}; we see that runtime grows roughly linearly with increasing demands but remains under our threshold of 106.6 seconds even if our current demands were to double.

A more realistic evolution would simultaneously scale both the demands and topology (capturing the addition of new links and increased link capacity). To model such evolution, we look at historical snapshots of B2's demand and topology over a 3-year period. Using the same experimental setup as above, we measure T_{comp} for each snapshot to obtain the trend in T_{comp} shown in Figure 16. Under the assumption that future growth will follow historical trends, we can extrapolate the scaling trend from Figure 16 to when T_{comp} intersects our threshold of 106.6s. We find that dSDN can perform within our performance target for nearly 15 years into the future. We note that this is a conservative estimate, as it assumes no change to router CPUs or to RSVP-TE's convergence time over this period.

In summary, our results so far show that dSDN can significantly lower the convergence time of our current network and that, even for a large production network, router CPU resources can adequately support dSDN's computational requirements, and can do so into the future.

T_{comp} for other networks. To verify that our results hold for other networks, Figure 15, focusing on the blue bars, shows T_{comp} for a variety of topologies, including B4 (using production traffic demands) and external networks from the open TopologyZoo dataset [30] ranging in size from 11 to 197 nodes. For the external networks, we generate traffic demands using a gravity-based model [52], as done in prior work [49, 63].

Performance optimizations. While our results show that dSDN performs well even with current routers and TE implementations, we see technological and algorithmic opportunities to reduce dSDN's T_{comp}. On the technological side, Figure 13 shows the performance gains if vendors were to upgrade to higher speed cores. In addition, there are likely algorithmic optimizations that can be implemented in the near term, *e.g.,* pre-computation, incremental computation, and more aggressive parallelism. We briefly explore one such option here: pre-computing the first shortest path for each (source, destination) pair to be considered by the solver. Originally, our solver recalculated the shortest path any time available capacity changed. Instead, we now pre-compute shortest paths, and at runtime check that the path still has nonzero capacity, rerunning Djikstra's when this is not the case. This cache only needs to be recomputed if a new link is added to the topology in a network upgrade event, but remains valid for any capacity changes including full loss and restoration of an existing link's capacity. We repeat the experiments from Figure 15 with caching enabled. Our results — in orange bars on Figure 15 — indicate that caching alone can speed up computation by up to 2.5x for the largest topology. We leave exploring additional optimizations to future work.

6 Related Work

SDN in WANs. A rich body of work describes the evolution of modern hyperscalar WANs [11, 12, 14, 25–27, 32]. We view dSDN as a further evolution of SDN by moving its goals and principles back onto the router, an option that was unavailable to early SDN practitioners.

Network Complexity and Availability. Network complexity and its impact on network availability has been examined by prior work. In particular, Govindan *et al.*[19] undertake a comprehensive review of failures in Google's network and offer general guidance on improving diagnostics, management, and configuration. They highlight the pitfalls of complexity and the importance of designing for availability but do not explicitly advocate simplified designs.

Designing for Availability. Noting that "configuration and software bugs are inevitable," Krishnaswamy *et al.* [31], take a different

approach from us. Their BlastShield architecture partitions the WAN into roughly-geographical vertical slices, each with its own control infrastructure and a controller running inter- and intra-slice routing. This contains the blast radius of a failure to one slice. In a related approach, Meta's EBB [11] slices the network horizontally into 8 parallel global networks, each with independent control infrastructures. Unlike dSDN, both EBB and Blastshield still rely on external control infrastructure for each slice, and differ from dSDN on details such as the use of source-routing vs. per-hop forwarding rules. We note however that the general principle underlying EBB and Blastshield is that of *sharding* the network and could be applied to dSDN. Specifically, dSDN could run on a horizontally sharded network (akin to EBB) thus containing data plane failures to a single shard. Therefore we view sharding as orthogonal but complementary to the question of external *vs.* inband control and leave it to future work to investigate the design of a sharded dSDN network.

