JURY: Validating Controller Actions in Software-Defined Networks

Kshiteej Mahajan* U Wisconsin–Madison
Rishabh Poddar* UC Berkeley
Mohan Dhawan IBM Research
Vijay Mann IBM Research

***Abstract*—Software-defined networks (SDNs) only logically centralize the control plane. In reality, SDN controllers are distributed entities, which may exhibit different behavior on event triggers. We identify several classes of faults that afflict an SDN controller cluster and demonstrate them on two enterprise SDN controllers, ONOS and OpenDaylight. We present JURY, a system to validate controller activities in a clustered SDN deployment, involving topological and forwarding state, without imposing any restrictions on the controller behavior. Our evaluation shows that JURY requires minimal changes to the SDN controllers for deployment, and is capable of validating controller actions in near real time with low performance overheads.**

***Keywords*-Software-Defined Network, Controller Cluster, High Availability, Validation.**

I. INTRODUCTION

Commercial SDN [1] controller offerings, like those from Big Switch, Cisco, HP, and Juniper, deploy controllers as a cluster, for purposes of high availability (HA) and scalability. However, these controllers, like other distributed systems, are prone to faults [36], [55]. A controller is deemed faulty if it exhibits actions other than normal, such as response omissions, incorrect/inconsistent responses, or timing faults, which may adversely affect SDN operations. This paper looks at the problem of validating controller actions involving topological and forwarding state in near real time, where one or more controllers in an HA cluster [2] may be faulty.

While automated tools, such as FAI [5], YaST [25] and OSCAR [20], reduce cluster misconfiguration, faults may still manifest due to bugs in SDN control plane, rolling software updates, resource availability, etc. Controller nodes may respond differently due to differences in their operating environments, i.e., kernel version, controller software version, available resources (CPU, memory, network bandwidth), etc. Kernel security patches and controller software upgrades are often applied in a sequential manner (since the OS or the controller may require a reboot), and potentially leave a window of time when not all replicas run the same code (and at the same OS patch level), thereby leaving the replicas susceptible to operational faults. Further, controllers are often implemented in high level languages such as JAVA/PYTHON, and it is not uncommon to experience memory bloats or leaks, which may render a server unresponsive.

Several documented examples of operational faults in HA clusters exist—(i) routine upgrade causes nodes to be out of sync [8], [21], (ii) nodes in sync initially quickly desynchronize under load, and depending on which node is hit, display different data [2], (iii), nodes fail to sync at the time of re-synchronization [15], [17], and (iv) nodes fail to synchronize due to an `rsync` process on a replica [22].

The inherent asynchrony and concurrency in SDN controller clusters, along with the non-transactional nature of OpenFlow further exacerbates the problem, resulting in situations where inconsistent controller state may exist due to a replica crash or connectivity failure between the nodes [49]. Faulty nodes may (i) replicate incorrect state [11], [19], [23], [55], (ii) generate incorrect network actions for switches under their control [6], [13], or (iii) completely omit a response [3], [16], [55]. Recent work [31], [55] lists numerous such faults that manifest in popular SDN controllers, either due to subtle bugs or arbitrary replica crashes, or a combination of them.

Faults can cause a controller to not only behave incorrectly, but also violate protocol specification in some cases [13], [31], [55]. Most prior work in the domain of SDN verification [27]–[29], [38], [45], [46], [48], [50] limits itself to verifying high-level controller behavior (and programs running atop it) against specified network invariants. Likewise, recent work [51], [56] on model checking SDNs validate network actions only. However, none of them consider controller correctness in a clustered setup.

We present JURY—a system that leverages consensus amongst nodes with equivalent network view, to validate controller responses affecting both state propagation and network actions, without limiting capabilities of the faulty controller(s). There are two key observations that motivate validation of controller actions using consensus in SDN HA clusters. First, an SDN HA cluster offers *output determinism* for incoming triggers. In other words, since controller replicas in a cluster share the same distributed state (both topological and forwarding) to offer a logically centralized view, replicated execution in *other* controller nodes produces a high-fidelity replica of the execution in the *original* controller node, thereby producing outputs same as the original. Second, all non-adversarial (i.e., both faulty and non-faulty) controller activities, involving topological and forwarding state actions, update the controller-wide caches to enable consistency of state across controller replicas. This logging helps in validating controller actions.

Guided by the above observations, we determine the veracity of controller actions by (i) intercepting all network and state triggers to/from each controller in the cluster, (ii) replicating them across randomly selected replicas, and (iii) comparing the responses within an out-of-band validator to reach consensus on every action. Given that a clustered setup is the only way

* Both authors contributed equally.

[1] We use the term SDNs to refer to OpenFlow-based SDNs.

[2] We use the term HA clusters to refer to both Active-Active and Active-Passive modes [9].

SDN controllers are deployed in production environments, JURY improves reliability and robustness of SDN control (over single controller setup) with minimal overheads.

Practical response validation, however, is contingent on two factors. First, controllers may follow different consistency models, and in presence of rapid changes in network state, all nodes may not share the same global network view. Second, some controller applications may have non-deterministic or time dependent actions, which must be accounted for during validation. We show that even in light of the above concerns, validation using consensus in an HA cluster is feasible.

We have built a prototype of JURY for the ONOS and OpenDaylight (ODL) controllers, and have evaluated them with a cluster of 7 controller replicas over a physical testbed and Mininet [14]. JURY successfully detected *all* faulty controller responses for a 7 node cluster with 2 faulty controllers, with an average detection time of ~129ms for ONOS and ~700ms for ODL. Further, JURY-enhanced ONOS reported just 0.35% false positives across three different benign traces. JURY is also capable of validating $1K$ policies in just ~1.2ms, and imposes moderate network overhead in the worst case, i.e., full replication across all nodes.

This paper makes the following contributions:

(**1**) We identify (§ III-B) and demonstrate (§ VII-A1 and Appendix A) broad classes of faults afflicting low-level controller correctness and performance in a clustered setup.

(**2**) We present JURY—a system to validate both topological state and forwarding actions in an HA clustered controller setup, and facilitate easy action attribution.

(**3**) We provide a practical design (§ IV) for JURY along with its policy engine (§ V) that enables administrators to specify fine-grained policies on controller state.

(**4**) We implement JURY (§ VI) for both ONOS and ODL controllers, and evaluate them (§ VII) to demonstrate that replication for detecting consistency in controller behavior is practical, accurate and involves low overhead.

II. BACKGROUND

SDNs, unlike traditional networks, separate the data and control functions of networking devices. The logically centralized SDN controller governs the *flows* that occur in the data plane. The control plane communicates with the data plane on its southbound interface using a standard protocol such as OpenFlow [53]. When a switch receives a packet for which it has no matching flow entry, it sends them to the controller as a `PACKET_IN` message. The controller uses these messages to (i) update its topological view of the network, and (ii) create one or more flow entries in the switch using `FLOW_MOD` commands, thereby directing the switch on how to handle similar packets in the future. The control plane also exposes a northbound API for administrators and other third party applications to install OpenFlow rules in the switches.

A. Clustered controllers

Fig. 1 shows a unified workflow for a clustered SDN controller. On receiving triggers from the network (①) or

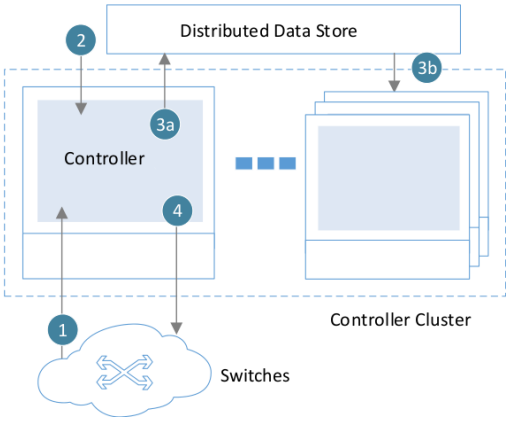


Fig. 1: SDN controller cluster.

administrator/applications (②), the controller may initiate updates to the distributed data store (③a and ③b), and possible network messages (④). Controller clusters follow standard HA architectures [9]: Active-Passive, wherein all switches connect to a single controller and others are passive replicas, and Active-Active wherein the network is partitioned in such a way that switches in each partition connect to a different controller in the cluster. In the Active-Active mode, the controllers may operate on different network of switches, but still share the same view of the network. Further, specific controllers can use more advanced configurations [4].

