Efficient and Consistent Flow Update for Software Defined Networks

Seminar Report

submitted in partial fulfillment of the requirement
for award of Degree of

BACHELOR OF TECHNOLOGY

In

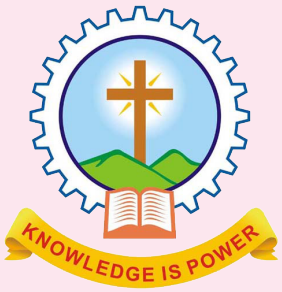
COMPUTER SCIENCE AND ENGINEERING

of

APJ Abdul Kalam Technological University

Submitted by

ARCHANA M



Department of Computer Science and Engineering
Mar Athanasius College of Engneering
Kothamangalam

Efficient and Consistent Flow Update for Software Defined Networks

Seminar Report

submitted in partial fulfillment of the requirement
for award of Degree of

BACHELOR OF TECHNOLOGY

In

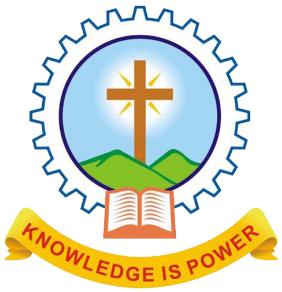
COMPUTER SCIENCE AND ENGINEERING

of

APJ Abdul Kalam Technological University

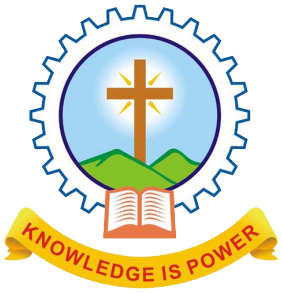
Submitted by

ARCHANA M



Department of Computer Science and Engineering
Mar Athanasius College of Engneering
Kothamangalam

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MAR ATHANASIUS COLLEGE OF ENGINEERING
KOTHAMANGALAM



CERTIFICATE

This is to certify that the report entitled **Efficient and Consistent Flow Update for Software Defined Networks** *submitted by* **Ms.ARCHANA M**, Reg. No. **MAC15CS015** *towards partial fulfillment of the requirement for the award of Degree of Bachelor of Technology in Computer science and Engineering from APJ Abdul Kalam Technological University for the year 2019 is a bonafide record of the seminar carried out by her under our supervision and guidance.*

.....	.....	.....
Prof. Joby George	**Prof. Neethu Subash**	**Dr. Surekha Mariam Varghese**
Faculty Guide	***Faculty Guide***	***Head of the Department***

Date: Dept. Seal

ACKNOWLEDGEMENT

First and foremost, I sincerely thank the 'God Almighty' for his grace for the successful and timely completion of the seminar.

I express my sincere gratitude and thanks to Dr. Solly George, Principal and Dr. Surekha Mariam Varghese, Head Of the Department for providing the necessary facilities and their encouragement and support.

I owe special thanks to the staff-in-charge Prof. Joby George and Prof. Neethu Subhash for their corrections, suggestions and sincere efforts to co-ordinate the seminar under a tight schedule.

I express my sincere thanks to staff members in the Department of Computer Science and Engineering who have taken sincere efforts in helping me to conduct this seminar.

Finally, I would like to acknowledge the heartfelt efforts, comments, criticisms, co-operation and tremendous support given to me by my dear friends during the preparation of the seminar and also during the presentation without whose support this work would have been all the more difficult to accomplish.

ABSTRACT

SDN provides flexible and scalable routing by separating control plane and data plane. With centralized control, SDN has been widely used in traffic engineering, link failure recovery, and load balancing .This work considers the flow update problem, where a set of flows need to be migrated or rearranged due to change of network status. During flow update, efficiency and consistency are two main challenges. Efficiency refers to how fast these updates are completed, while consistency refers to prevention of black holes, loops and network congestions during updates. There are four phases. The first phase partitions flows into shorter routing segments to increase update parallelism. The second phase generates a global dependency graph of these segments to be updated. The third phase conducts actual updates. The last phase deals with deadlocks. Through simulations, it is validated that this scheme not only ensures freedom of black holes, loops, congestions, and deadlocks during flow updates, but is also faster than existing schemes.

Contents

List of figures

List of tables

List of abbreviation

SDN	Software Defined Network
AvlBw	Available Bandwidth
TxBw	Transmission Bandwidth
PotUpTm	Potential Update Time
OVS	Open vSwitch
CLI	Command Line interface

Introduction

SDN is appealing because it provides flexible and dynamic routing by separating control and data planes. It has a global view on network topology due to its centralized control and is thus widely applied to traffic engineering [1], link failure recovery[2] and load balancing[3]. As network scales up, the frequency of flow updates, where a set of flows need to be migrated to new routes, also increases.

In a software-defined network, a network engineer or administrator can shape traffic from a centralized control console without having to touch individual switches in the network. The centralized SDN controller directs the switches to deliver network services wherever they're needed, regardless of the specific connections between a server and devices.

This process is a move away from traditional network architecture, in which individual network devices make traffic decisions based on their configured routing tables. The application layer, not surprisingly, contains the typical network applications or functions organizations use, which can include intrusion detection systems, load balancing or firewalls. Where a traditional network would use a specialized appliance, such as a firewall or load balancer, a software-defined network replaces the appliance with an application that uses the controller to manage data plane behavior.

The control layer represents the centralized SDN controller software that acts as the brain of the software-defined network. This controller resides on a server and manages policies and the flow of traffic throughout the network.These three layers communicate using respective northbound and southbound application programming interfaces.For example, applications talk to the controller through its northbound interface, while the controller and switches communicate using southbound interfaces, such as OpenFlow.

SDN encompasses several types of technologies, including functional separation, network virtualization and automation through programmability. Originally, SDN technology focused solely on separation of the network control plane from the data plane. While the control plane makes decisions about how packets should flow through the network, the data plane actually moves packets from place to place. In a classic SDN scenario, a packet arrives at a network switch, and rules built into the switch's proprietary firmware tell the switch where to forward the packet. These packet-handling rules are sent to the switch from the centralized controller. The switch also known as a data plane device queries the controller for guidance as needed, and it provides the controller with information about traffic it handles. The switch sends every packet going to the same destination along the same path and treats all the packets the exact same way.

Software defined networking uses an operation mode that is sometimes called adaptive or dynamic, in which a switch issues a route request to a controller for a packet that does not have a specific route. This process is separate from adaptive routing, which issues route

requests through routers and algorithms based on the network topology, not through a controller. The virtualization aspect of SDN comes into play through a virtual overlay, which is a logically separate network on top of the physical network. Users can implement end-to-end overlays to abstract the underlying network and segment network traffic. This micro segmentation is especially useful for service providers and operators with multi tenant cloud environments and cloud services, as they can provision a separate virtual network with specific policies for each tenant.

With SDN, an administrator can change any network switch's rules when necessary - prioritizing, deprioritizing or even blocking specific types of packets with a granular level of control and security. This is especially helpful in a cloud computing multi-tenant architecture, because it enables the administrator to manage traffic loads in a flexible and more efficient manner. Essentially, this enables the administrator to use less expensive commodity switches and have more control over network traffic flow than ever before. Other benefits of SDN are network management and end-to-end visibility[4]. A network administrator need only deal with one centralized controller to distribute policies to the connected switches, instead of configuring multiple individual devices. This capability is also a security advantage because the controller can monitor traffic and deploy security policies. If the controller deems traffic suspicious, for example, it can reroute or drop the packets.

SDN also virtualizes hardware and services that were previously carried out by dedicated hardware, resulting in the touted benefits of a reduced hardware footprint and lower operational costs. Additionally, software-defined networking contributed to the emergence of software defined wide area network (SD-WAN) technology. SD-WAN employs the virtual overlay aspect of SDN technology, abstracting an organization's connectivity links throughout its WAN and creating a virtual network that can use whichever connection the controller deems fit to send traffic.

Another challenge with SDN is there's really no established definition of software-defined networking in the networking industry. Different vendors offer various approaches to SDN, ranging from hardware-centric models and virtualization platforms to hyper-converged networking designs and controller less methods.Some networking initiatives are often mistaken for SDN, including white box networking, network disaggregation, network automation and programmable networking. While SDN can benefit and work with these technologies and processes, it remains a separate technology.

