LogSnap: Creating Snapshots of OpenFlow Data Centre Networks for Offline Querying

Jonathan Sherwin
Dept. of Computer Science
Cork Institute of Technology
Cork, Ireland
jonathan.sherwin@cit.ie

Cormac J. Sreenan
Dept. of Computer Science
University College Cork
Cork, Ireland
cjs@cs.ucc.ie

Abstract— **Software-Defined Networking (SDN) has enabled automated modification of the behavior of network devices to match changes in network policy. This facility has driven adoption of SDN in Data Centre Networks (DCNs), particularly multi-tenant DCNs, where network policies are used extensively and can change rapidly as tenants arrive, leave, and modify their resource usage. It is useful for a DCN operator to have a way to query the past state of a network, e.g. for debugging or verification. In a multi-tenant DCN whose behaviour changes frequently under the programmatic control of SDN, this is an important but complex function to provide. While SDN makes the problem more challenging, it also helps to provide the solution – changes in network policy are communicated in packets sent from an SDN controller to the network devices, and those packets are amenable to capture and analysis to reveal the state of the network. Our solution, LogSnap, records messages exchanged over time between an SDN controller and switches in a network, and can quickly recreate the network in an emulated environment for any point in the recorded history. We have evaluated the system for its accuracy, the speed with which it can recreate the network, and quantified the storage implications of speeding up network reproduction.**

Keywords— Software-Defined Networking, Data Centre Networks, Network Management, OpenFlow

I. INTRODUCTION

A multi-tenant DCN is a complicated environment, with multiple paths, high traffic rates, a high flow-arrival rate, and a turnover of tenants. The advent of SDN facilitated the growth and operation of DCNs, by automating network configuration tasks that had previously been mostly manual and disjointed, but also further increased the difficulty for a DCN operator of reasoning about and understanding what exactly is happening in the network. Thus, SDN and DCNs have been productive area of study for researchers – who have analysed and described the characteristics of DCNs [1], [2], and have addressed different aspects of using SDN to operate and manage a DCN [3-6].

One aspect that has not been adequately addressed to date is the provision of a way for a DCN operator to investigate the past behaviour of a network at some point in its history – e.g. 'What path did a packet take to get to this location at a particular time?' – or the network's potential behaviour under different conditions – e.g. 'If a switch was to fail at this time, what flows would have been affected?' Answering these types of questions requires the state of the network to be accurately reproduced as it was at a specific point in the past. Furthermore, reproducing the past state of the network requires a historical record, going back for months or even years, of all of the changes to the configuration of the network. In an SDN, the historical record can be compiled by continuously capturing the packets going between a controller and the switches in the network, e.g. packets of the OpenFlow protocol [7]. OpenFlow is a standard protocol used in SDNs for controllers to exchange messages with switches in order to implement the network policy. Capturing and processing the sheer quantity of OpenFlow messages in a historical log going back months or years presents a significant challenge. Other researchers take the approach of measuring and recording statistics to augment a DCN operator's knowledge of conditions in their network. However, recorded statistics are not sufficient for reproducing a network to the level of detail necessary to answer the types of questions we plan to support.

In this paper, we describe our work on a controller-independent solution for logging OpenFlow messages between one or more SDN controllers and switches in a DCN, to produce a historical log of OpenFlow messages for an extended duration of time. This log is processed to create snapshots of the state of the network at intervals through the historical time covered by the message log. Snapshots and log together are subsequently used for the purpose, on request, of reproducing the network as it was at any specific time (in terms of topology and set of flow-rules), with the intention of being able to answer queries posed by a DCN operator. We call our system 'LogSnap'.

Our *key contributions* described in this paper are: *firstly*, a solution to create succinct snapshots of the state of a network (devices, links, OpenFlow rules) using an OpenFlow message log, and *secondly*, using the data in a snapshot augmented with log messages to quickly and efficiently create an emulated copy of a network with the state that it had at any time covered by the message log. Both are achieved with no onus on DCN operators to provide any extra information to the system beyond the OpenFlow messages that it captures itself. Compared to the state of the art, we offer a more comprehensive, accurate reproduction of a network in a historical state, using a non-intrusive, controller-independent system to capture the historical data.

The rest of our paper is structured as follows: In §II, we outline our motivation and research challenges, and the architectural requirements and design goals we set for LogSnap.

Jonathan Sherwin is funded by Cork Institute of Technology.
Cormac Sreenan is funded by Science Foundation Ireland, grant #13/RC2077

§III describes how our implementation addresses the goals and requirements. §IV shows how LogSnap accurately reproduces targeted aspects of a network solely from a log of OpenFlow messages, how our approach of using snapshots shortens the time taken to recreate a network, and how snapshot storage space must be traded off against the maximum time to reproduce a network. §V describes related work, and in §VI we summarise our work and outline our plans for building on LogSnap.