OneWAN [32] reports on Microsoft's efforts to unify their separate WANs. Unification can be viewed as a form of simplification that is orthogonal and complementary to dSDN's approach of removing external control infrastructure.

Onbox Controllers. Custom code on routers has been explored with different methods [9, 22, 38], running simple functions such as traffic probing, telemetry collection, and tunnel switching [11, 31, 32]. dSDN differs in that we move *all* functions of a traditional cSDN controller onto the router and do so on standard vendor routers. The Path Computation Element (PCE) [54] working group of the IETF has been working on enabling the extraction of the computation portion of traditional algorithms into a self-contained module that can be moved *off* of the router, in contrast to dSDN which moves centralized computation *onto* routers.

Source Routing. Many papers propose creative uses of source-routing – *e.g.,* pathlet routing [17] constructs paths by concatenating smaller "pathlets" that are disseminated in-band, while FCP [36] marries source-routes with failure annotations for guaranteed packet delivery, and industry solutions propose schemes for "loose" source routing using waypoints [15] for improved scalability. dSDN's main contribution is the context in which we apply source routing and a practical approach to implementing strict source routing.

TE. There is extensive literature on TE solutions [11, 27, 34, 45, 46, 53, 59]. Previous work has also examined decomposing TE algorithms into domain-local [31] and node-local [50] formulations, which we do not do here — we run TE with a global view of the network state. dSDN does not innovate on the TE algorithm itself but rather on how we *deploy* it.

7 Conclusion

The placement of functionality is a fundamental decision when architecting a network, impacting both its technical solutions and who commands innovation. The architecture of network control planes has evolved from decentralized and inband control planes that favor vendor-driven innovation, to SDN's centralized and external control plane that favors operator-driven innovation. The contribution of our paper is in demonstrating the feasibility of a new design point, one based on decentralized and inband control but that favors operator-driven innovation. We believe this design point allows us to retain the benefits of SDN, while simplifying network infrastructure and hence improving availability. We view dSDN as a first step that we hope enables future investigations into the technical solutions and implications of this new design point.

Our work raises no ethical concerns.

Acknowledgments

We thank the anonymous reviewers and our shepherd James Hongyi Zeng for their insightful comments and helpful feedback. We also thank our colleagues at Google, including Jon Mitchell, Marcus Hines, Neha Manjunath, Chi-Yao Hong, Brad Morrey, Douglas Browning, Tyler Kurtz, Devon Hollowood, Lars Prehn, Hank Levy, David Culler, Brent Stephens, Christophe Diot, Amin Vahdat, Louie Plissonneau, Jayaram Mudigonda, Vasileios Pappas and others for their discussions and contributions to the development of dSDN.

References

[1] Alia Atlas, George Swallow, and Ping Pan. 2005. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090. (May 2005). https://doi.org/10.17487/RFC4090

[2] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Dr. Vijay Srinivasan, and George Swallow. 2001. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209. (Dec. 2001). https://doi.org/10.17487/RFC3209

[3] Ahmed Bashandy, Clarence Filsfils, Stefano Previdi, Bruno Decraene, Stephane Litkowski, and Rob Shakir. 2019. Segment Routing with the MPLS Data Plane. RFC 8660. (Dec. 2019). https://doi.org/10.17487/RFC8660

[4] Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, USA, 335–348.

[5] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: towards an open, distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2620728.2620744

[6] Robert T. Braden, Lixia Zhang, Steven Berson, Shai Herzog, and Sugih Jamin. 1997. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205. (Sept. 1997). https://doi.org/10.17487/RFC2205

[7] Ross Callon. 1990. Use of OSI IS-IS for routing in TCP/IP and dual environments. RFC 1195. (Dec. 1990). https://doi.org/10.17487/RFC1195

[8] Carl Lebsack, Marcus Hines, Paul Borman, Anees Shaikh, Rob Shakir, Wen Bo Li, et al. 2018. gNMI - gRPC Network Management Interface. https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md. (Jun 2018).

[9] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. FBOSS: Building Switch Software at Scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 342–356.

