NetCP: Consistent, Non-interruptive and Efficient Checkpointing and Rollback of SDN

Ye Yu
University of Kentucky

Chen Qian
University of California Santa Cruz

Wenfei Wu
Tsinghua University

Ying Zhang
Facebook

***Abstract*—Network failures are inevitable due to its increasing complexity, which significantly hampers system availability and performance. While adopting checkpointing and rollback recovery protocols (C/R for abbreviation) from distributed systems into computer networks is promising, several specific challenges appear as we design a C/R system for Software-Defined Networks (SDN). The C/R should be coordinated with other applications in the SDN controller, each individual switch C/R should not interrupt traffic traversing it, and SDN controller C/R faces the challenge of time and space overhead. We propose a C/R framework for SDN, named NetCP. NetCP coordinates C/R and other applications to get consistent global checkpoints, it leverages redundant forwarding tables in SDN switches for C/R so as to avoid interrupting traversing traffic, and it analyzes the dependencies between controller applications to make minimal C/R decision. We have implemented NetCP in a prototype system using the current standard SDN tools and demonstrate that it achieves consistency, non-interruption, and efficiency with negligible overhead.**

I. Introduction

Despite that many efforts have been put to add reliability and high availability to networking systems, the performance can still be severely impacted by hardware failures, software errors, software bugs, and configuration mistakes, collectively called *failures* in this paper. Since failures are often inevitable in any large complex systems and online services usually require for high availability (e.g., 99.999% of the time), in the case of failures it is essential to restore the network to a previous working state first instead of fixing problems. For example, on April 11th 2016 in Google Compute Engine (GCE) [3], a network configuration update triggered a management software bug, and wrong rules were generated and installed into network devices, which cascadingly caused a series of network unreachable. The outage was resolved 18min after its appearance by the team "*reverting the most recent configuration changes* made to the network even before knowing for sure what the problem was". 12 hours later, "the team was confident that they had established the root cause as a software bug in the network configuration management software".

Rollback-recovery mechanisms in distributed systems are well-studied. Existing rollback-recovery protocols can be classified into log-based ones and checkpoint-based ones [13]. In the log-based approach, all changes are recorded, and the recovery would replay logs until a consistent state; in the *checkpoint-based* approach, the system periodically records its states during normal operations, where the record is called *checkpoint*, and overwrites states with checkpoints upon failures. As the traffic in modern computer networks tends to be more dynamic and diverse with time (e.g., public cloud, online services), log-based rollback-recovery is costly on storage and computation. On the other hand, Software-Defined Networks (SDN) get more widely deployments with its flexibility and programmability to handle the dynamics and diversity [17], [18]. Thus, in this paper, we focus on *checkpointing and recovering (C/R for abbreviation) SDN*.

In SDN, a centralized controller configures all distributed switches, and switches do not communicate control messages with each other. Without distributed protocols, the SDN architecture makes us naturally choose a *coordinated C/R approach* [13], i.e., the controller decides when and which one of the switches and itself to make a checkpoint or recovery.

Despite the convenience from the SDN architecture, its difference from traditional distributed systems also leads to several challenges when adopting coordinated C/R.

1) Network checkpoints have another level of consistency[1] — cross-update consistency. In details, a *network update* usually has multiple devices involved (e.g., set up a path), when checkpointing multiple devices, we must guarantee all or none of the individual device updates in the same network update are recorded in the multiple checkpoints; otherwise, the network would have configuration errors after recovery (e.g., black holes, loops).
2) The SDN controller has multiple applications control the network in parallel (e.g., routing, topology discovery), each of which makes network updates independently. Since C/R should not overlap in-transit network updates for consistency reasons, it is necessary and challenging to coordinate C/R with updates.
3) During the control plane C/R, switches are still handling network traffic. The C/R should be managed without interrupting switches processing in-flight network traffic.
4) The SDN controller is monolithic; it is necessary and challenging to make an efficient checkpoint of the controller without full snapshot to avoid interruptions

[1]There are another two concepts about consistency, and they are different from cross-update consistency: message consistency in distributed system C/R means the situation where if sender's checkpoint records a message, the receiver's checkpoint also records it [13]; a consistent update for networks means all operations in the update are executed in an order so that network preserves good properties, e.g., loss freedom, congestion freedom [19], [25].

978-1-5386-2542-2/18/$31.00

We propose a C/R framework for SDN, called NetCP. NetCP checkpoints the SDN data plane and control plane periodically, and recovers both planes to a more recent checkpoint when failure happens. In addition to the correctness/consistency guarantee in traditional distributed system C/R, NetCP overcomes the challenges above. NetCP coordinates switches to make consistent global checkpoints; it leverages a locking mechanism to achieve the coordination; SDN switches uses redundant forwarding tables to record and recover updated rules, so that C/R can be managed without interrupting running traffic; and the SDN controller is dissected into applications, and C/R are performed based on dependencies between applications.

We implement NetCP on POX, OpenDayLight (ODL) and Open vSwitch (OVS), and experiment shows that NetCP achieves correct and consistency in SDN C/R, the running traffic is not interrupted during C/R, and the application-level C/R for SDN controller can significant reduce C/R time compared with naive monolithic C/R. The overhead of NetCP C/R is negligible.

Our work is closely related to the recent efforts in SDN debugging [22], [27]. In particular, while some of them focus on replaying data plane packets [16], [33], [35], we concentrate more on C/R of switches and the controller. Moreover, we find that existing work [31] has pointed out the need for a C/R method, which have not been explicitly studied and implemented in the literature.

Overall, we make the following contributions in this work.

1) We are the first to systematically study the feasibility of network C/R for both SDN data plane and control plane. And we address several challenges to implement a C/R system in SDN. The models and protocols proposed in this work may lead to valuable research of network C/R in future.
2) We develop a framework called NetCP for SDN C/R, which includes global consistency mechanisms, non-interruptive C/R design on switches, and efficient controller C/R approaches.
3) We address a number of important problems, including maintaining TCP connections to switches during checkpointing, control plane rollback following application dependencies, composition/decomposition of updated rules, and speedup of the rollback process.
4) We demonstrate the feasibility of NetCP by implementing it on ODL, POX and OVS and evaluation on mininet.

