Recursive SDN for Carrier Networks

James McCauley[‡▲], Zhi Liu[†], Aurojit Panda[‡], Teemu Koponen[◇]
Barath Raghavan[▲], Jennifer Rexford[♦], Scott Shenker[‡▲]

[‡]UC Berkeley, [▲]ICSI, [†]Tsinghua University, [◇]Styra, [♦]Princeton

ABSTRACT

Control planes for global carrier networks should be programmable and scalable. Neither traditional control planes nor new SDN-based control planes meet both of these goals. Here we propose a framework for recursive routing computations that combines the best of SDN (programmability through centralized controllers) and traditional networks (scalability through hierarchy) to achieve these two desired properties. Through simulation on graphs of up to 10,000 nodes, we evaluate our design's ability to support a variety of unicast routing and traffic engineering solutions, while incorporating a fast failure recovery mechanism based on network virtualization.

CCS Concepts

•**Networks** → **Programmable networks;** *Network control algorithms;* Network design principles; Programming interfaces; Routing protocols; Traffic engineering algorithms;

Keywords

Software Defined Networking, SDN, Carrier Networks, Network Routing, Hierarchical Networks

1 Introduction

Software-Defined Networking (SDN) has made great progress in various contexts, most notably within datacenters and in private WANs that interconnect datacenters. However, there has been surprisingly little published work on using SDN in a more traditional networking context: that of global-scale carrier networks (such as operated by Deutsche Telekom, France Telecom, Verizon, NTT, AT&T, and others). These carrier networks are far more geographically dispersed than datacenter networks (by roughly four orders of magnitude), while having far more nodes than the global networks that are used solely to interconnect those datacenters (by roughly three orders of magnitude). There are SDN designs that can handle large numbers of nodes (*e.g.,* Kandoo [3]), and SDN designs that can handle global networks that interconnect a limited number of datacenters (*e.g.,* B4 [5]), but to our knowledge there are no SDN designs that simultaneously handle both the numerical scale and geographic scope of today's carrier networks.

This paper proposes a recursive approach to SDN routing[1] – called Recursive SDN (RSDN) – that leverages the hierarchical structure of carrier networks to achieve the programmability of SDN networks while retaining the scalability (through hierarchy) of legacy networks. In RSDN, each level of the route computation acts on a set of aggregates (called logical cross-bars, or LXBs), and then communicates a summary of the results to the appropriate parent and child LXBs. This limits the number of nodes any individual route computation has to handle, while also ensuring that route computations are as local as possible (*i.e.,* only involve the affected LXB and, recursively, its children).

It is important to clarify what RSDN does and does not do. RSDN is a *framework* for scalable SDN routing, not a particular routing algorithm. That is, RSDN provides a recursive structure for computing routes; a wide variety of recursive designs for unicast routing and traffic engineering can be built within this framework, as we illustrate later in this paper.[2]

Moreover, RSDN focuses only on edge-to-edge packet delivery, which is only a small subset of control plane functionality. For example, in addition to routing, network control planes are often used to enforce policies (*e.g.,* through the use of ACLs) and invoke middlebox functionality (by ensuring packets traverse the appropriate middleboxes). *We investigate no such additional tasks.* Instead, we follow the approach espoused in [2] and [13] in which all non-routing functionality is implemented at the network edge (and need not be controlled by RSDN). We adopt this approach because it can support the necessary functionality while creating a network modularity with a clean separation of concerns.

However, RSDN does more than merely facilitate route computation. Because availability is a crucial requirement for carrier networks, RSDN incorporates a mechanism for rapid and localized recovery from failures that is independent of the particular routing algorithm. Thus, RSDN is a framework that both provides high availability and support for implementing various routing algorithms. We now turn to its design.

2 Overview and Context

Recursive structure: RSDN exploits the locality that is found in almost all networks (and particularly in carrier networks) by clustering the network into aggregates, which we call logical cross-bars (LXBs). These LXBs act like switches – they have a set of external ports and can provide transit between those external ports. We repeat this process of aggregation on the LXBs to build a hierarchical structure with each k-level LXB being comprised of multiple (k+1)-level LXBs, and with links

[1] We use the term "routing" broadly to encompass the choice of routes in unicast, multicast/anycast, and traffic engineering.

