Ripcord: A Modular Platform for Data Center Networking

Martin Casado
David Erickson
Igor Anatolyevich Ganichev
Rean Griffith
Brandon Heller
Nick Mckeown
Daekyeong Moon
Teemu Koponen
Scott Shenker
Kyriakos Zarifis

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-93
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-93.html

June 7, 2010

Ripcord: A Modular Platform for Data Center Networking

Martin Casado[†], David Erickson[‡], Igor Ganichev[⋆],
Rean Griffith[⋆], Brandon Heller[‡], Nick Mckeown[‡],
Daekyeong Moon[⋆], Teemu Koponen[†], Scott Shenker[⋆],
Kyriakos Zarifis[⋆]
[⋆] - UC Berkeley, [‡] - Stanford University, [†] - Nicira Networks

ABSTRACT

Data centers present many interesting challenges, such as extreme scalability, location independence of workload, fault-tolerant operation, and server migration. While many data center network architectures have been proposed, there has been no systematic way to compare and evaluate them—apples-to-apples—in a meaningful or realistic way. In this paper, we present Ripcord, a platform for rapidly prototyping, testing, and comparing different data center networks. Ripcord provides a common infrastructure, and a set of libraries to allow quick prototyping of new schemes.

We built a prototype of Ripcord and evaluated it in software and running on a real network of commodity switches. To evaluate Ripcord, we implemented and deployed several schemes, including VL2 and PortLand. A key feature of Ripcord is its ability to run multiple routing applications, side-by-side on the same physical network. Although our prototype implementation is not production quality, we believe that Ripcord provides a framework for both researchers and data center operators to implement, evaluate, and (eventually) deploy new ideas.

1. INTRODUCTION

The meteoric growth of data centers over the past decade has redefined how they are designed and built. Today, a large data center may contain over one hundred thousand servers and tens of thousands of individual networking components (switches, routers or both). Data centers often host many applications with dynamic capacity requirements, and differing service requirements. For example, it is not uncommon for the same data center to host applications requiring terabytes of internal bandwidth, and others requiring low-latency streaming to the Internet.

Their sheer scale, coupled with application dynamics and diversity, makes data centers unlike any systems that have come before. And to construct and manage them, network designers have had to rethink traditional methodologies. A prevailing design principle is to use *scale-out* system design. Scale-out systems are generally characterized by the use of redundant commodity components. Managed workloads are constructed so the system can gracefully tolerate component failures. Capacity is increased by adding hardware without requiring new configuration state or system software.

While scale-out design is well understood for building compute services from commodity end-hosts, it is a relatively new way to build out network capacity while retaining a rich service model to applications. Other authors have explained clearly [5,12] how traditional data center networks stood in the way of supporting highly dynamic applications, scale-out bandwidth, and the commodity cost model such systems are suited for.

Due to these limitations, the research community, and the largest data center operators – those with the deepest pockets – innovate fast, moving towards new schemes that allow them to construct systems with the requisite properties for their operations.

While many schemes remain proprietary and unpublished, some notable data center network designs have been described. VL2 [5] uses Valiant load balancing and IP-in-IP encapsulation to spread traffic over a network of unmodified switches. On the other hand, PortLand [12] modifies the switches to route based on a pseudo-MAC header, and aims to eliminate switch configuration. Other researchers have proposed Monsoon [7] and FatTree [1]; Trill [17] and DCE [4] have been proposed as standards.

Each proposal holds a unique point in the design space, and subtle differences can have large ramifications on cost and performance, raising the question: *How can we evaluate which scheme is best for a given data center, or for a given service?* And how can we build on the work of others, modifying an existing scheme to suit our needs?

In this paper we set out to answer these questions. Specifically, we developed and describe a new platform, called Ripcord, that is designed so that researchers can quickly prototype new data center network solutions, and then compare multiple schemes - side by side, apples to apples. Ripcord includes a collection of library components to facilitate rapid prototyping (*e.g.* multi-path routing protocols, topology mappers, ...). But perhaps most interestingly, Ripcord allows *several* data center network schemes to run simultaneously on the same physical network. We illustrate this later by running VL2 and Portland at the same time. A researcher may use this capability to evaluate two schemes side by side; an experimental data center can host multiple researchers at the same time; a multi-tenant hosting service may provide different customers with different networks; and a multi-service data center may use schemes optimized for different services (*e.g.* one scheme for map-reduce, alongside another for video streaming).

We evaluate the generality of Ripcord by implementing multiple data center proposals and running them in tandem in software, and individually in both software and on a test data center made from commercial hardware. We present our results in Section 5.

Contributions: In summary, we believe Ripcord makes the following contributions:

1. It allows different data center networking schemes to be compared side by side in the same network.
2. It allows multiple schemes to be run simultaneously in the same network.
3. It allows a researcher to build and deploy a new data center network scheme in a few hours; or download and modify an existing one.

The rest of the paper is arranged as follows: In Section 3 we describe the architecture of Ripcord in detail, and then in Section 4 we describe our first prototype. Our first experiences with Ripcord suggest that by reusing existing technologies, it is possible to meet our goals with a relatively simple system. In Section 4.8 we describe how we implemented three different schemes on Ripcord: VL2, Portland, and VL2 with middlebox traversal [10], and compare their performance in Section 5.

2. OVERVIEW OF RIPCORD

Ripcord's design follows directly from four high-level design requirements: The system must allow researchers to *prototype quickly*, with minimum interference from the platform itself. Ripcord must allow experimenters to *evaluate* a new scheme, and compare it *side-by-side* with others. Finally, it must be easy to *transfer and deploy* a new scheme to physical hardware.

To help researchers prototype quickly, and to encourage code re-use, Ripcord is modular and extensible. We chose a logically centralized design, allowing the experimenter to create control logic for the entire data center without concern for how decisions are distributed.

If we are to help experimenters evaluate and compare different schemes, we need to understand the main criteria they will use. Based on recent proposals (and the needs of data centers) the most challenging criteria are:

Scalability.

Large data centers scale to many thousands of servers or millions of virtual machines (VMs). The experimenter will need ways to compare topologies, routing and addressing schemes and their consequences on forwarding tables and broadcasts. Ripcord's scalability is discussed in Section 6.

Location Independence.

Dynamic resource provisioning in data centers is much more efficient if resources can be assigned in a location-agnostic manner and VMs can migrate without service-interruption. Ripcord must support routing that operates at different layers, and novel addressing schemes.

Failure Management.

Scale-out data centers are designed to tolerate network failures. Ripcord must provide a means to inject failures to links and switches, to explore how different schemes react.

Load balancing.

Data centers commonly spread load to avoid hotspots. Ripcord must enable randomized, deterministic and pre-defined load-balancing schemes.

Isolation and Resource Management.

If multiple experiments are to run simultaneously - just as multiple services run concurrently in a real data center - Ripcord must isolate one from another.

