Orion: Google's Software-Defined Networking Control Plane

Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat, *Google*

https://www.usenix.org/conference/nsdi21/presentation/ferguson

This paper is included in the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Orion: Google's Software-Defined Networking Control Plane

Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin
Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi
Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni
Amr Sabaa, Shidong Zhang, Min Zhu, Amin Vahdat
Google
orion-nsdi2021@google.com

Abstract

We present Orion, a distributed Software-Defined Networking platform deployed globally in Google's datacenter (Jupiter) and Wide Area (B4) networks. Orion was designed around a modular, micro-service architecture with a central publish-subscribe database to enable a distributed, yet tightly-coupled, software-defined network control system. Orion enables intent-based management and control, is highly scalable and amenable to global control hierarchies.

Over the years, Orion has matured with continuously improving performance in convergence (up to 40x faster), throughput (handling up to 1.16 million network updates per second), system scalability (supporting 16x larger networks), and data plane availability (50x, 100x reduction in unavailable time in Jupiter and B4, respectively) while maintaining high development velocity with bi-weekly release cadence. Today, Orion enables Google's Software-Defined Networks, defending against failure modes that are both generic to large scale production networks as well as unique to SDN systems.

1 Introduction

The last decade has seen tremendous activity in Software-Defined Networking (SDN) motivated by delivering new network capabilities, fundamentally improving network reliability, and increasing the velocity of network evolution. SDN starts with a simple, but far-reaching shift in approach: moving network control, such as routing and configuration management, from individual hardware forwarding elements to a central pool of servers that collectively manage both real-time and static network state. This move to a logically centralized view of network state enables a profound transition from defining pairwise protocols with emergent behavior to distributed algorithms with guarantees on liveness, safety, scale, and performance.

For example, SDN's global view of network state presents an opportunity for more robust network verification and intent-based networking [16, 17]. At a high level, SDN affords the opportunity to transition the network from one consistent state to another, where consistency can be defined as policy compliant and blackhole-free. This same global view and real-time control enables traffic engineering responsive to topology, maintenance events, failures, and even fine-grained communication patterns such that the network as a whole can operate more efficiently and reliably [2, 12, 13]. There is ongoing work to tie end host and fabric networking together to ensure individual flows, RPCs, and Coflows meet higher-level application requirements [1, 11, 22], a capability that would be hard or impossible with traditional protocols. Perhaps one of the largest long-term benefits of SDN is support for software engineering and qualification practices to enable safe weekly software upgrades and incremental feature delivery, which can hasten network evolution by an order of magnitude.

While the promise of SDN is immense, realizing this promise requires a production-grade control plane that meets or exceeds existing network performance and availability levels. Further, the SDN must seamlessly inter-operate with peer legacy networks as no network, SDN or otherwise, operates solely in its own tech island.

In this paper, we describe the design and implementation of Orion, Google's SDN control plane. Orion is our second generation control plane and is responsible for the configuration, management, and real-time control of all of our data center (Jupiter [28]), campus, and private Wide Area (B4 [15]) networks. Orion has been in production for more than four years. The SDN transition from protocols to algorithms, together with a micro-services based controller architecture, enables bi-weekly software releases that together have not only delivered over 30 new significant capabilities, but also have improved scale by a factor of 16, availability by a factor of 50x in Jupiter and 100x in B4, and network convergence time by a factor of 40. Such rapid evolution would have been hard or impossible without SDN-based software velocity

Orion's design centers around a constellation of independent micro-services, from routing to configuration management to network management, that coordinate all state through an extensible Network Information Base (NIB). The NIB sequences and replicates updates through a key-value abstraction. We describe the performance, semantic, and availability

requirements of the NIB and the development model that allows dozens of engineers to independently and simultaneously develop, test, and deploy their services through well-defined, simple, but long-lived contractual APIs.

While Orion has been a success at Google, neither our design nor the SDN approach are panaceas. We describe four key challenges we faced in Orion—some fundamental to SDN and some resulting from our own design choices—along with our approach to addressing them:

#1: Logically centralized control require fundamentally high performance for updates, in-memory representation of state, and appropriate consistency levels among loosely-coordinating micro-service SDN applications.

#2: The decoupling of control from hardware elements breaks fate sharing in ways that make corner-case failure handling more complex. In particular, control software failure does not always mean the corresponding hardware element has failed. Consider the case where the control software runs in a separate physical failure domain connected through an independent out-of-band control network. Either the physical infrastructure (control servers, their power or cooling) or control network failure can now result in at least the perception of a sudden, massively correlated failure in the data plane.

#3: Managing the tension between centralization and fault isolation must be balanced carefully. At an extreme, one could imagine a single logical controller for all of Google's network infrastructure. At another extreme, one could consider a single controller for every physical switch in our network. While both extremes can be discarded relatively easily, finding the appropriate middle ground is important. On the one hand, centralization is simpler to reason about, implement, and optimize. On the other, a centralized design is harder to scale up vertically and exposes a larger failure domain.

#4: In a global network setting, we must integrate existing routing protocols, primarily BGP, into Orion to allow interoperation with non-SDN peer networks. The semantics of these protocols, including streaming updates and fate sharing between control and data plane, are a poor match to our choice of SDN semantics requiring adaptation at a number of levels.

This paper presents an introductory survey of Orion. We outline how we manage these concerns in its architecture, implementation, and evolution. We also discuss our production experiences with running Orion, pointing to a number of still open questions in SDN's evolution. We will share more details and experiences in subsequent work.

2 Related Work

Orion was designed with lessons learned from Onix [18]. Unlike Onix's monolithic design with cooperative multi-threading, Orion introduced a distributed design with each application in a separate process. While Onix introduced a NIB accessible only to applications in the same process, Orion's is accessible by applications within and across domains, providing a mechanism for hierarchy, which few existing controllers incorporate (Kandoo [35] being an exception). Hierarchy enabled fabric-level drain sequencing [1] and optimal WCMP-based (Weighted Cost Multi-Pathing) routing [36].

We distribute Orion's logic over multiple processes for scalability and fault-tolerance, a feature shared with other production-oriented controllers such as ONOS [4] and OpenDaylight [24], and originally proposed by Hyperflow [30]. Unlike our previous design, Orion uses a single configuration for all processes, applied atomically via the NIB, precluding errors due to inconsistent intended state.

Orion uses database-like tables to centrally organize state produced and consumed by SDN programs, a feature shared with a few other OpenFlow controllers such as ONOS [4], Flowlog [27], and Ravel [32]. The combination of all of these techniques – hierarchy, distribution, and database-like abstractions – allowed Orion to meet Google's availability and performance requirements in the datacenter and WAN.

While Orion is an evolution in the development of OpenFlow controllers, its modular decomposition of network functions (e.g., routing, flow programming, switch-level protocols, etc.) is a design goal shared with pre-OpenFlow systems such as 4D/Tesseract [33] and RCP [6]. Single-switch operating systems that similarly employ microservices and a centralized database architecture include Arista EOS [3] and SONiC [25].

3 Design Principles

We next describe principles governing Orion's design. We established many of these during the early stages of building Orion, while we derived others from our experience operating Orion-based networks. We group the principles into three categories: environmental – those that apply to production networks, architectural – those related to SDN, and implementation – those that guide our software design.

3.1 Principles of production networks

Intent-based network management and control. Intent-based networks specify management or design changes by describing the new intended end-state of the network (the "what") rather than prescribing the sequence of modifications to bring the network to that end-state (the "how"). Intent-based networking tends to be robust at scale, since high-level intent is usually stable over time, even when the low-level state of network elements fluctuates rapidly.

For example, consider a situation where we wish to temporarily "drain" (divert traffic away from) a cluster while we simultaneously add new network links to augment the ingress and egress capacity of the cluster. As those new links turn up, the stable drain intent will also apply to them, causing the underlying networking control system to avoid using them.

In Orion, we use an intent-based approach for updating the network design, invoking operational changes, and adding new features to the SDN controller. For example, we capture

[1] Fabric-level drain sequencing refers to redirecting traffic in a loss-free manner, throughout the fabric, away from a target device being drained.

intended changes to the network's topology in a model [26], which in turn triggers our deployment systems and operational staff to make the necessary physical and configuration changes to the network. As we will describe later, Orion propagates this top-level intent into network control applications, such as routing, through configuration and dynamic state changes. Applications react to top-level intent changes by mutating their internal state and by generating intermediate intent, which is in turn consumed by other applications. The overall system state evolves through a hierarchical propagation of intent ultimately resulting in changes to the programmed flow state in network switches.

