Full-Stack SDN

Debnil Sur
VMware

Ben Pfaff
VMware Research

Leonid Ryzhyk
VMware Research

Mihai Budiu
VMware Research

ABSTRACT

The conventional approach for building software-defined network systems requires separately developing the management, control, and data planes. Manually written code connects the management plane's configuration to the control plane, and the control plane generates the data planes' configurations as small program fragments that scatter across the codebase. Scalability and correctness become increasingly challenging as such a system develops and grows.

In contrast, in our approach, called Nerpa, all three planes are programmed in a unified way. In Nerpa a transactional database stores management plane state. The control plane is implemented in a specialized query language which automatically executes in an incremental fashion, improving scalability. Finally, the data plane is programmed in P4. To aid correctness, all three parts are type-checked together, and tools generate code for data movement between planes.

We have published a prototype implementation using an open-source license. We believe that full-stack SDN can build more robust and maintainable networked systems.

CCS CONCEPTS

• **Networks** → **Programmable networks**;

KEYWORDS

Software-defined networking, network programming, enterprise networks

ACM Reference Format:
Debnil Sur, Ben Pfaff, Leonid Ryzhyk, and Mihai Budiu. 2022. Full-Stack SDN. In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3563766.3564101

HotNets '22, November 14–15, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9899-2/22/11…$15.00
https://doi.org/10.1145/3563766.3564101

1 INTRODUCTION

Recent advances allow significant programmatic control over network operations: software-defined networking (SDN) has broadly enabled the management of network devices [12, 18, 31, 32]. High-speed, programmable data planes let developers define complete and arbitrary processing of individual packet metadata [3, 9]. But despite these advances, it remains difficult and error-prone to program the *entire* network.

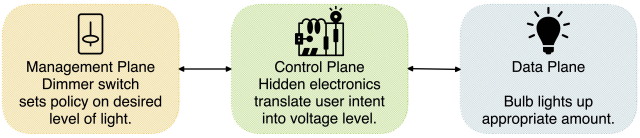

Figure 1: Management, control, and data planes for a dimmable light bulb.

The network software architecture is typically divided into three parts: the management, control, and data planes, illustrated by analogy in Fig. 1 for a dimmable light bulb. The management plane handles network policies, such as maintenance and monitoring [17], and provides APIs for administration. The control plane decides how packets are forwarded, transformed, or dropped. The data plane carries user traffic.

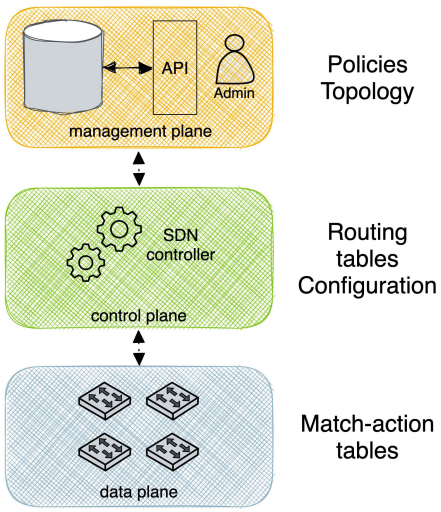

Figure 2: Architecture of today's software-defined networks.

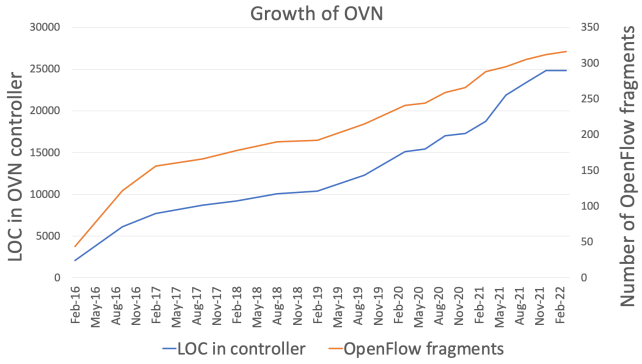

Figure 3: The growth of OVN's controller codebase and the number of OpenFlow fragments over time.

