CCU: Algorithm for Concurrent Consistent Updates for a Software Defined Network

Radhika Sukapuram, Gautam Barua
Dept. of Computer Science and Engineering
Indian Institute of Technology Guwahati
Assam, India 781039
Email: {r.sukapuram,gb}@iitg.ernet.in

***Abstract*—A Software Defined Network has its control plane separated from its data plane, with the control plane providing abstractions for development of control programs that alter the state of the network by updating the rules installed in switches. Switch updates are central to an SDN and updates must be such that there are no packet drops or loops. An update property which ensures that the traces generated for a packet are either due to the old network configuration or due to the new configuration, but never a combination of both, thus preventing packet drops and loops, is called per-packet consistency. We envisage that large data center networks supporting multiple tenants and network virtualization will need a large number of concurrent updates due to VM creation, VM migration, network management applications etc. There are algorithms in the literature that enable concurrent updates and improve update speed but they are either not general or do not preserve per-packet consistency. This paper proposes an update algorithm, Concurrent Consistent Updates (CCU), that is general, is per-packet consistent and that enables concurrent disjoint updates. Since the size of a TCAM on a switch is small, rule tables implemented in software are used to supplement the TCAM. The algorithm makes use of a Software Rules Table to restrict updates to only the internal switches that genuinely need an update and to reduce update installation time.**

I. Introduction

Massive data centres, virtualization and cloud computing make networks very complex. In existing networks, switches need to be often individually and manually configured, whether to change Access Control Lists (ACL) or to add or remove nodes from the network, or for traffic engineering (TE), as there is no programmatic interface available to dynamically and centrally control switch behaviour. Network hardware is complex and hard to manage and therefore creates lock-in with a specific vendor. Both the interfaces to switches and the software are closed and proprietary. Innovations in networking are harder because changes to networking equipment take time. Networking equipment is varied, with middle-boxes such as intrusion detectors, firewalls, server load balancers and network address translators, in addition to routers and switches, adding to the complexity [1].

A network is composed of a control plane and a data plane: the data plane examines packet headers and takes a forwarding decision by matching a forwarding table, while the control plane builds the forwarding table. A Software Defined Network (SDN) has its control plane separated from the data plane [2]. It also provides abstractions to the control plane so that the control plane is programmable. All the switches are logically connected to a controller and the controller programs the switches by updating the rules in their rule tables to implement functions such as routing, isolation, TE etc. Dynamically updating switches is fundamental to SDN applications. The protocol between the controller and the switches and the specifications of the switches themselves are defined by the Open Networking Foundation and is called Openflow [3]. All the complexity of managing the network is within the controller, while operators can write control programs above the controller to implement their functions.

An Openflow switch consists of one or more flow tables, a group table, and a channel that interfaces with the Controller. A flow table consists of a set of flow entries or rules. Each flow entry has match fields, priority, counters, timers, and instructions that need to be applied to matching packets. Flow entries are installed in order of priority. Packets are matched with the match fields and the instructions associated with the first matched entry are executed. If there is no matching entry, the instructions in the table-miss flow entry are executed - this entry has wildcards for all match fields and has the lowest priority. The instruction may be an action such as forwarding a packet to a port or modifying a packet header, or a modification of the pipeline. Using the Openflow protocol, the Controller can add, delete or modify flow entries. The timer associated with a flow entry is used to specify the maximum amount of time before it is deleted by the switch. The counter is updated when packets are matched with a flow entry. Depending on the rules installed, an Openflow switch can behave as a router, a switch or a middle-box. SDN has gained success in the industry with usage of SDN in their WAN by Google [4], for virtualisation in their product NSX, by VMWare [5] and for their data centre networks, by Microsoft [6], in addition to commercial Openflow switches from various vendors and multiple Controller platforms [1].

During updates, it is desirable that several properties are preserved, one of which is per-packet consistency (PPC), where the traces generated by a packet entering the network must either be due to the network configuration before the update is installed or due to the network configuration after the update is installed, and never a combination of both [7]. This prevents packets from looping or getting dropped during an update, which is significant, as the update speed is far lesser than the packet switching speed.