A.1 Transparency in controller actions

Logical centralization of controllers is enabled by the use of data distribution platforms, such as Hazelcast and Infinispan. All nodes maintain the same network view by propagating state changes, i.e., both topological and forwarding updates, to controller-wide caches built atop a distributed data store. This state synchronization enables the controllers to transparently issue directives to both *local* and *remote* switches in the SDN. Switches are local to a controller if it directly governs them, else they are remote. For example, enterprise controllers, like ONOS and ODL, can issue a `FLOW_MOD` to a remote switch by simply writing to the cache that manages the flow rules. The shared data store ensures that the remote switch's governing controller receives the cache update, which in turn issues the actual `FLOW_MOD`.

A.2 Nature of controller actions

The southbound interface is used by all network switches to send messages to the controller over OpenFlow. A controller can then respond to these triggers with zero or more instructions; such controller activities are referred to as "reactive". The northbound interface (typically REST APIs) may be used by third party applications or administrators to send instructions to network entities. Since this is done in an unsolicited manner from the switches' perspective, such instructions are termed "proactive". In addition, some controller modules are truly proactive and may send instructions to the switches without receiving any triggers on either interface. For example, a module may send some specific instructions periodically or at some specific time of the day. In production settings, a

controller can be both proactive and reactive, i.e., it performs proactive flow rule installations at the start to take care of most flows, but may still react to some unforeseen packets/flows.

For the purpose of this paper, we classify triggers from a controller's perspective. We refer to all triggers on the southbound and northbound interfaces as "external" triggers. These external triggers encompass all the reactive triggers as well as the Web-based REST triggers to the controllers. All other triggers that originate within the controller are referred to as "internal". For example, all triggers due to an administrator logging into a controller, or the truly proactive applications are classified as internal triggers. SDN controllers handle both internal and external triggers transparently by writing ensuing actions to controller-wide caches, followed by zero or more network messages. A controller may even delegate triggers to other controllers by writing to the distributed data store, e.g., issuing a FLOW_MOD for a remote switch.

A.3 Side-effects of controller actions

There are three types of side-effects possible for every controller activity involving topological state and forwarding actions: (i) writes to cache only (C), (ii) writes to network only (N), and (iii) writes to both cache and network (CN). In a clustered setup, network side-effect (N) on a local or remote switch can be achieved only through a cache write (C). For example, controllers read the flow rules cache to create and subsequently issue FLOW_MOD messages to the destination switch. Note that correctly behaving controllers always write to the cache, so only a network side-effect without any cache updates is indicative of a misbehaving controller.

III. Controller validation

We consider an SDN HA cluster and place no restrictions on the behavior of individual controllers, switches, hosts or controller applications. However, we assume that controllers (i) share a common distributed store, (ii) are non-adversarial, and (iii) do not willfully tamper with the network communication. Faulty controllers may still write incorrect entries to the network and/or controller caches.

A. Problem statement

Clustered controllers offer output determinism, i.e., given two replicas C_i and C_j with the same network state ψ, and each eliciting actions A_i and A_j on a trigger τ, then

$$\{C_i \equiv C_j\} \Rightarrow \{A_i = A_j\}$$

In a clustered setup of k controllers $C_{1...k}$, which elicit actions $A_{1...k}$ on replicated trigger τ, we reduce the problem of determining if C is non-faulty to determining consensus amongst replica actions $A_{1...k}$ on every τ, i.e.,

$$\text{NonFaulty}(C) \iff \forall \tau, \psi \ \{A_C = \text{Consensus}(A_1, ..., A_k)\}$$

However, certain controller applications may respond in a non-deterministic fashion, i.e., $\{A_1 \neq A_2 \neq ... \neq A_k\}$. We discuss extensions to our core consensus mechanism to deal with such scenarios in § IV-C.

B. Controller fault classes for validation

Like any other distributed system, an SDN HA cluster is susceptible to five broad classes of failures: (i) crash (or fail-stop), (ii) response omission, (iii) timing, (iv) response (both incorrect value and state transition), and (v) arbitrary (or byzantine). JURY, using the replicated state machine approach, can provide detection for all but crash failures, which would be reported as response omissions. However, faulty controller actions can be detected only if they are externalized as cache/network side-effects. Based on the above observations and those in § II-A3, all faulty controller actions (for timing, response and arbitrary failures) can be categorized as follows:

(**1**) **Type-I** ($T1$): In this class of activities, the faulty controller responds to external triggers with incorrect writes to controller-wide caches or network, or both.

ONOS DATABASE LOCKING. Clustered ONOS controllers occasionally reject switches' attempts to connect due to particular timing issues between the switch connects, causing the replicas to encounter a "failed to obtain lock" error from their distributed graph database [55].

ONOS MASTER ELECTION. In an older version of ONOS, link liveness between adjacent switches (governed by separate controllers) was tracked by electing the controller with a higher ID as the master. This election happens when a new LLDP packet arrives at the controller. If the master dies and later reboots with an ID lower than the other controller, both replicas will incorrectly believe that they are not responsible for tracking the link's liveness, and the controller with previously higher ID will incorrectly mark the link as unusable [55].

(**2**) **Type-II** ($T2$): In this class of activities, an administrator or a controller application proactively issues a write to both the caches and the network. However, the entries written are inconsistent with each other.

ODL FLOW_MOD DROPS. In ODL, FLOW_MOD messages issued to switches are first read by MD-SAL (ODL's in-memory data store), and then sent to the OpenFlow plugin, which maintains a queue of these tasks. Since, there is no control over the order of these egress calls, sporadically FLOW_MOD messages may be lost when writing them to the network [13], thereby creating inconsistency between the FLOW_MOD cache and the network.

(**3**) **Type-III** ($T3$): This class of activities is similar to $T2$ where an administrator or application proactively issues a faulty write to both the caches and the network. However, unlike $T2$, the entries written are consistent with each other, which makes such faults hard to detect.

ODL INCORRECT FLOW_MOD. With an older version of ODL, the OpenFlow 1.0 switch silently accepted a FLOW_MOD whose `Match` did not have proper hierarchy of fields set. The switch installed the flow discarding incorrect fields, causing inconsistency between flows on the switch and the data store [23]. Thus, while FLOW_MOD messages issued are consistent with those in the cache, switch behavior resulted in inconsistencies.

While some of these faults may be resolved as controllers mature, others would still persist due to the asynchrony and

Scenario	Nature	Faulty Action	Validation
$T1$	Reactive	Either C, or N, or both	✓
$T2$	Proactive	Either C or N, or both but C ≠ N	✓
$T3$	Proactive	Both C and N where C = N	✓*

Table 1: Classes of faulty controller actions to be validated. ✓* indicates that it may be possible to validate using policies.

concurrency in SDN clusters. Table 1 summarizes the three categories of faulty controller activities. Irrespective of the nature of fault, validation of $T1$ actions is possible since a chain of events starting from the trigger to the response can be tracked. Validation of $T2$ actions is possible due to observable inconsistencies between the side-effects. In the event of no inconsistencies, i.e., class $T3$ actions, validation is hard as there are no external triggers (unlike $T1$ actions) and even a faulty action is deemed legitimate. However, class $T3$ actions can be validated against administrator specified policies.

C. Difficulty of response validation in SDNs

Practical response validation using consensus in clustered SDN setups is hard because of (i) different controller consistency models, (ii) rapid changes in network state, and (iii) non-deterministic or probabilistic or time dependent actions of controller applications. Further, comparison of controller responses against a single reference controller alone does not suffice as the reference itself might be faulty, or can even become unresponsive.

Prior work [32], [33], [38], [51], [56], [59] neither addresses these concerns, nor prevents faulty controller responses, nor offers guarantees about network responses amongst the replica nodes in a clustered setup. While some prior work [37], [45], [46], [48], [50] provides rich policy frameworks for fast verification of network-wide invariants, it does not consider shared state modification amongst clustered SDN controllers. Thus, no simple combination of the above techniques is sufficient to validate controller activities involving topological state and forwarding actions.

IV. JURY

JURY leverages two key observations to enable controller nodes in the cluster to validate the actions of other nodes.