SDN technology emerged with a lot of hype around 2011, when it was introduced alongside the OpenFlow protocol[6]. Since then, adoption has been relatively slow, especially among enterprises that have smaller networks and fewer resources. Also, many enterprises cite the cost of SDN deployment to be a deterring factor. Main adopters of SDN include service providers, network operators, telecoms and carriers, along with large companies, like Facebook and Google, all of which have the resources to tackle and contribute to an emerging technology.

During flow update, efficiency and consistency are two main challenges. Efficiency refers to how fast a set of given flow updates are completed, while consistency refers to prevention of blackholes,loops, congestions and deadlocks during updates. Blackhole-free means that there is no tentative flow leading to a deadend . Loop-free means that there is no tentative circular route in the network. Congestion-free means the sum of flows on a link should not exceed its link capacity.

This is where the need for SDN arises.SDN can control its update order and is expected to maintain these properties by computing a suitable update schedule . A global dependency graph is generated to dynamically adjust update schedules.To increase its update speed uses critical nodes and local dependency graph to speed up updating, but it incurs congestions in some cases. It also does not have a well-designed mechanism for handling deadlocks.

A dynamic algorithm for flow update[5] in a SDN by considering both consistency and efficiency is designed which consists of four phases. In the first phase, each flow to be updated is partitioned into multiple segments to increase update parallelism. In the second phase, a global dependency graph is generated from these flow segments. In the third phase , update rules of flow segments and adjust the dependency graph dynamically for efficiency while maintaining network consistency. In the last phase deadlocks that are not yet solved in the third phase are handled.

Related works

SDN due to its centralized control is thus widely applied to traffic engineering[7]. Combining SDN with traditional network as shown in Fig 2.1 leads to faster link failures recovery. Cost of flow updates is a key issue influencing efficiency of SDN and deciding optimized routing paths is critical as SDN scales up[8].

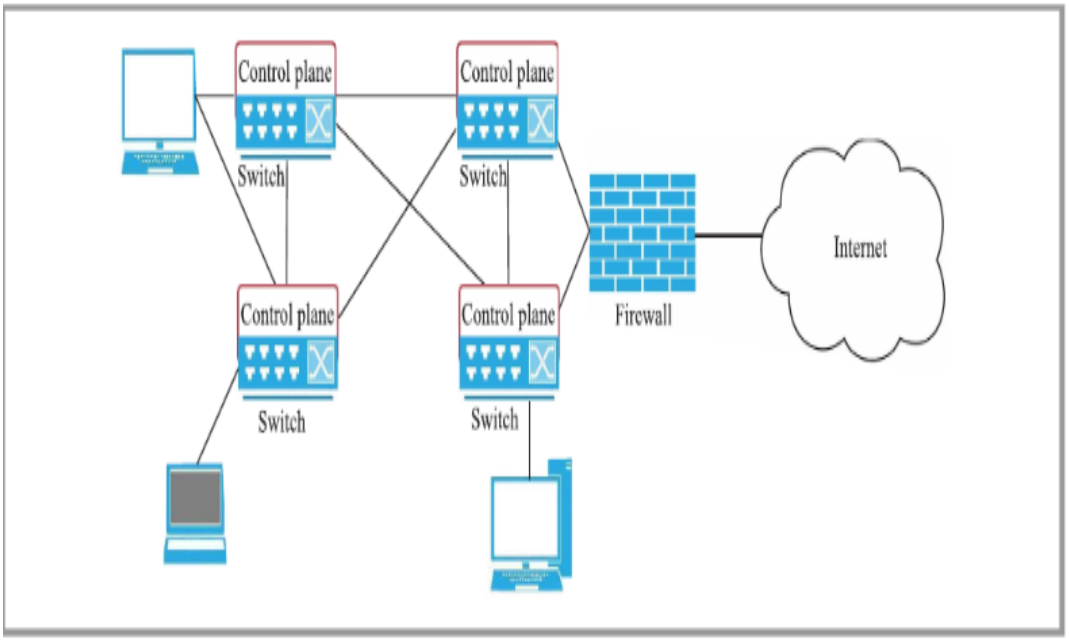


Figure 2.1: Traditional network architecture

Static scheme for scheduling flow updates can decrease transition delay. However static schemes arrange update order in advance and the update order can not be adjusted during update, which is not suitable for network with frequent update needs.Several related works were performed.

One of the most widely used SDN enabler is the Open-Flow v.1.3 protocol. It allows the controller to manage the OpenFlow switches. The OpenFlow switches contain one or more ow tables, a group table, and a secure OpenFlow channel (Fig 2.2). The ow tables and the group table are used for packet look up and then to forward the packets.The OpenFlow channel is an abstraction layer. It establishes a secure link between each of the switches and the controller via the OpenFlow protocol. This channel abstracts the underlying switch hardware. As of OpenFlow version 1.5, a switch can have one or more OpenFlow channels that are connected to multiple controllers.SDN is, generally, a ow-based control strategy. Through the OpenFlow a controller can dene how the switches should treat the ows.

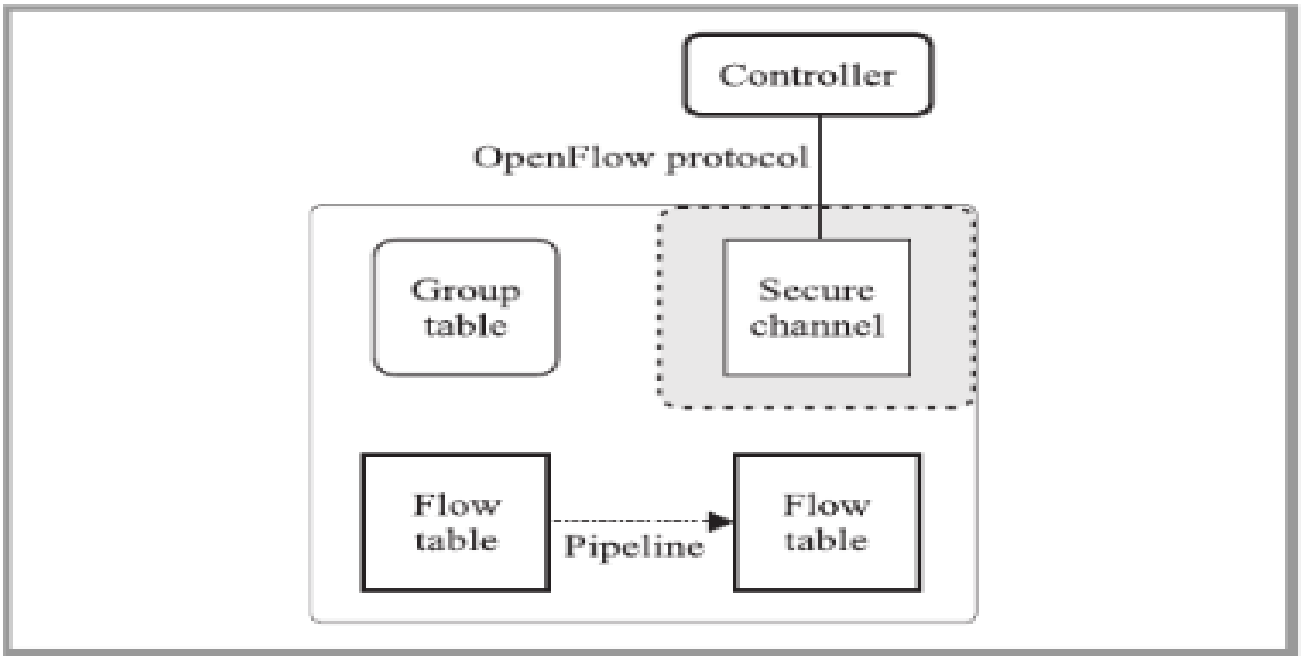


Figure 2.2: Main openFlow switch components

In a SDN when a source node sends data to the destination, the switch sends the rst packet to the controller, since it doesn't know how to treat this packet.The controller calculates the path for this packet and installs the appropriate rules in the switches on the packet's path. The trac engineering extensions toIS-IS and OSPF standard, extends these protocols by incorporating the trac load while selecting a path. In these approaches during link state advertisements, routers advertise the trac load along with link costs. After routers ex-change link costs and trac loads, then they calculate the shortest path for each destination. These standards require the routers to be modied to collect and exchange trac statistics.

