Proceedings of the 2016 Winter Simulation Conference
T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, eds.

CONVENUS: CONGESTION VERIFICATION OF NETWORK UPDATES IN SOFTWARE-DEFINED NETWORKS

Xin Liu
Dong Jin

Department of Computer Science
Illinois Institute of Technology
10 West 31st Street
Chicago, IL, USA

Cheol Won Lee
Jong Cheol Moon

National Security Research Institute
1559, Yuseong-daero, Yuseong-gu
Daejeon, SOUTH KOREA

ABSTRACT

We present *ConVenus*, a system that performs rapid congestion verification of network updates in software-defined networks. *ConVenus* is a lightweight middleware between the SDN controller and network devices, and is capable to intercept flow updates from the controller and verify whether the amount of traffic in any links and switches exceeds the desired capacity. To enable online verification, *ConVenus* dynamically identifies the minimum set of flows and switches that are affected by each flow update, and creates a compact network model. *ConVenus* uses a four-phase simulation algorithm to quickly compute the throughput of every flow in the network model and report network congestion. The experimental results demonstrate that *ConVenus* manages to verify 90% of the updates in a network consisting of over 500 hosts and 80 switches within 5 milliseconds.

1 INTRODUCTION

The growing and rapid adoption of software-defined networking (SDN) architectures enables fast innovation of modern network applications. On one hand, the logically-centralized control and direct network programmability offered by SDN simplifies the network application design. On the other hand, SDN allows multiple users and applications (potentially complex and error-prone and unaware of each other) to concurrently operate the same physical network. It is critical to verify that the network preserves the desired behaviors, such as congestion-freedom, by eliminating the conflicting or incorrect rules from the application layer.

Researchers investigate techniques to analyze the network configurations or the static snapshots of the network state to discover bugs and errors, but those approaches do not scale well due to the exponentially grown problem space, and thus they typically operate offline. Online verification tools are also explored to check dynamic snapshots with the focus on the network-layer connectivity (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013, Zhou, Jin, Croft, Caesar, and Godfrey 2015), but not on the network congestion, which could lead to network performance degradation and system security breaches. In this paper, our goal is to efficiently perform network congestion verification in the context of SDN as the network state evolves in real time. We present *ConVenus*, a system for **Con**gestion **Ve**rification of **N**etwork **U**pdates in **S**oftware-defined Networks. *ConVenus* sits between the SDN controller and the network layer, and it intercepts each update from the controller and verifies whether the congestion-free property still holds before applying the update to the network. *ConVenus* can raise alarms immediately, or even block the updates that violate the congestion-freedom invariant.

The core design of *ConVenus* is based on the dynamic data-driven application system (DDDAS) paradigm (Darema 2004) that involves dynamically incorporating real-time data (e.g., flow updates and

network states) into computations (in particular, flow rate estimation through a novel efficient four-phase simulation algorithm) in order to steer the verification process in an SDN-based application system. To address the challenges of real-time verification, *ConVenus* is designed to be stateful and incremental to speed up the verification process. It maintains a compact network model with the flow states, and dynamically refines the model as the network state evolves, by extracting the minimum set of network elements affected by the new flow update, including flows and switches. The efficient problem space reduction enables *ConVenus* to achieve high verification speed. We develop a prototype system of *ConVenus* and perform extensive evaluation on a campus network topology with 80 switches and 500+ hosts. The experimental results show that 90% of the update verification take less than 5 ms to complete with 10 ms as the upper bound.

The remainder of the paper is organized as follows. Section 2 introduces the background of SDN with the related work on network verification and traffic engineering. Section 3 overviews the architecture of *ConVenus*. Section 4 presents the network modeling and the simulation algorithm for congestion verification. Section 5 describes the generation of minimum affected network model to speed up the verification. Section 6 performs evaluation of *ConVenus*, and Section 7 concludes the paper with future work.

2 BACKGROUND AND RELATED WORK

SDN is an emerging computer network architecture. It decouples the control and forwarding functions in the traditional network devices and centralizes the control logic in the SDN controller(s) (ONF 2016). The new architecture enables direct programmability and global visibility of the network. Users can develop complex network applications using high-level languages that are compiled by the controller into a set of low-level instructions for the hardware devices. While SDN is a trending technology to enable rapid innovation in computer networks, there are still numerous challenges that the research community must address. One key challenge is how to efficiently verify and debug network applications, because SDN-based networks are still complex distributed systems. To address the issue, researchers statically analyze snapshots of the network state to detect system faults (Cadar, Dunbar, and Engler 2008, Mai, Khurshid, Agarwal, Caesar, Godfrey, and King 2011, Kazemian, Varghese, and McKeown 2012). However, those approaches operate offline, and thus find bugs only after they occur. Online verification tools are also developed (Kazemian, Chang, Zeng, Varghese, McKeown, and Whyte 2013, Khurshid, Zou, Zhou, Caesar, and Godfrey 2013, Zhou, Jin, Croft, Caesar, and Godfrey 2015) to check dynamic snapshots, but they focus on reachability-based invariants, such as loop-freedom, not about network congestion. To handle congestion-free updates, zUpdate uses an optimization programming model (Liu, Wu, Zhang, Yuan, Wattenhofer, and Maltz 2013) and Dionysus uses dynamic scheduling atop a consistency-preserving dependency graph (Jin, Liu, Gandhi, Kandula, Mahajan, Zhang, Rexford, and Wattenhofer 2014). The difference between those works and *ConVenus* are that (1) *ConVenus* has much faster verification speed and is designed for online congestion verification, and (2) we do not consider the transient changes in network updates, which we plan to explore in the future. In addition, the global visibility and uniform southbound interfaces offered by SDN also enable efficient traffic engineering to prevent congestion. Existing works focus on flow management, fault tolerance, topology update, and traffic analysis/characterization (Agarwal, Kodialam, and Lakshman 2013). Those mechanisms reside at the application layer and it is possible to generate conflicting low-level switch rules among different applications. *ConVenus* takes a different approach by residing below the application layer to intercept the network flow updates from the SDN controller for congestion verification.