II. Motivation, Challenges, Architecture, and Design

A. Motivation

Multi-tenant DCNs operate under extensive and frequently-updating policies. These policies define what host devices are allowed to communicate with each other, and where traffic is allowed to go. In an OpenFlow-based SDN, a changed policy will result in changes to OpenFlow rules, which represent part of the network state. When a problem occurs, by the time the DCN operator has an opportunity to address it, the network state most likely has changed, and it may never be possible to identify the root cause of the problem, or the conditions under which it occurred, without a means (e.g. a tool) to provide a view of the network state at and around the time of the problem.

Understandably, DCN operators do not want a tool that will increase their workload by requiring constant monitoring, or require changes to switch or controller configurations to accommodate the tool. Nor do they want a tool that consumes a significant amount of resources. We have identified an opportunity for a system that works efficiently and unobtrusively, gathering information that is already present in a network (OpenFlow messages), but that is usually not recorded. Using a log of this information, we can rebuild the state of the network at any time, to provide a platform for querying, investigation and problem-solving. We have opted to use OpenFlow messages as the basis for our historical log as OpenFlow is the mostly broadly-used open standard SDN protocol, but we believe our design applies to, and our implementation could be adapted for, any control protocol used between SDN controllers and network devices.

B. Research Challenges

Designing and implementing the system presents some interesting challenges and scope for a novel contribution:

- How can we devise a scheme for passive, accurate recording of the history of OpenFlow messages exchanged on a network?
- How can a network be reproduced in a resource-efficient manner?

C. Architectural Requirements

The key requirements that our system must address are:

- Capture of OpenFlow messages from a network, with timing information for each message observed.
- OpenFlow message storage for later analysis / retrieval.
- Analysis of the stored messages to create snapshots of the network (detailed record of topology and flow table state) at specific points in time.

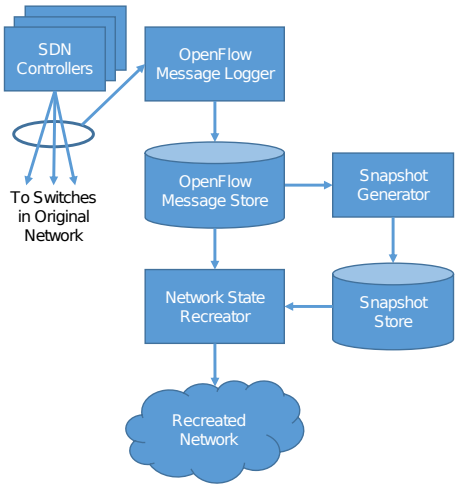

Fig. 1: Architecture Diagram

- Storage of snapshots for later retrieval,
- The ability to quickly create an emulated copy of the network at any required point in time.

These requirements can be realised with components of an architecture as illustrated in Fig. 1, and described below:

OpenFlow message logger. This component must un-intrusively capture and log OpenFlow messages. All OpenFlow messages exchanged between controller and switches should be captured. However, the logger must detect if messages have been missed, log the event, and continue capturing and logging.

OpenFlow message store. Every captured OpenFlow message must be stored in full, with the associated timestamp. The store must be searchable by timestamp, by date range, and by OpenFlow message attribute.

Snapshot generator. A snapshot is a detailed record of the observed topology of the network, i.e. the hosts, switches, host-to-switch links, inter-switch links, and the contents of the switch flow-tables at the time of the snapshot. The snapshot generator must operate solely by processing the OpenFlow message log.

Snapshot store. The snapshot store must be searchable by time and date, returning the closest snapshot at or after that time.

Network state recreator. Using a snapshot augmented with messages from the OpenFlow message log, this component must create an emulated network mirroring the topology and flow-table state of the original network from which OpenFlow messages were logged, for a specified time and date.

D. Design Goals

The logging and snapshot system is intended for use in a DCN. With that in mind, these design goals must be addressed:

1) Scalability: A DCN contains potentially hundreds of switches, connecting thousands of servers with each other, with the Internet, and possibly with other data centres. Our logging system must capture OpenFlow messages exchanged between switches and controllers. Controllers address scalability by

clustering – running multiple instances of the controller, and partitioning the set of switches so that each controller instance only has to deal with a manageable number of them. Furthermore, the number of interactions between controller and switches is often reduced by having the controller proactively configure most flow-rules rather than doing so reactively. The proactive approach means that the controller does not need to be consulted each time a new flow arrives at a switch, only when a flow arrives for which it does not have a matching proactively configured rule. In a multi-tenant DCN, rules are likely to be proactively installed or removed when a new tenant arrives, when an existing tenant leaves or changes their set of leased resources, or when an existing tenant's resources must be redistributed within the DCN. Our system must be designed to be scalable, but also to accommodate features of controllers intended to improve their scalability.

2) Controller- and switch-independence: Logging should work independently of which controller is in use within the DCN. This rules out the use of a controller-based application to log OpenFlow messages, or use of non-OpenFlow information, e.g. through the use of controller APIs. Logging should not depend on which type(s) of switches are in use. We restrict our system to an environment where all controllers and switches use OpenFlow, however there are different versions of OpenFlow, and not all features are implemented universally.