[2] This paper is a shortened version of a longer work [4] that contains more details about the design and our performance results. In particular, discussion of RSDN's approach to multicast/anycast routing is omitted here entirely.

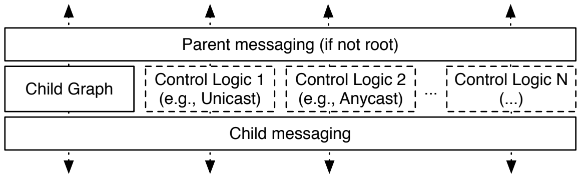
Parent messaging (if not root)
Child Graph
Control Logic 1 (e.g., Unicast)
Control Logic 2 (e.g., Anycast)
...
Control Logic N (...)
Child messaging

Figure 1: Software structure in a normal (non-leaf) LXB. Solid boxes are parts of the framework, dashed boxes are "user"-provided control logic. Leaf LXBs are similar, but include repair functionality, and the bottom interface faces physical switches rather than child LXBs.

between the LXBs on the same level (or *tier*). For example, levels of aggregation may include: PoP and/or datacenter, access network, regional network, and country (or continental) network. This kind of aggregation is standard in networking (and, in particular, is used in approaches such as PNNI [11] to aggregate the topology), but here we leverage the hierarchical structure to form an explicitly recursive SDN control plane.

We associate a logical controller with each LXB; this controller is aware of the parent LXB as well as the child LXBs and the links between them, and handles all control plane computations for the LXB (such as how to route between its child LXBs). While it will typically be replicated across multiple physical machines for reliability, this controller is – logically speaking – a single entity.

This hierarchy of logical controllers provides a recursive programming model (depicted in Figure 1), where for upward-bound computations, each LXB accepts state from its children, performs some local computation on this state, and then exports information to its parent. Downward computations are the inverse. The nature of the state being pushed up or down and the nature of the local computation are fully general, though a scalable control logic design is likely to make state less detailed as it flows up the hierarchy, and more concrete and detailed as it flows down the hierarchy. Moreover, computations need not be strictly upwards or downwards, but can push information in both directions as needed.

In the simplest case, the same exact control logic code may be run at all levels of the hierarchy, but sometimes there is benefit in doing otherwise; *e.g.,* one might use a different routing algorithm within datacenters than is used between them.

Role of RSDN: RSDN is an unabashedly clean-slate design intended to answer the fundamental (and previously unanswered) question of whether SDN could simultaneously cope with the numerical scale and geographic scope of carrier networks. To be clear, our recursive structure is not needed to address either geographic scope or numerical scale by themselves. As shown by designs for interconnecting datacenters (which have the former but not the latter) and controlling datacenters (which have the latter but not the former), current SDN approaches suffice for each challenge individually (though RSDN may provide a cleaner and more flexible scaling story than current datacenter designs). Recursion is only needed when confronting these dual design challenges simultaneously, which most notably occurs when trying to design a coherent control plane for carrier networks.

While RSDN is a clean-slate design for providing the core routing functions of carrier networks, it can cleanly coexist with other aspects of carrier control planes (whether legacy or not) that can be implemented at the edge (such as BGP, access controls, and middlebox insertion).

What Matters: Issues like route computation time, path stretch, and routing state are important properties of a routing algorithm. However, RSDN supports a wide variety of route computation algorithms offering a range of tradeoffs between these metrics. Thus, while these quantities are important for evaluating whether a particular route computation algorithm is suitable for a given network, they are not properties of RSDN itself. What really matters about RSDN is whether: (i) RSDN enables a broad enough class of routing algorithms (broadly defined) to meet various needs and (ii) RSDN-controlled networks can respond to failures quickly enough to achieve high availability. The first issue is addressed by our description and evaluation of two unicast routing schemes (Section 3) and two traffic engineering schemes (Section 4). The second issue is addressed by our inclusion of a network repair mechanism in RSDN (Section 5), which provides – independent of any routing algorithm – a rapid failure recovery mechanism.