The rest of this paper is organized as follows. We first give an overview of the system design in Section III, followed by the detailed description of global C/R coordination in Section IV. And then we describe the C/R design for switches and the controller in Section V and VI. Section VII presents the evaluation results. Finally, we discuss related issues in Section VIII and conclude this work in Section IX.

II. Related Work

Checkpointing and rollback for computing systems have been extensively studied in distributed systems. In [13], Elnozahy et al. classified rollback protocols into log-based approaches and checkpoint-based ones, and in [23], Koo et al. elaborated the design of a checkpoint-based rollback protocol. In [26], Sancho described how to incrementally checkpoint a distributed system. However, as we described in Section I, SDN network has three unique requirements and challenges which cannot be straightforwardly solved by the traditional approaches: the cross-update consistency among all switches' checkpoints, the non-interruption requirements to the running network traffic, and the time/space efficiency requirements to checkpoint/rollback an SDN controller.

For SDN, our work is the first to discuss C/R for both network data plane and control plane collaboratively from an academic viewpoint in detail. A set of works focus on individual nodes in the whole network, for example, Ravana [20] and LegoSDN [11] provide a fault-tolerance mechanism for the controller only, and FTMB [29] focused on the C/R mechanism for individual middleboxes. In SDN troubleshooting works, the solutions usually have the capability to snapshot or record network states. For example, SDN verification solutions [21], [27], [32], [34] snapshot the device rules for verification. All these solutions fail to consider the cross-update consistency requirements and do not provide non-interruptive rollback mechanisms.

More closely related works to ours are the recent proposals on network-wide record and replay. In particular, Handigol *et al.* proposes ndb [16] for debugging SDN data plane, which allows programmers to set breakpoints for packets and backtrack the forwarding history.

OFRewind [33] records and replays network events in a hypervisor, which focuses on filtering packets to improve scalability. HotSwap [31] is the most relevant work to ours. It solves the controller upgrade problem by replaying the recorded network events from the initial state of a new controller. They did not directly save any forwarding states of the switches. LegoSDN [11] provides a re-design of the controller architecture in SDN to enhance the liability and fault tolerance in SDN control plane using similar checkpoint and rollback techniques. However, different from NetCP control plane, which enables both per-application and full-controller checkpoint, LegoSDN targets to support SDN checkpoints in application level. Meanwhile, LegoSDN requires applications to communicates with each other under the proposed I/O protocol, which may introduce high overhead for inter-application communication and requires modification on both the application and controller source code.

III. NetCP Overview

In this section, we discuss the necessity of C/R capability in SDN, requirements of system design, NetCP design overview, and the difference between NetCP and existing work.

A. Motivating examples

C/R is an essential capability for SDN, it reduces the risk of network outage and improves network availability. We list a few scenarios where networks benefit from C/R.

Fast failure recovery. In the case of network failures (e.g., switch down, OpenFlow channel connection interruption), operators are required to minimize network outage time. So it is important that the network state is recovered to its intended operational state as quickly as possible. In most implementations today, after such failures, the rules in a switch flow table are removed and later relearned gradually by querying the controller.

Misconfiguration correction. Software bugs and misconfigurations in manual management are inevitable, causing violations in networks such as black holes and loops. Some mis-configurations source from human operators. They may cause no network performance downgrade or affect no network invariants and can not be easily detected by verification or troubleshoot tools, until the configuration is deployed and misbehaved packets are observed. Incrementally fixing problems after they happen usually makes the system more complicated and error-prone. Thus, a proper C/R mechanism is necesssary to provide backups before any deployment or configuration.

Policy configuration changes. Modern computer networks are usually updated frequently to accommodate various dynamic network traffic (e.g., online services and public cloud [17], [18]). Coordinate new and existing policy is still an ad-hoc and manual process. Similarly, NetCP can checkpoint the network periodically, so if policy has conflict, it provides a way to eliminate the newly-added policy.

B. Design requirements

In the environment of SDN, NetCP should satisfy the following requirements.

- **Cross-update Consistency.** As discussed in Section I, a network-wide checkpoint consists of all individual switches' checkpoints. For any network update (e.g., set up paths), all or none of operations on individual switches in this update is recorded in the network-wide checkpoint; an update should never be partially recorded.
- **Non-interruption for network traffic.** Online service provider usually requires their service to be online 7-24, and network traffic also exists with services. Thus, when NetCP performs C/R, traffic in the network should not be dropped and services should not be interrupted.
- **Time/space efficiency for C/R.** NetCP should perform C/R quick enough so that the outage duration can be reduced, and availability can be guaranteed (e.g., 99.999% of the time). NetCP has better reduce storage space for checkpoints, as storing/loading them into/from the persistent storage system (e.g., disk) usually takes time.
- **Timeliness of the checkpoints.** Newer checkpoints record more recent network states, and recovery from new checkpoints can avoid losing too much previous computation and results. Thus, NetCP should checkpoint the network in a timely manner.

C. Design Overview

As shown in Fig. 1, NetCP consists of four parts, (1) a switch coordinator works as an application in the SDN controller, which decides when and which switch perform C/R, (2) a controller C/R executor stands aside of the controller, which is specifically in charge of the C/R of the controller itself, (3) each application in the controller is modified so that itsnetwork update operations can be compatible with the network C/R in the switch coordinator, and (4) each switch has its routing tables organized friendly for C/R and has a local agent to execute the C/R operation.

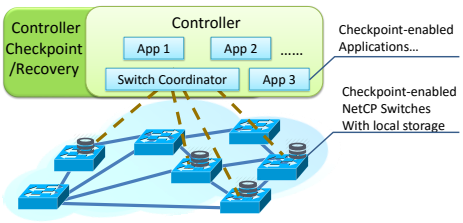

Fig. 1. Overview of NetCP

Global Coordination. The switch coordinator communicate with the agent on each switch to control the C/R of each switch. It issues checkpoint commands either immediately after each network update or periodically, depending on the timeliness requirements on the checkpoints. To achieve cross-update consistency, a checkpoint command is executed only between two network updates. This is achieved using a locking mechanism in both the switch coordinator and all other modified controller applications, with which C/R never overlaps with network updates.