Rules are typically stored in a TCAM, which is a scarce resource. To supplement a TCAM, a rule table can be implemented in software, in conjunction with a TCAM [8]. Enhanced Two-Phase Update with a Software Rules Table

(E2PU-SRT) [9], specifies a mechanism to address failures and to delete the old rules, for a per-packet consistent update. It uses a Software Rules Table (SRT) to speed up updates. While earlier work that preserves PPC [7] requires rules in every switch to be modified for every update, [9] mentions how to avoid that.

Suppose the existing rules in a network are labelled v_0. The algorithm to install a new set of updates of version 1, v_1, as per E2PU-SRT, is as follows: 1) The controller sends "Commit" with v_1 rules to all the affected switches. The internal switches install these rules in the SRT, while the ingresses just store the rules internally. 2) Each switch that processes "Commit" responds with "Ready to Commit". 3) After receiving "Ready to Commit" from the expected switches, the controller sends "Commit OK" to the ingresses and the internal switches. The ingresses install v_1 rules and simultaneously start tagging packets with v_1. 4) All the switches that process "Commit OK" respond with "Ack Commit OK" with their current time, T_o. 5) After receiving "Ack Commit OK" from all the ingresses, the controller denotes the latest T_o as T_l and sends "Discard Old" to the switches that have old rules, to delete the old rules. 6) The internal switches that receive "Discard Old" delete the old rules whenever their current time T_c exceeds $T_l + M$, where M is the maximum life-time of a packet in the network, while ingresses delete the old rules immediately. 7) After deleting the old rules, each switch sends a "Discard Old Ack" to the controller.

The objective of this paper is to describe a general, concurrent, per-packet consistent, update algorithm.

II. Motivation for Concurrent PPC updates

A. Motivation for per-packet consistent updates

E2PU-SRT has two interesting properties: 1) it does not make assumptions about the toplogical properties of the update or the nature of flows and therefore works for all situations 2) it preserves PPC.

A concurrent update is one in which updates from more than one SDN application or multiple updates from the same application can be installed simultaneously. An Equivalence Class (EC) of packets is the set of packets that use the same set of forwarding actions from the ingress to the egress. If one or more updates is disjoint, the ECs of packets affected by any two of those updates is disjoint.

E2PU-SRT, as it exists, does not allow concurrent disjoint updates. This is because every update affects every rule in every switch: thus effectively, there are no disjoint updates (even though, in reality, the updates may be disjoint).

If all the paths affected by an update are known in advance, using the mechanisms identified in [10] or [11], then concurrent updates are possible preserving PPC, using E2PU-SRT. Suppose for an update, an internal switch requires this rule to be inserted: "*tcp_port=80, forward2*". This rule affects a very large number of paths (ECs) in a network. Identifying the number of paths that affect a large number of ECs is not possible in a manner that is fast enough for real implementations [10]. Only for the scenarios where the paths can be identified, disjoint updates can be installed in parallel.

Preserving PPC is important because it will prevent packet drops and loops during updates.

B. Motivation for concurrency

In a large data centre that supports network virtualization and multiple tenants, the following will generate network updates:

1) Each tenant will require multiple VMs to be created [12], all belonging to a virtual network of a certain topology. VMs will need to be assigned to servers depending on what the VM allocator wishes to optimize. To isolate one tenant from another, for rate limiting etc., the controller will need to update the virtual switches on servers. The physical switches will need to be updated to implement the virtual network to physical network mapping [13], [14].
2) Orthogonal to this, there will be an "SDN applications store" consisting of applications such as MicroTE [15], which proposes updating the network every few seconds for TE, and Hedera [16], which proposes updates when the load changes, for better flow scheduling. The frequency of updates will also depend on the number of apps that can be meaningfully run simultaneously. A list of apps is in [17].
3) VM migration may involve migrating the entire virtual network, further generating updates [18], which are more complex than the above categories, as it involves cloning switches and allowing other updates while the migration is in progress. VM migration is stated as a solution for TE, energy savings, disaster management, cloud bursting etc. and will need to be frequently done.

