SDNTrace Documentation

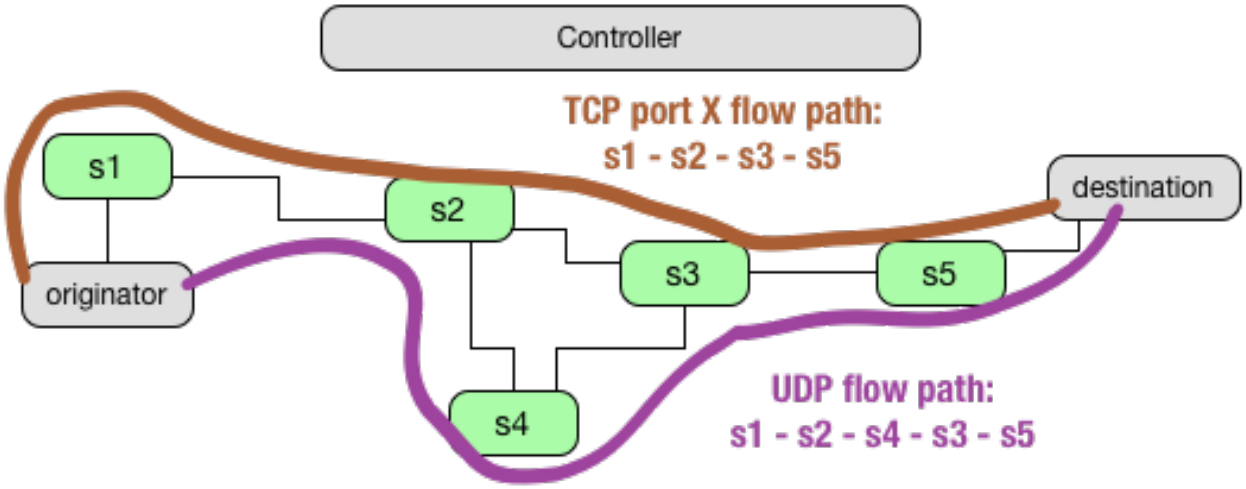
Release 0.1

Deniz Gurkan, Amir Ali Kouhi Kamali, Long Tran, Nicholas Bastin

March 15, 2016

Contents

Demonstration setup for SDNTrace is composed of a topology of SDN devices, a controller, two end points that participate in the tracing of the path for individual **flow** definitions. Protocol Design section outlines problem statement, our motivation, design goals, and constraints. The Implementation of SDNTrace Protocol includes the algorithm and how the protocol has been implemented as a northbound application. Finally, GENI DEMO is where we explain the demonstration and the workflow of the experiment.



CHAPTER 1

LEXICON

1. To-be-traced packet: A data plane packet that represents a flow defined with L2-L7 header field(s) with possible path(s) to a destination host
2. SDNTrace protocol: Message exchange mechanism and the associated message format designed to determine the path of a to-be-traced packet in a software-defined network
3. NB App: The application that uses the RYU controller to run the SDNTrace protocol
4. Originator (node): An end host that issues the trace request for a to-be-traced packet to a destination host
5. Probe: An SDNTrace protocol packet, TraceRequest, that is sent by the originator node
6. Hop Object ID: A preferred identification object for the nodes on the path of the to-be-traced packet from originator to destination node.

SDNTrace implementation in this repo is a northbound application (NB app) running on RYU controller. The originator node (probe is sent from this node) is running a packet generator that sends probe packets to the network while the NB app facilitates the creation of the reply packets with a hop object ID for each node on the path of the to-be-traced packet.

In the topology above, there are two paths carrying two different protocols. One is for UDP packets and another for TCP packet flow. This demonstration involves the identification of hop IDs as DPIDs of the OpenFlow switches on the path of each flow for tcp and udp protocols.

