Languages for Software-Defined Networks

Nate Foster*, Michael J. Freedman†, Arjun Guha*, Rob Harrison‡,
Naga Praveen Katta†, Christopher Monsanto†, Joshua Reich†, Mark Reitblatt*,
Jennifer Rexford†, Cole Schlesinger†, Alec Story*, and David Walker†
*Cornell University †Princeton University ‡U.S. Military Academy

***Abstract*—Modern computer networks perform a bewildering array of tasks, from routing and traffic monitoring, to access control and server load balancing. Yet, managing these networks is unnecessarily complicated and error-prone, due to a heterogeneous mix of devices (*e.g.*, routers, switches, firewalls, and middleboxes) with closed and proprietary configuration interfaces. Software-Defined Networks (SDN) are poised to change this by offering a clean and open interface between networking devices and the software that controls them. In particular, many commercial switches support the OpenFlow protocol, and a number of campus, data-center, and backbone networks have deployed the new technology. Yet, while SDN makes it possible to program the network, it does not make it easy. Today's OpenFlow controllers offer low-level APIs that mimic the underlying switch hardware. To reach SDN's full potential, we need to identify the right higher-level abstractions for creating (and composing) applications. In the Frenetic project, we are designing simple and intuitive abstractions for programming the three main stages of network management: (i) monitoring network traffic, (ii) specifying and composing packet-forwarding policies, and (iii) updating policies in a consistent way. Overall, these abstractions make it dramatically easier for programmers to write and reason about SDN applications.**

I. Introduction

Traditional networks are built out of special-purpose devices running distributed protocols that provide functionality such as topology discovery, routing, traffic monitoring, and access control. These devices have a tightly-integrated control and data plane, and network operators must separately configure every protocol on each individual device. Recent years, however, have seen growing interest in software-defined networks (SDNs), in which a logically-centralized *controller* manages the packet-processing functionality of a distributed collection of switches. SDNs make it possible for programmers to control the behavior of the network directly, by configuring the packet-forwarding rules installed on each switch [1]. Note that although the programmer has the illusion of centralized control, the controller is often replicated and distributed for scalability and fault tolerance [2].

SDNs can both simplify existing applications and also serve as a platform for developing 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 instead of using a more complicated distributed protocol [3]. To conserve energy, the controller can selectively shut down links or even whole switches after directing traffic along other paths [4]. To enforce fine-grained access control policies, the controller can consult an external authentication server and install custom forwarding paths for each user [5]. To balance the load between back-end servers in a data center, the controller can split flows over several server replicas and migrate flows to new paths in response to congestion [6], [7].

But although SDNs makes it *possible* to program the network, it does not make it easy. Protocols such as OpenFlow [1] expose an interface that closely matches the features of the underlying switch hardware. Controllers such as NOX [8], Beacon [9], and Floodlight [10] support the same low-level interface, which forces applications to be implemented using programs that manipulate the state of individual devices. Supporting multiple tasks at the same time—such as routing and access control—is extremely difficult, since the application must ultimately install a single set of rules on the underlying switches. In addition, a network is a distributed system, and all of the usual complications arise—in particular, control messages sent to switches are processed asynchronously. Overall, writing applications for today's SDN controller platforms is a tedious exercise in low-level distributed programming.

The goal of the Frenetic project is to raise the level of abstraction for programming SDNs. To replace the low-level imperative interfaces available today, Frenetic offers a suite of declarative abstractions for querying network state, defining forwarding policies, and updating policies in a consistent way. These constructs are designed to be modular so that individual policies can be written in isolation and later composed with other components to create sophisticated policies. This is made possible in part by the design of the constructs themselves, and in part by the underlying run-time system, which implements them by compiling them down to low-level OpenFlow forwarding rules. Our emphasis on modularity and composition—the key principles behind effective design of complicated software systems—is the key feature that distinguishes Frenetic from other SDN controllers.

Our initial work on Frenetic rethinks how to support the three main pieces of the "control loop" for running a network:

- *Querying network state:* Frenetic offers a high-level query language for subscribing to streams of information about network state, including traffic statistics and topology changes. The run-time system handles the details of polling switch counters, aggregating statistics, and responding to events.
- *Expressing policies:* Frenetic offers a high-level policy language that makes it easy for programs to specify the packet-forwarding behavior of the network. Different modules may be responsible for (say) topology discovery, routing, load balancing, and access control. Individual

modules register these policies with the run-time system, which automatically composes, compiles, and optimizes them with programmer-specified queries.

