Nerpa: Network Programming with Relational and Procedural Abstractions

Debnil Sur, Ben Pfaff, Leonid Ryzhyk, and Mihai Budiu
{dsur,bpfaff,lryzhyk,mbudiu}@vmware.com
VMware

ABSTRACT

We introduce Nerpa, a new methodology for building programmable networks. Nerpa automates many aspects of the process of programming the network stack. To aid correctness, it ensures type safety across the management, control, and data planes. To improve scalability, an incremental control plane recomputes state in response to network configuration changes. We have published an implementation and examples.

1 INTRODUCTION

In the most popular approach for programming networks, the developer separately programs the management, control, and data planes. In a common approach, the management plane is implemented as an API backed by a database, the control plane as an SDN controller written in an imperative language such as Java or C++, and the data plane using flow-programmable switch software or hardware [1–6, 8].

We have identified two primary challenges in our experience building systems in this form. **Correctness** is the first challenge. The control plane typically programs the network by issuing installing small program fragments, e.g. OpenFlow flows, in network devices. The controller serves as a specialized compiler converting policies into these program fragments. As an SDN system adds features over time, flow rule fragments for various tables and priorities end up scattered across a large codebase. It becomes difficult for the developer to be confident of overall correctness across the feature combination matrix.

Scalability is the second challenge: as the system grows, the SDN controller must still respond quickly to changes. In return, this demands **incrementality**. In response to a change, the controller should not recompute and redistribute the entire network state. Instead, the recomputation should be proportional to the amount of modified state. Writing an incremental controller in an imperative language such as Java or C++ demands either an unnatural coding style or ad hoc, fragile support for incremental changes that seem important in practice [7, 10].

We present Nerpa, a prototype of a programming framework intended to address these classes of issues. To address correctness, Nerpa pairs the DDlog control plane with a P4 data plane to write a complete type-checked program. This provides more obvious correctness than an OpenFlow-like data plane. While the latter has some structure, it is not apparent to the controller, which just generates program fragments. To address scalability, the programmer writes a Nerpa control plane in Differential Datalog, or DDlog for short, a declarative language whose implementation is fully and automatically incremental [9]. Our implementation is available as an open-source project with an MIT license at https://github.com/vmware/nerpa, including a tutorial and a demo application.

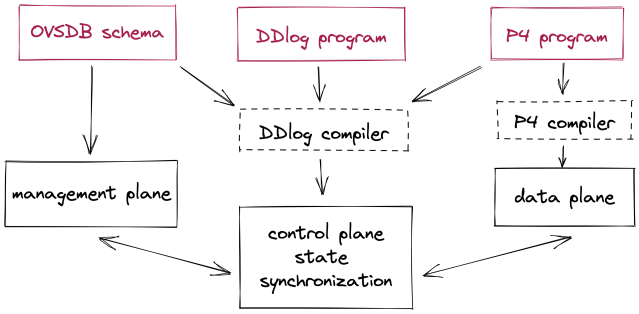

Figure 1: The vision for Nerpa. The network programmer writes the red boxes.

2 DESIGN

The Nerpa programming framework coordinates three pieces of software. These correspond to the three classes of computations executed by network devices:

Management plane: The system administrator configures the Nerpa management plane by populating and modifying the contents of an Open vSwitch Database (OVSDB) instance. Its schema represents the high-level structure of the network.

Control plane: A DDlog program computes a set of output relations from the contents of some input relations. The Nerpa DDlog program has two kinds of input relations: (1) representing the current network configuration, synchronized from the management database and (2) representing notifications from data plane packets and events. The control plane output relations correspond to entries for P4 tables. The Nerpa programmer implements the control plane program to compute the output relations as a function of the input relations. The DDlog compiler automatically makes this computation an incremental process.

Data plane: The data plane is programmed using P4. The Nerpa controller uses the P4Runtime API to install DDlog output relations as table entries in the P4-enabled switch.

A Nerpa programmer supplies three files, as shown in Figure 1: an OVSDB schema, which defines the management plane; a DDlog program, whose rules define the control plane; and a P4 program, which implements the data plane. To interface between these software programs, Nerpa automates many tasks that previously required writing glue code, by generating the code that orchestrates

Author's address: Debnil Sur, Ben Pfaff, Leonid Ryzhyk, and Mihai Budiu, {dsur,bpfaff, lryzhyk,mbudiu}@vmware.com, VMware.

the data movement between the planes. To facilitate this, DDlog input and output relations are generated from the OVSDB schema and the compiled P4 program. The Nerpa controller reads changes from OVSDB and transforms them to inputs to the DDlog program. It also transforms DDlog outputs into P4 table entries, and writes those entries to the switch using the P4Runtime API. When the P4 program sends a digest back to the Nerpa controller, the controller transforms it into input to a DDlog relation whose contents can also influence the controller's behavior, forming a feedback loop. In the compilation process, Nerpa typechecks the data definitions and ensures that only well-formed messages are exchanged.