3 SYSTEM OVERVIEW

We develop a verification system, *ConVenus*, to preserve the congestion-free property of the network. *ConVenus* is a shim layer that resides between the SDN controller and the network layer as shown in Figure 1. *ConVenus* intercepts the updates issued by the SDN controller, dynamically updates the network model, efficiently computes the new flow rates assume the new update is installed in the network, and performs

congestion verification, i.e., whether each flow has the desired throughput and whether the aggregated flow rate at every network device exceeds the link bandwidth. Updates that pass the congestion verification are applied to the data plane, otherwise, *ConVenus* reports the congestion issues to the network operators with the set of affected flows and estimated new flow rates.

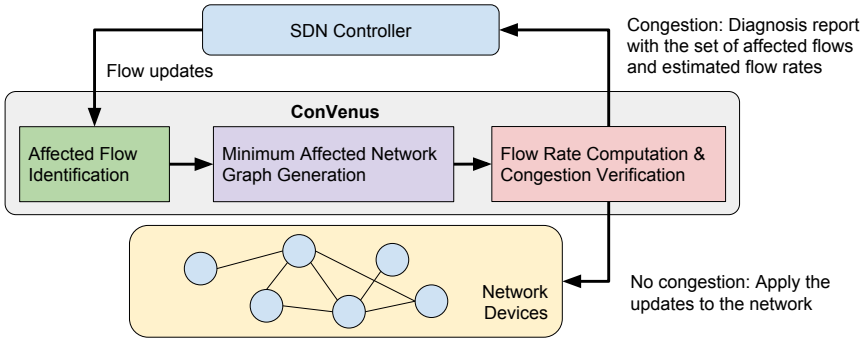


Figure 1: *ConVenus* sits between the SDN controller and network devices to intercept and verify every flow update to preserve the congestion-free property.

ConVenus models the network devices as a set of connected output ports. We model each flow as a directed path from an ingress switch to an egress switch. We assume the ingress rates of the flows are known before verifying the updates. The flow input rates can be derived from application specification or acquired from the statistics collected at of the OpenFlow switch flow entries specified in (ONF 2014). *ConVenus* consists of three key components to perform the congestion verification upon receiving an update from the SDN controller.

- Affected Flow Identification. We determine the smallest set of flows, whose rate will be potentially affected by the update.
- Minimum Affected Network Graph Generation. We create a network graph consisting of the affected flows and ports identified in the previous module. Congestion may only occur in this subnetwork. The size of the subnetwork is often significantly reduced compared with the entire network, and thus greatly improve the verification speed. This is particularly useful for online verification. Our algorithm to identify the minimum affected network graph works under the assumption the current network is congestion-free before applying the new updates.
- Flow Rate Computation and Congestion Verification. We develop a four-phase simulation algorithm to quickly compute the rate of each flow (including the rates at all the intermediate ports along the flow path) in the minimum affected network graph generated by the previous module. The detailed flow rates are then used to determine whether the update will cause congestion and if so, at which portion of the network and by how much.

In this paper, we focus on the congestion-free property, but the design of *ConVenus* intends to provide a generic framework for verifying other invariants. For example, it is straightforward to incorporate the reachability-based invariants described in VeriFlow (Khurshid, Zou, Zhou, Caesar, and Godfrey 2013) into *ConVenus*. We plan to explore other security policy and network invariants in the future work.

4 NETWORK FLOW SIMULATION AND MODELING FOR CONGESTION VERIFICATION

This section presents the network flow model in *ConVenus* and the simulation algorithm for fast flow rate computation of the entire network upon receiving a controller update. The estimated flow rates are then compared with the desired bandwidth requirements to determine whether congestion would occur if the update is applied to the network. We describe the modeling assumptions, the problem formulation, and the four-phase simulation algorithm for fast flow rate computation.

