Spring 1-1-2016

Efficient Synthesis of Network Updates

Nilesh Jagnik
University of Colorado at Boulder, nija5459@colorado.edu

Efficient Synthesis of Network Updates

by

Nilesh Jagnik

B.Tech., Indian Institute of Technology, 2014

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2016

This thesis entitled:
Efficient Synthesis of Network Updates
written by Nilesh Jagnik
has been approved for the Department of Computer Science

_____

Prof. Pavol Černý

_____

Prof. Sriram Sankaranarayanan

_____

Dr. Ashutosh Trivedi

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Jagnik, Nilesh (M.S., Computer Science)

Efficient Synthesis of Network Updates

Thesis directed by Prof. Pavol Černý

Sofware Defined Networking simplifies network control, configuration and setup by abstracting some steps in these processes. Packet traffic routes in a Software Defined Network(SDN) may need to change due to policy changes. To implement these policy changes, we need to ensure that the SDN is updated consistently. Any packet in the network should be routed either according to the new policy or the old policy. If a packet can take a route which lies partially in the old policy and partially in the new, the network is said to be in an inconsistent state. We present an algorithm that produces an order of node updates which preserves consistency in an SDN. This algorithm will either find an order that preserves efficiency in the SDN or fail stating that such an order does not exist for the SDN.

When a node is updated, its routing tables are changed. However, it may take some time for the network to experience this change as there may be some slow packets on the old routes outgoing from it. For this reason, we may need to wait before we update the next node. Our algorithm finds a consistency preserving order that needs minimum number of waits.

Acknowledgements

First and foremost, I would like to thank my thesis advisor Pavol, for showing interest in me, motivating me, guiding me, believing in me, appreciating me and investing time in me. All credit for everthing that I have been able to present in this thesis, goes to Pavol. In my life, I shall strive to be a good teacher and a good person like you.

Next, I would like to thank my parents and sister, who have supported me throughout my time at CU Boulder. There is no satisfaction that compares to seeing pride in your parents' eyes because of what you have accomplished.

A special thanks goes to Jed, for his suggestions and valuable inputs on this thesis.

I also feel gratitude towards the entire country of U.S.A, which gave me the opportunity to study and discover my interests.

Contents

Figures

Figure

Chapter 1

Introduction

Software Defined Networking(SDN) is revolutionizing the networking industry, by abstracting low level networking tasks and providing high level APIs for network programmers. However, there are still very few mechanisms that reliably abstract the updating of global configurations. Even if initial and final configurations are correct, naïvely updating individual nodes can lead to incorrect transient behaviors, including loops, black holes, and access control violations. In this study, we present an approach for automatically synthesizing updates that are guaranteed to preserve consistency properties with minimal wait time throughout the update. We ensure per-packet consistency [9], a guarantee that every packet traversing the network will follow one global configuration, either old or new, throughout the update. Every packet traverses a route specified either by the policy in place prior to the update or by the updated policy and not a mixture of both. This would ensure that there are no transient loops or blackholes in the network as long as there are none in the initial and final network configurations.

To reliably update all nodes on the network, we may need to pause the update mechanism(wait) after an update, in order to let traffic along paths that were removed get flushed from the network. In Chapter 5, we shall see that a wait may not be required after every update. We find an order of updates such that the number of waits required is minimal.

The network in our model is a directed graph, with non-weighted edges, and we update it from an initial configuration, to a final configuration, with minimal waits, while preserving per-packet consistency throughout the update. Such an order may not always exist, in which case, we state

that a consistency preserving order does not exist. Other papers have presented search-based approaches [4, 3, 5, 11, 8, 7, 10], that deploy models to prune search trees based on heuristics and constraints for preserving consistency. We present a polynomial time algorithm that updates nodes sequentially, and whenever possible, avoids waiting after an update.

Chapter 2 presents a few examples of consistent network updates which illustrate the challenges for finding a consistent order of updates. Chapter 3 formalizes our network model, presents precise definitions of consistency and waits, and states our problem statement. Chapter 4 presents a polynomial time algorithm, OrderUpdate, which finds an order of updates that preserves consistency. In this chapter, we also prove the correctness and completeness of the OrderUpdate algorithm. Chapter 5 presents a modification, PickAndWait, to the OrderUpdate algorithm that minimizes the number of required waits. This modification adds a degree of determinism to the non-deterministic OrderUpdate algorithm using a greedy scheme. We prove that the OrderUpdate algorithm with the PickAndWait modification, is complete, and produces an order of consistent updates with minimal number of waits. We finally discuss related work in Chapter 6 and conclude our results in Chapter 7.

