Simon: Scriptable Interactive Monitoring for SDNs

Yiming Li

Supervised by Shriram Krishnamurthi

Department of Computer Science
Brown University

Spring, 2015

Abstract

Software-Defined Networking helps to simplify network control and configuration. However, understanding and debugging SDN involve some challenges for some set of common operations. Operators of debugging not only can determine behaviors of data- and control-plane, but also need to perform some interactive actions during a fact. We present SIMON (Scriptable Interactive Monitoring), probing SDN behaviors by monitoring networks, filtering events and reusing operators. Debugging with SIMON takes SDN as a black-box, which is independent of controller and network. Since SIMON is scriptable and interactive, users can define own reactive functions by using the power of Scala to refine the desired goals.

1 Introduction

Software-Defined Networking decouples the network control from forwarding functions resulting in being programmable and abstracting the applications among hundreds of devices, such as switches, routers and firewalls. Although logically centralized controller programs can simplify network management, even in a simple and emulated network, such as Mininet [3], debugging is still a laborious part. Unlike the traditional networks, SDN brings bugs in controller program or applications on the upper level, such as flow-table consistency problems [5, 7].

Debugging is a complex activity for dynamic network because this is difficult to replay the same process. In addition, the programmer may miss some knowledge of controller when developing a application. For instance, to implement a shortest-path routing application, the algorithm is correct except from forgetting to handle ARP packet, which is a real case in our networking course. By the tool Wireshark (`https://www.wireshark.org/`), finally, we found the bug via ping and observation. But, the tool only captures all the packets without automatically testing or debugging. This costs much time for the programmer to guess and reconsider the logic of program. Is it possible to debug or verify the network in real time? The first approach is that we can alarms some warnings after taking effect, such as NetSight [1], which focus on data-plane and installing flow-table rules escape their notice. To avoid error occurring (before taking effect), VeriFlow [2] introduces network-wide invariants involving access control policies between controller level and network level. But, the simple error can still satisfy

invariants: while installing OpenFlow rules, we miss necessary PacketOut messages. Therefore, we need some way to perform debugging and testing on both network and control state. Moreover, this is hard for programmer to verify the network manually, especially, in very high speeds, i.e., within milliseconds.

SIMON is an experiment in using reactive programming for network debugging, simplifying the burdensome aspects of the activity. It captures events on all plances, i.e., data-plane, control plane and northbound API events (communicate with other servers, such as REST). We have listed the futures.

- *Interactive:* SIMON exposes networking events as streams which users can explore. REPL (Read-Eval-Print Loop, an interactive prompt) easily loads user's input and provides any feedback.
- *Scriptable:* users can automate repetitive debugging tasks. While debuggers offer some built-in commands, users often define commands customized to specific programs in order to reuse them in the future.
- *Visible:* SIMON exposes data-plane events, control-plane events and northbound API messages. The powerful monitor guarantees to provide all the events as order timely.
- *Black-box testing:* it does not assume the user is knowledgeable about intricacies of the controller being debugged. This simplifies the understanding of network control and configuration. However, the programmers need to know the desired goals in order to take advantage of the rich API in SIMON.
- *Compatible:* support 1.0 - 1.3 OpenFlow messages and all the controllers. We have tested FlowLog (1.0), Ryu (1.1 and 1.3), FloodLight (1.3) and RouteFlow(Quagga, 1.0) with RFProxy.

2 Related Work

We describe in two main aspects: designing network monitor and programmable debugger.

Network Monitor NetSight [1] implements four applications to diagnose problems based on collecting a network's packet history. Compared to SIMON, the invariant checker is limited to data-plane. FortNOX [6] monitors flow-rule updates to prevent and resolve rule conflicts. However, it ignores other plane's events. STS [9] analyzes network logs to identify a minimal sequence of inputs that trigger a given bug. Although it captures packets from both data- and control-plane, tracing the past events delay and affect forwarding analysis.

Programmable Debugger The reactive programming is inspired by MzTake [4], where MzTake uses DrScheme REPL to debug Java programming. This is like a traditional program debugging, it allows users to take advantage of the rich API, i.e., view past events. Similar to SIMON, OFRewind [11] not only records SDN control plane traffic, also allows to replay the past streams. However, it is neither interactive nor scriptable to the feature SIMON provides. In addition, SIMON has introduced significant invariant checking for both planes. VeriFlow [2] creates a middle level to allow real-time verification. However, this may bring bug even the program does not violate any invariants in Section 3.