The requirements above led to the following high-level approach. In the next section, we describe the design in greater detail.

Logically centralized control.

At the heart of Ripcord is a logically centralized control platform. Ripcord's centralized approach reflects a common trend in recent proposals (e.g., VL2's directory service, PortLand's fabric manager). While logically centralized, Ripcord can scale by running several controllers in parallel.

Multi-tenant.

In Ripcord, each *tenant* manages a portion of the data center and controls routing. A tenant can be an experiment (*e.g.* PortLand and VL2). Alternatively, in a production data center, the tenant is a management and routing scheme tailored to support a service (*e.g.* MapReduce or content delivery). Ripcord supports multiple tenants concurrently by managing the resources they use in the network.

Modular.

The central platform is modular. It maintains a shared current view of the topology and the state of each switch; it enforces isolation between tenants (i.e. controls which resources they are allowed to view and control). Tenants can select among a variety of routing schemes, arranged as modules connected in a pipeline.

2.1 Our Prototype

Our Ripcord prototype builds upon and replaces the default applications of NOX [8]. NOX is a logically centralized platform for controlling network switches via the OpenFlow [11] control protocol. We summarize NOX and OpenFlow briefly in the appendix. We chose these

Events	Semantics
SWITCH_JOIN	Switch joined network.
SWITCH_LEAVE	Switch left network.
PACKET_IN	Packet without matching flows came in.
STATS_REPLY	Flow stats are available.

Table 1: Network events expected from switch.

Commands	Semantics
FLOW_MOD	Installs/removes flows.
PACKET_OUT	Sends out a given packet.
STATS_REQUEST	Polls flow stats.

Table 2: Expected switch commands.

technologies because they provide a clean vendor-agnostic abstraction of the underlying network, and NOX provides a well-defined API to control the network as a whole. Our prototype could, in principle, be built on any network control abstraction offering the set of events and commands listed in Table 1 and Table 2.

3. DESIGN

To help orient the reader we use an example walkthrough as a a high-level introduction to key components in the system. The following steps describe how Ripcord handles incoming flows by passing them to the correct tenant for routing and setup in the network.

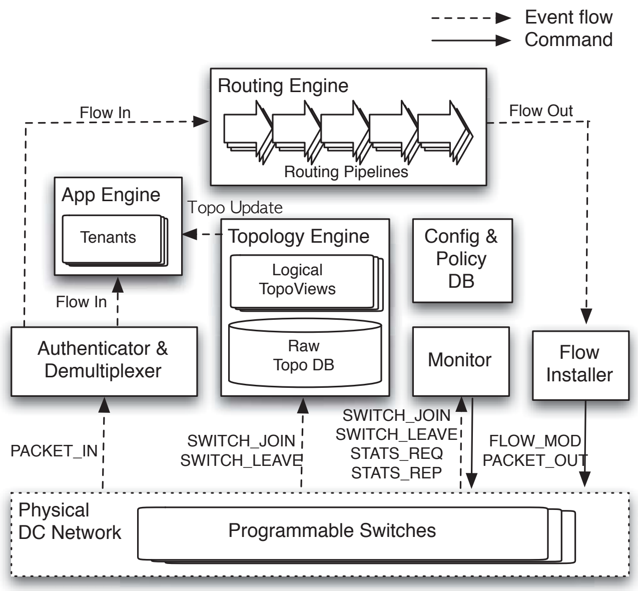

Figure 1: Ripcord architecture and event flow diagram

3.1 Example Walkthrough

Configuration.

The first step in the deployment of a Ripcord data center is to provide the **Configuration and Policy Database** with the particulars of the network. This includes topology characteristics (FatTree/Clos/etc), the tenants, and a mapping from the tenants to the available routing applications (PortLand/VL2/etc).

Startup.

The **App**, **Routing** and **Topology** Engines are instantiated based on administrative information fed to the Configuration and Policy Database. At this point, optional network bootstrap operations (e.g. proactive installation of flows) are carried out. In addition, the Routing Engine and the App Engine register to receive notification on each incoming flow.

Running.

In this state, Ripcord listens for incoming routing requests. These requests are generated as events by switches each time they receive a packet for which there is no existing flow table entry. When Ripcord receives the routing request it makes sure that the packet is either processed by a responsible (per-tenant) **Management App** and its routing pipeline or discarded. The sequence of steps for handling routing requests is outlined below:

1. When a switch receives a packet for which there is no matching flow-table entry it creates a routing request containing the packet and notifies the **Authenticator-Demultiplexer**.

2. The Authenticator-Demultiplexer, receives the routing request, tags it with the identifier of the tenant that should handle it, and, if the routing request is legitimate, notifies the **App Engine**. If the routing request is not legitimate, e.g., it would result in traffic between isolated networking domains, it is denied and the packet is discarded.

3. The App Engine dispatches routing requests to the point of contact associated with each tenant – its **Management App**. The Management App determines whether it should discard the incoming packet, process it, e.g. to handle control requests like ARP, or propagate the request to the **Routing Engine**.

4. When the Routing Engine receives the routing request it invokes the tenant's predefined routing pipeline, which computes a route and then the Routing Engine informs the **Flow Installer**.

5. Finally, the Flow Installer sends out commands to select switches, along the path inserting flow entries in their tables thus establishing the new flow on the selected path.

Monitoring.

Under normal operation, the Monitoring module tracks switches as they join or leave the network. With "always-on" passive monitoring, the network is constantly supervised for abnormalities. If an aberrant behavior is detected, the operator can invoke active monitoring commands to delve into the problem and troubleshoot.

3.2 Components

Figure 1 depicts Ripcord's high-level architecture. It consists of the following seven components:

1. **Config & Policy DB:** is a simple storage for platform-level configuration and data center policy information. Administrators configure the Database with global network characteristics as well as tenant-specific policies. This centralized configuration provides ease of management. As this module merely stores the configuration, the actual policy enforcement is delegated to other components.

2. **Topology Engine:** maintains a global topology view by tracking *SWITCH_JOIN* as well as *SWITCH_LEAVE* events. This allows for real-time network visualization, expedites fault-detection and simplifies troubleshooting. The component also builds per-tenant logical topology views which are used by App and Routing Engines when serving a specific tenant.

3. **Authenticator-Demultiplexer:** performs admission control and demultiplexes to the correct application. Upon receipt of a *PACKET_IN* event, it invokes the Configuration/Policy Database and resolves the tenant in charge of the packet. If the packet is not legitimate, the component drops it. Otherwise, it passes on the routing request to the App and Routing Engines, as a *FLOW_IN* event tagged with the packet and tenant information.

4. **App Engine:** each tenant can have its own management app. Hence, the Management App can be seen as a centralized controller for a particular tenant. This component typically inspects incoming packets in a *FLOW_IN* event and updates its internal state. For example, PortLand's fabric manager can be implemented as a management app on Ripcord. On receipt of a *FLOW_IN* event, the App Engine dispatches the event to a proper app based on tenant information associated with the event.