Related Work: While we lack space to cover related work in any detail here (see our longer submission [4] for an extended treatment), we do wish to touch on a few salient items. First, RSDN shares a number of attributes with many systems described in the vast literature on area-based and recursive routing. To be clear, we make no claim of novelty for hierarchical routing, since the basic notion is almost as old as routing itself [7]. But RSDN is not a routing algorithm or protocol. Rather, it is a *software framework* for the programmatic and recursive computation of the various aspects of routing (*e.g.,* unicast, traffic engineering) – the specific routing algorithms we discuss in this paper are merely examples.

In terms of applying SDN to global-scale networks, B4 is the most relevant work [5]. B4 is a sophisticated traffic engineering solution for a network that interconnects a few dozen Google datacenters, and we see it as a brilliant solution to a different problem. As a routing control plane, we note that B4 copes with geographic scope, but not numerical scale. As an exercise in traffic engineering, B4 leverages the ability to control its own traffic at the edge and to give some traffic low-quality service in order to achieve extremely high utilizations; in the traffic engineering designs we present here (which more closely represent current carrier requirements), we do not assume one can throttle edge traffic nor that there are low-quality classes of service one can use to keep utilization high.

RSDN's network repair algorithm is a generalization of link protection (as in MPLS FRR [10]) and an application of network virtualization (similar to that described in [1]), and thus has roots in the previous literature, but we are not aware of any existing work that combines them in the way we do here.

Topologies: To evaluate RSDN, we performed simulations on synthetically generated graphs with three levels of hierarchy (in which we ignore the tree-like access networks where routing is trivial). Our randomized topology generator is generally consistent with the description of "heuristically optimal" networks described in [8], and we varied our topologies across two dimensions – size (little, medium, big) and degree of connectivity (low, medium, high) – resulting in nine topologies varying from 119 to 255 switches and from 224 to 547 links. We assigned latencies to the links in these topologies reflecting their structure (*i.e.,* links within an LXB had smaller latencies than those between them), and in our results on stretch we report on both the hopcount and latencies of the end-to-end paths. To demonstrate RSDN's ability to scale, our unicast routing simulations focus on larger topologies with sizes between approximately $1,000$ and $10,000$ switches, and degrees of connectivity spanning over a factor of three (ranging both above and below what we believe to be realistic). The other topologies are smaller because computing the optimal non-RSDN traffic engineering solutions (which we need for comparison) on large networks is overly time consuming.

3 Unicast Routing

As mentioned above, although RSDN supports routing, RSDN is not a routing *protocol*. One key difference is that non-SDN routing is often monolithic, with protocols conflating separable issues such as topology discovery, link state discovery, and route computation. When treating networking as a software problem, however, these may be factored more cleanly. For example, in our discussion of RSDN routing in this section, we strictly consider the problem of route computation. That is, we do not discuss mechanisms for discovering the topology or the state of links within the topology. While these aspects of the overall routing problem can be built as additional, *largely decoupled* control logics fitting into the RSDN framework, we do not focus on them here: state discovery varies depending on the link type and is typically not deep (*e.g.,* just a mechanism for forwarding the results of a BFD [6] agent up to the appropriate tier), and topology discovery does not apply to the use case we are examining in this paper (in carrier networks, the topology is carefully planned and centrally mapped; we thus assume that each controller is simply given a list of the links it "owns" – the ones between its child LXBs).

That said, the design space for unicast routing solutions is vast. The goal of RSDN is not to pick one particular approach, but to enable operators to choose or design one suited for their needs while leveraging the recursive structure of RSDN to scale. To illustrate how RSDN supports route computation, we discuss two representative examples that we call Fine-Grained Routing (FGR) and Coarse-Grained Routing (CGR). Both are implemented using an *up* phase and then a *down* phase, where one should think of the upward pass as when LXBs collect information from their child LXBs (or from physical switches at the base of the hierarchy) and summarize it for their parents, and the downward pass as when LXBs select routes and then pass on the results to their children, providing enough context for the children to do the same for *their* children.

