Diagnosis of Faults in Data Centre Networks and Network Visualization using Data Plane Programmability and Relational Queries

THESIS

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Sankalp Sanjay Sangle
ID No. 2016A7TS0110P

Under the supervision of:

Dr. Mun Choon CHAN
&
Dr. Virendra S SHEKHAWAT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
June 2020

Declaration of Authorship

I, Sankalp Sanjay Sangle, declare that this Thesis titled, 'Diagnosis of Faults in Data Centre Networks and Network Visualization using Data Plane Programmability and Relational Queries' and the work presented in it are my own. I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Certificate

This is to certify that the thesis entitled, "*Diagnosis of Faults in Data Centre Networks and Network Visualization using Data Plane Programmability and Relational Queries*" and submitted by Sankalp Sanjay Sangle ID No. 2016A7TS0110P in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor
Dr. Mun Choon Chan
Associate Professor,
School of Computing, National University of Singapore
Date:

Co-Supervisor
Dr. Virendra S Shekhawat
Assistant Professor,
BITS Pilani, Pilani Campus
Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Bachelor of Engineering (Hons.) Computer Science

Diagnosis of Faults in Data Centre Networks and Network Visualization using Data Plane Programmability and Relational Queries

by Sankalp Sanjay Sangle

Data Centers have emerged as the basic unit for providing web services to customers by major technology companies. They house servers in the thousands and tens of thousands, with servers often serving as backups to ensure reliability. The reliability of data center network infrastructure is critically important for ensuring seamless user experience. The nature and distribution of faults in data center networks isn't completely understood, and many faults like microbursts due to fan-in traffic go undetected. A closer look is needed at the type of faults that occur in data center networks and on how they can be diagnosed. Modern data centres operate at throughputs of close to a few hundreds of Gbps, implying packets coming in at a few hundreds of nanoseconds apart. Most monitoring solutions offer resolution of the order of milliseconds, and hence are inadequate for recording and detection of events that last for sub millisecond intervals. This thesis aims to discuss techniques for diagnosing faults in data centers, develop a monitoring system to view them, at nanosecond raesolution. It is also demonstrated that appropriate calculations can be performed on the SQL queries to extract quantities like ingress throughputs, egress throughputs, relative ratios of packets in a network, (all at nanosecond resolution) thereby demonstrating that collecting relational data about switches is a simple and yet powerful way to extract meaningful data and create relevant visualizations[4] which can help a network administrator determine the root cause of faults in the network and address them.

Acknowledgements

This thesis has been written in a period when the world is at a standstill due to the threat of COVID-19. As such, it has been a difficult but deeply enriching experience in Singapore, and it would be ungrateful of me to not give people their due acknowledgements.

I am grateful to my supervisor, Dr. Mun Choon Chan, and co-supervisor Dr. Virendra S Shekhawat for their guidance and availability during the period of my thesis. I am also thankful for the guidance provided by Ph.D. candidates Pravein Govindan Kannan and Nishant Budhdev, and for their gracious nature and helpfulness in addressing all my doubts and making me feel comfortable in a foreign place. The numerous Zoom calls with Professor Chan, Pravein and Nishant were a beautiful learning experience for me. I am indebted to my roommate turned friend Yashdeep Thorat for having been a source of comfort, motivation, and stability when things turned difficult to bear. I am grateful to my family for their constant support, for being able to fall back on them, and for always being a phone call away.

I am thankful to Singapore for having proven to be a place where I felt comfortable and at ease right from the first day. The culture and warmth of Singaporeans was a delight to observe and experience and I hope to one day, when it is safe again, return to Singapore and relive some of my experiences.

Finally, I might not have taken the leap and tried to apply for such an opportunity had it not been for the timely intervention and push by Miloni, and for that I am indebted to her...

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Thesis Topic

Data Centers have emerged as the basic unit for providing web services to customers by major technology companies. They house servers in the thousands and tens of thousands, with servers often serving as backups to ensure reliability. Traffic in data centers is often organized in a tree like structure, with top of rack switches, aggregate switches and core switches. The tree like structure is easy to maintain and scale to meet the growing number of clients.
The reliability of data center network infrastructure is critically important for ensuring seamless user experience. Companies like Facebook[8] and Google[5] have spent considerable investments into developing frameworks for automating fault detection and repair of networks. Even so, the nature and distribution of faults in data center networks isn't completely understood, and many faults like microbursts due to fan-in traffic go undetected. A closer look is needed at the type of faults that occur in data center networks and on how they can be diagnosed.
Modern data centres operate at throughputs of close to a few hundreds of Gbps, implying packets coming in at a few hundreds of nanoseconds apart. Most monitoring solutions offer resolution of the order of milliseconds, and hence are inadequate for recording and detection of events that last for sub millisecond intervals. This motivates the need for having a diagnosis system to distinguish events at nanosecond level by leveraging advances in data plane programming[6].
This thesis aims to discuss techniques for diagnosing faults in data centers, develop a monitoring system to view them, at nanosecond resolution. It is also demonstrated that appropriate calculations can be performed on the SQL queries to extract quantities like ingress throughputs, egress throughputs, relative ratios of packets in a network, (all at nanosecond resolution) thereby demonstrating that collecting relational data about switches is a simple and yet powerful way to extract meaningful data and create relevant visualizations[4] which can help a network administrator determine the root cause of faults in the network and address them.