As shown in Fig. 2, the management plane is often implemented as an API backed by a reliable database [17], and the control plane as an SDN controller written in an imperative language such as Java or C++ [31, 38, 54]. The data plane is built using flow-programmable switch software or hardware [8, 23, 46]. Different teams implement each plane separately with different technologies. This process creates **correctness** and **scalability** challenges.

Within this paradigm, adding new network features that span all three planes requires a significant cross-team co-design, development, and testing effort. Complexity stems from orchestration required across planes and interactions between features within a plane. For example, in OpenFlow systems, the SDN controller acts as a specialized compiler that converts high-level policies into data plane program fragments: OpenFlow flows. The control plane installs fragments in network devices (e.g., switches). At any given time, a switch executes an OpenFlow program constructed from all currently installed fragments. Additional network features require new flow rule fragments for tables and associated priorities. These become scattered over the controller's quickly growing code base. The controller must handle various edge cases when translating policies into these fragments, and ensure that any possible combination of runtime policies generates a legal OpenFlow program for the targets.

The Open Virtual Network (OVN), a commercially deployed system for virtual network management [45], illustrates this trend. OVN manages a complex distributed system including an OVSDB database system [16] and multiple Open vSwitch data planes [48]. It provides L2/L3 virtual networking in support of many network features: logical switches and routers, security groups, L2/L3/L4 ACLs, tunnel overlays, and logical-physical gateways. Fig. 3 shows that over time, the controller's code base and the number of OpenFlow program fragments scattered throughout it have grown at a similar rate. This sprawl hurts maintenance and network correctness.

Moreover, today's control plane implementations can scale poorly. Traditional imperative programming languages used in controllers do not support **incrementality**. We contend this is essential for performance at scale. Network state changes dynamically due to many classes of events: policy updates, changes in load, maintenance, link failures, etc. In response to a change, the controller should not recompute and redistribute the entire network state. Instead, it should perform an *incremental* amount of work – proportional to (ideally linear in) the size of *modified* state, not of the entire network state [31]. Else, it will struggle with data center-sized deployments [31].

Unfortunately, imperative languages like Java or C++ offer zero support for writing incremental controllers. Writing incremental programs by hand requires a verbose and confusing coding style (reacting to concurrent events), or ad-hoc approaches for incremental computation [40, 61]. Consider labeling reachable nodes in a graph, a standard problem for computing forwarding tables. A full computation can be done in tens of lines of Java. But an *incremental* Java implementation, supporting dynamic insertions and deletions of network links and only recomputing changed labels, is much harder. Such an implementation in our organization's networking virtualization platform required several thousand lines of code. Despite developer and QA efforts, it required multiple releases to debug. A controller consists of many algorithms that benefit from incremental implementation.

A general-purpose programming language supporting incremental computation remains elusive. However, in the context of relational databases, the related concept of incremental view maintenance (IVM) is well understood [19]. Materialized views are defined by queries as a function of other database tables and views. On changes to the tables and source views, IVM computes and applies only incremental changes to the dependent views, rather than recomputing their full contents. Novel programming languages that compute over relations and collections offer automatic incremental view maintenance. They can be used to incrementalize queries for a rich class of programs, including recursive queries [11, 43, 53]. For example, in Differential Datalog, or DDlog for short, the programmer writes a specification for a non-incremental program using a rich dialect of Datalog. The DDlog compiler then generates an efficient incremental implementation. This only processes *input changes* or *events*, and it produces only *output changes* instead of entire new versions of the outputs.

Consider the following DDlog implementation of the network labeling problem described above. `GivenLabel` and `Edge` are input relations, and `Label` is the output view:

```
Label(n1, label) :- GivenLabel(n1, label).
Label(n2, label) :- Label(n1, label),
                    Edge(n1, n2).
```

This program maintains `Label` for any insertions or deletions in `Edges` or modifications of `GivenLabel`. DDlog automatically generates an incremental version. It provides scalability but is far shorter than the equivalent incremental

implementation in Java or C++. A slightly refined version is used in a production network controller with customers.