FGR uses a recursive approach to compute shortest paths across an entire RSDN network. In the upward pass, each LXB's controller computes shortest paths between each of its border switches (that is, switches with links which reach outside the LXB). These are pushed upward to the parent controller as a distance matrix. Once the parent has distances between the borders of its children, it can compute shortest paths between its own border switches which it pushes up to its own parent. In the downward phase, controllers push down actual forwarding rules based on the paths computed during the upward phase. These accumulate down the hierarchy; when the base is reached, the forwarding state is complete. FGR scales better than a flat shortest path computation because of information hiding (a parent's view of its children is only in terms of distance matrices between their border switches) and parallelization (controllers on a tier can operate in parallel).

CGR is a coarser grained approach to address cases where FGR does not scale sufficiently. We do not have space to describe the algorithm in detail, but the starting point for the design is to only compute shortest paths between siblings. However, to reduce stretch, we found it useful to extend the computation to find shortest paths from a switch to the *closest* switch in the LXB which contains the destination and is a sibling of the LXB containing the source. Upon reaching this "closest" switch, the process repeats recursively one tier down the hierarchy until the final destination is reached. See [4] for details.

Tier	CGR	FGR	APSP
3	0.407s	0.391s	—
2	5.709s	6.036s	—
1	6.745s	6.811s	—
0 (Root)	7.215s	249.081s	682.452s

Table 1: Times to run computation on topology X up to a given level in the hierarchy using FGR, CGR, and an all-pairs shortest path algorithm over the entire topology (APSP).

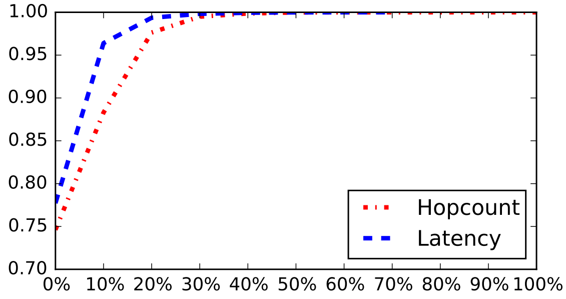

Figure 2: The CDF of network stretch (as fractional increase over the original path) for CGR on topology X.

3.1 Performance of Routing Algorithms

We chose FGR and CGR merely to demonstrate that RSDN could incorporate widely varying routing designs, and there are many other routing designs one could devise, but here we discuss how FGR and CGR perform in terms of stretch, computation time, and routing table size. For clarity, our presentation focuses on results from a single very large topology ("topology X") which has $10,355$ switches and $47,595$ links with a mean link latency of 14.584ms. However, we also performed extensive experiments with other topologies mentioned in Section 2, and the general conclusions we draw here about RSDN's performance are consistent with the results from those additional experiments.

Computation Time: We compared our Python RSDN route computations with a Python All-Pairs-Shortest-Path (APSP) computation. The numbers here are not indicative of how an optimized C++ computation would perform, but the relative performance provides some measure of the underlying computational complexities. Route computation times are shown in Table 1 for FGR and CGR up to each tier (this is relevant because after an initial computation, one only needs to recompute to establish good routes after a failure; in only very few cases – a failure of a country-to-country link in our topologies – does this require recomputing all the way up to the root). Note that FGR beats APSP due to RSDN's parallelization even though they both compute globally shortest paths. CGR trades off path length for significantly better performance.