Switch C/R. NetCP checkpoints a switch by taking a snapshot of the flow table in the switch. Due to the scalability issue (storage, recovery time), we do not choose to store all switch rules in the controller; instead, NetCP makes each switch responsible for checkpointing its state. To make the local C/R on each switch efficient for time and storage space and non-interruptive for network traffic, NetCP organizes the table in each switch into three groups, one for routing, one for incremental checkpointing, and one for recovery (see Section V for details), and its algorithm for C/R based on the three-group tables can achieve both efficiency and non-interruption properties.

Controller C/R. NetCP provides two approaches to checkpoint the controller. One approach is full controller C/R, which leverage Linux process snapshot and recovery. A checkpoint of the controller includes the controller process's address space and the register states; during its recovery, the new process is spawned which initializes its address space from the checkpoint file and resets its registers. This function is essential for the control plane to recover from significant failures such as a crash of the entire controller program.

The second approach is application-level C/R. It makes an anatomy of the applications (implemented as threads) in SDN controller, finds out dependencies between applications, checkpoints critical states, and recovers only failed applications (avoiding rollbacking the entire controller).

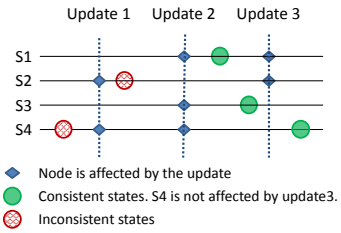

Fig. 2. Consistent vs. inconsistent checkpoints

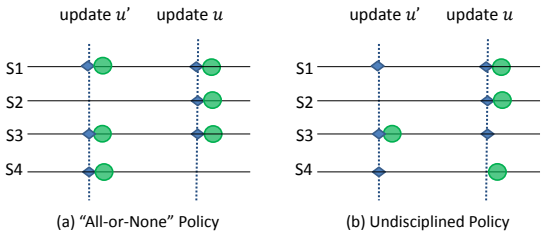

Fig. 3. "all-or-none" policy reduces occurrence of the domino effect.

IV. Consistent SDN C/R

In this section, we present how NetCP performs C/R for the whole SDN data plane. During C/R, cross-update consistency is preserved by making network updates non-overlap with C/R, and this non-overlapping property is guaranteed by a locking mechanism.

A. Data Plane C/R Coordination

Cross-update consistency. Networks constantly go through configuration, policy, or topology changes. Each of these high-level changes will result in a set of changes in the forwarding rules, which we refer to as a *network update*. A network update usually has multiple devices involved; for example, setting up a path in a topology must have all on-path switch to configure corresponding rules. A switch is affected by an update if the update contains a rule to be added to, removed from, or modified in the switch. Fig. 2 shows how four switches in a network are affected by network updates. For example, s_2 and s_4 are both affected by Update 1

A data plane[2] checkpoint has *cross-update consistency* means that for any one network update in the network, either *all of none* of the individual device updates in the network update are recorded in corresponding device checkpoints.

Maximum recoverable checkpoints (MRCs). The SDN architecture is modeled as a logical controller C and a group of switches $S=\{s_1, s_2, ..., s_n\}$. We use c_t^s to denote the snapshot of the flow table in switch s at time t. When the controller requests the switches in the network to rollback to checkpoints that are made before T. The controller should find a set of consistent checkpoints $C = \{c_{s_1}^{t_1}, c_{s_2}^{t_2}, ..., c_{s_n}^{t_n}\}$, where $t_i \leq T$. We define the *maximum recoverable checkpoints* (MRC) of the SDN date plane to be the latest set of checkpoints before T yet the network-wide consistency condition is satisfied. By recovering to MRC, least computation and update results of the network are lost.

Data plane C/R Algorithms. To achieve MRC, NetCP applies four checkpointing policies:

1) All checkpointing events on switches are triggered by checkpoint request messages sent from the controller.
2) The controller requests switches to make checkpoints *only* right after a network update. Hence, we use c_s^u to denote a checkpoint on switch s after network update u.

[2]A data plane represents all switches in the network.

Algorithm 1 Coordination of Network Update and C/R

```
1: semaphore CRmtx, Umtx, NUmtx := 1, 1, 1; int readcount := 0
2: function NetworkUpdate
3:     P(CRmtx); P(Umtx); readcount++;
4:     if readcount=1 then P(NUmtx);            ▷ exclude C/R
5:     V(Umtx); V(CRmtx);                       ▷ allow following updates
6:     Do Network Update
7:     P(Umtx); readcount – –;
8:     if readcount=0 then V(NUmtx);
9:     V(Umtx);
10: function NetworkCR
11:     P(CRmtx);          ▷ exclude following C/R and updates
12:     P(NUmtx);                               ▷ wait for existing updates
13:     Do Network Checkpoiting or Recovery
14:     V(NUmtx); V(CRmtx)
```

3) For a network update affecting a set of switches, if one switch is requested to make a checkpoint then all others are also requested to make checkpoints.
4) If the controller decides to request checkpointing, all the switches whose states have changed after their latest checkpoints shall also be requested to make checkpoints.

B. Achieving Coordination Using Mutex

The first two policies intend to reduce unnecessary checkpoints. For example, based on them, a switch will not take a new checkpoint if there is no update since the last checkpoint.