2.1 Traffic engineering

SDN combined with traditional network leads to faster link failures recovery.The network will have traditional IP routers and SDN switches coexisting which take advantage of SDN and propose an approach to guarantee traffic reachability in case of link failure by redirecting traffic on the failed link to SDN switches through pre-configured IP. With the help of coordination among SDN switches it is able to explore multiple backup paths for the failure recovery.

An important technique to optimize a network and improve network robustness is traffic engineering. As traffic demand increases, traffic engineering can reduce service degradation and failure in the network. To allow a network to adapt to changes in the traffic pattern, the research community proposed several traffic engineering techniques for the traditional networking architecture. However, the traditional network architecture is difficult to manage. Software Defined Networking (SDN) is a new networking model, which decouples the control plane and data plane of the networking devices. It promises to simplify network management, introduces network programmability, and provides a global view of network state. To exploit the potential of SDN, new traffic engineering methods are required. This paper

surveys the state of the art in traffic engineering techniques with an em- phasis on traffic engineering for SDN. It focuses on some of the traffic engineering methods for the traditional network ar- chitecture and the lessons that can be learned from them for better traffic engineering methods for SDN-based networks. This paper also explores the research challenges and future directions for SDN traffic engineering solutions.

To address this problem the research community started working on trac engineering and proposed new ways to improve network robustness in response to the growth of trac demands. Trac engineering reduces the service degradation due to congestion and failure, e.g. link failure.Fault tolerance is an important property of any network. It is to ensure that if a failure exists in the network, still the requested data can be delivered to the destination. Network operators have to manually congure these multi vendor devices to respond to a variety of applications and event in the network. Often they have to use limited tools such as CLI and some times scripting tools to convert these high-level conguration policies into low-level policies. This makes the management and optimization of a network dicult, which can introduce errors in the network. Other problems with this architecture can cause oscillations in the network, since control planes of the devices are distributed, innovation is dicult because the vendors prohibit modication of the underlying software in the devices.

From trac engineering point of view, even though these techniques perform well, they suer from several limitations such as, they take routing decision locally, and it is dicult to change the link weights dynamically. In addition, while sending trac these techniques consider few criteria, such as link capacity.

The new networking paradigm, SDN, has introduced new characteristics such as :

- separation of the control plane functionality, and the data plane functionality
- network programmability, SDN provides an open standard, which allows external applications to pro-gram the network
- facilitates innovation, new protocols and control applications can be introduced because OpenFlow provides the required abstractions, so we do not need to know the switch internals and conguration
- ow management, through the OpenFlow a controllercan dene ows in dierent granularity, and how theswitches should treat the ows

2.2 B_4: software defined wide area network

B4 is a private WAN connecting Google's data centers across the planet. B_4 has a number of unique characteristics:

- massive bandwidth requirements deployed to a modest number of sites
- elastic traffic demand that seeks to maximize average bandwidth
- full control over the edge servers and network, which enables rate limiting
- cost sensitivity

Elastic bandwidth demands

Majority of B_4 data center traffic involves synchronizing large data sets across sites.These applications benefit from as much bandwidth as they can get but can tolerate periodic failures with temporary bandwidth reductions.

Moderate number of sites

B_4 must scale among multiple dimensions. Targeting data center deployments meant that the total number of WAN sites would be a few dozen.

End application control

Control of both the applications and the site networks connected to B_4 is done . Hence relative application priorities and control bursts at the network edge can be enforced rather than through overprovisioning or complex functionality in B_4.

Cost sensitivity

B_4's capacity targets and growth rate led to unsustainable cost projections.The traditional approach of provisioning WAN links to protect against failures and packet loss, combined with prevailing per-port router cost, would make the network prohibitively expensive.

Problem statement

A network architecture can be divided into control plane,data plane and management plane as shown in Fig 2.3. Data plane forwards packets by looking up flow tables. Control plane configures and updates flow tables for data plane. Management plane is responsible for monitoring and configuring network devices. In SDN, it separates control plane software and data plane hardware.

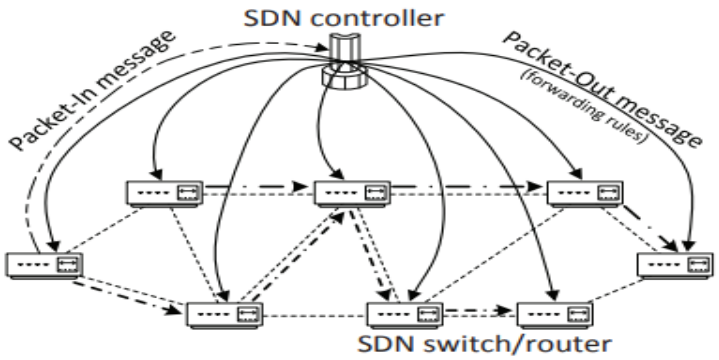


Figure 2.3: Software defined network architecture

One controller can monitor and manage multiple routers or switches. Network devices thus only need to forward packets. The controller is executed by pure software and commands data planes by protocols such as OpenFlow. In this way, new communication protocols can be more easily realized with little hardware constraint. Therefore, SDN can save hardware upgrade costs and adapt to ever increasing software needs.

A software-defined networking (SDN) architecture (or SDN architecture) defines how a networking and computing system can be built using a combination of open, software-based technologies and commodity networking hardware that separate the SDN control plane and the SDN data plane of the networking stack.

Traditionally, both the SDN control plane and data plane elements of a networking architecture were packaged in proprietary, integrated code distributed by one or a combination of proprietary vendors. The OpenFlow standard, created in 2008, was recognized as the first SDN architecture that defined how the control and data plane elements would be separated and communicate with each other using the OpenFlow protocol. The Open Network foundation is the body in charge of managing OpenFlow standards, which are open source. However, there are other standards and open-source organizations with SDN resources, so OpenFlow is not the only protocol that makes up SDN.

In the SDN architecture, the splitting of the control and data forwarding functions is referred to as disaggregation , because these pieces can be sourced separately, rather than deployed as one integrated system. This architecture gives the applications more information about the state of the entire network from the controller, as opposed to traditional networks where the network is application aware. SDN Applications are programs that communicate behaviors and needed resources with the SDN Controller via application programming interface (APIs). In addition, the applications can build an abstracted view of the network by collecting information from the controller for decision-making purposes. These applications could include networking management, analytics, or business applications used to run large data centers. For example, an analytics application might be built to recognize suspicious network activity for security purposes.

The SDN Controller is a logical entity that receives instructions or requirements from the SDN Application layer and relays them to the networking components. The controller also extracts information about the network from the hardware devices and communicates back to the SDN Applications with an abstract view of the network, including statistics and events about what is happening.The SDN networking devices control the forwarding and data processing capabilities for the network. This includes forwarding and processing of the data path.

The SDN architecture APIs are often referred to as northbound and southbound interfaces, defining the communication between the applications, controllers, and networking systems. A Northbound interface is defined as the connection between the controller and applications, whereas the Southbound interface is the connection between the controller and the physical networking hardware. Because SDN is a virtualized architecture, these elements do not have to be physically located in the same place.

2.3 Efficient and consistent flow update

Consider a flow update example where flow f1 = A $->$ B $->$ E needs to be migrated to flow $f1^{'}$ = A $->$ C $->$ D $->$ E (Fig 2.4) . Control plane first sends control messages carrying update rules to switches on target paths. During updates, packets are still being transmitted in the network.It is critical to maintain the efficiency and consistency properties.

- Blackhole Problem
- Loop Problem
- Congestion Problem
- Deadlock Problem

Blackhole problem

The existence of blackholes leads to dropping packets at certain nodes, thus decreasing network throughput.The network needs to make changes such as deleting the old rules from switch A and switch B and adding the new rules to switch A, switch C and switch D. When only partial updates are done, blackholes may appear.

Loop problem

To prevent blackholes add new rules first and then delete old rules.This might form a loop during the transition. If switch A and switch C are updated, but switch E still uses the old rule. A loop C $->$ B $->$ E $->$ C appears.