Align control plane and physical failure domains. One potential challenge with decoupling control software from physical elements is failure domains that are misaligned or too large. For misalignment, consider the case in which a single SDN controller manages network hardware across portions of two buildings. A failure in that controller can cause correlated failures across two buildings, making it harder to meet higher-level service SLOs. Similarly, the failure of a single SDN controller responsible for all network elements in a campus would constitute too large a vulnerability even if it improved efficiency due to a centralized view.

We address these challenges by carefully aligning network control domains with physical, storage, and compute domains. As one simple example, a single failure in network control should not impact more than one physical, storage, or compute domain. To limit the "blast radius" of individual controller failures, we leverage hierarchical, partitioned control with soft state progressing up the hierarchy (§5.1). We explicitly design and test the network to continue correct, though likely degraded, operation in the face of controller failures.

3.2 Principles related to an SDN controller

SDN enables novel approaches to handling failures, but it also introduces new challenges requiring careful design. The SDN controller is remote from the network switches, resulting in the lack of fate sharing but also the possibility of not being able to communicate with the switches.

Lack of fate sharing can often be used to our advantage. For example, the network continues forwarding based on its existing state when the controller fails. Conversely, the controller can repair paths accurately and in a timely manner when individual switches fail, by rerouting around them.

React optimistically to correlated unreachability. The loss of communication between controller and switches poses a difficult design challenge as the controller must deal with incomplete information. We handle incomplete information by first deciding whether we are dealing with a minor failure or a major one, and then reacting pessimistically to the former and optimistically to the latter.

We start by associating a ternary health state with network elements: (i) *healthy* with a recent control communication (a switch reports healthy link and programming state with no packet loss), (ii) *unhealthy*, when a switch declares itself to be unhealthy, when neighbouring switches report unhealthy conditions or indirect signals implicate the switch, and (iii) *unknown*, with no recent control communication with a switch and no indirect signals to implicate the switch.

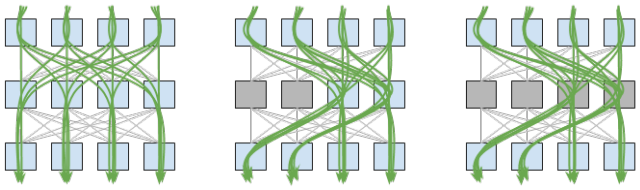

Figure 1: Network behavior in three cases: **Normal** (left): A network with healthy switches. Flows from top to bottom switches use all middle switches. ***Fail Closed*** (mid): With few switches in unknown state (grey), the controller conservatively routes around them. ***Fail Static*** (right): With enough switches in unknown state, the controller no longer routes around newly perceived failed switches.

A switch in the *unknown* state could be malfunctioning, or it could simply be unable to communicate with a controller (a fairly common occurrence at scale). In comparison, the *unhealthy* state is fairly rare, as there are few opportunities to diagnose unequivocal failure conditions in real time.[2]

The controller aggregates individual switch states into a network-wide health state, which it uses to decide between a pessimistic or an optimistic reaction. We call these ***Fail Closed*** and ***Fail Static***, respectively. In *Fail Closed*, the controller re-programs flows to route around a (perceived) failed switch. In *Fail Static*, the controller decides not to react to a switch in an *unknown*, potentially failed, state, keeping traffic flowing toward it until the switch state changes or the network operator intervenes. Figure 1 illustrates an example of normal operation, *Fail Closed* reaction, and *Fail Static* condition.

In *Fail Static*, the controller holds back from reacting to avoid worsening the overall state of the network, both in terms of connectivity and congestion. The trade-off between *Fail Closed* and *Fail Static* is governed by the cost/benefit implication of reacting to the *unknown* state: if the element in the *unknown* state can be avoided without a significant performance cost, the controller conservatively reacts to this state and triggers coordinated actions to steer traffic away from the possible failures. If the reaction would result in a significant loss in capacity or loss in end-to-end connectivity, the controller instead enters Fail Static mode for that switch. In practice we use a simple "capacity degradation threshold" to move from *Fail Closed* to *Fail Static*. The actual threshold value is directly related to: (1) the operating parameters of the network, especially the capacity headroom we typically reserve, for example, to support planned maintenance; (2) the level of redundancy we design in the topology and control

[2] It is not common for a software component to be able to self-diagnose a failure, without being able to avoid it in the first place, or at least repair it. Slightly more common is the ability to observe a failure from an external vantage point, e.g. a neighboring switch detecting a link "going down."

domains. We aim to preserve a certain amount of redundancy even in the face of capacity degradation.

In our experience, occurrences of *Fail Static* are fairly common and almost always appropriate, in the sense that they are not associated with loss of data plane performance. Often *Fail Static* is triggered by failures in the control plane connectivity between the SDN controller and the network elements, or by software failures in the controller. Neither of these directly affect the data plane health.

We view the ability to *Fail Static* as an advantage of SDN systems over traditional distributed-protocol systems. Distributed systems are also subject to some of the failures that could benefit from a *Fail Static* response. However, they are not easily amenable to realize a *Fail Static* behavior because they only have a local view. It is far easier for a centralized controller to assess if it should enter *Fail Static* when it can observe correlated network failures across its entire domain.

Exploit both out-of-band and in-band control plane connectivity. In a software-defined network, a key consideration is connectivity between the "off box" controller and the data plane switches. We must solve the bootstrap problem of requiring a functioning network to establish baseline control. Options include using: i) the very network being controlled ("in-band") or ii) a (physically or logically) separate "out-of-band" network. While a seemingly simple question, considerations regarding pure off-box SDN control, circular dependencies between the control and dataplane network, ease of debuggability, availability, manageability and cost of ownership make this topic surprisingly complex.

The simplest approach is to have a physically separate out-of-band control/management plane network (CPN) for communication between controller and switches orthogonal to the dataplane network. This approach cleanly avoids circular dependencies, keeping the control model simple and enabling easy recovery from bugs and misconfiguration. Ideally, we would like the out-of-band control network to be highly available, easy to manage and maintain, and cost effective. In the end, a separate CPN means installing and operating two distinct networks with different operational models and independent failure characteristics. While failure independence is often a desirable property, subtle or rare failure scenarios mean the entire data plane could go down if *either* the dataplane or control plane fails. We describe our choice of hybrid CPN for Orion in §5.

3.3 Software design principles

Enabling a large-scale software development environment was a key motivation for building our own SDN controller. Critical to the success of SDN is the ability to safely deploy new functionality across the network incrementally with frequent software releases. This, in turn, means that a substantial team of engineers must be able to develop multiple independent features concurrently. The need to scale engineering processes led to a modular system with a large number of decoupled components. At the same time, these components had to interact with one another to realize a tightly-coupled control system reflecting the structure and dynamics of network control. We achieved this goal through:

- a *microservice architecture* with separate processes rather than a monolithic block which we adopted in our first generation SDN controller [18], for software evolvability and fault isolation.
- a *central pub-sub system* (NIB) for all the communication between microservices, which took care of the tightly-coupled interaction across processes.

Failure domain containment (§3.1) imposes an upper limit to the size of control domains. Nevertheless, we were concerned with the performance, scalability, and fault model of a single NIB to coordinate all communication and state within a control domain. We satisfied our performance concerns through benchmarking efforts, and fault tolerance concerns by limiting control domain scope and the ability to fail static, including between control domains.

Based on years of experience, the NIB has been one of our most successful design elements. It manages all inter-component communications, allows us to create a "single arrow of time," establishing an order among the otherwise concurrent events across processes. This brought significantly useful side effects including much improved debuggability of the overall system. It also allows us to store event sequences (NIB traces) in external systems and use them for offline troubleshooting and independent validation of subsystems, which we use in component-level regression testing.