Stretch: FGR produces no stretch, as it computes exact shortest paths except in the case when a shortest path must exit and

Routing	Label	LPM Cons.	LPM Rand.
CGR	211.56	382.09	13088.38
FGR	2942.76	1296.02	41206.13

Table 2: Average table size with labels, highly-aggregatable (consecutive) IP prefixes, and poorly-aggregatable (random) IP prefixes.

later re-enter the same LXB (which we believe to be rare and did not occur in our topologies). The stretch induced by CGR on topology X is shown in Figure 2, which shows the CDF of stretch over all source-destination pairs using both hopcount-based routing and latency-based routing (the latter minimizes the path latency, thus accounting for geographic distance). For hopcounts, fewer than 15% of the pairs have stretch over 10%, while for latency, approximately 5% do. These latency results were consistent with the average of the additional topologies we tested: 75% of the paths have no additional stretch and 92% of the paths have have less than 10% stretch. For hopcount-based routing on these additional topologies, the results were somewhat worse: 70% of the paths have no additional stretch, 73% of the paths have have less than 10% stretch, but 93% of the paths have less than 20% stretch.

Table Size: RSDN can incorporate a variety of approaches to forwarding tables and address assignments, and here we discuss two of them. First, we considered an MPLS-like scheme where one or more labels are applied to the packet at the network edge, and all forwarding is done on these labels. Table 2 shows the resulting average forwarding table sizes that arise using this approach for both FGR and CGR on topology X, with CGR being significantly smaller than FGR (about 93% savings). We then tried using IP addresses for forwarding (with LPM aggregation in the tables) considering two address allocation schemes. The first took a set of about half a million prefixes from Route Views [12] and assigned them consecutively to the exterior ports (putting the same number of prefixes on each port); the second assigned these randomly to the exterior ports. Unsurprisingly, the easily aggregated consecutive assignment yielded smaller tables, though CGR provided about 70% savings for both. These results show that: (i) RSDN can use either labels or aggregatable addresses, and (ii) if small table size is important, CGR provides an effective and relatively low-stretch way of accomplishing this (with either labels or reasonable address allocation).

4 Traffic Engineering

As with unicast routing, our goal here is not to pioneer new TE paradigms but instead to merely show that RSDN can scalably achieve TE goals. There are many TE designs, but most of them can be grouped into two categories. In the first, the traffic matrix is fed into an offline solver that generates optimal routes. The second is used in conjunction with multipath routing: a feedback system relays congestion information about a path to its source, and the source preferentially steers traffic over less-loaded paths. We have implemented recursive versions of both approaches, which we term Recursive Linear Programming (RLP) and Recursive Split Tuning (RST), and where the latter is enabled by a multipath variant of FGR.

For our evaluation, we focus purely on whether good routes can be chosen, and ignore real-world issues such as flow divisibility and packet reordering (which are issues for any TE design). The metric by which we evaluate TE is the utilization level of the maximally loaded link, which should be minimized. We compare both of our RSDN TE implementations against a "gold standard" – a straightforward global linear program that achieves the minimum possible value of this metric.

Topology	RLP	RST	None	G-S
little-low	1.00	1.00	1.48	1.00
little-med	1.00	1.00	1.60	1.00
little-high	1.00	1.00	1.20	1.00
middle-low	1.00	1.22	2.22	1.00
middle-med	1.00	1.00	1.58	1.00
middle-high	1.00	1.00	1.84	1.00
big-low	1.00	1.00	1.79	1.00
big-med	1.00	1.00	1.56	1.00
big-high	1.00	1.00	1.87	1.00

Table 3: Maximal link load normalized by that achieved by the globally optimal gold-standard on the nine topologies (labeled by their relative size and connectivity).

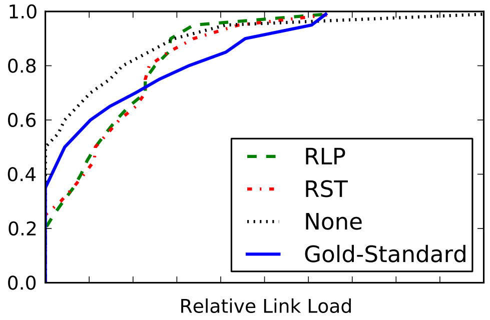

Figure 3: CDF of the load on links in the big-med topology when using each of the traffic engineering methods.

