Flow-level State Transition as a New Switch Primitive for SDN

Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, Ramesh Govindan
University of Southern California

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]; C.2.1 [**Network Architecture and Design**]; C.2.4 [**Distributed Systems**]: Network operating systems

Keywords

Software-defined Network; State Machine

1. INTRODUCTION

Software-defined networking has changed networking by separating the control plane from the data plane. While there have been many innovations on the controller applications for different network management needs, most of these applications still rely on flow-based rules in the data plane. These flow-based rules often match on multiple packet header fields (e.g., source/destination IP addresses), take predefined actions on the matching packets (e.g., dropping the packet, forwarding it to an outgoing port), or maintain counters (e.g., the number of packets or bytes).

The controller saves flow-based rules to the switches in two modes: proactive and reactive. In the *proactive* approach, the controller populates rules in switches ahead of time for all the flows coming to the switch. However, the proactive approach requires a priori knowledge of events at switches, and how to handle these events.

The *reactive* approach supports more dynamic applications, but has poor performance. In the reactive approach, switches often send the events (e.g. the first packet of each flow) to the controller, and the controller installs flow-based rules based on these events. However, this introduces significant overhead (CPU, memory, etc.) at the switch, high performance overhead (i.e., delay and throughput), and scalability problems due to the limited communication channel between the switches and the controller [1]. For example, consider a stateful firewall that denies unsolicited inbound traffic if it cannot find a corresponding outbound flow in established state. The controller becomes aware of the state of outbound flows by receiving the TCP signals (e.g., SYN, FIN) from the switch and denies the unsolicited inbound flows and installs forwarding rules for others upon receiving their first packet. This means that switches have to send *multiple* packets of the same flow to the controller, and the controller has to reactively change the flow-level rules based on these incoming packets.

1) `Task:=(StateMachine,InstanceMapping)`
2) `StateMachine:=({State},{Transition},{Action},Filter)`
3) `State:=(name,{Variable})`
4) `Variable:=(name,#bits)`
5) `Transition:=(StartState,Condition,TargetState,`$\mathcal{F}$`)`
6) `Condition:=`f_1`(StartState.Variables,Packet)`$\rightarrow$`True|False`
7) $\mathcal{F}$`:=`f_2`(State.Variables,Packet)`$\rightarrow$`TargetState.Variables`
8) `Action:=(State,Condition,Instruction,Priority)`
9) `Filter:=`f_3`(Packet)`$\rightarrow$`True|False`
10) `InstanceMapping:=`f_4`(Packet)`$\rightarrow$`Index`

Table 1: FAST abstraction

To reduce the controller's involvement in many dynamic applications, many works recognize the limitations of flow-based rules and have proposed specific optimizations in the data plane. DevoFlow [1] reduces the controller overhead by introducing rule cloning and measurement triggers. OpenFlow 1.3 supports rate limiting by allowing switches to track flow rates and tag/drop excess traffic without the controller involvement. Open vSwitch adopts learn actions for software switches that can install new rules when traffic matches an old one.

Instead of proposing yet another specific optimization, we aim at identifying a new generic data-plane abstraction to replace flow-based rules. We observe that many networking tasks can be expressed as local state machines over a flow or an aggregate of flows. For example, to implement the stateful firewall, the controller may install a state machine on the switch to keep track of TCP states. The associated action on each state can allow/deny inbound traffic based on the TCP state.

We propose FAST (Flow-level State Transitions) as a new switch abstraction. FAST allows the controller to proactively program state transitions, and allows the switches to run *dynamic* actions based on local information. FAST supports a wide range of dynamic applications and can be easily implemented with today's commodity switch components.

2. DESIGN & EVALUATION

FAST includes three parts: (1) an abstraction that allows operators to program their state machines for a variety of applications; (2) a FAST controller that translates state machines to the data plane API and manages the interaction of local state machines with network-wide tasks; (3) a FAST data plane that includes a pipeline of tables to support state machines with commodity switch components. The detail of each component comes next.

Abstraction: Table 1 describes how a task is defined in FAST using state machines. The parentheses represent a tuple, curly brackets show a set, and right arrows define the output of a function. We now highlight the important aspects of the task definition: States may require storing counters that represent many states over the

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.
ACM 978-1-4503-2836-4/14/08.
http://dx.doi.org/10.1145/2619239.2631439.