- *Reconfiguring the network:* Frenetic offers abstractions for updating the global configuration of the network. These abstractions allow a programmer to reconfigure the network without having to manually install and uninstall packet-forwarding rules on individual switches—a tedious and error-prone process. The run-time system ensures that during an update, all packets (or flows) are processed with the old policy or the new policy, and never a mixture of the two. This guarantee ensures that important invariants such as loop freedom, connectivity, and access control are never violated during periods of transition between policies.

Together, these abstractions enable programmers to focus on high-level network management goals, instead of details related to handling low-level rules and events. The following sections describe each of these components in more detail. Readers interested in using the system may download our compiler, which is available online [11]. The Frenetic web site also contains technical papers and reports which discuss the language design, compiler infrastructure, update mechanisms, and other technology in further detail [12].

II. Querying Network State

Many SDN programs react to changes in network state, such as topology changes, link failures, traffic load, or the arrival of particular packets at specific switches. To monitor traffic, the controller can poll the counters associated with the rules installed on switches, which maintain a counter for every forwarding rule that keeps track of the number of packets and bytes processed using that rule. However, programmers must ensure that the rules installed on switches are fine-grained enough to collect the desired traffic statistics. For example, 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 rules is tedious and anti-modular—rules installed by one module may be too coarse to be executed side-by-side with rules installed by a different module.

Frenetic's query language allows programmers to express *what* they want to monitor, leaving the details of *how* to actually collect the necessary traffic statistics to the run-time system. This not only makes it easy for programmers to specify a single query, if that is all they need, but also allows them to write many different queries without worrying about their interactions—the run-time system selects rules at the appropriate granularity to satisfy all of the queries registered with the system.

A. Query Language Design Considerations

Frenetic's query language allows programmers to control the information they receive using a collection of high-level operators for classifying, filtering, transforming, and aggregating the stream of packets traversing the network.

High-level predicates: Many monitor applications classify traffic using packet headers. For example, suppose the programmer wants to tally all web server traffic excluding the host with IP source address `1.2.3.4`. To represent the negation in a switch flow table, we would need to use two rules—a high-priority rule matching packets from `1.2.3.4` with TCP source port `80`, and a lower-priority rule matching all remaining traffic with TCP source port `80`. Frenetic allows programmers to specify predicates like "`srcip!=1.2.3.4 & srcport=80`," leaving the details of how to construct and optimize switch-level rules to the run-time system. In general, Frenetic programmers can specify sets of packets from primitive predicates over standard OpenFlow headers (like `srcip`, `dstip`, `vlan`, *etc.*), their location in the network (switch and ingress port), and ordinary set-theoretic operators (union, intersection, difference, complement, *etc.*).

Dynamic unfolding: Switches have limited space for rules, which can make it difficult to install all the necessary rules in advance. For example, suppose a programmer wants to collect a histogram of traffic by source IP address. Rather than installing rules for each of the 2^{32} possible IP addresses, a typical SDN application would install rules reactively as traffic arrives from different sources. In Frenetic, the programmer can register a query that uses operators such as "Select(bytes)" and "GroupBy([srcip])," and the run-time system dynamically generates the appropriate rules. To do this, it initially sends all traffic to the controller. Upon receiving the first packet from a specific source IP address, the run-time system generates and installs a rule matching future traffic from that host. After receiving a packet from another source IP address, the run-time system generates and installs a second rule. These rules process future traffic from those hosts using efficient hardware, and the counters maintain the necessary information needed to implement the query.

Limiting traffic: A common idiom in SDN programming is to send the first packet of a traffic aggregate to the controller, and reactively install rules for handling future such packets. However, since the controller and the switch do not communicate instantaneously, multiple packets may arrive at the controller before the rules are installed. Rather than force programmers to handle these unexpected packets, Frenetic allows a query to specify the number of packets it wants to see using operators such as "Limit(1)". The run-time system automatically handles any extra packets by applying the forwarding policy registered by the application.

Polling and combining statistics: Many programs need to receive periodic information about traffic statistics. Rather than requiring the programs to manually poll switch-level counters, and register callbacks to query those counters again in the future, Frenetic queries can specify a query interval using operators such as "Every(60)." The run-time system automatically queries the traffic counters periodically and aggregates the resulting values, returning a stream of results to the application.

