Model Checking Software-Defined Networks with Flow Entries that Time Out

Vasileios Klimis(iD), George Parisis(iD) and Bernhard Reus(iD)

University of Sussex, UK

{v.klimis, g.parisis, bernhard}@sussex.ac.uk

Abstract—Software-defined networking (SDN) enables advanced operation and management of network deployments through (virtually) centralised, programmable controllers, which deploy network functionality by installing rules in the flow tables of network switches. Although this is a powerful abstraction, buggy controller functionality could lead to severe service disruption and security loopholes, motivating the need for (semi-)automated tools to find, or even verify absence of, bugs. Model checking SDNs has been proposed in the literature, but none of the existing approaches can support dynamic network deployments, where flow entries expire due to timeouts. This is necessary for automatically refreshing (and eliminating stale) state in the network (termed as *soft-state* in the network protocol design nomenclature), which is important for scaling up applications or recovering from failures. In this paper, we extend our model (MoCS) to deal with timeouts of flow table entries, thus supporting soft state in the network. Optimisations are proposed that are tailored to this extension. We evaluate the performance of the proposed model in UPPAAL using a load balancer and firewall in network topologies of varying size.

I. Introduction

Software-defined networking (SDN) [1] revolutionised network operation and management along with future protocol design; a virtually centralised and programmable controller 'programs' network switches through interactions (standardised in OpenFlow [2]) that alter switches' flow tables. In turn, switches push packets to the controller when they do not store state relevant to forwarding these packets. Such a paradigm departure from traditional networks enables the rapid development of advanced and diverse network functionality; e.g., in designing next-generation inter-data centre traffic engineering [3], load balancing [4], firewalls [5] and Internet exchange points (IXPs) [6]. Although this is a powerful abstraction, buggy controller functionality could lead to severe service disruption and security loopholes. This has led to a significant amount of research on SDN verification and/or bug finding, including static network analysis [7], [8], [9], dynamic real-time bug finding [10], [11], [12], [13], and formal verification approaches, including symbolic execution [14], [15], [16] and model checking [17], [10], [16], [18] methods. A comprehensive review of existing approaches along with their shortcomings can be found in [19].

Model checking is a renowned automated technique for hardware and software verification and existing model checking approaches for SDNs have shown promising results with respect to scalability and model expressivity, in terms of supporting realistic network deployments and the OpenFlow

standard. However, a key limitation of all existing approaches is that they cannot model forwarding state (added in network switches' flow tables by the controller) that expires and gets deleted. Without this, one cannot model nor verify the correctness of SDNs with soft-state which is prominent in the design of protocols and systems that are resilient to failures and scalable; e.g., as in [20], where flow scheduling is on a per-flow basis, and numerous network protocols where in-network state is not explicitly removed but expires, so that overhead is minimised [21].

In this paper, we extend our model (MoCS) [17] to support soft-state, complying with the OpenFlow specification, by allowing flow entries to time out and be deleted. We propose relevant optimisations (as in [17]) in order to improve verification performance and scalability. We evaluate the performance of the proposed model extensions in UPPAAL using a load balancer and firewall in network topologies of varying size.

II. MoCS SDN Model

The MoCS model [17] is formally defined by means of an action-deterministic transition system. We parameterise the model by the underlying network topology, λ, and the controller program, CP, in use. The model is a 6-tuple $\mathcal{M}_{(\lambda, \text{CP})} = (S, s_0, A, \hookrightarrow, AP, L)$, where S is the set of all states the SDN may enter, s_0 the initial state, A the set of actions which encode the events the network may engage in, $\hookrightarrow \subseteq S \times A \times S$ the transition relation describing which execution steps the system undergoes as it perform actions, AP a set of atomic propositions describing relevant state properties, and $L : S \to 2^{AP}$ is a labelling function, which relates to any state $s \in S$ a set $L(s) \in 2^{AP}$ of those atomic propositions that are true for s. Such an SDN model is composed of several smaller systems, which model network components (hosts[1], switches and the controller) that communicate via queues and, combined, give rise to the definition of $\hookrightarrow$. A detailed description of MoCS' components and transitions can be found in [17]. Due to lack of space, in this paper, we only discuss aspects of the model that are required to understand and verify the soundness of the proposed model extensions, and examples used in the evaluation section. Figure 1 illustrates a high-level view of OpenFlow interactions, modelled actions and queues, including the proposed extensions discussed in Section III.