4.1 Modeling Assumptions

Scheduling Policy. When multiple flows are aggregated at a switch port, the scheduling policy determines the bandwidth allocation to each flow. In theory, all the existing scheduling policy in traditional switches can be realized in SDN switches too. In practice, First-Come-First-Serve (FCFS) scheduling policy is commonly used. In *ConVenus*, we assume that multiple flows aggregated in a port are scheduled according to the FCFS scheduling policy. Our model is designed to be easily extended to other scheduling policies by changing the bandwidth allocation rules. For instance, the rules for Fair Queuing scheduling policy were investigated in a prior work (Jin and Nicol 2010).

Buffering Strategy. In *ConVenus*, we assume every switch adopts the output buffering strategy, although other buffering strategies can be easily incorporated into our algorithms described in Section 4.3. According to the OpenFlow specification (ONF 2014), each output port is assigned with one or more output queues. The action of forwarding a packet is required to specify which port to send the packet to, but it is optional to specify the specific queue.

Ingress Flow Rate. In *ConVenus*, we assume the ingress rates of all the existing flows in the network are known. The rates can be derived from the application specification and/or be estimated from the flow statistics stored in the SDN switches. How to obtain the precise flow ingress rates is not a focus of this paper, and we leave the development of a flow rate monitor (e.g., continuous interception of OpenFlow statistic messages from switches to controller) to compute ingress flow rates as our future work.

4.2 Problem Formulation and Notations

All the notations used in remainder of this paper are summarized in Table 1.

Table 1: Notations.

Symbol	Explanation	Symbol	Explanation
Q	Set of all ports in the network	μ_q	Bandwidth of port q
$Q_{ingress}$	Set of ingress ports in the network	$\lambda_{f,q}^{in}$	Input rate of flow f at port q
Q_{egress}	Set of egress ports in the network	$\lambda_{f,q}^{out}$	Output rate of flow f at port q
Q_f	Ordered Set of ports in f's path	Λ_q^{in}	Aggregated input rate at port q
A_f	Affected flow set in respect to flow f	Λ_q^{out}	Aggregated output rate at port q
N_f	Minimum affected network in respect to flow f	R_f	Ordered Set of the rates of f at $q \in Q_f$
$\lambda_{f,q}$	Flow rate of f at port q, including both input and output rate		
$S(\lambda_{f,q})$	State of a flow f at a port q; $\{settled, bounded, unsettled\}$		

The data plane is modeled as a collection of switches connected by unidirectional links. The sending endpoint of a link is attached to a switch's output port. There is an output buffer associated with each output port. Essentially, we can model a network N as a set of output ports Q and a set of flows F. Each output port $q \in Q$ resides either on an end-host or a switch. Each flow $f \in F$ is represented as a tuple $< Q_f, R_f >$, where Q_f is an ordered set of ports $(q_1, q_2, ..., q_{|Q_f|})$, which is the path that the flow passes through, i.e., q_1 is the output port of the first switch in the flow that connects to the source, $q_{|Q_f|}$ is the output port of the last switch in the flow that connects to the sink, and the remaining ports are on the intermediate switches along the communication path. Note that any adjacent ports (q_i, q_j) in the sequence must be connected. R_f is an ordered set of input and output rates of f passing through the same sequence of ports, $q \in Q_f$.

We denote the bandwidth of an output port q as μ_q, and the input and output rate of a flow f that passes through q as $\lambda_{f,q}^{in}$ and $\lambda_{f,q}^{out}$. F_q denotes the set of flows passing through q. The aggregated input rate at port q is denoted by Λ_q^{in}, which equals to the summation of the input rates of all the flows that pass through q, i.e., $\Lambda_q^{in} = \sum_{f \in F_q} \lambda_{f,q}^{in}$. We have the following two definitions of congestion.

Definition 1 A port q is congested if and only if $\mu_q < \Lambda_q^{in}$.

Definition 2 A network is congested if and only if at least one port is congested.

Given the bandwidth of every port and the ingress rate of every flow, our first objective is to determine whether the network has congestion. Our second objective is to discover, for each flow, the input rate and the output rate at each port along its path (i.e., $\lambda_{f,q}^{in}$ and $\lambda_{f,q}^{out}$ for every f and q). Note that the output rate of a flow leaving the current port is equal to the input rate of the flow entering the next port, and the egress rate of the flow is equal to the flow output rate leaving the last switch along the path.

4.3 Simulation Algorithm of Flow Rate Computation

A key component of *ConVenus* is the module for quickly computing the flow rate changes of all the affected flows due to the new network update from the controller. The results are used to identify (1) whether the new update will cause network congestion, (2) the set of switch ports that the congestion occur, and (3) the input and output rate of the congested flows along the communication paths. We developed a four-phase simulation algorithm to achieve fast flow rate computation in *ConVenus* as motivated by several prior works (Nicol and Yan 2006, Jin and Nicol 2010). The four phases include: (1) flow rate update, (2) reduced dependency graph generation, (3) flow rate computation using fixed-point iteration, and (4) residue flow rate computation, as shown in Figure 3. To illustrate the algorithm, we first present the basic rules for the flow rate computation with FCFS scheduling policy, and then the circular dependence among the affected flows, and finally the step-by-step description of each phase in the simulation algorithm.