Next, we discuss the principles of intent based control, introduced in 3.1, reconciliation of state as well as the implications of various failure modes in an SDN-based system:

Intent flows from top to bottom. The top level intent for the system as a whole is the operator intent and the static configuration. As intent propagates through the system via NIB messages, it triggers local reactions in subsystems that generate *intermediate intent* consumable by other sub-systems. Higher-level intent is authoritative and any intermediate intent (also known as *derived state*) is rebuilt from it. The programmed switch state is the *ground truth* corresponding to the intent programmed into dataplane devices.

The authoritative intent must always be reflected in the ground truth. The controller ensures that any mismatch is corrected by migrating the ground truth toward the intended state in a way that is minimally disruptive to existing data plane traffic. We refer to this process as "state reconciliation".

Reconciliation is best performed in the controller which has a global view since minimal disruption often requires coordination across switches such that changes are sequenced in a graceful and loop-free manner. Reconciliation is a powerful concept that allows reasoning about complex failure modes such as Orion process restarts as well as lack of fate sharing between the data and control planes.

Availability of high level intent is crucial to keep the top-

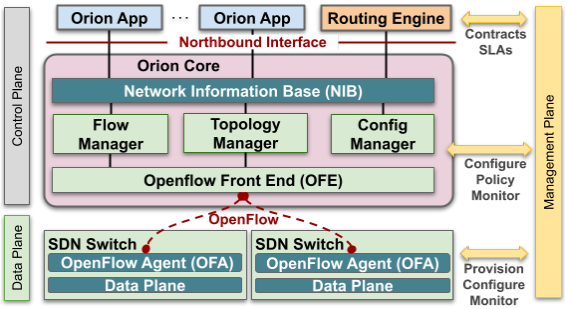

Figure 2: Overview of Orion SDN architecture and core apps.

down intent-based system simple. To achieve this goal we minimize the time when the intent is temporarily unavailable (e.g., because of process restarts or communication failure between components).

4 Architecture

Figure 2 depicts the high-level architecture of Orion, and how it maps to the textbook ONF view [7]. For scalability and fault isolation, we partition the network into domains, where each domain is an instance of an Orion SDN controller.

The data plane consists of SDN switches at the bottom. Orion uses OpenFlow [23] as the *Control to Data Plane Interface (CDPI)*. Each switch runs an OpenFlow Agent (OFA) for programmatic control of forwarding tables, statistics gathering, and event notification. The control plane consists of *Orion Core* in the center and *SDN applications* at the top. The control plane is physically separate from the data plane and *logically* centralized, providing a global view of the domain. Though logically centralized, the Orion Core controls the network through distributed controller processes. The NIB provides a uniform SDN *NorthBound Interface* for these applications to share state and communicate requirements. The Orion Core is responsible for (i) translating these requirements into OpenFlow primitives to reconcile switches' programmed state with intended state and (ii) providing a view of the runtime state of the network (forwarding rules, statistics, data plane events) to applications.

4.1 Orion Core

The **NIB** is the intent store for all Orion applications. It is implemented as a centralized, in-memory datastore with replicas that reconstruct the state from ground-truth on failure. The NIB is coupled with a publish-subscribe mechanism to share state among Orion applications. The same infrastructure is used externally to collect all changes in the NIB to facilitate debugging. The NIB must meet the following requirements:

- *Low External Dependency*. As Orion programs the network supporting all higher-level compute and storage services, it cannot itself depend on higher-level services.
- *Sequential Consistency of Event Ordering*. To simplify coordination among apps, all apps must see events in the same order (*arrow of time* [20]).

Of note, *durability* [14] was not a requirement for the NIB because its state could be reconstructed from network switches and other sources in the event of a catastrophic failure.

NIB Entities. The NIB consists of a set of NIB *entity* tables where each entity describes some information of interest to other applications or observers both local or external to the domain. Some of the entity types include:

- *Configured network topology*. These capture the configured identities and graph relationship between various network topological elements. Examples include `Port`, `Link`, `Interface`, and `Node` tables.
- *Network run-time state*. This could be topological state, forwarding state (e.g. `ProgrammedFlow` table), protocol state (e.g. `LLDPPeerPort` table), statistics (e.g. `PortStatistics` table).
- *Orion App Configuration*. Each app's configuration is captured as one or more NIB tables, e.g. `LLDPConfig`.

Protocol Buffer Schema. We represent the schema for each NIB entity as a protocol buffer message [8]. Each row in that NIB entity table is an instantiation of this schema. The first field of each entity schema is required to be a *NIBHeader* message which serves as the key for that entity. The NIB does not enforce referential integrity for foreign keys; however, inconsistencies fail an internal health-check.

An example entity represented in the NIB is a `Link` entity. A link is modelled as foreign key references to `Port` and `Node` entities respectively. This expresses the connection between two ports of two switches. Additionally, a status (up, down, or unknown), is modelled as part of the Link entity. The full protocol buffer is shown in the appendix.

Protocol buffers allow us to reuse well-understood patterns for schema migrations. For example, adding a new field to a table has built-in support for backward and forward compatibility during an upgrade despite some applications still running with the previous schema.

NIB API. The NIB provides a simple RPC API (*Read, Write, Subscribe*) to operate on NIB tables. The *Write* operation is atomic and supports batching. The *Subscribe* operation supports basic filtering to express entities of interest. The NIB notification model provides sequential consistency of event ordering. It also supports coalescing multiple updates into a single notification for scale and efficiency reasons.

The **Config Manager** provides an external management API to configure all components in an Orion domain. The domain configuration is the set of app configurations running in that domain. For uniformity and ease of sharing, an app config consists of one or more NIB tables. To ensure a new configuration is valid, it is first validated by the running instance. The semantics of pushing config need to be atomic, i.e. if one or more parts of the overall config fail validation, the overall config push must fail without any side effects. Since Orion apps that validate various parts of the config run decoupled, we employ a two-phase commit protocol to update

the NIB: The config is first staged in *shadow* NIB tables, and each app verifies its config. Upon success, we commit the shadow tables to live tables atomically.

The **Topology Manager** sets and reports the runtime state of network dataplane topology (node, port, link, interface, etc.). It learns the intended topology from its config in the NIB. By subscribing to events from the switches, it writes the current topology to tables in the NIB. The Topology Manager also periodically queries port statistics from the switches.

The **Flow Manager** performs flow state reconciliation, ensuring forwarding state in switches matches intended state computed by Orion apps and reflected in the NIB. Reconciliation occurs when intent changes or every 30 seconds by comparing switch state. The latter primarily provides Orion with switch statistics and corrects out-of-sync state in the rare case that reconciliation on intent change failed.

The **OFE (Openflow FrontEnd)** multiplexes connections to each switch in an Orion domain. The OpenFlow protocol provides programmatic APIs for (i) capabilities advertisement, (ii) forwarding operations, (iii) packet IO, (iv) telemetry/statistics, and (v) dataplane event notifications (e.g. link down) [23]. These are exposed to the Topology and Flow Manager components via OFE's northbound RPC interface.

Packet-I/O. Orion supports apps that send or receive control messages to/from the data plane through OpenFlow's Packet-I/O API: a ***Packet-Out*** message sends a packet through a given port on the switch, while a ***Packet-In*** notification delivers a data plane packet punted by the switch to the control plane. The notification includes metadata such as the packet's ingress port. Orion apps can program punt flows and specify filters to receive packets of interest.

Orion Core apps are network-type agnostic by design. No "policy" is baked into them; it belongs to higher-level SDN applications instead. Core apps program, and faithfully reflect, the state of the data plane in the NIB in a generic manner.

4.2 Routing Engine

Routing Engine (RE) is Orion's intra-domain routing controller app, providing common routing mechanisms, such as L3 multi-path forwarding, load balancing, encapsulation, etc.

RE provides abstracted topology and reachability information to client routing applications (e.g. an *inter-domain routing* app or a *BGP speaker* app). It models a configured collection of switches within an Orion domain as an abstract routing node called a *supernode* [13] or *middleblock* [28]. Client routing applications provide *route advertisements* at supernode granularity, specifying nexthops for each route in terms of aggregate or singleton external ports.