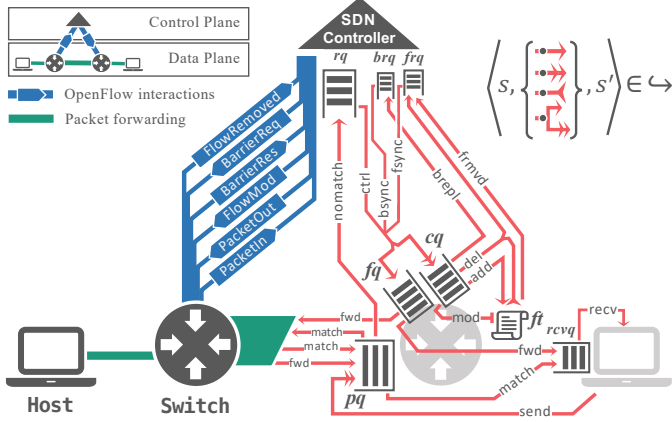[1]A host can act as a client and/or server.

Fig. 1: A high-level view of OpenFlow interactions (left half) and modelled actions (right half). A red solid-line arrow depicts an action which, when fired, (1) dequeues an item from the queue the arrow begins at, and (2) (possibly) adds an item in the queue the arrowhead points to (or multiple items if the arrow is double-headed). Deleting an item from the target queue is denoted by a reverse arrowhead; modifying in, by a hammerhead. A forked arrow denotes (possibly) multiple targeted queues.

States and queues: A *state* is a triple (π, δ, γ), where π is a family of hosts, each consisting of a receive queue (*rcvq*); δ is a family of switches, consisting of a switch packet queue (*pq*), switch forward queue (*fq*), switch control queue (*cq*), switch flow table (*ft*); γ consists of the local controller program state $cs \in CS$, and a family of controller queues: request queue (*rq*), barrier-reply queue (*brq*) and flow-removed queue (*frq*). So π and δ describe the *data-plane*, and γ the *control plane*. The network components communicate via the shared queues. Each transition models a certain network event that will involve some of the queues, and maybe some other network state. Concurrency is modelled through interleavings of those events.
Transitions: Each transition is labelled with an action $\alpha \in A$ that indicates the nature of the network event. We write $s \xrightarrow{\alpha} s'$ and $(s, \alpha, s') \in \hookrightarrow$ interchangeably to denote that the network moved from state s to s' by executing transition α. The parts of the network involved in each individual α, i.e. *packets, rules, barriers, switches, hosts, ports* and *controller states*, are included in the transition label as parameters; e.g., $match(sw, pkt, r) \in A$ denotes the action that switch sw matches packet pkt by rule r and, as a result, forwards it accordingly, leading to a new state after transition.
Atomic propositions: The propositions in AP are statements on (1) controller program states, denoted by $Q(q)$ which expresses that the controller program is in state $q \in CS$, allowing one to reason about the controller's internal data structures, and (2) packet header fields – those packets may be in any switch buffer *pq* or host buffer *rcvq* (but no other buffers). For instance, $\exists pkt \in sw.pq . P(pkt)$ is a legitimate atomic proposition that states that there is a packet in sw's packet queue that satisfies packet *pkt* property P.
Topology: λ describes the network topology as a bijective map