- Since controller nodes share the same distributed state to offer a logically centralized view, replicated execution in other controller nodes produces a high-fidelity replica of the execution in the original controller node, thereby producing outputs same as the original. Consensus amongst nodes with equivalent network view ensures that the specific controller action was non-faulty.
- All non-adversarial controller activities (i.e., both faulty and non-faulty), involving topological and forwarding state actions, update the controller-wide caches to enable consistency of state across controller replicas. This logging helps in validating controller actions.

The consensus mechanism is inspired by traditional solutions [35], [40], [43], [47], which use it for purposes of robustness and accountability in networked systems and runtime environments. However, JURY's goal is to only detect inconsistencies in controller responses arising due to low-level correctness and/or performance faults, and not resolve or prevent disputes, which would be prohibitively expensive at line speeds. Thus, JURY is light-weight and offers near real time performance. Further, it does not affect the *safety* and *liveness* of the underlying distributed system of the controllers, since it does not interfere with any controller operations.

WORKFLOW. Fig. 2 provides a schematic workflow for JURY, which enhances a normal SDN HA cluster at several levels to determine the veracity of controller responses to both internal and external triggers. JURY comprises of three main components—a replicator and a controller module on each replica, and an out-of-band validator. Tasks marked in solid grey are existing actions performed by nodes in the cluster, while those in dotted red are new tasks introduced by JURY.

JURY supports validation using three key features.

(**1**) JURY intercepts (①a and/or ② in Fig. 2), and replicates the relevant trigger (①b and/or ③b) in the controller that received the given trigger (called primary controller) to k randomly chosen controllers (called secondary) within the cluster, and generate additional responses. JURY ensures that all triggers follow the exact same control sequence in the secondary controllers. Since a cluster provides output determinism, all nodes will generate the same response for the same trigger, which may be writes to cache or network, or both. Note that for some other given trigger, the primary and the corresponding k secondary controllers could be different.

(**2**) JURY maps all controller responses to incoming triggers, thereby providing precise action attribution.

(**3**) JURY transmits these responses (①c, ③c and ④c) to an out-of-band validator to determine within a specified time limit whether a controller action was valid or not.

A. Trigger interception and replication

(**1**) **External triggers.** Controllers generate responses on external triggers, e.g., `PACKET_IN` on the southbound interface, which may lead to a $T1$ fault. JURY leverages a custom replicator on the primary controller node to intercept and replicate network messages (both northbound and southbound) to k randomly chosen secondary controllers (①b). The replicator executes outside the controller binary on the primary, and thus a faulty controller would not affect the integrity of the replicated trigger.

The replicator sets up TCP channels to ensure reliable and in-order delivery to the secondary controllers. The JURY module within the secondary controller uniquely taints all such replicated messages. Thus, ①b is tainted to identify the trigger and the primary controller that sent it. JURY propagates this taint throughout the processing pipeline, using it to differentiate between (i) triggers to, and (ii) responses from primary/secondary controllers.

(**2**) **Internal triggers.** Unlike external triggers that are triggered via the network, internal triggers may originate from within the controller (or one of its applications) or are initiated by the administrator. Thus, triggers for these types of proactive

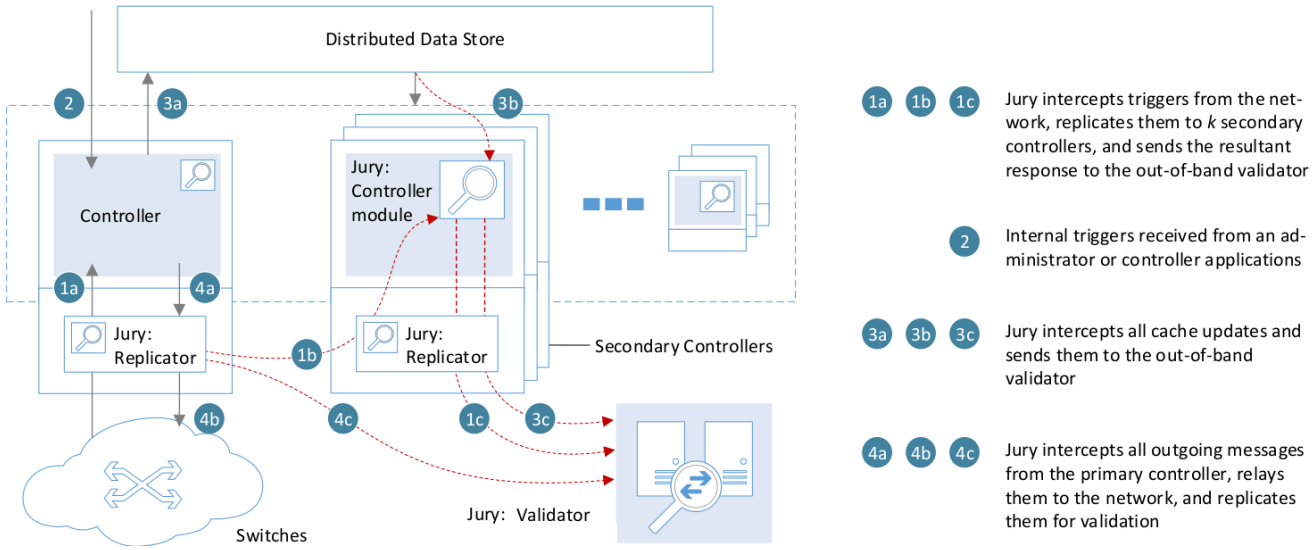


Fig. 2: JURY workflow. JURY is applicable to several HA controller setups [4], including Active-Active and Active-Passive. Solid grey arrows denote existing tasks in an HA cluster, while dotted red arrows indicate new tasks initiated by JURY for validation.

actions (classes $T2$ and $T3$) cannot be explicitly intercepted. JURY leverages the fact that controllers write all resultant actions to controller-wide caches, and instead intercepts all cache updates by hooking into the controller's cache manager (③b). Most data distribution platforms already provide (i) reliable replication of cache events (originating from internal triggers) to other cluster nodes, and (ii) authentication of cluster nodes, which JURY relies upon for action attribution.

Since, unlike external triggers, interception of cache events happens within the controller, it is possible that a faulty controller may incorrectly alter the entries before writing them to the caches. JURY therefore provides administrators with additional mechanisms (as will be explained later in § V) to detect such faulty controller updates. In case the faulty controller writes directly to the network, JURY intercepts such outgoing network messages (④c) and sends them to the validator, thereby uniquely identifying the sender node.

JURY's consensus mechanism requires that all replicated triggers (①b and ③b) experience the same network state within the secondary controllers, which then elicit identical responses. However, SDN controllers follow different consistency models, i.e., either strong or eventual. In strongly consistent controllers, all replicated triggers always experience the same state, while in eventually consistent controllers, only a subset of the controllers may have equivalent network view, particularly in presence of rapid changes in network state.

B. Action attribution

Precise action attribution is required to identify controllers that misbehave in response to triggers. However, since triggers for $T2$ and $T3$ actions originate within the controller, JURY uses different mechanisms to achieve action attribution for external and internal triggers. Once a response has been mapped to its trigger, JURY transmits the responses to a validator (①c, ③c and ④c).

(1) External triggers. JURY leverages the taint on the received message to accurately determine if the response elicited at a secondary controller corresponds to an external network trigger. It tracks the propagation of the tainted message within the controller, and marks the response with the same taint. Responses to a replicated network message may involve writing to controller-wide caches and/or issuing `FLOW_MOD` messages over the network. However, multiple such responses arising from secondary controllers may cause network instability and unpredictable behavior. Thus, JURY records and sends the secondary responses for validation, and drops all `FLOW_MOD` messages or writes to the caches generated as part of these responses. In other words, JURY does not induce any cache/network side-effects due to processing of triggers by secondary controllers. While faulty controllers may incorrectly alter the taint identifying a controller, a majority of responses from replicas with equivalent network view ensures that all $T1$ faults are detected.

(2) Internal triggers. Action attribution as described for $T1$ faults cannot be applied to $T2$ or $T3$ faults as these actions are untainted. JURY's response validator determines the origin of these actions from the corresponding cache events replicated across the cluster. Specifically, JURY intercepts all cache updates at both the primary and secondary controllers and transmits them to the validator. However, a faulty controller may also write to the network bypassing the cache altogether. JURY can trivially identify the origin of the network message in such cases since it intercepts all outgoing network traffic.