B. Example Frenetic Queries

To illustrate how Frenetic supports querying of network state, we present two simple examples. These example use

a syntax that closely resembles SQL, including constructs for selecting, filtering, splitting, and aggregating the streams of packets flowing through the network.

MAC learning: An Ethernet switch performs MAC learning to identify what interface to use to reach a host. MAC learning can be expressed in Frenetic as follows:

```
Select(packets) *
GroupBy([srcmac]) *
SplitWhen([inport]) *
Limit(1)
```

The `Select(packets)` clause states that the program needs to receive actual packets (as opposed to traffic statistics). The `GroupBy([srcmac])` subdivides the set of queried packets into subsets based on the `srcmac` header field, resulting in one subset for all packets with the same source MAC address. The `SplitWhen([inport])` clause, like a `GroupBy`, subdivides the set of selected packets into subsets; however, whereas `GroupBy` produces *one* subset of *all* packets with particular values for the given header fields, `SplitWhen([inport])` does not—it generates a new subset each time the header values change (*e.g.*, when the `inport` changes). Together, the `GroupBy([srcmac])` and `SplitWhen([inport])` clauses state that the program wants to receive a packet only when a source MAC address appears at a new ingress port on the switch. The `Limit(1)` clause says that the program only wants to receive the first such packet, rather than all packets from that source MAC address at the new input port. The result is a stream of packets that the program can use to update a table mapping each MAC address to the appropriate ingress port.

Traffic histogram: As another example, consider the following query, which measures the traffic volume by destination IP address on a particular link:

```
Select(bytes) *
Where(inport=2 & srcport=80) *
GroupBy([srcip]) *
Every(60)
```

The `Select(bytes)` clause states that the program wants to receive the total number of bytes of traffic, rather than the packets themselves. The `Where(inport=2 & srcport=80)` clause restricts the query to Web traffic arriving on ingress port `2` on the switch. The `GroupBy([srcip])` states that the program wants to aggregate traffic based on the source IP address. The `Every(60)` says that the traffic counts should be collected every 60 seconds. The result is a stream of traffic statistics that the program can use as input to any other control logic.

Frenetic's query language is expressive, but also gives the programmer a reasonable sense of the cost of evaluating a query. For example, the MAC-learning query sends a packet to the controller whenever a host appears at a new input port, whereas the traffic-measurement query sends a packet to the controller to generate a forwarding rule, and subsequently polls the counter associated with that rule once per minute. The run-time system takes care of the details of installing rules in the switches. For the MAC-learning query, the run-time system initially directs all packets to the controller, but gradually installs packet-forwarding rules (as specified by the controller program) after the first packet of a new MAC address has been seen. If additional packets arrive at the controller—say, because of delays in installing these rules—the run-time system handles these packets automatically, rather than exposing them to the controller program. Similarly, for the traffic-monitoring query, the run-time system installs rules that match on destination IP addresses, to ensure the switches maintain separate counters for different IP addresses.

III. Composing Network Policies

Most networks perform multiple tasks, such as routing, monitoring, and access control. Ideally, programmers would be able to implement these tasks independently, using separate modules. But the programming interfaces available today make this difficult, since packet-handling rules installed by one module often interfere with overlapping rules installed by another module. Frenetic's policy language has a number of features that are designed to make it easy to construct and combine policies in a modular way.

A. Creating Modular Programs

As an example, consider a simple program that combines repeater functionality (*i.e.*, code for forwarding packets that arrive in one interface out the other) with web-traffic monitoring functionality. Abstractly, these tasks are completely orthogonal, so we should be able to implement them as independent modules and combine them into a program that provides both pieces of functionality.

Suppose that the repeater is implemented by a module that generates rules that match all traffic coming in on ingress port `1` and that forward it to output `2`, and vice versa. The monitoring component is implemented by a rule that matches all traffic with TCP source port `80` arriving on ingress port `2`. The monitoring component does not care how packets matching the rule are forwarded—it only needs to access the counters associated with the rule.