3 Ideal Models

SIMON allows the operator to subscribe to the interesting events, which is analogous to a filter. However, SIMON can provide far more flexibility. We show how to debug a simplistic example, a stateful firewall with one switch. A stateful firewall should (1) allow all traffic from internal to external ports, but (2) deny traffic arriving at external ports unless (3) it involves hosts that have previously communicated.

When the initial ping occurs from internal port to external port (ignore ARP packet), the packet will arrive controller to trigger two rules:

1. One PacketIn with payload ICMP sends to controller.
2. Controller sends out FlowMod to install two rules to allow access in the bi-direction.
3. Controller also sends out PacketOut with payload ICMP.

The programmer may ignore (3) PacketOut, resulting in dropping the first ICMP packet. Since OpenFlow rules have been installed correctly, later ping command timeouts and retries, the connection will still work. To detect this bug and check any violated events, we define expectations via streams from SIMON monitor:

1. If packet arrives the internal port, we expect it will pass in the external port.
2. If packet arrives the external port without any packet from internal port, we not expect it passes the internal port.
3. If packet arrives the external port after some packet arrives the internal port, we expect it passes the internal port.

For (2) and (3), we need to keep the state of network. Also, while none of expectations are involved of OpenFlow messages, we focus on actual behaviors without considering about installing rules. As follows, we list three expressions corresponding to these expectations. The first line filters SIMON's stream to only leave ICMP packets. Originally, this was cold observable [1] , which meant that multiple subscribers to Simon.nwEvents() would cause the same message to be processed multiple times. But, we make it as a hot observable [2] to maintain consistency via `publish` in building the stream of events (here, we don't show the code that is a low level under SIMON). Compared to call back functions in Section 4.2, observable streams are interactive for subsequent computations.

```
val ICMPStream = Simon.nwEvents().filter(SimonHelper.isICMPNetworkEvents)

val e1 = ICMPStream.filter(isInInt).flatMap(e =>
                        Simon.expect(ICMPStream, isOutSame(e), Duration(100, "milliseconds"))
    )

val e2 = ICMPStream.filter(isInExtNotAllow).flatMap(e =>
                        Simon.expectNot(ICMPStream, isOutSame(e), Duration(100, "milliseconds
    "), ICMPStream.filter(inOppositeSrcdst(e))))

val e3 = ICMPStream.filter(isInExtAllow).flatMap(e =>
                        Simon.expect(ICMPStream, isOutSame(e), Duration(100, "milliseconds"))
    )
```

The following lines above set more details to filter the packet by defining several helper functions. For example, `isInExtAllow` ("is incoming on external port, allow to pass", similar to `isInInt`, "is incoming on internal port"), we always filter to get incoming packet to expect next packet by `e.direction`. `fwswitchid` and `fwexternals` are initialized parameters that which switch displays the stateful firewall and which interface is external. To keep the track of state, we create a mutable set `allow` to remember pair of source and destination. Then, using SIMON's build-in `rememberInSet` function to set the update, where if any streams come from internal port, the set will add the opposite address pair automatically.

```
def isInExtAllow(e: NetworkEvent): Boolean = {
                e.direction == NetworkEventDirection.IN &&
                e.sw == fwswitchid && fwexternals(e.interf) &&
                allowed((e.pkt.eth.dl_dst, e.pkt.eth.dl_src))}

val allowed = new scala.collection.mutable.TreeSet[Tuple2[String, String]]();

Simon.rememberInSet(ICMPStream, allowed,
                {e: NetworkEvent => if(isInInt(e))
                        Some(new Tuple2[String,String](e.pkt.eth.dl_src, e.pkt.eth.dl_dst))
                        else None})
```

[1] Cold observable: Sequences that are passive and start to produce notifications on request once subscribed to behavior.
[2] Hot observable: Sequences that are active and always produce notifications regardless of subscriptions.

Back to three basic expressions, we use SIMON's `expect` and `expectNot` to assert the future event during a wait time. `isOutSame` helps to judge whether the incoming packet from argument is same to the future outing packet including ICMP sequence number. Since each packet sends quickly from one host to another host, such as 10ms, we set a timeout to alert them whether any unexpected event occurs. So, for `expect`, after timeout, no expected event happens, the steam emits an `ExpectViolation` event. If any event emits before that wait time, it will produce that `ExpectSucceed` event. After defining these expected events, we merge them, then subscribe to `printwarning` function, which gives the user a hint: red printing means bad things happen, while green printing is the normal process.

4 SIMON's Implementation

4.1 Architecture

SIMON's architecture is shown in Figure 4.1. The events captured from network or controller are separated from debugging process. The current implementation is based on a Mininet-emulated SDN running different kinds of controller applications. The concurrent network events receiving from network need to sort by timestamp, then send out to a remote debugger. To keep the oriented programming in both sides, we use Gson API (`https://code.google.com/p/google-gson/`) to convert Java Objects into their JSON representation as a string for deserializing and serializing. The programmers taking the network as a block-box pay more attention into writing a simple script for the sake of desiderata.