The third "all-or-none" policy reduces the occurrence of the *domino effect* which may force switches to roll back to very early states. For example, as shown in Fig. 3 (b), a network update u affects three switches s_1, s_2, and s_3 and only s_1 and s_2 are requested to make checkpoints $c_{s_1}^u$ and $c_{s_2}^u$. Suppose a failure is detected on s_1 and s_1 rolls back to the checkpoint $c_{s_1}^u$. Since s_3 does not make a checkpoint immediately after u, it is possible that s_3 has to roll back to a checkpoint made earlier than u. This roll back "invalidates" the network update u and thus s_1 and s_2 have to roll back to checkpoints earlier than u, too. $c_{s_1}^u$ and $c_{s_2}^u$ then become useless. Such situation may continue to happen and eventually may lead all switches to roll back to the initiation states, which is called the domino effect. By applying the "all-or-none" policy, NetCP completely prevents the domino effect among the set of switches affected by the same network update.

By applying the fourth policy, NetCP makes it easy to find MRCs. All switches whose states have changed are requested to make a checkpoint after a network update. Hence, the set of the last checkpoints on all switches is always network-wide

consistent. When the controller decides to recover the network state to the MRCs before T, it just requests all switches to roll back to their last checkpoints made before T.

We let L denote the maximum waiting time for checkpointing. When there is a network update, the controller checks whether there is at least one switch that has not been requested for checkpointing for time L. If such switch exists, it requests to checkpoint the switch. This algorithm prevents switches from being too often or too rarely checkpointed. Administrators may choose a proper frequency of checkpointing by tuning the parameter L.

Note that, in the 2nd policy above, NetCP "make checkpoints only right after a network update". That is, the checkpointing operation never overlaps with network update operations. In SDN controller, the NetCP application and other network applications that cause network updates run in parallel, how to coordinate them to avoid overlapping is a challenge. We adopt a locking mechanism to overcome this challenge.

The locking mechanism is designed with the following principles.

1) When C/R start, if there are existing network updates, the C/R task waits for them to finish.
2) When C/R start, all following network updates that have not started are blocked, waiting for the C/R task to finish.
3) When there is no C/R task, network update tasks (from different applications) do not block each other[3].

Algorithm 1 describes the logic of the locking mechanisms. The function `NetworkUpdate()` (line 3-10) is used to modify applications in the controller that generate network update; and the function `NetowrkCR()` (line 11-15) is used by the NetCP application to issue C/R request.

V. Non-interruptive Switch C/R

Checkpointing and recovering a switch is not a trivial task, as there are network flows running through. The C/R operation should not interrupt the traversing traffic. We leverage the multi-table design in SDN switches, divide tables into three groups, and assign rules to different tables during normal operation, checkpointing, and recovery. And this three-group design can achieve incremental checkpointing for efficiency and non-interruption C/R for running traffic.

Requirements. There are two requirements for switch C/R. First, as network update may be frequent, continuous checkpointing would be costly for storage and disk I/O time. Thus, it is important that the switch C/R supports *incremental checkpointing*. That is, if requested, the switch is able to checkpoint the delta between current states (i.e., flow tables) and the latest checkpoint. For an incremental checkpoint, a switch only stores the difference between its current state and the previously checkpointed state: the rules that have been changed. Upon receiving a rollback request to a checkpoint c_s^t, the switch reconstructs the state using a full checkpoint c_s^{full} and the incremental checkpoints sequence after c_s^{full}. In incremental checkpointing, the restoring time increases for combining multiple checkpoints. Hence, though much less frequently, full checkpoints are still requested to switches in order to guarantee a reasonably fast restore time. Overall, the I/O cost for making checkpoints is significantly reduced, especially when the network is relatively stable.

Second, for the availability requirement from network applications, switches should serve traversing traffic without interruption. Even during the checkpointing and recovery, the switch should not drop packets. For example, during recovery, the rules in switch forwarding tables should be replaced by historical ones in the checkpoint, naively clear the table and install historical rules results in a gap in time when there are no rules in the table, causing packets to be dropped. Even replacing rules one by one would cause a small period of table inaccessible, because updating a table would trigger a write lock on the table, excluding table lookups.

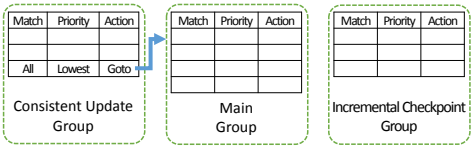

Fig. 4. Flow Table Design of NetCP

Three groups of tables. OpenFlow [7] allows a switch to maintain multiple sequentially organized flow tables. Each table contains multiple rules, and each rule can be modified atomically. By default, on each switch, the packet processing starts from the first flow table and goes along the pipeline. When a packet matches a rule, the corresponding action will be executed (e.g., forward the packet to a port). When a packet matches a rule with "Goto" action, the switch will continue to try to match the packet with another flow table. Packets that miss all the flow tables will be dropped or forwarded to the controller, according to the switch configuration.

As shown in Fig. 4, the flow tables are categorized into three groups, namely the *Consistent Update Group* (CUG), the *Main Group* (MG), and the *Incremental Checkpoint Group* (ICG). Each group may include one or more tables.

- The MG contains the normal flow tables of an OpenFlow switch.
- The CUG is used during rollback. It maintains replicas of the safe rules in the MG to ensure update consistency.
- The ICG includes the differential rules between checkpoints, *i.e.,* rules that have been changed since the last checkpoint. Its purpose is to support to take incremental checkpoints.

The three-level design is consistent with the OpenFlow pipeline and can be easily implemented on OpenFlow switches. In each table in the CUG, there is a table-miss rule, which contains a wildcard match field that matches all packets with the lowest priority. The action of this rule is "Goto" the first table in the MG. Packets that miss all the tables in the CUG and MG will be forwarded to the controller or dropped according to the switch configuration.

[3]There is an issue to schedule multiple network updates simultaneously. This issue can be solved by the consistent update solutions in [19], [25], and is out of the scope of this paper.

A. Switch C/R Algorithms

The switch C/R agent works as the following algorithm.

On receiving a flow update message. The switch adds, modifies, or deletes the rules in the MG accordingly. For each added or modified rule, a replica is stored in the ICG. For each removed rule, a rule with the same match field is stored in the ICG. The replica is marked as removed with lower priority than the table-miss rule, so that no packet would match the deleted rule. If there is an existing table entry in the ICG shares the same match fields with the one to be stored, the existing one is replaced. Note that, adding rules in the ICG does not affect the MG. Hence, ongoing flows will not be blocked.