Chapter 2

Examples

Let us consider some easy cases. In Figure 2.1 and Figure 2.2, C_i edges are solid and C_f edges are dashed. In Figure 2.1, there is a trivial update order - A,H1,B. Note that H2 does not need to be updated as it has no outgoing edges. However, in Figure 2.2, no matter the order you update nodes in there will always be inconsistency. This is because H_1 can not be updated unless downstream path from C to H_2 is updated. But also, C can not be updated unless the upstream path from H_1 to C is updated. We refer to this case as the Double Diamond case. There is a circular dependency between H_1 and C.

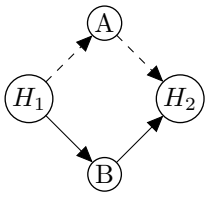

Figure 2.1: A trivial update

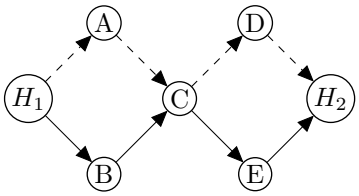

Figure 2.2: Double Diamond case. No update sequence exists.

However, we can not say that the presence of a double diamond implies that there can not be a solution. In Figure 2.3, there is a double diamond between D and G(there are mutiple double diamonds in this figure), but updating B and waiting removes the old traffic incoming to D. The nodes D, E, G, F, H, I and J have no incoming traffic. These disconnected nodes can be updated without worrying about consistency. So, the circular dependency is removed. A valid update order(not considering waits) would be $A, H_1, K, L, B, D, E, F, G, H, I, J, C, M$. So we see that it is

not trivial to know whether an update order exists, and if it exists, to find it.

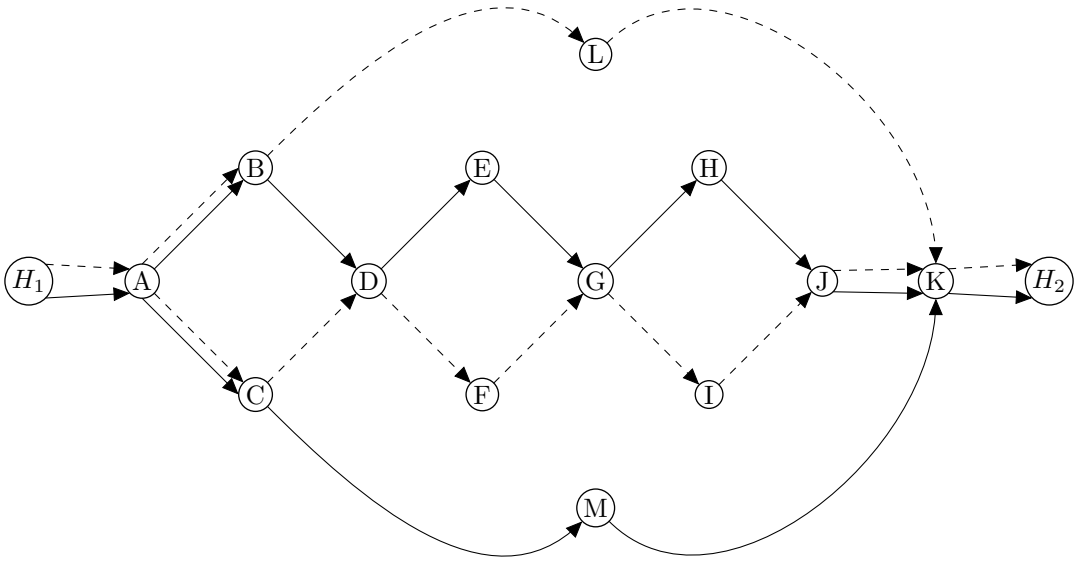

Figure 2.3: Removable Double Diamond. Here C_i edges are solid and C_f edges are dashed.

These examples lead us to believe that we need a systematic algorithm to find out whether a consistency preserving update order exists, and if it does, then find it. We shall present an algorithm that does this and also prove its correctness and completeness.

Chapter 3

The Network Model

Network and Configurations. A topology of the network graph G, is a tuple (N, E), where N is a set of nodes; and E is a given set of directed edges. The configuration, $C \in \mathcal{P}(E)$, of a network G is the set of edges present in G at some time instant. Updating a node removes some old edges and adds new edges, thus changing the configuration of G. Our goal is to bring the network, from an initial configuration C_i, to a final configuration C_f. Random is a function $\mathcal{P}(N) \rightarrow N$, that given a set of nodes P, picks a random node $a = random(P)$ from it.

Sources and Sinks. The directed graph G, in any configuration, has only one source H_1 and one sink H_2.

Cycles. The graph G is acyclic in any configuration. Cycles in G are undesirable as it would mean that traffic can loop forever in the network.

Updates. Let R be the set of all sequences that can be formed using nodes in N without repetition. To update a graph G, we define *upd* as a function $\mathcal{P}(E) \times R \rightarrow \mathcal{P}(E)$, that given a configuration C and a sequence of nodes S, outputs an updated configuration $C' = upd(C, S)$.