[10] Byung-Gon Chun, Sylvia Ratnasamy, and Eddie Kohler. 2008. NetComplex: A Complexity Metric for Networked System Designs. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, USA, 393–406.

[11] Marek Denis, Yuanjun Yao, Ashley Hatch, Qin Zhang, Chiun Lin Lim, Shuqiang Zhang, Kyle Sugrue, Henry Kwok, Mikel Jimenez Fernandez, Petr Lapukhov, et al. 2023. EBB: Reliable and evolvable express backbone network in meta. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 346–359.

[12] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: an intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 87–98.

[13] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. 2021. Orion: Google's Software-Defined Networking Control Plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 83–98. https://www.usenix.org/conference/nsdi21/presentation/ferguson

[14] Mikel Jimenez Fernandez and Henry Kwok. 2017. Building Express Backbone: Facebook's new long-haul network. *Engineering at Meta* (May 2017).

[15] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. 2018. Segment Routing Architecture. RFC 8402. (July 2018). https://doi.org/10.17487/RFC8402

[16] Tony Fyler. 2023. Azure Outage Disconnects Thousands. https://techhq.com/2023/01/azure-outage-disconnects-thousands. (Jan. 2023). Accessed: 2024-1-30.

[17] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. 2009. Pathlet Routing. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/1592568.1592583

[18] Deepthi Gopi, Samuel Cheng, and Robert Huck. 2017. Comparative analysis of SDN and conventional networks using routing protocols. In *2017 International Conference on Computer, Information and Telecommunication Systems (CITS)*. 108–112. https://doi.org/10.1109/CITS.2017.8035305

[19] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 58–72. https://doi.org/10.1145/2934872.2934891

[20] Timothy G Griffin and Brian J Premore. 2001. An experimental analysis of BGP convergence time. In *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*. IEEE, 53–61.

[21] Timothy G Griffin and Gordon Wilfong. 1999. An analysis of BGP convergence properties. *SIGCOMM Comput. Commun. Rev.* 29, 4 (Aug. 1999), 277–288.

[22] Saif Hasan, Petr Lapukhov, Anuj Madan, and Omar Baldonado. 2017. Open/R: Open Routing for Modern Networks. https://engineering.fb.com/2017/11/15/connectivity/open-r-open-routing-for-modern-networks/. *Engineering at Meta* (Nov. 2017).

[23] Daniel Hertzberg. 2018. *Docker Containers on Arista EOS*. Technical Report. Arista.

[24] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, USA, 295–308.

[25] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, and Mohan Nanduri. 2013. *Achieving High Utilization with Software-Driven WAN*. Technical Report MSR-TR-2013-54. https://www.microsoft.com/en-us/research/publication/achieving-high-utilization-with-software-driven-wan/

[26] Chi-Yao Hong, Subhasree Mandal, Mohammad A. Alfares, Min Zhu, Rich Alimi, Kondapa Naidu Bollineni, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Jeffrey Liang, Kirill Mendelev, Steve Padgett, Faro Thomas Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jon Zolla, Joon Ong, and Amin Vahdat. 2018. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN. In *SIGCOMM'18*. https://conferences.sigcomm.org/sigcomm/2018/program_tuesday.html

[27] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally Deployed Software Defined WAN. In *Proceedings of the ACM SIGCOMM Conference*. Hong Kong, China. http://cseweb.ucsd.edu/~vahdat/papers/b4-sigcomm13.pdf

[28] Santosh Janardhan. 2021. Details About The October 4 Outage. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/. *Engineering at Meta* (Oct. 2021). Accessed: 2024-1-30.

[29] Juniper. 2021. Running Third-Party Applications in Containers. In *Introducing Junos OS Evolved*. Juniper, 84–88.

[30] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE J. Sel. Areas Commun.* 29, 9 (Oct. 2011), 1765–1775.

[31] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. 2022. *Decentralized cloud wide-area network traffic engineering with BlastShield*. Technical Report. Microsoft. 325–338 pages.