We propose a *unified environment* for full-stack SDN programming. Our vision, called Nerpa, combines relational and procedural abstractions to realize SDN's high-level approach and programmable data planes' fine-grained control. This includes two important insights:

(1) An automatically incremental control plane improves scalability. The network developer should not write and optimize an incremental controller by hand. Instead, control plane programs should be written in a modern programming language, designed for computing over collections and compiled to be incremental. Database queries then represent network features: forwarding rules are views defined from high-level policies and network state. The compiler turns these queries into incremental programs. These compute changes to forwarding rules from changes in policies and state. The written code is simple, declarative, and no longer responsible for handling multiple concurrent state changes.

(2) Co-designing the management, control and data planes helps overall correctness. Many data structures for the control plane program can be generated from corresponding structures in the data and management planes. Tooling can then automate and simplify data conversion when moving data between the planes. This saves developer time spent writing glue code and interfaces between different software components.

We have implemented a prototype of this programming framework. It uses an OVSDB management plane [16], P4 data plane [9], and DDlog control plane [53]. The prototype is available as an open-source project with an MIT license [2].

The rest of the paper is organized as follows: we identify specific challenges in programming the entire network in §2; we present the Nerpa vision in §3 and our prototype implementation in §4. We discuss related work in §5 and conclude in §6 with the potential implications of full-stack SDN.

2 BACKGROUND AND MOTIVATION

2.1 Control plane scalability

Many types of scalability challenges emerge in deployed networks. We focus on those that stem from changes in network state and could benefit from an incremental control plane.

Small, frequent configuration changes happen in network deployments. Their rapid occurrence at high scale can hamper performance. Recomputing the state of an entire network on each change requires significant CPU resources across compute nodes and creates high control plane latency [61]. Consider Robotron, Meta's top-down system for managing a massive production network [56]. Configurations are generated from FBNet, an underlying object store that models network components. Each day on average, more than 50 lines change across models for many reasons: new network components, changed attributes, logic changes, and more. In particular, backbone devices average a dozen changes per week, with over 150 lines updated per change [56]. These require continuous re-configurations and are updated incrementally. Simultaneously, the global disparity between the resource consumption of big and small jobs has only grown: the largest 1% of jobs represent over 99% of both compute and memory utilization [57]. A poorly-timed configuration issue could delay jobs or cause a critical workload to fail.

An incremental control plane can help address these challenges. It only computes the data plane changes that correspond to configuration changes or events. The operator must specify how input changes translate to output changes. This avoids recomputing the entire network, reduces planning for undesirable side effects, and helps debug those that occur.

2.2 Incremental programming

Unfortunately, incremental programs are difficult to write. SDN controllers are typically written in a traditional imperative language, like Java or C++. Making these incremental can increase the amount of code by an order of magnitude [52]. It is hard to detect and fix new bugs, particularly at scale.

A networking-specific challenge stems from control plane computations that require recursion or iteration, such as graph reachability for routing tables. The SDN controller receives periodic updates from routers. It applies each update to the network topology graphs and computes a form of all-pairs shortest-path (with constraints imposed by routing policies). This iteration cannot be expressed by standard database queries. But it can be implemented using recursive queries, that compute routing updates until no more changes are produced. Traditional database techniques for incremental view maintenance do not support such queries, but DDlog does.

Incrementally programmed networked systems reflect these difficulties. Consider `ovn-controller`, OVN's local controller daemon written in C. The past half-decade has seen multiple efforts to make the code incremental. The first consisted of 21 patches [41]. Due to issues with reliability and verifying correctness, the changes were reverted [40]. An alternative implementation provided by eBay followed a more disciplined approach with an engine based on C callbacks. This reduced latency by 3× and CPU cost by 20× in production. It was eventually merged upstream after much back-and-forth, and several bugs were later found in production. Many limitations and difficulties remain. The developer must explicitly identify incremental changes. The code's complexity makes it difficult to understand, to update, or to confirm an update's success [61]. It is also difficult to test, since many code paths are only exercised when a deployment takes a particular series of steps to arrive at a given configuration.