RE disaggregates the route advertisements from its clients into individual node-level reachability over respective external ports and computes SPF (Shortest Path First) paths for each prefix. RE avoids paths that traverse drained, down or potentially miscabled links.[3] It also reacts to local failure by computing the next available shortest path when the current set of nexthops for a prefix becomes unreachable. For improved capacity, RE performs load balancing within a domain by spreading traffic across multiple viable paths, and through non-shortest-path forwarding, as requested by client apps. RE also manages the associated switch hardware resources (e.g. Layer-3 tables) among its client routing apps.

A key highlight of Orion Routing Engine is the ability to do loss-free sequencing from the currently programmed pathing solution to a new pathing solution. This may happen in reaction to changes in network states (e.g. a link being avoided). In a legacy network, the *eventually consistent* nature of updates from distributed routing protocols (e.g. BGP) can result in transient loops and blackholes in the data plane. In contrast, RE exploits its global view to sequence flow programming: before programming a flow that steers traffic to a set of switches, RE ensures the corresponding prefixes have been programmed on those nexthop switches. Analogous checks are done before removing a flow.

Figure 3 walks through an end-to-end route programming example. As evident from the sequence of operations, the NIB semantics lend themselves to an asynchronous ***intent-based programming model*** (as opposed to a strict *request-response* interaction). A common design pattern is to use a pair of NIB tables, where one expresses the *intent* from the producer, while the other captures the *result* from the consumer. Both *intent* and *result* tables are versioned. An app can change the intent many times without waiting for the result, and the result table is updated asynchronously.

4.3 Orion Application Framework

The Orion Application Framework is the foundation for every Orion application.The framework ensures developers use the same patterns to write applications so knowledge of one SDN application's control-flow translates to all applications. Furthermore, the framework provides basic functionality (e.g. leader-election, NIB-connectivity, health-monitoring) required by all applications in all deployments.

High Availability. Availability is a fundamental feature for networks and thereby SDN controllers. Orion apps run as separate binaries distributed across network control server machines. This ensures applications are isolated from bugs (e.g., memory corruption that leads to a crash) in other applications.

Beyond isolation, replicating each application on three different physical machines ensures fault tolerance for both planned (e.g. maintenance) as well as unplanned (e.g. power failures) outages. The application framework facilitates replication by providing an abstraction on top of leader election as well as life-cycle callbacks into the application.

An application goes through a life-cycle of being activated, receiving intent/state updates from the NIB, and then being deactivated. Identifying/arbitrating leadership and its transition (referred to as *failover*) among replicas is abstracted and

[3] A link is considered miscabled when a port ID learned by a neighbor node via LLDP and reported to Orion does not match the configured port ID.

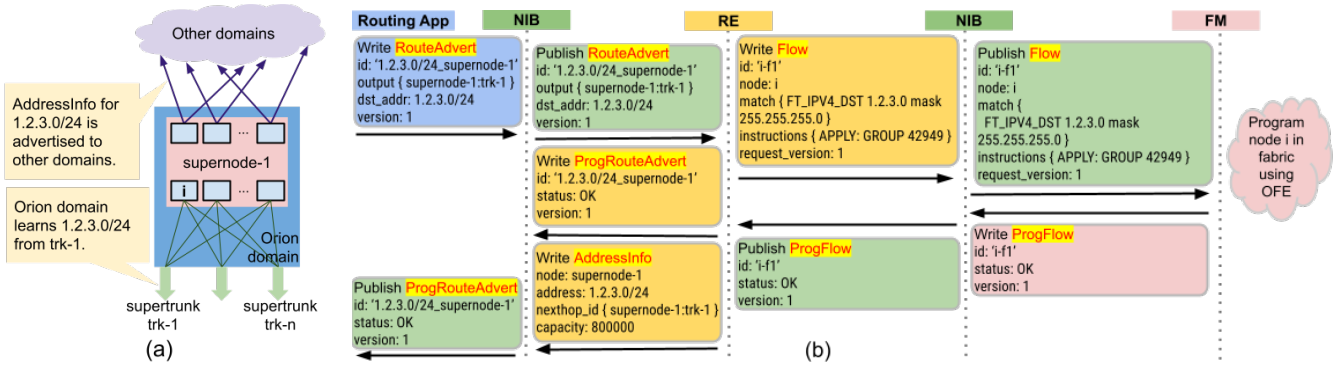

Figure 3: Intent-based route programming on abstracted domain topology: (a) Routing Engine learns external prefix 1.2.3.0/24 over trk-1, and programs nodes to establish reachability. (b) Example of end-to-end route programming. The Routing App provides a high-level RouteAdvert on supernode-1 via the NIB. Routing Engine translates the RouteAdvert to a low-level Flow update on node i and sends to Flow Manager. Acknowledgements follow the reverse direction to the Routing App. Similar route programming applies to all domain nodes.

thereby hidden from the application author, reducing surface area for bugs as well as complexity.

Capability Readiness Protocol. One of the challenges we faced previously was an orderly resumption of operation after controller failover. In particular, when a controller's NIB fails, the state of the new NIB needs to be made consistent with the runtime state of the network, as well as the functional state of all apps and remote controllers. In an extreme case, an Orion requirement is to be able to, without traffic loss, recover from a complete loss/restart of the control plane. To support this, the Orion architecture provides a *Capability Readiness Protocol*. With this protocol, applications have a uniform way of specifying which data they require to resume operation, and which data they provide for other applications.

A *capability* is an abstraction of NIB state, each can be provided and consumed by multiple apps. Capability-based coordination keeps the Orion apps from becoming "coupled", in which a specific implementation of one app relies on implementation details or deployment configuration of another app. Such dependencies are a problem for iteration and release velocity. For example, multiple apps can provide the capability of "producing flows to program", and the Flow Manager can be oblivious to which ones are present in the domain.

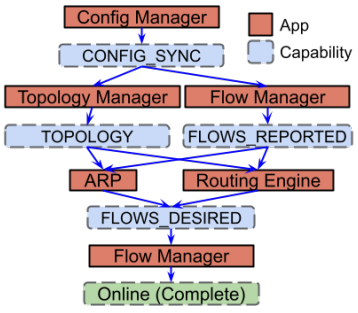

Figure 4: Capability Readiness graph for flow programming.

The Capability Readiness protocol requires, after a NIB failover, that all apps report readiness of their flows before Flow Manager begins reconciling NIB state to the physical switches. This prevents unintentional erasure of flow state from switches, which would lead to traffic loss. As Figure 4 shows, the required and provided data that each application specifies creates a directed acyclic graph of capabilities *depended upon* and *provided*, and thus the complete NIB state is reconciled consistently after any restart. Apps can have mutual dependency on different capabilities as long as they do not form a loop. A healthy Orion domain completes the full capability graph quickly on reconciliation, a condition we check in testing and alert on in production. Since this graph is static, such testing prevents introducing dependency loops.

In the event of a total loss of state, Config Manager retrieves the static topology from Chubby [5], an external, highly-available service for locking and small file storage. It then provides a `CONFIG_SYNC` capability to unblock Topology Manager and Flow Manager. The two connect to switches specified in the config and read switch states and programmed flows. Then, ARP and Routing Engine can be unblocked to generate intended flows that need to be programmed; they also provide their own `FLOWS_DESIRED` capability to Flow Manager, which proceeds to program the switches.

Apps that retrieve their state from a remote service must explicitly manage and support the case in which the service is unavailable or disconnected to prevent prolonged domain reconciliation delays. Cached data is typically used until the authoritative source of the inputs can be reached.

5 Orion-based Systems

Among the many design choices when implementing Orion to control a specific network, three prominent ones include the mapping of network elements to controllers, the method of controller to switch communication, and connectivity to external networks running standard routing protocols. We first review these common choices across two Google network architectures, Jupiter and B4, and then describe specific details for each architecture. Less relevant in a networking context,

details of the NIB implementation are in the appendix.

Control domains. The choice of elements to control in an Orion domain involves multiple tradeoffs. Larger domains yield optimal traffic distributions and loss-free route sequencing for more intent changes, at the price of increased blast radius from any failure. In Jupiter, we use a hierarchy of partitioned Orion domains; in B4, a flat partitioning of Orion domains communicating with non-Orion global services. Each came with challenges in production, which we review in §7.

Control channel. As discussed in §3.1, we faced tradeoffs when designing the Control Plane Network (CPN) connecting Orion controllers to the data plane. The cost and complexity of a second network led us to a hybrid design where only the Top-of-Rack (ToR) switches were controlled in-band.