Paths. Let Q we the set of all possible directed paths in network G. For obtaining paths, we define *paths* as a function $N \times N \times \mathcal{P}(E) \rightarrow \mathcal{P}(Q)$, that given a start node s, an end node t, and a configuration C, outputs a set of all paths $P = paths(s, t, C)$ between s and t in configuration C. We define the $\in$ operation for a path p and configuration C so that, if $p \in C$, then all edges in path p lie in set C. *nodes* is a function $Q \rightarrow \mathcal{P}(N)$, that given a path q returns a set $S = nodes(q)$ of all nodes on a path.

Consistency. A configuration C is a consistent configuration iff $\forall p \in paths(H_1, H_2, C) : p \in C_i \vee p \in C_f$. There are no paths between H_1 from H_2 that satisfy neither the old policy nor the new policy. This means that if an edge in C lies only in $C_f(C_i)$ then all paths in C that include this edge must lie in $C_f(C_i)$.

Packets. We assume that all packets in the network are of a single type. This assumption lets us focus on consistency issues which occur due to the network graph alone.

Waits. Let $C_c = upd(C_i, U)$ be the current configuration reached after updating a sequence U. Let $C_w = upd(C_i, U')$, where U' is a prefix of U, be an intermediate configuration which was reached while updating sequence U. And we chose to update some node n, s.t. $\exists p \in paths(H_1, n, C_w) : p \notin C_c \wedge \exists q \in paths(n, H_2, upd(C_c, n)) : q \in C_f$, then if there were no waits between C_w and C_c, we need to wait before updating n. This wait is required because some old configuration had a old path which was removed(only C_i paths can be removed). So, traffic along this removed path needs to be flushed as there is a C_f-only path downstream after the update. Not waiting would send C_i traffic on a C_f path, resulting in inconsistency.

Commands. Our update mechanism consists of two commands - update and wait. The update command updates a specified node and changes the configuration of G. Since $|N|$ nodes are updated, the number of update commands is always a constant. The wait command simply pauses the update mechanism for some time to allow packets along old edges to get flushed from the network. The wait command does not change the configuration of G.

The Network Synthesis Problem. We need to find a sequence of commands $U = v_1, v_2, ...v_n$ such that:

(1) After executing the sequence U, G is in configuration C_f.

(2) Configuration of G after executing each command is consistent.

(3) n is minimal.

Or state that such a U does not exist.

Chapter 4

The OrderUpdate Algorithm

4.1 Conditions for updating nodes

At any point of time during the update, let us refer to the intermediate configuration at this time as C_c, denoting current configuration. We shall assume that C_c is consistent and we find a node s to update such that the new configuration $upd(C_c, s)$ is also consistent. Since $C_c = C_i$ initially, this assumption is correct for the first node update. And so, starting from C_i, at every point of time, we find a node to update so that the updated configuration is also consistent. In every intermediate configuration C_c, if a node s follows conditions in Figure 4.1, we can update it. We now explain why these conditions are necessary for getting an order of updates that maintains consistency. To update a node, it needs to satisfy an upstream and a downstream condition of any one of the types listed below. In every case, there is an upstream condition on a node that gives us information about the paths that the packets on the network have taken to reach it from H_1. Based on this information, it may or may not be safe to route the traffic downstream to H_2 according to the new routing policy. For every upstream condition, a downstream condition must be satisfied by a node to be updated. In general, a node that satisfies an upstream condition is called a **candidate** node and if this candidate satisfies the downstream condition, then it is called a **valid** node.

- **Type-A or Disconnected** Nodes - Some nodes in C_c have no traffic incoming to them. Updating these nodes does not cause any change in network traffic and thus maintains

Type	Upstream(Condition for $paths(H_1, s, C_c)$)	Downstream(Condition for $paths(s, H_2, C_c)$)
A	$Z_a = \nexists p \in paths(H_1, s, C_c)$	-
B	$Z_b = \neg Z_a \wedge \forall p \in paths(H_1, s, C_c) : (p \in C_i \wedge p \in C_f)$	$\forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_i \vee p \in C_f$
C	$Z_c = \neg Z_b \wedge \forall p \in paths(H_1, s, C_c) : p \in C_f$	$\forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_f$
D	$Z_d = \neg Z_b \wedge \forall p \in paths(H_1, s, C_c) : p \in C_i$	$paths(s, H_2, upd(C_c, s)) \neq \phi \wedge \forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_i$
E	$Z_e = \neg Z_a \wedge \neg Z_b \wedge \neg Z_c \wedge \neg Z_d = (\exists p_f \in paths(H_1, s, C_c) : p_f \in C_f \wedge p_f \notin C_i) \wedge (\exists p_i \in paths(H_1, s, C_c) : p_i \in C_i \wedge p_i \notin C_f)$	$\forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_i \wedge p \in C_f$

Figure 4.1: Necessary conditions for updating a node s

consistency. There is no distinction between a candidate and valid node in this type because there is no downstream condition.