C. Response validation

The validator receives multiple responses with information about network writes and cache updates from the primary as well as the k secondary controllers in response to a trigger. The validator, based on deviation from expected consensus across the responses from the controllers, validates the action taken

```
CONTROLLER_RESPONSE_ VALIDATOR(𝕊, k)
Input: 𝕊 : stream of incoming controller responses
       k : number of secondary controllers
Output: 𝕆_τ : validation result for each trigger τ
Initialize: Ψ_id : per-controller id indexed state
            V_τ : per-trigger validation store
            N_τ : per-trigger response count
foreach ρ[= (id, τ, entry)] ∈ 𝕊 do
    N_τ++
    θ_τ = START_TIMER_IF_NOT_STARTED(θ_τ)
    V_τ = V_τ ∪ (Ψ_id, ρ)
    /* update state for this controller id */
    if (IS_CACHE(ρ))) then      Ψ_id = Ψ_id ∪ entry
    if (TIMER_EXPIRED(θ_τ) || 2k + 2 == N_τ) then
        if ((k + 2 < N_τ) || (TAINTED_ENTRY_EXISTS(V_τ))) then
            /* external trigger */
            𝕆_τ =POLICY_CHECK(SANITY_CHECK(CONSENSUS(V_τ)))
        else
            /* internal trigger */
            𝕆_τ =POLICY_CHECK(CONSENSUS(V_τ))
    if (False == 𝕆_τ) then      RAISE_ALARM()
end
```

Algorithm 1: Algorithm to validate controller responses.

by the primary controller. JURY relays to the validator (a) 1 network write event generated at the primary controller, (b) $k+1$ accompanying cache updates from primary and secondary controllers combined, and (c) k responses generated from the replicated execution of external triggers at each secondary controller. Therefore, the validator in total receives at most $2k + 2$ responses for each external trigger and at most $k + 2$ responses for internal triggers. Note that the $k+1$ cache updates are replicated automatically to all cache instances and require no explicit propagation.

CONTROLLER STATE MAINTENANCE. The validator maintains a succinct current state for every controller in the cluster. Specifically, it keeps a running count of the total cache updates received per controller along with a copy of the latest update, which sufficiently represents each controller's current state. Note that in-order processing of cache updates is necessary for accurate state management per controller. Since inter-controller state synchronization via these updates happens over TCP, it preserves the order in which the updates occur. JURY relays these updates to the validator in the same order.

VALIDATION. Algorithm 1 lists the steps to validate all controller responses. Each response from a controller has three entries–controller identifier (*id*), trigger identifier (τ) and controller response body (*entry*). The presence of taint helps classify the controller response as a replicated execution event from a secondary controller and the nature of *entry* helps classify the response as either a primary network write event or a cache update. The validator maintains a list of responses received per trigger (V_τ), along with the number of responses received (N_τ). N_τ is used to classify the trigger as internal or external. A value of N_τ greater than $k + 2$ or the presence of a tainted response in V_τ indicates an external trigger, else the trigger is internal. Depending on the trigger type, appropriate action is taken on V_τ to decide the output.

For both external and internal triggers, the validator executes the CONSENSUS mechanism on responses stored in V_τ. In event of a network write, the validator performs a SANITY_CHECK to assert if the primary controller's network write is consistent with the cache updates across all the controllers. Finally, irrespective of the trigger, the validator also performs a POLICY_CHECK (§ V) on the primary controller's response to validate it. The CONSENSUS mechanism also addresses the following challenges:

(A) TRANSIENT STATE ASYNCHRONY. Whenever a response is received from a particular controller, a snapshot of the controller's current state is stored along with the response in V_τ. This snapshot serves as an indication of the execution environment at that controller, and makes JURY's CONSENSUS mechanism for external triggers state-aware. The primary controller's action is validated against those secondary controllers that were in the same snapshot state as the primary. This averts any false positives owing to transient inconsistency in state if the controllers are eventually consistent.

(B) NON-DETERMINISM. In some instances, the controller's application logic might result in non-deterministic actions. The CONSENSUS mechanism infers the same if each controller's response on the given trigger is distinct from others, and labels the action for this trigger as non-faulty. In other scenarios where not all non-deterministic outputs are distinct, the CONSENSUS mechanism employs majority amongst controllers with equivalent state. While the latter may result in false positives, it is not possible to address all forms of non-determinism without prior knowledge of the application logic.

(C) SLOW REPLICAS. JURY also has a per-trigger timer θ_τ that starts when the first response is received. θ_τ's timeout serves as a deadline for validating the responses. This is useful to rule out responses generated by slow replicas from the consensus, as they are more prone to be faulty. However, the replicas in an HA cluster in a LAN are unlikely to suffer from high latency. If no response is received from the primary controller before the timeout, JURY flags a timeout fault for the controller. With a stricter timeout, this mechanism may lead to a high number of false alarms. JURY can mitigate this concern using an adaptive timeout based on the most recent latency trends of the controller responses. Currently, JURY requires administrators to set the validation timeout.

V. JURY'S POLICY FRAMEWORK

JURY provides a light-weight policy framework that enables administrators to centralize enforcement of fine-grained checks on controller actions. These checks must be specified in the constraint language listed in Table 2. Unlike Merlin [57], which provides a framework to manage all SDN resources, JURY's policy framework is specific to specifying topological and forwarding states only. Each policy has four components—controller, trigger, cache and destination. *controller* lists the controller id(s) whose actions must be validated. *Trigger* directs the validator to check the policy on internal, external or all triggers. *Cache* specifies the data store for which the entries must be validated upon a specified operation, such as create, update or delete. Lastly, *destination* specifies whether the side-effect is local or remote, or the entire network. Fig. 3 shows an example policy that raises an alarm

Feature	Description
Controller	CONTROLLERID | *
Trigger	INTERNAL | EXTERNAL | *
Cache	ARPDB | HOSTDB | EDGEDB | FLOWSDB | etc.
Destination	LOCAL | REMOTE | *

Table 2: JURY's policy language. This language can be used to express constraints on controller actions.

```
<Policy allow="No">
  <Controller id="*"/>
  <Action type="Internal"/>
  <Cache ="EdgesDB" entry="*,*" operation="*"/>
  <Destination value="*"/>
</Policy>
```

Fig. 3: Example policy. This policy raises an alarm if *any* controller proactively modifies the EdgesDB cache.

if any controller proactively updates the EdgesDB cache to modify any part of the network topology.

The validator evaluates the policies after reaching consensus amongst secondary replicas with equivalent network state. It computes values of the policy directives for exactly one of the matching responses, and checks for membership in the set of all policies, i.e., an external trigger that generates $2k + 2$ responses requires just one response to be checked per policy, which significantly reduces the validation effort. The validator flags a response if it does not match the consensus or violates administrator-specified policies. In event of an alarm, JURY extracts information about the offending controller, trigger and the associated response, and presents it to the administrator for further action. JURY's precise action attribution helps the administrator to diagnose problems quickly.

VI. IMPLEMENTATION

We have built a prototype of JURY for the ONOS and OpenDaylight (ODL) controllers based on the design described in § IV and § V. Of the several open-source SDN controllers available, only these had mature support for clustering. While JURY is applicable to several different controller configurations [4], including the standard HA Active-Active and Active-Passive modes, we experimented with ANY_CONTROLLER_ONE_MASTER [4] setup for ONOS, and SINGLE_CONTROLLER [4] setup for ODL. JURY's controller modifications for ONOS and ODL required ~250 and ~550 lines, respectively, while the validator was written in ~450 lines. We now list a few salient features of our implementation.

A. Trigger replication

We achieve trigger replication using programmable soft switches (or OVSes). To optimize the implementation, we configured replication rules at the OVS as per the controller's clustering mechanism. For ONOS, we configured the OVS as a transparent proxy to forward all packets normally, and ensure that the secondary controllers receive a replica of the forwarded packet. For ODL, we configured the OVS to connect to the secondary controllers in OpenFlow mode, while normally forwarding packets to the primary controller. We subsequently installed rules on the OVS to forward all incoming packets to the secondary controllers. However, this configuration causes the OVS to forward packets as PACKET_IN messages to the secondary controllers. If the trigger to the primary controller was already a PACKET_IN, the OVS encapsulates it and the secondary controllers now receive a doubly encapsulated PACKET_IN. We therefore decapsulated such messages at the secondary controllers before processing them further.