- *Separation of control and data plane:* When we embarked on building B4 and Jupiter, we embraced the SDN philosophy in its purest form: software-defined control of the network based on a logically centralized view of the network state outside the forwarding devices. To this end, we did not run any routing protocols on the switches. For the control plane, we ran a separate physical network connected to the switches' management ports. We ran conventional on-box distributed routing protocols on the CPN. Compared to the data plane network, the CPN has smaller bandwidth requirements, though it required N+1 redundancy.
- *CPN scale and cost:* Typical Clos-based data center networks are non-oversubscribed in the aggregation layers [28] with oversubscription of ToR uplinks based on the bandwidth requirements of compute and storage in the rack. A Clos network built with identical switches in each of its N stages will have the same number of switches (say, K) in all but two stages. The topmost stage will have $K/2$ switches since all ports are connected to the previous stage. The ToR stage will have SK switches, where S is the average oversubscription of uplinks compared to downlinks. Thus, the number of ToR switches as a fraction of the total is $2S/(2S+2N-3)$. In a Clos network with $N=5$ stages and an average ToR oversubscription, S, ranging from 2-4, ToR switches account for 36% to 53% of the total. Thus, not requiring CPN connectivity to them substantially reduces CPN scale and cost.
- *CPN cable management:* Managing ToRs inband removes the burden of deploying individual CPN cables to each rack spot in the datacenter.
- *Software complexity of inband ToRs:* Since ToRs are the leaf switches in a Clos topology, their inband management does not require on-box routing protocols. We designed simple in-band management logic in the switch stack to set the return path to the controller via the ToR uplink from which the ToR's CPU last heard from the controller.
- *Availability and debuggability considerations:* Over the years, we have hardened both the CPN and the inband-controlled ToR to improve availability. "Fail static" has been a key design to reduce vulnerability to CPN failures. Furthermore, we introduced in-band backup control of devices connected to the CPN for additional robustness.

External connectivity. We use BGP at the border of datacenter networks to exchange routes with Google's wide-area networks: B2 (which also connects to the Internet) and B4. These routes include machine addresses and also unicast and anycast IPs for Google services. BGP attributes such as communities, metrics, and AS path propagate state throughout Google's networks. In addition to reachability, this can include drain state, IPv6-readiness, and bandwidth for WCMP.

The use of BGP is a necessity for eventual route propagation to the Internet, but a design choice internally. The choice was made to simplify inter-connection with traditional, non-SDN routers as well as previous SDN software [18]. BGP also brings operator familiarity when defining polices to specify path preferences during topological changes.

An Orion app, **Raven** [34], integrates BGP and IS-IS into Orion. Raven exchanges messages with peers via Orion's Packet-I/O. Raven combines these updates with local routes from the NIB into a standard BGP RIB (Route Information Base). Routes selected by traditional Best-Path Selection are then sent, depending on policy, to peer speakers as BGP messages, as well as the local NIB in the form of `RouteAdvert` updates. To reduce complexity, Raven's associated BGP "router" is the abstract supernode provided by RE (§4.2).

Unfortunately, BGP is somewhat mismatched with our design principles: it uses streaming rather than full intent updates, its local view precludes a threshold-based fail static policy and global rebalancing during partial failures, and it ties control-plane liveness to data-plane liveness. In our production experience, we have had both kinds of uncorrelated failures, which, as in non-SDN networks, become correlated and cause a significant outage only due to BGP. By contrast, Orion's fail static policies explicitly consider control-plane and data-plane failure as independent. Adding fail static behavior to these adjacencies is an area of ongoing development.

5.1 Jupiter

We initially developed Jupiter [28], Google's datacenter network, with our first generation SDN-based control system, Onix [18]. The Orion-based solution presented here is a second iteration based on lessons from the Onix deployment.

The Jupiter datacenter network consists of three kinds of building blocks, each internally composed of switches forming a Clos-network topology: (i) aggregation blocks [28] connected to a set of hosts, (ii) FBRs (Fabric Border Routers, also called Cluster Border Routers in [28]) connected to the WAN/Campus network, and (iii) spine blocks that interconnect aggregation blocks and FBRs.

We organize Orion domains for Jupiter hierarchically as shown in Figure 5. First, we map physical Orion domains to the Jupiter building blocks. Each physical domain programs switches within that domain. Aggregation block domains es-

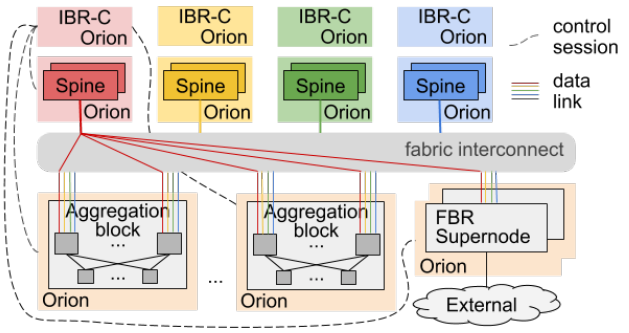

Figure 5: Jupiter topology overlaid with Orion domains partitioned by color. Colored links and spine domains are controlled by the respectively colored IBR-C. Uncolored aggregation block/FBR domains are controlled by all IBR-Cs. Only control sessions and data links of the red color are displayed.

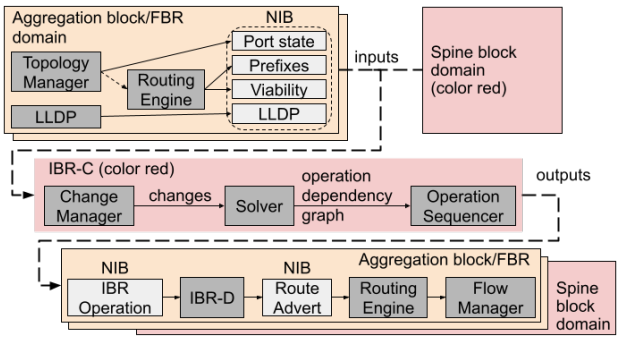

Figure 6: Jupiter fabric-level IBR-C control flow of one color.

tablish connectivity among hosts attached to it. FBR domains use Raven to maintain BGP sessions with fabric-external peers. Multiple spine blocks can map to a single Orion domain, but each domain must contain fewer than 25% of all spine blocks to limit the blast radius of domain failure.

Second-level Orion domains host a partitioned and centralized routing controller IBR-C (Inter-Block Routing Central). Operating over Routing Engine's abstract topology, IBR-C aggregates network states across physical domains, computes fabric-wide routes, and programs physical domains to establish fabric-wide reachability. While these virtual domains start from the same foundations, they do not contain some Orion core apps for controlling devices directly.

To avoid a single point of failure, we partitioned (or *sharded*) IBR-C into four planes called "colors," each controlling 25% of the spine blocks and hence a quarter of paths between each pair of spine blocks and aggregation blocks/FBRs. Therefore, the blast radius of a single controller does not exceed 25% of the fabric capacity. Sharding centralized controllers avoids failures where a single configuration or software upgrade affects the whole fabric. Additional protection was added to stage configuration changes and upgrades to avoid simultaneous updates across colors. While sharding provided higher resiliency to failures, the trade-off was an increased complexity in merging routing updates across colors in aggregation block and FBR domains, as well as a loss in routing optimality in case of asymmetric failures across colors. We have considered even deeper sharding by splitting aggregation blocks into separate domains, each controlling a portion of the switches. This option was rejected due to even higher complexity while marginally improving availability.

Figure 6 illustrates the fabric-level control flow of one IBR-C color. IBR-C subscribes to NIBs in all aggregation block/FBR domains and spine domains of the same color for state updates. After aggregation at Change Manager, the Solver computes inter-block routes and Operation Sequencer writes the next intended routing state into NIB tables of corresponding domains. IBR-D (Inter-Block Routing Domain), a domain-level component, merges routes from different IBR-C colors into `RouteAdvert` updates. Finally, Routing Engine and Orion Core program flows as shown in Figure 3.