- **Type-B** - In C_c, all the upstream paths lie in both C_i and C_f. s can be updated if all downstream paths, after updating s, lie in either C_i or C_f.
- **Type-C** Nodes - Type B upstream condition was not satisfied. But in C_c, all the upstream paths lie in C_f. s can be updated if all downstream paths, after updating s, lie in C_f.
- **Type-D** Nodes - Neither Type A, Type B or Type C upstream condition was satisfied. In C_c, all upstream paths lie in C_i. In this case there need to be two downstream conditions. The first condition states that there is a path from s to H_2 after the update. This is so that there are no sinks between s and H_2. This condition is required, because in this case, there are no upsteam C_f paths. Since theer is one sink and one source, having an upsteam C_f path is enough to ensure that there will be a downstream C_f path after the update. The second condition states that all downstream paths, after updating s, must lie in C_i.
- **Type-E** Nodes - A node which does not satisfy any of the above upstream condition is a Type-E node. There are some upstream paths in C_i and some upstream paths in C_f but these two sets are disjoint. To update this node s, all downstream paths, after the update

should be in both C_i and C_f.

The upstream conditions in Figure 4.1 are exhaustive and mutually exclusive. And for each upstream condition, if the corresponding downstream condition is not satisfied, then updating the node will result in a inconsistent state. Hence, the stated conditions are exhaustive and any node that can be updated must satisfy one of the conditions in Figure 4.1.

The basic functionality of the ORDERUPDATE algorithm is searching for and updating valid nodes. We start with a set of nodes N to be updated, which initially contains all nodes, and after updating we remove the updated nodes from N. Finally, $N = \emptyset$ and $C_c = C_f$ since all nodes are updated.

4.2 Correctness and Completeness of ORDERUPDATE Algorithm

The upstream conditions in Figure 4.1 are exhaustive. The corresponding downstream conditions for each type need to be satisfied for updates to be consistent. Hence if Algorithm 1 produces a sequence, it is correct. Since the upstream conditions are exhaustive, if any node was not included in U in Line 14 of Algorithm 1, then it can not be updated in C_c.

Property 1: Removing any paths from C_c does not make C_c an inconsistent state. All remaining paths are still consistent. Another way of saying this is that removing some paths upstream or downstream to a valid node in C_c maintains validity of the node in C_c.

Property 2: Once a C_f path is established between two nodes, it can not be broken by any other update. This is because any update removes C_i paths from C_c but does not remove C_f paths from C_c.

Lemma 1: If $T = UVnY$ is a valid sequence, then if n was valid after updating sequence U, $T' = UnV'Y$ is valid sequence where V' is a permutation of V, i.e $V' = \pi(V)$.

Updating n before any nodes in V adds some paths to $C_c = upd(C_i, U)$ and removes some paths. From Property 1, we know that removing paths from C_c does not change the validity of nodes in V. But there are some nodes for which upstream or downstream paths were added. We now prove

that there exists an order V' in which we can update nodes in V to form an equivalent sequence T'.

Let us argue for all nodes f in V moving along the sequence V from left to right. Here $C_c = upd(C_i, UV(f))$ where $V(f)$ denotes the longest prefix of V that ends before f and $C'_c = upd(C_i, UnV')$ where initially V' is empty sequence:

- Case 1 - There exists some upstream paths from n to f in C'_c which were not in C_c- Node f is downstream from n. Any upstream paths added to C'_c are from C_f. This is because updating the node n removes C_i edges and adds C_f edges to C'_c. f could be a valid node of one of the following types in C_c:
  - TA: This node was disconnected in T, so the only upstream paths in C'_c are from n. All paths from n to f in C'_c are in C_f but not C_i, so downstream paths from f to H_2 are in C_f. f is not updated still all downstream paths in C'_c are in C_f. So outgoing edges from f in C_i are in C_f. We do not update f here. This is because all downstream paths from f in C'_c are already in C_f. Since all outgoing edges from f in C_i are in C_f, updating f would have only added some additional paths to C'_c. From Property 1, not updating f does not affect the validity of any other node in the sequence V. We refer to these nodes that we left out as ***C1-TA*** nodes and we will add them to V' in some other way.
  - TB: A Type-B node becomes a Type-C node if a upstream C_f path is added to it. Since there is an upstream C_f only path to f from n , all downstream paths from f in C'_c are in C_f. Paths starting from f in C'_c are a subset of the paths in C_c because some nodes in the sequence V were not updated. Paths in C_c after updating f were either in C_i or C_f. Paths which were in C_i must exist before updating f as well. If any of these C_i paths existed in C'_c, then they have to be in C_f as in C'_c all paths downstream from f are in C_f. Hence all paths after updating f will be in C_f. Update f. $V' = V'f$.