There are algorithms in the literature that enable concurrent updates and improve update speed but they are either not general or do not preserve PPC. This is further discussed in section VI.

III. Concurrency Requirements

We envisage a network model where applications from either the same controller or different controllers issue updates. Each update gets a unique tag from the controller (or a central entity in the case of multiple controllers). After the controller receives "Ready to Commit" from all the switches that process "Commit" the update is said to be *stage1-complete*. After the controller receives "Ack Commit OK" from all the switches that process "Ack Commit", the update is said to be *stage2-complete*. After the controller receives all the "Discard Old Ack" messages, the update is *complete*.

The controller sends a "Commit" to the switches as soon as a tag is allocated. Let there be several "Commit" messages sent, belonging to different versions. If an update $n + 1$ is stage1-complete before an update n, it must be possible to proceed with the rest of the update for $n + 1$, without waiting for n to be stage1-complete. Ideally, this must be possible in all situations. Practically, this depends on the level of concurrency. This is further explained in section V.

An update $u2$ that intersects with an update $u1$ may begin as soon as $u1$ is complete and not earlier. An application will issue $u2$ only after $u1$ is complete.

IV. Our Contribution

The algorithm stated below accomplishes the following: C1) it is PPC C2) it allows concurrent disjoint updates C3) it makes no assumptions on the sequence of updates or the nature of rules and is therefore general C4) it minimises the number of internal switches to be updated by restricting the update to only those internal switches that require a genuine rule change. C5) it provides a trade off between concurrency and packet header overhead.

V. CCU: Concurrent Consistent Updates

A. Rule installation in internal switches

We assume Openflow switches to make the descriptions easy. An internal switch has a rule table implemented in an SRT, "in series" with a rule table implemented in a TCAM. A packet, on entering an internal switch, is first matched with the rules in the SRT. If there is no match, it is forwarded to the TCAM.

All rules are first inserted into the SRT, at a higher priority than the previous version of the rules. The new rules check packets for a specific version number. Whenever the previous version of rules, if any, is deleted, the version field of the current version of rules is changed to don't cares so that they cease checking packets for a version number. If there is no previous version to be deleted, then the new rule need not check for a version number, to begin with. The old rules will typically be in the TCAM, though they can be in the SRT too, but always at a lower priority than the new rules.

B. Version tagging of packets

Each packet has a version tag, followed by 4 bits, called the status bits. The status bit denotes the status of the update, with the most significant bit denoting the status of the update which is 1 less than the value of the version tag, the next significant bit 2 less than the value of the version tag and so on. Each status bit is set to 1 if the update corresponding to that version is stage1-complete. This means that rules belonging to that version are operational and that packets, if any, are getting switched according to those rules. If the version tag of a packet is n, all updates with versions less than or equal to $n-4$ are in the state "Commit OK sent", or later. For example, $10:0110$ indicates that update versions 10, 8 and 7 are stage1-complete, and 5 and below are in the state "Commit OK sent", or later. Update versions 9 and 6 are not stage1-complete. The updates for which the status bit is set are called *companion updates*. The number of status bits may be increased to improve concurrency or decreased to reduce the overhead.

C. Algorithm Description

Assume that a set of rules with version number n need to be installed. The switches where the rules need to be inserted are called the affected switches. $current$ denotes the version tag currently sent in packets and $status$ is the half-byte that denotes the status bits that the packets are tagged with.

The controller queues update requests from applications in app_queue. The controller also maintains a table, $update\_table$, which has *the state of the update*, for each update. The update states are **stage1-complete**, **"Commit OK" sent**, **stage2-complete** and **"Discard Old" sent**. The updates that are queued are disjoint.

The message exchanges from E2PU-SRT are reproduced below and alterations made where required for concurrent updates.