B. Correct response elicitation

In ONOS, replicated PACKET_IN messages received at the secondary controllers are processed normally eliciting desired responses. However, all outgoing network/cache responses at the secondary controllers are captured and sent to the validator before being dropped, thereby causing no side-effects. We required ~200 LOC to ensure correct response elicitation in ONOS. In ODL, we stripped the doubly encapsulated PACKET_IN messages at the secondary controllers and attached metadata to the message according to the context at the primary controller. Thus, each secondary controller exhibits the same responses as the primary controller. We modified ~400 lines in ODL to ensure correct response elicitation.

C. Action attribution

JURY extracts the origin of cache updates from events generated by the data stores (Hazelcast and GossipStores in ONOS, and Infinispan in ODL), and sends them to the validator. For external triggers, JURY's controller module uniquely taints incoming PACKET_IN or REST queries received from the replicator module. ONOS reactively installs source-destination based flow rules, and the forwarding module preserves the PACKET_IN context. Thus, JURY required ~50 lines of change to taint the outgoing FLOW_MOD messages. In contrast, ODL proactively installs destination-based flow rules as soon as it receives PACKET_IN messages for ARPs indicating host discovery, i.e., even before the first traffic packet is sent as a PACKET_IN message. Thus, instead of making wide ranging modifications in ODL to support taint propagation, we implemented our own forwarding module, which reacts to PACKET_IN messages of the actual traffic to install source-destination based flow rules. We modified just ~150 lines in ODL to implement action attribution and our own forwarding module.

VII. EVALUATION

In § VII-A, we evaluate JURY's accuracy by measuring fault detection time, false alarms generated under benign conditions, and also compare its performance with related work. In § VII-B, we measure JURY's effect on network latencies, cluster throughput, and policy validation.

EXPERIMENTAL SETUP. Our physical testbed consists of 7 servers (running controllers) connected to 14 switches (IBM RackSwitch G8264) arranged in a three-tiered design with 8 edge, 4 aggregate, and 2 core switches. All of our servers are IBM x3650 M3 machines having 2 Intel Xeon x5675 CPUs with 6 cores each (12 cores in total) at 3.07 GHz, and 128 GB of RAM, running 64 bit Ubuntu v12.04. We installed an ONOS cluster (v1.0.0) and an ODL cluster (Hydrogen v1.0) with 7 replica controllers. Note that clustering support in recent versions of ODL (Helium and Lithium) is still not stable.

All experiments used controllers provisioned with 12 cores and 64 GB memory, and each running atop a separate server.

Host-to-validator communication and inter-controller traffic was isolated to avoid any performance penalties due to replication. The controller JVM was provisioned with a maximum memory of 2GB starting with an initial pool of 1GB. We also used a network of 24 Mininet switches and hosts, arranged in a linear topology, to drive traffic to our controllers. The Mininet network connects to the controllers via OVSes running on the servers. The validator was run on a separate host connected to the servers via an out-of-band network.

NOTATION. In the following sections, n refers to the cluster size, k is the replication factor ($k=2$ means traffic is sent to a primary controller and to 2 other secondary controllers), and m is the number of faulty controllers.

A. Accuracy

JURY uses timeouts to impose a threshold on all validation decisions. We first empirically determined validation timeout for our ONOS and ODL controller clusters. We used a linear topology of 24 Mininet switches and 24 hosts to drive traffic to the cluster of 7 controllers at different PACKET_IN rates for a duration of 60s. We initiated random host joins, link tear downs and flows between hosts, and determined the time taken to reach consensus on controller actions with $k=2$, 4 and 6.

Fig. 4a plots the results for ONOS. We observe that with increasing k at a peak PACKET_IN rate of $\sim 5.5K$, the time taken to reach consensus increases. This is because JURY waits for responses to arrive from all replicas before checking for controllers with equivalent network view and determine the consensus. In effect, as k increases, more responses are required to achieve a majority. If k is constant, an increase in m also increases the detection time, as more time is required to attain majority responses. Based on this empirical evidence, we select the validation timeout as the 95^{th}%ile for each k. Thus, for an ONOS cluster with $n=7$, $k=6$ and $m=0$, the validation timeout is ~97ms. However, in presence of faulty controllers, i.e., $n=7$, $k=6$ and $m=2$, the validation timeout increases to ~129ms. Fig. 4b shows validation times for ONOS with varying PACKET_IN rates for $k=6$ and $m=0$. We observe that with increase in PACKET_IN rate, validation time also increases.

Fig. 4c plots the results for an ODL cluster with similar configurations. We observed the validation timeouts to be ~500ms for $k=6$, $m=0$ and ~700ms for $k=6$, $m=2$, at a PACKET_IN rate of ~500. Note that validation timeouts for ONOS were significantly lower than ODL. This discrepancy in validation timeouts exists because ONOS is much more responsive than ODL even when the controller's FLOW_MOD generation pipeline saturates.

In all these experiments, we observed no inconsistencies in replica states. Thus, validation timeout was based on time to reach consensus amongst $k+1$ responses.

A.1 Controller faults and their detection

We briefly discuss how JURY detects the real faults described in § III-B. We refer interested readers to Appendix for additional fault examples.

(**1**) **ONOS database locking**: A switch is deemed connected after its FEATURES_REPLY is accepted at the controller, following which the controller writes the switch entry into the shared cache. Unlike the primary, secondary controllers on receipt of the FEATURES_REPLY do not lock the cache for writing (since JURY prevents any side-effects of replicated execution). Thus, the validator receives updates from the secondary controllers, but the response from the primary controller times out due to database locking. The lack of taint on responses at the validator helps detect the offending primary controller.

(**2**) **ONOS master election**: Controllers issue periodic LLDP messages to switches under them for topology discovery. However, when the secondary controllers receive a replicated LLDP PACKET_IN after the original master reboots (with a lower ID), they will not participate in liveness tracking since they do not govern the concerned switch, and do not write to the cache. Thus, only updates from the two controllers governing the switches corresponding to the link are received. JURY detects a fault, since these updates differ due to the liveness algorithm.

(**3**) **ODL FLOW_MOD drops**: JURY intercepts all writes to the distributed cache received at secondary controllers and relays them to the validator. Note that FLOW_MOD messages are also written to the network. Thus, cache entries are checked against lack of network write, which identifies the offending controller.

(**4**) **ODL incorrect FLOW_MOD**: JURY is a controller solution, and thus, cannot detect a mismatch in the flow rules installed on the switch and the cache. However, to prevent a system from reaching such an undesirable state, we use a policy that specifies the correct hierarchy of match fields in the cache entry. This specification of the entry fields helps the JURY's validator to detect any cache entry without the specified hierarchy of fields, thereby preventing any mismatch in the hardware and software flow rule entries.

We now describe three synthetic faults and how JURY detects them.

(**1**) **Link failure**: Consider a scenario where an LLDP PACKET_IN triggers an update for a new link in the network topology. However, a faulty controller incorrectly updates the `LinksDB` cache to disable a critical link in the network. This fault is of type $T1$, where an external trigger evokes a response from the controller. JURY detects the fault based on the discrepancies in network and cache side-effects observed amongst cluster nodes.

(**2**) **Undesirable FLOW_MOD**: An administrator issues a FLOW_MOD to a switch in the network, and correct flow rules are written to the cache. However, a faulty controller incorrectly modifies the flow rules and instead issues a FLOW_MOD that drops all packets arriving at the destination switch. This fault is of type $T2$, and works for both local and remote switches. JURY detects the fault by sanity checking the network write against cache updates at other cluster nodes.

(**3**) **Faulty proactive action**: An administrator or controller application incorrectly updates the `LinksDB` cache, which brings down a critical network link. This fault is of class $T3$, where an internal trigger results in a controller writing same entries to the cache and network, and thus cannot be

detected by the cluster itself. JURY validates such actions using administrator-specified policies. For example, a policy prohibiting controller updates to topological state in absence of external triggers would safeguard against such actions.

DETECTION. We use synthetic and adapted scenarios (from the real faults) described above, and measure the detection accuracy with ONOS and ODL controller clusters with 24 Mininet hosts. We wrote a driver program to inject combination of the faults in different parts of the network, and used JURY to validate controller actions in the worst case for cluster size $n = 7$, i.e., full replication ($k = 6$) and two faulty replicas ($m = 2$). We repeated the experiment 10 times and in each case the JURY-enhanced controller successfully detected the fault within ~129ms for ONOS and ~700ms for ODL, well within the validation timeout for the corresponding configuration.