  - TC: A Type-C node stays a Type-C node if some upstream paths from C_f are added to C'_c. This node can is still valid. We update it here. $V' = V'f$.
  - TD: Updating a Type-D node does not add any paths to its current configuration. It only removes some paths from it. Updating f may remove some paths from C'_c but from Property 1, this update is safe. $V' = V'f$.
  - TE: The Type-E downstream condition is very strong. Any node satisfying this condition can be updated no matter what type of candidate it is. So adding an upstream path may change the candidacy of this node but its downstream condition will still be fulfilled. f is still valid for update. $V' = V'f$.

- Case 2 - There exist some downstream paths from f to n in C'_c which were not in C_c- Node f is upstream from n. Any downstream path added was in C_f because updating n can only add C_f paths to C'_c. There can not be any C_i only paths from f to n in C'_c else there would be inconsistency on the downstream C_f paths. So, all paths from f to n in C'_c are in C_f. Since the paths added in C'_c are also C_f paths, f can be updated because updating f can only add more C_f paths between f and n. $V' = V'f$.
- Case 3- There exist no downstream paths to or upstream paths from f in C'_c which were not in C_c - Since no paths were added, f stays valid and can be updated.$V' = V'f$.
- Case 4- There exist both downstream paths to and upstream paths from f in C'_c which were not in C_c - This case is not possible since there can not be any cycles in C'_c.

So far, we removed C1-TA nodes from sequence V and created a subsequence V'. However we need to add these nodes to V' in some order to make sure that $V' = \pi(V)$. To achieve this we find the C1-TA node which has no other C1-TA node as its descendant in C_f. We update this node, add this node to V' and repeat this process until no C1-TA node is remaining. There is always one such node without C1-TA descendants in C_f because there are no cycles in the graph. We shall now prove that updating this node f maintains consistency. Let $C_1 = upd(C_i, UVn)$ and $C'_c = C_2 = upd(C_i, UnV')$. The only difference between C_1 and C_2 is that some C_f paths in C_1 are

missing from C_2 because some C1-TA nodes were not updated. Updating f in C_2 will add these paths to C_2. Since there is no C1-TA descendant, updating f will make C_1 and C_2 identical w.r.t paths starting at f.

We have obtained $V' = \pi(V)$. After this point, since updating the same set of nodes in any order leads to the same configuration, $upd(C_i, UVn) = upd(C_i, UnV')$, nodes in Y can be updated in sequence. Hence we proved that $T' = UnV'Y$ is a correct sequence.

Theorem - Algorithm 1 generates a valid order of updates if there exists one.

Let $Q = s_1, s_2....s_n$ be an valid sequence of updates, and $Q_{alg} = s'_1, s'_2, ...s'_n$ be the solution generated by Algorithm 1. Let r be the first node s.t. $\forall i < r : s_i = s'_i$. Then using Lemma 1, there is another sequence $Q' \equiv Q$ s.t. $\forall i \leq r : s_i = s'_i$. Using this argument for every index from i to n, we can find a valid sequence $Q'' \equiv Q_{alg}$.

Algorithm 1: ORDERUPDATE

Input: Set of nodes to be updated N, Network Initial Configuration C_i, Network Final Configuration C_f

Result: An order of consistent node updates

$W \leftarrow \emptyset$ // Waitlist is initially empty

$C_c \leftarrow C_i$ // C_c starts with the initial value of C_i

while $C_c \neq C_f$ // Stop when C_c and C_f are equal

do

  $V_a \leftarrow CN_a \leftarrow \{s \mid s \in N \wedge \nexists p = path(H_1, s, C_c)\}$ // Type-A Valid/Diconnected Nodes

  $CN_b \leftarrow \{s \mid s \in N \wedge s \notin CN_a \wedge (\forall p \in paths(H_1, s, C_c) : (p \in C_i \wedge p \in C_f))\}$ // Type-B Candidates

  $V_b \leftarrow \{s \mid s \in CN_b \wedge (\forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_i \vee p \in C_f)\}$ // Type-B Valid Nodes

  $CN_c \leftarrow \{s \mid s \in N \wedge s \notin CN_a \wedge s \notin CN_b \wedge (\forall p \in paths(H_1, s, C_c) : p \in C_f)\}$ // Type-C Candidates

  $V_c \leftarrow \{s \mid s \in CN_c \wedge (\forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_f)\}$ // Type-C Valid Nodes

  $CN_d \leftarrow \{s \mid s \in N \wedge s \notin CN_a \wedge s \notin CN_b \wedge s \notin CN_c \wedge (\forall p \in paths(H_1, s, C_c) : p \in C_i)\}$ // Type-D Candidates

  $V_d \leftarrow \{s \mid s \in CN_d \wedge paths(s, H_2, upd(C_c, s)) \neq \phi \wedge (\forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_i)\}$ // Type-D Valid Nodes

  $CN_e \leftarrow \{s \mid s \in N \wedge s \notin CN_a \wedge s \notin CN_b \wedge s \notin CN_c \wedge s \notin CN_d\}$ // Type-E Candidates