1) The controller checks if there are messages from the network, associated with an ongoing update. If there are messages, it goes to the appropriate step below, depending on the state of the update. If not, it checks whether there are any update requests in app_queue. If there are any requests, it goes to step 2.
2) The controller retrieves the first update request from app_queue, gets the next available tag for this update, say n, and sends "Commit" with the new rules to all the affected switches. All the affected internal switches (this could include the ingress switches that receive rules by virtue of them acting as internal switches for the other paths that belong to this update) install the new rules into the SRT. The new rules are such that they have higher priority than any version lesser than n. The ingress switches do not yet install either the rules that tag packets with n or the policy rules of version n, but store them internally.
3) Each switch that processes "Commit" sends back "Ready to Commit".
4) After the controller receives "Ready to Commit" from all the switches, it marks n as stage1-complete in $update\_table$. It calls the procedure resume_update() to resume updates, which is described in detail in section V-D. The procedure sends "Commit OK", if required, to *all the ingress switches*. The controller also sends the value of the status bits to be sent, along with the correct tag, in "Commit OK". As soon as the ingress switches receive "Commit OK" they stop sending packets tagged with the previous version and switch over to the new version. They also modify the status bits in the packet header. Let us assume that a "Commit OK" with n as the version is sent at this stage. (It is possible that due to previous updates not being stage1-complete, the update n is stalled, in which case, "Commit OK" is not sent for version n).
5) Each ingress that processes "Commit OK" sends "Ack Commit OK" to the controller. Each ingress sends the current time with this message, called T_o.
6) After receiving "Ack Commit OK" from all the ingresses, the controller notes the latest value of T_o received and saves it as T_l. It marks n and its companion updates that are not yet marked stage2-complete, as stage2-complete, in $update\_table$. Now it sends "Discard Old" to *all* the switches where rules were either inserted or deleted or both, for update n and its companion updates, unless already sent, as indicated in $update\_table$. It sends T_l and the rules to be deleted (old version) as a part of "Discard Old".
7) After each internal switch receives "Discard Old", when the current time of the switch $T_c > T_l + M$, where M is the maximum lifetime of a packet within the network, it does the following: a) it deletes the list of old rules received in "Discard Old" 2) it sets the version number field of the version n and its

Fig. 1. Algorithm to resume updates

1: **procedure** RESUME_UPDATE() ▷ Resumes the next set of pending updates in `update_table` by sending "Commit OK".
2: $size$ = Get number of consecutive updates from $current - 4$, in increasing order, whose bits are set to 1, by checking $status$
3: **if** $size \neq 0$ **then** ▷ The new window position is $current + size$
4: $temp$ = Position where the first update less than or equal to $current + size$ is stage1-complete
5: **if** $temp > current$ **then**
6: $current = temp$
7: **end if** ▷ Otherwise no change to $current$
8: **end if**
9: Update $status$ by reading $update\_table$
10: **if** "Commit OK" not already sent for at least one of $current$ or companion updates **then**
11: Send "Commit OK" with $current$ as the version and $status$ as the status bits
12: State of $current$ and companion updates for which the state is "stage1-complete" in $update\_table$ = *"Commit OK" sent*
13: **end if** ▷ If "Commit OK" sent, do nothing
14: **end procedure**

companion updates to don't cares. These rules may be moved to the TCAM any time from now. The ingresses delete the old rules as soon as they receive "Discard Old".

8) Each switch that processes "Discard Old" sends a "Discard Old Ack". When the controller receives all the "Discard Old Ack" messages the update is complete. The controller deletes the entries belonging to the completed updates from $update\_table$. The procedure continues from step1.

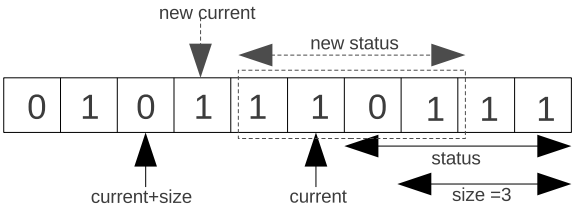
D. Resuming an update

Fig.1 shows the algorithm to send "Commit OK" for updates that are stage1-complete and to move the status window, if it is appropriate to do so.