A high level architecture diagram of the proposition is shown in Figure 1.1

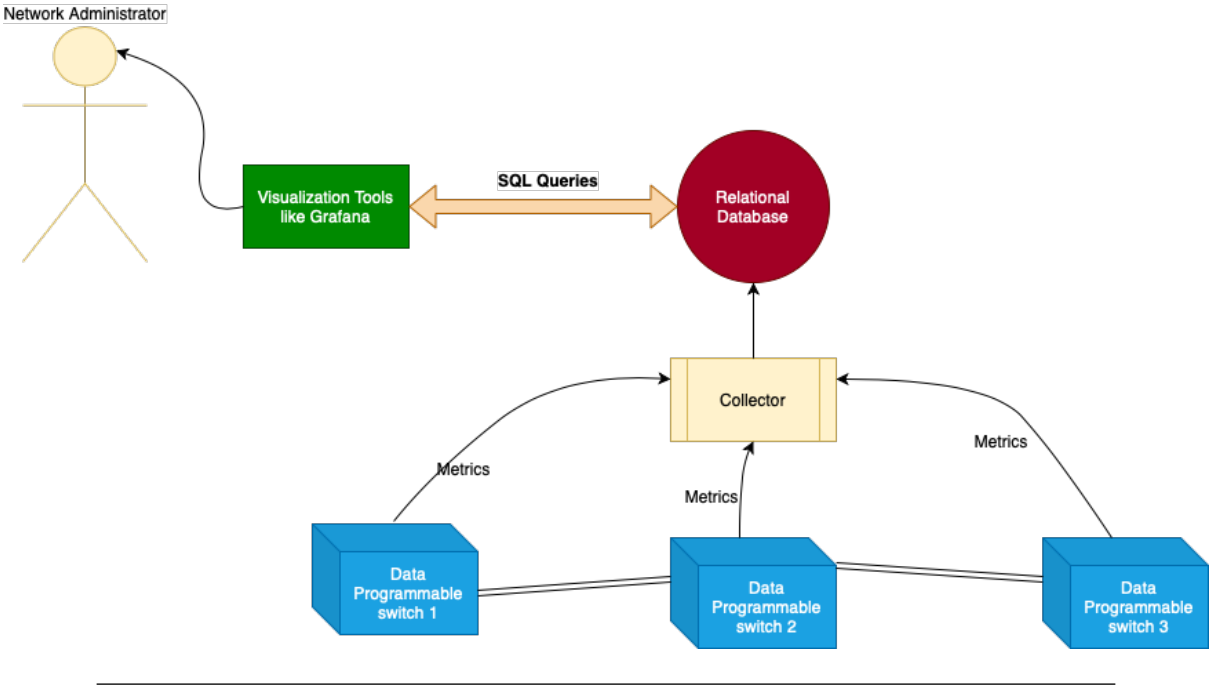

FIGURE 1.1: Architecture Diagram of proposed system

Chapter 2

Background Literature

Historically, computer networks have always been relatively blackboxed entities that provide no substantial facilities for configuration. The devices that are present in them, from traditional switches and routers, to other middleware like firewalls and network address translators, often execute software that is proprietary, and hence rigid by nature. These devices are configured by network administrators using highly specialised interfaces that vary from vendor to vendor. The software for communicating with these interfaces is not centralized, and network devices need to be individually configured to achieve the desired functionality. This frozen structure has hindered flexibility and increased costs of network management.
It is interesting to note that this structure was by design and not an oversight. The nascent stage Internet needed to be absolutely reliable and fault tolerant if widespread adoption was to take place. One way to minimize failures in the network was to make the network devices rigid and inflexible. However, by the turn of the century, the increase in number of end hosts and the widespread adoption of the Internet has given rise to a need for managing large flows of traffic effectively. The end host explosion has made the question of programmable networks much more pragmatic to discuss.

2.1 Why do we need programmable networks?

The ability to remotely configure behaviour of switches/routers finds use in a number of applications, some of them being:

- Dynamic Routing of heavy hitters
- Testing of new protocols/routing policies
- In-Network Computing

- Layer 4 Load Balancing
- Network Monitoring and Debugging (The focus of this thesis)

A programmable network is easier to monitor and control than traditional networks. Network devices whose forwarding policies can be customized can be made to behave like a router, a switch, a firewall, a NAT, or any other device we can conceptualize within the constraints of data plane switch programmability. This saves money and physical labour that would have been spent in the purchase and set up of highly specific-function network devices. A programmable network is, by nature, facilitative to network innovation and reduces the barrier to introduction of new protocols/policies.