  $V_e \leftarrow \{s \mid s \in CN_e \wedge (\forall p \in paths(s, H_2, upd(C_c, s)) : p \in C_i \wedge p \in C_f)\}$ // Type-E Valid Nodes

  $U \leftarrow V_a \cup V_b \cup V_c \cup V_d \cup V_e$

  if $U = \emptyset$ **then**

    EXIT // No consistent order of updates exists

  end

  $s = PickAndWait()$ // By default, pick a random node, and wait

  $C_c \leftarrow C_c - \{e | e = edge(s, t) \in C_i\}$ // Remove old outgoing edges from C_c

  $C_c \leftarrow C_c \cup \{e | e = edge(s, t) \in C_f\}$ // Add new outgoing edges to C_c

  $N \leftarrow N - \{s\}$ // Remove updated nodes from node list

end

Chapter 5

Minimizing Waits

In the previous section, we showed that Algorithm 1 produces a consistent order of updates. In this section, we shall extend Algorithm 1 by modifying the *PickAndWait()* subroutine on Line 18.

5.1 Purpose of Waits

When Algorithm 1 picks a node to update, it removes all outgoing edges from it on Line 19, and updates the current network configuration. In the next iteration, it picks a node which is valid in the current configuration. If we did not flush the packets on these old edges, our network may not have actually reached the current configuration we assume it to be in. In Figure 2.3, we need to have a wait between B and D.

5.2 Condition for Waits

We require a wait if there is an non-flushed C_i-only path upstream which got removed and there is a C_f only path in the current configuration after the update. These conditions are formalized in the $waitUp()$ and $waitDown()$ clauses in Algorithm 2, in Line 1 and Line 2. To keep track of flushed paths, we maintain a waitlist W. Any updated node which had traffic flowing through it on a C_i-only path would be added to waitlist W. We refer to these nodes as **waitlist** nodes. A node which has doesnot have any upstream C_i path is called a **non-waitlist** node. An example is this is C1-TA node from Lemma 1. Any node which has an ancestor in W on a C_i-only path which

was previously removed needs to wait.

5.3 A Greedy strategy

We define priorities and update nodes based on these priorities. The idea here is to delay waits. Delaying waits would allow the waitlist to build up as much as possible before it is flushed. We try to keep the wait as far down the sequence as possible. This presents the possibilty that some of the waits in a non-optimal sequence would be pushed out of the sequence. Any node which does not need to wait is given a higher priority, priority P_0. Nodes that need to wait are given priority P_1.

5.4 Proof of Optimality

Property 3: If a node $n \in P_0$ in configuration $C_1 = upd(C_i, S)$, where S is a sequence of nodes, then for all sequences S' of nodes not in S, if n is valid in configuration $C_2 = upd(C_i, SS')$, $n \in P_0$ in C_2.

Suppose in C_2, $n \in P_1$, then either an upstream conditions was added to n in C_2 or a downstream condition was added to n in C_2 or both.

- Case 1- Upstream condition was added in C_2 - This means that in C_1, n had an upstream C_i path which got removed in C_2. There can't be a downstream C_f-only path in $upd(C_1, n)$. So, If an upstream condition was added, downstream condition was added too. This case is equivalent to case 2.

- Case 2- Downstream condition was added in C_2 - The downstream C_f-only path was not present in $upd(C_1, n)$ but is present in $upd(C_2, n)$. This path is added by some node r in sequence S'. r was downstream from n and connected to it in C_1. In $upd(C_1, n)$, n had no downstream C_f-only paths, so n and r are connected by a C_i-only path. So ancestors of n are ancestors of r and would have been flushed when r was updated. Also, since r added a C_f-only path downstream, all C_i-only paths upstream have been removed. In this case,

$n \in P_0$.

- Case 3- Both upstream and downstream condition was added in C_2. This case is already covered by case 2.

This proves Property 3. Any node that becomes priority P_0 at some time, it stays priority P_0 whenever it is valid in future.

Lemma 2: If $T = UVnY$ is a valid sequence, and after updating sequence U, $n \in P_0$, then $T' = UnV'Y$ is valid sequence with lesser or equal waits. Here V' is a permutation of V, i.e $V' = \pi(V)$.