The procedure first checks whether the status window can be moved. For that, it determines the number of consecutive updates in $status$, starting from the lowest version number, that are stage1-complete, to know which updates can be removed from $status$ (line 2). Let the number be $size$. The status window cannot be moved $size$ bits because the version $current + size$ may not be stage1-complete, as illustrated in Fig.2. The updates marked "0" in the figure are not stage1-complete and the the updates marked "1" are stage1-complete. So the algorithm finds the first version less than or equal to $current + size$ that is stage1-complete (line 4). Now it sets $current$ to this value, as long as this is greater than $current$. If there are no new updates that are stage1-complete and $size$ is non-zero, we just set the correct $status$ bits to 1, with $current$ remaining the same.

If there are no consecutive updates that are stage1-complete, $size$ is 0 and the window cannot be moved at all, but it is possible that some updates currently within the window

Fig. 2. Adjusting the status window



are stage1-complete and "Commit OK" needs to be sent for those. It sets bits in $status$ by determining the status from $update\_table$. If, for at least one of the updates in $current$ or its companion updates, a "Commit OK" has not been sent, it sends the "Commit OK" (line 10).

E. Behaviour at the data plane

When an internal switch receives a packet with version number n, the internal switch attempts to match the packet with a *valid* rule present in the SRT that has a version number less than or equal to n. A rule is valid if the status bit associated with the rule is set to 1 or its version is less than $n - 4$. $10 : 0110$ indicates that the rules for update versions 10, 8 and 7, and 5 and below are valid for packets tagged with that version number and status bits.

The ingresses tag all the packets with the same version and set the same status bits, upon receiving "Commit OK". For ease of implementation, the instruction for tagging packets with a version number and status bits may be separated from the rest of the instructions for the packet (such as forwarding to a port). A single rule tagging all packets with the desired version number and status bits may be installed in a flow table, implemented in software - thus only this instruction needs change and can be changed quickly when a "Commit OK" is received. All packets to an ingress may match the rule in this table first. The packet may then be forwarded to the next flow table in the ingress, which performs the rest of the actions as required.

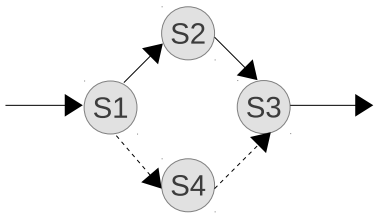
F. Handling Timeouts

The controller must start a timer after sending "Commit", for each version number n. If the timer times out without receiving a "Ready to Commit", the application must be informed. The application must instruct the controller whether to delete the installed rules or to proceed with additional updates with the same version tag. This must be done to enable later stage1-complete updates to proceed. This can be easily accomodated in the algorithm, but for ease of exposition it is not included.

G. Updating only the affected internal switches

A rule "*tcp_port=80, forward 2*" needs to be inserted in an internal switch S1, to reduce the load on another switch S4, as shown in Fig.3. Let two adjacent switches S2 and S3 also have new rules on account of this. The new path is shown in solid lines and the old path in broken lines. Let the version associated with this change be n. After the update is stage1-complete, the switches between the ingresses and S1

Fig. 3. Updating only the affected internal switches



(not shown in the figure) match each packet with a rule whose version field has a value less than or equal to n. When a packet with tag n reaches S1, S2 and then S3, it matches a rule with version n, which is the newly inserted rule. Subsequent switches from S3 to the egress match this packet with rules whose version fields are less than n. Thus switches without genuine rule changes are not affected.

H. Analysis of the algorithm

Suppose a packet P is stamped with a version number n by an ingress switch. Let the new path of the packet as prescribed by the update n be P_n and let the old path be P_o. Only the switches that need a genuine rule change are updated with rules that check for version n. On the rest of the switches the old rules (which do not check for version numbers) are used.

Suppose the controller has decided that the update n, which is stage1-complete, can proceed to the next stage. Now all the ingresses start tagging packets with version number n. Suppose the first internal switch that the packet P traverses is A and it has a rule that matches version n. That rule is applied to the packet. Similarly, all other switches along P_n that have rules of version n will match that rule with P. If the next switch on P_n is B and that does not have a rule of version n, P will match a rule on B that does not check for a version. (A or B cannot have a rule matching P that will check for a version less than n, as such a rule would indicate an intersecting update in progress, which the algorithm does not support.)