which associates one network interface (a pair of networking device and physical port) to another.
Specification Logic: The properties of the SDNs to be checked in this paper are safety properties, expressed in linear-time temporal logic without 'next-step' operator, $LTL_{\backslash\{\bigcirc\}}$. We have enriched the logic by modal operators of *dynamic logic* [22], allowing formula construct of the form $[\alpha(\vec{x})]P$ stating that whenever an event $\alpha(\vec{x})$ happened, P must hold. Note that P may contain variables from x. This extension is syntax sugar in the sense that the formulae may be expressed by additional state; e.g., $[match(sw, pkt, r)](r.fwdPort = \text{drop})$ states that if *match* happened, it was via a rule that dropped the packet. This permits specification formulae to be interpreted not only over states, but also over actions that have happened. The model checking problem then, for an SDN model $\mathcal{M}_{(\lambda, \text{CP})}$ with a given topology λ, a control program CP and a formula φ of the specification logic as described above, boils down to checking whether all runs of $\mathcal{M}_{(\lambda, \text{CP})}$ satisfy φ, short $\mathcal{M}_{(\lambda, \text{CP})} \models \Box\varphi$.
SDN Operation: End-hosts send and receive packets (*send* and *recv* actions in Figure 1) and switches process incoming packets by matching them (or failing to) with a flow table entry (rule). In the former case (*match* action), the packet is forwarded as prescribed by the rule. In the opposite case (*nomatch* action), the packet is sent to the controller (*PacketIn* message on the left side of Figure 1). The controller's packet handler is executed in response to incoming *PacketIn* messages; as a result of its execution, its local state may change, a number of packets (*PacketOut* message) and rule updates (*FlowMod* message), interleaved with barriers (*BarrierReq* message), may be sent to network switches. Network switches react to incoming controller messages; they forward packets sent by the controller as specified in the respective *PacketOut* message (*fwd* action), update their own forwarding tables (*add/del* actions), respecting set barriers and notifying the controller (*BarrierRes* message) when said barriers are executed (*brepl* action). Finally, upon receiving a *BarrierRes* message, the controller executes the respective handler (*bsync* action), which can result in the same effects as the *PacketIn* message handler.
Abstractions: To obtain finitely representable states, all queues in the model must be finitely representable. For packet queues we use multisets, subject to $(0, \infty)$ abstraction [23]; a packet either does not appear in the queue or appears an unbounded number of times. The other queues are simply modelled as finite sets. Modelling queues as sets means that entries are not processed in the order of arrival. This is intentional for packet queues but for controller queues this may limit behaviour unless the controller program is order-insensitive. We focus on those controller programs in this paper.

III. MODELLING FLOW ENTRY TIMEOUTS

In order to model soft-state in the network, we enrich our model with two new actions that model flow entry timeouts and subsequent handling of these timeouts by the controller

program. Note that in our model, timeouts are not triggered by any kind of clock; instead, they are modelled through the interleaving of actions in the underlying transition system that ensure that flow removal (and subsequent handling by the controller program) will appear as it would for any possible value of a timeout in a real system.

The new actions are defined as follows: $frmvd(sw, r)$ models the timeout event, as an action in the transition system that removes the flow entry (rule) r from switch sw and notifies the controller by placing a *FlowRemoved* message (see Figure 1) in the respective queue (*frq*). The $fsync(sw, r, cs)$ action models the call to the *FlowRemoved* message handler. As a result of the handler execution, the controller's local state (*cs*) may change, a number of packets (*PacketOut* messages) and rule updates (*FlowMod* messages), interleaved with barriers (*BarrierReq* message), may be sent to network switches. In order to model timeouts, rules are augmented with a *timeout* bit which, when true, signals that the installed rule can be removed at any time, i.e., the *frmvd*-action can be interleaved, in any order, with any other action that is enabled at any state later than the installation of this rule.