2.2 SDN and the idea of separation of Control Plane and Data Plane

Software Defined Networking (SDN) is an umbrella term used to address efforts made to make networks programmable. The behaviour of traditional switches is defined in hardware, whereas 'software-defined' switches would have the programmability addressed earlier. One important concept of SDN is *the decoupling of Control and Data planes*. Control plane refers to protocols like OSPF, BGP, Multicast which govern traffic handling on a higher scale. The data plane performs packet switching based on the policies that the control plane dictates. In a general sense, the control plane is the intelligence layer, and the data plane is the manifestation of it in a standalone switch.

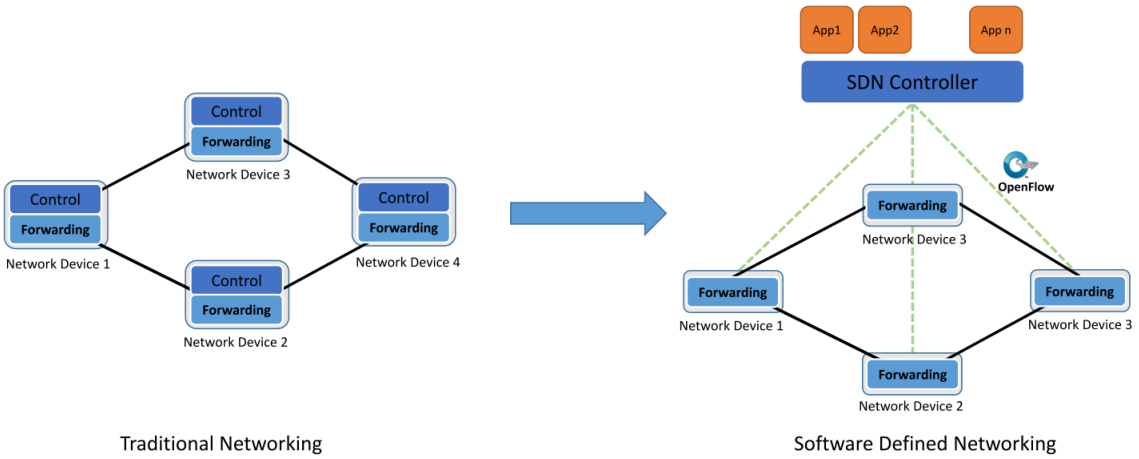

FIGURE 2.1: Separation of Control and Data planes

A traditional switch has the forwarding logic baked into the circuit, and tight integration of data

and control planes. The idea behind SDN is to decouple the planes (Figure 2.1), and have a programmable control plane that can communicate with the data plane via a standardized API (like OpenFlow[7]) so as to offer freedom in specifying how a switch handles packets rather than 'hardcoding' it into the switch.

2.3 A Brief History of Programmability in Computer Networks

2.3.1 Early approaches towards programmability (1990s - late 2000s)

The early days of SDN research revolved around what were known as Active Networks[1]. In this, two approaches were used:

- *Capsule model*, where code to be run at the nodes is carried within packets
- *Programmable router/switch model* where the code is pushed to the nodes by out-of-band means

While capsule models were feasible, there were concerns about attackers injecting malicious code into a router and compromising the network. The programmable model was much more aligned to what SDN is today. Most of the opposition towards adopting these approaches was due to a mixture of concerns over reliability and performance, and also because it was important to ensure proliferation of the Internet by keeping the network core as simple as possible.

2.3.2 OpenFlow (and its limitations)

OpenFlow started off as a research project that was implemented at Stanford University[7] by researchers in a campus wide network. As mentioned before, it defines the API for communication between a decoupled data plane and control plane. OpenFlow quickly became popular due to it being easy to adopt (most commodity switches required a firmware upgrade). The OpenFlow specification provides means to specify entries in match-action tables(called rules) where a match is made against the headers of the packet, and an action is taken(forwarding, dropping, broadcasting). Widespread adoption by companies like Google[5] boosted its popularity and many new switch vendors entered the established markets by undertaking early adoption of OpenFlow. There is a false belief that OpenFlow is the same as SDN; SDN is a general term used for addressing techniques that make networks programmable. It provides no direction or proposal on *how* to go about doing that. OpenFlow is merely one concrete way to give a level of programmability to networks.
OpenFlow however, does not offer any data plane programmability support. There is no means

to specify new protocols and fields to match against, or on how to perform reassembly of packets; consider a scenario where one needs to test a new protocol. OpenFlow enabled switches will not be able to perform match-actions on fields specified in the protocol and will have to wait until that protocol gets added into the specification (which can take of the order of months/years). This use case motivates the desire to make the data plane itself programmable.

2.4 Towards Data Plane Programmability

Data plane programming offers the flexibility of describing how the packet headers can be parsed and matched onto non-standard fields (for example, those of a new protocol) and modified, as well as the order in which the headers are reassembled before being transmitted by the switch. It is natural to think that there will be a tradeoff between throughput/speed and customizability, and this was the notion in the research community for a number of years. However, programmable switches have been made possible by recent advances networking architectures[2] in Figure 2.2. They expose such functionality in the data plane:

- Flexible parsing and deparsing on packet headers
- Ingress/Egress processing using match-action tables
- Stateful maintenance of network states using SRAMs

These switches offer throughputs of close to 1 Tbps (the rate demanded by modern data centres) even with the ability to process and modify the packet. Such speeds would be impossible if packet processing was to happen in CPU due to various overheads, which is what makes data plane programmability so attractive.