Convergence. We care about two types of convergence in Jupiter: data plane convergence and control plane convergence. Data plane convergence ensures there are valid paths among all source/destination pairs (no blackholing) while control plane convergence restores (near) optimal paths and weights in the fabric. Workflows that require changes to the network use control plane convergence as a signal they can proceed safely. Convergence time is the duration between a triggering event and all work complete in data/control plane.

Jupiter's reaction to link/node failures is threefold. First, upon detection of link-layer disruption, switches adjacent to the failed entity perform local port pruning on the output group. However, this is not possible if no alternative port exists or peer failure is undetectable (e.g., switch memory corruption). Second, RE programs the domain to avoid this entity. This is similar to switch port pruning, but could happen on non-adjacent switches within the domain. For failures that do not affect inter-block routing, the chain of reaction ends here. Otherwise, in a third step, RE notifies IBR-C of the failure, as shown in Figure 6. When fabric-wide programming is complete, IBR-C signals the control plane has converged. This multi-tier reaction is advantageous for operations, as it minimizes data plane convergence time and thus traffic loss.

Since a single entity failure can lead to successive events in different domains (e.g., spine switch failure causing aggregation block links to fail), it could trigger multiple IBR-C and domain programming iterations to reach final convergence. Many independent events also happen simultaneously and get processed by Orion together, which can further delay convergence. Hence, we will evaluate Orion's performance in example convergence scenarios in §6.

Implementation Challenges. One challenge with the original Jupiter implementation [28] was optimally distributing traffic across multiple paths. Capacity across paths can differ due to link failures and topology asymmetry (e.g., different link count between a spine-aggregation block pair). In order to optimally allocate traffic, Jupiter/Orion employs WCMP to vary weights for each path and nexthop. Due to the precise

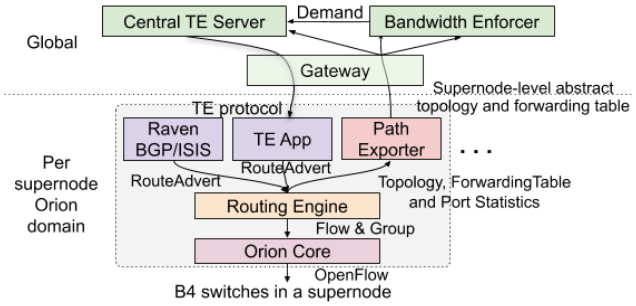

Figure 7: B4 Control Diagram

weight computation for each forwarding entry, weights need to be adjusted across the entire fabric to fully balance traffic. Another challenge was transient loops or blackholes during route changes. This is due to asynchronous flow programming in traditional routing protocols and in our previous SDN controller [18]. With Orion-based Jupiter, we implement end-to-end flow sequencing in both IBR-C and RE.

At the scale of Jupiter, network events arrive at IBR-C at a high frequency, which sometimes surpasses its processing speed. To avoid queue buildup, IBR-C prioritizes processing certain loss-inducing events (e.g., link down) over noncritical events (e.g., drain). Upon an influx of events, IBR-C only preempts its pipeline for loss-inducing events. It reorders/queues other events for batch processing upon the completion of higher priority processing. This is a trade-off to minimize traffic loss while avoiding starvation of lower priority events. §6 quantifies the benefits of this approach in more detail.

5.2 B4

Onix [18], the first-generation SDN controller for B4, ran control applications using cooperative multithreading. Onix had a tightly-coupled architecture, in which control apps share fate and a common threading pool. With Onix's architecture, it was increasingly challenging to meet B4's availability and scale requirements; both have grown by 100x over a five year period [13]. Orion solved B4's availability and scale problems via a distributed architecture in which B4's control logic is decoupled into micro-services with separate processes.

Figure 7 shows an overview of the B4 control architecture. Each Orion domain manages a B4 supernode, which is a 2-stage folded-Clos network where the lower stage switches are external facing (see details in [13]). In B4, Routing Engine sends ingress traffic to all viable switches in the upper stage using a link aggregation group (LAG), and uses two-layer WCMP to load-balance traffic toward the nexthop supernode.

The **TE App** is a traffic engineering agent for B4. It establishes a session with global TE server instances to synchronize the tunnel forwarding state. It learns TE tunneling ops from the primary TE server, and programs the ops via the `RouteAdvert` table. In addition, TE App also supports Fast ReRoute (FRR), which restores connectivity for broken tunnels by temporarily re-steering the traffic to the backup tunnel set or BGP/ISIS routes.

The **Path Exporter** subscribes to multiple NIB tables and exports the observed dataplane state to the global services. It reports the dataplane state at the supernode level, including the abstract topology (e.g., supernode-supernode link capacities), the abstract forwarding table (TE tunnels and BGP routes), and the abstract port statistics.

The **Central TE Server** [13,15] is a global traffic engineering service which optimizes B4 paths using the TE protocol offered by the TE App in each domain. The **Bandwidth Enforcer** [19] is Google's global bandwidth allocation service which provides bandwidth isolation between competing services via host rate limiting. For scalability, both the Central TE Server and Bandwidth Enforcer use the abstract network state provided by the Path Exporter.

6 Evaluation

We present microbenchmarks of the Orion NIB, followed by Jupiter evaluation using production monitoring traces collected since January 2018. Orion also improved B4 performance, as published previously [13].

NIB performance. To characterize NIB performance, we show results of a microbenchmark measuring the NIB's read/write throughput while varying the number of updates per batch. A batch is a set of write operations composed by an app that updates rows of different NIB tables atomically. In Figure 8, we observe throughput increase as the batch size becomes larger. At 50K updates per batch, the NIB achieves 1.16 million updates/sec in read throughput and 809K updates/sec in write throughput.

Write throughput at 500 updates per batch sees a decline. This reveals an implementation choice where the NIB switches from single-threaded write to multi-threaded write if the batch size is greater than 500. When the batch size is not large enough, up-front partitioning to enable parallelism is more expensive than the performance improvement. This fixed threshold achieves peak performance on sampled production test data and performs well in production overall. It could be removed in favor of a more dynamic adaption strategy to smooth the throughput curve.

Data and control plane convergence. One key Jupiter performance characteristic is convergence time (§5.1). We measure convergence times in several fabrics, ranging from 1/16-size Jupiter to full-size Jupiter (full-size means 64 aggregation blocks [28]). Figure 9 captures data and control plane convergence times of three types of common daily events in our fleet. The measurements are observed by Orion; switch-local port pruning is an independent decision that completes within a few milliseconds without Orion involvement.

In node/link down scenarios, the data plane converges within a fraction of a second. Both data and control plane convergence times become longer as the fabric scales up by 16x. This is mainly because a larger number of aggregation

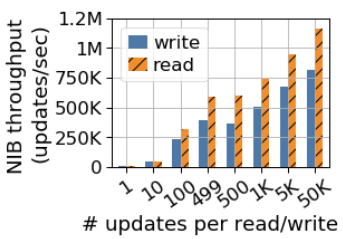

Figure 8: NIB throughput.

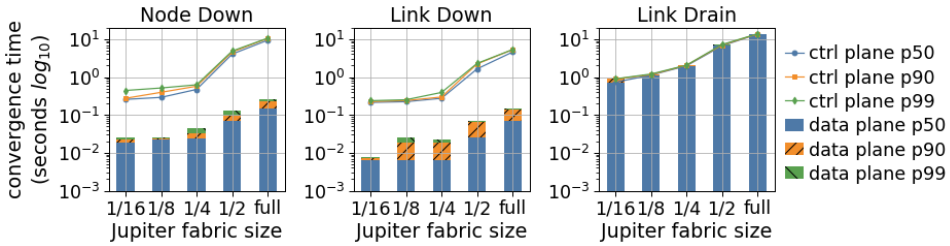

Figure 9: Jupiter data/control plane convergence time in response to various network events.

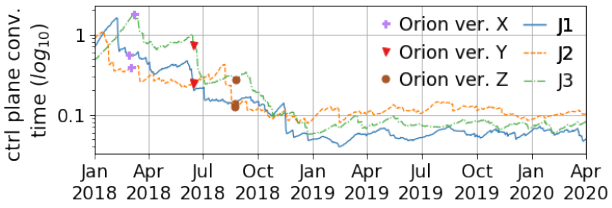

Figure 10: Time series of full fabric control plane convergence on three large Jupiter fabrics. Y-axis is normalized to the baseline convergence time in January 2018. Orion releases with major performance improvements are highlighted by markers.

