Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations

(Brief Reflections on Abstractions for Network Programming)

Ryan Beckett
Microsoft
Ryan.Beckett@microsoft.com

Ratul Mahajan
University of Washington
and Intentionet
ratul@cs.washington.edu

Todd Millstein
UCLA
and Intentionet
todd@cs.ucla.edu

Jitendra Padhye
Microsoft
padhye@microsoft.com

David Walker
Princeton University
dpw@cs.princeton.edu

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

We reflect on the historical context that lead to Propane, a high-level language and compiler to help network operators bridge the gap between network-wide routing objectives and low-level configurations of devices that run complex, distributed protocols. We also highlight the primary contributions that Propane made to the networking literature and describe ongoing challenges. We conclude with an important lesson learned from the experience.

CCS CONCEPTS

• **Networks** **Network control algorithms**; **Network reliability**; **Network management**; • **Software and its engineering** **Automated static analysis**; **Domain specific languages**;

KEYWORDS

Propane; Domain-specific Language, BGP, Synthesis, Compilation, Fault Tolerance, Distributed Systems

1 HISTORICAL CONTEXT

When Software-Defined Networking (SDN) burst onto the scene around 2008, it offered a new means to control packet forwarding from a single centralized vantage point. Neither network engineers nor researchers would be bound to the slowly-evolving technology provided by traditional routers. Instead, for many in the networking community, SDN opened up the opportunity to program new algorithms that use network-wide data to optimize performance. The decisions made by these new algorithms could then be realized by pushing specific packet-processing rules out to a distributed collection of OpenFlow-enabled switches.

While many networking researchers examined *what* new algorithms could be implementing using SDN, a few, in collaboration with programming languages researchers, began to explore *how* best to express these algorithms. With a plethora of new algorithms and much new infrastructure being developed, there are were bound to be many bugs, which in the short term at least, might make such network services less reliable. Hence, it made sense to consider whether there might be new programming paradigms that, through their design, might be able to cut down on certain classes of software errors.

Frenetic was the first language that provided programmers with a higher-level interface to OpenFlow switches [4]. It responded to control events from switches, such as failures or traffic statistics, and used the information it received to compute declarative data plane policies. These policies specified much of the same information as lists of OpenFlow rules, but unlike OpenFlow, they were composable. For instance, one could specify a network monitoring policy separately from a network routing policy and then implement both policies simultaneously, without fear of conflicts, by invoking a single function. Under the covers, Frenetic generated OpenFlow rules and installed them "consistently" [11] to avoid transient bad behaviors. Composition and consistent updates raised the level of abstraction at which network engineers could implement policy and helped fend off some of the errors that can arise due to poorly-synchronized rule installation.

Soon afterwards, inspired by Frenetic, researchers looked to extend the set of policies network engineers could craft to specify intended routing behavior. For instance, NetCore [8] generalized Frenetic's simple packet patterns, allowing users to classify traffic using more general boolean predicates over packets, and demonstrated it was possible to compile such programs to OpenFlow. Pyretic [9] extended NetCore's policy language to allow a new kind of *sequential composition* to complement Frenetic's *parallel composition*). FatTire [10] went a step further, adding the abstraction of *paths*, specified by regular expressions. These paths represented the first real *network-wide* abstraction.

Thus, in the five-year period between 2008 and 2013, researchers had converged on two essential elements of stateless, network-wide data plane programming: (1) regular expressions to describe paths and (2) boolean predicates to classify packets and to choose the paths. In 2014, Anderson *et al.* [1] recognized that network programming languages possessing this combination of features were instances of a well-known algebraic structure called a Kleene Algebra with Tests

(KAT) [6]. Through this connection, a rich semantic theory and a host of algorithmic techniques could be adapted from the world of formal algebra to the study of network programs.

2 PROPANE'S CONTRIBUTIONS

While SDN was the hot topic of the day in academia, most networks, both big and small, continued using traditional routing protocols such as OSPF and BGP. Even modern data centers, which might use SDN around the edge, often continued to use traditional routing in their core. In part, this continued use is certainly a legacy software and hardware issue. But it is also likely that traditional routers, which have the benefit of being plug-and-play, simply perform well enough in many common cases. Finally, regardless of whether a network uses SDN internally or not, it is necessary to communicate with peers, and the only current means of doing so is via BGP. Hence, interfacing with traditional distributed protocols remains essential, even in the SDN world.

The primary contribution of Propane is to demonstrate that the kinds of high-level abstractions used to specify *data plane* policy in SDN could also be adapted to specify *control plane* policy in traditional networks. In particular, fundamental network-wide abstractions such as paths, specified via regular expressions, and boolean expressions, used to classify traffic, could be combined, as in the SDN world, to denote the expected flow of data plane traffic.

However, there are two significant differences at the level of operator specifications. First, whereas SDN programs concern themselves only with intra-domain routing, Propane could simultaneously specify both intra- and inter-domain traffic patterns. Hence, in a single uniform notation, network engineers could specify the way they wished to interact with their peers and how they wanted to route traffic through their own network. Second, when failures occur or a route is withdrawn by the peer, the control plane will recompute its routing decisions and choose a backup route—these backups must be specified up front, along with the primary routes. Propane added relations between paths to allow operators to specify path preferences such that a lower-preference path is taken only when a higher-preference path is unavailable.