To support our examples, we add to the set of *FlowMod* messages a *modify flow entry* instruction. In [17] we only used $add(sw, r)$ and $del(sw, r)$ messages, for installing and deleting rule r at switch sw, respectively. We now add $mod(sw, f, a)$ to these messages. This instructs switch sw that if a rule is found in $sw.ft$ that matches field f, its forwarding actions are modified by a. If no such rule exists, $mod(\cdot)$ does not do anything.

Optimisation: To tackle the state-space explosion, we exploit the fact that some traces are observationally (w.r.t. the property to be proved) equivalent, so that only one of those needs to be checked. This technique, referred to as *partial-order reduction* (POR) [24], reduces the number of interleavings (traces) one has to check. To prove equivalence of traces, one needs actions to be permutable and invisible to the property at hand. This is the motivation for the following definition:

Definition 1 (SAFE ACTIONS) Given a context CTX = $(\text{CP}, \lambda, \varphi)$, and SDN model $\mathcal{M}_{(\lambda, \text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$, an action $\alpha(\cdot) \in A(s)$ is called *safe* if it is (1) *independent* of any other action β in A, i.e. executing α after β leads to the same state as running β after α, and (2) *unobservable* for φ (also called φ-*invariant*), i.e., $s \models \varphi$ iff $\alpha(s) \models \varphi$ for all $s \in S$ with $\alpha \in A(s)$.

The following property of controller programs is needed to show safety:

Definition 2 (ORDER-SENSITIVE CONTROLLER PROGRAM) A controller program CP is order-sensitive if there exists a state $s \in S$ and two actions α, β in $\{ctrl(\cdot), bsync(\cdot), fsync(\cdot)\}$ such that $\alpha, \beta \in A(s)$ and $s \xrightarrow{\alpha} s_1 \xrightarrow{\beta} s_2$ and $s \xrightarrow{\beta} s_3 \xrightarrow{\alpha} s_4$ with $s_2 \neq s_4$.

In [17] we already showed that certain actions are safe and can be used for PORs. We now show that the new $fsync(\cdot)$ action is safe on certain conditions.

Lemma 1 (SAFENESS PREDICATES FOR *fsync*) For transition system $\mathcal{M}_{(\lambda, \text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$ and a formula $\varphi \in \text{LTL}_{\setminus\{\bigcirc\}}$, $\alpha = fsync(sw, r, cs)$ is safe iff the following two conditions are satisfied:

Independence CP is not order-sensitive
Invisibility if $Q(q)$ in AP occurs in φ, then α is φ-invariant

Proof. See [25] Appendix A. □

Given a context CTX = $(\text{CP}, \lambda, \varphi)$ and an SDN network model $\mathcal{M}_{(\lambda, \text{CP})} = (S, A, \hookrightarrow, s_0, AP, L)$, for each state $s \in S$ define $ample(s)$ as follows: if $\{\alpha \in A(s) \mid \alpha \text{ safe }\} \neq \varnothing$, then $ample(s) = \{\alpha \in A(s) \mid \alpha \text{ safe }\}$; otherwise $ample(s) = A(s)$. Next, we define $\mathcal{M}^{fr}_{(\lambda, \text{CP})} = (S^{fr}, A, \hookrightarrow_{fr}, s_0, AP, L^{fr})$, where $S^{fr} \subseteq S$ the set of states reachable from the initial state s_0 under $\hookrightarrow_{fr}$, $L^{fr}(s) = L(s)$ for all $s \in S^{fr}$ and $\hookrightarrow_{fr} \subseteq S^{fr} \times A \times S^{fr}$ is defined inductively by the rule:

$$\frac{s \xrightarrow{\alpha} s'}{s \xrightarrow{\alpha}_{fr} s'} \quad \text{if } \alpha \in ample(s)$$

Now we can proceed to extend the POR Theorem of [17]:

Theorem 1 (FLOW-REMOVED EQUIVALENCE) Given a property $\varphi \in \text{LTL}_{\setminus\{\bigcirc\}}$, it holds that $\mathcal{M}^{fr}_{(\lambda, \text{CP})}$ satisfies φ iff $\mathcal{M}_{(\lambda, \text{CP})}$ satisfies φ.