This document is divided into three parts:

1. The protocol and its design decisions
2. Implementation as a NB app on RYU controller
3. The demonstration of its basic functionality on GENI testbed

Note: Acknowledgment: This work was partially funded by National Science Foundation ACI grant award no. 1341019.

Contents:

1.1 REFRESH OFTEN!

This page is updated **very frequently**.

Please refresh your browser to load revisions and new material.

1.2 Protocol Design

This is a work in progress with a demo implementation to illustrate ideas and expectations on a tracing tool for SDN deployments.

1.2.1 Motivation

The project has been inspired by a requirements discussion during the TechExchange meeting of the Internet2 in October 2014.

A project charter has been created and shared with the Internet2 SDN workgroup to initiate participation from the community on the expectations from such a troubleshooting tool around the university campuses, particularly, for those deploying any SDN network devices: charter document

Current network troubleshooting tools such as `ping`, `traceroute`, and `tcpdump` help network engineers achieve some insight in conjunction with protocols such as the SNMP in legacy networks. However, deeper visibility into data plane behaviour in SDN may be possible. And, development of troubleshooting tools may address more such visibility of the networks.

1.2.2 Existing `traceroute` Tool

The protocol design evolves around the requirements by keeping in mind of the current functionality provided by the *traceroute* tool. Requirements from TechExchange encompasses a path tracing capability for L2 and above, with multiple domains of networks, and possibly some information on crossing the boundaries between SDN and non-SDN.

We examined the *traceroute* as a path tracing tool to baseline our goal with the SDNTrace protocol. In this respect, an SDNTrace tool should yield at least as much information as the traceroute tool. We identified many shortcomings of the traceroute tool. A useful presentation on this topic is in the NANOG on October 7, 2014.

1.2.3 SDNTrace Protocol Requirements

After identifying major shortcomings of current traceroute, we tried to address them with any opportunities we see may be possible with SDN and other means while addressing the requirements listed in the charter. Below is a high-level wish list on protocol design. However, we were able to only implement part of this list.

We are including the list here for open discussion on future participation:

- **A hop exists**: What identifier do we want to know about the hop? E.g.: IP address, name, MAC, AS number, organization, description, who to call, DPID of the switch, etc., operator-specified stuff (up to the operators of the network)
- **Processing of probe packets in the data path**: Any tracing method will have at least similar limitations to traceroute. In order to increase information about the datapath, a design decision has been made that trace probes will be processed as exception packets in the slow path.
- **Timestamping** - not addressed in this implementation.
- **Reverse path determination**: same routers may have different interface names/identifiers, and the reverse path may actually be physically different.
- Multiple domain case: ??
- Probes should look like data packets.

- Repetitive measurements, sequence numbering of the probe packets, accumulate responses from the hops up to the point at the network. Building resiliency against packet loss in the network.
- NTP - synchronize clocks at hops so data probe can be timestamped at hops. Removes the confusion created by the protocol support at tunnel points at MPLS, and the RTT being equal for many hops within the tunnel.

1.2.4 SDNTrace Protocol Design

Path is a list of hops on the way to the destination, each hop should be identified at L2 level (identifier: switch id-DPID, network admin tel. no, physical address, name of the device in SNMP, belonging to a domain/service provider)

Optimize/focus on the most common/important cases from our wish list above for this first stab at the protocol design

- One domain with SDN devices to be traced for path
- L2 and up is to be discovered
- All packets to be processed in the CPU through an exception packet handling mechanism
- Timestamp, reverse traceroute, hidden box, info about domains on the path, L2 devices - are key in making this better than traceroute: implementation of these elements will be a work in progress.