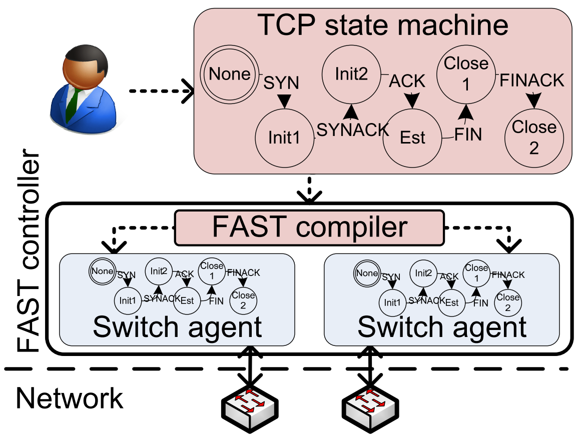
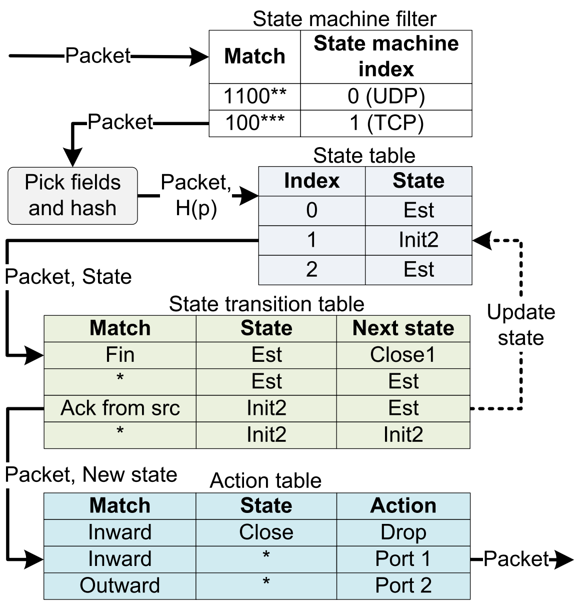


Figure 1: FAST architecture

variable values. Later, the controller can map the state names and their variables to a bit string representing an extended set of states where each counter value makes a new state. Transitions leverage a guard condition which can match on state variables. Actions execute a standard OpenFlow instructions. The programmer can also filter the traffic going through a set of state machines, and finally the abstraction includes a state mapping function over packet headers to map a packet to the right state machine instance.

Controller: The FAST compiler translates user's state machine definitions to the actual code (switch agents) that can run state machines at individual switches (Figure 1). It uses the knowledge of topology and switch constraints to make switch agents specific to the switch capabilities and configures them to install the state machines only on a subset of switches (e.g., ingress). A switch agent has three responsibilities: (1) It converts state machines to the switch API. (2) If the switch does not support a required feature to implement a state machine, the switch agent falls back to the reactive approach and performs part of the state machine by receiving packet-ins. (3) If a network-wide task (e.g., traffic engineering) requires to be informed of a local event (state change), the switch agent programs the switch to send packet-ins to report to the task.

Data-plane: Figure 2 describes FAST data plane on top of the current flow-based multi-table architecture. While this is not the only architecture to support state machines, especially, in the software switches, we believe it is a transition step from the flow-based rules to a state-machine based data plane. We use TCP state machine as a running example to illustrate the components of state machines, which can be implemented with hardware currently available on commodity switches. The design contains four tables. The state machine filter table is shared among multiple types of state machines, but the other three are specific to each state machine definition. The state table keeps track of state machine instances and is implemented as a hash-table. We decouple actions from state transitions for two reasons. First, it can be more compact: a single entry in action table can cover the actions in multiple states. For example, regardless of whether the state machine is in the Established or Init1 state, one can specify the same action (e.g., forward to port 1). More important, however, is that we *need* this decoupling for our problem domain. While there is one instance of a TCP state machine, the action corresponding to a flow being in a specific state may depend on the flow itself (e.g. output port). The switch agent installs a state machine on data plane by adding a rule in the filter table, specifying the hash fields and the initial state for the state table, and installing state transition rules and action rules in the corresponding tables.



State machine filter

Match	State machine index
1100**	0 (UDP)
100***	1 (TCP)

State table

Index	State
0	Est
1	Init2
2	Est

State transition table

Match	State	Next state
Fin	Est	Close1
*	Est	Est
Ack from src	Init2	Est
*	Init2	Init2

Action table

Match	State	Action
Inward	Close	Drop
Inward	*	Port 1
Outward	*	Port 2

Figure 2: Implementing TCP state machine in FAST data plane

Approach	Mean	5^{th}	95^{th}
Proactive	1.85	1.45	3.68
Reactive	84.8	57.84	109.7
FAST	3.02	1.34	5.93

Table 2: Comparing flow completion delay (ms) for 100 experiments

We have implemented a FAST prototype with POX and Open vSwitch. At the controller side, a switch agent proactively installs state machines using OpenFlow protocol. We have instantiated the state machines and changed their states leveraging the Open vSwitch learn action. We use TCP state machine as an example. The state machine transits on SYN, SYNACK, ACK, FIN, and FINACK flags; thus we added the matching on TCP signals in Open vSwitch and OpenFlow commands. Table 2 compares the delay of going through the TCP state machine for a flow as the time between initiating a connection with only one data packet until terminating it. FAST has much smaller delay comparing to the reactive approach because there is no need to send the signals to the controller. However, due to the overhead of calculating the hashes and updating the state table, its delay is larger than simple proactive approach. Note that the proactive approach cannot track TCP state machine. We also compared the throughput of an iperf connection in FAST vs. proactive approach. As the iperf connection remains in the Established state, this shows the overhead of only hashing and checking the current state. The throughput of FAST is very close to proactive approach (7.8 Gbps vs 8.2 Gbps, respectively with standard deviation of 0.2Gbps over 30 tries).

To summarize, compared to the primitive of flow-based rules, FAST can support more flexible networking tasks, improves the performance and scalability of the SDN controller, and can be easily implemented with commodity switch components. The future work at the controller side includes partially installing, consistent installing, verifying, and composing state machines. At the data plane, we plan to use richer set of features at switches such as flexibly parsing the packet header and to optimize memory usage of state machines to fit in switches with limited resources.

3. REFERENCES

[1] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.