While any program can be made incremental, some may not see performance benefits. Developers must consider their applications' usage patterns. For example, OVN's load balancer benchmark cold starts `ovn-controller` with large

load balancers and then deletes each. This is a worst-case for incremental computation: changes occur multiple times and cannot be easily parallelized, but automatically incrementalizing the code still requires memory-intensive data indexing. On this benchmark, a DDlog controller took 2× the CPU time and 5× the RAM as the C implementation [47].

2.3 Programmable data planes

The rise of data plane programmability raises additional issues but introduces a mechanism to address the above control plane challenges. Network devices now expose low-level packet processing logic to the control plane through a standardized API. This can be leveraged when writing the control plane.

Data plane languages are seeing strong industry adoption. Mainstream chip vendors are commercializing programmable switching ASICs (application-specific integrated circuits). Broadcom's Trident-4 and Jericho-2 are programmable using NPL [26], while Intel/Barefoot's Tofino [3] and Cisco's Silicon One [15] support P4 [9]. In particular, P4 supports many targets, from ASICs to software switches [39].

The execution of programmable data planes is driven by *policies* created by the control plane. These policies are encoded into *table entries* written by the control plane and read by the data plane. These table entries generalize traditional forwarding table entries and can encode a rich set of policies (e.g., forwarding, load-balancing, firewall rules, encapsulation and decapsulation, etc.). A data plane table can be conceptualized as a *database view* for the global control plane state relevant to the controlled device. For example, a switch's forwarding table entries are the entries from the global controller routing table describing the links connected to the switch. This transforms programming data plane policies into a traditional incremental view maintenance program [19]—the exact problem solved by an incremental control plane.

In summary, an incremental approach could improve control plane scalability. But the difficulties of traditional incremental programming have hindered deployment. Programmable data planes motivate a possible solution.

3 NERPA

We envision a unified environment for programming the entire network. This should provide network operators with correctness and scalability guarantees. Relational database abstractions model management plane entities and data plane tables. A fully and automatically incremental control plane program sits between them. These relations are used in rules that define how data plane table entries are computed from management plane policies. The network developer writes a fully type-checked program that spans the entire network.

The Nerpa programming framework coordinates three pieces of software, one per network plane:

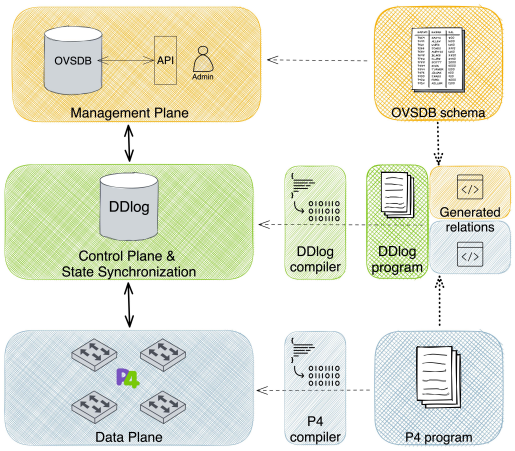

Figure 4: The vision for Nerpa, using the tools in our prototype implementation. The network programmer provides the OVSDB schema, DDlog program, and P4 program.

Management plane: The system administrator configures the management plane by populating and modifying the contents of a database instance. The database schema represents the network's structure and policies. Tables represent network links, network devices (e.g., switches, interfaces, virtual machines), high-level administrative structures (e.g., administrative domains), security policies, and more.

Control plane: The control plane is driven by two different kinds of input relations: (1) relations representing the current network configuration, obtained from the management database and (2) relations representing notifications from data plane packets and events. The control plane computes output relations, which correspond to tables in the managed data planes. An incremental control plane only computes *changes* to output relations given *changes* to input relations.

Data plane: The Nerpa controller, in charge of state synchronization, installs the data from the controller output relations as entries in the programmable data plane tables.

Nerpa generates the code that interfaces between planes. This automates tasks that previously required writing glue code. The (DDlog) schema of the relations for the control plane is generated from the schemas of the management plane and the data plane program tables. The Nerpa controller reads changes from the management plane and transforms them to inputs to the control plane program. When the controller receives a message from the data plane, it transforms the message into a row insertion into an input relation. This input relation's contents can also influence the controller's behavior, forming a feedback loop. In the compilation process, Nerpa typechecks the data definitions and database schema, ensuring that only well-formed messages are exchanged.