5. **Routing Engine:** this module calculates routes through a multi-stage process: starting as a loose source route between the source-destination pair, a path is gradually filled through each of the routing pipeline stages. One pipeline stage may consist of zero or more routing modules. Ripcord does not limit the size of routing pipeline. It does, however, enforce the order of stages so as to help verify routing modules' composability. Table 3 describes these stages.

   This small routing module is far easier to verify and manage than a larger, all-in-one routing algorithm package. At the same time, it gives great flexibility as the routing algorithm is not predetermined, but defined by the arrangement of the routing modules. Hence, new routing algorithms can be easily deployed as long as the underlying topology supports them. One can, for instance, shift from shortest-path to policy-based routing merely by replacing one of its routing modules in the *ComputeRoute* stage. The routing pipeline for each Ripcord tenant is configured in the Config/Policy Database as a list of routing modules. We envision that open source developers will contribute routing modules and datacenter administrators will evaluate new routing algorithms on Ripcord.

6. **Flow Installer:** is in charge of translating `FLOW_OUT` event into hardware-dependent control message to modify the switch flow table. We introduce this indirection layer to make Ripcord independent of a particular switch control technology.

7. **Monitor:** provides support for *passive* and *active* statistics collection from network elements. Passive collection periodically polls switches for aggregate statistics, while active collection is targeted to probe a particular flow in the network.

   When a switch joins the network, the component records its capabilities (e.g., port speeds supported) and then maintains a limited history of its statistics *snapshots*. Snapshots contain aggregate flow statistics (e.g., flow, packet and byte counts for a switch), summary table statistics (e.g., number of active flow entries and entry hit-rates), port statistics (e.g., bytes/packets received, transmitted or dropped) and their changes since the last collection.

4. IMPLEMENTATION

Our Ripcord prototype consists of a technology-independent core library (implementing the seven components explained in Section 3), and NOX-dependent wrapper code. It totals 6,988 lines of Python code plus NOX's standard library.

4.1 Configuration & Policy Database

When Ripcord starts, this module reads a directory of configuration files describing the configuration and policy, expressed as key-value pairs. The configuration language is described in JSON because of its ability to richly express dictionary and array types. New configurations can be loaded dynamically via command line arguments, for example to instantiate a new tenant or debug a running system. The policy database needs to be quite general: For example, an administrator might set a policy such as '*Packets sent from the host with MAC address A to the host with IP address B should be routed by tenant Y*'. The policy may be based on any combination of the following fields: the unique ID of the switch the packet was received at, the incoming port on

Stage	Description
TweakSrcDst	The source/destination information is altered at this stage. This is usually for the purpose of loadbalancing among hosts.
InsertWayPoints	This stage inserts particular switches or middleboxes to traverse (e.g. for security reasons)
Loadbalance	This stage can alter a loose source route computed so far to loadbalance among switches and links.
ComputeRoute	This stage completes route(s). If previous stages generated multiple routes, this stage selects a final one.
TriggerFlowOut	This stage triggers Flow_Out event with the computed route.

Table 3: Ripcord's routing pipeline stages. Earlier stages in the table cannot appear later in routing pipeline. Each routing module should be in one of these stages.

that switch, source MAC and IP addresses, and destination MAC and IP addresses.

4.2 Topology

When Ripcord starts, this module loads the topology from a configuration file. We assume the topology is known in advance, has a regular layered structure (*e.g.* tree, multi-root tree, fat-tree, Clos, ...), and is relatively static. Each layer is assumed to consist of identical switches; but the number of layers, ports and link speeds may vary. The regular structure makes it quick and easy for the routing engines to traverse the topology. Routing engines may view the entire topology, or be restricted t o view only the part of the topology they control. The module has APIs to filter by layer or power status, or return the physical port numbers connecting two switches. See Figure 4 and Figure 5 below for examples of pre-existing Ripcord topologies.

4.3 Authenticator-Demultiplexer

When Ripcord starts, this module builds a lookup table from the configuration and policy database, to map incoming traffic to the correct tenants. The process is triggered by a new `PACKET_IN` event when a switch doesn't recognize a flow. The Authenticator-Demultiplexer generates a `FLOW_IN` event and hands the App Engine an ID identifying which tenant(s) to alert.

4.4 App Engine and Management Apps

When Ripcord starts this module instantiates the management application for each tenant; Figure 2 shows how a management application is configured. In the example, the App Engine instantiates a Python class `ripcord.apps.PortLand` and assigns it AppID 1. AppID is the demultiplexing key sent by the Authenticator-Demultiplexer module.

The App Engine is responsible for dispatching `FLOW_IN` events to the correct tenant management application. The App Engine instantiates applications without knowing their internal implementation, and so is independent of the details of each tenant. A management application is free to implement whatever it chooses, so long as it provides an event handler for `FLOW_IN`.

For example, in our implementation of PortLand, the management application performs ARPs and maintains the AMAC-PMAC mapping table. A management application may tag an event with additional information for its routing modules; by default the event is propagated to the routing engine when the management application returns `CONTINUE`.

```
"apps": [
 {"id": 1,
  "name": "ripcord.apps.PortLand",
  "param": ["firewall=false", "verbosity=debug"],
  "routing": {
   "modules": [
    {"name": "ripcord.routing.PLComputeRoutes",
     "param": ["max_selection=4"]},
    {"name": "ripcord.routing.PLPickRoute",
     "param": ["selection_criteria=random"]},
    {"name": "ripcord.routing.PLOpenFlowTrigger",
     "param": []}
   ]}
 }]
```

Figure 2: App configuration example (*PortLand*). Each app is assigned a unique app identifier. It also specifies a name in the form of a path to Python class and routing pipeline in the form of a list of routing modules.

```
"default": {
 "routing": {
  "expandable": true,
  "modules": [
   {"name": "ripcord.routing.FixSwitch",
    "param": []},
   {"name": "ripcord.routing.InsertMB",
    "param": ['10.0.0.2', '10.0.0.3']}
  ]}
}}
```

Figure 3: Routing policy example.

4.5 Routing Engine and Per-tenant Routing Pipeline

The Routing Engine is responsible — for each tenant — for passing `FLOW_IN` events to the correct sequence of routing modules (based on the AppID). When Ripcord starts, the module generates a pipeline for each tenant from the configuration database. For example, Figure 2 shows how a pipeline of three routing modules is defined for PortLand. The name of each routing module is its Python class name so that the routing engine can correctly locate the module. The routing pipeline can be of any length, although each routing module must be in one of the routing stages in

Table 3 and follows the order of routing stages as described in Section 3. Each stage invokes associated routing modules to progressively complete a source route. The last stage triggers `FLOW_OUT` to convert the computed source route into a series of flow entries and to program switches.