4.3.1 Flow Rate Computation under the FCFS Scheduling Policy

We define the following rules for calculating the output rates of all the flows aggregated at a particular switch port, given the corresponding input rates and the bandwidth information.

$$\lambda_{f,q}^{out} = \begin{cases} \lambda_{f,q}^{in}, & \text{if } \Lambda_q^{in} \leq \mu_q \\ \lambda_{f,q}^{in} \times \frac{\mu_q}{\Lambda_q^{in}}, & \text{otherwise} \end{cases} \tag{1}$$

If the aggregated rate is less than or equal to the port's bandwidth, every flow's output rate is the same as the input rate; If the aggregated rate is greater than the bandwidth (i.e., the port is congested), the flow output rate is proportional to its arrival rate under the FCFS scheduling policy.

4.3.2 Circular Dependence Among Affected Flows

The objective of the simulation algorithm is to find the input and output rates of all flows at all the ports along the path. The basic idea is to propagate and update the flow rate values along the path (i.e., a sequence of ports) for every flow based on the Equation 1. We are necessarily left with circular dependences among some flow variables. Let us illustrate the circular dependence with a simple example shown in Figure 2. Assume both q_1 and q_2 are congested, we have the following equations for flow f_1 and f_2:

$$\begin{aligned} \lambda_{f1,q1}^{out} &= \lambda_{f1,q1}^{in} \times \frac{\lambda_{f1,q1}^{in}}{\lambda_{f1,q1}^{in} + \lambda_{f2,q1}^{in}} \\ \lambda_{f2,q2}^{out} &= \lambda_{f2,q2}^{in} \times \frac{\lambda_{f2,q2}^{in}}{\lambda_{f1,q2}^{in} + \lambda_{f2,q2}^{in}} \end{aligned} \tag{2}$$

Since $\lambda_{f2,q1}^{in} = \lambda_{f2,q2}^{out}$ and $\lambda_{f1,q2}^{in} = \lambda_{f1,q1}^{out}$, we find that $\lambda_{f1,q1}^{out}$ and $\lambda_{f2,q2}^{out}$ essentially depend on each other. Such circular dependence relationship can be extended to multiple flows with multiple ports involved. We address this issue by identifying all the flow variables that are involved in each circular dependence, constructing a dependency graph, and applying fixed-point iteration to solve the equations to derive the output flow rates, and the details are presented in the next section.

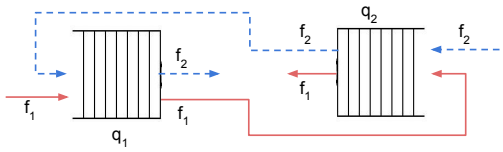


Figure 2: An example of circular dependence between two flows.

4.3.3 Simulation Algorithm for Network Flow Rate Computation

Given a set of flow input rates, the objective of the algorithm is to efficiently compute the output flow rates at the destination hosts as well as the traffic loads at all the intermediate switches along the flow paths, in order to detect the network congestion. Each flow is in one of the three state: settled, bounded or unsettled. A settled flow has a finalized flow rate; a bounded flow has a known upper bound on its flow rate; an unsettled flow is neither bounded or settled. Figure 3 illustrates the procedures of the flow rate computation algorithm, which consists of four phases.

- Phase-I propagates the flow rate and state from ingress points throughout the network. The goal is to settle flows and resolve as many ports as possible. We calculate the flow rate and state of all the output flows of a port based on Equation 1 under the FCFS scheduling policy, and then pass the rate and state of the output flow to the next switch's input along the flow path. In the case of no circular dependence among the involved ports and flows, all the flow rates are settled, and the output results are used to check and determine the flow-level congestion in the network. In the case of circular dependencies among some flow variables, Phase-I will assign an upper bound of the rate to those flows. If the input flow is settled, the upper bound is derived by ignoring all bounded input flows. If the input flow is bounded, the upper bound is derived by ignoring all other bounded input flows and treat it as settled with the rate set to the bounded rate. The remaining three phases mainly focus on addressing the circular dependencies to compute the flow rates.
- Phase-II identifies all the flow variables whose values are circularly dependent, and constructs one or more (directed) dependency graphs. The vertex set is composed of all output ports containing unsettled and bounded flows. The directed edges are the flows involved in the circular dependence.
- Phase-III formulates a set of non-linear equations for the flows in each dependency graph, as illustrated in Equation 2. We use the fixed-point iteration method to solve those equations. Note that the initial values are the bounded values calculated in Phase-I.
- Phase-IV substitutes the solutions into the system and continues to compute and update the rates of the remaining unsettled flows that are affected by those flows in the circular dependency graph(s).

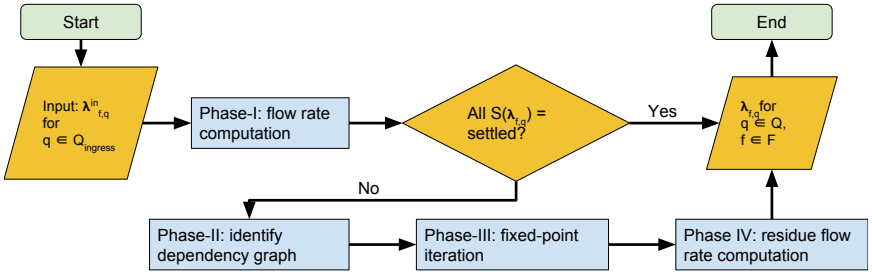


Figure 3: Flow Chart of the Four-phase Simulation Algorithm for Network Flow Rate Computation.