Congestion problem

During updates, no link should experience tentative bandwidth overflow. If there are two flows f1 and f2 with rates 0.7 and 0.8 Mbits/s. Assume link capacity to be 1 Mbits/s .If we update f1 first, the load of link will tentatively become 0.7+ 0.8 $>$ 1 Mbits/s during transition.

Deadlock problem

If flow f1 and f2 need to be switched. In this case, each flow waits for the other flow to be removed before it can be migrated. Therefore, none of them can be updated causing deadlock.

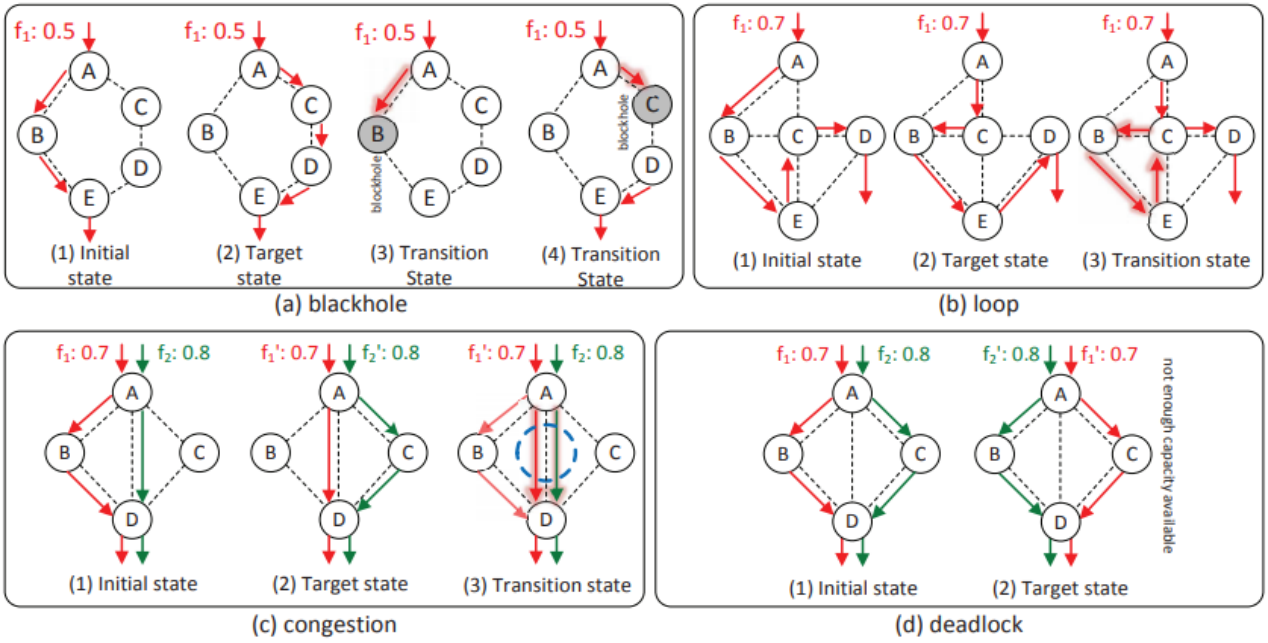


Figure 2.4: Blackhole,loop,congestion and deadlock problems

2.4 Problem formulation

Consider a SDN with N switches s_i , 1 $\leq$ i $\leq$ N. Switches are connected by a set of links and the link between s_j and s_k is denoted by $l_{j,k}$. The link capacity of $l_{j,k}$ is denoted by $C_{j,k}$. In the network there are a set F of flow pairs. Each flow pair (f_i ,$f_i^{'}$) F implies the need of migrating the transmission flow f_i to a new path $f_i^{'}$. A flow f_i is written as a sequence s_{i1} $->$ s_{i2} $->$ s_{i3} $->$ ..., such that $l_{ij,i\,j+1}$ is a link. The required transmission bandwidth of both f_i and $f_i^{'}$ is b_i (Mbits/s). During flow update all the four consistency properties are maintained and schedule an updating order Rˆ = R1 $->$ R2 $->$ ..., where Ri refers to the i^{th} update round. Each round Ri consists of a set of segment pairs denoted by (g1, $g1^{'}$),(g2, $g2^{'}$), ... where each segment pair (gi, $gi^{'}$) is a partial flow pair in F and represents migrating gj to $gj^{'}$.

Partition a flow into segments and all segment pairs in R_i can be updated in parallel. The goal is to find an update order Rˆ for updating F with the minimal update time T (Rˆ) while guaranteeing consistency during the whole updating process.

Proposed method

Updating flow f to $f^{'}$ includes adding new rules to $f^{'}$ and deleting old rules from f. However, updating rules on these switches in an arbitrary order may cause problems. To prevent blackholes and loops a reverse order update scheme is used. The controller adds new rules with reverse order and then deletes old rules with forward order. As an example, rules should be added to $f^{'}$ with the order $s_D -> s_E -> s_B -> s_C -> s_A$ and be deleted from f with the order $s_A -> s_B -> s_E -> s_C -> s_D$. For switch s_E, once its new rule is added the packets transmitted to it via f1 will be forwarded to s_D by the new rule. Thus, the loop problem will not appear. On the other hand, the blackhole problem is prevented since we add rule at s_D before s_C .

Proposed scheme mainly contains four functions. Function ParFlow() tries to partition flow pairs into segment pairs. We then generate a dependency graph from these segment pairs by function GenDepGraph(). Function GenRnd() is to schedule a sequence of non-deadlock rounds from G^. If there is no deadlock in the function, it is expected to handled all flow updates. Otherwise, RsvDead() is called to resolve one deadlock. Then this is repeated until G^ becomes null.

- Segmenting Flow Pairs
- Generating Dependency Graph
- Generating Non-deadlock Rounds
- Resolving Deadlock

Proposed algorithm works as follows:

1. S = ϕ
2. // Partition flows into segments
3. for each (f, $f^{'}$) F do
4. S = S $\bigcup$ ParFlow(f, $f^{'}$)
5. end for
6. // Dependency graphs generation
7. G^ = GenDepGraph(S)
8. while G^ $\neq$ Null do

9. // Generate non-deadlock rounds
10. GenRnd(Gˆ)
11. // Resolve deadlock
12. if Gˆ $\neq$ Null do
13. RsvDead(Gˆ)
14. end if
15. end while

3.1 Segmenting flow pairs

To increase update parallelism, function ParFlow(f,$f^{'}$)tries to partition a flow pair (f, $f^{'}$) into segment pairs (g_1, $g_1^{'}$),(g_2, $g_2^{'}$), ... such that g_j and $g_j^{'}$ are partial paths of f and $f^{'}$ respectively, and g_j and $g_j^{'}$ share the same start and end nodes.

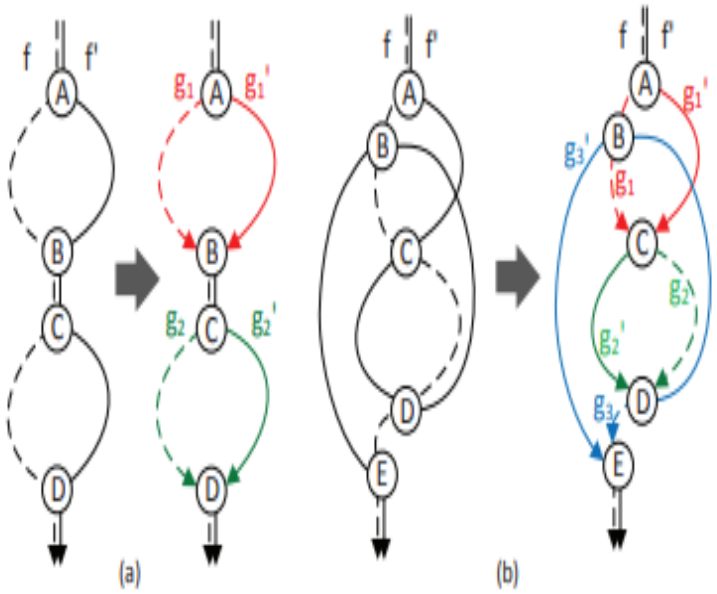


Figure 3.1: Examples of partitioning flow pairs into segment pairs

Consider an example shown in Fig 3.1, where (f, $f^{'}$) is divided into (g_1, $g_1^{'}$),(g_2, $g_2^{'}$). Both g_1 and $g_1^{'}$ start at A and end at B and both g_2 and $g_2^{'}$ start at C and end at D. Second case shows a more complicated example where (f, $f^{'}$) is divided into (g_1, $g_1^{'}$),(g_2, $g_2^{'}$),(g_3, $g_3^{'}$).Except start and end nodes,a segment pair never overlaps.