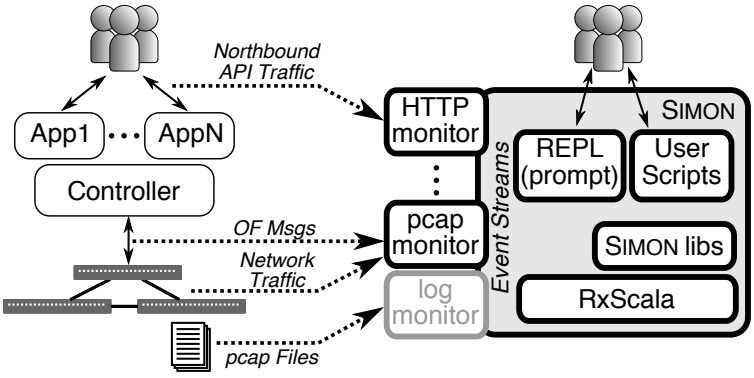

Figure 4.1: SIMON's architecture. Events are captured from the network by monitors, which feed into SIMON's interactive debugging process. The users only need to write some scripts for testing, and then, adjust the code in application side.

Monitor The basic monitor depends on a Java wrapper for nearly all libpcap library, i.e., JnetPcap 1.4 (`jnetpcap.com`), which does not assume the user is knowledgeable about controller in use. In order to deserialize both data- and control-plane, we use Floodlight (`www.projectfloodlight.org/floodlight/`) APIs. In our prototype, only some network events can be captured (ARP, IP, TCP, UDP, PackIn, PackOut and FlowMod) that are easy to extend. Users also can implement other monitors, for example, the HTTP monitor that we use to capture REST API calls to debug a firewall running on the top of Ryu controller (`http://osrg.github.io/ryu/`). Although packets come from different monitors, they feed into one buffer. The programmers has a responsibility to make a distinction among them once send out to debugger.

To avoid delay, the monitors generate the thread for each switch interface to capture packets with timestamp from the kernel. Since the average switching time for threads is 10ms, the network events may be out of order. We provide one buffer to store all the packets and reorder after each 200ms. The experiment has shown 50ms is enough to guarantee more than 96% of the packets in order (Figure 4.2).

Here, we limit the speed of sending, such as TCP with 100Mb/s. Otherwise, the high speed network makes no sense to debug and test.

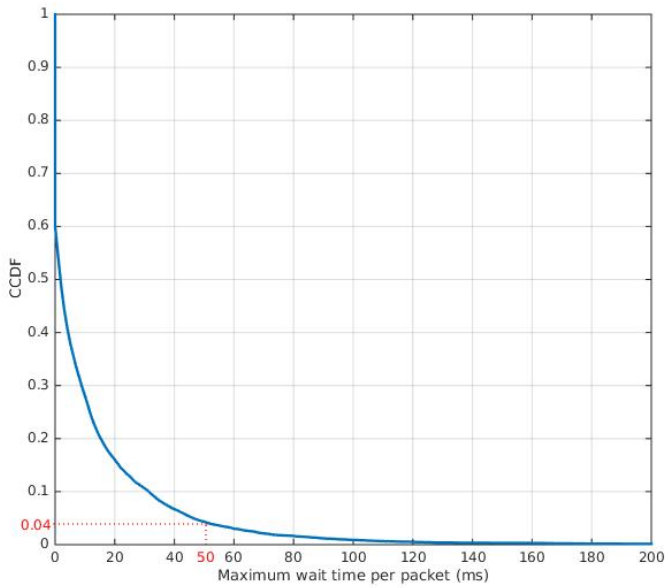

Figure 4.2: The data is collected on the shortest-path routing with four switches and four hosts by pinging 200 times and running Iperf simultaneously.

Debugger All the events captured by monitors finally arrive the remote debugger. Synchronously handling each event provides a simple approach to operate and verify network. But, this quickly becomes untenable because it blocks other events. In Section 4.2, we compare callback and reactive programming. Using Scala's ReactiveX library (`reactivex.io`), the users can consume the streams or do some compositional programming. To provide a modern console, SIMON is wrapped in REPL with the defined helper functions.

