Logic Programming for Software-Defined Networks

Naga Praveen Katta
Princeton University

Jennifer Rexford
Princeton University

David Walker
Princeton University

1. Introduction

In the past, most networks were built out of a collection of special-purpose devices running distributed algorithms that process topology information, define routing and access control policies, perform traffic monitoring and execute other services. These networks were usually managed through a set of complex, low-level, and heterogeneous interfaces that allowed users to configure separately firewalls, network address translators, load balancers, routers and switches. Overall, such an approach to network configuration involved managing thousands of lines of brittle, low-level code in different domain-specific languages that expressed *how* various complex routing mechanisms should work as opposed to *what* high-level policy should be implemented. It was a remarkably complex and error-prone task.

Recently [4, 10], however, we have seen the emergence of a new kind of network architecture, referred to as *software-defined networking* (SDN) that has the potential to do away with the plethora of special purpose network devices and the low-level interfaces we find in conventional networks. In a software-defined network, a logically centralized *controller* machine manages a distributed set of *switches*. The controller is a general purpose machine capable of performing arbitrary computations, such as those computations that are necessary to infer network topology and make routing decisions. When the controller decides on a routing policy, it implements that policy by instructing the switches to install the necessary *packet-forwarding rules*. Each such packet-forwarding rule includes a *predicate*, an *action* and a *priority*. Predicates match certain packets based on the values in the packet header fields (*e.g.*, the packet's source IP address, destination IP address, source MAC address, *etc.*). If two different rules match a packet, the rule with the higher priority is triggered. When a rule is triggered, its *action* takes effect. Typical actions include the action to drop the packet, to forward the packet out one of the ports on the switch, to flood the packet out all ports on the switch, to rewrite one or more header fields before forwarding, or to forward the packet to the controller for further, more general processing or analysis.

In addition to forwarding packets, each switch records certain statistics. Specifically, it records the number of packets that match each rule and the number of bytes processed by each rule. The controller may request such statistics to help it make forwarding decisions.

Today's most common controller programming platforms such as NOX [6] and Beacon [1] provide a programming interface that supports a low-level, imperative, event-driven model in which programs react to network events (like packet arrival, link status update *etc.*) by explicitly installing and uninstalling individual, low-level packet processing rules rule-by-rule and switch-by-switch. In such a situation, programmers must constantly consider whether (un)installing switch policies will effect other/future events monitored by the controller, and must explicitly coordinate multiple asynchronous events at the switches, to perform even simple tasks. One would prefer to be able to specify the current forwarding policy at a high level of abstraction and have a compiler and run-time system manage the tedious details of installing individual switch-level rules, automatically.

One effort in this direction is FML [7], which is a high-level, domain-specific SDN programming language based on datalog. It comes equipped with a set of very high-level built-in policy operators that allow/deny certain flows, waypoint flows through a firewall or provide quality of service. When the network forwarding policy falls in to the space of policies that can be described by an FML program, the code for implementing the policy is extremely compact and elegant. Unfortunately, however, while new policy operators can be added, the additions come by coding outside the FML language itself in C++ or some other language. Moreover, the granularity at which FML operates is a unidirectional network flow (*i.e.*, set of related packets). This means that a resulting policy decision applies equally to all packets within the same flow — it is not possible to move or redirect a flow as it is processed. Consequently, while FML provides network operators with a very useful set of SDN abstractions, the programming model is somewhat inflexible.

Frenetic [5] proposes a different kind of language design built around a combination of (1) a declarative query language with an SQL-like syntax, (2) a functional stream-processing language, and (3) a specification language for describing packet forwarding. The query language provides a declarative means for monitoring network state. The functional component provides a flexible way to manage, combine, process and manipulate the data generated by network measurement or other sources. The third component allows the programmer to generate arbitrary packet-forwarding policies at a high level of abstraction. The Frenetic run-time system [11] removes the burden of considering the interactions between packet-monitoring policies generated by queries and the packet forwarding policies generated by the third component.

In this paper, we present the preliminary design of a new language, called Flog that combines ideas found in both FML and in Frenetic. First, from FML, we adopt the idea of using logic programming as the central paradigm for controlling software-defined networks. Logic programming appears to be a good fit for this domain because of the success of FML and because so much of SDN programming is table-driven collection and processing of network statistics. We were also inspired by the success of recent work on construction of distributed applications, network protocols and overlay networks using logic programming languages like NDlog [9], Overlog [8], Dedalus [3] and Bloom [2]. Each of these systems make it possible to construct powerful distributed applications in very few lines of code. Our work differs from these latter efforts primarily due to the fact that we have specialized the semantics and implementation of our language for use in the context of software-defined networks.