This lemma is an extension of Lemma 1. In addition to n being a valid node, n is a Priority P_0 node. We shall prove that if n satisfies these constraints, then T' can be constructed using the same V' as in Lemma 1. In T', after updating U and n, we first update all nodes in V which were not C1-TA, in order. We then update all C1-TA nodes in a downstream first order. This way, we build V' from V in two phases:

- Phase 1 - In this phase we update all but C1-TA nodes. Let us argue for each f in Phase 1. For the following proof, C_c refers to the configuration before f is updated in T and C'_c is the corresponding configuration in T':
  - Case 1 - If $f \in P_1$ in T, then in T', $f \in P_0$ or $f \in P_1$. In either case, f does not add any waits in T' as compared to T.
  - Case 2 - If $f \in P_0$ in T. If $f \in P_0$ in T', then no additional waits are added to T' due to f. However, if $f \in P_1$ in T', $waitUp(f)$ and $waitDown(f)$ are both true in T'. This change in priority could be due to one of the following reasons:
    - Upstream condition was added in T'. Downstream condition was unchanged. The change in upstream condition could be because of two reasons:
      (1) Some nodes, which were flushed from the waitlist in T, were not flushed in T'. This was because:

(a) A C1-TA node which had priority P_1 in T was not updated. We add a wait a before f. This wait was shifted from one node to another. Using property 3, we can say that the C1-TA node would have priority P_0 whenever it is updated. This wait is possibly delayed with respect to Phase 1 nodes. Phase 2 nodes do not get added to the waitlist and so, only considering the nodes that will get added to waitlist, this wait either stays in its relative position or moves to the right.

(b) Some node before f satisfied Case 1 and did not wait. We wait before f. Here again, we shift the wait from one node to another. The wait here has shifted right.

(2) Some upstream C_i-only upstream paths were not removed from C_c are removed in C_c'. Since the downstream condition is unchanged, it exists in C_c, and f cannot have any upstream C_i-only paths. All such paths would have been removed from C_c as well. This case is not possible.

– Downstream condition was added in T'. The upstream condition may or may not have been added in T'. A downstream C_f-only path exists in T' which did not exist in T. We saw in Lemma 1, that not updating C1-TA nodes does not add paths in the network. This path could only have been added because of n, and that too, only if n is downstream from f in C_c and C_c'. There is a path from f to n in C_c. This path can not be C_f-only since f did not satisfy the downstream condition in T. This path is a C_i path. Since a downstream condition for f can only be caused by n, this path exists in C_c' as well. When n was updated, it added a C_f-only downstream path, so all upstream C_i-only paths would have been removed. Additionally, since n was updated in T', we know that all ancestors on these C_i-only upstream paths were already flushed or non-waitlist nodes. Since all ancestors of f were flushed or non-waitlist nodes, it

is not possible for f to have priority P_1.

- Phase 2 - These nodes do not have upstream C_i paths so they do not get added to the waitlist. Additionally, there does not need to be more than one wait in this phase. This is because, before the beginning of this phase, all upstream C_i-only paths would have been removed(these nodes were disconnected in C_c), and ancestors along these paths, added to waitlist. Adding one wait would flush all these ancestors at once.

  * If f had priority P_1 in T, then if it still has priority P_1, then the wait before it was not shifted and if we wait before f there would not be any additional waits in T' as compared with T.

  * If f had priority P_0 in T, and P_1 in T', there can be 2 reasons for this, like the corresponding case in Phase 1:

    – Upstream condition was added. Downstream condition was unchanged. Since these nodes have all their C_i-only upstream paths removed from C'_c, the only way an upstream condition was added was if some nodes were not flushed. In Phase 1, all waits either stayed in place or were shifted right. The only way that some nodes that were flushed in T were not flushed in T' is that one wait shifted right and got dropped. We can add this wait here.

    – Downstream condition was added. This case is the same as the case for Phase 1 and is not possible.

We proved that waits in sequence $UnV' \leq$ waits in sequence UVn. We have also seen that all waits are either at the same position or are delayed. This delay in waits would mean that at the end of UnV', more nodes would be flushed and fewer nodes would be on the waitlist as compared to UVn. So, waitlist in sequence $UnV' \subseteq$ waitlist in sequence UVn. This would mean that while updating Y in T', the number of waits either stays the same or reduces. Hence, we proved That waits in sequence $T' \leq$ waits in sequence T.

Lemma 3: If $T = UVnY$ is a valid sequence, and after updating sequence U, $P_0 = \emptyset \wedge n \in P_1$, then $T' = UnV'Y$ is valid sequence with lesser or equal waits. Here V' is a permutation of V, i.e $V' = \pi(V)$.

Here again, V' is formed from V in the same way as before. In T, there is a wait before any node in V is updated because $P_0 = \emptyset$ after updating sequence U. If n is updated before V, then by Property 3, the first node in V' would have to be in P_0, and thus one wait from V' is removed and added before n in T'. The argument for rest of the nodes in V' and Y would stay the same. In T if there was a wait before n, then there still is one. However, if in T, there was no wait before n, then one wait is borrowed from nodes in V'. Overall, waits in sequence $T' = UnV'Y \leq$ waits in sequence $T = UVnY$.

Theorem 2: Algorithm 1 and Algorithm 2 produce a valid order of updates with minimal number of waits if there exists one.