3) Fast reproduction of network state: In order to make the system usable for a DCN operator, the time taken to reproduce the network should be reasonable. We envisage that our system could be recording OpenFlow messages for weeks, months, or even years, and that a DCN operator could require the network state to be replicated for any point in that history in order to query some aspect or behaviour of the network. With message logs stretching over months or years, clearly it will take some time to generate the state information to replicate a network as it was at a specific date and time. However, the time taken should be deterministic – not simply proportional to the difference between the date and time at which OpenFlow message recording started and the desired date and time for which to reproduce the network state.

4) Passive operation: Logging should not require any change to the setup or configuration of the controller or switches, and should have minimal impact on their operation. E.g. a proxy-based design would require controller and switch configuration to route OpenFlow messages through the proxy, and risk the proxy becoming a bottleneck as the network scales.

III. Implementation

We implemented LogSnap with the software components shown in Fig. 2 and outlined below, to meet the architectural requirements and design goals identified in the previous section.

A. Logging Subsystem

Our OpenFlow Packet Filter uses the libpcap packet-capture library [11] to passively capture packets containing OpenFlow messages as layer 2 frames, achieving our design goal of *controller- and switch-independence*. Captured frames are

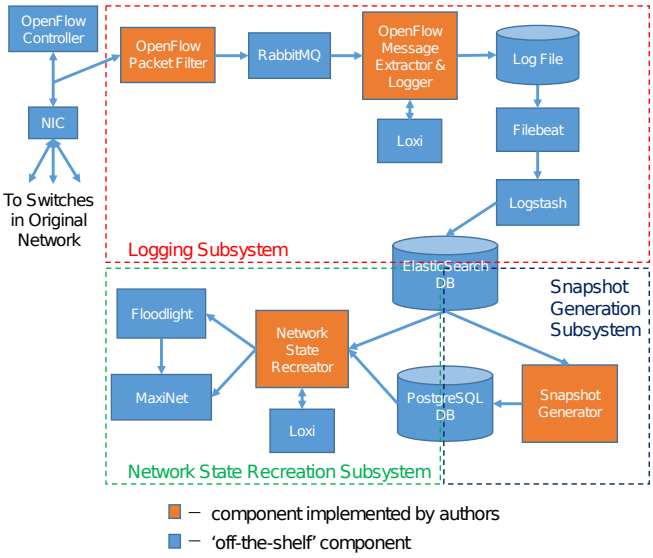

Fig. 2: Implementation Diagram

tagged with their capture time, and enqueued to a RabbitMQ [8] message queue for subsequent parsing and further processing by the OpenFlow Message Extractor & Logger. Separating capturing from logging has several benefits: Only the OpenFlow Packet Filter needs to run on the same physical or virtual host as the OpenFlow controller, minimising use of resources that might be needed by the controller, and supporting our design goal of *passive operation*. The clustered controller scenario mentioned earlier can be leveraged for our design goal of *scalability* by deploying an OpenFlow Packet Filter instance for each controller instance. Our implementation accommodates a single controller or a cluster of controllers running on a single physical server. Extending the solution to support distributed controllers would require time-synchronisation between servers, for example by using the Precision Time Protocol (PTP).

Recent revisions of the OpenFlow Switch Specification (e.g. [29]) state that "The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP." The authors of [30] provide reasons for not using TLS in a DCN. Our implementation currently assumes TLS is not in use, however it could be extended to decrypt captured packets if necessary using the DCN TLS keys, as can be done with Wireshark, for example.

With a passive capture approach there is a risk of packets being missed at very busy times, with no option to request packet retransmission, or to exercise flow-control. Since a missed OpenFlow packet could seriously impact accurate reproduction of network state, we stress-tested our capture filter code by generating packets using the D-ITG traffic generator [28]. In our test environment, the filter reliably captured packets at 355 Mb/s, more than an order of magnitude greater than the data rates seen in the verification experiments described in §IV. While we have confidence in our packet filter, nevertheless it detects if a packet has been missed and the event is logged in the message log in order to subsequently notify the DCN operator.

A DCN operator should be aware that hardware OpenFlow switches vary in how quickly they update their flow-tables under heavy load [31], although the high-end switches more likely to

be used in DCNs perform best. LogSnap could be extended to take switch performance into account – this would require a profile for each type of switch used. An alternative approach, proposed in our previous work [32], leverages a software switch (Open vSwitch, which is immune to the update issues) to mask the problems of the hardware switches in the original network.

The OpenFlow Message Extractor & Logger reassembles the TCP stream for each switch-controller connection from the captured frames, and extracts OpenFlow messages from the streams. Each message is parsed using the Loxi OpenFlow protocol library [9] before being serialised and logged to a text-encoded log file. A single instance of the OpenFlow Message Extractor & Logger process can be used, or as many as one instance per switch, for the purpose of *scalability*. Each instance will emit messages to a separate log file. A new line added to a log file by the Extractor & Logger describes a single OpenFlow message. Filebeat ships each line to LogStash for processing and storage in ElasticSearch (Filebeat, LogStash and ElasticSearch are part of the Elastic Stack family of projects [10]).