5 SPEEDING UP CONGESTION VERIFICATION

It is critical to perform congestion verification at high speed, because delaying the updates can damage the network state consistency and harm the real-time application requirements such as fast failover. To speed up the process, we investigate ways to reduce the problem space by identifying the minimum set of flows

and ports that are affected by the flow update, and performing the verification describe in Section 4 only on the network model consisting of those elements. This approach also reduces the possibility to have circular dependence, which further increases the speed by skipping phase II to IV in the simulation algorithm. Based on the assumption that the existing network is congestion-free, we derive a set of theorems to generate the minimum affected network for different types of flow update, including flow removal, flow addition, and flow modification. An ongoing work is to develop an efficient graph search algorithm to speed up the verification in other scenarios that the congestion-free network assumption does not hold, e.g., one can tolerate short-term network congestion in order to achieve quick update installation.

5.1 Flow Removal

We claim that removing a flow from a congestion-free network neither causes any congestion nor changes the flow rate of any other flows in the network.

Theorem 1 Given a congestion-free port, removing an input flow (or reducing its rate) neither makes the port congested nor changes the output rates of other flows sharing the same port.

Proof. Equation 1 indicates $\lambda_{f,q}^{out} = \lambda_{f,q}^{in}$ for every flow f at a congestion-free port q (i.e., $\Lambda_q^{in} \leq \mu_q$). A flow removal or a flow rate reduction decreases Λ_q^{in}, and thus cannot cause congestion, and the rate of every other flow remains unchanged as $\lambda_{f,q}^{in}$. □

A congestion-free network contains no congested ports according to Definition 2. Therefore, it is safe to forward any flow removal updates to the data plane and doing that will not change the rates of any existing flows in the network.

5.2 Flow Insertion

We first introduce the concepts of **affected flow set** and **minimum affected network**, and then describe the algorithm to construct them, and finally present a set of theorems to prove the correctness of the algorithm.

Definition 3 Given a newly inserted flow f^*, the affected flow set A_{f^*} is the set of all the flows (including f^*) in the network whose rates may be changed due to the insertion.

Definition 4 Given a newly inserted flow f^*, the minimum affected network N_{f^*} is the network consisting of all the possible congested ports and the affected flow set due to the insertion.

Algorithm 1 illustrates how *ConVenus* generates A_{f^*} and N_{f^*}. Figure 4 presents an example of the input and output used in Algorithm 1.

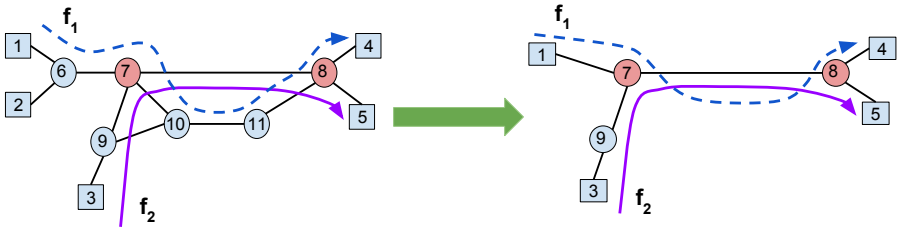


Figure 4: A simple example for Algorithm 1: Consider a network N with one existing flow $f_1 =< (1,6,7,10,11,8,4), R_{f_1} >$ and one new flow $f_2 =< (3,9,7,8,5), R_{f_2} >$. The minimum affected network N_{f_2} contains f_2 and the modified $f_1 =< (1,7,8,4), R_{f_1} >$, and the port set {3, 9, 7, 8, 5, 1, 4}.

Theorem 2 Inserting a flow into a port q makes the output rates of all the existing flows passing through q either decrease or remain the same.

Algorithm 1: Generation of the affected flow set and the minimum affected network

Input : A network N, and a flow f^* to be inserted
Output: Affected flow set A_{f^*} and minimum affected network N_{f^*}

Add f^* into A_{f^*}
for *each flow* f *in* N **do**
  if $Q_f \cap Q_{f^*} \neq \varnothing$ **then**
    add f into A_{f^*}
  end
end
Add all ports $q \in Q_{f^*}$ into N_{f^*}'s port set
for *each flow* f *in* A_{f^*} **do**
  for *each port* $q \in Q_f$ **do**
    if $q \notin Q_{f^*}$ **then**
      /* include the ingress and egress ports of an affected flow */
      if $q = q_1$ *or* $q = q_{|Q_f|}$ **then**
        add q into N_{f^*}'s port set
      else
        /* Removing the unaffected ports in flow f */
        remove q from Q_f
        remove $\lambda_{f,q}$ from R_f
      end
    end
  end
end
Add all flow $f \in A_{f^*}$ into N_{f^*}'s flow set
return A_{f^*}, N_{f^*}

Proof. There are three possible situations based on equation (1). Note that a new flow insertion to q increases Λ_q^{in}.