The proof is a consequence of Lemma 1 applied to the proof of Theorem 2 in [17]. See [25] Appendix A for a detailed proof.

IV. EXPERIMENTAL EVALUATION

In this section we experimentally evaluate the proposed extensions in terms of verification performance and scalability. We use a realistic controller program that enables a network switch to act both as a load balancer and stateful firewall (see §V-CP1). The load balancer keeps track of the active sessions between clients and servers in the cluster (see Figure 2), while, at the same time, only allowing specific clients to access the cluster. Soft state is employed here so that flow entries for completed sessions (that were previously admitted by the firewall) time out and are deleted by the switch without having to explicitly monitor the sessions and introduce unnecessary signalling (and overhead). In the underlying SDN model, the *frmvd* action is fired, which, in turn, deletes the flow entry from the switch's table and notifies the controller of that. This enables the *fsync* action that calls the flow removal handler.
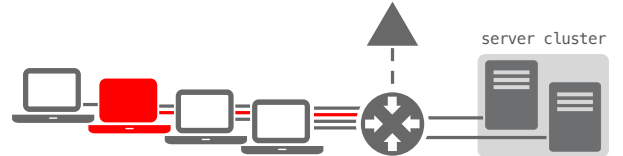

Fig. 2: Four clients and two servers connecting to an OF-switch. ■ is not white-listed.

A session is initiated by a client which sends a packet (*pkt* in §V-CP1) to a known cluster address; servers are

not directly visible to the client. Sessions are bi-directional therefore the controller must install respective rules to the switch to allow traffic to and from the cluster. The property that is checked here is that (1) the traffic (i.e. number of sessions, assuming they all produce similar traffic patterns), and resulting load, is uniformly distributed to all available servers, and (2) that traffic from non-whitelisted clients is blocked. More concretely, "*a packet from a 'dodgy' address should never reach the servers, and the difference between the number of assigned sessions at each server should never be greater than 1*", formally,

$$\Box \ \big(\forall s_i, s_j \in Servers \ \forall pkt \in s_i.rcvq \ .$$
$$\neg pkt.src = dodgy \ \wedge \ \big|sLoad[s_i] - sLoad[s_j]\big| < \ 2\big) \quad (\varphi)$$

where $sLoad$ stores the active session count for each server.

In the first (buggy) version of the controller's packet handler (shaded grey in §V-CP1) and flow removal handler §V-CP2, the controller program assigns new sessions to servers in a round-robin fashion and keeps track of the active sessions (array *deplSessions* in the provided pseudocode). When a session expires, the respective flow table entry is expected to expire and be deleted by the switch without any signalling between the controller, clients or servers[2]. As stated above, this controller program does not satisfy safety property φ because the controller does nothing to rebalance the load when a session expires. Our model implementation[3] discovered the bug in the topology shown in Figure 2 with 3 sessions in 11ms exploring 202 states.

In the second (still buggy) version of the controller, session scheduling is more sophisticated (shaded blue in §V-CP1); a session is assigned to the server with the least number of active sessions. Although the updated load balancing algorithm does keep track of the active sessions per server, this controller is still buggy because no rebalancing takes place when sessions expire. In a topology of 4 clients and 2 servers, we were able to discover the bug in 52ms after exploring 714 states.

We fix the bug by allowing the controller program to rebalance the active sessions, when (1) a session expires and (2) the load is about to get out of balance, by moving one session from the most-loaded to the least-loaded server (§V-CP3). In the same topology as above, we verified the property in 625ms after exploring 15068 states.[4]

Next, we evaluate the performance of the proposed model and extensions for verifying the correctness of the property in a given SDN. We do that by verifying φ with the correct controller program, discussed above, and scaling up the topology in terms of clients, servers and active sessions. Results are listed in Table I and state exploration is illustrated in Figure 4.