COMPARISON WITH RELATED WORK. JURY's fault detection time for both ONOS and ODL is sub-second even in the worst case ($k=6$, $m=2$). Fleet [52] uses heavy cryptographic mechanisms to both validate and resolve disputes, but has detection times of several seconds. STS [55] and NICE [31] take different approaches to debug OpenFlow networks, but are offline systems. Recent work [37], [45], [46], [48], [50] validates controller correctness against network invariants in order of milliseconds. However, unlike JURY, it does not consider faults (§ III-B) that involve shared state modification in an SDN cluster.

A.2 False alarms with benign traffic

We sanity check JURY's validation by measuring the false alarms generated for an ONOS cluster in presence of benign traffic without any policies enforced. We used a set of publicly available traces representing traffic within an enterprise (LBNL) [12], a university (UNIV) [10] and a cyber-defense exercise (SMIA) [7], and replayed them on our Mininet setup connected to the ONOS controller. Fig. 4d plots the detection times for these three categories of traces. We consider a worst case scenario with two faulty replicas to compute the false positives. Based on the validation timeout for $k=6$ and $m=2$ (i.e., 95$^{\text{th}}$%ile from Fig. 4a), JURY reports a false positive rate of just 0.35% across all three traces.

B. Performance

JURY's overheads would be worst in a reactive controller setup, since it would replicate incoming triggers to other secondary controllers, thereby increasing both compute and network overheads for the cluster. With proactive network management in data centers, as advocated by B4 [42] and NVP [49], JURY would perform significantly better with lower number of false positives.

We perform experiments with both ONOS and ODL [3]. However, in the interest of space, we report results with ONOS only, and contrast with ODL wherever required. Unless specified, references to ONOS and ODL stand for their JURY-enhanced versions.

B.1 Cluster throughput

Overall cluster throughput is affected by how quickly a controller can react to incoming PACKET_IN messages, generate flow entries and dispatch FLOW_MOD packets to the switches for flow setup, under various cluster sizes. We observe the cluster's FLOW_MOD throughput when all nodes experience the same PACKET_IN rate.

PRELIMINARY STUDY WITH CBENCH. We ran Cbench [1] in throughput mode against each controller in a 7 node ONOS cluster, and observed that it quickly throttles each controller, causing the cumulative FLOW_MOD throughput to plummet to zero. Fig. 4e plots the results. The blue line shows the bursty nature of Cbench's PACKET_IN traffic at one of the cluster nodes. The red line indicates the FLOW_MOD throughput, which lags behind the PACKET_IN rate and quickly falls to zero. Analyzing the packet traces, we observed several TCP packets with notifications like "transmission window full" on the switch's end and "zero window" on the controller's end. Both these messages indicate an overwhelmed controller that delays processing of the incoming PACKET_IN stream, subsequently leading to complete loss of the outgoing FLOW_MOD messages. For these reasons, we decided not to use Cbench for measuring cluster throughput.

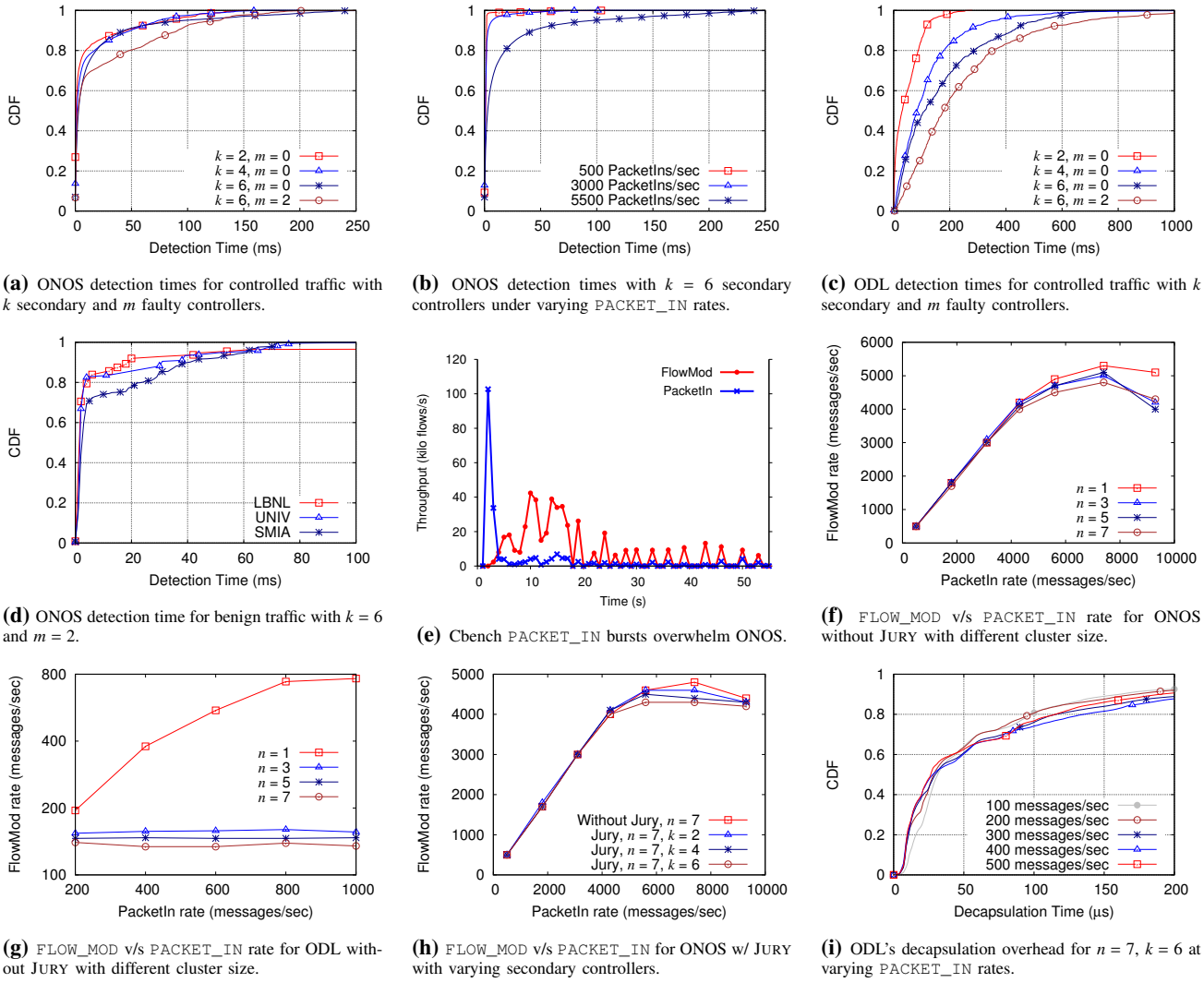
IMPACT OF CLUSTERING ON THROUGHPUT. Due to shortcomings of Cbench, we use tcpreplay [24] to initiate new TCP connections for 10s from several Mininet hosts simultaneously. Each TCP packet results in a TCAM miss, which subsequently generates a PACKET_IN and elicits a FLOW_MOD. Fig. 4f plots the FLOW_MOD throughput for vanilla ONOS with varying PACKET_IN rates under different cluster sizes, i.e., $n = 1, 3, 5, 7$. We observe that the FLOW_MOD throughput has direct correlation with the PACKET_IN rates, and saturates at $\sim 5K$ [4] when the PACKET_IN rate is $\sim 7.5K$. Further, clustering in vanilla ONOS does not affect the FLOW_MOD throughput significantly, and the overhead is <8% with $n = 7$ at the saturation point. We attribute this consistency to ONOS's use of Hazelcast, which uses multicast to deliver messages to the cluster nodes.

In contrast, vanilla ODL's performance (Fig. 4g) is significantly hampered by any amount of clustering. In cluster mode but with a single node ($n = 1$), ODL saturates at a peak FLOW_MOD rate of ~800, and at $n = 7$, it drops down to ~140. Thus, ODL's cluster mode performance is limited by Infinispan. This huge discrepancy in the peak FLOW_MOD throughput for ONOS and ODL can be attributed to their consistency models—ONOS is eventually consistent, while ODL is strongly consistent.