- If q is congestion-free before and after the flow insertion, then all the existing flows' output rates remain unchanged, i.e., $\lambda_{f,q}^{out} = \lambda_{f,q}^{in}$.
- If q is congestion-free before the flow insertion and congested after the flow insertion, then all the existing flows' output rates decrease, because $\lambda_{f,q}^{in} \times \frac{\mu_q}{\Lambda_q^{in}} < \lambda_{f,q}^{in}$.
- If q is congested before and after the flow insertion, then all the existing flows' output rates decrease, because Λ_q^{in} is increased.

□

Theorem 3 When a flow f^* is inserted into a congestion-free network, the possibly congested ports are $q \in Q_{f^*}$.

Proof. We prove by contradiction. We assume that, after inserting f^*, there exists a congested port $\hat{q} \notin Q_f$, i.e., $\mu_{\hat{q}} < \Lambda_{\hat{q}}^{in}$. Therefore, at least one flow $\hat{f}$ ($\hat{f} \neq f^*$ because of the definition of Q_{f^*}) passing through $\hat{q}$ increases the input rate. If $\hat{f}$ does not share any ports with f^*, or $\hat{f}$ passes through $\hat{q}$ before sharing any ports with f^*, then $\hat{f}$'s rate remains the same. If $\hat{f}$ and f^* share ports before passing through $\hat{q}$, according to Theorem 3, $\hat{f}$'s rate remains the unchanged or decreases. Either way, we have a contradiction. □

Theorem 3 is a key step to prove the correctness of Algorithm 1. Theorem 3 shows that the generated N_{f^*} contains all the congested ports, since we add all $q \in Q_{f^*}$. Equation 1 indicates that passing through a congestion-free port does not change the flow rate. This justifies our claim that A_{f^*} contains the exact set of the affected flows and those flows can only change the rates at the ports in N_{f^*}.

5.3 Flow Modification

For a flow modification update, i.e., changing route of an existing flow in the network, *ConVenus* simply treats the update as a set of flow removal and insertion updates (typically, a removal operation followed by an insertion operation). Since we do not consider the transient network congestion during the updates, the two operations are identical.

6 EVALUATION

6.1 Experiment Setup

To evaluate the performance of *ConVenus*, we design network scenarios based on the *campus network* model, which is a key baseline network model originally designed for benchmarking parallel network simulation (Nicol 2009). The entire topology is an abstraction of a ring of simplified campus networks as shown in the left portion of Figure 5. Each simplified campus network consists of a ring of access switches, each of which has a number of hosts directly connected to it, as shown in the right portion of Figure 5. Communication across different campus networks must pass through the ring of the exchange switches connected by their own gateways.

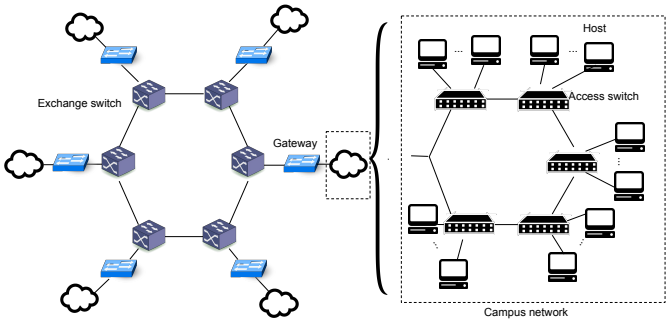


Figure 5: An example of campus network topology.

In this work, we constructed a network topology consisting of a ring of eight campus networks, which requires eight gateway and eight exchange switches. Within a campus network, every access switch connects to eight hosts. Each switch is modeled as one port in *ConVenus*. Therefore, the network has 80 ports and 512 hosts in total. We set the link bandwidth to be 10 Mbps between the hosts and the access switches, 100 Mbps between the access switches themselves as well as between the access switches and the gateways, and 1 Gbps between the gateways and exchanges switches as well as between the exchange switches themselves. To generate a flow insertion update, the SDN controller randomly selected two different hosts in the network, one as the source and the other as the destination of the flow. The flow update contains state variables including a unique identifier, an ingress rate of 10 Mbps, and a shortest path between the two hosts. To generate a flow deletion, the controller randomly selected an existing flow in the network using the flow identifier. While the random flow selection is a reasonable assumption, we plan to deploy *ConVenus* on a physical SDN network in order to perform high fidelity evaluation. Each experiment consisted of two stages. During stage 1, initially there was no flow in the network, and we issued random flow insertion updates (one update at a time) until 250 flows were successfully inserted in the network. We configured *ConVenus* not to apply the flow update to the network if the update did not pass the congestion verification. During stage 2, with 250 flows in the network, we randomly generated 400 flow updates (50% are flow insertions and 50% are flow deletions), and passed them to *ConVenus*. We repeated 10 times for each set of experiments, and the results are discussed in the next section.