In addition to per-tenant routing pipelines, the data center operator may want to impose *global* routing constraints. For example, traffic for all tenants may be forced to pass through a firewall; or, each tenant may be required to run on isolated paths. Ripcord represents these constraints by a global routing policy. For instance, the policy represented in Table 3 means an application can define its own routing pipeline (i.e., `expandable`), but is subject to two mandatory routing modules.

4.6 Monitoring Implementation

The Monitoring module maintains a snapshot of the state of switches and flows (see Table 4). The module listens for switch join/leave events, and periodically collects switch-level aggregate statistics, flow table statistics, and port statistics.

Fields	Semantics
dpid	switch id
collection_epoch	collection cycle
epoch_delta	distance from previous cycle
collection_timestamp	time captured
ports_active	number of active ports
number_of_flows	flows currently active
bytes_in_flows	size of active flows
packets_in_flows	packets in active flows
total_rx_bytes	total bytes received
total_tx_bytes	total bytes transmitted
total_rx_packets_dropped	receive drops
total_tx_packets_dropped	transmit drops
total_rx_errors	receive errors (frame,crc)
total_tx_errors	transmit errors
delta_rx_bytes	change in bytes received
delta_tx_bytes	change in bytes transmitted
delta_rx_packets_dropped	change in receive drops
delta_tx_packets_dropped	change in transmit drops
delta_rx_errors	change in receive errors
delta_tx_errors	change in transmit errors

Table 4: Information included in a monitoring snapshot.

The module keeps snapshot histories - the configuration file determines the size of the history, the collection frequency, and where old snapshots should be stored. The module also provides an API for *active* statistics collection (Table 5), for detailed metrics of switch and flow performance. Hence, it can be used to build high-level modules to visualize the entire network or for troubleshooting.

4.7 Flow Installer

When the route has been decided, the switches need to be updated. The Flow Installer module takes `FLOW_OUT` events, and generates binary OpenFlow control packet(s) which are passed to NOX for delivery to the correct switches. The `FLOW_OUT` event includes the <header match, action> pair for each switch the flow traverses.

Functions	Roles
get_all_switch_stats(swid)	returns all snapshots for a switch
get_latest_switch_stats(swid)	returns last snapshot for a switch
get_all_port_capabilities(swid)	returns the port capability map for SW
get_port_capabilities(swid,port_id)	returns capabilities of a specific port
get_flow_stats(swid, flow_spec)	returns specific flow statistics

Table 5: API exposed by the monitoring module for active statistics collection.

4.8 Case studies

To illustrate further, we describe how we implemented three routing engines in Ripcord: Proactive, VL2 and PortLand. As a metric of complexity, Table 6 reports the lines of code needed for each implementation.[1]

Implementation	Lines of code
Proactive	200
VL2	576
VL2 w/ middlebox traversal	616
PortLand	627

Table 6: Lines of code of sample routing implementation

Proactive.

This is the simplest base design. Host addresses and locations are loaded from the topology database, paths are chosen using spanning-tree, hashes, or random selection, and corresponding flow entries are installed into the switches. This eliminates flow setup time for applications which cannot tolerate reactive flow installation, at the expense of more entries in the flow table. As an extension—although really as a baseline—Ripcord can also learn MAC addresses, and reactively install flows using the listed path selection methods, similar to today's traditional layer-2 networks.

VL2.

Our VL2 routing engine uses a pipeline with three routing modules. The first module `VL2LoadBalancer` is in the `Loadbalance` stage, and implements the Valiant load balancing. It picks a random intermediate core router (from the set that are up), creating a partial route with the source, intermediary and destination (and optionally other nodes such as middle boxes, or switches added for QoS). We add the optional optimization to route flows directly when the source and destination share a common ToR switch.

[1]Because we do not have the source code from the authors' implementations, our versions are from our own implementations of their schemes.

Switch and Direction	Match	Action
ToR, Up	in_port, src_mac, dst_mac	replace dst_ip with destination's ToR IP addr insert coreID into the highest order byte of src_ip
ToR, Down	in_port, src_mac, dst_mac	restore original dst_ip and src_ip
Aggregation, Up	in_port, highest order byte of src_ip	forward to a port
Aggregation, Down	in_port, dst_ip	forward to a port
Core, Down	dst_ip	forward to a port

Table 7: OpenFlow entries realizing compact VL2 routing.

Next, in the `ComputeRoute` stage, the `VL2ComputeRoute` module completes the route by identifying the shortest path from source to intermediary, and intermediary to destination. If there are multiple shortest paths, one is chosen at random (but only if the switches are up). If a switch is marked down (e.g. for maintenance) it is not used.

Finally, `VL2Open-FlowTrigger`, calculates the flow entries to realize the chosen route. We use the design described in [16] because it is simple, and supports middle-box traversal (see Table 7).

Comparing VL2 as defined by its authors with VL2 implemented on Ripcord, we make the following observations. VL2 uses double IP-in-IP encapsulation to route packets from the source to the core switch (anycast and ECMP), and then onto the destination ToR. In Ripcord, our implementation simply overwrites the destination IP address with the IP address of the destination's top-of-rack switch (ToR), and explicitly routes it via a randomly chosen core switch. In VL2, the destination's ToR decapsulates the packet to restore its original form, whereas we directly instruct the destination's ToR to overwrite the IP addresses with original values. The implementation is different, but the outcome is identical.

Just as in VL2, ARP packets are not broadcast to the whole network, but are forwarded to the controller; the management application handles them and replies directly to the source host. Unknown broadcast types can be rate limited, or sent to a host to be satisfied.

PortLand.

PortLand routes traffic by replacing the usual flat MAC destination address (AMAC) with a source-routed pseudo-MAC (PMAC). The PMAC encodes the location of the destination. The source server is "tricked" into using the PMAC when it sends an ARP — a special fabric manager replies to the ARP with the PMAC instead of the AMAC. The egress ToR switch converts the PMAC back into the correct AMAC to preserve the illusion of transparency for the unmodified end host.

Portland's fabric manager is centralized, and is naturally implemented as a Ripcord management application. The application assigns each AMAC a PMAC based on its ToR switch. Like in PortLand, ARP requests are intercepted and the management application replies (without routing the ARP request).

PortLand routes flows with a pipeline of three routing modules: `PLComputeRoutes`, `PLPickRoute` and `PLOpenFlowTrigger`. Although the modules are sufficient to implement the PortLand's routing, we allow it to be extended with other routing modules (e.g., middlebox interposition module or load balancer). Hence, `PLComputeRoutes`, which belongs to the `ComputeRoute` stage, first examines loose source routes computed by the previous routing stages. If no route is given, it takes a pair of ingress switch and the ToR switch of destination address as a loose source route. Then, it completes each loose source route by computing a shortest path between each two consecutive hops in the source route. Hence, this routing module results in a list of complete source routes. `PLPickRoute` is also in the `ComputeRoute` stage and it randomly selects a route among those computed by `PLComputeRoutes`. Finally, `PLOpenFlowTrigger` converts the selected route into a sequence of flow entries to be installed in switches along the path.