[32] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, Himanshu Raj, Luis Irun-Briz, Jamie Gaudette, and Erica Lan. 2023. OneWAN is better than two: Unifying a split WAN architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 515–529. https://www.usenix.org/conference/nsdi23/presentation/krishnaswamy

[33] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[34] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 157–170. https://www.usenix.org/conference/nsdi18/presentation/kumar

[35] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. 2001. Delayed Internet routing convergence. *IEEE/ACM transactions on networking* 9, 3 (2001), 293–306.

[36] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. 2007. Achieving Convergence-Free Routing Using Failure-Carrying Packets. *SIGCOMM Comput. Commun. Rev.* 37, 4 (aug 2007), 241–252. https://doi.org/10.1145/1282427.1282408

[37] Frederic Lardinois. 2020. IBM Cloud suffers prolonged outage. *TechCrunch* (June 2020).

[38] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. 2013. SoNIC: Precise Realtime Software Access and Control of Wired Networks. *NSDI* (April 2013).

[39] Tony Li and Henk Smit. 2008. IS-IS Extensions for Traffic Engineering. RFC 5305. (Oct. 2008). https://doi.org/10.17487/RFC5305

[40] Marcus Hines, Rob Shakir, Sam Ribeiro, Eric Breverman, et al. 2018. gNOI - gRPC Network Operations Interface. https://github.com/openconfig/gnoi. (Dec 2018).

[41] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (Mar 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[42] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[43] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, USA, 305–320.

[44] OpenConfig Project. 2015. OpenConfig. https://www.openconfig.net/. (2015).

[45] Konstantina Papagiannaki, Nina Taft, and Anukool Lakhina. 2004. A Distributed Approach to Measure IP Traffic Matrices. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement (IMC '04)*. Association for Computing Machinery, New York, NY, USA, 161–174. https://doi.org/10.1145/1028788.1028808

[46] K. Papagiannakit, N. Taft, and C. Diot. 2004. Impact of flow dynamics on traffic engineering design principles. In *IEEE INFOCOM 2004*, Vol. 4. 2295–2306 vol.4. https://doi.org/10.1109/INFCOM.2004.1354652

[47] Chetan Patel. 2022. *MPLS 12-Label Push*. Technical Report. Arista.

[48] Abhinav Pathak, Ming Zhang, Y Charlie Hu, Ratul Mahajan, and Dave Maltz. 2011. Latency inflation with MPLS-based traffic engineering. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC '11)*. Association for Computing Machinery, New York, NY, USA, 463–472.

[49] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. 2023. {DOTE}: Rethinking (Predictive) {WAN} Traffic Engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1557–1581.

[50] Alejandro Ribeiro, Nikolaos D Sidiropoulos, and Georgios B Giannakis. 2008. Optimal Distributed Stochastic Routing Algorithms for Wireless Multihop Networks. *IEEE Trans. Wireless Commun.* 7, 11 (Nov. 2008), 4261–4272.

[51] Rob Shakir, Xiao Wang, Nathaniel Flath, et al. 2017. gRIBI - gRPC Routing Information Base Interface. https://github.com/openconfig/gribi. (jul 2017).

[52] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. 2002. Experience in measuring backbone traffic variability: models, metrics, measurements and meaning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment (IMW '02)*. Association for Computing Machinery, New York, NY, USA, 91–92. https://doi.org/10.1145/637201.637213

[53] Matthew Roughan, Mikkel Thorup, and Yin Zhang. 2003. Traffic Engineering with Estimated Traffic Matrices. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement (IMC '03)*. Association for Computing Machinery, New York, NY, USA, 248–258. https://doi.org/10.1145/948205.948237

[54] Julien Meuric Scudder, Dhruv Dhody. 2023. Path Computation Element (PCE) Working Group Charter. (2023).

[55] Mike Shand and Les Ginsberg. 2008. Restart Signaling for IS-IS. RFC 5306. (Oct. 2008). https://doi.org/10.17487/RFC5306

[56] Claude E Shannon. 1949. A theorem on coloring the lines of a network. *J. Math. Phys.* 28, 1-4 (April 1949), 148–152.