On receiving a checkpoint message. NetCP switches are capable of making two kinds of checkpoints, incremental and full checkpoints. A checkpoint message from the controller specifies the type. If the switch is requested to make an incremental checkpoint, it simply takes a snapshot of the rules in the ICG and stores the snapshot in its local storage. These rules reflect how the flow tables in the Main Group have changed since the last checkpoint. After that, the rules in the ICG will be cleared. When the switch is requested to take a full checkpoint, it will store all rules in the MG as a checkpoint.

For software switches running on a hypervisor, the checkpoint can be stored in the external memory. For hardware switches, it can be stored in an external storage disk, or a distributed file system.

On receiving a rollback message. The switch first computes the safe rules by comparing the current flow table and the target checkpoint. Then it generates replicas of these rules into the CUG before any modification of the MG being executed. Then the MG is cleared, and rules from the checkpoint are loaded to MG. In this way, every packet is first processed by the CUG during the rollback. Packets that match the safe rules will not be processed by the MG. Therefore, the requirement of update consistency is ensured. Note that the most packets would match the safe rules, only a small fraction of packets do not go through the safe rules.

After rollback recovery, a switch clears both the CUG and ICG to prepare for the next round of checkpointing or rollback.

B. Data plane implementation

We implemented the NetCP data plane module on Open vSwitch (OVS) [5]. OVS is a software implementation of a virtual multi-layer network switch. The controller communicates with OVS switches via OpenFlow connections, which is based on TCP or TLS.

We implemented the NetCP checkpoint/rollback mechanism on OVS by introducing new commands into the existing managing tools. For making checkpoints of a switch, we reuse the code handling the `FLOW_STATS_REQUEST` message[4] to get the flow table of the switch and write the table as a file. For recovering switch state, we reuse the code handling `FLOW_MOD` message[5]. The program reads the checkpoint file and installs the rules into the OVS flow table.

We also extended the functionality of OVS by implementing a new extension for the OpenFlow protocol. We introduce two new types of messages, the `NetCP_REQUEST` and the `NetCP_REPLY` messages. Upon receiving these messages, the switch will carry out the corresponding actions and reply a `NetCP_REPLY` message.

Corner cases. The controller may not receive the `REPLY` message from a switch that has been requested to checkpoint due to three reasons: (i) the `REQUEST` message is lost; (ii) the switch encounters a failure making checkpoint and hence does not complete the checkpoint process; and (iii) the `REPLY` message is lost. For the first two cases, the controller should re-send the `REQUEST` message to request a new checkpoint. However, re-sending the `REQUEST` message may cause an unnecessary checkpoint for the third case. Hence, we request the controller to send the `REQUEST` message with the same `TOKEN` field if it does not receive the `REPLY`. The switch is able to detect the same `TOKEN` field and reply a `REPLY` directly without making a checkpoint in the third case. For the other two cases, it will make a new checkpoint.

VI. Efficient Controller C/R

NetCP provides two approaches to checkpoint and rollback an SDN controller: the full controller C/R and application-level C/R. The first approach is more straightforward, complete, and costly, and the second approach is proposed to reduce the high overhead and provide flexibility.

A. Full Controller C/R

Many aspects of SDN debugging work have mentioned the need to perform a checkpoint for the entire controller for various purposes [11], [28], [31]. However, none of them have done a complete study on how to achieve it. The closest work is in [28], where the proposed system forks the entire controller process and runs it as a child process for later use. However, the forked processes are not saved into non-volatile storage and hence cost a large amount of memory. This approach has the efficiency problem to support multiple checkpoints across time.

For Linux/Unix systems, a number of C/R software tool packages [1], [10], [12] are available. With the help of these tools, the state of the running process can be easily dumped and recovered. Hence, it is intuitive to utilize these tools for the SDN controller program to achieve checkpointing and rollback on the process level. However, in SDN environment, the controller and switches keep communicating via TCP or TLS connections. Based on our investigation, we find that existing checkpointing tools have poor support on handling TCP connections. To handle network connections, we allow the controller intentionally break its TCP connections before checkpointing. Then it ignores any new incoming connection request. Hence, it becomes a stand-alone process without

[4] `FLOW_STATS_REQUEST` messages requests the switch to reply the status of the current flow table.

[5] `FLOW_MOD` messages are used to add, modify, or remove one or more flow table entires.

external communication and it thus can be checkpointed using the existing tools. We implemented the proactive approach for the POX controller [8] using the BLCR [12] tool. We added a customized signal handler for POX, so that the following two procedure can be triggered before and after checkpointing or rollback. When a checkpoint is requested, the controller closes the connection. After the checkpointing or rollback, the controller resumes to listen to the port and wait for future connections. We also modified POX, so that it handles reestablished connection correctly and the mapping between the connections and switch abstractions in the controller can also be recovered after a rollback.

B. Application-level checkpointing and rollback

Basic application C/R. Full controller C/R is not efficient and flexible in some scenarios. The controller may contain multiple applications, each of which is a module resides as a component of the controller, and is responsible for a particular management task, e.g., routing, monitoring, access control, server load balancing, and service chaining. These applications execute independently and each can be modified independently. Application level C/R is to store and recover state for each application separately. Controllers of this kind may be very large. For example, the OpenDdaylight (ODL) controller [6] consumes at least 1G memory during normal execution. C/R on such large piece of memory is expensive on both time and storage.

We take the ODL Helium controller as an example and study the structure of its applications. Most applications can be abstracted as a model that reads from an input data set, does internal processing, and writes to an output data set, which in turn is the input of another application. Such processing is event-driven, i.e., when an application writes its own output data set, it notifies the dependent applications their new input. Table I gives a summary of the input/output data of four common applications of a controller. We also observe similar designs in modularly designed controllers, such as Floodlight [2] and Beacon [14].