B. *Snapshot Generation Subsystem*

The Snapshot Generator queries Elasticsearch for OpenFlow messages, and processes them in chronological order. As it processes the messages, it learns about the topology of the network that the messages came from. It learns about the existence (and continued existence) and identities of switches and controllers from the messages that they exchange with each other. It learns about links between switches from the messages generated by the topology discovery process of the controller, e.g. LLDP packets contained in PACKET_IN messages. It learns about hosts, and how the hosts are connected to switches from ARP request packets contained in PACKET_IN messages. The Snapshot Generator also maintains a flow-table for each switch. As FLOW_MOD messages are processed, rules are added, modified or deleted from the relevant flow-table. Table I lists some types of OpenFlow messages that contain key information used by the Snapshot Generator.

Snapshots, written to a PostgreSQL [12] database, consist of the set of controllers, switches, hosts, links, and flow-table contents that were known and valid at the time of the snapshot. A new snapshot is written when a certain number of OpenFlow messages have been processed since the last snapshot. This number is an operator-configurable parameter. Setting it lower reduces the maximum time taken to reproduce a network, but increases storage requirements because snapshots are recorded more frequently. The trade-off between reproduction time and snapshot storage requirements is explored further in Section IV.

C. *Network State Recreation Subsystem*

A data centre operator uses the Network State Recreator to replicate an original network for a specific date and time covered by the logged set of OpenFlow messages. The Network State Recreator searches the PostgreSQL DB for the closest snapshot at or before the specified date and time. If there is a snapshot with that exact timestamp, then that is all the information required to recreate the network. If there is a difference between the snapshot timestamp and the specified time and date, then the Elasticsearch DB is queried for all OpenFlow messages with timestamps later than the snapshot timestamp but earlier than or matching the specified time and date, and the snapshot data is updated to represent the state of the network at the later point in time. In this way, we achieve our design goal of *fast reproduction of network state* - the system can quickly replicate the original network at any date and time without having to start from the very beginning of the OpenFlow message history.

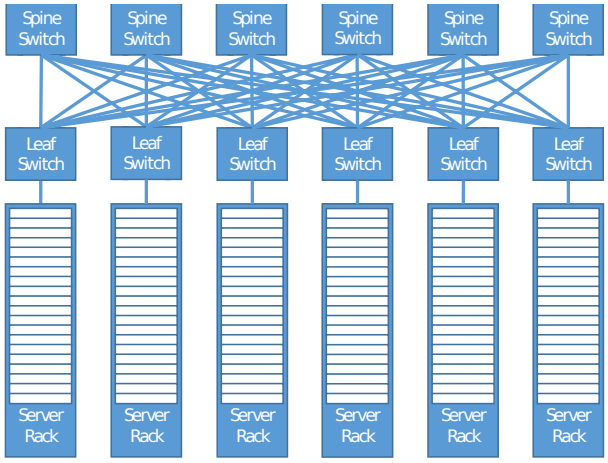

Fig. 3: DCN Topology

Of course, it is quickest to reproduce the network state for a date and time that matches the timestamp of a snapshot, and slowest to reproduce the network state for a date and time just before the timestamp of a snapshot. This is illustrated and considered further in Section IV.

The network is reproduced with MaxiNet [14], a distributed extension of the Mininet [15] network emulator. MaxiNet allows us to create very large networks (towards our goal of *scalability*) in emulated form. To populate switch flow-tables, we use Floodlight [13], modified to act purely as a delivery mechanism for flow-rules captured from an original network. Any OpenFlow controller could be adapted for this purpose, but Floodlight was a natural choice as it uses the same Loxi protocol library used in our Logging Subsystem.

TABLE I. INFORMATION IN OPENFLOW MESSAGES

OpenFlow Message Type	Encapsulated Information
FEATURES_REPLY	Switch DPID
STATS_REPLY	Switch ports
PACKET_IN	Inter-switch links; host-switch links
FLOW_MOD	Flow-rule additions / modifications / deletions

IV. EVALUATION

A. *Test Environment*

We devised a DCN topology for which to test LogSnap, illustrated in Fig. 3. This network consisted of six racks of twenty hosts and one top-of-rack switch each. The six top-of-rack (also referred to as leaf) switches were connected in a folded-clos topology to six spine switches to create a spine-leaf network architecture, as is commonly used in a DCN.

The use of MaxiNet described so far has been to host the recreated network. However, for our experimental testing we

also used MaxiNet to first host the original network – an emulated network design to reflect the topology and conditions of a busy DCN. This allowed us to capture OpenFlow messages, and also to verify that LogSnap accurately recreates a network from the log of OpenFlow messages.

Our test environment consists of three physical machines, connected via two Ethernet switches. Three MaxiNet worker VMs were hosted on a single server (Intel Xeon E5506, 2.13GHz, 8 cores, 32GB RAM). The worker VMs ran Ubuntu Server 18.04, with MaxiNet 1.2, Mininet 2.2.2, and Open vSwitch [16] 2.9.0 installed. A PC (Intel Core 2 Quad Q9400, 2.66GHz, 4 cores, 8GB RAM) was used to control experiments, host a controller VM, run the MaxiNet FrontEnd server, RabbitMQ 3.7.8, and Filebeat 6.3.1. Depending on the stage of an experiment, it also ran the OpenFlow Packet Filter and OpenFlow Message Extractor & Logger, or the Snapshot Generator, or the Network State Recreator. A second PC (Intel Core 2 6400 2.13GHz, 2 cores, 4GB RAM) hosted a database VM running Ubuntu Server 18.04, with Elasticsearch 6.4.3, Logstash 6.4.3, and PostgreSQL 10 installed. The three physical machines each had two NICs, each of which was connected to a Gigabit Ethernet switch. One of these networks was used for control of experiments, and communication with databases, the other solely for the exchange of OpenFlow messages between controller and switches.