4.2 Callback vs Reactive Programming

Callback Idea Models In general, if the operator results in a set of events, the natural solution is to use synchronous calls. We implement debugger by callback function, where this is what we did for debugger initially. Here, we provide abstract `verifier` class for programmer to extend, set any filter by `addInterestedEvents` and check expected events or not by `checkEvents`:

```
abstract public class Verifier implements Runnable {
        abstract public void verify(NetworkEvent event);

        protected void addInterestedEvents(PacketType t){
                interestedEvents.add(t);
        }

        protected void addExpectedEvents(NetworkEvent eve) {
                ...
        }

        protected void addNotExpectedEvents(NetworkEvent eve) {
                ...
        }

        protected void checkEvents(NetworkEvent e) {
                ...
        }
}
```

Each event fires immediately after the operator from `verify`. Because synchronous calls block, this is obvious to handle each event, then to generate next expected or not expected events list to verify the future events. We show a small stateful firewall application, where internally-arriving traffic triggers installing rule for arriving packet from external port:

```
public class StateFirewallVerifier extends Verifier {
        @Override
        public void verify(NetworkEvent event) {
                if (event.direction == NetworkEventDirection.IN) {
                        if (allowInterfs.contains(event.interf)) {
                                addExpectedEvents(generateEvent(event))
                        } else {
                                addNotExpectedEvents(generateEvent(event))
                        }
                } else {
                        checkEvents(event);
                }
        }
}
```

The main problem here is to block other events when dealing with one event. Instead, an asynchronous communication is necessary. However, this brings concurrency issues if several callback functions try to modify the same state. In addition, both synchronous and asynchronous callbacks focus an inversion of a program's usual control logic: collecting events push into callbacks. It moves the complexity into callback's internal state, which means no replay, be difficult to register a new callback, maintain state or integrate with existing code.

Reactive programming Instead, reactive programming maintains the perspective that the program pulls events, which automatically abstracts the callback, encapsulates updated streams and handle the consistency. In effect, the programmers can consume dynamic data streams by involving some functions. What's more, replay is easy to provide, especially, the user needs to deal with same past events in different functions. More interestingly, the users define program-specific commands to analyze and understanding the network with automate debugging program.

4.3 SIMON API

To be maximally useful and minimally intrusive, we define the rich functions. It is also convenient for users to access these via console so that they can implement problem-specific commands. Figure 4.3 contains a selection of ReactiveX operators (top), as well as the current built-in SIMON functions for network debugging and monitoring (bottom).

Show Events Although the expectations above have detected the bug, we can display these events in one new window by `showEvents` window without cluttering the prompt, similar to tcpdump. For the previous example, we show the initial incoming event following second incoming event and second outgoing event. Therefore, the bug is that drop the initial packet in switch.

Do we miss the first packOut that should be sent out from controller or switch drops that? To find the reason, we need to capture the OpenFlow events that are related to that initial ICMP packet. In SIMON, we provide that powerful function, called `cpRelatedTo` ("control-plane related to"), which produces the PacketIn, FlowMod and PacketOut related to that packet. For instance, `showEvents( ICMPStream.flatMap(cpRelatedTo))`, this captures these events (default is JSON string, here, just to make it clear). This is very obvious that we miss the outgoing PacketOut from controller to switch. Therefore, we conclude the bug is from application.

```
Incoming PacketIn, from switch to controller.
Installing FlowMod, allow host1 to host2.
Installing FlowMod, allow host2 to host1.
```

Replay Events Sometimes, the users hope to apply different functions for the past events. In ReactiveX scala, it has provided `cache` operator. In order to avoid overhead, this replay function only starts to

filter	Applies a function to every event in the stream, keeping only events on which the function returns true.
map	Applies a function to every event in the stream, replacing that event with the function's result.
flatMap	Like map, the function given returns a stream for each event, which flatMap then merges.
cache	Cache events as they are broadcast, allowing subscribers to access event history.
timer	Emit an event after a specified delay has passed.
merge	Interleave events on multiple streams, producing a unified stream.
takeUntil	Propagate events in a stream until a given condition is met, then stop.
subscribe	Calls a function (built-in or user-defined) whenever a stream emits an event.

expect	Accepts a stream to watch, a delay, and a function that returns true or false on events. Produces a stream that will generate exactly one event: an ExpectViolation or the first event on which the function returned true.
expectNot	Similar to expect, but the function describes events that violate the expectation.
cpRelatedTo	Accepts a packet arrival event and returns the stream of future PacketIns and PacketOuts that contain the packet, as well as FlowMods whose match condition the packet would pass.
showEvents	Accepts a stream and spawns a new window that displays every event in the stream.
isICMP	Filtering function, recognizes ICMP traffic. (Similar functions exist for other traffic types.)
isOutSame	Accepts an incoming-packet event and produces a function that returns true for outgoing-packet events with the same header fields.