Based on this observation, we propose the basic application-level C/R method. For each application, we periodically store its input data set, including network policies, current flow status, and output of other dependent applications. During rollback, the difference between the previous input data and the current input data is sent to the program as external events. The program updates its internal states, and produces relevant outputs. This requires minor modification on the application to add such functionalities. However, there are still two issues with this basic approach and NetCP handles them as follows.

Issue 1: application dependencies. Multiple applications in the controller may be intermingled with each other, so changes in one application may affect other applications as well. For instance, the input changes to a routing application will result in computed path changes, which will be a new input to the rule management application. Hence, we propagate the rollback effect of one application to other related applications.

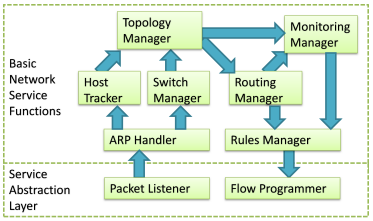

Fig. 5. Application dependency graph of the ODL controller

The dependencies across applications can be learned from analyzing the controller code. For example, ODL adapts the OSGi (Open Services Gateway initiative) standard, which is widely used by the controller with modular designs. In ODL, each application registers for the set of events of interest, and will be notified once these events occur. An example analysis results of the ODL controller is shown in Fig. 5. The dependency graph across the applications is a directed acyclic graph and there is no dependency loop. Similarly, we observe no looping dependencies in Floodlight and Beacon. The loop-free property suggests the propagation of rollback effect will terminate after certain rounds of message passing. This property is reasonable for SDN designs in modularized models, since it simplifies the management.

Issue 2: rule composition across applications. The output of different applications may be composed together at later applications. For example, both the routing and the monitoring application may affect the rules with the same matching field. These rules may be composed to reduce the number of rules [15]. Per-application rollback may cause only a part of the composed rules to change. To support rollback, we provide a reversion operation to the composed rules, which requires keeping track of the rules before composition. For example, the simplest rule composition is the aggregation based on prefixes [24], which is used in ODL controller.

To handle the composed rules, we modify the rule management component. We first maintain the set of raw rules before the aggregation and give each of them an index. Then in the final rule table, for each rule, we add a field to record the set of raw rules from which this rule is aggregated. If any of the raw rules are modified after rollback, the aggregated rule affected is identified and recomputed. Note that the rule de-aggregation problem is not specific to the C/R system. Our solution can be generalized to handle other rule modifications in the context of rule aggregation.

C. Full controller versus application-level

Compared to the controller level C/R, the application-level approach is more time and memory efficient when the C/R only affects one or a few applications. Another merit of the application-level approach is that it does not break the existing TCP connections. However, the full controller approach may be preferred when most of the applications have state changes.

VII. Evaluation

We implement NetCP on OVS, POX, and ODL. We evaluate its correctness, performance, as well as overhead.

TABLE I EXAMPLE INPUT AND OUTPUT FOR SDN APPLICATIONS

Application	Input	Output
Topology manager	Host List, Switch List, Edge List	Topology Abstraction
Monitoring manager	Topology Abstraction, Monitoring Policy, Flow Paths	Monitoring Rules
Routing manager	Topology Abstraction	Flow Paths
Rules manager	Flow Paths, Monitoring Rules	Flow-level Rules

A. Implementation

We implemented the NetCP data plane module on Open vSwitch (OVS) [5] by introducing new commands into its managing tools. For making checkpoints of a switch, we reuse the code handling the `FLOW_STATS_REQUEST` message, which requests the switch to reply the status of the current flow table. For recovering switch state, we reuse the code handling `FLOW_MOD` message, which is used to add, modify, or remove one or more flow table entries. Note that, all of these tools are running programs on an ordinary hypervisor machine, and checkpoint files are stored in its file system, which can be either a local file system or distributed file system.

We implemented full controller C/R approach for the POX controller [8] with the BLCR library [12], and application-level controller C/R approach on ODL controller [6]. We use Mininet [4] as the network emulator to connect multiple OVS instances.

B. Data Plane C/R

We set up a Mininet network with a linear topology of 26 switches and 40 hosts on each end. The link delay between two switches is set 1 ms and the control channel delay is 10 ms. The hosts ping each other with a 50 ms interval.

C/R overhead. We measure the time to make a checkpoint or rollback on OVS. Fig. 6 shows the time of performing C/R with varying number of rules. The overhead grows linearly with the number of rules for both C/R. The time of a rollback is five times as that of taking a checkpoint. Even with 10,000 rules, the time taken for checkpointing is less than 0.1 sec and that of a rollback is less than 0.4 sec. I/O contributes a major portion of the overhead.

Correctness and network performance during rollback. In each experiment, end hosts keep performing pairwise pings. At time 0, all switches restart and try to reconnect the controller. For a simple start, forwarding rules are learned by querying the controller. For NetCP, each switch rolls its internal flow table back to a previously checkpointed state. The end-to-end RTT reduces to normal faster after the rollback than after a simple restart.

We run the experiment for 100 times. Fig. 7 shows how RTT changes after a restart with and without rollback recovery of a typical experiment. For a simple start, communications are resumed after 1.5 sec, and all RTTs of ping packets drop down to the normal value after 2.1 sec. Using NetCP, communications resumed at 0.6 sec and the convergence time is 1.1 sec, which is much shorter than those of simple start. The cumulative distribution of the time for an RTT going back to normal is shown in Fig. 8. There is a significant gap between the two curves: NetCP uses 1.5 sec to make the RTTs go back to normal, while simple restart takes more than 2.5 sec.

C/R effectiveness. We measure the average *number of lost updates at the recovery stage*. It is defined as follows. When a rollback is needed, the goal of this recovery is to find a set of consistent checkpoints at all switches before time T_0. The state of a switch may have changed after the checkpoint, and these updates are lost when the switch rolled back. Hence, we use the number of missed updates to characterize how close the checkpoints are to T_0. Less lost updates indicate a more effective checkpoint method. We compare NetCP with a naïve approach: all switches create regular checkpoints synchronously after a fixed time interval. We use the real-world traffic data collected by CAIDA [9] and an ISP topology by Rocketfuel [30]. Fig. 9 shows the number of lost updates versus the number of checkpoints. When checkpoints are made more frequently, the MCR has fewer lost updates. We find that the NetCP has fewer lost updates when making the same number of checkpoints.