3 IMPLEMENTATION

We describe some details of our current prototype implementation.

3.1 Language-Specific Tooling

The glue layers between all these services are all written in Rust. Rust's low-level control and memory safety fit Nerpa's goals well. DDlog programs are also compiled to Rust by the DDlog compiler; DDlog is in the core of the Nerpa stack. As a result, we built Rust libraries for interfacing with OVSDB and P4Runtime. The OVSDB Rust library uses the Rust `bindgen` crate to generate Rust foreign-function interface bindings to OVSDB's C codebase. The P4Runtime Rust library uses the P4Runtime Protocol Buffer definition files to generate Rust code for the API calls. It then exposes an end user friendly API. Both libraries are included in the Nerpa repository. We hope other projects in the P4 ecosystem find them useful.

3.2 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. A packet digest can be used to send notifications to the control plane over the Stream RPC. Output changes modify entries in the match-action tables using the Write RPC.

Since all communication flows through the control plane, DDlog relations serve as the natural intermediate data representation. To represent inputs from the management plane we used `ovsdb2ddlog`, a tool which generates DDlog relations from an OVSDB schema. We implemented `p4info2ddlog`, a tool to generate DDlog relations from a P4 program. It consumes the "P4info" binary file, produced by the p4 compiler, describing the tables and other objects in the P4 program. From this, `p4info2ddlog` generates an input relation for each packet digest and an output relation for each match-action table. It also generates helper functions in Rust to convert data between P4Runtime and DDlog types. This approach enables co-design of the control plane and data plane and a close integration between the two.

3.3 Example: Simple Network Virtual Switch

The Nerpa repository includes a simple network virtual switch as an example implemented in Nerpa, called `snvs`. This implements

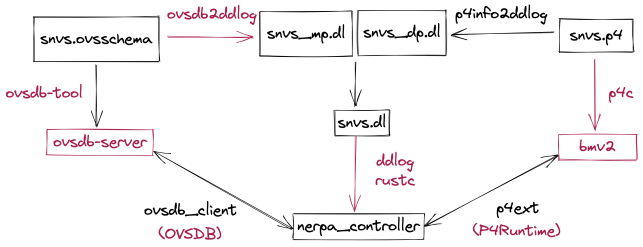

Figure 2: The `snvs` example program as processed by Nerpa. Black text represents a tool or program written as part of Nerpa, while red represents an external program.

several important networking features, including VLANs, MAC learning, and port mirroring. The Nerpa integration test executes all layers of the stack, using OVSDB, the DDlog runtime, and the P4 behavioral simulator BMv2.

The code snippets below implement a very simplified version of the VLAN assignment feature. We generate an output relation from a P4 match-action table and an input relation from an OVSDB table. A Datalog rule then derives the output relation from the input relation. This shows how Nerpa's pieces fit together.

P4 match-action table

```
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;
  }
}
```

DDlog output relation generated from the P4 program

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

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

OVSDB schema

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

DDlog input relation generated from the OVSDB schema

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

DDlog rule for VLAN assignment written by programmer

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

4 CONCLUSIONS

Nerpa uses relational and procedural abstractions to improve the correctness and scalability of network programs. OVSDB and the DDlog data representation are used in the relational, incrementally programmed control plane. This is co-designed with the imperative data plane program, written in P4. We have provided a prototype and example program of a simple network virtual switch. In future, we plan to implement increasingly complex network programs.

REFERENCES

[1] 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 *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, page 1–6, 2014.

[2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, page 1–12, 2007.

[3] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM Computer Communication Review (CCR)*, 38(3):105–110, July 2008.

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

[5] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 203–216, Seattle, WA, April 2014.

[6] Teemu Koponen, Martin 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 System Design and Implementation (OSDI)*, page 351–364, USA, 2010.

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

[8] Justin Pettit, Ben Pfaff, Han Zhou, and Ryan Moats. Practical OVN: Architecture, deployment and scale of OpenStack networking. http://openvswitch.org/support/slides/OVN_Austin.pdf, April 28 2016. OpenStack Summit.

[9] Leonid Ryzhyk and Mihai Budiu. Differential Datalog. In *Datalog 2.0*, Philadelphia, PA, June 4-5 2019.

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