JURY'S IMPACT ON THROUGHPUT. We measure JURY's

[3] ODL is *proactive* and sets up flows in advance so that the controller does not get any PACKET_IN events. Thus, for experiments, we modified ODL to make it *reactive*, as described in § VI-C.

[4] The FLOW_MOD throughput for ONOS is less than that observed by prior work [18], [30], which reports microbenchmark results for batch installation of flow rules when directly invoking APIs to the flow subsystem. In contrast, we measure the performance of the entire packet processing pipeline, including any thread contention and locking effects. Further, unlike JURY, prior work had *flow rule backup* turned off, since ONOS utilizes Hazelcast for backing up flow rules, which can cause high variation in flow rule burst install rate.



(a) ONOS detection times for controlled traffic with k secondary and m faulty controllers.

(b) ONOS detection times with $k = 6$ secondary controllers under varying PACKET_IN rates.

(c) ODL detection times for controlled traffic with k secondary and m faulty controllers.

(d) ONOS detection time for benign traffic with $k = 6$ and $m = 2$.

(e) Cbench PACKET_IN bursts overwhelm ONOS.

(f) FLOW_MOD v/s PACKET_IN rate for ONOS without JURY with different cluster size.

(g) FLOW_MOD v/s PACKET_IN rate for ODL without JURY with different cluster size.

(h) FLOW_MOD v/s PACKET_IN for ONOS w/ JURY with varying secondary controllers.

(i) ODL's decapsulation overhead for $n = 7$, $k = 6$ at varying PACKET_IN rates.

Fig. 4: Accuracy and performance of JURY-enhanced ONOS and ODL.

impact on the cluster throughput by observing the FLOW_MOD rate achieved with $n = 7$, under varying PACKET_IN throughput and replication factors of $k = 2$, 4 and 6. Fig. 4h shows the results of our experiment. Even in the worst case with full replication, i.e., $n = 7$, $k = 6$, we observe that the FLOW_MOD throughput experiences a drop of <11% over the base case of $n = 7$. This drop in FLOW_MOD throughput results from the increased computation in the primary controller due to Hazelcast cache updates initiated by secondary controllers. Thus, JURY is not the bottleneck in eliciting FLOW_MOD messages under different cluster settings. ODL also reported similar overheads across all cluster configurations.

Additionally, we measure the PACKET_OUT throughputs for vanilla ONOS and observe that it is significantly higher than the FLOW_MOD rate, and remains unaffected by any amount of clustering. We also observe that the PACKET_OUT throughput in ONOS saturates at ~220K with Cbench, while the FLOW_MOD throughput peaks at just ~5K. Thus, the controller's FLOW_MOD pipeline is the real bottleneck in achieving higher FLOW_MOD rates. We observed similar discrepancies in PACKET_OUT and FLOW_MOD throughput trends for ODL as well.

B.2 Impact of JURY's pipeline

(1) **Traffic replication.** JURY uses OVS rules to replicate packets to secondary controllers at line speeds. Thus, JURY incurs no performance overheads due to OVS replication. However, JURY does incur network overheads due to replicated PACKET_IN messages, traffic to the validator and inter-controller communication.

We observe that for a 7 node ONOS cluster with full replication ($k = 6$), i.e., switches connect to all controller instances, at PACKET_IN rate of 5.5K, the inter-controller communication (via Hazelcast) extensively dominates the network traffic, reaching 142 Mbps (96.3%). In contrast, with JURY enabled and $k = 2$, 4, 6, traffic due to replicated PACKET_IN messages and the validator (at a PACKET_IN rate of 5.5K) stands at just ~14.2 Mbps (8.8%), ~25.2 Mbps (14.6%) and ~36.1 Mbps (19.6%), respectively. The overhead is less at

lower PACKET_IN rates. Note that JURY replicated PACKET_IN messages induce no side-effects, and thus do not contribute to any inter-controller traffic.

We observe that for $k = 6$ and a PACKET_IN rate of $5.5K$, the secondary controllers send ~4 Mbps worth of Hazelcast messages each to request/notify the primary controller about mastership status of the switches. To ensure high availability (HA) of controllers, it is essential that switches be connected to all or some of the other controllers. Thus, in an actual HA deployment, inter-controller traffic will significantly dominate JURY's network overheads due to replication and validation.

ODL behaves similarly. At PACKET_IN rates of 500 with $n=7$, $k = 6$, inter-controller traffic from Infinispan was 37 Mbps, while JURY overheads were just 12 Mbps.

(**2**) **Action attribution.** ODL must decapsulate secondary PACKET_IN messages received from the OVS (recall § VI). We measured these overheads for different replication factors k, and observed that across all PACKET_IN rates, 80% packets have a decapsulation overhead of <150μs (Fig. 4i). We also compared the absolute time for vanilla ODL's forwarding module and our custom forwarding module to elicit a FLOW_MOD from the instant a relevant PACKET_IN was received. We observed that JURY incurs <1ms overhead at the 95% mark.

(**3**) **Policy validation.** We study the impact of policies on validation time by matching FLOW_MOD responses against a set of simulated policies and those consensuses already approved by the validator. We observed that as the policies increase from 100 to $1K$, the validation time increases linearly from 200μs to 1.2ms. Even with $10K$ policies, JURY takes just 11.2ms for response validation.

VIII. Limitations and Future Work

(**1**) JURY relies on validation timeouts for raising alarms about suspected controller actions. A lower timeout can raise numerous false alarms, while a higher value may result in increased detection times, which may be acceptable if real time detection is a lower priority than false positives. Adaptive timeouts can significantly reduce the number of false alarms in networks with high churn. We leave determination of adaptive timeouts for future work.

(**2**) JURY cannot effectively address all forms of non-determinism in the controller/application logic. The problem can be partially alleviated if it were possible to identify actions from non-deterministic applications.

(**3**) JURY does not provide mechanisms to validate the effectiveness of policies. It is possible that some type $T3$ faults may manifest if policies are not comprehensive.

IX. Related work

VERIFICATION. Fleet [52], unlike JURY, addresses the problem of malicious administrators in SDN clusters. Fleet omits the challenges posed by (a) proactive actions issued by controllers, and (b) cache corruption due to faults, which JURY addresses. Fleet also assumes an administrator can affect only the controller's configuration, and cannot direct a controller to send messages to the switch. Moreover, Fleets requires substantial modifications to ensure correct routing. In contrast, JURY places no restrictions on administrator/controller behavior, and is compatible with existing SDN setups.

VeriCon [29] symbolically verifies if a controller program is correct on all topologies and network events. Machine verified controllers [38] develops a formal language and its runtime that provides guarantees on controller behavior. Anteater [50], Header Space Analysis (HSA) [45], NetPlumber [46], VeriFlow [48], and Sphinx [37] can verify network constraints in real time. However, unlike JURY, they do not validate constraints or anomalies on controller actions in SDN clusters.

NICE [31] uses model checking and symbolic execution of OpenFlow applications to determine invalid system states in an SDN. Unlike JURY, NICE is offline and assumes a singleton controller running a simplified OpenFlow model. Other recent work [51], [56] checks whether an SDN satisfies a given safety property, but limits itself to validating network actions only in singleton setups. In contrast, JURY supports validation of both shared state and network actions in SDN HA clusters.

TROUBLESHOOTING. OFRewind [58] allows record and replay of control plane traffic. NetSight [41] enables retroactive examination of paths taken by OpenFlow data plane packets. STS [55] automatically identifies a minimal sequence of inputs responsible for triggering a given bug. In contrast, JURY detects inconsistent controller actions in HA clusters.

FAULT TOLERANCE. Using state machine replication [32], [33], [59] for tolerating Byzantine faults is expensive and requires high redundancy. While JURY leverages redundancy for replicated execution, it is light-weight since it only detects inconsistencies and does not resolve them. Like prior work [35], [40], [43], [47], JURY uses consensus for robustness and accountability. Akella *et al.* propose to re-architect SDNs for high availability in the presence of network failures [26]. FatTire [54] presents a new language for writing fault-tolerant network programs. Ravana [44] addresses the issue of crash failures and recovery in SDN controllers, while LegoSDN [34] focuses on application-level fault-tolerance caused by software bugs. However, unlike Ravana, LegoSDN does not consider complete controller crash failures. JURY, meanwhile, focuses on response, timing and arbitrary failures in SDN clusters.