4 IMPLEMENTATION

Our current prototype implementation is shown in Fig. 4.

4.1 Prototype Technologies

We use the Open vSwitch Database (OVSDB) [48] for the management plane. OVSDB is well-suited for state synchronization, because it can stream a database's ongoing series of changes, grouped into transactions, to a subscriber that registered for these events.

We use DDlog to implement a centralized control plane. Note that each data plane does also have a synthesized *local* control plane, which programs tables on the local devices and relays events to the centralized control-plane using the P4Runtime API. DDlog has several key properties that improve on past incremental and relational languages.

Streaming APIs for performance: At runtime a DDlog program accepts a *stream of updates* (i.e. from OVSDB) to input relations – inserts, deletes, or modifications. It produces a corresponding stream of updates to the computed output relations. DDlog changes are also grouped into transactions. These maintain important policy invariants and are much easier to reason about than events or database triggers.

Types for correctness: Pure Datalog lacks concepts like types, arithmetic, strings, or functions. DDlog's powerful type system includes Booleans, integers and floats, and data structures like structures, unions, vectors, and maps. These can be stored in relations and manipulated by rules (queries). DDlog can perform many operations directly over structured data, including the full relational algebra. Rules can include negation (like SQL "EXCEPT"), recursion, and grouping.

Procedural language for expressivity: DDlog has a powerful procedural language that can express many deterministic computations, used in more complex network features, e.g., string processing, regular expressions, iteration, etc.

We use P4 to program the data plane. P4 has emerged as the generally preferred language for data plane programming with a robust and growing ecosystem. In particular, the P4Runtime API specifies how the control plane can control the elements of a P4-defined data plane. Our current prototype assumes a single P4 program for all network devices. But our solution can generally support multiple classes of devices (e.g., spine, leaf switches), each running a different P4 program. The management plane relations should reflect these classes.

In Nerpa the glue code, such as the SDN controller and state synchronization pieces, are written in Rust. Rust's low-level control and memory safety fit Nerpa's goals well. Rust can also be easily linked against existing Java or C++ programs. DDlog programs are also compiled to Rust by the DDlog compiler. The Rust libraries for interfacing with OVSDB and P4Runtime are in our repository [2]. This also includes `p4c-of`, which compiles P4 into OpenFlow and allows the use of high-performance software switches [49].

Nerpa helps developers use these technologies more easily and with correctness guarantees. Generating the APIs between layers reduces verbose, error-prone interoperability work when using the technologies individually. While a learning curve does still exist, particularly for the control plane's Datalog syntax, ongoing work provides a direct Rust-based query interface to the same powerful abstractions [1, 11].

4.2 Control and Data Plane Co-Design

Data exchange between the different planes requires an intermediate data representation. The control plane reads input changes from the management plane and writes output changes to the data plane. The data plane can also send notifications to the control plane, as in MAC learning. In Nerpa, changes from the management plane are represented by changes in OVSDB state. Communication between the control plane and data plane uses the P4Runtime API. Packet digests send notifications to the control plane, and output changes can modify entries in the match-action tables.

Since all communication flows through the control plane, DDlog relations serve as the natural intermediate data representation. Nerpa's tooling generates an input relation for the controller for each table in the OVSDB management plane; it also generates a controller input relation for each packet digest in the P4 program. An output relation for the controller is generated for each match-action table in the P4 program. Finally, generated helper functions in Rust convert data between P4Runtime and DDlog types. This enables close integration and co-design of the control plane and data plane.

As a brief example, the code snippets in Figure 5 present a simplified version of VLAN assignment. The declaration of the output relation `InVlan` is generated from a P4 match-action table, as seen in Fig. 5(a). The input relation `Port` is generated from an OVSDB table shown in Fig. 5(b). A programmer can write a Datalog rule as in Fig. 5(c) to compute the contents of the output relation from the input relation.