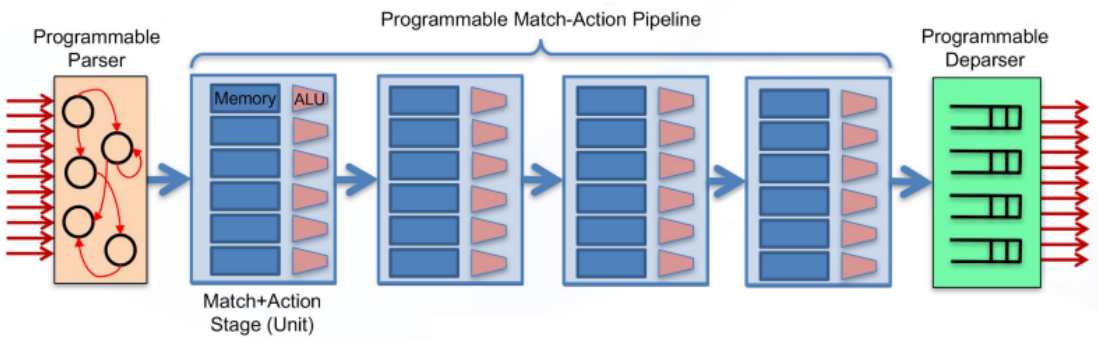

FIGURE 2.2: Generic Data Programmable Switch Architecture

2.4.1 Constraints in data plane programmable switches

In order to maintain packet processing at line-rate with no slow down in throughput, certain constraints need to be followed which impose a limit on the packet processing flexibility. Some of these are:

- No looping constructs in P4[3] (the language used to program switches)
- no floating point computations
- no exact multiply/divides (only approximated by bit-shifts)
- only one read-modify-write per stage to maintain processing at line rate

Even so, data plane programmability has found a number of applications in recent years including Network Monitoring and Debugging (the subject matter of this thesis), time synchronization, detection of elephant or heavy hitter flows[10], and In-Network Computing.

2.4.2 The P4 Programming Language

P4 (*Programming Protocol Independent Packet Processors*)[3] is a programming language developed specifically for programming data plane switches. The language is maintained by the P4 Language Consortium and is widely used for data plane programmability today. It provides constructs for specifying deterministic parsers for header parsing, action constructs for specifying steps to be taken in case of a match, and also steps to be taken in packet reassembly.

Chapter 3

Experiments

3.1 Experimental Setup

Data Center networks often use a fat-tree topology as this topology is easy to maintain and to scale. To simulate a fat-tree topology as seen in data center networks, 4 physical servers and 2 data plane programmable switches were used. Each switch was virtualized to create 5 switches using 10G loopback links. So, we have 10 switches in total, with 4 of them as top-of-rack or edge switches, 4 of them as aggregate switches, and 2 as core switches (Figure 3.1)

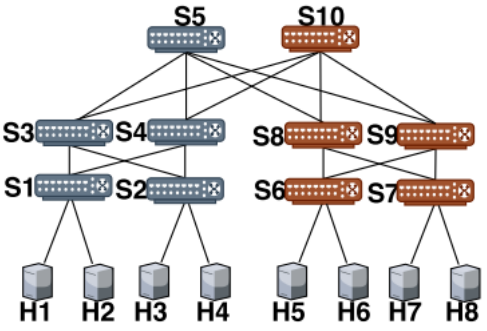

FIGURE 3.1: Evaluation Topology

3.2 SQL Database Schema

The relational schema for the packet records collected from switches is as shown in Figure 3.2

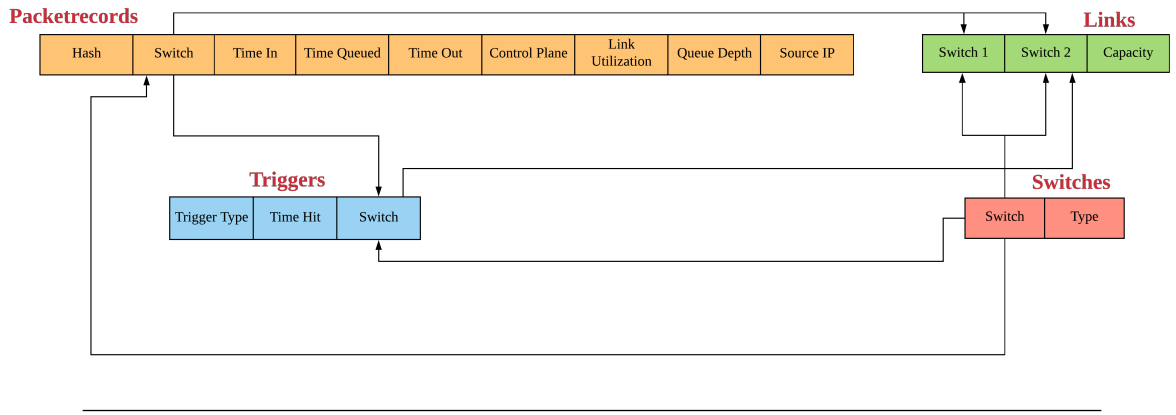

FIGURE 3.2: Database Schema

3.2.1 The Switches table

This table stores two columns described below:

- Switch - The identifier of the switch, a string ID that is used to uniquely identify each switch
- Type - This column specifies the type of the switch. For the purpose of our demonstrations, Switch type could either be a ToR switch (top of rack) or a non-ToR switch.