We evaluate TE by running simulations on the nine topologies described in Section 2. Table 3 shows the worst-case link load for RLP and RST and compares them to the results from the global optimal (against which all the results are normalized) and from merely using shortest-path routing with no TE. In all cases, RLP matches the performance of optimal to within less than a percent. RST is also generally within a percent of optimal; a single case misses by 22%. Additionally, leveraging locality means the recursive approaches converge faster: on average, RLP reaches its best case in 65% of the time needed by the gold-standard, and RST in only 15%.

In Figure 3 we look at the distribution of link loads in one of the topologies. While the goal of the gold-standard TE algorithm is only to minimize the maximal load, the RLP and RST algorithms do a better job of spreading the load around (even if they may not always achieve the same mini-max load). That is, RLP and RST have fewer highly loaded links compared to the gold-standard, having pushed some of that load off to more lightly loaded links; the gold-standard does not bother trying to decrease loads on less-than-maximally-loaded links (such load spreading *could* be made an objective of a global solver, but comes "for free" with our recursive approach).

5 Network Repair

While the previous two sections have examined control logic written using the RSDN framework, this section examines a feature of the framework itself, which all control logics automatically benefit from: network repair.

Nbr[a]	Neighbors of a
HavePath	Nodes to which a already has a path.
NeedPath	Nodes to which a needs a path.
Vnodes	Nodes that a "virtualizes" (includes the forwarding table for)
EnsurePath(a,b)	Ensures there is a path between a and b given the current state of the network. Returns true if the link between a and b is up, has working link protection, or if a repair path can be computed.

Table 4: Network repair sets and functions for node a.

Algorithm 1 Network Repair Algorithm.

```
▷ Repair algorithm for node a
NeedPath $\leftarrow$ Nbr[a]                    ▷ Initialize NeedPath
HavePath $\leftarrow \emptyset$                          ▷ Initialize HavePath
Vnodes $\leftarrow$ { a }                         ▷ Initialize Vnodes
while NeedPath $\neq \emptyset$ do
    b $\leftarrow$ member(NeedPath)              ▷ Take element b
    NeedPath $\leftarrow$ NeedPath$-$b
    if EnsurePath(a,b) $=$ true then
        ▷ If we can find a path from a to b, we're done
        HavePath $\leftarrow$ HavePath $\cup$ b
    else
        ▷ If no path is found, virtualize b
        Vnodes $\leftarrow$ Vnodes $\cup$ b
        ▷ Need paths to neighbors of b that we can't reach yet
        NewNbrs $\leftarrow$ Nbr[b]$-$HavePath$-$Vnodes
        NeedPath $\leftarrow$ NeedPath $\cup$ NewNbrs
    end if
end while
```

A common practice for improving network availability is to implement link protection, in which for every link between two routers (or nodes, in the text below) a and b, an alternate path (not including that link) is precomputed and then immediately used as a failover route when the link goes down. This works as long as the failover route remains up, but cannot cope when multiple failures knock out both the link and the failover path. RSDN uses link protection, but then adds a more general network repair mechanism that can recover from *all* failure scenarios (as long as a path exists).

Our network repair approach is inspired by network virtualization. When routes are computed, we note the state of the network (*i.e.,* which nodes and links are up). We then embed a virtual version of this network within whatever the current physical network happens to be; this virtual network clearly supports the previously computed routes, so they need not be recomputed (though can be, lazily, for optimality). Note that RSDN's repair approach does not rely on the recursive structure and could therefore be implemented on any SDN network.

To understand our repair algorithm, consider a particular network state (in terms of which nodes and links are up and down), and the sets and functions shown in Table 4; the network repair algorithm at node a is as given in Algorithm 1. After applying this simple procedure for each node, every node a has a set of internal tables that can be used to virtually route *through* unreachable nodes. That is, suppose that if a were to send a packet destined for some node x, its first two hops would ordinarily be nodes b and c. When a link fails, if the responsible controller discovers that a can no longer reach b (because EnsurePath(a,b) fails), then the controller *virtualizes* b within a (by including b's routing table in a), determines where b would have sent the packet (in this case, node c), and computes a repair path to c using EnsurePath(a,c). If this succeeds, then a can forward the packet directly to c; if it fails, then the procedure recurses and a imports c's routing table, determines where c would have routed the packet, and then attempts to directly route to that node.