DCT2Gen [17] generates traffic based on profiles created from DCN traces. To verify LogSnap, we use a test profile and traffic generator made available by the DCT2GEN authors. The profile contains details of 3667 TCP connections that the traffic generator normally schedules over a 10 second period between 120 hosts (arranged evenly in six racks). The traffic generator can be configured with a time-dilation factor to compress or expand the time over which the TCP connections are scheduled.

In our original network, we use ONOS [18] 1.13, but any OpenFlow controller could be used. ONOS and OpenDayLight are the two controllers most likely to be found in a DCN. As mentioned earlier, we expect mostly pro-active forwarding to be used in a multi-tenant DCN, and the arrival and departure of tenants or modification of tenant resource requirements to trigger the majority of OpenFlow rule changes. Since we do not have traces or configurations from a multi-tenant DCN, we have reactive forwarding enabled in ONOS to reflect the level of rule changes that we would expect in a large, multi-tenant DCN.

B. Testing Approach

To evaluate LogSnap, we devised two different types of test. Firstly, we needed to verify the correct functioning of the system, by showing that a recreated network matches the original network. This is important because the recreated network is generated solely from the information gleaned from OpenFlow messages, and not, for example, from any externally provided model of the network. Secondly, we show the benefit of snapshots in setting an upper bound to the time it takes to compile the state information required to recreate the network.

C. Testing - Functional Verification

To provide confidence in the correct functioning of LogSnap, we needed to verify that a recreated network matches the original network for a number of fundamental conditions.

TABLE II. LOGSNAP FUNCTIONAL VERIFICATION

Network Elements Checked	What was confirmed between Original and Recreated Network
Switches	Number of switches; switch DPIDs
Links between switches	Pairs of DPIDs; port numbers
Flow Tables	Number of flow-rules on each switch; flow-rule content
Hosts	Number of hosts; host IP addresses
Host Connectivity	ICMP success or fail between host pairs

In order to verify that the original and recreated networks were the same, we ran experiments gathering extra information from the original network at a specific point in the experiment while our Logging Subsystem captured and logged OpenFlow messages. Recreating the network for that specific point in time, we collected the same set of information again, and compared with the first set.

ONOS' REST API was used to obtain the list of inter-switch links, hosts and host-switch links that ONOS learned about on the original network. Open vSwitch's ovs-ofctl tool gave a dump of the flow-table from each switch on the network. Linux fping was used to test connectivity between all pairs of hosts on the network. Note that connectivity between a pair of hosts depends on whether flow-rules have been installed to allow packets flow between the hosts. Our verification check was not that all host pairs tested could communicate, rather that the result of a connectivity test between a pair of hosts in the recreated network was the same as the result of a connectivity test between the same pair of hosts in the original network (whether that result was a pass or a fail). On a network containing 120 hosts, this generated 14280 results on the original network to compare with the results of the same tests on the recreated network. As an example, the output of one set of connectivity tests on an original network showed connectivity between 48 host-pairs. Repeating the tests on a recreated network gave the same results.

The modified Floodlight controller used to populate flow-tables on a recreated network also has a REST API, through which a list of inter-switch links, and a list of hosts and host-switch links was extracted. The output was compared with the data collected from ONOS. We used ovs-ofctl and fping as we had done on the original network, and compared those results.

Each verification experiment ran with traffic generated for 120 seconds, split into four 30-second intervals. At the end of a 30-second interval, the traffic generator was stopped to free up resources on the MaxiNet workers to allow us to gather information. At the start of the next interval, the traffic generator started again, but with TCP connections shifted between racks. This was to ensure that the same set of flows was not running unchanged in each interval. Between traffic generation intervals, we quiesced the controller, waited for switches to deal with any outstanding FLOW_MOD requests, and gathered switch flow-table contents and topology and connectivity information – this sequence of actions took around 2 minutes. Since there were four traffic-generation intervals, in each experiment we had three verification points. For the full duration of the experiment, our Logging Subsystem was capturing OpenFlow messages. Having run the Snapshot Generation Subsystem on the log of

captured messages, we then recreated the network separately for each verification point, and gathered and compared information.

The set of verification checks is listed in Table II. The results we recorded over many cycles of experiments confirm the correct operation of LogSnap, although the results of our host checks initially gave us pause: while a recreated network consistently had the same number of switches as the original network, sometimes it had fewer hosts. This happens more often on a network that was recreated for a time close to the startup-time of the original network. The difference is because a host is only learned of at the control plane when that host actively sends data or has a packet addressed to it.