Now, consider a Python program, roughly following the NOX controller API, that combines these two components.

```
def switch_join(s):
  pat1 = {inport:1}
  pat2web = {inport:2, srcport:80}
  pat2 = {inport:2}
  install(s, pat1, DEFAULT, [fwd(2)])
  install(s, pat2web, HIGH, [fwd(1)])
  install(s, pat2, DEFAULT, [fwd(1)])
  query_stats(s, pat2web)

def stats_in(s, xid, pat, pkts, bytes):
  print bytes
  sleep(30)
  query_stats(s, pat)
```

When a switch joins the network, the controller invokes the `switch_join` event handler, which installs three rules to handle (i) traffic arriving on ingress port `1`, (ii) web traffic

arriving on ingress port 2, and (iii) non-web traffic arriving on ingress port 2. The second rule has HIGH priority, so the web traffic matches this rule rather than the lower-priority third rule; this ensures the switch correctly collects traffic statistics for web traffic. Note that, in isolation, the pat2 pattern would match *all* incoming traffic, including the web traffic. But the presence of the higher-priority pat2web rule ensures that only non-web traffic matches the pat2 pattern. Having non-web traffic "fall through" to this lower-priority rule is a much more concise way to represent the forwarding policy than (say) having separate rules for every possible TCP source port.

The call to query_stats generates a request for the counters associated with the pat2web rule. When the controller receives the reply, it invokes the stats_in handler. This function prints the statistics polled on the previous iteration of the loop, waits 30 seconds, and then issues a request to the switch for statistics matching the same rule.

The interesting aspect of this program is that it is a "mash-up" of the logic for the repeater and the monitor. The first and third rules come from the repeater program, and the second rule and the stats_in handler from the monitoring program. The second rule needs HIGH priority (to ensure the correct functioning of the monitoring logic) and actions [fwd(1)] (to ensure the correct functioning of the repeater logic). Ideally we would be able to tease apart the code for the monitor and repeater and place each in a separate module. This would allow the monitor to be reused with many different forwarding policies, and the repeater to be reused with many different monitoring queries. Unfortunately, doing this separation is impossible in NOX and other similar controller platforms—both the monitoring specification and forwarding specification are too tightly-coupled to the OpenFlow interface for manipulating low-level rules to be separated or abstracted out. Consequently, even for this simple example, the logic becomes quite complicated, as the programmer must think about multiple tasks (and their interactions) at the same time. In more sophisticated examples, the programmer would need to use multiple levels of priorities, and perform even more complex combinations of policies.

By contrast, Frenetic's policy language makes it easy for programmers to write and compose independent modules. Here is the same program implemented in Frenetic:

```
def repeater():
  rules=[Rule(inport:1, [fwd(2)]),
         Rule(inport:2, [fwd(1)])]
  register(rules)

def web_monitor():
  q = (Select(bytes) *
       Where(inport=2 & srcport=80) *
       Every(30))
  q >> Print()

def main():
  repeater()
  monitor()
```

In the code above, the repeater function implements the repeater and *only* the repeater. The command to register the forwarding policy generated by this code passes it to the run-time system for processing. Likewise, the web_monitor function implements the monitor and *only* the monitor. It defines a query q and then pipes the results of that query into a print function (the >> operator pipes a stream generated from one component into another). The main function assembles these components into a single program. It could easily swap out the given network monitor for another one, without touching repeater code, or change the forwarding logic in the repeater without touching the monitor. Importantly, the responsibility for installing specific OpenFlow rules that realize both components simultaneously is delegated to the run-time system. For this example, the run-time system would generate the same rules as the manually-constructed rules in the switch_join function listed above.

Combining the repeater and the monitor is an example of *parallel* composition, where conceptually both modules act on the same stream of packets. The repeater module applies a forwarding policy (*i.e.*, "writes" network state) and the monitoring module queries the traffic (*i.e.*, "reads" network state). If multiple modules apply a forwarding policy, parallel composition essentially performs a "union" of the actions in the two policies. If one module forwards a packet out port 1, and the other forwards the same packet out port 2, parallel composition would result in an application that forwards the packet out *both* ports. While desirable in some settings, parallel composition is not the only way to combine modules together. In particular, for policies that express negative constraints, such as a firewall, we definitely do not want to take the "union" of all policies—the whole point of a firewall is to drop packets, no matter what other policies are installed in the system! To handle such negative constraints, Frenetic's policy language also includes a restriction operator that allows a programmer to filter policies using packet predicates. Recently we have also added support for *sequential* composition where, conceptually, one module acts on the packets output by the other module. For example, if the first module modifies the packet (*e.g.*, a load-balancing module that modifies the destination IP address to identify a specific back-end server), the second module matches on the modified header fields (*e.g.*, a routing module that forwards packets toward that server based on the new destination IP address).