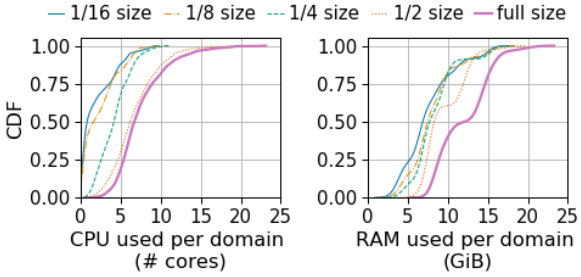

Figure 11: Orion CPU/RAM usage in Jupiter domains.

blocks and spine blocks require more affected paths to be re-computed and re-programmed. Data plane convergence is 35-43x faster than control plane convergence, which effectively keeps traffic loss at a minimum.

Jupiter's reaction to link drains is slightly different from failure handling. Drains are lossless, and do not include an initial sub-optimal data plane reaction to divert traffic. Instead, Orion only shifts traffic after computing a new optimal routing state. Therefore, data and control plane convergence are considered equal. Overall, control plane convergence time for link drain is on par with node/link down scenarios.

We have continuously evolved Orion to improve Jupiter scalability and workflow velocity. Key to this were enhancements in IBR-C such as prioritized handling of select updates, batch processing/reordering, and a conditionally preemptive control pipeline. Figure 10 shows the trend of three large fabrics from January 2018 to April 2020; the control plane convergence time in January 2018 was before these improvements. Deployed over three major releases, each contributing an average 2-4x reduction, the new processing pipeline (§5.1) delivered a 10-40x reduction in convergence time.

Controller footprint: CPU and memory. Orion controller jobs run on dedicated network control servers connected to the CPN. This pool is comprised of regular server-class platforms. We measure CPU and memory usage of each controller job (including all three replicas) and group them by domain. Figure 11 shows that even in a full-size Jupiter, Orion domains use no more than 23 CPU cores and 24 GiB of memory.

7 Production Experience

We briefly review some challenging experiences with Orion when adhering to our production principles of limited blast-radius and fail-static safety, and some more positive experiences from following our software design principles.

7.1 Reliability and robustness

Failure to enforce blast radius containment. As described in §5.1, the inter-block routing domain is global but sharded into four colors to limit the blast radius to 25%. A buggy IBR-C configuration upgrade caused a domain to revoke all forwarding rules from the switches in that domain resulting in 25% capacity loss. Since high-priority traffic demand was below 25% of the fabric's total capacity, only after all four domains' configurations were pushed did the workflow flag (complete) high-priority packet loss in the fabric. To prevent such a "slow wreck" and enforce blast radius containment, subsequent progressive updates proceeded only after confirming the previous domain's rollout was successful, and not simply the absence of high-priority packet loss.

Failure to align blast radius domains. A significant Orion outage occurred in 2019 due to misalignment of job-control and network-control failure domains. Like many services at Google, these Orion jobs were running in Borg cells [31]. Although the Orion jobs were themselves topologically-scoped to reduce blast radius (§3.1), their assignment to Borg cells was not. As described in the incident report [9], when a facility maintenance event triggered a series of misconfigured behaviors that disabled Orion in those Borg cells, the resulting failure was significantly larger than Google's networks had been previously designed to withstand. This outage highlighted the need for all management activities (job control, configuration update, OS maintenance, etc.) to be scoped and rate-limited in a coordinated manner to fully realize the principle of blast-radius reduction. In addition, it highlighted a gap in our fail-static implementation with regards to BGP.

Failure to differentiate between missing and empty state. In 2018, a significant B4 outage illustrated a challenge when integrating Orion and non-Orion control layers. A gradual increase in routes crossed an outdated validation threshold on the maximum number of routes a neighborhood should receive from attached clusters. Consequently, the TE app suppressed the data to the B4-Gateway, which correctly failed static using cached data. However, subsequent maintenance restarted B4-Gateway instances while in this state, clearing the cached data. Lacking any differentiation between *empty* data and *missing* data, subsequent actions by the TE Server and Bandwidth Enforcer resulted in severe congestion for low-priority traffic. Within Orion, the capability readiness protocol prevents reading missing or invalid derived state.

Orion's post-outage improvements and continuous feature evolution such as loss-free flow sequencing and in-band CPN backup, brought substantial improvements in data plane availability to Jupiter (50x less unavailable time) and B4 (100x less unavailable time [13]).

7.2 Deployability and software design

With Orion, we moved to a platform deeply integrated with Google's internal infrastructure. This enabled us to leverage existing integration testing, debugging, and release procedures to increase velocity. We also moved from a periodic release cycle to a continuous release: we start software validation of the next version as soon as the current version is ready. This reduces the amount of "queued up" bugs, which improves overall velocity.

Release cadence. SDN shifts the burden of validation from "distributed protocols" to "distributed algorithms", which is smaller. Software rollouts are also faster: the number of Network Control Servers is orders of magnitude smaller than the number of network devices. The embedded stack on the devices is also simpler and more stable over time.

Orion's micro-service-based architecture leads to clear component API boundaries and effective per-component testing. Onix, on the other hand, was more monolithic, and our process required more full-system, end-to-end testing to find all newly introduced bugs which was less efficient.

In steady state, after initial development and productionization, it took about five months to validate a new major Onix release. The process was manual, leveraging a quality assurance team and iterative cherry-picking of fixes for discovered issues. With Orion, we shrank validation latency to an average of 14.7 days after the initial stabilization phase, with a target of eventually qualifying a release every week.

Release granularity. As a distributed, micro-service-based architecture, each Orion application could release at its own cadence. In our move from the monolithic Onix to Orion, we have not yet leveraged this flexibility gain. Since Orion is widely deployed and has high availability demands, we strive to test all versions that run at the same time in production, for instance as some applications are upgraded but other upgrades are still pending. An increase in release granularity would increase both the skew duration and total number of combinations that need to be tested. Therefore, we releas all Orion applications that make up a particular product (e.g., B4 or Jupiter) together.

Improved debugging. Serializing all intent and state changes through the NIB facilitates debugging: Engineers investigating an issue can rely on the fact that the order of changes observed by all applications in the distributed system is the same and therefore establish causality more easily.

Storing the stream of NIB updates for every Orion deployment also allowed us to build replay tooling that automatically reproduces bugs in lab environments. This was first used in the aftermath of an outage: only the precise ordering of programming operations that occurred in production, replayed to a lab switch, reliably reproduced a memory corruption bug in Google's switch firmware. This enabled delivering as well as, more importantly, verifying the fix for this issue.

8 Conclusion and future work

This paper presents Orion, the SDN control plane for Google's Jupiter datacenter and B4 Wide Area Networks. Orion decouples control from individual hardware elements, enabling a transition from pair-wise coordination through slowly-evolving protocols to a high-performance distributed system with a logically centralized view of global fabric state. We highlight Orion's benefits in availability, feature velocity, and scale while addressing challenges with SDN including aligning failure domains, inter-operating with existing networks, and decoupled failures in the control versus data planes.

While Orion has been battle-tested in production for over 4 years, we still have open questions to consider as future work. These include (i) evaluating the split between on-box and off-box control, (ii) standardizing the Control to Data Plane Interface with P4Runtime API [10], (iii) exploring making the NIB durable, (iv) investigating fail-static features in BGP, and (v) experimenting with finer-grained application release.

Acknowledgments For their significant contributions to the evolution of Orion, we thank Deepak Arulkannan, Arda Balkanay, Matt Beaumont-Gay, Mike Bennett, Bryant Chang, Xinming Chen, Roshan Chepuri, Dharti Dhami, Charles Eckman, Oliver Fisher, Lisa Fowler, Barry Friedman, Goldie Gadde, Luca Giraudo, Anatoliy Glagolev, Jia Guo, Jahangir Hasan, Jay Kaimal, Stephen Kratzer, Nanfang Li, Zhuotao Liu, Aamer Mahmood, John McCullough, Bhuva Muthukumar, Susan Pence, Tony Qian, Bharath Raghavan, Mike Rubin, Jeffrey Seibert, Ruth Sivilotti, Mukarram Tariq, Malveeka Tewari, Lisa Vitolo, Jim Wanderer, Curt Wohlgemuth, Zhehua Wu, Yavuz Yetim, Sunghwan Yoo, Jiaying Zhang, and Tian Yu Zhang. We also thank our shepherd, Katerina Argyraki, Jeff Mogul, David Wetherall, and the anonymous reviewers for their valuable feedback.