[57] Adam Simpkins. 2015. *Facebook Open Switching System ("FBOSS") and Wedge in the open*. Technical Report.

[58] Richard Speed. 2021. AWS runs into IT Problems. https://www.theregister.com/2021/12/15/aws_down. (Dec. 2021). Accessed: 2024-1-30.

[59] Ashwin. Sridharan, Roch Guerin, and Christophe Diot. 2005. Achieving near-optimal traffic engineering solutions for current OSPF/IS-IS networks. *IEEE/ACM Transactions on Networking* 13, 2 (2005), 234–247. https://doi.org/10.1109/TNET.2005.845549

[60] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 787–803. https://www.usenix.org/conference/osdi20/presentation/tang

[61] The gRPC Authors. 2016. gRPC. https://grpc.io/. (2016). Accessed: 2024-1-30.

[62] The Kubernetes Authors. 2015. Production-Grade Container Orchestration. https://kubernetes.io/. (2015).
[63] Paul Tune and Matthew Roughan. 2013. Internet Traffic Matrices: A Primer. (2013).
[64] Kannan Varadhan, Ramesh Govindan, and Deborah Estrin. 2000. Persistent route oscillations in inter-domain routing. *Computer networks* 32, 1 (2000), 1–16.
[65] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.
[66] Anoop Vetteth. 2019. Docker Inside Cisco Catalyst 9000 Switches. https://blogs.cisco.com/networking/application-hosting-on-catalyst-9000-series-switches. (June 2019).
[67] WIRED. 2019. The Catch-22 that Broke the Internet. https://arstechnica.com/information-technology/2019/06/the-catch-22-that-broke-the-internet/. (June 2019).
[68] Yunhong Xu, Keqiang He, Rui Wang, Minlan Yu, Nick Duffield, Hassan Wassel, Shidong Zhang, Leon Poutievski, Junlan Zhou, and Amin Vahdat. 2022. Hashing Design in Modern Networks: Challenges and Mitigation Techniques. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 805–818.

Appendices

Appendices are supporting material that has not been peer-reviewed.

A MPLS Sublabels

As mentioned in §3.2, modern routers support pushing 12 MPLS labels which is enough for our operational production path lengths to be able to do strict source routing. However some networks may either be more constrained in the number of MPLS labels supported due to comprising older hardware, or have higher operational requirements requiring path lengths of greater than 12. These situations require considering alternative dataplane techniques.

A.1 Compressing Strict Paths

One common solution is to use loose source routing in which a subset of nodes in the path are picked as waypoints, shortening the number of header labels necessary [11, 31]. However this requires a separate underlay, such as IS-IS, to provide the connectivity between waypoints, which goes against our goal of simplification.

To enable strict source routing using fewer labels, we propose a scheme we call MPLS Sublabels, in which we encode multiple router hops with a single label. Importantly, our sublabel encoding scheme is done with no coordination across routers beyond the standard initial link-state exchange, so maintains our "consensus-free" property (§3.1).

In our scheme, we split a single MPLS label into a number of sublabels, each corresponding to a specific directed link. The link sublabels are configured by the operator for each router and distributed in NodeStateUpdate messages. In our implementation, these sublabels are 10 bits, allowing two sublabels to fit in one MPLS label as shown in Figure 17, uniquely representing a *pair* of subsequent edges along the route.