The partioning takes place as follows.Set M is initially empty and will contain final segment pairs at the end. Node S is an index to $f^{'}$ that helps us to traverse $f^{'}$. Function Start($f^{'}$) returns the first node of $f^{'}$, while End($f^{'}$) returns the last node of $f^{'}$.

In the while-loop, we traverse $f^{'}$ in the forward direction using index s. If s tail has different routing rules for f and $f^{'}$, a new segment pair (g, $g^{'}$) starting from s and ending at tail is generated. Here, s is a splitting node that f and $f^{'}$ go into different directions and Find Tail(s) returns the node next to s in $f^{'}$ that also appears in the rest part of f. Then we move index s to tail. Otherwise, we move index s to Next(s), the node immediately next to s in $f^{'}$. At the end, M contains all these generated segment pairs and serves as the returned value of this function.

Proposed algorithm works as follows:

1. M= ϕ
2. s=start($f^{'}$)
3. while s $\neq$ end($f^{'}$)
4. if s uses a different rule in f and ($f^{'}$)
5. tail = F indTail(s)
6. g = sub-flow of f from s to tail
7. $g^{'}$=sub-flow of $f^{'}$ from s to tail
8. M = M $\bigcup$ (g, $g^{'}$)
9. s=tail
10. else
11. s = Next(s)
12. end if
13. end while
14. return (M)

Consider the example f = A $->$ B $->$ C $->$ D $->$ E and $f^{'}$ = A $->$ C $->$ D $->$ B $->$ E. Since the search is based on node order in $f^{'}$, from A, the next common node of f and $f^{'}$ is C, so the first segment pair is (g_1,$g_1^{'}$) = (A $->$ B $->$ C, A $->$ C). From C, the next pair is (g_2,$g_2^{'}$) = (C $->$ D, C $->$ D). From D, the next pair is (g_3,$g_3^{'}$) = (D $->$ E, D $->$ B $->$ E). So totally three pairs are returned.

This algorithm only partitions one flow pair at a time . After each call to ParFlow(), the returned set is unioned with set S. Finally, S will include all generated segment pairs.

3.2 Generating dependency graph

To derive the dependency relations among segment pairs of S. This helps prevent potential congestions and deadlocks. We will write l $->$ (g, $g^{'}$) if the migration of g to $g^{'}$ depends on the availability of bandwidth on link l. We will write (g, $g^{'}$) $->$ l if the migration of g to $g^{'}$ will release some bandwidth to link l.

Our goal is to first generate a dependency graph G and then extend it to an augmented graph Gˆ. From each segment pair (g,$g^{'}$) S, we construct m + n + 1 nodes, where m and n are the numbers of links in g and $g^{'}$, respectively. The (g, $g^{'}$) itself generates one node, called segment node, also denoted by (g, $g^{'}$) for convenience. Each link of g and $g^{'}$ generates one node, called link node, also denoted by the link's label. This gives m + n link nodes. From each link node l of $g^{'}$, we generate a directed edge l (g, $g^{'}$). For each link node of g, we generate a directed edge (g, $g^{'}$) l.

Consider an example with six segment pairs as shown in Fig 3.2.Derive subgraphs for each segment pairs.After having all subgraphs, we can combine them into a dependency graph by merging all link nodes belonging to the same link into one node.Next, we augment G to Gˆ by adding information to each link and segment node. A link node $l_{j,k}$ has one extra element.

- AvlBw: available bandwidth, which is the total bandwidth of $l_{j,k}$ subtracted by the sum of bandwidths of all flows passing it.

A segment node (g, $g^{'}$) has two extra elements

- TxBw : transmission bandwidth bi of g and $g^{'}$.
- PotUpTm : Potential update time, which is the longest possible time to update all nodes depending on (g,$g^{'}$).

To calculate PotUpTm for Gˆ. This element reflects the accumulated update time of a dependency sequence. From each segment node r, we construct a breadth first spanning tree in G with r as root. Then a tentative value tm(x) is computed for each segment/link node x in a bottom up manner on the tree. Intuitively, a segment node will add its update time on itself, while a link node will not. For a leaf node y, if it has no outgoing dependency, its tm(y) = 0; otherwise, its tm(y) ='L', which means "Large value". The label 'L' will remain there when we go up recursively. Finally, PotUpTm(r) is set to tm(r).

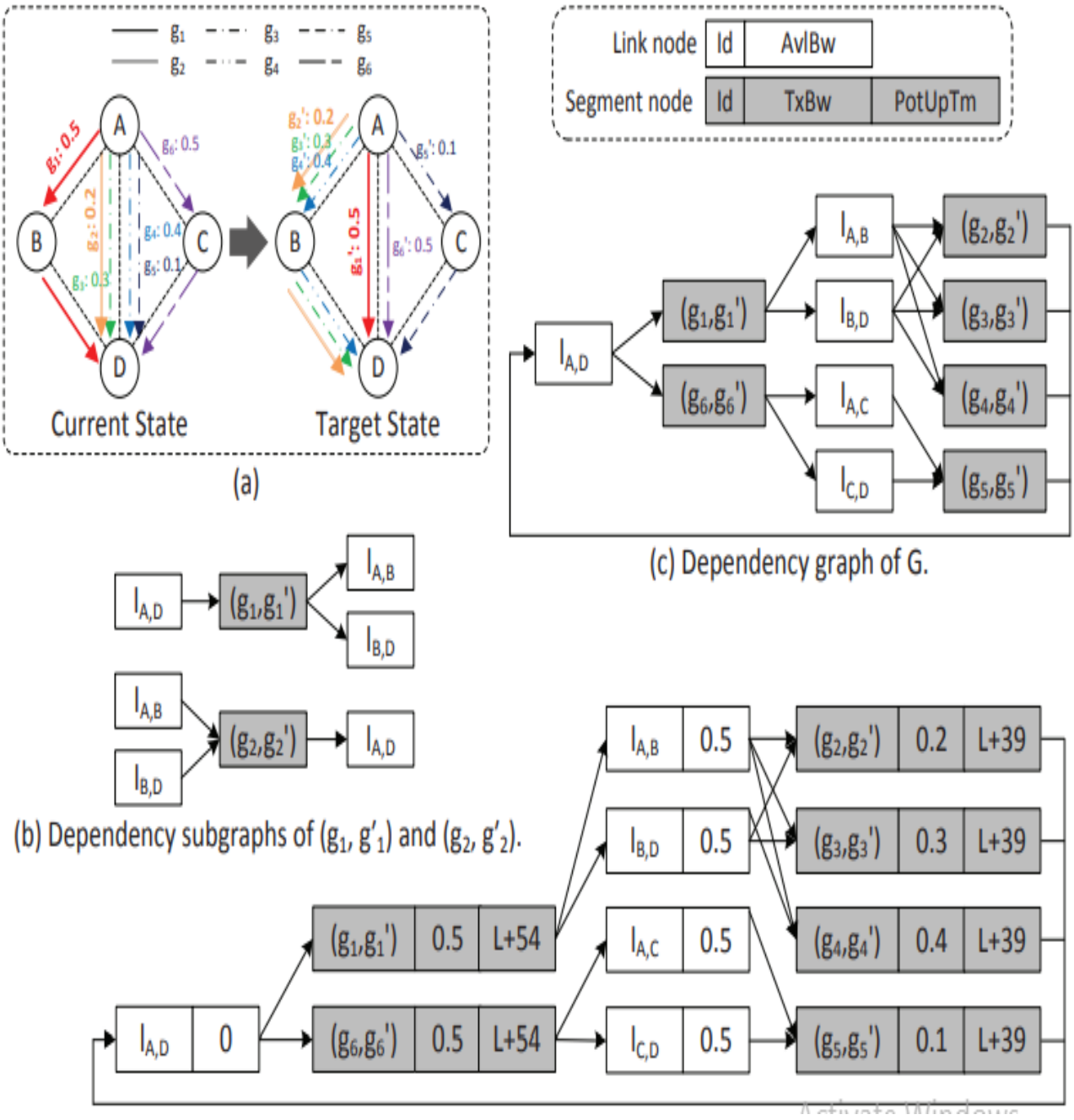


Figure 3.2: Dependency graphs G and Gˆ

3.3 Generating non-deadlock rounds

Now, Gˆ contains all dependency information. Algorithm GenRnd is to schedule flow updates round by round until finishing all updates or encountering unsolvable deadlocks. The process works as follows.