Now let us examine what happens when the next update $n+1$ is stage1-complete. Let the set of packets affected by update n be S. Packets belonging to S have the same set of forwarding actions from the ingress to the egress. Let update $n+1$ be such that it does not affect S. All the ingresses now start stamping all packets with the version number $n+1$ and with the status bit for n set to 1. A packet P belonging to S will reach switch A. Let A have a rule that checks for version $n+1$ installed. Since P will not match the rule that checks for version $n+1$ (since update $n+1$ does not affect P) and will match only the rule that checks for version n, the switch will check if the status bit associated with n is set to 1 in P. Since it is, the switch A continues to match P with the rule that checks for n. On switches similar to A, the same behaviour will follow. In switch B, there may be a rule that checks for version $n+1$, but again, P will not match that rule. It will continue to match the rule that does not check for a version number, on B and switches similar to B. The behaviour is similar when the ingresses start stamping packets with version numbers greater than $n+1$.

What happens to the packets belonging to S, *before* the update n is stage1-complete? They have version numbers less than n. Hence even after version n rules are installed, they will not match those rules. These are the old packets that take the path P_o. They get switched along P_o using the old rules that do not check for a version number and continue to get switched along that path even after update n is installed, thus preserving PPC. These old rules get deleted only after $T_c > T_l + M$, by which time, all the old packets would have exited the network - therefore no old packet is dropped.

Thus the new packets belonging to S, regardless of their version numbers, get switched exclusively along the path P_n and the old packets exclusively along the path P_o, preserving PPC. If the paths overlap, the same rules are used by the switches in the overlapping region, but that does not violate PPC.

The status bits indicate to the internal switches the versions of the rules that are not yet stage1-complete and therefore must not be used. Since only 4 bits are used, there can be a gap of utmost 4 updates that are not stage1-complete between two stage1-complete updates $n+5$ and n. If updates $n+1$ through $n+4$ are not stage1-complete and update $n+5$ is, update $n+5$ can proceed to the next stage, by setting all the four status bits to 0. However, if update $n+6$ becomes stage1-complete next, then it cannot proceed to the next stage unless update n is stage1-complete, thus limiting concurrency. If the size of the status window is set to 5 bits, then update $n+6$ can proceed. Thus by changing the size of the status window, concurrency can be improved.

VI. Related Work

An update algorithm called CCG [11] uses fast concurrent update methods but only preserves properties that are not as strict as PPC and falls back to a two-phase update (2PU) [7] for these cases: 1) If an update affects multiple ECs, calculating the paths affected by the update becomes time-consuming 2) If all paths must traverse n waypoints, the old and the new paths can be thought of to consist of n-1 segments. The update of a new segment may depend on the update of an old segment. 3) If properties such as path length constraints need to be met. [19] provides an algorithm that complies with the property of relaxed loop freedom (RLF), which ensures that there are only transient loops during the update, that there are no loops between the source and the destination, that new packets do not loop and that only a constant number of packets loop. This algorithm finds the minimum number of rounds required to update a network, preserving RLF and without using tags. While this does not prevent concurrent disjoint updates, though it is not explicitly addressed, it is unclear if this is faster than a regular 2PU, especially if the network is large; also, it does not preserve PPC. Dionysus [20] preserves PPC and is concurrent, but works only when any forwarding rule at a switch matches exactly one flow. It does not work where the network uses wildcard rules or longest-prefix matching.

Statesman [21] has mechanisms to identify conflicts with the existing network state, with requests from other apps and with dependencies, and resolve them. [22], [23] and [12] deal with various ways of conflict resolutions for concurrent updates. While the emphasis of these efforts is conflict detection and resolution, we are looking at concurrent installations of updates that are already identified as non-conflicting, and preserving PPC.