Table I lists performance of the model checker for verifying the correct controller program with PORs disabled on the

[2]It is worth stressing that modelling such functionality is not supported by existing model checking approaches, such as [17] and [18], where flow table entries can only be explicitly deleted by the controller.

[3]UPPAAL [26] is the back-end verification engine for MoCS and all experiments were run on an 18-Core iMac pro, 2.3GHz Intel Xeon W with 128GB DDR4 memory.

[4]Note that the *fsync*-optimisation was not enabled in the examples above.

left and with PORs enabled on the right, respectively. For each chosen topology we list the number of states explored, CPU time used, and memory used. The topology is shaped as in Figure 2, and parametrised by the number of clients (ranging from 3 to 5) and servers (ranging from 2 to 5), as indicated in Table I. The number of required packets and rules, respectively, is shown in grey. These numbers are always uniquely determined by the choice of topology. Where there are no entries in the table (indicated by a dash) the verification did not terminate within 24 hours.

The results clearly show that the verification scales well with the number of servers but not with the number of clients. The reason for the latter is that for each additional client an additional packet is sent, which, according to programs §V-CP1 and CP3, leads to 7 additional actions without timeouts and to 12 with timeouts. The causal ordering of these actions is shown in Fig. 3. The sub-branch in red shows the actions that appear due to a timeout of the added rule. Thus, the number of states is exponential in the number of clients: every new action in Fig. 3 leads to a new change of state, thus doubling the possible number of states. This exponential blow-up happens whether we have timeouts or not. With timeouts, however, we have worse exponential complexity as there are more new states generated.
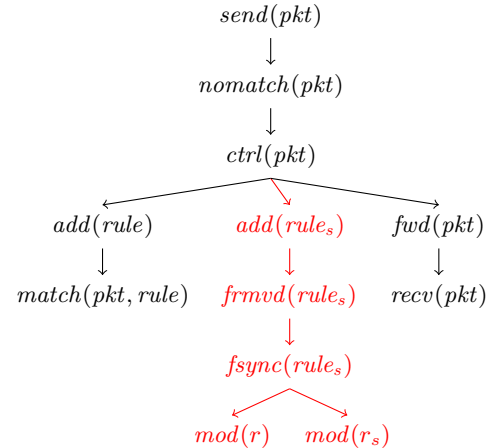
$$send(pkt)$$
$$\downarrow$$
$$nomatch(pkt)$$
$$\downarrow$$
$$ctrl(pkt)$$

$$add(rule) \qquad add(rule_s) \qquad fwd(pkt)$$
$$\downarrow \qquad\qquad \downarrow \qquad\qquad \downarrow$$
$$match(pkt, rule) \quad frmvd(rule_s) \quad recv(pkt)$$
$$\downarrow$$
$$fsync(rule_s)$$

$$mod(r) \qquad mod(r_s)$$

Fig. 3: The causal enabling relation between actions for an additional packet *pkt*; only the relevant arguments are shown using the same nomenclature as in the pseudocode.

The results also demonstrate that, for network setups with three clients, the POR optimisation reduces the state space – and thus the verification time – by about half. For more clients the reduction is far more significant, given that the verification of the unoptimised model did not terminate within 24 hours. This is not surprising as the number of possible interleavings is massively increased by the non-deterministic timeout events.

V. CONTROLLER PROGRAMS

CP1 implements the *PacketIn* message handler that processes packets sent by switches when the *nomatch* action is fired. The two different versions of functionality discussed in the paper are defined by the *leastConnectionsScheduling*
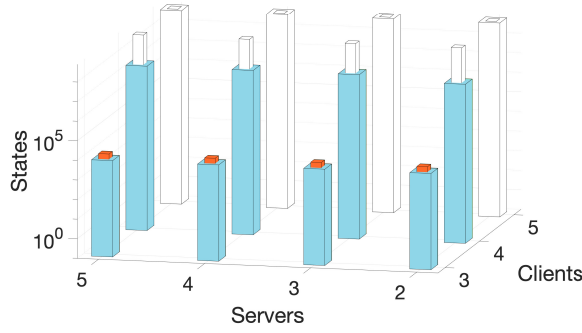
Fig. 4: Explored States (logarithmic scale). Wide bars represent the optimised model and narrow ones (inside) the unoptimised model. Uncoloured bars represent non-termination.