This procedure is initiated whenever a neighbor fails (or otherwise becomes unreachable), and results in a working set of routing tables. The computational complexity of this operation scales not with the overall network size but with the complexity of computing repair paths between nearby nodes (*i.e.,* when EnsurePath finds that there is no direct or protected link between a and b and must compute a new path). The maximal distance between any two nodes where this function is invoked is the most consecutive unreachable nodes in a path. It is very unlikely that this distance will ever be more than a few hops.

5.1 Evaluation of Network Repair

When considering failures, we ran experiments using a failure model based on [9], which randomly fails and recovers links according to distributions measured in the Sprint network, and apply this same model to switches. We looked at a variety of failure scenarios, but only report on two here: where links fail at an average rate of once per day, and where they fail once per ten days. In each case we arbitrarily set the node failure rate to half of the link failure rate.

Since no recovery mechanism can provide connectivity when the physical network is disconnected, the appropriate performance metric is the percent of time physically connected pairs are connected by routing; or, for short, the connectivity of the physically reachable pairs (which we denote by *CPRP*).

In analyzing the performance of network repair, we make the following two assumptions. First, we only consider failures once they have been detected by the sending switch, since there is nothing a routing or recovery mechanism can do to deal with undetected failures (other mechanisms – such as forward error correction over multiple paths – can deal with this eventuality, but simple routing itself cannot). Second, we assume that the time to repair a failure is, on average, 50ms. This is based on estimates of how long it takes (i) for messages to travel from switch to controller and back (to which we allocate 10ms in each direction, which is roughly the average over all switch-to-controller latencies in our topologies; this number is low because most links are quite local to their bottom-tier controller), (ii) for the controller to compute repair paths and generate a response (to which we allocate 10ms, which is more than reasonable),[3] and (iii) the time it takes to install new routes (to which we allocate 20ms). This last quantity is by far the most variable, as it depends on the number of routes, the router technology used, and other factors outside our control. It is also the least fundamental and, thus, the most amenable to improvement by technological advancement and technology choice; for example, note that exact-match insertions (as in MPLS, which can be used for internal forwarding in RSDN) are typically far faster than when inserting for LPM.

We present performance results of network repair on the three big topologies used in Section 4; we also ran these same

[3] We measured the time taken to compute the on-demand repair paths and found it to be minimal. The mean time spent per failure using our Python code is less than 0.5ms and the maximal time less than 10ms *for all controllers combined.*

Sim.	Topo	None	Link	Repair
light	big-low	97.8%	99.79%	99.999977%
	big-med	98.7%	99.85%	99.999982%
	big-high	98.1%	99.77%	99.999983%
heavy	big-low	82%	96.2%	99.99979%
	big-med	86%	97.9%	99.99983%
	big-high	85%	96.1%	99.99982%

Table 5: Average CPRPs for no recovery, link protection, and RSDN repair on all big topologies under light and heavy failure scenarios.

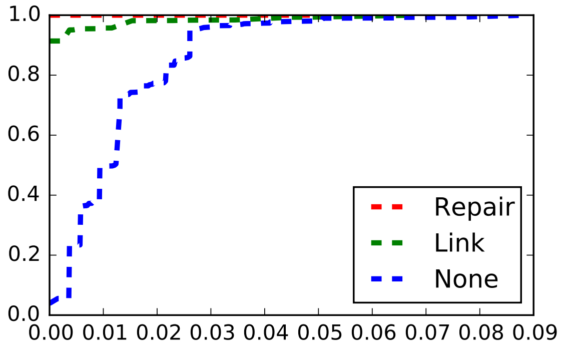

Figure 4: CDF of CPRP (x-axis) for no recovery, link protection, and RSDN repair on the big-med topology under the light failure scenario.

experiments using the topologies of other sizes from Section 4, but omit the (similar) results here for space. We considered three possible strategies: (i) no recovery, (ii) recovery using only link protection, and (iii) RSDN's network repair (which includes link protection). We then recorded what fraction of paths between all source-destination pairs are connected by the three recovery strategies over two days of simulated time.