The ingress ToR replaces the source AMAC with the source PMAC for the return journey. Aggregate switches and core switches route solely based on the destination PMAC. Because our OpenFlow implementation does not support longest prefix matching on MAC addresses, we currently match full destination address. A flow entry in the egress ToR switch is to restore the destination PMAC back to AMAC.

4.9 Additional Capabilities

Because of its fine-grained control over routing, Ripcord can do many things a current data center network cannot. We describe some examples below.

Middle-box Traversal..

Flows can easily be routed through arbitrary middle-boxes by inserting a waypoint in a loose source route (in the `InsertWayPoints` routing stage). In the `ComputeRoute` stage, the complete path is calculated to traverse the waypoints. As an experiment, we implemented a routing module `VL2MiddleBoxInserter` to insert a random middle-box (specified in a configuration file) into the VL2 routing pipeline. Thus, the complete `VL` pipeline becomes:

If the middle box doesn't modify the packet header,

```
VL2MiddleBoxInserter
  ⟹VL2LoadBalancer
    ⟹VL2ComputeRoute
      ⟹VL2OpenFlowTrigger
```

our implementation handles an arbitrary number of middle-boxes per path. If the packet header is changed, the portion of the route after the middle-box needs to be recomputed. Alternatively, we could define a model for each middle-box class. While out of the scope of this paper, [10] indicates that this approach has potential.

Seamless fail-over..

Topology changes are detected by the Topology Database and any management application affected by the change is notified, so it can take remedial action.

4.10 Development Tools

Although Ripcord is designed to be easy to port to physical hardware, a large scalable data center is beyond the budget of most researchers. Scale limitations may only expose themselves in larger topologies, therefore we need tools to test designs at scale — if only functionally. The development of Ripcord led to two new tools to address these issues, which may have value outside of data center network development.

4.10.1 Mininet

Emulating large networks is not easy. Existing solutions for emulating OpenFlow networks typically uses one VM per switch interconnected by Virtual Distributed Ethernet. Each VM consumes a lot of memory (32MB for a reduced Debian kernel), and realistically only about fifty VMs (switches) can be emulated per server. Added to the time to reboot a VM, it is hard to emulate reasonably sized networks.

To help Ripcord users develop new tenants, we used Mininet - a new tool for emulating large networks on a single PC. Mininet can support thousands of OpenFlow switches on a single Linux server (or VM), using a combination of Linux network namespace virtualization and Linux virtual ethernet pairs [15]. Each virtual host can have its own IP and ethernet port, while each software switch can have its own ethernet connections. Mininet builds this virtual topology from a Ripcord topology description by creating Ethernet pairs, then moves each endpoint into the correct namespace. The standard OpenFlow Linux reference switch supports multiple kernel datapaths, so packets sent across multiple network hops remain in the kernel. As a result, Mininet cross-section bandwidth exceeds 2 Gb/s on a well-provisioned machine. Hosts and switches share the same code, and so the incremental cost of an additional switch or host is small. A new topology or routing engine can be booted in seconds (rather than minutes). Mininet is available for Ripcord users to develop new tenants. It also proved invaluable for developing Ripcord itself.

Of course Mininet uses software-based switches and so does not provide performance fidelity. For valid performance results, you need hardware.

4.10.2 Virtual-to-Physical Mapping

When we have a working implementation (verified with Mininet), we need to transfer it to hardware. Ideally, we would have access to a huge network of switches each with large numbers of ports. Given this is unlikely, we can slice a physical switch into multiple “virtual” switches. Some OpenFlow switches can be sliced by physical port. For example, a k=4 three layer Fat Tree, which requires twenty 4-port virtual switches, can be emulated by two 48-port physical switches and a number of physical loopback cables. Unfortunately, not every OpenFlow switch supports slicing.

Instead, we chose to slice switches at the controller, by implementing a virtual-to-physical mapping layer between Ripcord components and the NOX API. Since in Ripcord the base topology is known in advance, the mapping can be statically defined. The result is that Ripcord routing engines and applications use virtual addresses, while NOX sees physical addresses. For example, when a switch connects, it has an ID that must be translated from physical to virtual, which may cause one physical switch join event to become multiple virtual switch join events. Almost every OpenFlow message type must undergo this virtual-physical translation in both directions, including flow modifications, packet ins, packet outs, and stats messages. In many ways the slicing layer resembles FlowVisor [14] which also sits between the switch and controller layers.

Ripcord’s slicing layer has been used to build k=4, 80-port Fat Trees from a range of hardware configurations, including two 48-port switches, one 48-port switch and two 24-port switches, and even from eight 4-port switches combined with a 48-port switch, for the testbed described in Section 5.

5. EVALUATION

We evaluate Ripcord against its intended purpose, to evaluate and compare different approaches in a consistent way. With this goal in mind, we demonstrate three routing engines (All Pairs Shortest Path [APSP], PortLand, VL2) and an application, Middlebox Traversal. We evaluate each one on the Mininet software emulator, and then deploy it on a hardware testbed. We evaluate relative differences between implementations, looking at how flow setup delays and switch state requirements vary.

5.1 Software testbed

The software testbed is an instance of Mininet running inside a Debian Lenny virtual machine, allocated one CPU core and 256MB of memory. Mininet spawns kernel-mode software OpenFlow reference switches, running version 0.8.9r2. The controller is NOX 0.6, with Ripcord core components and applications on top. NOX runs on a Debian VM on an Ubuntu 8.04 machine with 4GB of RAM and an Intel Q6600 quad-core 2.4 GHZ CPU.

5.2 Hardware testbed

The hardware testbed implements a k=4 three-layer Fat Tree running at 1Gb/s. Aggregation and core switches are implemented by slicing a 48-port 1GE switch (Quanta LB4G) running OpenFlow. Eight 4-port NetFPGAs act as edge switches. The OpenFlow implementation on the NetFPGAs can rewrite source and destination MAC addresses at line-rate (required for PortLand) and can append, modify and remove VLAN tags to distinguish multiple simultaneous routing engines.

5.3 Experiments on Software Testbed

Our first topology is the k=4 Fat Tree in the left-hand side of Figure 4. The graph on the right hand side show a CDF of the ping times from the left-most host in our topology to all other hosts, sending one ping at a time. The graph contains curves from all four routing engines: APSP, VL2, MBT (Middle Box Traversal using VL2), and PortLand. We further break up VL2, MBT, and PortLand into two separate configurations. In the first configuration (*Hot*) permanent flow entries are pre-installed into switches, resulting in no trips to the controller. In the second configuration (*Cold*) the ARP caches are filled, but no flow entries are pre-installed into the switches. When a packet arrives at an edge switch and there is no matching flow, it heads for the controller, where its trip through the routing engine pipeline may generate flow entries for multiple switches. The APSP routing engine only supports Hot operation. It does no reactive lookups and simply pushes out all possible paths directly to the switches.

