FAST: A Simple Programming Abstraction for Complex State-Dependent SDN Programming *

Kai Gao
Tsinghua Univeristy
gaok12@mails.tsinghua.edu.cn

Chen Gu
Tongji University
gc19931011jy@gmail.com

Qiao Xiang
Tongji/Yale University
qiao.xiang@cs.yale.edu

Yang Richard Yang
Tongji/Yale University
yry@cs.yale.edu

Jun Bi
Tsinghua University
junbi@tsinghua.edu.cn

ABSTRACT

Handling state dependencies is a major challenge in modern SDN programming, but existing frameworks do not provide sufficient abstractions nor tools to address this challenge. In this paper, we propose a novel, high-level programming abstraction and implement the *Function Automation SysTem (FAST)*. With the two key features, *i.e.*, *automated state dependency tracking* and *efficient re-execution scheduling*, we demonstrate that FAST substantially simplifies state-dependent SDN programming and boosts the performance.

CCS Concepts

•Networks → Programming interfaces; Network control algorithms;

Keywords

SDN, Programming abstraction, State dependency

1. INTRODUCTION

A common characteristic of many network control-plane functions is that their computation depends on network states. For example, basic routing algorithms such as the shortest path depend on the topology, QoS-based routing depends on both topology and the current resource allocations, and security functions (*e.g.*, access control, policy-based forwarding) depend heavily on the current state of configured security policies.

Implementing aforementioned control-plane functions in a correct and efficient manner, however, can be complex. Existing frameworks such as OpenDaylight and ONOS recognize this complexity and introduce datastores and broker services with a powerful pub/sub API to enable state dependent programming. However, many programming complexities remain.

Firstly, it is still the responsibility of the programmers to handle the complexity of identifying dependent data and subscribing to their changes. This, however, is not trivial. For example, a production implementation of the Dijkstra algorithm using a priority queue[2] touches only a subset of links, depending on the specific source and destination locations. Missing a subscription to a touched link can lead to *inconsistency* between the calculated path and the current network topology. On the other hand, naive approaches to simplify the programming by oversubscribing to changes on all links in the whole topology, lead to *unnecessary* re-executions.

Secondly and more importantly, it can be more complex than one might think to handle state changes correctly. Consider a control function f which depends on a certain state variable v_s. When the value of v_s changes, the naive solution, which simply update the outcome by re-executing f using the new value, can lead to errors. Consider the QoS routing example which finds a path and makes bandwidth reservations along the path. Assume a link on the path fails, and the function is to be re-executed to find an alternative. It is important that the previously reserved bandwidth be released first, to avoid "garbage" bandwidth reservations. However, releasing the reservations may trigger other data change events and lead to cascading effects.

We address the preceding complexities by exploring a novel and substantially simpler control-plane program-

* This research is supported by the ***National Science Foundation*** (CNS-1018502, CC*IIE 1440745), the ***National Natual Science Foundation of China*** (No.61472213 and No.61502267) and the ***Google Faculty Research Award***.

SIGCOMM '16, August 22-26, 2016, Florianopolis , Brazil
 ISBN 978-1-4503-4193-6/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2934872.2960424

ming abstraction where state changes are transparent to programmers. The tasks to manage state dependency tracking and schedule re-executions are automated by our runtime system, FAST.

2. PROGRAMMING ABSTRACTION

There are two different views for a programmer in FAST, as demonstrated in Figure 1.

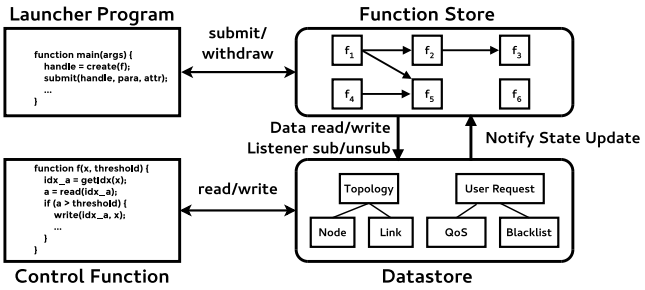


Figure 1: Programmers' view of FAST.