4.3 Example: Simple Network Virtual Switch

The Nerpa repository includes `snvs`, a simple network virtual switch. This implements key networking features, including VLANs, MAC learning, and port mirroring. The Nerpa integration test executes the full network stack, using OVSDB, the DDlog runtime, and the P4 behavioral simulator BMv2.

As a preliminary scalability evaluation, we added 2,000 ports to the system. We then measured the time between (1) the OVSDB client reading a new port from OVSDB and (2) the data plane entry being added to the P4 table. The first time difference noted was 0.013 seconds, and the last was 0.018 seconds. This scaling demonstrates incrementality at work.

While imperfect, lines of code (LOC) help quantify the maintenance challenges for developers discussed in §1. `snvs` consists of 350 LOC of DDlog (250 of rules, 100 of generated relations); 300 of P4; 5 OVSDB tables with 2–5 fields each; and 50 of generated Rust glue code. 700 total LOC is at least an order of magnitude less than an incremental implementation of similar features in Java or C [52].

```
table InVlan {
    key = {
        std_meta.in_port: exact @name("port");
        hdr.vlan.isValid(): exact @name("has_vlan") @nerpa_bool;
        hdr.vlan.vid: optional @name("vid");
    }

    actions = {
        Drop;
        SetVlan;
        UseTaggedVlan;
    }
}
```

```
output relation InVlan(
    port: bit<9>,
    has_vlan: bool,
    vid: Option<bit<12>>,
    priority: bit<32>,
    action: InVlanAction
)

typedef InVlanAction = InVlanActionDrop
    | InVlanActionSetVlan {vid: bit<12>}
    | InVlanActionUseTaggedVlan
```

(a) A P4 match-action table and the generated DDlog output relation of the controller. The P4 tables produce *output* control-plane relations because the P4 program is the controlled entity.

```
"Port" {
    "columns": {
        "id": {
            "type": {
                "key": {"type": "integer"}
            }
        },
        "tag": {
            "type": {
                "key": {"type": "integer"},
                "min": 0, "max": 1
            }
        },
    }
}
```

```
input relation Port (
    _uuid: uuid,
    id: integer,
    tag: Option<integer>,
)

primary key (x) x._uuid
```

(b) An OVSDB schema fragment of the management plane and the generated DDlog input relation for the control plane. Management-plane relations produce *input* control relations.

```
input relation Port (
    _uuid: uuid,
    id: integer,
    tag: Option<integer>,
)

primary key (x) x._uuid
```

```
output relation InVlan (
    port: bit<9>,
    has_vlan: bool,
    vid: Option<bit<12>>,
    priority: bit<32>,
    action: InVlanAction
)
```

```
InVlan(port, false, None, 1, InVlanActionSetVlan{vid}) :-
    Port(.id = port, .tag = tag),
    var vid = match tag {
        None -> 0,
        Some{v} -> v
    }.
```

(c) A hand-written DDlog rule executed by the control plane, computing the output relation from the input relations.

Figure 5: Nerpa examples. Within each subfigure, color is used to mark corresponding parts of input and output.

5 RELATED WORK

In this section, we discuss existing attempts to make programmable network systems more scalable and correct.

Control plane scalability. Past work has used either relational, database-style abstractions or incremental computation. The few that incorporated both were limited by their chosen tools. None leveraged the data plane.

Various database abstractions and principles have been used to model networks. Many commercial systems use databases to represent network state [6, 8, 10, 24, 42, 55, 56, 58]. Modeling networks as various database is an old idea, including SNMP [20]. These range from SQL dialects [50] to Prolog-based monitoring [51, 59]. Declarative networking [33, 34], the most relevant of this work, explored distributed Datalog-based languages for programming networks and distributed systems. It primarily implemented distributed routing protocols in declarative languages. This inspired a family of Datalog-based languages targeting networks [7, 13, 14, 21, 27, 28, 37]. But none used incremental computation and programmed both the control and data planes.