APSP experiences slightly higher delay compared to PortLand (Hot) and VL2 (Hot), due to the higher number of wildcard flow entries in the software switch, which are scanned linearly to determine a match. PortLand (Hot) and VL2 (Hot) both show similar curves, and since they leverage topology information to reduce flow state, switch traversal is faster. MBT (Hot) is roughly one and a half times worse than VL2 (Hot) because it must traverse a middle box in both directions, and experiences delays from our repeater agent running on the middle box. VL2 (Cold), MBT (Cold), and PortLand (Cold), as expected, trail by over two orders of magnitude, because both the ping request and response must pass up to the controller and back. Note that these numbers are from an unoptimized Python implementation, running on a single thread, with a worst case traffic pattern. The specific ping time of 10 ms is unimportant; our goal here is functional correctness. For example, we can see from the graph that our PortLand implementation is slower than the VL2 implementation in the *Cold* setting, possibly because of its unoptimized memory accesses (PMAC-AMAC mapping table) and the latency to install more entries at core switches and aggregation switches.

Our second topology is a Clos network shown on the left-hand side of Figure 5. The graph shows a CDF of ping times from the left-most host in the topology. This is the topology used in the VL2 paper's evaluation, except instead of a mix of 10 Gb/s and 1 Gb/s links, we have one link speed of whatever the CPU will support. The general trends are the same; flow setups are two orders of magnitude more expensive than forwarding. Middlebox traversal has an unexpected knee. Our hunch is that the additional flow entries required by multiple hops exceeds the CPU cache, which given linear lookups, would cause poor cache locality. These graphs are useful for comparing the different routing engines, but clearly CPU overheads from running in software result in low performance fidelity.

(Note to reviewer: APSP is not present; it will be added in the final version of the paper.)

5.4 Experiment on Hardware Testbed

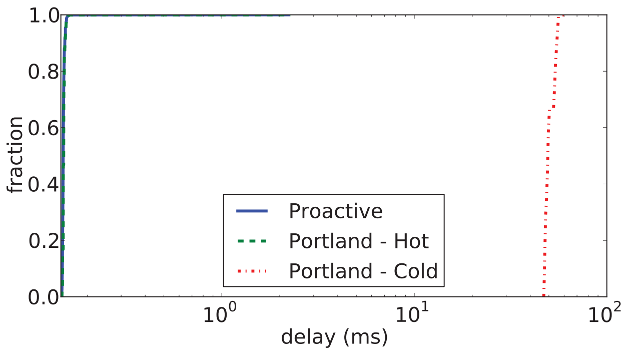

Figure 6: CDF of 1 to many host ping delays on a Quanta 48x1GE switch + NetFPGA topology

The hardware testbed described in Section 5.2 implements a k=4 Fat Tree, with twelve core and aggregation switches and eight NetFPGAs for edge switches. Individual virtual switches are connected together via physical loopback cables, and all packets are processed in hardware at line-rate. Both switch types use the OpenFlow 0.8.9 reference distribution.

Figure 6 confirms our expectation of lower variance in hardware than in software — overall we can expect greater performance fidelity. PortLand (Hot) and Proactive show identical delay curves, with minimal variation. (PortLand (Cold) exposes a limitation of our edge switch implementation, and needs to be corrected).

Next, we attempt to gain insight into tradeoffs between state management and flow setup delay.

5.5 Flow Table Size

To test our implementation and to illustrate the consequences of choosing different flow entry timeout intervals, we preformed the following two tests. First, we run our VL2 Ripcord application on the Clos topology on the software testbed, with permanent flow entries, and perform an all-to-all ping. After the test, we query all the switches and record the number of flows entries in each switch. Table 8 presents the data. The choice of a CRC-based hash function

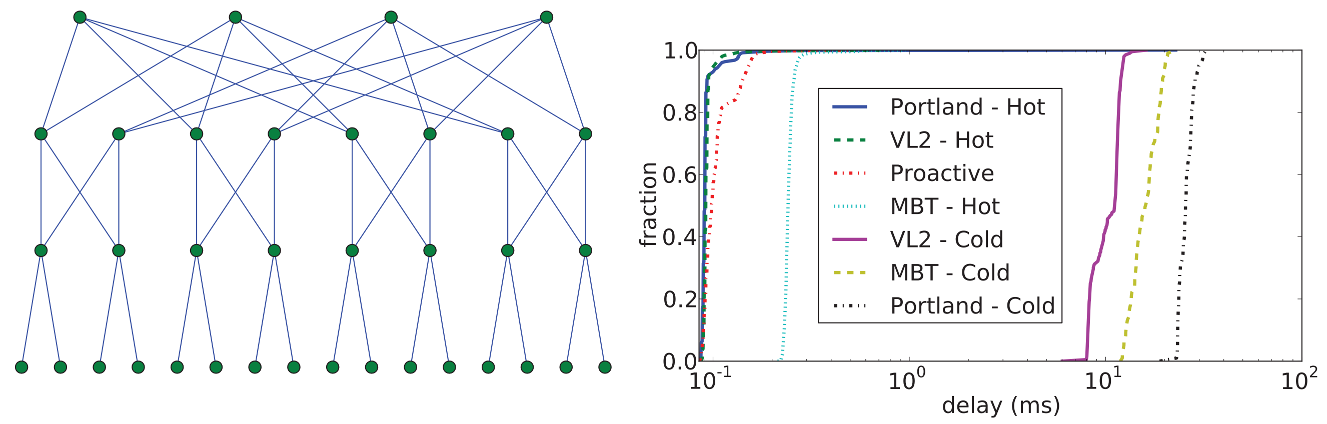

Figure 4: Fat Tree topology and CDF of 1-to-many host ping delays on Mininet using the topology. Leaf nodes in the topology represent end hosts.