B. Efficient Run-Time System

The run-time system ensures that each module runs correctly, independent of the other modules. To understand how the run-time system performs composition, consider our initial Frenetic run-time system [13], which had a reactive, microflow-based[1] strategy for installing rules on switches. At

[1]A *microflow* rule is a rule in which all header fields of a packet are specified exactly. Microflow rules are the easiest to compile because every pair of microflow rules are either identical or disjoint. In contrast, *wildcard* rules match large groups of packets, such as all packets with source IP address matching the prefix 1.*. When one uses wildcards over multiple packet headers, rules may partially overlap, meaning one must manage rule priorities and rule shadowing carefully, and the compilation problem becomes more challenging.

the start of execution, the flow table of each switch is empty, so every packet is sent to the controller and passed to the `packet_in` handler. Upon receiving a packet, the run-time system iterates through all of the queries, and then traverses all of the registered forwarding policies to collect a list of actions for that switch. It then processes the packet in one of two ways: (i) if no queries depend on receiving future packets of this sort, it installs a forwarding rule that applies the actions to packets with the same header fields or (ii) if some queries do depend on receiving future packets of this sort, it applies the actions to the current packet, but does not install a rule, since that would prevent those packets from reaching the controller.

In effect, this strategy dynamically unfolds the policy expressed in the high-level rules into switch-level rules, moving processing off the controller and onto the switches, without interfering with any queries. While this reactive, microflow strategy is relatively simple to understand, sending packets to the controller is expensive—*e.g.*, the developers of DevoFlow measured a total round-trip latency of 2.5ms on a HP ProCurve 5406zl switch [14]. Consequently, the current Frenetic run-time system is *proactive* (generating rules before packets arrive at the switches) and uses *wildcard rules* (matching on larger traffic aggregates) [15]. It uses an intermediate language, called *NetCore*, for expressing packet-forwarding policies and a compiler that proactively generates as many OpenFlow-level rules for as many switches as possible, but where impossible (or intractable), uses an algorithm called *reactive specialization* to dynamically unfold switch-level rules on demand.

There are three main situations where the NetCore compiler cannot proactively generate all the rules it needs to implement a policy: (i) the policy involves a query that groups by IP address (or other header field)—such a query would require one rule for each of the 2^{32} IP addresses if generated ahead of time, but only one rule for each IP address that actually appears in the network if unfolded dynamically; (ii) the policy involves a function that cannot be implemented natively or efficiently on the switch hardware, and (iii) the switch does not have space for additional wildcard rules. In these cases, the compiler can fall back to the microflow-based strategy, and use the plentiful exact-match rules available on switches. In other cases, the NetCore compiler generates policies completely proactivity and no packets are diverted to the controller.

As an example of the second situation, consider a predicate that matches all destination IP addresses with a first octet of `90`, a third octet of `70`, and a fourth octet of `60`. Most of today's OpenFlow switches do not support arbitrary wildcards in IP addresses, and instead match only on an address *prefix* like `90.80.*.*`. To handle arbitrary predicates, the run-time system generates an *overapproximation* that matches a superset of the traffic specified by the original predicate (e.g., `90.*.*.*`), and installs rules that direct such traffic to the controller for further processing. For example, a packet with destination address `90.80.70.60` would go to the controller, causing the run-time system to *reactively specialize* the list of rules in the switch by installing a high-priority rule matching destination address `90.80.70.60`. This ensures that the switch handles all future packets with this destination address, while still directing packets with other destination IP addresses in `90.*.*.*` to the controller.

The run-time system's ability to generate overapproximations, and reactively refine the rules based on the actual traffic in the network, allows Frenetic to support policies with arbitrary functions that the switches cannot implement. The run-time system can also customize rules to the capabilities of the switch (e.g., whether the switch supports prefix patterns vs. arbitrary wildcards). This makes NetCore programs more portable. For the predicate above, the run-time system would generate a single rule when the underlying switch can support arbitrary wildcard patterns, or the overapproximation `90.*.*.*` with reactive specialization if the switch can only support prefix patterns, or the overapproximation `*.*.*.*` if the switch could not even support prefixes.