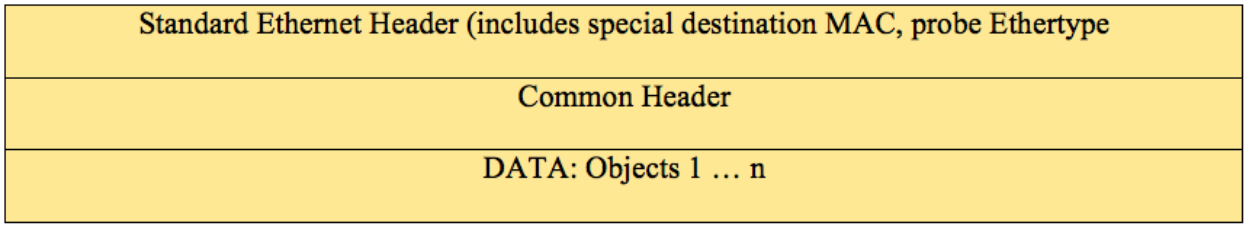
In essence, the protocol is composed of a probe message that carries the *to-be-traced* packet as payload.

Protocol messages are:

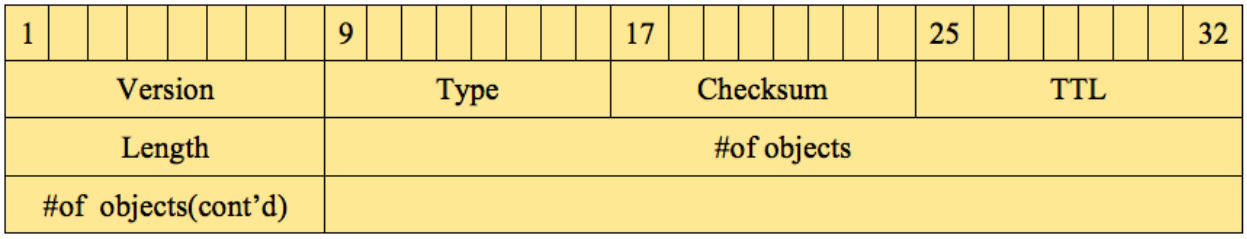
1. TraceRequest message (the probe sent by the originator host to the network)
2. TraceReply message (accumulates the hop object id's of all nodes on the path of the packet being traced)

The header fields of these packets follow exactly the format of an Ethernet packet with a reserved destination MAC address to identify itself as a packet that needs special handling by the controller.

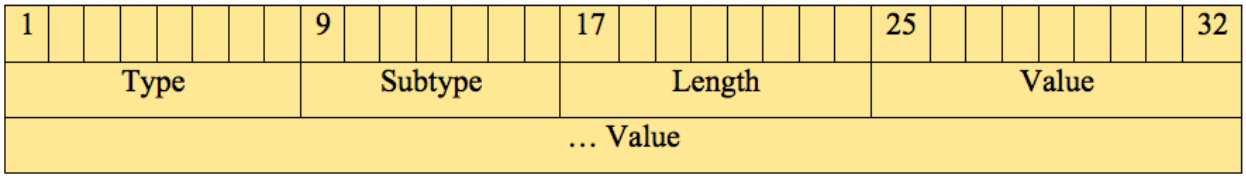
probe packet format:



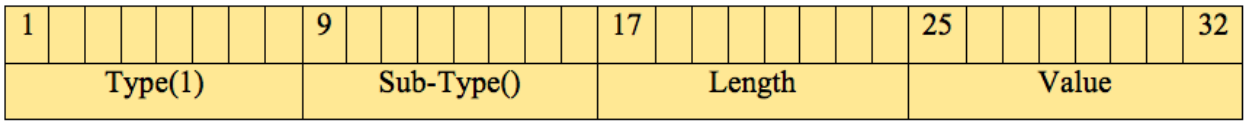
format of the common header:



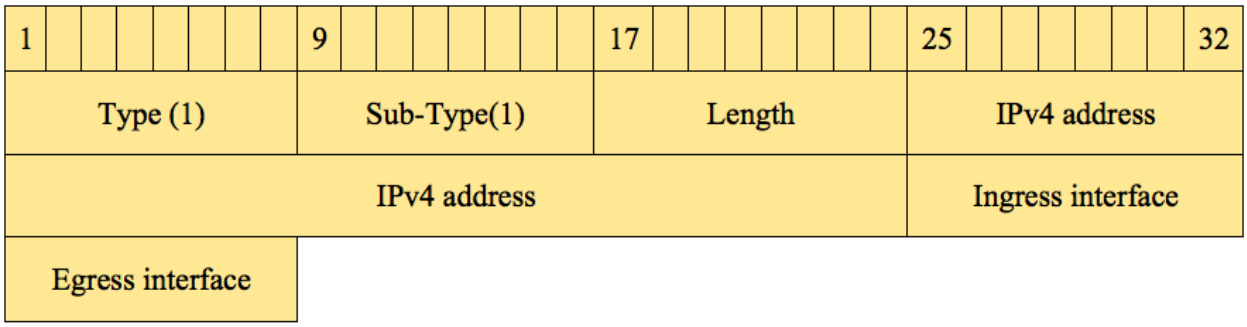
format of the object header:

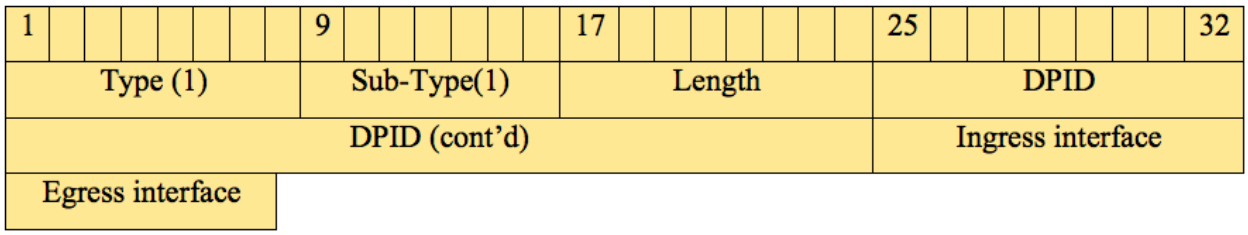


hop object ID:

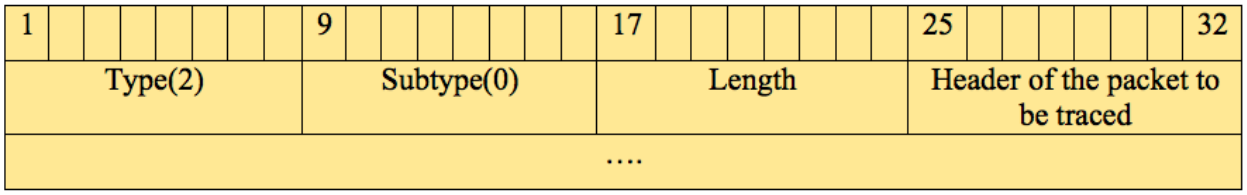


Example object IDs for ipv4 Object and OpenFlow DPID:





The packet to be traced:



A sample probe packet for SDNTrace protocol (i.e., the TraceRequest):

1 … 9 … 17 … 25 … 32			
Ethernet Header Source: $H2_{MAC}$ Destination: special Mac address			
Version 1	Type Request	Checksum	
TTL 255	Unused	Length	
Type FlowSpec	Sub-type 0	Length	Ethernet Header+IP Header
Ethernet source:$H2_{MAC}$ Ethernet destination: $H1_{MAC}$ source IP address: $H2_{ip}$ Ethernet destination : $H1_{IP}$			
type Hop	Sub-type DPID	Length	DPID S1
DPID (cont'd)		Ingress interface $S1_1$	Egress interface $S1_2$
type Hop	Sub-type DPID	Length	DPID S2
DPID (cont'd)		Ingress interface $S2_1$	Egress interface $S2_1$

1.3 Implementation of SDNTrace Protocol

This is a work in progress with a demo implementation to illustrate ideas and expectations on a tracing tool for SDN deployments.