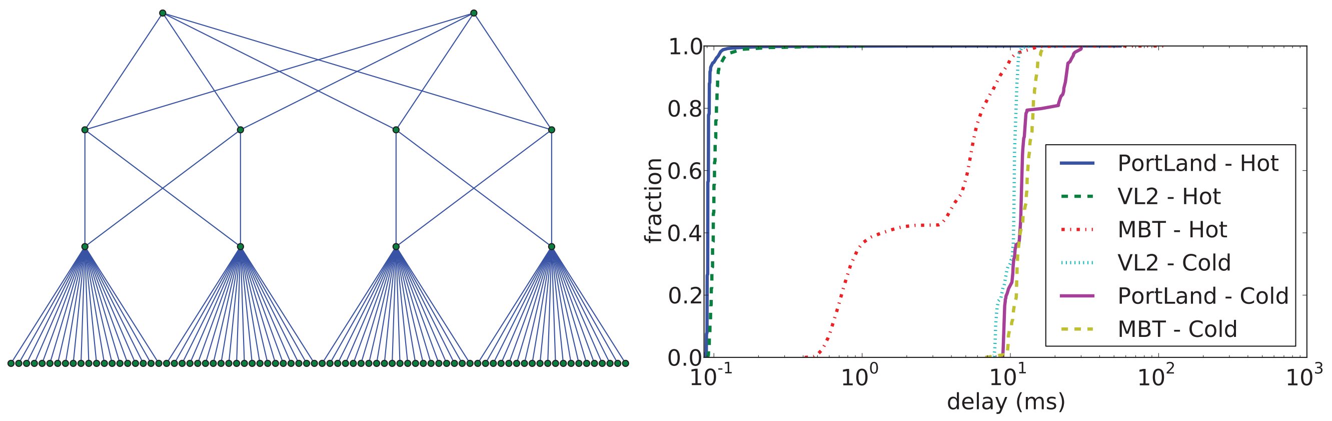

Figure 5: Clos topology and CDF of 1-to-many host ping delays on Mininet using the toplogy. Leaf nodes in the topology represent end hosts.

to pick a path, combined with a symmetric test and topology, yields evenly distributed flow entries at each level. The ToR switch has many entries because it keeps per flow state. As discussed in section 6, one way to solve this problem is to move the per-flow packet manipulation functionality to the hosts. Indeed, [5] performs the IP-in-IP encapsulation at the host. In the context of OpenFlow, this solution can be best realized by running an Open vSwitch [13] at the hosts. Ripcord would control it just like any other OpenFlow switch.

The second experiment is the same as the first one, except that the idle flow timeout interval is set to 3 seconds. Every 10 ms, we poll the switches and record the number of flow entries. Table 9 shows the average, maximum, and 95th percentile of the number of flow entries in each switch. As the table shows, because only a few entries are actively used at any given time, expiring the idle ones dramatically reduces the table size.

Switch Type	# Instances	# Entries (per instance)
Core	2	4
Aggregation	4	10
ToR	4	2780

Table 8: The number of flow entries installed at each switch by VL2 implementation with no flow idle timeout.

Switch ID	Type	Avg # Entries	Max	95th percentile
0	Core	3.88	4	4
1	Core	3.86	4	4
2	Aggr	7.61	10	8
3	Aggr	7.50	10	9
4	Aggr	7.73	10	9
5	Aggr	7.75	10	9
6	ToR	142.78	336	74
7	ToR	140.00	292	190
8	ToR	141.40	294	64
9	ToR	143.69	325	154

Table 9: The number of flow entries installed at each switch by VL2 implementation with flow idle timeout of 3 sec.

5.6 Running Simultaneous Ripcord Applications

To test Ripcord's ability to run multiple management applications simultaneously, we run several experiments with both VL2 and Portland controlling a subset of traffic. We randomly divided the hosts into two groups and configured the hostmanager to classify the traffic within the first group as belonging to VL2 and the traffic within the second group as belonging to Portland. In each experiment, we run pings between hosts in each group and repeated the experiments on both Clos and FatTree topologies.

Figure 7 illustrates a simplified scenario and shows the informaton flow. Hosts H1 and H3 belong to VL2 and

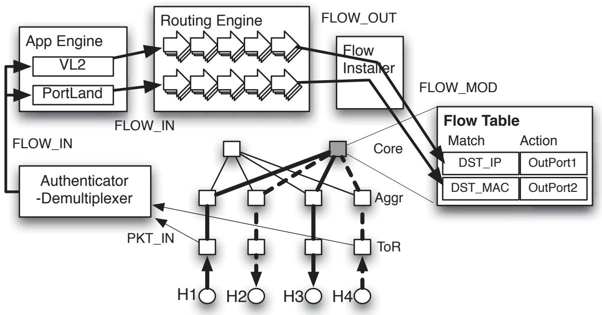

Figure 7: Diagram illustrating simultaneous running of multiple Ripcord Management Applications

hosts H2 and H4 belong to Portland. H1 is sending packets to H3 (path is shown in bold), and H4 is sending packets to H2 (path is shown with a dashed line). When the first packets from these flows hit the first hop ToR switches, the switches do not have any matching entry and hence they forward the packets to the controller. There Authenticator-Demultiplexer classifies the traffic and delivers the `FLOW_IN` event to the appropriate application, which eventually installs the necessary entries in all switches on the path.

As the figure shows, some switches can be common to both paths. These switches will contain flow entries for both applications. Hence it is critical to make sure that applications do not install conflicting entries. In general, this separation can be achieved by tagging traffic belonging to different applications with distinct VLAN IDs. In our case, because of the specifics of Portland's and VL2's implementations, their flow entries could not possibly collide and we did not implement VLAN tagging because our hardware testbed did not support this optional functionality.

6. SCALABILITY

A primary goal of Ripcord is to provide a research platform for data center network architecture experimentation. To this end, a fundamental requirement for the platform is that it not hinder experiments with reasonable network sizes or limit the experimentation to artificially small data center networks, which would have little value for the community. After all, many of the challenges in data center networking stem directly from their scaling requirements.

We consider two of the primary scalability concerns with dynamic state in the network and introducing centralization into the architecture: *a)* the number of concurrent, active flows may exceed the capacity of the switches, and *b)* a single controller may become overwhelmed by the number of flow setup requests. We consider each of these cases in turn.

While it is our experience that long-lived, any-to-any communication in a data center is rare, there still exists the potential for exhausting switch sate in the network. We first point out that for most approaches, this problem is limited to the ToR switches as aggregation and core layers can often handle flows in aggregate. Secondly, if per-transport flow policy is not required, flows can be set up on a per-source/destination pair basis which is limited by the number of servers attached to the switch. Today chipsets are available which can support tens of thousands of flows, which is suitable for moderate to large size workloads.

Another approach described in [13] proposes pushing network switching state into the hypervisor layer on end hosts to overcome the hardware limitations of ToR switches. With such a software-based switch, the issue of exhaustion at first-hop switch can also be alleviated.

Flow setups can become a scalability bottleneck of the platform in terms of flow setup latencies and throughput. We'll discuss the scalability of setup latency first. While many of the data center applications like MapReduce tolerate delays on the order of tens of milliseconds, applications that have strict latency requirements may not tolerate any extra delay incurred by setting up flow entries. For such demanding applications, assuming a large enough flow table at the ToR switch either in hardware or software, Ripcord can pre-install flow entries towards all possible network destinations into the switch. In this proactive mode, therefore, the network virtually behaves as if MPLS-tunneled, and applications are not exposed to additional latency for flow setup. If the luxury of large flow tables is not available, additional decision hints such as application priorities or communication pair likelihood can be used to select preloaded flows. In our current prototype, such information could be stored in the configuration database module and/or topology database module.