- The while loop will potentially generate a round after each iteration.
- This is to initialize a new empty round R. Segment nodes are sorted by their PotUpTm in a descending order, and then by their TxBw in a descending order.
- If all links on $g^{'}$ have enough bandwidths for updating (g, $g^{'}$), we update their AvlBw and append (g, $g^{'}$) to set R.
- If the set R is not empty, we update the segment pair (g, $g^{'}$) and release bandwidth to link l.
- After updating, the segment pair (g, $g^{'}$) is removed in the set R.

Consider the example of previous case. The sorted list is: (g_1, $g_1^{'}$),(g_6, $g_6^{'}$),(g_4, $g_4^{'}$),(g_3, $g_3^{'}$),(g_2, $g_2^{'}$),(g_5, $g_5^{'}$).To find the first round, we check the sorted segment pairs one by one. Only (g_4, $g_4^{'}$) and (g_5, $g_5^{'}$) can acquire enough bandwidths to update. Node (g_4, $g_4^{'}$) takes 0.4 bandwidth from L_{AB} and L_{BD} and then node (g_5, $g_5^{'}$) takes 0.1 bandwidth from L_{CD} and L_{AC}. So R = (g_4,$g_4^{'}$) ,(g_5, $g_5^{'}$). After taking out bandwidth and after finishing update PotUpTm needs to be recalculated for all segment pairs. PotUpTm of (g_1, $g_1^{'}$) is reduced to L+27 and that of (g_6, $g_6^{'}$) is reduced to 15.

In this example as shown in Figure 3.3, all dependencies can be resolved, and the final sequence of round is S = R1 $->$ R2 $->$ R3 $->$ R4 = (g_4,$g_4^{'}$) ,(g_5, $g_5^{'}$) $->$ (g_1, $g_1^{'}$) $->$ (g_3, $g_3^{'}$),(g_2, $g_2^{'}$) $->$ (g_6, $g_6^{'}$) .

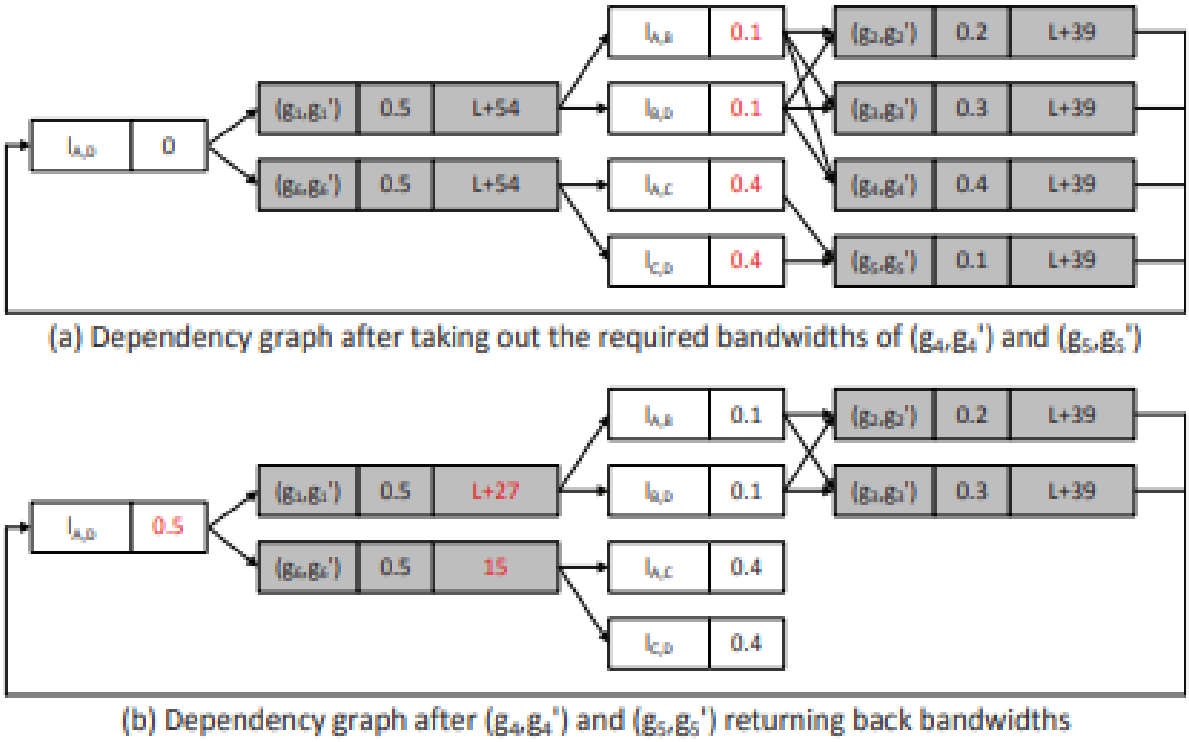


Figure 3.3: Change of dependency graph during a round

The proposed algorithm works as follows:

1. while True do
2. R= ϕ
3. Sort the segment nodes by P otUpTm and TxBw
4. for each (g, $g^{'}$) in the sorted list do
5. if L $->$(g, $g^{'}$) : AvlBw(l) $\geq$T xBw(g, $g^{'}$)then
6. for each l $->$(g, $g^{'}$) do
7. AvlBw(l) = AvlBw(l) T xBw(g, $g^{'}$)
8. end for
9. R= R$\bigcup$ (g, $g^{'}$)
10. end if
11. end for
12. if R= ϕ then
13. break while

14. else
15. Conduct flow update in R
16. for (g, $g^{'}$)$- >$l(g, $g^{'}$)ϵ R do
17. AvlBw(l) = AvlBw(l) + T xBw(g, $g^{'}$)
18. end for
19. Remove all (g, $g^{'}$)ϵ R from G$\hat{}$
20. if G$\hat{}$ has no segment node, then break while
21. end if
22. end if
23. end while

3.4 Resolving deadlock

After the above steps, if there is any segment pair remaining in G$\hat{}$, function RsvDead is called. It is impossible to resolve a deadlock unless some party in the loop is willing to yield. Therefore, we will identify a pair and reduce its bandwidth requirement so as to migrate it. The first step is to calculate yield ratio . This ratio refers to the percentage of throughput loss during updating. We can restore its bandwidth after the whole migration is done . We favor the one with the lowest yield ratio and update its TxBw. Then the rest of these steps are to conduct the update of (g, $g^{'}$). and reflect the update in G$\hat{}$. Calling this function contributes one round consisting of only one segment pair. Then we will loop back .

The algorithm works as follows:

1. for each (g, $g^{'}$) G$\hat{}$
2. calculate yr(g, $g^{'}$)
3. end for
4. select (g, $g^{'}$) with the minimal yr(g, $g^{'}$)
5. T xBw(g, $g^{'}$)=T xBw(g, $g^{'}$) $\times$ (1 yr(g, $g^{'}$))
6. for each l $- >$(g, $g^{'}$)
7. AvlBw(l) = AvlBw(l) T xBw(g, $g^{'}$)
8. end for
9. Conduct update for (g, $g^{'}$)

10. for (g, $g^{'}$)$- >$l
11. AvlBw(l) = AvlBw(l) + T xBw(g, $g^{'}$)
12. end for
13. R= R (g, $g^{'}$) and remove (g, $g^{'}$) from Gˆ

3.5 Analysis of time complexity

In the proposed scheme, it works in four phases. The first phase costs O(M N) to partition $|F|$ flow pairs into shorter routing segments to increase update parallelism. Here, a flow can be partitioned into at most O(N) shorter segments.