TABLE I: Performance by number of clients and servers

Clients	Servers	Packets	Rules	without POR			with POR		
				States	CPU user time	Resident memory [KiB]	States	CPU user time	Resident memory [KiB]
3	2	3	13	15,068	553ms	9,516	8,264	317ms	9,016
3	3	3	19	15,068	700ms	10,688	8,264	322ms	8,792
3	4	3	25	15,068	841ms	11,936	8,264	483ms	10,488
3	5	3	31	15,068	987ms	15,280	8,264	563ms	12,844
4	2	4	17	–	–	–	13,244,474	13.2m	2,508,528
4	3	4	25	–	–	–	24,623,435	30.77m	5,432,004
4	4	4	33	–	–	–	24,623,435	37.23m	13,129,916
4	5	4	41	–	–	–	24,623,435	42.64m	15,443,136
5	2	5	21	–	–	–	–	–	–

constant. When *leastConnectionsScheduling* is false, server selection is done in a round-robin fashion, whereas, in the opposite case, the controller assigns the new session to the server with the least number of active sessions.

CP2 implements the naive (and buggy) *FlowRemoved* message handler. When soft state expires in the network, the handler merely updates its local state to reflect the update in the load.

CP3 implements a more sophisticated (and correct) *FlowRemoved* message handler. When soft state expires in the network, the handler updates its local state to reflect the update in the load and re-assigns active sessions from the most to the least loaded server, by updating the flow table of the switch accordingly.

VI. CONCLUSION AND FUTURE WORK

We have proposed model checking of SDN networks with flow entries (rules) that time out. Timeouts pose problems due to the great number of resulting interleavings to be explored. Our approach is the first one to deal with timeouts, exploiting partial-order reductions, and performing reasonably well for small networks. We demonstrated that bug finding works well for SDN networks in the presence of flow entry timeouts. Future work includes exploring flow removals with timeouts that are constrained by integer to *enforce certain orderings* of timeout messages as well as improvements in performance, for instance, by using bounded model checking tools for concurrent programs.

Controller Program CP 1: *PacketIn* Message Handler

```
1:  handler pktIn(pkt, sw)
2:    if pkt.srcIP ≠ dodgy_client then
3:      if ¬deplSessions[pkt.srcIP] then
4:        if ¬leastConnectionsScheduling then
          // Round-Robin rotation
5:          server ← server mod 2 + 1
6:        else
          // Least-Connections scheduling
7:          server ← min(sLoad[])
8:        end if
        // Initialisation of flow to server
9:        rule.srcIP         ← pkt.srcIP
10:       rule.in_port       ← pkt.in_port
11:       rule.fwdPort       ← server
        // Initialisation of symmetric rule_s
12:       rule_s.srcIP       ← server
13:       rule_s.destIP      ← pkt.srcIP
14:       rule_s.fwdPort     ← pkt.in_port
15:       rule_s.timeout     ← true
        // Initialisation of drop rule rule_d
16:       rule_d.srcIP       ← dodgy_client
17:       rule_d.fwdPort     ← drop
        // Deployment of rules
18:       send_message(FlowMod(add(rule)), sw)
19:       send_message(FlowMod(add(rule_s)), sw)
20:       send_message(FlowMod(add(rule_d)), sw)
        // Update firewall state table
21:       sLoad[server]++
22:       deplSessions[pkt.srcIP] ← true
23:     end if
      // PacketOut: sending pkt out through sw
24:     send_message{PacketOut(pkt, server), sw)
25:   end if
26: end handler
```