3.2.2 The Triggers table

This table stores three columns described below:

- Trigger Type - an identifier for the type of trigger, which can indicate what was the cause of the fault in the network. As an example, the switch might be configured to create a trigger when the queue depth at a switch exceeds a certain number.
- Time Hit - the time at which the trigger condition was raised, in the format of a 64 bit representation. The most significant bit in the decimal representation of the 64 bit number would denote the number of seconds passed, while the remaining digits would denote the precision in nanoseconds. For example, the number 1435756194 (stored as a 64 bit number) would mean 1.435756194 seconds.
- Switch - the identifier of the switch that raised the trigger.

3.2.3 The Links table

This table stores three columns described below:

- Switch 1 - the identifier of the source switch for the link.
- Switch 2 - the identifier of the destination switch for the link.
- Capacity - the link capacity in Gbps.

3.2.4 The Packetrecords table

This table stores the following 9 columns corresponding to each packet, as described below. There exists such a tuple for every packet that leaves any switch in the network.

- Hash - the hash of a packet, to identify it uniquely
- Switch - the identifier of the switch the packet has just left.
- Time In - the time at which the packet enters the switch, in nanosecond precision.
- Time Queued - the duration for which the packet is queued in the switch.
- Time Out - the time at which the packet leaves the switch, in nanosecond precision.
- Control Plane - an identifier for the version of control plane that is presently running in the switches.
- Link Utilization - the current egress link utilization of the switch, as measured in the data plane of the switch itself. This field is later used to corroborate our calculations for egress throughput.
- Queue Depth - the current length of the queue in the switch.
- Source IP - the source IP of the packet that the 9-tuple represents.

3.3 Collected Traces

10 collection scenarios were run on the topology testbed described above for use in evaluation of the process described in the following chapters. Out of these, 5 scenarios were cases of microbursts of the synchronized incast variety, and the others were heavy hitter bursts.
Each of these traces had a total of 10 switches and 6 flows, each with a source IP from *10.0.0.1* to *10.0.0.6*. More details about thet traces will be given in the following chapters.

Chapter 4

Extracting insights from Relational Data

4.1 Description of Methodology

The relational schema available for each collection trace and the meaning of each of the fields collected has been described in 3. In this section, the various preprocessing done on each scenario so as to derive useful quantities for visualizing the network and understanding what exactly is happening in the network is described. All preprocessing code https://github.com/sankalp-sangle/FlaskDebugger/blob/master/preprocess.py is written in Python3 and the mysql.connector library is used for communication with the MySQL database.
We describe how to derive three quantities from the existing schema described earlier. The usefulness of these quantities will be shown in Chapter 5.

- Time Series of relative ratio of packets in queue for each flow (as defined by source IP) in each switch in the network.
- Ingress throughput for each flow in each switch in the network.
- Egress throughput for every link as seen in the network.

4.2 Time Series of relative ratio of packets in queue for each flow

4.2.1 Motivation

This quantity helps to understand the composition of the queue in a switch at any point in time. By composition, we mean that if, at a point t in time, the size of the queue in a switch is S packets, and the number of packets of a flow i is n, then our plan is to compute n / S for all flows at evenly spaced points of time. We do this for all switches. Having a plot of this quantity will help us understand how the distribution of a queue changes as packets arrive and leave the queue.

4.2.2 Computation

We use the following process for computing the quantity described above.

1. Select an interval I which will serve as the gap between successful computations of the relative ratios.
2. For a point in time *Tcurrent*, perform the following query on the Packetrecords table.

```
SELECT source_ip, COUNT(hash)
FROM packetrecords
WHERE time_in < Tcurrent AND time_out > Tcurrent
GROUP BY source_ip;
```

3. This query returns to us the list of flows inside the queue of a switch at a particular time Tcurrent (as ensured by the WHERE clause) as well as the number of packets of each flow. The ratios can then be obtained by taking the sum of the packets and dividing each flows number of packets by the total number just computed.
4. This query is then repeated at every I spaced interval (by incrementing Tcurrent by I each time) in the duration of the records of the switch.

4.2.3 Visual Result

To show how a quantity like this will be helpful, refer to Fig. 4.1 where the queue depth has been plotted. The queue depth on it's own does not give us much information on what happened

in the switch, but the composition of the queue (Fig. 4.2 (using the relative ratio quantity computed above) gives us a much more revealing insight into how the queue built up over time. We can clearly understand that the flow coloured in blue is the dominant flow in the queue and is probably a heavy hitter flow. Also refer to Fig. 4.3 where the absolute relative ratio of flows has been plotted against time.