The **datastore** provides the API for **control functions** to access and modify persistent network state. As FAST handles data changes internally, the control function f contains no event driven code and is programmed to execute with the current state.

The **function store** is the key concept of FAST. It is built on top of the datastore and manages the meta information about submitted control functions for better analysis and scheduling. The **launcher programs** can submit certain control functions to the FAST function store and withdraw them when appropriate. They can also configure the *parameters* passed to the control functions and set *attributes* to specify the behaviours of certain control functions. For example, launchers can specify the *precedence* relationship between different control functions to create a workflow, or using *groups* to enforce that all updates in a group be committed as a single transaction.

3. SYSTEM COMPONENTS

FAST function store is driven by a sophisticated runtime, shown in Figure 2, which includes novel algorithms to automate complex tasks of state-dependent programming. Specifically, the runtime system consists of the following key components.

Event dispatcher Classify events such as state changes and submission/withdrawal of control functions, and dispatch them to corresponding components.
Restoration module Restore the system state after certain events so that the current state is consistent with the status of the function instances.
Min-makespan scheduler Schedule execution orders to minimize the maximum sum of control plane computing latency and the control path outbound latency.
Instance executor Execute and monitor the control functions to obtain fine-grained state dependencies.

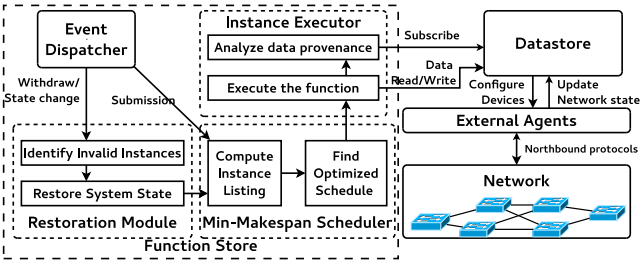


Figure 2: Runtime system for FAST.

Datastore Store the persistent network states and also act as the the storage for function metadata.

4. PRELIMINARY EVALUATION

We implement a prototype of FAST and evaluate its performance using Open vSwitch to simulate real topologies. We demonstrate the efficiency of FAST arisen from state awareness and fine-grained dependency tracking.

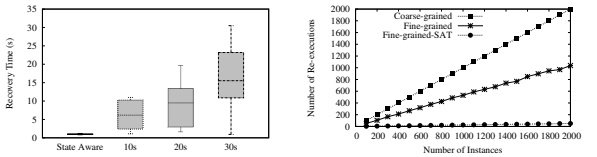
Figure 3a demonstrates the recovery time of restoring end-to-end connectivities after the control plane is notified of a random link failure event. We compare a routing function using FAST with periodic path computation engines in Floodlight[1] with various timeout values. We observe that being state-aware substantially improves the end-to-end recovery time.

Figure 3b shows how the number of re-executions changes with the number of running functions for different state tracking strategies. From the result, we see that re-executions both increase linearly but the slope with fine-grained state tracking is much smaller as expected, because FAST only monitors the touched links so that some link change events will not trigger re-executions. In addition, we find that allowing users to specify the satisfiability attribute of instances in FAST substantially reduces the number of re-executions, which is approximately 1/40 and 1/20 of those caused by the coarse-grained system and FAST without SAT attributes.

5. REFERENCES

[1] Floodlight OpenFlow Controller. http://floodlight.openflowhub.org/.

[2] F. B. Zhan. Three fastest shortest path algorithms on real road networks: Data structures and procedures. *Journal of geographic information and decision analysis*, 1(1):69–82, 1997.



(a) Benefits of state-awareness (b) Benefits of being fine-grained

Figure 3: Evaluation results.