Language Abstractions for Software-Defined Networks

Nate Foster
Cornell University

Michael J. Freedman
Princeton University

Rob Harrison
US Military Academy

Christopher Monsanto
Princeton University

Mark Reitblatt
Cornell University

Jennifer Rexford
Princeton University

Alec Story
Cornell University

David Walker
Princeton University

1. Introduction

For the past 30 years, networks have been built the same way: out of special-purpose devices running distributed algorithms that provide functionality such as topology discovery, routing, traffic monitoring, and access control. Recent years, however, have seen growing interest in a new kind of network architecture in which a logically-centralized *controller machine* manages a distributed collection of programmable switches. These *software-defined networks* (SDNs) make it possible for programmers to directly control the behavior of the network by configuring the packet-forwarding rules installed on switches [6]. Figure 1 depicts the architecture of a traditional network, where each independent switch consists of tightly-integrated control and data planes, and of an SDN, where a collection of switches are managed by a single program running on the controller.[1]

SDNs both simplify existing applications and also provide a platform for developing exciting new ones. For example, to implement shortest-path routing, the controller can calculate the forwarding rules for each switch by running Dijkstra's algorithm on the graph of the network topology rather than running a more complicated distributed protocol [9]. To enforce a fine-grained access control policy, the controller can consult an external authentication server and install a custom forwarding path for each user [2]. And to balance the load between back-end servers in a data center, the controller can migrate flows in response to server load and network congestion [10].

Although SDNs provide an architecture that makes it possible to program a collection of switches, the network is still a distributed system, and all of the usual complications arise: control messages may be delayed or lost, packets may be dropped, and the devices may experience unexpected failures. Unfortunately, current SDN platforms such as NOX [4] and Beacon [1] provide a low-level interface that supports manipulating the state of individual devices, but little else. Hence, writing applications for SDNs today is a tedious exercise in low-level distributed programming.

The goal of the Frenetic project is to develop declarative constructs that make it possible for programmers to describe the behavior of a network at a suitably high-level of abstraction, leaving tedious implementation details to a compiler and run-time system. This paper summarizes the results of our initial efforts, focusing on three areas where language-based abstractions can be leveraged to provide tangible benefits to programmers.

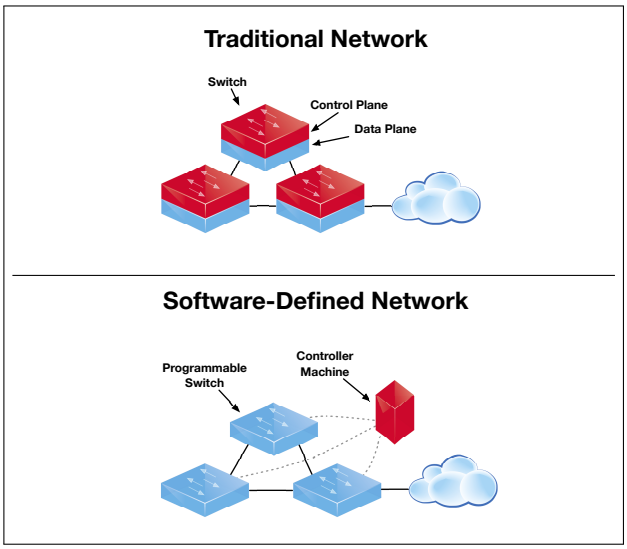

Figure 1. Network architectures.

2. Network Policy Abstractions

Today's SDN controllers support an event-driven programming model in which programs react to network events (*e.g.*, topology changes, packets for which a switch does not have a forwarding rule, etc.) by manipulating the rules installed on switches, or by crafting packets to emit onto the network. This complicates programs in several ways. For one thing, it splits control between two levels of execution—the program on the controller and the rules on the switches. Programmers must constantly consider whether installing or uninstalling rules will mask future events received at the controller, and they must explicitly coordinate multiple asynchronous events to perform simple tasks (*e.g.*, discovering the switches connected to a particular link). For another, it forces programmers to manage a host of low-level details

[1] The controller may be replicated for scalability and fault tolerance [5].

involving the switch hardware (*e.g.*, whether to use wildcard or exact-match rules to implement the desired forwarding policy, and the values to use for rule timeouts and priority levels). Frenetic's NetCore policy language abstracts away from many of these details involving the underlying distributed system [7]. Instead of having to write programs in terms of low-level network events and forwarding rules, NetCore programmers write a simple specification that captures the intended forwarding behavior of the network. The compiler partitions this specification into suitable code fragments for the controller and the switches. It also generates the communication patterns between the controller and switches, using a run-time system that handles the bookkeeping related to installing and uninstalling individual forwarding rules. A notable feature of NetCore is that it allows policies to be described in terms of arbitrary functions that cannot be directly realized on switches. To handle such policies, the compiler generates an underapproximation of the overall policy using a simple static analysis, and then uses partial evaluation to refine the underapproximation at run time using the actual packets seen in the network. Lastly, NetCore has a clear formal semantics that provides a basis for reasoning about programs.