The second phase generates a global dependency graph of the segments to be updated. Thus, it costs O(M $\times$ N) because there are M flow pairs and each flow pair has at most $(N-1)$ segments which connect to/from at most $(N-1)$ links. Then, it recursively calculates the potential update time for all segments and links. Since there are at most $(N-1)$ segment nodes and $(N-1)$ link nodes to be updated, it costs O(N) to traverse all nodes and links in the spanning tree. Thus, the second phase costs totally $O(M \times N) + O(N) = O(M \times N)$.

The third phase conducts actual updates and then adjusts dependency graphs accordingly. To generate the non-deadlock schedules, it first costs O(MNlogMN) to sort at most MN segment nodes based on the potential update time and transmission bandwidth. Then, these segments cost $O(MN \times N)$ to check the available bandwidth of all the links belonging to them and costs $O(MN \times N)$ to conduct flow update accordingly. Thus, the third phase costs $O(MNlogMN) + O(MN \times N) + O(MN \times N) = O(MNlogMN + MN^2)$.

The last phase deals with deadlocks, if any, and then loops back to phase three if necessary. To resolve deadlocks, it first costs O(MN) to calculate the yield ratio for all deadlock segments in Gˆ. Then, it costs O(MN) to select the segment with the minimal value of yield ratio. Next, it costs O(1) to calculate the new transmission bandwidth. It costs O(N) to update the available bandwidth of all links in this selected segment after applying the new transmission bandwidth. Thus, the fourth phase totally costs O(MN) + O(MN) + O(1) + O(N) = O(MN).

Since the third phase schedules at least one segment in each round and the last phase resolves at least one deadlock in each loop, they need to take at most K = $M \times N$ rounds. Therefore, the proposed scheme totally costs $O(M \times N) + O(M \times N) + K \times (O(MNlogMN + MN^2) + O(MN)) = O(KMNlogMN + KMN^2)$. Thus, the complexity will become O(MNlogMN + MN^2).

Performance evaluation and comparison

In this section, we develop a simulator by Python language and implement our scheme to validate its performance. The simulator includes two types of network topology: mesh network and data center network .

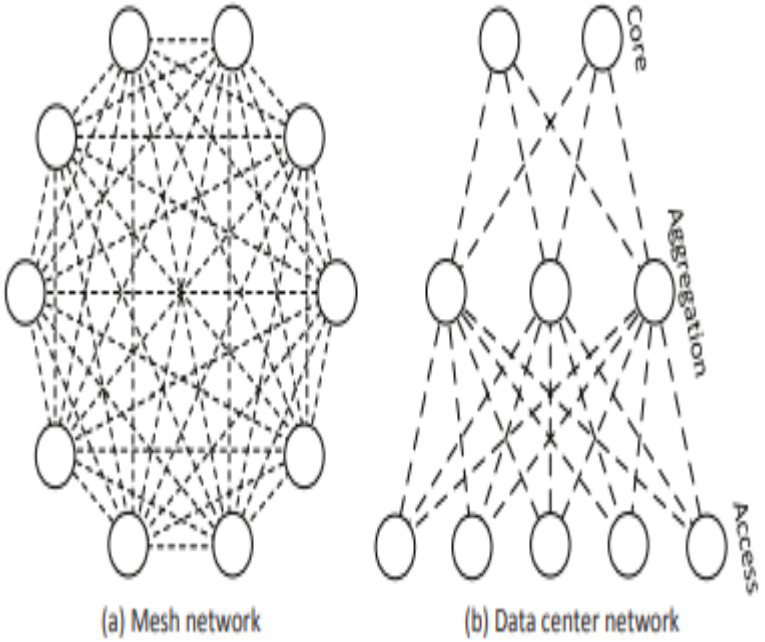


Figure 3.4: Network topology

. In these two topologies as given in Fig 3.4, there are 100 switches with the link bandwidth of 1(Gbps), where the switches are full duplex and the traffic flows can be transmitted to or from two connecting switches. In addition, the transmission pairs and routes are generated randomly . Note that the data center network contains three layers: core layer (12 switches), aggregation layer (22 switches), and access layer (66 switches). Each data flow can be transmitted through: (1) access layer $->$ aggregation layer $->$ core layer or (2) core layer $->$ aggregation layer $->$ access layer, accordingly.

In the simulation, we compare our scheme against Cupid scheme.To simulate a real SDN environment, we add several real parameters into our simulation, such as the latency of real switches. We adopt the measured latencies of off-the shelf SDN switches , including the time needed for adding a rule to a switch and deleting a rule in rule table. In addition, according to Open vSwitch which can support a maximum of 255 flows in a switch, is adopted. Thus, the number of flows —F— in our simulation is = 1000, 2000, 5000, 10000, 15000 and 25500 flows.

The performance metrics include:

- update completion efficiency
- total update time
- throughput loss

Light Network Load

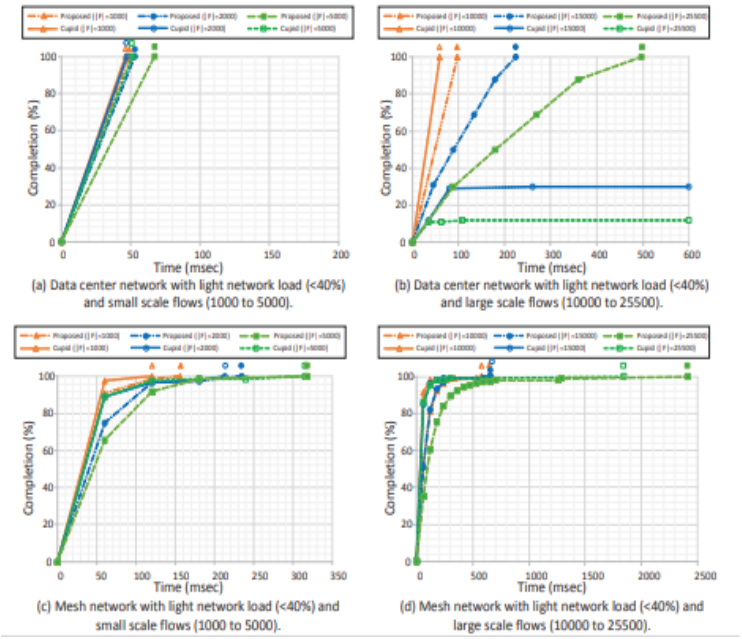


Figure 3.5: Light network loads

First, we investigate the number of flows on update completion efficiency when the network load is light as shown in Fig 3.5. We can see that most schemes achieve 100 percent update completion in both mesh and data center networks except for Cupid . The update efficiency of Cupid in small scale flows is slightly better than proposed ,because it does not check all links' capacities for flow pairs. Thus, Cupid incurs more throughput loss when the topology is complicated, such as the mesh network. Since Cupid considers only critical nodes and local dependency graph to perform update, the flows passing through multiple hops encountering congestion cannot be detected. Therefore, Cupid needs to spend more time on resolving deadlocks especially in large-scale flows, e.g., —F— = 15000 25500 . Contrarily, proposed scheme can detect congestion problems and reduce throughput loss significantly during flow updating, thus taking extra update time as in table 3.1.