References

[1] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-Optimal Network Design for Coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 16–29, 2018.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.

[3] Arista. Arista EOS Whitepaper. https://www.arista.com/assets/data/pdf/EOSWhitepaper.pdf, 2013.

[4] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'05, page 15–28, USA, 2005. USENIX Association.

[7] Open Networking Foundation. SDN Architecture Overview. https://www.opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf, 2013.

[8] Google. Protocol buffers: Google's data interchange format. https://code.google.com/p/protobuf/, 2008.

[9] Google Cloud Networking Incident 19009, 2019. https://status.cloud.google.com/incident/cloud-networking/19009.

[10] The P4.org API Working Group. P4 Runtime Specification. https://p4.org/p4runtime/spec/v1.2.0/P4Runtime-Spec.html, 2020.

[11] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[12] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 15–26, 2013.

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

[14] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[15] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally Deployed Software Defined WAN. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013.

[16] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, April 2013. USENIX Association.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, April 2013. USENIX Association.

[18] Teemu Koponen, Martín Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[19] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM'15*, 2015.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[21] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[22] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communication Review (CCR)*, 38:69–74, 2008.

[24] J. Medved, R. Varga, A. Tkacik, and K. Gray. OpenDaylight: Towards a Model-Driven SDN Controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6, 2014.

[25] Microsoft. SONiC System Architecture. https://github.com/Azure/SONiC/wiki/Architecture, 2016.

[26] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[27] Timothy Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[28] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hanying Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM '15*, 2015.

[29] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *Annual International Cryptology Conference*, pages 570–596. Springer, 2017.

[30] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, page 3, USA, 2010. USENIX Association.

[31] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[32] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A Database-Defined Network. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Hong Yan, David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: a 4D network control plane. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 27–27, 2007.

[34] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 432–445, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012.

[36] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page Article No. 5, 2014.

9 Appendix

An example table schema definition in the NIB.

```
message Link {
  enum Status {
    STATUS_UNKNOWN = 0;
    DOWN = 1;
    UP = 2;
  }
  optional NIBHeader nib_header = 1;
  optional string name = 2;
  optional Status status = 3;
  // references Node.nib_header.id
  optional string src_node_id = 4;
  // references Port.nib_header.id
  optional string src_port_id = 5;
  // references Node.nib_header.id
  optional string dst_node_id = 6;
  // references Port.nib_header.id
  optional string dst_port_id = 7;
}
```

9.1 Orion Core Implementation

To productionize the Orion systems described in this paper, we address some common engineering challenges in the Orion Core. In particular, Orion's architectural choices require great care with both the memory footprint and performance of all data flowing through the NIB. Here, we illustrate multiple implementation choices we made to scale this architecture.

State replication and synchronization. Orion enables a large group of developers in Google to author new SDN applications. To simplify interacting with the intent/state stored in the NIB, we synchronize all relevant data into an app-local cache that trails the NIB's authoritative state. The state visible to each application is always a prefix of the sequential atomic writes applied to the NIB. Each application's ability to see a prefix and therefore (potentially) not the most recent NIB state is acceptable given that the NIB itself is only a trailing reflection of the global system state. Applications do not subscribe to the NIB data they wrote as they were previously responsible for the write.

The app-local cache allows developers to access data as they would access an in-memory hash table. Additionally, since data is local, developers write fast applications by default as they do not have to reason about network round-trips to load data from the NIB. This replication approach requires transmitting cached data efficiently to reduce reaction time to data-plane events. While sequential communication through the NIB has many upsides, e.g., reduced complexity and traces for debugging, it should not dominate reaction time to data-plane events.

Additionally, we require a compact memory representation of all state to support a large number of micro-services, with overlapping subsets of the NIB state in their local caches. Protocol buffers are not well-suited for this requirement because they hold copies of the same nested and repeated fields in more than one entity (e.g., the same output port ID string in multiple forwarding rules). Orion works around this space inefficiency by exploiting the read-only nature of cached data. As all entity mutations are sent directly to the NIB for full serializability, it is possible to de-duplicate select sub-messages and data types in the app-local cache to conserve memory. Given our schema, this reduces memory overhead by more than 5x while retaining the same read-only API offered by regular protocol buffers.

Hash table / key size. Both the dictionary implementation as well as the key length used for the data in the NIB influence memory utilization. In early testing, about 50% of memory was consumed by key-strings and dictionary data structures.

Many data entries in the NIB use descriptive identifiers. For instance, a physical port on a switch, represented in the `Port` table of the NIB, combines the FQDN (Fully Qualified Domain Name) of its parent switch and its own port name identifier as the entity key in the NIB. As the payload per port is only a handful of simple data types, the key size may be larger than the payload. To reduce the memory impact of many entities with relatively small payloads, the NIB initially stored all data in a *trie* modelled after [21]. This was advantageous both because of prefix compression and because it enables inexpensive creation of consistent snapshots.

While we retain the M-Trie data structure for its copy-on-write capability, we have changed to storing a SHA-1 hash of the identifier only. This reduces the NIB memory footprint at the price of a theoretical, but not practical [29], collision risk. If a collision occurred, it would require human intervention.

Large updates. Large updates must be reflected in all SDN apps that consume changed data as their input. In some situations, for instance extensive rerouting, or, when an SDN app restarts and needs to be brought in sync with the NIB, the size of changed intent/state can be multiple gigabytes.

To reduce wall-clock time for such synchronization and peak memory spikes, Orion handles large atomic writes by splitting them into many small ones with special annotations delineating that the small writes are part of one atomic batch. This allows pipeline processing in the NIB, data transfer and cache application on the receiver. As long as the partial small updates are guaranteed to be exposed only after the complete set is available, this optimization is transparent.

In case of synchronizing client application and NIB state after a disconnect, the update size is kept to a minimum to reduce transmission delay. Each application supplies the NIB with a vector of ID-hashes of entities and entity hashes. Matching ID-hashes and entity hashes between the NIB and the application cache tells the NIB to skip updating such

entities. All other entities with mismatched or missing hashes will be replaced/deleted/inserted accordingly.

Software upgrade strategy. Since the controller is distributed across multiple apps running in separate binaries, software upgrades lead to version skew across applications. While this version skew is commonly short-lived, it can persist in rare cases as manual intervention is required to resolve the skew when an upgrade is stuck in a partially succeeded state. Given this, a key functionality that Orion provides to developers is enabling features atomically across applications when all are ready. Likewise, Orion can deactivate features atomically in case a participating app no longer supports it.

As an example, take two applications that interact through a NIB table *A*. Consider that the interaction pattern is changed to go through table *B* in a later version. Both applications have to ensure that the appropriate table is used depending on the readiness of the peer application. To reduce bug surface, Orion allows describing which applications need to support a certain feature for it to be enabled and abstracts this negotiation from the application developers. Developers simply protect the new interaction through table *B* inside a condition check, the condition is automatically marked true by Orion based on the feature readiness of both sides. If one of the two applications interacts through the old table, the feature stays disabled. Once both applications support the feature, Orion will enable it atomically.

9.2 Jupiter Fabric Drain

Orion provides support for network management operations. In order to take a network element out of service for repair or upgrade, the control systems needs to first drain it, i.e. divert traffic from it. Orion provides explicit handling and tracking of drain state. Drain Conductor (DC) is an Orion application in a separate non-sharded virtual control domain, providing an external API for network management systems to drain, undrain and check drain status for each element. Once DC receives a drain request for a networking element, it persists the new drain intent in Chubby [5] and dispatches the drain intent to the NIB in a physical domain. Drain Agent (DA), an application running in each physical domain, subscribes to drain intent changes published by DC, and dispatches the drain intent across routing applications. A routing app processes the drain intent by de-preferencing a networking element and updates the drain status. Finally, DC subscribes to drain statuses across physical domains and provides drain acknowledgements via its API.