D. Testing - Demonstrating the Benefit of Snapshots

For a large DCN, with a sizeable number and a turnover of tenants, the OpenFlow message history well be of significant volume. Recreating a network from that history can be time, memory, and compute intensive. Fig. 4 compares how long it took to recreate a network for different points in its history with and without the facility of being able to build the network state starting from a snapshot.

Clearly, from the figure, the time to recreate a network without the use of snapshots increases for later network recreation times. It grows almost linearly with the number of OpenFlow messages that need to be processed in order to generate the state of the network at the required recreation time. It is not exactly linear as some OpenFlow messages require more processing than others - e.g. PACKET_IN messages are fully parsed because they may contain information about the network topology, FEATURES_REQUEST messages are only taken as confirmation that the controller that sent the message still exists.

Conversely, there is an upper limit on how long it takes to recreate a network starting from a snapshot. The limit is at the point just before the next snapshot, and so depends on the frequency of snapshots. In these experiments, a snapshot was taken every 75K OpenFlow messages.

The results shown in Fig. 4 were based on a message log containing approximately 500K OpenFlow messages. The log is of roughly three minutes of data. Fig. 5 shows the number of

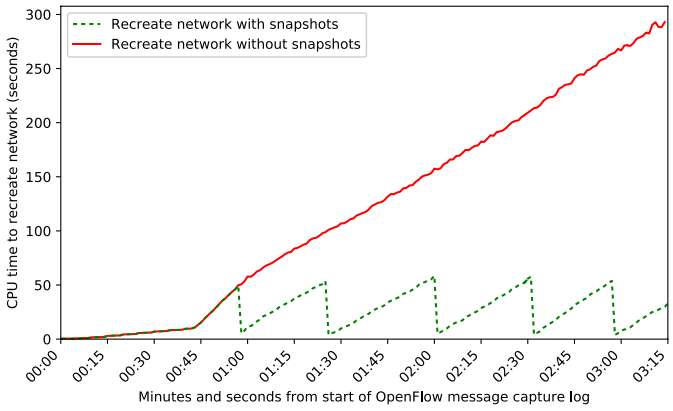

Fig. 4: CPU time required to recreate a network as it was at a specific time

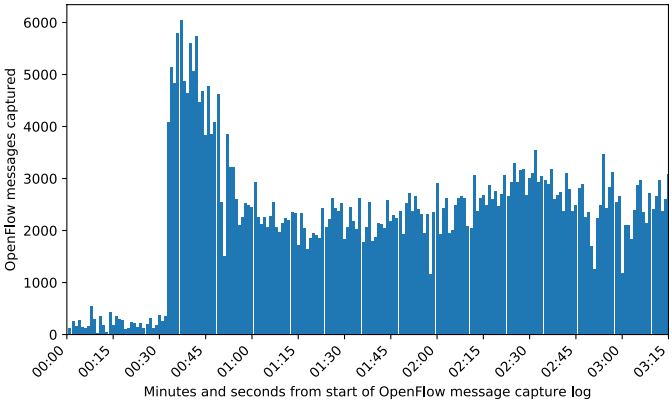

Fig. 5: OpenFlow messages captured per second by capture filter

OpenFlow messages logged each second, generated based on the message timestamps. Traffic generation started about 45 seconds after message logging began, as can be observed by the spike to ~6K OpenFlow messages per second: the spike consists largely of ARP request packets in PACKET_OUT messages sent by the controller to switches, to determine the locations of destination hosts that have not been learned of previously.

E. Message Log and Snapshot Storage Requirements

The space required to store the message log relates to the number of messages in the log, although not all messages are the same size. Our 3 minute ~500K message log took 318MB of space in Elasticsearch, without any optimisation to reduce disk usage. Extrapolating, this grows to 153GB per day, or 56TB per year.

Storage requirements for snapshots can be calculated from some key parameters. A snapshot consists of information about controllers, switches, hosts, links, and the flow-rules for each switch. On any given network, the number of controllers (C), switches (S), hosts (H), and links (L) is known and is reasonably static – with infrequent changes due to DCN maintenance or reconfiguration. The set of flow-rules is more dynamic, depending mostly on how many traffic flows are active (F) at the time of the snapshot. Most flow-rules are only configured on switches on the path that packets need to follow to get from source to destination. According to [1], approximately 75% of Cloud DCN traffic is rack-local, so packets only need to traverse one top-of-rack switch to get to their destination. The remaining 25% of traffic must traverse three switches (two leaf and one spine) in our network to reach their destination. Assuming that the storage requirement for one device (controller, switch or host) or link is represented by a mean constant value J, and for a flow-rule is represented by a mean constant value K, we can calculate the storage requirement B for a single snapshot as:

$$B = J * (C + S + H + L) + K * (F * 0.75 + F * 0.25 * 3)$$

In our test network, with $C = 1$, $S = 12$, $H = 120$ and $L = 156$, approximately $F = 100$ flows active at one time, and with $J = 235$ bytes and $K = 760$ bytes (figures calculated from our experimental data), this yields an estimated storage requirement $B = 182$ KB for one snapshot.