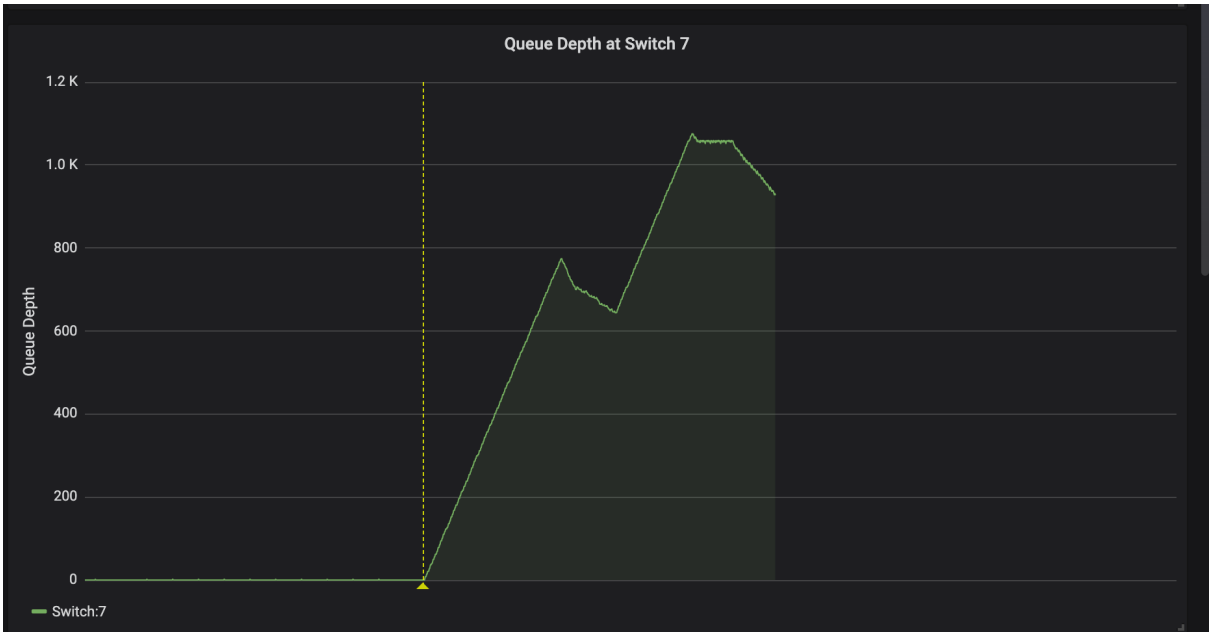

FIGURE 4.1: Queue Depth at a switch

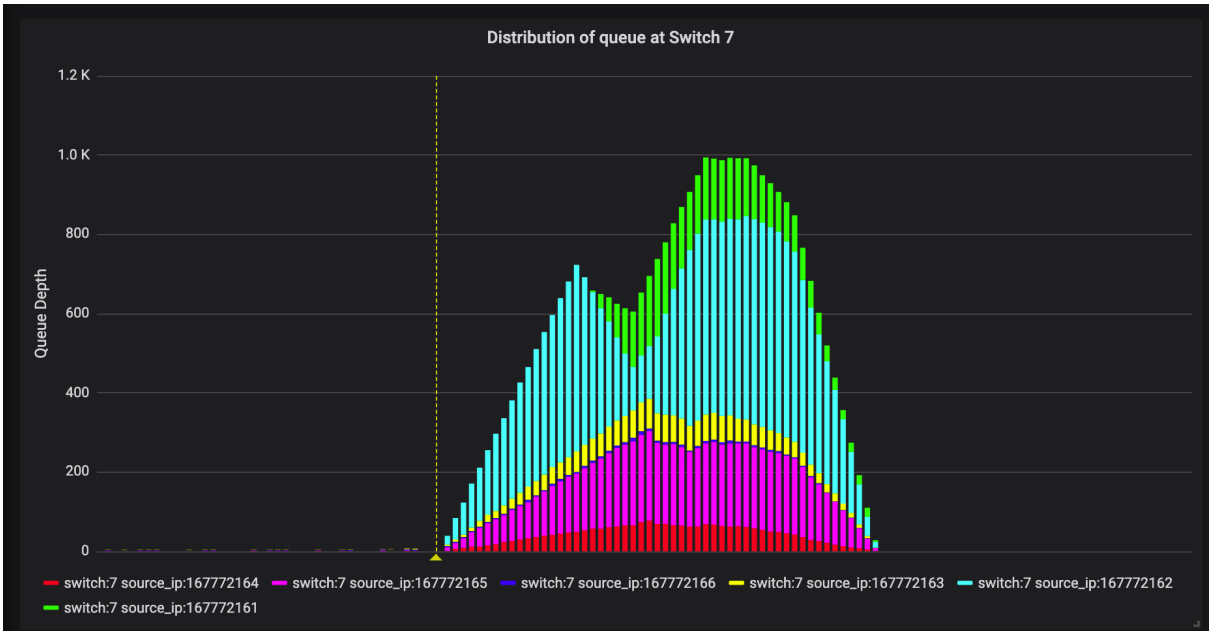

FIGURE 4.2: Queue Depth composition at a switch. Each colour represents packets from one particular flow / source IP