Second, from Frenetic, we adopt the idea that typical controller programs may be factored in to three key components: (1) a mechanism for querying network state, (2) a mechanism for processing data gleaned from queries and other sources and (3) a component for generating packet-forwarding policies that our underlying run time system can automatically push out to the network of switches for us.

In the following sections, we describe our preliminary work on the development of Flog.

2. Flog: An SDN Logic Programming Language

Flog is designed as an event-driven, forward-chaining logic programming language. Intuitively, each time a network event occurs, the logic program executes. Execution of the logic program has two effects: (1) it generates a packet-forwarding policy that is subsequently compiled and deployed on switches, and (2) it generates some state (a set of relations) that is used to help drive the logic program when the next network event is processed. The following paragraphs describe the major components of any Flog program.

Network Events. Controllers for software-defined networks process many different kinds of events: switches come online and go offline; ports on switches become active or inactive; packets arrive at the controller and require handling; statistics gathered by switches arrive and require processing. We believe our basic infrastructure should be able to accomodate any of these network events, but this paper, we consider just one type of network event: packet arrival at the controller.

In order to define the kinds of packets that a particular application is interested in (and hence should be forwarded to the controller), the programmer defines a *flow identification rule*. Such rules have the following form.

```
flow(h1=X1,h2=X2,...), constraints --> rel(X1,X2,...).
```

The keyword `flow` identifies the fact that this is a flow identification rule. Each of `h1`, `h2`, *etc.*. are the names of particular packet fields such as `srcip` (the source IP address), `dstip` (the destination IP address), `vlan` (the VLAN tag), *etc.* The capitalized `X1`, `X2`, *etc.* are (user-defined) logic variables. The effect of such a rule is to generate the tuple `rel(X1,X2,...)` for every packet detected in the network with a unique set of values (`X1,X2,...`) in the given fields such that the constraints hold. For example, this rule:

```
flow(srcip=IP,vlan=V), V > 0 --> myvlans(IP,V).
```

will generate a network event every time a packet with a *new* srcip-vlan tag pair is detected in the network, provided that the vlan tag is greater than 0. When such a network event generated, the rest of the logic program will be executed. The initial data for the logic program will include the tuple `myvlans(IP,V)`.

Note that if two successive packets, both with identical srcip and vlan tags are detected, the logic program will only run once – the first time. That first run should generate a forwarding policy capable of handling successive packets of the same kind. This semantics is similar to the semantics adopted by Frenetic's `Limit(1)` policies [5]. This default strategy ensures that at most one packet per flow must go to the controller – an important factor as packets that are processed at the controller suffer orders of magnitude more latency than packets processed in hardware on switches. One way to override this default strategy is to use the special `split(field)` constraint in a flow identification rule. The `split(field)` constraint declares that a flow is over (*i.e.*, it is *split* in half) and that additional packets can come to the controller whenever the value in the specified `field` changes. This idea is also adopted directly from Frenetic. We will illustrate the use of `split` more concretely in the next section.

Information Processing. After identifying network events, a Flog programmer writes a logic program to process the facts generated by such events. A typical logic program has multiple *logic inference rules* and any one of these rules may be *triggered* at any point in time to infer new facts from existing facts. Tthis process continues until no new fact can be derived, in which case this execution halts. In Flog, similar to Dedalus [3], there are two kinds of logic programming rules that may be used for information processing. The first kind of rule, written

```
fact1, fact2, ... --> factn
```

is a very standard datalog-like rule. When `fact1`, `fact2`, ..., match facts in the current database, `factn` is generated and added to the current database. The second kind of rule is written as follows.

```
fact1, fact2, ... +-> factn
```

Like the first rule, when `fact1`, `fact2`, ..., match facts in the current database, `factn` is generated and added to the current database. However, in addition, `factn` is copied in to a new database, which will be used the next time the logic program is executed on the subsequent network event.[1] It is through this second sort of rule that state persists from one iteration to the next. All facts not explicitly saved through this mechanism are deleted when the processing required for the current network event is complete.

Policy Generation. The final component of a Flog program involves generating a routing policy for network switches. To specify the switch policy, we use the following syntax.

```
h1(F1), h2(F2), ... |> action, level(i)
```

The constraints on the left of the `|>` specify the kinds of packets that match the packet-forwarding rule. They do so by specifying the packet fields (and switch and ports) that match the rule. The `action` on the right of the `|>` specifies where to forward or flood the packets or how to modify them. The `level` specifies the priority of the rule.

3. Examples

Now that we have outlined the main components of a Flog program, we illustrate its use through a couple of simple example programs.