We have implemented this current *work-in-progress* version as a northbound application on RYU. We have tested and demonstrated our implementation during the GENI Engineering Conference 23, GEC23.

1.3.1 Algorithm of SDNTrace Protocol Demonstration

1. Connect to all switches in the network (assumes there is a forwarding table constructed in all switches between orginator and the destination host).
2. Write flows into the switches that will forward all SDNTrace messages to the controller by recognizing the special ETH_TYPE used.
3. Write corresponding forwarding flow rules into the switches for the paths to be followed in the demonstration (i.e., the tcp and udp network packet paths).
4. Before a trace message arrives, at a port status change, retrieve all flow tables from all switches that are connected to the controller.
5. Determine the destination MAC address for the to-be-traced packet from `flowspec` object.
6. Determine the type of the SDNTrace message: request or reply
    - If directly attached and trace type is request start TraceReply message construction.
    - If not directly attached and trace message type is reply: retrieve and append the hop Object ID (i.e., DPID) to the trace objects, compose the trace request packet and send back with the forwarding information to the switch.
    - If the message is a reply, forward the reply message on through the ingress port this message came from originally based on information retrieved from the existing Hop Object in the reply message.

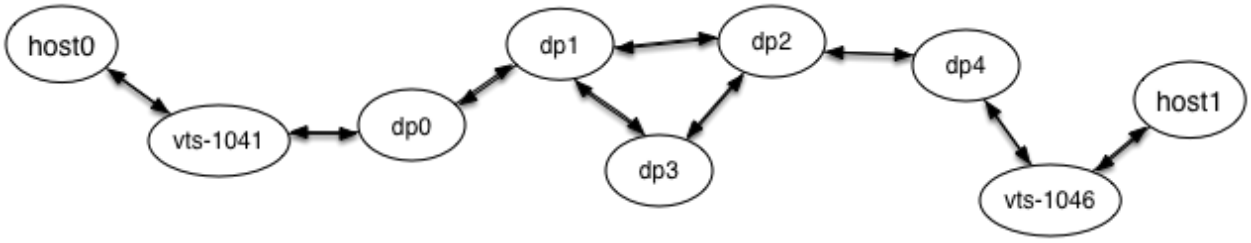
1.4 GENI DEMO

This is a work in progress with a demo implementation to illustrate ideas and expectations on a tracing tool for SDN deployments.

Note: All experiment control of the GENI demonstration is achieved using the `geni-lib` tool at its official documentation here. Installation of `geni-lib` is required to set up and control the experiments: please use installation instructions at the geni-lib documentation site.

1.4.1 Multi-path Topology

The topology below has been created using the VTS (virtual topology services) on GENI:



There are six datapath elements. The internal circuits connect:

```
dp0 to dp1
dp1 to dp2
dp1 to dp3
dp3 to dp2
dp2 to dp4
```

And, there are only two PG circuits to attach VM end points on the topology:

```
dp0 with originator
dp4 with destination
```

The paths for the two application flows are to be setup with flow rules on switches as:

```
TCP flow on: dp0, dp1, dp2, dp4, destination
UDP flow on: dp0, dp1, dp3, dp2, dp4, destination
```

The GENI reservations are accomplished using the geni-lib scripts posted in the repo's gec23demo directory. `OFLoopTopology.py` reserves the topology described here.

1.4.2 Controller Setup

At the controller node, install RYU controller and follow instructions from the README file of this repo to set repo and NB app up with its dependencies.

Start SDNTrace application:

```
ryu-manager ryu-app/ryu-trace-app.py
```

1.4.3 Originator Setup

At the orginator node, install the packages listed in README and run to send a probe packet:

```
cd sdntrace/
python trace-client/trace-client-with-GUI.py
```

1.4.4 Experiment Workflow

- Before starting the controller, modify topology using the portDown feature of VTSAM:

```
    VTSAM.UtahDDC.portDown(context, "sdntraceDemo", "dp3:0")
```

- Start the controller, and verify that the datapaths are all connected at this time with this directed acyclic graph.