Naively, in a network with n (directed) links there are n^2 unique pairings of links, which grows very large as n approaches its maximum (unique) value of 1024 and thus the resultant MPLS labels may not fit into limited hardware memory. Our key observation is that the space of possible pairs of sublabels that can occur in a single MPLS label is (a) severely restricted by the fact that only adjacent links can show up as a pair of sublabels, and (b) entirely derivable from the information each node knows about its direct

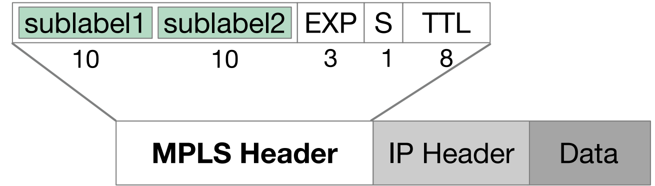


Figure 17: *The MPLS header contains 20 bits of label space, along with 12 other bits. In subsequent figures we omit the 12 extraneous bits and show just the portion containing the two sublabels.*

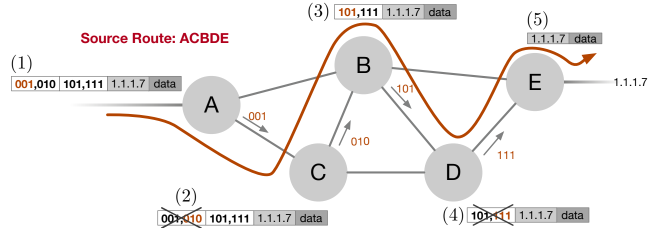


Figure 18: *Network-level view of how sublabels-based packet forwarding is experienced. Only traversed edge labels are included. Crosses denote a label being popped after lookup.*

neighbors. The first observation means that for a network with max degree k, there are at most $2k^2$ feasible MPLS entries formed from adjacent pairs of sublabels[6] for each router that thus need to be programmed into its hardware. Notably, this number grows with the average degree but is independent of the total number of routers or links. In our WAN, this would result in an average MPLS table size of a few hundred entries and a worst case of a few thousands of entries, both of which comfortably fit in the many tens of thousands limit of standard hardware. The second observation means that this set of labels can be programmed without needing any central or distributed coordination.

To populate a router's static lookup table, we enumerate all possible pairs of sublabels that a router could see, as shown in Table 1. For each created label, we encode an egress port and associated action — either keeping the label if forwarding along the first link in the pair, or popping it if forwarding along the second, thus revealing the next label to the subsequent router.

To program a source route, the ingress router compresses it into pairs of link sublabels and puts the resulting label stack into the forwarding table. Odd-length paths are padded with a null-sequence sublabel Ø, which is a 10-bit sequence of 0s. More concretely, for any given node R, let ℓ_{in}^{R} represent the sublabel for any ingress link to R, ℓ_{out}^{R} represent the sublabel for any egress link from R, and ℓ_{out}^{N} represent any egress link from R's neighbor N. Then for a node R, its MPLS forwarding table is made up of four types of entries with full MPLS label keys and (action, interface) pairs as values as shown in Table 1.

These entries are computed and installed at a router once it has received updates from all of its neighbors, and remain static for a given topology.

[6]We can see this from Table 1; in a single router's forwarding table, there are $k(k-1)+k(k-1)+k+k$ valid and loop-free unique combinations of sublabels.

Label	Action	Out Interface
concat(ℓ_{in}^R, ℓ_{out}^R)	pop_label	$intf(\ell_{out}^R)$
concat(ℓ_{out}^R, ℓ_{out}^N)	keep_label	$intf(\ell_{out}^R)$
concat(ℓ_{in}^R, Ø)	pop_label	$intf(IP\ Dest)$
concat(ℓ_{out}^R, Ø)	keep_label	$intf(\ell_{out}^R)$

Table 1: *The resultant MPLS forwarding entries at a single router in general form, where "concat(a,b)" means combining sublabels a and b into a single 20-bit MPLS label, and "intf" means interface.*

A.2 Handling Large Networks

The routing scheme as described above naively requires global uniqueness for sublabels. However, large networks will likely have more links than the 1024 unique values possible with 10-bit sublabels. Using a trick from past work [17], we relax the label-uniqueness constraint to require only *local* uniqueness: as long as each node can unambiguously pick the next interface on which to send the packet based on the pair of sublabels it sees, the path can be validly followed.

Hence, the link IDs a node advertises can be any 10-bit ID for each link as long as all possible valid pairs of link IDs that any given router could see are unique.