Two especially relevant languages are Flowlog [44] and Nlog [31]. Flowlog programs both the control and data planes. It models everything, including events and packets, as tables. But it does not support recursion, iteration, or joins, and is not incremental. Both incremental and relational, Nlog is most similar to DDlog and has been used in industrial-scale network virtualization. When system state changes, an Nlog program decides a new virtual forwarding policy. But it shares Flowlog's limits: it does not support recursion or negation, lacks a type system, and requires external C++ functions for many operations. It does not leverage the data plane for correctness or scalability. It cannot modify controller state.

Networking-specific languages. Advances generally focus on a network subsystem, particularly the data plane. The most relevant work verifies the data plane program at network devices. Early tools analyzed a snapshot of the complete data plane [4, 5, 29, 35, 36]. Newer solutions [22, 25, 28, 30, 60] use incrementality in production. But none apply the data plane to control plane design. We extend past work to improve other parts of the network.

6 CONCLUSIONS

We combine several technologies: SDN, for higher-level control over network policy; programmable data planes, to define packet processing; and compilers for incremental programming languages, to make traditional algorithms incremental.

These tools help rethink network programming across the entire stack. We believe that this can improve both correctness and scalability. Interfaces can be generated from the management and data planes. Programs can be type-checked across all layers. This increases developers' confidence in system correctness and, in turn, their feature velocity. An incremental control plane helps scale the system.

We plan to validate our vision through bottom-up implementations of increasingly complex network programs. Beyond our first steps in network virtualization, many other applications exist: radio networks, edge computing, cloud monitoring and control, and more.

REFERENCES

[1] Database stream processor. https://github.com/vmware/database-stream-processor. Retrieved September 2022.

[2] Nerpa: Network programming with relational and procedural abstractions. https://github.com/vmware/nerpa. Retrieved October 2022.

[3] Barefoot Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2020.

[4] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44, 2010.

[5] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126, 2014.

[6] Arista. EOS: The next generation extensible operating system. https://www.arista.com/assets/data/pdf/EOSWhitepaper.pdf, 2016.

[7] H. Ballani and P. Francis. CONMan: A step towards network manageability. In *SIGCOMM*, August 2007.

[8] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an open, distributed SDN OS. In *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, page 1–6, 2014.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[10] Broadcom Corporation. Broadcom SDKLT. https://github.com/Broadcom-Network-Switching-Software/SDKLT, October 2017. Retrieved January 2021.

[11] M. Budiu, F. McSherry, L. Ryzhyk, and V. Tannen. DBSP: Automatic incremental view maintenance for rich query languages, March 2022.

[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, page 1–12, 2007.

[13] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative configuration management for complex and dynamic networks. In *Conference on emerging Networking EXperiments and Technologies (Co-NEXT)*, 2010.

[14] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. DECOR: DEClarative network management and OpeRation. *SIGCOMM Computer Communication Review (CCR)*, 40(1):61–66, Jan. 2010.

[15] R. Chopra. ONE silicon, ONE experience, MULTIPLE roles. https://blogs.cisco.com/sp/one-silicon-one-experience-multiple-roles, December 2019.

[16] B. Davie, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Gude, A. Padmanabhan, T. Petty, K. Duda, and A. Chanda. A database approach to SDN control plane design. *SIGCOMM Computer Communication Review (CCR)*, 47(1):15–26, Jan. 2017.

[17] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *ACM Internet Measurement Conference*, 2015.

[18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *SIGCOMM Computer Communication Review (CCR)*, 38(3):105–110, July 2008.

[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 157–166, Washington, D.C., USA, 1993.

[20] D. Harrington, R. Preshun, and B. Wijnen. RFC 3411: An architecture for describing simple network management protocol (SNMP) management frameworks. https://tools.ietf.org/html/rfc3411, December 2002. IETF.

[21] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *Workshop on Research on Enterprise Networking (WREN)*, pages 1–10, 2009.

[22] A. Horn, A. Kheradmand, and M. Prasad. Delta-Net: Real-time network verification using atoms. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 735–749, 2017.

[23] V. Inc. VMware NSX network virtualization and security platform. https://www.vmware.com/products/nsx.html. Retrieved 2021.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, pages 3–14, 2013.