			(Proposed / Cupid)	(Proposed / Cupid)			(Proposed / Cupid)	(Proposed / Cupid)
Networkload	Topology	# of Flow	Total update time(msec)	Throughput losses	Topology	# of Flow	Total update time(msec)	Throughput losses
Light	Mesh	1000	156 / 120	0 / 0	Data center	1000	49 / 46	0 / 0
		2000	234 / 213	0 / 0		2000	53 / 47	0 / 0
		5000	318 / 315	4.54E-5 / 7.98E-5		5000	67 / 51	0 / 0
		10000	642 / 579	7.75E-5 / 1.65E-4		10000	98 / 59	0 / 0
		15000	654 / 657	1.15E-4 / 3.48E-2		15000	225 / 343,449	0 / 0
		25500	2,412 / 1,842	1.85E-3 / 1.18E-1		25500	498 / 631,491	0 / 0

Table 3.1: Total update time and throughput loss of all schemes in Light network load

Medium network load

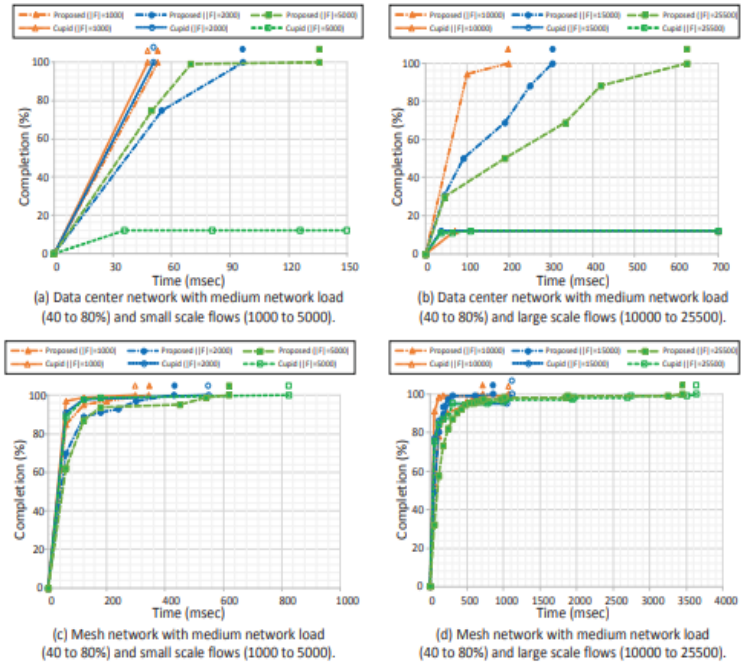


Figure 3.6: Medium Network Loads

Next, when the network load increases to medium level as in Fig 3.6, the congestion problem is more likely to happen during flow updating. We can see that proposed scheme outperforms Cupid when the number of flows increases to 10000, 15000 and 25500 . The reason is that more flows result in more deadlocks. Since Cupid addresses deadlocks flow by flow, it takes longer time to resolve deadlocks. Contrarily, proposed scheme can find all possible schedules and update flows concurrently to solve deadlocks. Thus, it can complete in a short time when the network load increases as shown in table 3.2.

			(Proposed / Cupid)	(Proposed / Cupid)			(Proposed / Cupid)	(Proposed / Cupid)
Networkload	Topology	# of Flow	Total update time(msec)	Throughput losses	Topology	# of Flow	Total update time(msec)	Throughput losses
Medium	Mesh	1000	345 / 297	2.65E-3/ 1.16E-2	Data center	1000	53 / 48	0 / 0
		2000	432 / 549	9.33E-3 / 2.35E-2		2000	97 / 51	0 / 0
		5000	620 / 825	1.81E-2 / 3.41E-2		5000	136 / 21,502	0 / 0
		10000	711 / 1071	4.81E-2 / 5.82E-2		10000	198 / 44,339	0 / 0
		15000	854 / 1,112	2.24E-2 / 6.09E-2		15000	304 / 455,103	0 / 0
		25500	3,445 / 3,631	1.65E-2 / 1.55E-1		25000	625 / 2,035,530	0 / 0

Table 3.2: Total update time and throughput loss of all schemes in medium network load

Heavy network load

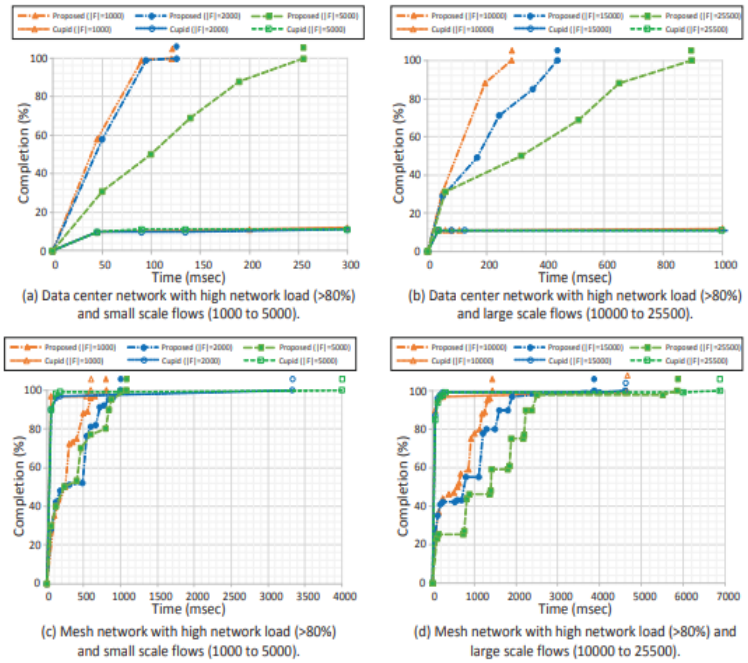


Figure 3.7: Heavy Network Loads

Finally, when the network is under heavy load, as shown in Fig 3.7 there are potentially more congestions and deadlocks. Thus, all schemes take more time to update. We can see that proposed scheme still outperforms Cupid in terms of total update time and throughput loss when the number of flows is larger than 1000. Since more flows in a data center network need to pass through access layer, aggregation layer, and then core layer, the bottleneck between core layer and aggregation layer should be addressed first. Cupid needs more time to handle its deadlock procedure and may not complete in short time as in table 3.3.

			(Proposed / Cupid)	(Proposed / Cupid)			(Proposed / Cupid)	(Proposed / Cupid)
Networkload	Topology	# of Flow	Total update time(msec)	Throughput losses	Topology	# of Flow	Total update time(msec)	Throughput losses
High	Mesh	1000	804 / 600	1.42E-4 / 2.27E-1	Data center	1000	121 / 13,005	0 / 0
		2000	1005 / 3333	1.56E-4 / 2.08E-1		2000	126 / 28,944	0 / 0
		5000	1,080 / 3,999	2.03E-4 / 2.42E-1		5000	255 / 147,978	0 / 0
		10000	1,428 / 4,656	3.05E-4 / 2.52E-1		10000	285 / 301,167	0 / 0
		15000	3,876 / 4,615	3.58E-4 / 3.62E-1		15000	439 / 891,081	0 / 0
		25500	5,865 / 6,876	2.8E-3 / 5.44E-1		25000	897 / 3,564,323	0 / 0

Table 3.3: Total update time and throughput loss of all schemes in heavy network load

Conclusion

This paper have addressed the efficiency and consistency issues for software defined networks in terms of Blackhole problem, Loop problem, and Deadlock problem when migrating flows. We have proposed an efficient flow update scheme which considers both efficiency and consistency by four phases. First, it partitions flows into shorter routing segments to increase update parallelism. Second, it generates a global dependency graph for network status maintenance. Third, it conducts actual updates and adjusted dependency graphs. Finally, it deals with deadlocks when loops exists.

Through simulations it have been verified that proposed scheme can not only ensure freedom of blackholes, loops, congestions, and deadlocks during flow updates, but also run faster than existing schemes. For future directions, it is possible to use SDN software, such as Open vSwitch to build a prototype. Proposed simulation is at flow level and it deserves to study packet-level simulations.

References

[1] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu et al., "B4: Experience with a Globally-deployed Software Defined WAN," in Proc. ACM SIGCOMM, vol. 43, no. 4, pp. 3–14, 2013.

[2] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic Engineering in Software Defined Networks," in Proc. IEEE INFOCOM, pp. 2211– 2219, 2013.

[3] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling Fast Failure Recovery in OpenFlow Networks," in Proc. Design of Reliable Communication Networks (DRCN), pp. 164–171, 2011.

[4] J. Liu, J. Li, G. Shou, Y. Hu, Z. Guo, and W. Dai, "SDN based Load Balancing Mechanism for Elephant Flow in Data Center Networks," in Proc. IEEE Int'l Symposium on Wireless Personal Multimedia Communications.

[5] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," in Proc. ACM SIGCOMM, pp. 539–550, 2014.

[6] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free Consistent Data Plane Update in Software Defined Networks," in Proc. IEEE INFOCOM, pp. 1–9, 2016.

[7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-driven WAN," in Proc. ACM SIGCOMM, pp. 15–26, 2013.

[8]] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pp. 323–334, 2012.