Assigning sublabels to links to satisfy this requirement is effectively a variant of the multigraph edge coloring problem [56], where the uniqueness constraint is on the possible MPLS labels (pairs of sublabels) that a router could see. From Table 1, we see that for a network with max degree k, ensuring that for any node the labels for its ingress and egress links are mutually unique guarantees no combined MPLS label collisions, and requires a total of $2 \times k$ globally-unique sublabels. Thus we can easily encode edges for a large network with a maximum degree of 50, using $ceil(log_2(2 \times 50)) = 7$ bits, which is within our limit of 10 bits per label, and leaves ample room for future growth in node degree up to 512.

B cSDN Programming Tail Performance

In §4 we explain the per-hop-programming process that many cSDN systems use, and in §5.1.1 we measure the overall T_{prog} for such a system and find it to be quite long, with a median of over 50s.

To better understand the measurements in §5.1.1 and to inform the simulation we do in §5.2, we examine logs from cSDN programming infrastructure and routers to characterize the performance of path programming in B4. Programming a full path of n routers requires $n-1$ transit entry programming events that occur in parallel and establish each hop of the path, followed by one headend encap entry event that enables directing traffic onto the path.

We observe the transit entry programming times to vary greatly between routers at all percentiles (median, 90th, 99th) by a factor of 10, and varying per-router 4x-11x between the median and the 99th percentile, going upwards of 70 seconds at the tail at the most loaded routers.

Figure 19 shows a distribution from a two-week period over a total of 433,210,304 entry programming events of transit and encap programming times, aggregated over all routers in the solid line and for the slowest router in the dashed line. The slowest router is

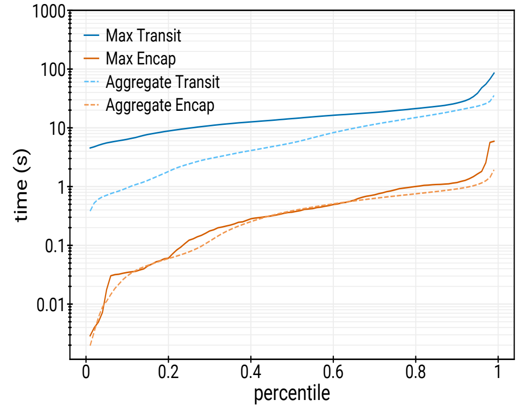


Figure 19: *Distribution of transit entry and encap entry programming times across all routers and for the most heavily loaded router.*

unfortunately slow precisely because it is most loaded and transits the most paths, thus multiplying its impact.

In conclusion, this provides insight on why the total T_{prog} in such a system is particularly long; the two-phase programming operation described above means programming a path is gated by the slowest response from transit entry programming along that path, and network-wide convergence is gated by the slowest path.

C Smart Fast Reroute (FRR)

Link failures can range from partial to complete capacity loss. In the case of complete capacity loss, the router is no longer able to forward any packet on the downed link. However during the reconvergence period, traffic may still arrive from a stale headend intended to traverse the downed link. To ensure this traffic is not dropped, existing systems use a Fast Reroute (FRR) mechanism to pre-install backup paths for links to go around the failure and continue on the intended path. These backups are critical to meeting SLOs during convergence, and are typically picked to be the shortest alternate path between a link's endpoints [1].

However, because TE does not reserve this backup capacity, this can result in considerable congestion on backup paths during large reconvergence events leading to dropped traffic. An analysis of our WAN shows that FRR congestion is a leading cause for performance incidents, lasting up to several minutes and impacting hundreds of flows. We explore using dSDN's position on-box and knowledge of demand and capacity to select bypass paths dynamically and more intelligently, in a capacity-aware manner. Because dSDN acts in-band and has a complete view of traffic demands, it can compute bypasses with high capacity and recompute them as demand changes, unlike the static bypasses established by RSVP-TE.