ACCOUNTABILITY. PeerReview [40] detects when a node deviates from the expected algorithm. However, unlike JURY, it cannot detect problems due to interactions between multiple nodes, and does not provide diagnostics for what went wrong. Accountable virtual machines (AVMs) [39] ensure correct execution of remote processes, and rely on the server to record all incoming/outgoing messages. JURY also records state/network events, and sends them to an out-of-band validator.

X. Conclusion

We describe JURY, a system that uses consensus amongst cluster nodes with equivalent network view to validate controller activities involving topological and forwarding state in near real time. We have built a prototype of JURY for ONOS and ODL controllers, and our evaluation shows that

JURY is practical, accurate and imposes minimal overheads. We believe that JURY-enhanced clustered controllers provide a more robust mechanism for SDN control as opposed to singleton deployments.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their feedback.

REFERENCES

[1] "Cbench," https://goo.gl/Pb0VTc.
[2] "Cluster members out of sync," https://goo.gl/Sw7GM6.
[3] "Clustering: Failed to deploy flows," https://goo.gl/dM1W4z.
[4] "Connection Mgmt." https://goo.gl/hviSnN.
[5] "FAI - Fully Automatic Installation," http://fai-project.org/.
[6] "FlowRules stay in PENDING-ADD state," https://goo.gl/VkHTnd.
[7] "FOI Data Set." http://goo.gl/IA8cLX.
[8] "Hadoop nodes out of sync," https://goo.gl/plKs1Y.
[9] "High Availability Node Configurations," https://goo.gl/jaAmuW.
[10] "IMC 2010 data set," http://goo.gl/LA5iiG.
[11] "Journal recovery error on restart," https://goo.gl/w6TFjj.
[12] "LBNL/ICSI Enterprise Tracing Project," http://goo.gl/1MKUTO.
[13] "FLOW_MOD lost from MD-SAL to switch," https://goo.gl/vsDAiT.
[14] "Mininet," http://mininet.org/.
[15] "Nodes out of sync at time of re-syncing," https://goo.gl/UF2X1g.
[16] "ODL Helium Clustering: Flow Delete fails," https://goo.gl/erkxxd.
[17] "ODL Lithium: Node does not rejoin," https://goo.gl/jXun6N.
[18] "ONOS Flow Subsystem Burst Throughput," https://goo.gl/Jw7S3l.
[19] "ONOS Link Detection Inconsistent." https://goo.gl/MU6Q5q.
[20] "OSCAR," https://goo.gl/WrGrr2.
[21] "Recovering replicas stuck in init. state," https://goo.gl/MJxgZD.
[22] "Repair out of sync cluster nodes connection," https://goo.gl/plsI0t.
[23] "Specific flow validation in OpenFlow plugin," https://goo.gl/RfnX9d.
[24] "Tcpreplay," http://tcpreplay.synfin.net/.
[25] "YaST - Yet another Setup Tool," http://en.opensuse.org/Portal:YaST.
[26] A. Akella *et al.*, "A Highly Available Software Defined Fabric," in *HotNets'14*.
[27] E. Al-Shaer *et al.*, "FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures," in *SafeConfig'10*.
[28] ——, "Network Configuration in A Box: Towards End to End Verification of Network Reachability & Security," in *ICNP'09*.
[29] T. Ball *et al.*, "VeriCon: Towards Verifying Controller Programs in Software-defined Networks," in *PLDI'14*.
[30] P. Berde *et al.*, "ONOS: Towards an Open, Distributed SDN OS," in *HotSDN'14*.
[31] M. Canini *et al.*, "A NICE Way to Test Openflow Applications," in *NSDI'12*.
[32] M. Castro *et al.*, "BASE: Using Abstraction to Improve Fault Tolerance," *ACM Trans. Comput. Syst.*
[33] ——, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Comput. Syst.*
[34] B. Chandrasekaran *et al.*, "Tolerating SDN Application Failures with LegoSDN," in *HotNets '14*.
[35] B.-G. Chun *et al.*, "Diverse Replication for Single-machine Byzantine-fault Tolerance," in *ATC'08*.
[36] F. Cristian, "Understanding Fault-tolerant Distributed Systems," *Comm. ACM, Feb. 1991*.
[37] M. Dhawan *et al.*, "Sphinx: Detecting Security Attacks in Software-Defined Networks," in *NDSS'15*.
[38] A. Guha *et al.*, "Machine-verified Network Controllers," in *PLDI'13*.
[39] A. Haeberlen *et al.*, "Accountable Virtual Machines," in *OSDI'10*.
[40] ——, "PeerReview: Practical Accountability for Distributed Systems," in *SOSP'07*.
[41] N. Handigol *et al.*, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks," in *NSDI'14*.
[42] S. Jain *et al.*, "B4: Experience with a Globally-deployed Software Defined Wan," in *SIGCOMM '13*.
[43] F. Junqueira *et al.*, "Surviving Internet Catastrophes," in *ATC '05*.
[44] N. Katta *et al.*, "Ravana: Controller Fault-Tolerance in Software-Defined Networking," in *SOSR'15*.
[45] P. Kazemian *et al.*, "Header Space Analysis: Static Checking for Networks," in *NSDI'12*.
[46] ——, "Real Time Network Policy Checking Using Header Space Analysis," in *NSDI'13*.
[47] E. Keller *et al.*, "Virtually Eliminating Router Bugs," in *CoNEXT '09*.
[48] A. Khurshid *et al.*, "VeriFlow: Verifying Network-wide Invariants in Real Time," in *NSDI'13*.
[49] T. Koponen *et al.*, "Network Virtualization in Multi-tenant Datacenters," in *NSDI'14*.
[50] H. Mai *et al.*, "Debugging the Data Plane with Anteater," in *SIGCOMM'11*.
[51] R. Majumdar *et al.*, "Kuai: A Model Checker for Software-defined Networks," in *FMCAD'14*.
[52] S. Matsumoto *et al.*, "Fleet: Defending SDNs from Malicious Administrators," in *HotSDN'14*.
[53] N. McKeown *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev., April 2008*.
[54] M. Reitblatt *et al.*, "FatTire: Declarative Fault Tolerance for Software-defined Networks," in *HotSDN'13*.
[55] C. Scott *et al.*, "Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences," in *SIGCOMM'14*.
[56] D. Sethi *et al.*, "Abstractions for model checking SDN controllers," in *FMCAD'13*.
[57] R. Soulé *et al.*, "Merlin: A Language for Provisioning Network Resources," in *CoNEXT'14*.
[58] A. Wundsam *et al.*, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *ATC'11*.
[59] J. Yin *et al.*, "Separating Agreement from Execution for Byzantine Fault Tolerant Services," *SIGOPS Oper. Syst. Rev.*

APPENDIX

(1) FLOW DELETION FAILURE. ODL reported a byzantine bug [16], wherein with $50K$ flows in the MD-SAL data store, the deletion of flow rules by an administrator failed, and the controller locked up. Further attempts to restart the controller failed with several exceptions. However, other reported experiments with $150K$ flows worked. This fault is of type $T1$, where the administrator uses REST APIs (an external trigger) to initiate flow rule deletion. JURY can detect such faults using cluster consistency, i.e., difference in the responses from the primary and secondary controllers.

(2) LINK DETECTION INCONSISTENT. ONOS sometimes fails to detect all links in a network, specially both before and after a topology change. The number and type of links found in each run is variable, and re-runs occasionally yield different results. It was suspected that the issue was likely due to threading conflicts [19]. The above fault is of type $T1$ and can easily be detected via controller consistency.

(3) FLOW INSTANTIATION FAILURE. In ODL Helium, when trying to deploy flow entries to a switch from a controller application using `restconf`, the API returned success. However, no FLOW_MOD messages were sent from controller and no flows were installed on the switch [3]. The above fault is of type $T2$, where the flow entry modified in the data store was not sent out over the network. JURY can detect such inconsistencies since secondary nodes would receive cache updates, while no FLOW_MOD would be seen on the network.

(4) FLOW RULES STAY IN PENDING_ADD STATE. ONOS reports flow rules in PENDING_ADD state with a switch for a particular optical technology [6]. For a flow entry to go from PENDING_ADD to ADDED, ONOS compares flow rules in its store and flow entries from the switch. In case of any inconsistency, the rules remain in PENDING_ADD state. This fault is of type $T2$ and JURY detects it using controller consistency.