Note: You can verify connectivity of datapaths using the sliverstatus feature of VTSAM by calling the `VTSAM.UtahDDC.sliverstatus(context, "sdntraceDemo")`.

- The sliverstatus printed text will tell us whether our datapath elements are connected to the controller and what their DPIDs are:


```
{'geni_resources': [{'geni_error': '',
  'geni_status': 'ready',
  'geni_urn': 'urn:publicid:IDN+ch.geni.net:VTS-experiments+slice+sdntraceDemo:631cb687-1101-41c
  'geni_status': 'ready',
  'geni_urn': 'urn:publicid:IDN+ch.geni.net:VTS-experiments+slice+sdntraceDemo:631cb687-1101-41c
  'vts_datapaths':
  [{'client-id': 'dp0',
    'connected': False,
    'dpid': 'a2:88:be:9d:0c:b5:fd:f1',
    'ports': [{'client-id': 'dp0:1', 'name': 'vlan3693'},
    {'client-id': 'dp0:0',
    'name': '7137318802624'}]},
   {'client-id': 'dp1',
    'connected': False,
    'dpid': 'a0:51:5f:83:f9:a7:6e:fc',
    'ports': [{'client-id': 'dp1:2',
    'name': '1471796398048'},
    {'client-id': 'dp1:0',
    'name': '4383286160025'},
    {'client-id': 'dp1:1',
    'name': '1661875448123'}]},
   {'client-id': 'dp2',
    'connected': False,
    'dpid': 'a6:b9:8e:4c:e6:a8:51:07',
    'ports': [{'client-id': 'dp2:1',
    'name': '3872144899212'},
    {'client-id': 'dp2:0',
    'name': '1288915914021'},
    {'client-id': 'dp2:2',
    'name': '893223727791'}]},
   {'client-id': 'dp3',
    'connected': False,
    'dpid': 'a6:fc:67:0c:43:d3:a4:a9',
    'ports': [{'client-id': 'dp3:0', 'name': '53343421289'},
    {'client-id': 'dp3:1',
    'name': '946368135596'}]},
   {'client-id': 'dp4',
    'connected': False,
    'dpid': 'ba:32:63:0e:e6:4c:3a:51',
    'ports': [{'client-id': 'dp4:1', 'name': 'vlan3692'},
    {'client-id': 'dp4:0',
    'name': '1261830267274'}]}]
 }
```


- Once all nodes are connected the controller will also display the DPIDs read on the network, on controller terminal where the NB application is running:

```
ubuntu@ip-172-31-22-225:~/traceprotocol$ sudo ryu-manager --ofp-tcp-listen-port=7733 ryu-t
loading app ryu-trace-app.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
```

```
instantiating app ryu-trace-app.py of SimpleSwitch
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
{}
[12032605606802334889L]
{}
[12032605606802334889L, 1155211953961588710OL]
{}
[12032605606802334889L, 11711820412709371377L, 1155211953961588710OL, 120137899419259578 95
{}
[12032605606802334889L, 11711820412709371377L, 1155211953961588710OL, 134168951555345762 09
{}
[12032605606802334889L, 11711820412709371377L, 1155211953961588710OL, 134168951555345762 09
```

- From the originator node, issue a `ping` to the destination host to start the learning process for the controller:

```
...
{}
[12032605606802334889L, 11711820412709371377L, 1155211953961588710OL, 134168951555345762 09
Learned MAC 02:36:de:dc:de:9e at DPID 11711820412709371377
Learned MAC 02:aa:c7:52:80:b8 at DPID 13416895155534576209
```

- Before we run the trace, the topology needs to know the paths for each flow for this network. Create (or modify the existing `flowRules.py`) a flow definition file inside the `gec23demo` directory.
- The `flow.info` is used to write these flows into the datapaths in VTS topology using the addFlows feature:

```
d = json.loads(open("/Users/dgurkan/sdntrace/gec23demo/flow.info").read())
VTSAM.UtahDDC.addFlows(context, "sdntraceDemo", d)
```