Currently, the Frenetic run-time system supports OpenFlow 1.0 (*i.e.*, a single table in each switch). Composing multiple modules can easily lead to a multiplicative "blow-up" in the number of rules, particularly if the modules act on different packet-header fields (*e.g.*, a monitoring module that matches on TCP port numbers combined with a forwarding module that matches on destination IP addresses). This scalability problem is inherent to more sophisticated programs that combine multiple network-management tasks, whether the composition is performed "manually" by the programmer or automatically by a run-time system like ours. If anything, a smart run-time system should do a better job in applying optimizations that minimize the number of rules required to represent a policy. Ultimately, the "rule blow-up" problem is best addressed by having more sophisticated data-plane architectures, such as a pipeline of tables. The newer versions of the OpenFlow standard provide an interface to the tables available in modern switch hardware. In our future work, we plan to extend our run-time system to capitalize on these tables to represent sophisticated policies in a more compact fashion.

IV. Consistent Updates

Programs often need to transition from one policy to another—*e.g.*, due to topology changes, changes in network load or application behavior, planned maintenance, or unexpected failures. From the perspective of the programmer, it would be ideal if such transitions could be initiated via a single command that simply declares the new, global network configuration desired. Moreover, to avoid anomalies such as transient outages, forwarding loops, and security breaches, every transition must be implemented gracefully: all application-specific connectivity invariants should be preserved during migration from old to new policy.

We have designed high-level network update operations that implement configuration changes while guaranteeing that traffic will be processed consistently during the transition [16], [17]. The semantics of these update operations provides useful guarantees about network behavior during transitions, and yet are relaxed enough to admit practical implementations.

A. Per-Packet Consistent Updates

The primary update abstraction supported in Frenetic is a *per-packet consistent update*. A per-packet consistent update

guarantees that every packet flowing through the network is processed with exactly one forwarding policy. For example, if a per-packet consistent update transitions the network from policy A to policy B, it guarantees that every packet traversing the network is processed using the rules from A on all switches or the rules from B on all switches, but never a mixture of the two. A crucial consequence of this design is that if both A and B satisfy a *trace property—i.e.*, a property of the paths that packets take through the network—then all packets traversing the network either before, during, or after the transition will be guaranteed to obey that property. For example, if both A and B have no loops then no packet will encounter a loop. If both A and B filter packets from source IP address `1.2.3.4` then all such packets will be dropped. Generally speaking, trace properties encompass access control and connectivity properties, but do not encompass properties that concern the relationship between multiple packets, such as in-order delivery or congestion.

The semantics of per-packet consistent update helps developers write reliable dynamic network applications because such developers can ensure trace properties persist as network policy evolves—before any change in policy the developer need merely check the trace property holds of that next policy to be installed. This insight also makes it possible to build verification tools that automatically check trace properties as configurations evolve. For example, we have developed a tool that allows programmers to specify properties of networks such as loop freedom or access control using logical formulas. Using an off-the-shelf verification tool, we can check these properties against static NetCore policies. To verify *dynamic* policies, defined as a stream of static NetCore policies, we simply verify each individual policy in the stream and use per-packet consistent updates to manage the transition from one policy to the next. The per-packet consistent update ensures there are no unusual transient states that violate the properties of interest.

To implement per-packet consistency, we use a mechanism called *two-phase update* that stamps packets with a version at the ingress to the network and tests for the version number at all internal ports in the network. This mechanism can be implemented in OpenFlow using a header field to encode version numbers (e.g., VLAN tags or MPLS labels). To update to a new configuration, the controller first pre-processes the rules in the new configuration, adding an action to stamp packets at the ingress and test for the next version number elsewhere. Next, it installs the rules for internal ports, leaving the rules for the old configuration (whose rules match the previous version number) in place. At this point, every (internal) switch can process packets with either the old or new policy, depending on the version number on the packet. The controller then starts updating the rules for ports at network ingresses, replacing their old rules with new rules that stamp incoming packets with the new version number. Because the ingress switches cannot all be updated atomically, some packets entering the network are processed with the old policy and some packets are processed with the new policy for a time, but any individual packet is handled by a single policy. Finally, once all packets following the "old" policy have left the network, the controller deletes the old configuration rules from all switches, completing the update.