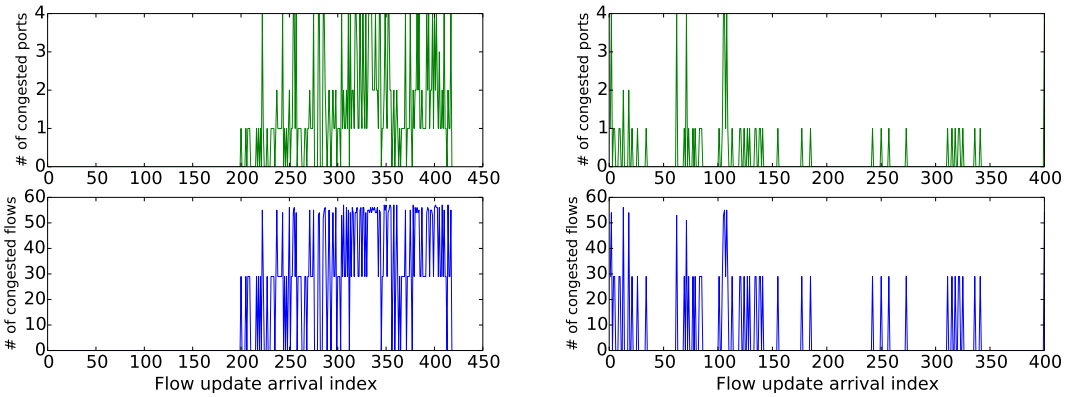
6.2 Experimental Results and Analysis

We first count the number of times that ConVenus reported congestion during the verification. Among those updates causing congestion, we also count the number of times that circular dependence occurred. The results are recorded in Table 2. We observe that in stage 1, around 174 flow insertions that would lead to network congestion were detected before we successfully inserted 250 congestion-free flows into the network. Among those congestion cases, more than one-third of them resulted in circular dependence, which required further processing using Phase-II through Phase-IV of our simulation algorithm described in Section 4.3. In stage 2, there were much less congestion cases (41 on average) and circular dependence

cases (3.5 on average). This is because the flow deletion updates did not cause any ports in the network to be congested as described in Section 5.1. We further plot the number of congested flows and ports for every flow updates in one trial for stage 1 and stage 2 in Figure 6a and 6b. In worst-case scenario, 4 ports are congested in total. The number is small because congestion can only occur on those ports along the path of the newly inserted flow.

Table 2: Congestion verification: # of flow updates resulting in network congestion and circular dependence. N_c is the # of network congestion occurrence and N_d is the # of circular dependence occurrence.

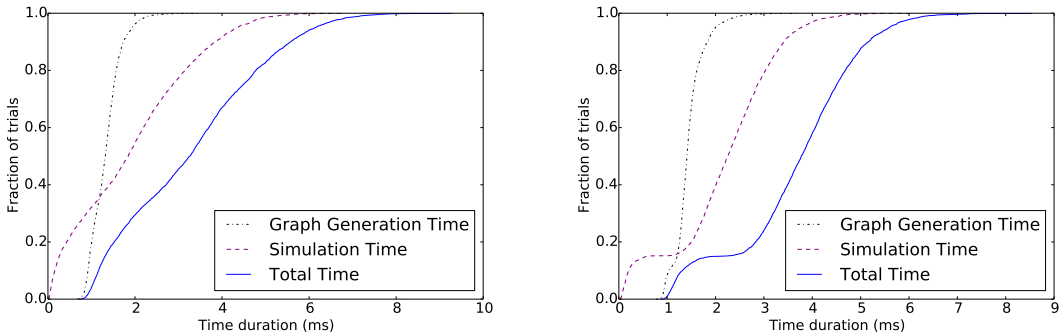
	Stage 1		Stage 2	
	N_c	N_d	N_c	N_d
Average	174.8	67.5	41.0	3.5
Standard Deviation	52.8	23.8	7.9	2.2



(a) Stage 1: Flow insertion updates. (b) Stage 2: Flow insertion and deletion updates.

Figure 6: Number of congested flows and ports for each flow update.

We next evaluate the verification speed of *ConVenus*. We record the execution time to perform congestion verification of each flow update for all the 10 experiments. We also record the total time and break down into the time for (1) generating the minimum affected network graph and (2) executing the simulation algorithm for flow computation and congestion verification. Figure 7a and Figure 7b plot the cumulative distribution functions (CDFs) of the verification time in stage 1 and stage 2.



(a) Stage 1: Flow insertion updates. (b) Stage 2: Flow insertion and deletion updates.

Figure 7: Cumulative distribution function of the update verification time, which consists of: (1) time to generate the graph to model the minimum affected network and (2) time to run the 4-phase simulation algorithm.