Fig. 10 shows the benefit by making incremental checkpoints. We use the same trace data and topology as those in Fig. 9. The total number of checkpoints made is fixed. Some checkpoints are incremental ones while the others are full ones. We let the ratio of incremental checkpoints vary and measure the total of the size of all checkpoints. Making incremental checkpoints reduce the total file size significantly.

To evaluate the update consistency during rollback recovery, we compare NetCP recovery method with a basic recovery method, in which the flow tables are cleared after switch restart and the rules are loaded directly into the flow table without any consistent control mechanism. Two pairs of hosts communicate via TCP across the network simultaneously, where the NIC rate is 1 Gbps. Each flow table contains thousands of rules. The term affected and safe flows are defined in Section V. For the affected flow, the policy is different in pre-rollback and post-rollback state. The safe flow has identical policy pre-rollback and post-rollback. Fig. 11 shows how the data transmission rate changes. We trigger a basic recovery at 3 sec and trigger another NetCP recovery at 11 sec. During basic recovery, both the affected and safe flows are blocked for a while. However, the rate of safe flow is not affected by NetCP recovery.

C. Full Controller C/R

We run the POX controller together with Mininet to emulate an SDN and evaluate the full controller C/R performance.

C/R overhead. We measure the time to make a checkpoint of the entire process and store it on disk and rollback using

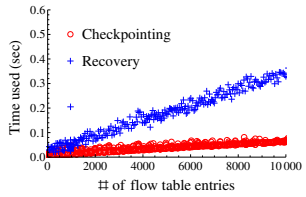

Fig. 6. Overhead of data plane C/R

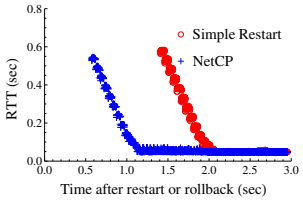

Fig. 7. End-to-end RTTs after data plane rollback or restart

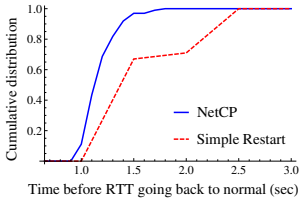

Fig. 8. CDF of time for RTTs going back to normal after data plane rollback or restart

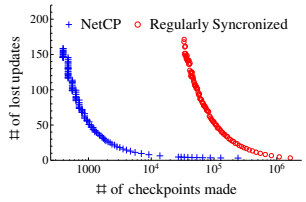

Fig. 9. Comparison of data plane C/R methods

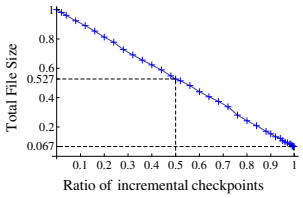

Fig. 10. Storage cost of incremental checkpointing

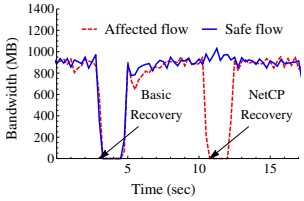

Fig. 11. Flow behaviors under basic and NetCP rollback

the checkpoint. Fig. 12 shows this value grows linearly as the size of the controller increases. We find that checkpointing and rollback has similar time delay under the same memory size. The delay increases linearly with the memory size. For a typical POX size (about 100 MB), the latency is only 0.5 sec.

Correctness and network performance during rollback are studied by examining the disruptions on the data plane with and without rollback recovery. We set up Mininet with a linear topology of 10 switches. Then we let the controller fail for a short duration and restart at time 0. We measure the RTT values of pairwise pings. Fig. 13 shows the delay changes. NetCP correctly recovers the connection after recovery. The communications recovered at 0.7 sec by NetCP and 1.5 sec by simply restarting the controller. The RTTs of NetCP then quickly converge to the value at 1.1 sec, while restart takes a longer time. The results demonstrate that NetCP improves end-to-end performance during controller rollback.

We run the experiments for 100 times. The cumulative distribution of the recovery time across all experiments is shown in Fig. 14. For 90% of the cases, NetCP constantly performs around 0.2 sec faster than the simple restart.

All of our experiments are carried on a commodity desktop computer. Disk I/O contributes the most overhead of the controller C/R. We expect a better performance when the controller is equipped with faster I/O devices.

D. Application Level C/R

ODL adopts a modular design for multiple application components which can be extended easily. A typical ODL process costs about 800 MB memory space and needs about 6 seconds for full controller checkpointing or rollback. We use the routing manager in ODL as an example for application-level C/R. This application calculates routing paths using Dijkstra algorithm upon request. We setup Mininet with random topologies, and let the hosts ping each other so that the controller computes all paths in the network. We measure the average time used for C/R the routing manager. We also measure the average time used to restart the application and re-compute all paths for comparison. We run each of the experiments for at least 50 times.

As shown in Fig. 15. While the number of switches goes up the checkpointing time is almost stable. However, rollback takes longer time, because the inter-application dependencies need to be handled. Restarting the application costs much longer time compared to rollback. For a typical random graph topology with 20 switches, the restart method takes 0.29 sec to complete, while NetCP rollback can be finished in 0.07 sec, reducing the time by 75%.

VIII. Discussion

Implementation on Hardware Switches. NetCP is also implementable on hardware SDN switches. OpenFlow switches can implement checkpointing and recovery by extending the OpenFlow modules and reusing the components that handle flow table operations. Some off-the-shelf routers come with attached hard drives, which can store checkpoint files; if no hard drives, a distributed file system can be used for the switch.

For OpenFlow switches, C/R can be easily implemented by extending the OpenFlow modules of these switches and reusing the components that processes `FLOW_STATUS_REQUEST` and `FLOW_MOD` messages. The checkpoint files can be stored in the attached storage. For those switches without attached storage, the checkpoint files can be stored in a distributed file system. The file system should be implemented so that the switches are able to access its checkpoint files with low latency. Hence, we believe it is practical to attach a non-volatile storage to an SDN switch.