Frenetic [24], a high-level language for writing network programs, supports functions that a program can call to compose non-conflicting rules and ultimately updates the network using 2PU [25]. NetKAT [26] , which is more powerful, supports a "slicing" [27] abstraction using which conflicting rules can be run on different slices of the network, which, at the physical level, just uses different tags. In both the cases, the emphasis is on conflict resolution and composition of policies within a program whereas we envisage CCU to be able to handle disjoint updates from any source, even from multiple controllers, at a level close to the switches.

[28] describes two algorithms, FIXTAG and REUSETAG, for fault-tolerant updates, when multiple controllers are present in a network. While FIXTAG allows fault-toleant concurrent updates, the tag complexity is exponential in the network size and therefore not practical, whereas REUSETAG reduces tag complexity but allows only sequential updates.

VII. Conclusions and Further Work

The paper describes an algorithm to perform concurrent disjoint updates that preserves per-packet consistency and that works for all scenarios, for an SDN. We observe that, increasing one or more of these overheads improves concurrency: overhead bits in the packet, processing in the switch or the number of messages. We note that the algorithm presented will work well if each concurrent update takes roughly the same amount of time. Therefore it must be examined whether updates can be sized in that manner. We also plan to implement this to understand the tradeoffs and compare the performance of a basic 2PU with this algorithm. A scheme to allow intersecting updates to also be applied concurrently is under development.

References

[1] N. Feamster, J. Rexford, and E. W. Zegura, "The road to SDN: an intellectual history of programmable networks," *Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219

[2] S. Shenker, M. Casado, T. Koponen, and N. McKeown, "A gentle introduction to SDN," 2012. [Online]. Available: http://tce.technion.ac.il/files/2012/06/Scott-shenker.pdf

[3] Open Networking Foundation, "Openflow switch specification version 1.5.0," 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.pdf

[4] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software defined WAN," in *SIGCOMM*. ACM, 2013, pp. 3–14.

[5] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram *et al.*, "Network virtualization in multi-tenant datacenters," in *Networked Systems Design and Implementation*, 2014.

[6] A. Greenberg, "Windows azure: Scaling SDN in the public cloud," in *ONS 2013*, 2013. [Online]. Available: http://www.opennetsummit.org/pdf/2013/presentations/albert_greenberg.pdf

[7] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM 2012*. ACM, 2012, pp. 323–334.

[8] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proceedings of the third workshop on Hot Topics in Software Defined Networking 2014*. ACM, 2014, pp. 175–180.

[9] R. Sukapuram and G. Barua, "Enhanced algorithms for consistent network updates," in *IEEE Conference on Network Function Virtualization and Software Defined Networks*. IEEE, forthcoming-2015.

[10] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[11] W. Zhou, D. Jin, J. Croft, M. Caesar, and B. P. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *NSDI 2015*. Oakland, CA: USENIX Association, May 2015, pp. 73–85.

[12] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul, "Democratic resolution of resource conflicts between SDN control programs," in *CoNEXT 2014*. ACM, 2014, pp. 391–402.

[13] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "Openvirtex: Make your virtual SDNs programmable," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 25–30.

[14] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang, "Meridian: an SDN platform for cloud network services," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 120–127, 2013.

[15] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *CoNEXT 2011*. ACM, 2011, p. 8.

[16] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI, 2010*, vol. 10, 2010, p. 19.

[17] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[18] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker, "Transparent, live migration of a software-defined network," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[19] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 15.

[20] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *SIGCOMM 2014*. ACM, 2014, pp. 539–550.

[21] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin, "A network-state management service," in *SIGCOMM*. ACM, 2014, pp. 563–574.

[22] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner, "Corybantic: Towards the modular composition of SDN control programs," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 1.

[23] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An API for application control of SDNs," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 327–338.

[24] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, no. 9. ACM, 2011, pp. 279–291.

[25] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger *et al.*, "Languages for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 128–134, 2013.

[26] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 113–126, 2014.

[27] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 79–84.

[28] M. Canini, P. Kuznetsov, D. Levin, S. Schmid *et al.*, "A distributed and robust SDN control plane for transactional network updates," in *INFOCOM 2015*, 2015.