The number of snapshots per day depends on the total number of OpenFlow messages that will be captured in one day, which in turn depends on the size of the network (controllers, switches, hosts, links) and on either the flow arrival rate and mean flow duration in a reactive OpenFlow network, or on the rate of policy changes in a proactive OpenFlow network. Taking the figure of 500K messages in 3 minutes mentioned earlier: If the snapshot interval is 75K messages, then a snapshot will be taken approximately every 27 seconds, resulting in 3,200 snapshots in 24 hours and requiring approximately 583 MB of storage for that day of snapshot data.

Reducing the snapshot interval increases the number of snapshots taken. E.g. halving the snapshot interval doubles the number of snapshots, and intuitively this should result in double the storage requirement. However, we have applied an optimisation to the snapshot database which reduces duplication between snapshots. Our approach is to track the range of time over which we have known about elements of the network – switches, hosts and links, for example. When an element is recorded in the database (as part of writing a snapshot), the time-range is recorded in the row for that element as a field of PostgreSQL's timestamp range (TSTZRANGE) data-type. If the element is still in the network at the time of the next snapshot, and has been there uninterrupted for the duration, the time-range field is updated rather than a new row being written. Reading a snapshot from the database is then a matter of searching for rows with time-ranges within which the snapshot time and date falls. We have observed, as a result of this optimisation, that halving the snapshot interval causes a 50% increase in the snapshot storage requirement.

V. Related Work

SDN data-plane activity monitoring and logging tools (e.g. Planck [19], SketchVisor [20]) generally either sample data-plane packets, or simply gather statistics. The data they collect does not describe the topology of the network, or reveal the forwarding logic of network devices. NetSight [21] is of particular note, recording complete packet histories for debugging purposes. A packet history is a record of the path followed by a packet during its lifetime, and could be used to at least partially infer the topology and forwarding logic of a network, if collected for all packets. However, NetSight is not passive, requiring extra work to be done by switches, and imposes a proxy between switches and controllers.

Several papers have been published on systems to monitor the control-plane in OpenFlow networks. In common with data-plane tools, though, most of these (e.g. OFMon [22], OpenNetMon [23], OpenTM [24]) gather statistics, rather than recording OpenFlow messages.

Logentries [25] collects both data-plane and control-plane log data for central storage. However, it is used for event logs such as those emitted by controllers and switches as part of their normal operation, rather than a log of OpenFlow messages.

Most production-quality SDN controllers do not offer a built-in facility to record OpenFlow messages, with the exception of Floodlight, through its PacketStreamer module. Messages can be filtered (e.g. by OpenFlow field values) before being streamed via a brokered message service to a waiting client. PacketStreamer is a Floodlight-specific module. The ONOS controller offers statistics, through its CPMan app. Similarly, OpenDaylight produces statistics via its Time Series Data Repository project. A non-intrusive approach has benefits over approaches requiring integration with a specific controller.

OFRewind [26] provides some functionality similar to LogSnap: recording and replay of OpenFlow control-plane traffic in (in their case) a campus network. OpenFlow message recording is effected via a proxy between switches and controller(s), i.e. it is not a non-intrusive solution like ours. Furthermore, OFRewind does not recreate a network from the recorded data - the target network must be provided by the network operator: either the original physical network, or an emulated network created by some means not integral to OFRewind. OFRewind guarantees to maintain the order of OpenFlow messages as they were recorded, it does not record the actual timestamps. Lastly, OFRewind does not create regular snapshots to speed up reproduction of network state for a particular time.

ForenGuard [27] logs and monitors OpenFlow messages to identify root causes of forwarding problems in an SDN network. ForenGuard is controller-specific, implemented on top of the Floodlight controller.

In comparison, LogSnap provides this unique combination of features, which makes it particularly attractive to a DCN operator looking for a passive solution for post-hoc analysis of network configuration:

- It is designed to work with any OpenFlow controller.
- OpenFlow messages are captured passively, with no impact on controller and switch communication.
- Automated reproduction of the original network, using a network emulator and topological data extracted from the OpenFlow message history.
- Fast reproduction of the original network for any time covered by the OpenFlow message history, based on snapshots created from that history.
- Complete sets of rules generated for each switch in the network at a particular time, representing the forwarding logic of that network at that time.

VI. Conclusions and Future Work

We have shown that it is possible to recreate the topology and state of a DCN in an emulated environment, by only using a complete historical log of OpenFlow messages captured from the original network, and we designed and implemented a solution for the task. The DCN topology and state can be recreated for any time and date that falls within the period of history covered by the historical log. We presented LogSnap, the solution we designed and implemented to non-intrusively log OpenFlow messages, create snapshots, and recreate a network. We illustrated the results of our experiments verifying that LogSnap correctly recreates a network, and showing the benefit of snapshots in capping at a deterministic value the time taken to recreate the network. Since the maximum time to recreate a network depends on the frequency of snapshots, we quantified the cost of more frequent snapshots versus their storage

requirements, as guidance for a DCN operator who may want to cap the network recreation time at some preferred value. Compared to the state of the art, LogSnap offers a more comprehensive, accurate reproduction of a DCN in a historical state, using a non-intrusive approach to capture historical data.