Multiple Controllers. In a multi-controller network, the controllers work collaboratively as a distributed system. Accordingly, the concepts such as consistent checkpoint and a virtual clock can also be applied in controller checkpointing. We leave the details of NetCP in controller environments as the future work.

In the control plane, the controllers coordinate the management of the switch among themselves. Hence, the controllers

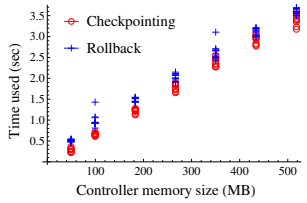

Fig. 12. Overhead of full controller C/R

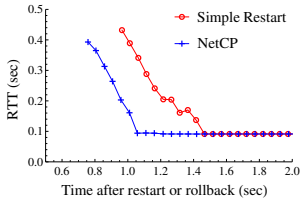

Fig. 13. End-to-end RTTs after controller restart or rollback

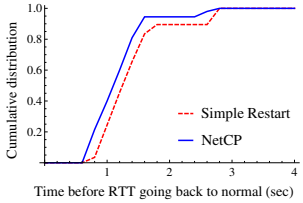

Fig. 14. CDF of time for RTT convergence after controller restart or rollback

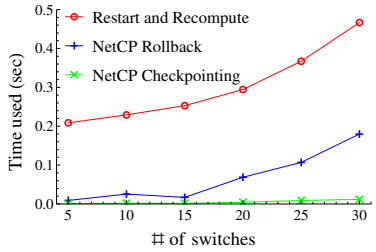

Fig. 15. Time cost of application-level checkpointing and rollback

works collaboratively as a distributed system. Accordingly, the concepts such as consistent checkpoint and virtual clock can also be applied in controller C/R. We leave the details of C/R in multiple controller environment as future work.

IX. Conclusion

This paper proposes NetCP, a novel checkpointing and rollback system for SDN. We developed a framework in which switches perform checkpointing under the coordination of the SDN controller to achieve global consistency across updates, each switch makes local checkpointing and recovery without interrupting network traffic, and the SDN controller can make fine-grained checkpointing. We demonstrated the feasibility of NetCP in terms of its correctness and performance for both the data plane and the control plane during checkpointing and failure recovery.

Acknoledgements

We thank the anonymous reviewers for their constructive comments and suggestions. Y. Yu and C. Qian were supported by National Science Foundation Grants CNS-1701681 and CNS-1717948. W. Wu was supported by the National Science Foundation of China (NSFC) under grant 61373002.

References

[1] CRIU, Checkpoint/Restore In Userspace. http://www.criu.org/.
[2] Floodlight OpenFlow Controller. http://www.projectfloodlight.org/.
[3] Google Compute Engine Incident 16007 . https://status.cloud.google.com/incident/compute/16007.
[4] Mininet. http://www.mininet.org/.
[5] Open vSwitch. http://www.openvswitch.org/.
[6] OpenDaylight Project. http://www.opendaylight.org/.
[7] OpenFlow. http://www.openflow.org/.
[8] POX. http://www.noxrepo.org/pox/.
[9] The CAIDA UCSD Anonymized Internet Traces 2013 - 2014. Mar. . http://www.caida.org/data/passive/passive_2013_dataset.xml.
[10] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. IPDPS'09*.
[11] B. Chandrasekaran and T. Benson. Tolerating sdn application failures with legosdn. In *HotSDN*, 2014.
[12] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. *Lawrence Berkeley National Laboratory*, 2005.
[13] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
[14] D. Erickson. The Beacon OpenFlow Controller. In *ACM HotSDN*, 2013.
[15] N. Foster et al. Frenetic: a network programming language. In *Proc. of ACM ICFP*, 2011.
[16] N. Handigol et al. Where is the debugger for my software-defined network? In *HotSDN*, 2012.
[17] C.-Y. Hong et al. Achieving high utilization with software-driven wan. 43(4):15–26, 2013.
[18] S. Jain et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM CCR*, 43(4):3–14, 2013.
[19] X. Jin et al. Dynamic Scheduling of Network Updates. In *Proc. of ACM SIGCOMM*, 2014.
[20] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proc. of ACM SOSR*, 2015.
[21] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying Network-wide Invariants in Real Time. *ACM SIGCOMM CCR*, 42(4):467–472, 2012.
[22] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. HotSDN '12, 2012.
[23] R. Koo and S. Toueg. Checkpointing and Rollback-recovery for Distributed Systems. *Software Engineering, IEEE Transactions on*, (1):23–31, 1987.
[24] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. In *Proc. of USENIX NSDI*, 2013.
[25] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *HotNets*, 2011.
[26] J. C. Sancho, F. Petrini, G. Johnson, and E. Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 58. IEEE, 2004.
[27] C. Scott et al. How did we get into this mess? isolating fault-inducing inputs to sdn control software. Technical report, EECS Department, University of California, Berkeley, Feb 2013.
[28] C. Scott et al. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proc. of ACM SIGCOMM*, 2014.
[29] J. Sherry et al. Rollback-recovery for middleboxes. *SIGCOMM CCR*, 45(4):227–240, August 2015.
[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of ACM SIGCOMM*, 2002.
[31] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. Hotswap: Correct and Efficient Controller Upgrades for Software-defined Networks. In *HotSDN*, 2013.
[32] H. Wang, C. Qian, Y. Yu, H. Yang, and S. S. Lam. Practical Network-wide Packet Behavior Identification by AP Classifier. In *Proc. of ACM CoNEXT*, 2015.
[33] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: enabling record and replay troubleshooting for networks. In *Proc. of USENIX ATC*, 2011.
[34] H. Zeng et al. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. of USENIX NSDI*, 2014.
[35] Y. Zhao, H. Wang, X. Lin, T. Yu, and C. Qian. Pronto: Efficient Test Packet Generation for Dynamic Network Data Planes. In *Proc. of IEEE ICDCS*, 2017.