3. Network Query Abstractions

Another complication in SDN programs today concerns the implementation of monitoring policies. Each switch maintains counters for each forwarding rule that keeps track of the total number of packets and bytes processed using it. To monitor traffic, the controller can poll the counters associated with particular rules. However, programmers must ensure that the rules installed on switches are fine-grained enough to collect the desired traffic statistics. For instance, to monitor the total amount of web traffic, the programmer must install rules that process (and count) traffic involving TCP port 80 separately from all other traffic. Managing these details is tedious and makes programs anti-modular—the rules generated by one module may be too coarse to be executed side-by-side with a different module. To support applications whose correct operation involves a monitoring component, Frenetic includes an embedded query language that provides effective abstractions for reading network state [3]. This language, which has syntax reminiscent of SQL, includes constructs for selecting, filtering, splitting, merging, and aggregating the streams of packets flowing through the network. Importantly, the language allows queries to be composed with each other and with forwarding policies, without any harmful interference. Again, the technology that makes this possible is the compiler and run-time system, which generates switch-level rules guaranteed to correctly implement all queries and the overall forwarding policy. The compiler also generates the control messages and handlers needed to query and tabulate the counters on switches.

4. Consistent Update Abstractions

Our most recent work on Frenetic focuses on the problem of implementing updates to network policy. Many SDN programs need to transition from one policy to another (*e.g.*, due to topology changes, planned maintenance, or unexpected failures). From the perspective of the programmer, it would be ideal if such transitions could be propagated atomically to the switches in the network. But atomic updates are impractical to implement—they would require disrupting the operation of the network during the update. We have been developing consistency abstractions that allow programmers to specify how updates to policy should be propagated to the devices in the network [8]. These abstractions are strong enough to provide useful guarantees about application behaviors, and yet relaxed enough to admit practical implementations. For example, *per-packet consistency* ensures that every packet flowing through the network uses a single version of the policy and not a mixture of old and new policies. This preserves all properties that can be expressed in terms of individual packets and the paths they take through the network—a class of properties that subsumes important structural invariants such as basic connectivity and loop-freedom, as well as access control policies. Going a step further, *per-flow consistency* ensures that sets of related packets are processed with the same policy. This can be used to enforce rich properties including the ordering of packets within a flow. It is also needed for applications such as load balancers, which need to ensure that packets in the same flow reach the same destination to avoid breaking connections. Frenetic provides an ideal platform for exploring such abstractions, as the compiler and run-time system can be used to perform the tedious bookkeeping related to implementing network updates. For example, to provide per-packet consistency, the run-time system can generate rules that tag packets with policy-version numbers and use two-phase commit to propagate rules to switches. Per-flow consistency can be implemented in similar fashion, with some additional analysis to identify existing flows. We are currently working to extend our compiler and run-time system with these abstractions.

5. Example Frenetic Application

Figure 2 presents code for a Frenetic application that: (1) learns the locations of hosts on the network, (2) forwards packets to known destinations, (3) floods packets to unknown destinations, and (4) periodically reports the largest k sources of traffic. Space constraints preclude giving a full explanation of the code. Details can be found in the original paper on Frenetic [3]. Note however that the program does *not* include any code that explicitly manipulates low-level switch state, sends control messages to switches, or handles asynchronous network events. Instead, the entire program consists of a few declarative specifications that are composed together in a simple and elegant way.

```
#####################
# Ethernet learning #
#####################
# Store learned mac addresses in table
def learn_mac(((switch,mac),packet),table):
   table[switch][mac] = inport(header(packet))
   return table

# Convert table to policy
def make_policy(table):
   policy = defaultdict(lambda:[])
   for switch,learned in table.items():
      patterns = false_fp()
      for mac,port in learned.items():
         rule = Rule(dstmac_fp(mac),[forward(port)])
         policy[switch].append(rule)
         patterns = patterns | dstmac_fp(mac)
      flood_rule = Rule(true_fp() - patterns,[flood()])
      policy[switch].append(flood_rule)
   return policy

# Initial table
init_table = defaultdict(lambda:{})

# Main function
def ethernet_learning():
   q = Select('packets') * \
       GroupBy(['switch','srcmac']) * \
       SplitWhen(['inport']) * \
       Limit(1)
   sf = Accum(init_table,learn_mac) >> \
        Lift(make_policy)
  return q  >> sf >> Register()
```

```
######################
# Top-k heavy hitters #
######################
# Constants
K = 2
WINDOW = 15

# Store new values in stats
def tabulate_stats(new,stats):
   for host,bytes in new.items():
      stats[host] += int(bytes)
   return stats

# Convert stats to list of top-k hosts
def topK(stats):
   l = sorted(stats.items(), key=lambda x:x[1])
   l.reverse()
   return l[:K]

# Initial stats
init_stats = defaultdict(lambda:0)

# Main function
def heavy_hitters():
   q = Select('sizes') * \
       GroupBy(['srcmac']) * \
       Every(WINDOW)
   sf = Accum(init_stats, tabulate_stats) >> \
        Lift(topK)
   return q >> sf >> Print()
```

Figure 2. Example application: Ethernet learning and top-k heavy hitters.

Acknowledgments Our work is supported in part by ONR grant N00014-09-1-0770 and NSF grants CNS-1111698 and CNS-1111520. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the NSF, ONR, or the US Military Academy.

References

[1] Beacon: A java-based OpenFlow control platform. See `http://www.beaconcontroller.net`, December 2011.

[2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4), August 2009.

[3] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Tokyo, Japan*, September 2011.

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

[5] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, BC*, pages 351–364, October 2010.

[6] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[7] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, PA*, January 2012. To appear.

[8] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *ACM Workshop on ACM Workshop on Hot Topics in Networks Programmable Routers for Extensible Services of Tomorrow (HotNets), Cambridge, MA*, November 2011.

[9] Scott Shenker, Martin Casado, Teemu Koponen, and Nick McKeown. The future of networking and the past of protocols, October 2011. Invited talk at Open Networking Summit.

[10] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE), Boston, MA*, March 2011.