- Bring the port that was down, back up again using:

```
VTSAM.UtahDDC.portUp(context, "sdntraceDemo", "dp3:0")
```

- Once the flows are written into the datapath elements in the topology, we expect paths for particular flows (in this example, there is a separate path for tcp flows and another path for udp). The flows can be checked on the datapaths using the dumpFlows feature of VTS:

```
VTSAM.UtahDDC.dumpflows(context, "sdntraceDemo", ["dp0", "dp1", "dp2", "dp3", "dp4"])
```

- Now, the topology has two paths setup for two different flows (differentiated by the network protocol type, 17 for udp, and 6 for tcp). Flows are written into the devices on the paths between the originator node and the destination. The controller has learned the forwarding of end hosts. We can run the client (or, originator) SDNTrace to insert a probe into the network to trace the path of either one of these flows. Using the GUI gives us only end destination MAC address match option:

```
sudo python trace-client-with-GUI.py eth1
```

- The command line version of `trace-client.py` has the option to specify the type of network protocol to be traced:

```
sudo python trace-client.py eth1 02:aa:c7:52:80:b8 tcp
```

- The client then should issue a probe for the packet to be traced. The controller will display all trace requests and replies received during the course of the trace:

```
ubuntu@ip-172-31-22-225:~/traceprotocol$ sudo ryu-manager --ofp-tcp-listen-port=7733 ryu-trace-a
loading app ryu-trace-app.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
```

```
instantiating app ryu-trace-app.py of SimpleSwitch
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
{}
[12032605606802334889L]
{}
[12032605606802334889L, 1155211953961588710OL]
{}
[12032605606802334889L, 1171182041270937137 7L, 1155211953961588710OL, 12013789941925957895L]
{}
[12032605606802334889L, 1171182041270937137 7L, 1155211953961588710OL, 13416895155534576209L, 120
{}
[12032605606802334889L, 1171182041270937137 7L, 1155211953961588710OL, 13416895155534576209L, 120
Learned MAC 02:36:de:dc:de:9e at DPID 11711820412709371377
Learned MAC 02:aa:c7:52:80:b8 at DPID 13416895155534576209
port modified 1
port modified 3
Trace Request: 11711820412709371377
Trace Request: 11552119539615887100
Trace Request: 12013789941925957895
Trace Reply: 12013789941925957895
Trace Reply: 11552119539615887100
Trace Reply: 11711820412709371377
```

- The result from the trace client should display the hop IDs on the path:

```
dgurka01@host0:~/sdntraceprotocol$ sudo python trace-client.py eth1 02:aa:c7:52:80:b8 tcp
WARNING: No route found for IPv6 destination :: (no default route?)
.
Sent 1 packets.
0180c20000000236dedcde9e882000020014620000000502002502aac75280b80236dedcde9e08004500002800010000
ethernet(dst='01:80:c2:00:00:00',ethertype=34848,src='02:36:de:dc:de:9e'), TraceReply(checksum=0
HopObjectDPID(dpid=11711820412709371377L,egress_interface=2,header=ObjectHeader(length=13,sub_ty
HopObjectDPID(dpid=11552119539615887100L,egress_interface=2,header=ObjectHeader(length=13,sub_ty
HopObjectDPID(dpid=12013789941925957895L,egress_interface=3,header=ObjectHeader(length=13,sub_ty
HopObjectDPID(dpid=13416895155534576209L,egress_interface=1,header=ObjectHeader(length=13,sub_ty
```

1.5 Acknowledgment

Acknowledgment: This work was partially funded by National Science Foundation ACI grant award no. 1341019 and EAGER award no. 1449151.

Disclaimer: Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1.6 Indices and tables

- genindex
- modindex
- search