The verification speed is high. Around 80% of the verification takes less than 5 ms in stage 1, and around 90% of the verification takes less than 5 ms in stage 2. We did not observe the long-tail behavior in the CDFs, and the verification time is bounded by 10 ms (the maximum time is 9.3 ms in stage 1 and 8.5 ms in stage 2). The evaluation results indicate that *ConVenus* is a suitable online verification tool for many network scenarios within such a delay bound. We also observe that 95.6% of the minimum affected network graph generation time is less than 2 ms in both stages, and most time was spent on executing the simulation algorithm. Therefore, we further break down the time spent in each of the four phases in the simulation. We observe that in both stages, the phase-I (i.e., flow rate computation) takes the majority of time (92.7% in stage 1 and 96.9% in stage 2). It is because (1) if no congestion is detected, the simulation stops at phase-I, (2) even if congestion occurs, but no circular dependence is generated, simulation does not have to run through phase II to IV; and (3) even if a circular dependence is generated, the graph size is bounded by the number of congested ports, which is small as shown in Figure 6.

7 CONCLUSION AND FUTURE WORK

We present *ConVenus*, a dynamic verification system to preserve the congestion-free property before applying the flow updates to an SDN-based network. We develop a dynamic data-driven network model and a simulation algorithm to perform the congestion verification. We also develop an optimization algorithm to reduce the problem size in order to achieve high-speed online verification. Our future work includes acquiring dynamic network flow rates at run time from the application layer and inject them into *ConVenus* to steer the verification process. We will also generalize *ConVenus* as a platform for verifying other network invariants and security policies. In addition, we will investigate consistency-enforcement algorithms to handle transient network faults caused by the network temporal uncertainty.

ACKNOWLEDGMENTS

This work is partly sponsored by the Maryland Procurement Office under Contract No. H98230-14-C-0141, the Air Force Office of Scientific Research (AFOSR) under grant FA9550-15-1-0190, and a cooperative agreement between IIT and National Security Research Institute (NSRI) of Korea. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Maryland Procurement Office, AFOSR and NSRI.

REFERENCES

Agarwal, S., M. Kodialam, and T. V. Lakshman. 2013, April. "Traffic Engineering in Software Defined Networks". In *INFOCOM, 2013 Proceedings IEEE*, 2211–2219.

Cadar, C., D. Dunbar, and D. Engler. 2008. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.". In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Volume 8, 209–224.

Darema, F. 2004. "Dynamic Data Driven Applications Systems: A New Paradigm for Application Simulations and Measurements". In *Computational Science-ICCS*, 662–669. Springer.

Jin, D., and D. Nicol. 2010. "Fast simulation of background traffic through Fair Queueing networks". In *Proceedings of the 2010 Winter Simulation Conference*, 2935–2946. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Jin, X., H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. 2014. "Dynamic Scheduling of Network Updates". In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 539–550.

Kazemian, P., M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. 2013. "Real Time Network Policy Checking Using Header Space Analysis". In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 99–112. Berkeley, CA, USA.

Kazemian, P., G. Varghese, and N. McKeown. 2012. "Header Space Analysis: Static Checking for Networks". In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 113–126. San Jose, CA.

Khurshid, A., X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. 2013. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 15–27. Lombard, IL.

Liu, H. H., X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. 2013. "zUpdate: Updating Data Center Networks with Zero Loss". In *Proceedings of the 2013 ACM Conference on SIGCOMM*, 411–422.

Mai, H., A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. 2011. "Debugging the Data Plane with Anteater". In *Proceedings of the 2011 ACM Conference on SIGCOMM*, 290–301.

David Nicol 2009. "Standard Baseline DARPA NMS Challenge Topology". http://www.ssfnet.org/Exchange/gallery/baseline/index.html.

Nicol, D. M., and G. Yan. 2006, January. "High-Performance Simulation of Low-Resolution Network Flows". *Simulation* 82 (1): 21–42.

ONF Accessed 2014. "OpenFlow Specification". https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

ONF Accessed 2016. "Open Networking Foundation". https://www.opennetworking.org.

Zhou, W., D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. 2015. "Enforcing Customizable Consistency Properties in Software-Defined Networks". In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 73–85.

AUTHOR BIOGRAPHIES

XIN LIU is a Ph.D. candidate in the Department of Computer Science at the Illinois Institute of Technology. His interests lie in network simulation and network verification in software-defined networks. His email address is xliu125@hawk.iit.edu.

DONG (KEVIN) JIN is an Assistant Professor in the Department of Computer Science at the Illinois Institute of Technology. He holds a Ph.D. degree in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign. His research interests lie in the areas of trustworthy cyber-physical critical infrastructures, cyber-security, simulation modeling and analysis, and software-defined networking. His email address is dong.jin@iit.edu.

JONG CHEOL MOON is a Senior Member of Engineering Staff in National Security Research Institute of Korea. He holds a master degree in Electronics with Information Security specialization in Kyungpook National University, Korea. His research interests lie in the areas of cyber security. His email address is jcmoon@nsr.re.kr.

CHEOL WON LEE is a Principal Member of Engineering Staff in National Security Research Institute of Korea. He holds a Ph.D. degree in Computer Engineering with Cyber Security specialization in Ajou University, Korea. His research interests lie in the areas of smart grid security, critical infrastructure protection, and cyber-physical security. His email address is cheolee@nsr.re.kr.