[25] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, et al. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 200–213. 2019.

[26] M. Kalkunte. Broadcom's new Trident 4 and Jericho 2 switch devices offer programmability at scale. https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale, June 2019.

[27] N. P. Katta, J. Rexford, and D. Walker. Logic programming for software-defined networks. In *Workshop on Cross-Model Design and Validation (XLDI)*, 2012.

[28] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 99–111, Lombard, IL, Apr. 2013.

[29] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, 2012.

[30] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–27, 2013.

[31] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 203–216, Seattle, WA, Apr. 2014.

[32] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Symposium on Operating System Design and Implementation (OSDI)*, page 351–364, USA, 2010.

[33] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: Language, execution and optimization. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, page 97–108, 2006.

[34] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM (CACM)*, 52(11):87–95, Nov. 2009.

[35] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 499–512, 2015.

[36] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. *SIGCOMM Computer Communication Review (CCR)*, 41(4):290–301, 2011.

[37] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Unified declarative platform for dynamic overlay composition. In *Conference on emerging Networking EXperiments and Technologies (Co-NEXT)*, 2008.

[38] 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.

[39] O. Michel, R. Bifulco, G. Retvari, and S. Schmid. The programmable data plane: abstractions, architectures, algorithms, and applications. *ACM Computing Surveys (CSUR)*, 54(4):1–36, 2021.

[40] R. Moats. ovn-controller: Back out incremental processing. https://github.com/openvswitch/ovs/commit/926c34fd7c2080543bf3ee63a4830e0dc5c4af12, August 2016.

[41] R. Moats. [ovs-dev][patch v21 0/8] add incremental processing., July 2016.

[42] J. C. Mogul, D. Goricanec, M. Pool, A. Shaikh, D. Turk, B. Koley, and X. Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 403–418, Santa Clara, CA, Feb. 2020. USENIX Association.

[43] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, Sept. 2016.

[44] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 519–531, 2014.

[45] OVN: Oven virtual network for Open vSwitch. https://github.com/openvswitch/ovs/tree/master/ovn. Retrieved January 2021.

[46] J. Pettit, B. Pfaff, H. Zhou, and R. Moats. Practical OVN: Architecture, deployment and scale of OpenStack networking. http://openvswitch.org/support/slides/OVN_Austin.pdf, April 28 2016. OpenStack Summit.

[47] B. Pfaff. Scaling sdn policy distribution. In *P4 Workshop*, 2022.

[48] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of Open vSwitch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, Oakland, CA, May 2015.

[49] B. Pfaff, D. Sur, L. Ryzhyk, and M. Budiu. P4 in open vswitch with ofp4. In *P4 Workshop*, 2022.

[50] C. M. Rogers. ANQL — an active networks query language. In *Active Networks*, pages 99–110, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[51] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a logic language for system health monitoring in distributed systems. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, page 31–37, 2002.

[52] L. Ryzhyk. DDlog tutorial for OVN developers. 2019.

[53] L. Ryzhyk and M. Budiu. Differential Datalog. In *Datalog 2.0*, Philadelphia, PA, June 4-5 2019.

[54] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab. Sdn controllers: A comparative study. In *2016 18th mediterranean electrotechnical conference (MELECON)*, pages 1–6. IEEE, 2016.

[55] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A network-state management service. In *SIGCOMM*, pages 563–574, 2014.

[56] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng. Robotron: Top-down network management at Facebook scale. In *SIGCOMM*, SIGCOMM, page 426–439, 2016.

[57] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[58] A. Wang, X. Mei, J. Croft, M. Caesar, and B. Godfrey. Ravel: A database-defined network. In *ACM Symposium on SDN Research (SOSR)*, 2016. http://ravel-net.org/.

[59] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. *SIGCOMM Comput. Commun. Rev.*, 34(1):15–20, Jan. 2004.

[60] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. APKeep: Realtime verification for real networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 241–255, 2020.

[61] H. Zhou. OVN controller incremental processing. In *Open vSwitch 2018 Fall Conference*, San Jose, California, 2018. http://www.openvswitch.org/support/ovscon2018/.