We evaluate 3 custom FRR strategies against the traditional shortest path FRR selection. These strategies choose one or more bypass paths for each failure and make use of capacity and diversity constraints. Given a failed link, the strategies considered compute bypasses as follows:

1. **FRR:** Choose the shortest path between the endpoints of the failed link. This replicates the existing behavior in our production WAN.
2. **Capacity-Aware FRR:** Choose the path between the link's endpoints with the highest spare capacity.

#	FRR	Capacity Aware	k Shortest Paths	k Capacity Aware Paths
1	5.51% (1)	6.24% (1)	2.53% (1)	0.0% (1.24)
2	2.09% (1)	0.24% (1.06)	1.16 % (1)	0.0% (1.09)
3	1.25% (1)	0.0% (1.02)	0.08% (1)	0.0% (1.02)
4	1.58% (1)	0.25% (1.02)	0.11% (1)	0.0% (1.08)
5	1.54% (1)	0.03% (1.06)	1.34% (1)	0.0% (1.08)
6	1.20% (1)	0.13% (1.16)	1.09% (1)	0.0% (1.19)

Table 2: *Blast radius and (median end-to-end latency inflation) for affected high priority traffic, for 6 performance alert instances, for each bypass strategy. Blast radius is defined as the % of metro pairs between which flows are degraded beyond the SLO).*

(3) **Multi-path FRR:** Choose the k shortest paths between endpoints. For each impacted flow, pick either the shortest bypass path with enough spare capacity or pick the highest capacity bypass. We experimented with different values of k and chose $k = 16$ as a balance between state size and effectiveness.
(4) **Capacity-Aware Multi-Path FRR:** Choose k paths with the most spare capacity and load balance across them.

We simulate a sample of the six most impactful performance alerts that were attributed to FRR congestion over a 14 day period. As done in §5.2, we measure blast radius — the instantaneous impact during convergence — after the link failures and before any repathing for high priority traffic. As we move away from the shortest bypass paths, we also consider the relative end-to-end latency inflation for paths impacted. These metrics are shown in Table 2 for the highest priority traffic class. We find that having smarter bypass strategies would have mitigated the availability hits completely in all six scenarios. Having multiple capacity-aware bypasses for each link eliminates drops for the cases analyzed, but we leave a thorough exploration of strategies for future work.

While the latency inflation from longer paths is mostly harmless, it may be more than 20%, as shown in the table. This is an inherent trade-off, and can be tuned to match operator preference by adjusting relative priorities of capacity and latency when selecting FRR paths.

D Transient Impact with Bypasses

As shown in the previous section, bypass paths for failed links can mitigate impact during convergence. Here, we evaluate cumulative impact for cSDN and dSDN, similar to §5.2, but with bypasses in effect. This is shown in Figure 20. The strategy used to select bypass paths is the capacity-aware multi-path strategy explained in the previous section. We measure the distribution of cumulative impact over network events sampled for a typical day in production.

We continue to see that dSDN's impact is much lower than in cSDN. Note that while bypasses here are not able to completely eliminate impact for lower priority traffic, they still lower this impact.

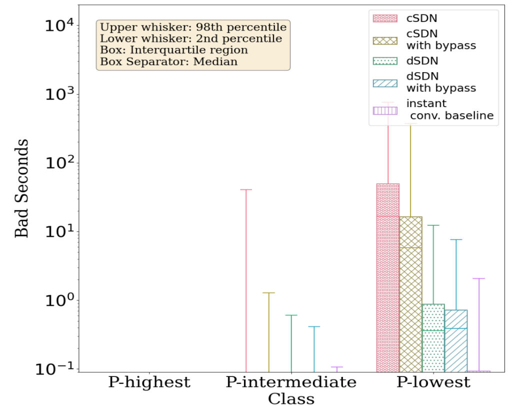


Figure 20: *Distribution of cumulative bad seconds over a day's worth of network events, for cSDN and dSDN, with and without bypasses in effect. An omniscient instantly converging protocol is shown as a baseline. Highest to lowest traffic classes are shown from left to right. The figure shows the 2nd, 25th, 50th, 75th, and 98th percentile points in the distribution of measured bad seconds.*