Let $Q = s_1, s_2....s_n$ be an optimal valid sequence, and $Q_{alg} = s'_1, s'_2, ...s'_n$ be the sequence generated by Algorithm 1 and Algorithm 2. Let r be the first node s.t. $\forall i < r : s_i = s'_i$. If $s'_r \in P_0$, then by Lemma 2, we can generate a sequence $Q' \equiv Q$ s.t. $\forall i \leq r : s_i = s'_i$. If $s'_r \in P_1$, then by Lemma 3, we can again generate a sequence $Q' \equiv Q$ s.t. $\forall i \leq r : s_i = s'_i$. Using this argument for every index from i to n, we can find a valid sequence $Q'' \equiv Q_{alg}$.

Algorithm 2: PickAndWait

Result: Return a node that minimizes waits in the sequence

$waitDown(q) = (\exists p \in paths(q, H_2, upd(C_c, q)) : p \in C_f \wedge p \notin C_i)$

$waitUp(q) = (\exists p \in paths(H_1, q, C_i) : p \notin C_c \wedge p \notin C_f \wedge (\exists s \in nodes(p) : s \in W))$

$P_0 \leftarrow \{s \mid s \in U \wedge \neg(waitDown(s) \wedge waitUp(s))\}$
// Nodes we can update without waiting

$P_1 \leftarrow \{s \mid s \in U \wedge waitDown(s) \wedge waitUp(s)\}$
// Nodes we can not update without waiting first

if $P_0 \neq \emptyset$ **then**
  $R = random(P_0)$ // Return any node in P_0
end
else
  WAIT // Need to wait before updating P_1 nodes
  $R = random(P_1)$ // Return any node in P_1
end
if $\exists p \in paths(H_1, R, C_c) : p \in C_i$ // If R is a waitlist node, add it to W
then
  $W \leftarrow W \cup \{R\}$
  // R is a waitlist node if it has traffic incoming on a C_i path
end
return R

Chapter 6

Related Work

There is a considerable amount work related to avoiding erratic transient behavior that arises while updating routes in a Software Defined Network.

Consistency. Our work is motivated by earlier work on network updates in SDN [9] that proposed the notion of per-packet consistency and provided mechanisms for consistent updates like two phase updates.

Exponential Search Based Algorithms. Dionysus [4] achieves fast, consistent network updates through dynamic scheduling of rule updates. Swan and zUpdate add support for bandwidth guarantees [3, 5]. The CCG [11] framework supports customizable consistency policies during network updates. McClurg et al. [8] present a update synthesis algorithm based on counter-example guided search and incremental model checking. More recent work [7] introduces event-driven consistent updates using network event structures to model contraints on updates. The FLIP [10] Algorithm computes policy preserving per-flow sequences.

Complexity results. Ludwig et al. [6] study schedules that minimize controller interactions(rounds) in a loop-free manner and show that deciding whether a k-round schedule exists, is NP-complete for $k = 3$. Förster et al. [2] show that for the basic consistency properties of loop and blackhole freedom, fast updates are NP-hard optimization problems and present a algorithm with provably minimal dependency structure. The constraint of per-packet consistency in our problem includes loop freedom and blackhole freedom, and finds a sequential update schedule. Brandt et al. [1] give a polynomial time algorithm to decide if congestion free configuration change is possible when

flows are splittable. This notion of congestion freedom is different than our notion of per-packet consistency as packets may take a path which lies partially in the old configuration and partially in the new configuration.

Chapter 7

Conclusion

Our discussion so far can be summarized into the following high level points:

- We presented a polynomial time algorithm, OrderUpdate, to find an order of consistent updates for the nodes in a network with unweighted directed edges and single packet type.
- We proved that the OrderUpdate algorithm is correct and complete.
- We then presented a modification, PickAndWait, to the OrderUpdate algorithm which finds an order of consistent updates with minimal number of waits.
- We proved that the OrderUpdate algorithm, with the PickAndWait modification, is correct, complete and optimal.

Bibliography

[1] Sebastian Brandt, Klaus-Tycho Förster, and Roger Wattenhofer. On Consistent Migration of Flows in SDNs. INFOCOM, 2016.

[2] Klaus-Tycho Förster, Ratul Mahajan, and Roger Wattenhofer. Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes. IFIP, 2016.

[3] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, pages 15–26, New York, NY, USA, 2013. ACM.

[4] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, pages 539–550, New York, NY, USA, 2014. ACM.

[5] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, pages 411–422, New York, NY, USA, 2013. ACM.

[6] Arne Ludwig, Jan Marcinkowski, and Stefan Schmid. Scheduling loop-free network updates: It's good to relax! In Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC '15, pages 13–22, New York, NY, USA, 2015. ACM.

[7] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerný. Event-driven SDN programs. CoRR, abs/1507.07049, 2015.

[8] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient synthesis of network updates. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pages 196–207, New York, NY, USA, 2015. ACM.

[9] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[10] Stefano Vissicchio and Luca Cittadini. Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In INFOCOM, 2016. To appear.

[11] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 73–85, Oakland, CA, May 2015. USENIX Association.