3.1 Stateful Firewall

A stateful firewall is a common device used to help protect a private corporate domain from malicious outsiders. The key idea is that the corporate domain is able to send any traffic it chooses to the outside world and an entity in the outside world is only allowed to send traffic in to the coporate domain if the corporate domain has first sent it a packet. To model this situation, we assume the network we wish to control contains just one switch with two ports. The external world is connected to port 1; the internal corporate domain is connected to port 2. By default, any packet that arrives on port 2 is routed across the switch and out port 1. In addition, the stateful firewall remembers the destination IP of such a packet. Any packet that arrives on port 1 is dropped unless the stateful firewall sees that the source IP of the packet matches one of the IPs that it has remembered. In this latter case, it forwards the packet across the switch and out port 2.

The Flog program in Figure 1 shows how to implement this application in a few lines of code. The first line declares that each packet with a unique destination IP address arriving at port 2 should be sent to the controller and be processed by the language run-time system. When such a packet is processed, the run-time system extracts the destination IP address `IP` from the packet and stores it in the `seen` relation.

[1] Sometimes, it is convenient to save a fact away for future use, but one does not want it to be used in the current derivation. If that is the case, one may use a slightly different operation written `++>`. Rules written with `+->` are actually syntactic sugar for a pair of a standard rule `-->` and a "next time step" rule `++>`.

```
# Network Events
flow(dstip=IP), inport=2 --> seen(IP).

# Information Processing
seen(IP) +-> allow(IP).
allow(IP) +-> allow(IP).

# Policy Generation
inport(2) |> fwd(1), level(0).

allow(IP) -->
  srcip(IP), inport(1) |> fwd(2), level(0).
```

Figure 1. Example application: Stateful Firewall

The next segment of the program describes the information processing stage. This stage is responsible for saving away all `IP` addresses that have been seen. It does so by using the temporal `+->` logic rules to save the IP addresses in the unary relation `allow`. All of the elements of the `allow` relation will be used in generating the current packet-forwarding policy and are also saved away for use when processing the next network event.

The last few lines of the program implement the policy generation stage. The first part installs a default packet-processing rule that allows any traffic that comes in on port 2 to be forwarded out on port 1. In addition, for any `IP` stored in the relation `allow`, the second part of this stage generates a specific packet-forwarding rule that states that all traffic appearing on port 1 of switch `SW` whose `srcip` field is `IP` is forwarded across the switch and out port 2.

3.2 Ethernet Learning Switch

An ethernet learning switch dynamically learns the association between hosts and ports as it sees traffic. It floods packets to unknown destinations, but outputs packets to hosts with known locations on the port the host is connected to.

Figure 2 gives the Flog program for a simple learning switch where hosts are not mobile (*i.e.*, they do not change the port on the switch they are connected to). In this case, the program monitors all the packets that arrive at the switch and groups them by source IP and inport, storing this information in the `seen` relation. In the information processing phase, the information is transferred to the persistent `learn` database. The policy generation phase first generates a low priority rule that unconditionally floods all packets that arrive at the switch. Then, based on information it has learned, it generates higher priority rules that perform more precise forwarding. Specifically, for every `learn(IP, P)` fact in the persistent database, the program generates a forwarding rule that directs packets with destination `IP` out port `P`.

An interesting variant of the basic learning switch is presented in Figure 3. In this case, we assume end hosts may be mobile. For example, a host may be a phone or a laptop and may move from one access point to another, thereby causing a change in the port they are connected to. In this case, we need the special `split` keyword that splits the flow whenever a host IP changes its inport `P`. In the information processing phase, we retain learned IP-port associations when they do not conflict with an IP-port association that we have just seen and we generate specific forwarding rules from all learned associations (provided the learned association has not just been overrided by a newly learned association).

Acknowledgements. We would like to thank Rob Simmons for several enlightening discussions and advice on logic programming.

References

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

```
# Network Events
flow(scrip=IP, inport=P) --> seen(IP, P)

# Information Processing
seen(IP, P) +-> learn(IP, P).
learn(IP, P) +-> learn(IP, P).

# Policy Generation
|> flood, level(0).

learn(IP, P) --> dstip(IP) |> fwd(P), level(1).
```

Figure 2. Example : Ethernet learning switch without mobility

```
# Network Events
flow(scrip=IP, inport=P), split(inport) --> seen(IP, P).

# Information Processing
seen(IP, P) +-> learn(IP, P).
seen(IP, P), learn(IP', P'), IP!=IP' +-> learn(IP',P').

# Policy Generation
* |> flood, level(0).

seen(IP, P) -->
  dst(IP) |> fwd(P), level(1).

seen(IP, P), learn(IP', P'), IP!=IP' -->
  dst(IP') |> fwd(P'), level(1).
```

Figure 3. Example: Ethernet learning switch with mobility

[2] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *Proceedings of CIDR '11*.

[3] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[4] 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.

[5] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of ICFP '11*.

[6] 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.

[7] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of WREN '09*.

[8] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proceedings of SOSP, 2005*.

[9] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *Proceedings of SIGCOMM '05*.

[10] 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.

[11] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *Proceedings of POPL '12*.