Controller Program CP 2: Naive *FlowRemoved* message handler

```
1:  handler flowRmvd(rule_s, sw)
2:    sLoad[rule_s.srcIP]--
3:    deplSessions[rule_s.destIP] ← false
4:  end handler
```

Controller Program CP 3: Correct *FlowRemoved* message handler

```
1:  handler flowRmvd(rule_s, sw)
2:    sLoad[rule_s.srcIP]--
3:    deplSessions[rule_s.destIP] ← false
4:    if max(sLoad[]) − min(sLoad[]) > 1 then
5:      r ← the rule in sw.ft with fwdPort = max(sLoad[])
6:      r_s ← symmetric rule of r
7:      cm ← mod(r, fwdPort ← min(sLoad[]))
8:      cm_s ← mod(r_s, srcIP ← min(sLoad[]))
9:      send_message(FlowMod(cm, sw))
10:     send_message(FlowMod(cm_s, sw))
11:     sLoad[max(sLoad[])]--
12:     sLoad[min(sLoad[])]++
13:   end if
14: end handler
```

REFERENCES

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN," *SIGCOMM Computer Communication Review*, 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[3] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: scaling flow management for high-performance networks," *SIGCOMM*, 2011.

[4] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using OpenFlow," *SIGCOMM*, 2009.

[5] H. Hu, G.-J. Ahn, W. Han, and Z. Zhao, "Towards a Reliable SDN Firewall," in *ONS*, 2014.

[6] N. Feamster, J. Rexford, S. Shenker, R. Clark, R. Hutchins, D. Levin, and J. Bailey, "SDX: A software-defined Internet exchange," *Open Networking Summit*, 2013.

[7] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*, 2011.

[8] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.

[9] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon: Towards Verifying Controller Programs in Software-defined Networks," in *PLDI*, 2014.

[10] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *PLDI*, 2015.

[11] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *POPL*, 2016.

[12] A. Horn, A. Kheradmand, and M. R. Prasad, "Delta-net: Real-time Network Verification Using Atoms," in *NSDI*, 2017.

[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *NSDI*, 2013.

[14] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *SIGCOMM*, 2016.

[15] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test Openflow Applications," in *NSDI*, 2012.

[16] Y. Jia, "NetSMC : A Symbolic Model Checker for Stateful Network Verification," in *NSDI*, 2020.

[17] V. Klimis, G. Parisis, and B. Reus, "Towards Model Checking Real-World Software-Defined Networks," in *CAV*, 2020.

[18] R. Majumdar, S. Deep Tetali, and Z. Wang, "Kuai: A model checker for software-defined networks," in *FMCAD*, 2014.

[19] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang, "A survey on network verification and testing with formal methods: Approaches and challenges," *IEEE Surveys & Tutorials*, 2019.

[20] M. Al-Fares, S. Radhakrishnan, and B. Raghavan, "Hedera: Dynamic Flow Scheduling for Data Center Networks." in *NSDI*, 2010.

[21] P. Ji, Z. Ge, J. Kurose, and D. Towsley, "A comparison of hard-state and soft-state signaling protocols," *IEEE/ACM Transactions on Networking*, 2007.

[22] V. R. Pratt, "Semantical considerations on Floyd-Hoare logic," in *FOCS*, 1976.

[23] A. Pnueli, J. Xu, and L. Zuck, "Liveness with $(0, 1, \infty)$-counter abstraction," in *CAV*, 2002.

[24] D. Peled, "All from one, one for all: on model checking using representatives," in *CAV*, 1993.

[25] V. Klimis, G. Parisis, and B. Reus, "Model Checking Software-Defined Networks with Flow Entries that Time Out (version with appendix)," *arXiv: 2008.06149 [cs.NI]*, 2020.

[26] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Developing UPPAAL over 15 years," *Software: Practice and Experience*, 2011.