Figure 4.3: Selection of Reactive Operators and SIMON API functions. The first table contains commonly-used operators provided by Reactive Scala (`reactivex.io/documentation/operators.html`). The second table contains selected SIMON API functions.

cache after the first subscribe. Once cache the events, any copy can applied to verify the network. However, this will emit all events as order with different time distribution.

Interact with Networks Mininet provides a easy way to control host in the current master branch using the `util/m` script. For Scala, the "!" operation will execute the preceding string as a shell command. To illustrative it, we filter the outgoing event in a special switch to send ping to 10.0.0.1 (h1). But, we also need to act as 10.0.0.2 (h2).

```
ICMPStream.filter(
  e => e.sw == fwswitchid &&
  e.direction == NetworkEventDirection.OUT)
    .subscribe( _ => "ping 10.0.0.1" ! );
```

5 Real Applications

We evaluate SIMON by three real applications. One is Ryu stateful firewall that has been described in Section 4, while another firewall changes the state via HTTP messages. We also test a complex application, shortest-path routing from third-party.

5.1 Ryu REST Firewall

To test the firewall from Ryu controller platform, we create an ideal model. However, this is different from stateful firewall that it has internal and external ports. The users can control REST firewall via sending REST messages to add or remove rules. In order to keep the dynamic state in our model, we define `policyList`, where the payload of any REST message is taken as a policy (rule) to insert into this list. Therefore, there are two kinds of expected events: the packet allowed by policy can pass the switch, otherwise, we should not capture it in the other side via `isMatchPolicyAllow` and `isMatchPolicyDeny` operators. For capturing REST messages, we just add another monitor to listen for messages from users connecting to the same debugger server. From the view of the programmer, streams produce the mixed events from REST messages and Mininet. In following example, we just filter them, such as `RESTStream`.

```
val RESTStream = Simon.nwEvents().filter(SimonHelper.isRESTNetworkEvents)

RESTStream.subscribe(addPolicy(_))

def addPolicy(e: NetworkEvent) {
        val p = new Gson().fromJson(new String(e.pkt.eth.ip.tcp.payload), classOf[Policy])
        policyList = insert(p, policyList)
}
```

5.2 Shortest-path Routing

We test the shortest-path routing by two different controller with the same topology, such as the final project of a networking course with FloodLight controller in Java and RouteFlow [8] creating a virtual Quagga (`quagga.net`) instance for each switch and emulates distributed routing protocols in an SDN. The ideal model uses the BellmanFord algorithm to compute the length of a single shortest path as a reference. To handle all the shortest paths, we count each hop and determine the length of packet track. Once packet has more hops than that of a shortest path, it will automatically generate a warning. In the model, only the network's topology is loading into the debugger without caring any kind of controller, which brings more benefits to use the same model for different implementations. Even, the users have no knowledge of the controller and the application, but merely a sense of what network behaviors they expect.

6 Conclusions

We have presented the design and implementation of SIMON in a realistic virtual environment (Mininet). SIMON's architecture separates the network events from handling events. The monitors capture all the visible events, not limited to data-plane. The reactive programming has sufficient library to deal with concurrent network events, compared to synchronous callbacks. Although implementing an ideal model takes roughly 100 lines of code, this programming logic makes users uncomfortable.

The network events temporarily stored into buffer are reordered via timestamp from kernel's time. However, this approach has modify the original time distribution of events; for this, timeout of expected events in SIMON has time inconsistency. Even, capturing events from real different switches need to synchronize clocks by NTP (Network Time Protocol) when extending to production of many networking environments. To make SIMON applicable to real networks, some cases only care about causal order. Therefore, we may introduce topological sort to reorder the packets depending on the track. If any ring occurs, tie can be broken via timestamps.

SIMON can integrate many other sources. For example, an OpenFlow proxy like FlowVisor [10] or NetSight's [1] collection plays a part of monitor to feed the captured messages to SIMON. Moreover, SIMON applies to a non-SDN network or networks with middleboxes. Thus, SIMON is a valuable tool for testing and verifying.

References

[1] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Networked Systems Design and Implementation*, 2014.

[2] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Networked Systems Design and Implementation*, April 2013.

[3] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks*, 2010.

[4] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering Journal*, 2006.

[5] P. Perešíni, M. Kuźniar, N. Vasić, M. Canini, and D. Kostić. OF.CPP: Consistent packet processing for OpenFlow. In *Workshop on Hot Topics in Software Defined Networking*, 2013.

[6] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[7] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2012.

[8] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[9] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. B. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2014.

[10] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *Operating Systems Design and Implementation*, 2010.

[11] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, 2011.