The next phase of our research is to build on the foundation provided by LogSnap. We are in the process of extending the system by adding a query engine. This will provide a mechanism for a DCN operator to investigate the historical record and uncover insights into the past behaviour of the constituent parts of the network. The query engine will use the OpenFlow message log, snapshots, and recreated networks, individually or combined, to answer queries as appropriate.

REFERENCES

[1] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC, 2010, pp. 267-280, doi: 10.1145/1879141.1879175.

[2] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," presented at the Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, 2015.

[3] T. Cucinotta, D. Lugones, D. Cherubini, and E. Jul, "Data centre optimisation enhanced by software defined networking," in IEEE International Conference on Cloud Computing, CLOUD, 2014, pp. 136-143, doi: 10.1109/CLOUD.2014.28.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM'11, 2011, pp. 254-265.

[5] D. Arora, T. Benson, and J. Rexford, "ProActive routing in scalable data centers with PARIS," in DCC 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Distributed Cloud Computing, 2014, pp. 5-10.

[6] P. Peresini, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, Fine-Grained Data Plane Monitoring," in CoNEXT'15 - Proceedings of the 2015 ACM International Conference on Emerging Network Experiments and Technologies, 2015.

[7] N. McKeown et al., "OpenFlow: enabling innovation in campus networks," SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69-74, 2008.

[8] RabbitMQ Open Source Message Broker. [Online]. Available: https://www.rabbitmq.com/

[9] Loxi OpenFlow Protocol Library. [Online]. Available: https://github.com/floodlight/loxigen

[10] Elastic Stack Open Source Distributed Log Management. [Online]. Available: https://www.elastic.co/products/

[11] libpcap network traffic capture library. [Online]. Available: https://www.tcpdump.org

[12] PostgreSQL Open Source Database. [Online]. Available: https://www.postgresql.org/

[13] Floodlight Open Source OpenFlow Controller. [Online]. Available: http://www.projectfloodlight.org/

[14] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl, "MaxiNet: Distributed Emulation of Software-Defined Networks," presented at the IFIP Networking Conference (Networking 2014), 2014.

[15] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in Proceedings of the 9th ACM Workshop on Hot Topics in Networks, Hotnets-9, 2010.

[16] B. Pfaff et al., "The design and implementation of Open vSwitch," in Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015, 2015, pp. 117-130.

[17] P. Wette and H. Karl, "DCT2Gen: A traffic generator for data centers," Computer Communications, Article vol. 80, pp. 45-58, 2016.

[18] P. Berde et al., "ONOS: towards an open, distributed SDN OS," presented at the Proceedings of the third workshop on Hot topics in software defined networking, Chicago, Illinois, USA, 2014.

[19] J. Rasley et al., "Planck: Millisecond-scale monitoring and control for commodity networks," in SIGCOMM 2014 - Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication, 2014, pp. 407-418.

[20] Q. Huang et al., "Sketchvisor: Robust network measurement for software packet processing," in SIGCOMM 2017 - Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication, 2017, pp. 113-126.

[21] N. Handigol, B. Heller, V. Jeyakumar, D. Mazi, and N. McKeown, "I know what your packet did last hop: using packet histories to troubleshoot networks," presented at the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, Seattle, WA, 2014.

[22] W. Kim, J. Li, J. W. K. Hong, and Y. J. Suh, "OFMon: OpenFlow monitoring system in ONOS controllers," in 2016 IEEE NetSoft Conference and Workshops (NetSoft), 6-10 June 2016, pp. 397-402.

[23] N. L. M. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," in IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World, 2014.

[24] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic Matrix Estimator for OpenFlow Networks," in Passive and Active Measurement Conference (PAM 2010), 2010.

[25] B. Siniarski, C. Olariu, P. Perry, T. Parsons, and J. Murphy, "Real-time monitoring of SDN networks using non-invasive cloud-based logging platforms," in Personal, Indoor, and Mobile Radio Communications (PIMRC), 2016 IEEE 27th Annual International Symposium on, 2016: IEEE, pp. 1-6.

[26] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: enabling record and replay troubleshooting for networks," presented at the Proceedings of the 2011 USENIX Annual Technical Conference, Portland, OR, 2011.

[27] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards Fine-grained Network Security Forensics and Diagnosis in the SDN Era," presented at the Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, Canada, 2018.

[28] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," Computer Networks, vol. 56, no. 15, pp. 3531-3547, 2012.

[29] OpenFlow Switch Specification Version 1.5.1, Standard. Open Networking Foundation, 2015. [Online]. Available: https://www.opennetworking.org/software-defined-standards/specifications/

[30] P. Goransson, C. Black, and T. Culver, Software Defined Networks: A Comprehensive Approach, 2nd ed. Morgan Kaufmann, 2016, pp. 94.

[31] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches," Computer Networks, vol. 136, pp. 22-36, 2018/05/08/ 2018.

[32] J. Sherwin and C. J. Sreenan, "Reducing the latency of OpenFlow rule changes in data centre networks," in 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), 19-22 Feb. 2018 2018, pp. 1-5.