A second key contribution of the Propane is to demonstrate how global specifications provided can be compiled and implemented correctly using traditional distributed protocols. To do so, the Propane system uses BGP as the implementation protocol because of its flexibility. However, it turns out that not all regular sets can be implemented by path vector protocols like BGP. Consequently, Propane defined an analysis to inform users when their policies cannot be implemented in BGP. The original SIGCOMM paper describes the analysis and compilation process briefly; Beckett's thesis [2], which won the 2018 ACM SIGCOMM dissertation award, fills in the details.

3 CONTINUING CHALLENGES

While the original work supplied a proof of concept, much work remained, and continues to remain, to provide the surrounding infrastructure necessary for Propane to be adopted by mainstream network engineers.

Scaling. One of the first challenges was to improve the scaling properties of Propane to make it attractive to engineers of large networks. To tackle this problem, in a follow-on system called Propane/AT [3], we stole an idea in use in industry: the notion of templates. Rather than specifying concrete locations and destinations, we allow Propane-like specifications to include template variables. In a first compilation phase, we analyze the Propane specification with respect to these parameters and an *abstract topology* and ensure that any legal instantiation of the parameters will generate a control plane and concrete topology with desirable properties (*e.g.*, the appropriate level of fault tolerance). In a second compilation phase, we instantiate the parameters and convert the abstract topology to a valid concrete topology. We found that this two-phase compilation strategy dramatically improved compilation times for Propane. Using abstract topologies, Propane can compile data center configurations up to two orders of magnitude faster. These generated configurations also allow certain kinds of incremental changes to network configuration without recompilation and reinstallation of all configurations on all switches.

Incremental deployment. A significant challenge in deploying Propane industrially is incremental deployment. At the moment, Propane is designed for new networks. How should Propane interact with existing networks that operate using completely different, ad hoc configuration system? This is a rich problem, worthy of future study, and the topic of an ongoing NSF grant [7].

One possible approach is to analyze the set of low-level configurations that are present in the existing network. Such an analysis could identify the paths allowed and synthesize a matching Propane specification. The simplest way of doing so may list all the paths through the network one-by-one. However, such a listing is likely very verbose and compact specifications are almost always much preferred over verbose ones. Another approach to finding a specification would be to try to find a compact generalization of the naive list of paths. However, doing so might involve admitting more paths in the specification than are present in the existing network. In other words, a compact generated specification might over-approximate the set of actual routes. But how much overapproximation will a user accept? What notation should one use to cut out overapproximations? These and similar questions require further research.

Incremental evolution. Once Propane is deployed, engineers will undoubtedly need to update policy from time to time to addess security vulnerabilities, add or remove hosts, expand capacity, or change routing decisions. While Propane/AT [3] provides mechanisms that allow for incremental update of the network topology, it does not provide mechanisms for changing policy. One approach to solving this problem would be to adapt recent research on network repair [5] to the problem. The idea here would be that the new Propane program would be the specification and the existing set of configurations be "repaired" to meet this new specification. Ideally, such repairs would generate a minimal set of commands that could be issued to update routers.

4 A KEY LESSON: THE IMPORTANCE OF SMALL MODELS

The programming languages that software engineers use on a day-to-day basis are extremely complicated. This is true of C++, Python, Haskell, and JavaScript. It is also true of Cisco IOS and Juniper

JunOS, the languages network operators use to program their networks. In order to understand how these languages work in a deep sense, how to program with them, and how to build reliable abstractions on top of them, one cannot tackle the whole language all at once. It is necessary to build small, simple models first—little idealized languages—which might not have every bell and whistle, but that are more uniform than their real-world counterparts. Such small models will almost always leave out elements of the real language, which will invariably frustrate practitioners. However, that is not a bug but a feature—it frees the researcher to focus on what is *left in* the small model and to ignore additional complexity that is *left out*.

In a sense, OpenFlow 1.0 was a small model for packet forwarding. It provided an understanding of the network dataplane so simple that even programming language researchers could understand it! Once we understood this simple forwarding model, we could manipulate, transform, abstract, and extend it. And then, finally, we could begin to apply the lessons learned in this simple setting to more expansive contexts. The latter step is where Propane comes in as it adapted ideas developed in the world of early SDN dataplane to the more complex setting of the traditional router. Going forward, we hope to see the networking community develop more small models for other components of the networking stack.

Acknowledgments. We thank R. Aditya, George Chen, Lihua Yuan, and the SIGCOMM reviewers for feedback on the original work. That work was supported in part by the National Science Foundation awards CNS-1161595 and CNS-1111520 as well as a gift from Cisco.

The current paper is supported in part by the National Science Foundation award 1703493. We thank Jennifer Rexford and George Varghese for many discussions about Propane over the years and for ideas concerning its evolution.

REFERENCES

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.

[2] R. Beckett. *Network Control Plane Synthesis and Verification*. PhD thesis, Princeton University, 2018. See http://arks.princeton.edu/ark:/88435/dsp01d217qs28v.

[3] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Network configuration synthesis with abstract topologies. In *PLDI*, 2017.

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[5] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu. Automatically repairing network control planes using an abstract representation. In *SOSP*, 2017.

[6] D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3), May 1997.

[7] T. Millstein, G. Varghese, and D. Walker. NeTS: Medium: Collaborative Research: Network Configuration Synthesis: A Path to Practical Deployment. https://www.nsf.gov/awardsearch/showAward?AWD_ID=1703493, July 2017. NSF CNS 1703493.

[8] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.

[9] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *NSDI*, 2013.

[10] M. Reitblatt, M. Canini, N. Foster, and A. Guha. FatTire: Declarative fault tolerance for software defined networks. In *HotSDN*, 2013.

[11] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.