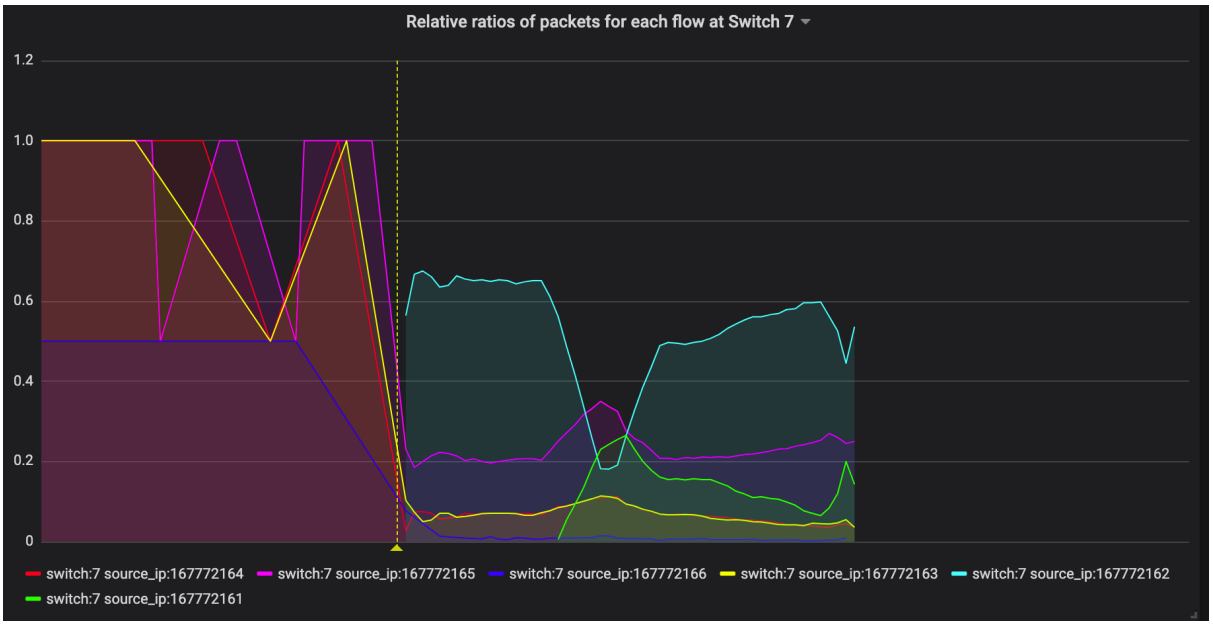

FIGURE 4.3: Relative ratios for flows at a switch

4.3 Ingress Throughputs for each flow in switch

4.3.1 Motivation

This quantity captures the bandwidth of incoming flows in a switch at any point in time. This is the classical definition of bandwidth such that if, in an interval of time T, p number of packets of a flow enter the switch, then our plan is to compute *packet size* * n / T for all flows across all such intervals for a switch. We do this for all switches. Having a plot of this quantity will help us understand the distribution of incoming flows into a switch.

4.3.2 Computation

We use the following process for computing the quantity described above.

1. Select an interval I which will serve as the window size for computing the throughput for a flow.
2. For an interval *[Tleft - Tright]* in time, perform the following query on the Packetrecords table.

```
SELECT source_ip, COUNT(hash)
FROM packetrecords
WHERE time_in > Tleft AND time_out > Tright
```

```
GROUP BY source_ip;
```

3. This query returns to us the list of flows that entered a switch in a particular time interval (as ensured by the WHERE clause) as well as the number of packets of each flow. The throughput for each flow can then be obtained by dividing the incoming number of bits (number of packets * packetsize) by the interval I (which is equal to the difference between Tleft and Tright).

4. This query is then repeated at all I length intervals (by incrementing Tleft and Tright both by I each time) in the duration of the records of the switch.

4.3.3 Visual Result

To show how a quantity like this will be helpful, refer to Fig. 4.4 and 4.5 which represent the individual ingress throughputs as well as the stacked (sum of all) ingress throughput for the switch. It can be seen that the ingress throughput is much more than 10 Gbps which is the capacity of the outgoing link, which is why the queue starts building up due to more packets coming in than those being sent out for a particular interval.