This *two-phase update* mechanism works in any situation, but it is not always necessary. In practice, many optimizations are possible when the new policy and old policy are similar [17]. If the policy changes affect only a portion of the network topology, or a portion of the traffic, the run-time system can perform the update on a subset of the switches and rules. If the new policy is a simple extension (e.g., adding policy for handling some portion of traffic) or retraction (e.g., removing policy for handling some portion of traffic), the updates become even simpler. Our prototype implementation applies these optimizations, often resulting in much more efficient mechanisms.

B. Per-Flow Consistency

Per-packet consistency, while simple and powerful, is not always enough. Some applications require that streams of related packets be handled consistently. For example, a server load-balancer needs all packets from the same TCP session to reach the same server replica, to avoid breaking connections. A *per-flow consistent update* ensures that streams of related packets are processed with the same policy—*i.e.*, all packets in the same flow are handled by the same configuration. Formally, a per-flow update preserves all trace properties, just like per-packet consistency. In addition, it preserves properties that can be expressed in terms of the paths traversed by sets of packets belonging to the same flow.

Implementing per-flow consistent updates is more complicated than per-packet consistency because the system must identify packets that belong to active flows. A simple mechanism can be obtained by combining versioning with rule timeouts [7]. The run-time system can pre-install the new configuration on internal switches, leaving the old version in place, as in per-packet consistency. Then, on ingress switches, the controller sets soft timeouts on the rules for the old configuration and installs the new configuration at lower priority. When all flows matching a rule complete, the rule expires and the rules for the new configuration take effect.

Note that if several flows match a rule, the rule may be artificially kept alive even though the "old" flows have completed—if the rules are too coarse, then they may never die! To ensure rules expire in a timely fashion, the controller can refine the old rules to cover a progressively smaller portion of the flow space. However, "finer" rules require more rules, a potentially scarce commodity. Managing the rules and dynamically refining them over time can be a complex bookkeeping task, especially if the network undergoes a subsequent configuration change before the previous one completes. However, this task can be implemented and optimized once in a run-time system, and leveraged over and over again in different applications.

V. Conclusion

The Frenetic language offers programmers a collection of powerful abstractions for writing controller programs for software-defined networks. A compiler and run-time system

implements these abstractions and ensures that programs written against them execute efficiently. Our work focuses on the three stages of managing a network—monitoring network state, computing new policies, and reconfiguring the network. Yet, these abstractions are just the beginning. We are currently exploring further ways to raise the level of abstraction for programming the network, including techniques for "slicing" the network [18] to provide isolation between multiple programs controlling different portions of the traffic, for monitoring and managing end hosts, for handling failures, and for virtualizing topologies. On the latter topic, we plan to support program modules that operate on different "views" of the network (*e.g.*, a load balancer that sees the network as a single switch, combined with routing that considers the full details of the underlying switches and links). We believe that these and other abstractions will continue to lower the barrier for creating new and exciting applications on software defined networks.

Acknowledgments: Our work is supported in part by ONR grants N00014-09-1-0770 and N00014-09-1-0652, NSF grants CNS-1111698 and CNS-1111520, TRUST, and gifts from Dell, Intel, and Google. Any opinions, findings, and recommendations are those of the authors and do not necessarily reflect the views of the ONR or NSF.

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[2] 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 *USENIX Symposium on Operating Systems Design and Implementation*, pp. 351–364, Oct. 2010.

[3] S. Shenker, M. Casado, T. Koponen, and N. McKeown, "The future of networking and the past of protocols," Oct. 2011. Invited talk at Open Networking Summit.

[4] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2010.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Transactions on Networking*, vol. 17, Aug. 2009.

[6] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using OpenFlow," Aug. 2009. Demo at *ACM SIGCOMM*.

[7] R. Wang, D. Butnariu, and J. 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*, Mar. 2011.

[8] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM CCR*, vol. 38, no. 3, 2008.

[9] "Beacon: A java-based OpenFlow control platform." See `http://www.beaconcontroller.net`, Dec. 2012.

[10] "Floodlight OpenFlow Controller." `http://floodlight.openflowhub.org/`.

[11] "Frenetic and NetCore compilers." `https://github.com/frenetic-lang/netcore`, Aug. 2012.

[12] "The Frenetic project." `http://www.frenetic-lang.org/`, Sept. 2012.

[13] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sept. 2011.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, pp. 254–265, Aug. 2011.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 2012.

[16] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!," in *ACM SIGCOMM HotNets Workshop*, Nov. 2011.

[17] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, pp. 323–334, Aug. 2012.

[18] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, Aug. 2012.