Table 5 shows the CPRP for these three graphs under the two failure scenarios. Note that even under the heavy failure scenario, where links are down roughly 5% of the time, network repair is able to provide "five 9s" of CPRP, while no protection offers no 9s and link protection offers a single 9. This high relative availability is because – in addition to covering the exact same failures that link protection covers in exactly same way – network repair can recover from *all* failures (subject only to table size limits on the switch), and link protection simply cannot. The only reason that repair does not achieve the maximum possible connectivity is due to the 50ms delay when controller involvement is required. Figure 4 shows the CDF of availability for the big-med graph under the light failure scenario. Here we see that the network repair results are almost the same as physical availability, and link repair trails noticeably behind. The heavy failure scenario – not shown here – is even more dramatic.

One might be surprised that such a mechanism can provide five 9s when the Internet is generally deemed to be less than three 9s reliable. The distinction is that we are not counting the case where the network is physically disconnected (since neither routing nor repair can help there) and are only measuring the availability for the physically connected pairs. What we show is that with network repair, the routing algorithm is no longer the availability bottleneck, regardless of the routing algorithm. Instead, the availability bottleneck is strictly physical connectivity, which must be addressed by other means.

6 Discussion

In this paper we propose RSDN as a way to manage global-scale carrier networks. Our focus is on how to scale the functionality that belongs in the network core (unicast, multicast, traffic engineering, etc.), leaving all other functionality to the edge. We find that RSDN provides a clean way to implement a range of designs over hierarchical network infrastructures. The routing designs presented here are not novel in themselves. Instead, the main contributions of RSDN are (i) a flexible and explicitly recursive programming model (which makes it easier to implement scalable versions of these and other routing designs) and (ii) an integrated network repair mechanism that improves possible availability by several orders of magnitude over simple link protection (relieving routing designs of the responsibility to respond quickly to failures).

Finally, RSDN is an exercise in clean-slate design, and we do not presume that such a design can be easily deployed in today's carrier networks. However, we do anticipate that SDN technologies will incrementally work their way into carrier networks, and thus – eventually – an RSDN-like system spanning different levels of the network (*e.g.,* datacenter, access, regional, and continental networks) may indeed be feasible.

7 References

[1] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *Proc. of PRESTO*, 2010.

[2] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving SDN. In *Proc. of HotSDN*, 2012.

[3] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proc. of HotSDN*, 2012.

[4] J. McCauley, Z. Liu, A. Panda, T. Koponen, B. Raghavan, J. Rexford and S. Shenker. Recursive SDN for carrier networks. *arXiv:1605.07734 [cs.NI]*, 2016.

[5] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: experience with a globally-deployed software defined WAN. In *Proc. of SIGCOMM*, 2013.

[6] D. Katz and D. Ward. Bidirectional forwarding detection (BFD). *RFC 5880*, 2010.

[7] L. Kleinrock and F. Kamoun. Hierarchical routing for large networks – performance evaluation and optimization. *Computer Networks*, 1(3), 1977.

[8] L. Li, D. Alderson, W. Willinger, and J. Doyle. A first-principles approach to understanding the Internet's router-level topology. In *Proc. of SIGCOMM*, 2004.

[9] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, and C. Diot. Characterization of failures in an IP backbone. In *Proc. IEEE INFOCOM*, 2004.

[10] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels. *RFC 4090*, 2005.

[11] Private Network-Network Interface specification version 1.1 (PNNI 1.1), 2002. ATM Forum.

[12] University of Oregon Route Views project. http://www.routeviews.org/.

[13] S. Shenker. Software-Defined Networking at the crossroads, 2013. https://youtu.be/WabdXYzCAOU.