Scaling flow setup throughput beyond the limits of a single controller requires that the platform support multiple controller instances. However, before diving into the details, it is worthwhile to explicitly differentiate between the scalability requirements research and production quality platforms for data center networks.

The goal of Ripcord, at least in its current incarnation, is not to provide a researcher with a production quality implementation. The implementation lacks in many aspects, such as in efficiency, robustness, and importantly Ripcord is built around a simple, centralized, single-host state sharing mechanism. If subjected to the extreme scalability and availability requirements of data center networks in production, this mechanism is clearly insufficient.

In research experiments the lack of extreme scalability and availability properties is a non-issue as long as the programming abstractions offered for the application developers are similar to the ones, which would be provided in systems designed to scale for production use. To this end, we briefly overview the scalability approaches of both Portland and VL2:

- Portland centralizes all state sharing into a fabric manager component, which manages the switch modules over OpenFlow. As such, the fabric manager corresponds directly with a single Ripcord controller instance.
- VL2 assumes no single, centralized controller instance, but uses a distributed directory system to share state among multiple controllers (agents). The directory system is essentially a strongly-consistent, reliable and centralized store, which has an eventually consistent caching layer for reads on top.

The descriptions above suggest that the design of Ripcord is well aligned with the scalability approaches of these individual proposals. In particular, Ripcord can provide the platform for centralized single-controller designs like PortLand, while for VL2 like designs, which rely on a distributed state sharing mechanism, Ripcord can provide a single-host configuration database with the identical semantics. This is clearly not scalable (nor highly-available), but the programming abstractions within the controllers connected to the database will be the same as if a distributed state sharing mechanism were used. Eventually, as Ripcord matures, it could also replace this centralized, single-host configuration database with a distributed database.

It is still an open question to what extent the platform should scale to provide value to different communities. The research community, which principally targets devising, rapidly developing, and evaluating new ideas, could be an immediate beneficiary, even with networks on smaller scales. For those networks, even our current Ripcord prototype can be an ideal vehicle since it is capable of emulating a 100-node data center on a modern laptop computer and it supports seamless porting from software emulation to real hardware testbeds. For a designer of a production data center seeking radically new approaches to improve networking performance, or trying to introduce competitive features, Ripcord may also prove valuable, as long as the size of the testbed is not on the order of the production network.

7. RELATED WORK

Ripcord is built on top of programmable switches and a logically centralized control platform. In our prototype we use OpenFlow [11] switches and NOX [8], an open-source OpenFlow controller. While NOX was designed to be a general controller platform applicable to many environments, Ripcord was designed around needs specific to the datacenter. This includes providing infrastructure for managing structured topology, location independence, and service quality as well as exposing higher-level abstractions, such as tenants.

While we chose NOX in large part due to our familiarity with it, Ripcord could also have been implemented within other centralized network control platforms such as Tesseract [6] or Maestro [3]. Like NOX, both of these projects provide centralized development platforms on top of which network control logics can be implemented.

The use of OpenFlow was similarly a matter of familiarity and convenience. However, other than the table and port abstractions, no low-level details of OpenFlow are exposed through Ripcord. Therefore, the Ripcord design should be compatible with other programmable datapath technologies that maintain table-entry level control of the network.

Having described related technologies for programmable switches and controllers we now discuss Ripcord in the context of recent data center networking proposals (e.g., VL2 [5], Monsoon [7], BCube [9], PLayer [10], and PortLand [12]). We note that each of these networking proposals presents a solution based on specific requirements, some of which overlap across solutions, but may be prioritized differently in each solution. As a consequence specific architectural choices are made that may make it difficult to accommodate new requirements, changes to data center environments or modifications to the solution that attempt to tailor/tweak it for another data center environment.

Ripcord is not in direct competition with any of these networking proposals, rather it provides a platform that allows network administrators to experiment with one or more of data center networking proposals (side-by-side if necessary), make modifications and evaluate the proposal in their own data center environments. Further, whereas Ripcord does not include or propose any novel distributed algorithms for managing data center networks, we posit that it provides a suitable platform for experimentation in this space based on its modular design.

Ripcord is also similar in spirit to the broad testbased work which allows multple experiments to share the same infrastructure. Notable recent proposals include VINI [2], and FlowVisor [14]. Ripcord differs from these and similar proposals in that our goal is to construct a modular platform *at the control level* which provides primitives useful in the data center context. To this end, we have designed multiple components (such as the topology and monitoring interfaces) which aid (and limit!) the applications suitable for running on Ripcord.

8. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 63–74, New York, NY, USA, 2008. ACM.

[2] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. of SIGCOMM*, Pisa, Italy, 2006.

[3] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. S. E. Ng. The Preliminary Design and Implementation of the Maestro Network Control Platform. Technical Report TR08-13, Rice University, 2008. Available at`http://www.cs.rice.edu/~eugeneng/papers/Maestro-TR.pdf`.

[4] Cisco. Data Center Ethernet. `http://www.cisco.com/go/dce`.

[5] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of SIGCOMM*, Barcelona, Spain, 2009.

[6] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR*, 35(5):41–54, 2005.

[7] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 57–62, New York, NY, USA, 2008. ACM.

[8] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

[9] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, pages 63–74, 2009.

[10] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):51–62, 2008.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.

[12] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 39–50, New York, NY, USA, 2009. ACM.

[13] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *8th ACM Workshop on Hot Topics in Networking (Hotnets)*, New York City, NY, October 2009.

[14] R. Sherwood, M. Chan, G. Gibb, N. Handigol, , T.-Y. Huang, P. Kazemian, M. Kobayashi, D. Underhill, K.-K. Yap, G. Appenzeller, and N. McKeown. Carving Research Slices Out of Your Production Networks with OpenFlow, 2009.

[15] SourceForge. Linux containers - network namespace configuration. `http://lxc.sourceforge.net/network/configuration.php`.

[16] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In *8th ACM Workshop on Hot Topics in Networking (Hotnets)*, New York City, NY, October 2009.

[17] J. Touch and R. Perlman. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement. RFC 5556 (Informational), May 2009.

APPENDIX

NOX and OpenFlow OpenFlow is a vendor-agnostic interface to control network switches and routers. In particular, it provides an abstraction of the flow-tables already present in most devices - they were originally placed there to hold firewall rules. OpenFlow allows rules to be placed in a table, consisting of a ¡header pattern, action¿ pair. If an arriving packet matches the header pattern, the associated action is performed. Actions are generally simple, such as forward to a port or set of ports, drop, or send to the controller. NOX is a network-wide operating system that controls a collection of switches and routers using the OpenFlow protocol. NOX provides a global view of the topology, and presents an API to hosted applications to both view and control the network state. A hosted application might reactively respond to new flows, choose whether to allow them and then install rules to determine their path. Or it could proactively add rules to define how new flows will be routed.