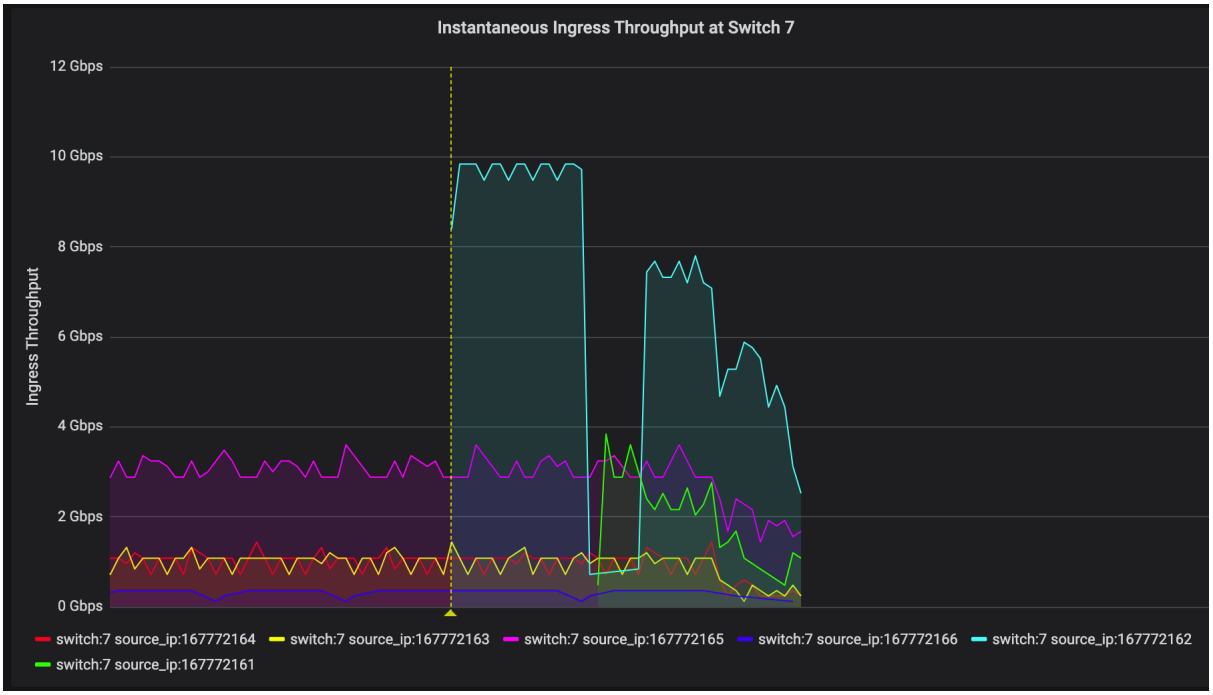

FIGURE 4.4: Individual Ingress throughputs at switch

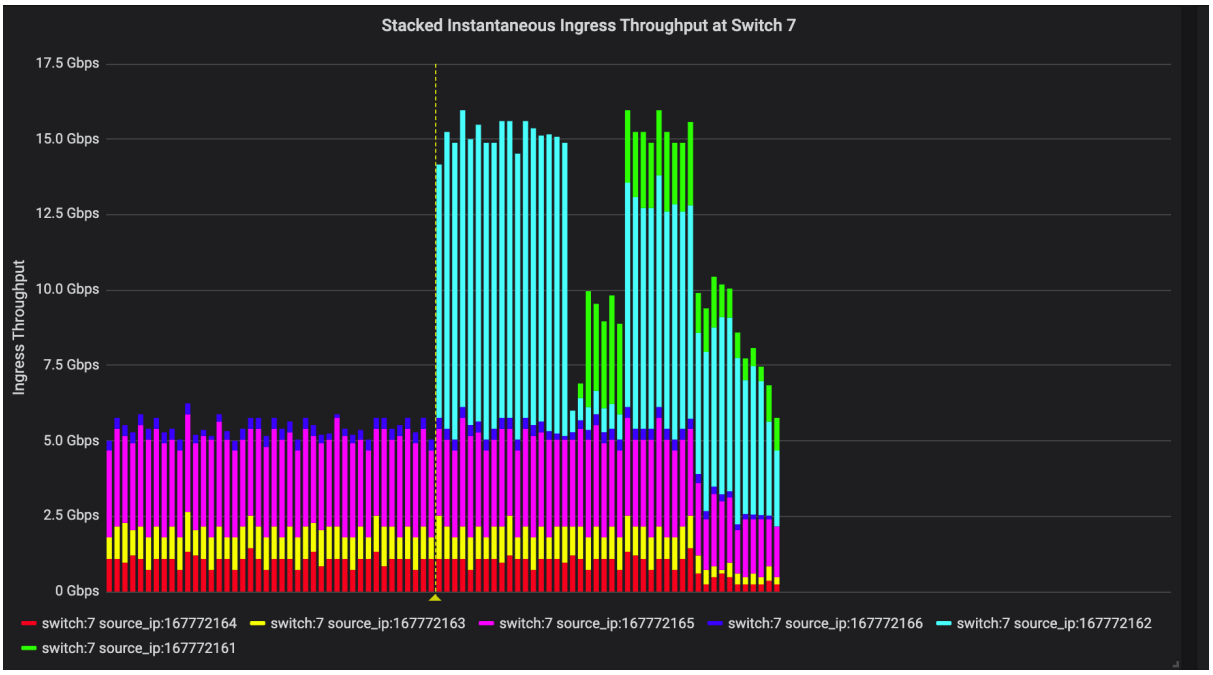

FIGURE 4.5: Stacked Ingress throughputs at switch

4.4 Egress Throughputs for each link in network

4.4.1 Motivation

This quantity captures the bandwidth at an interval of time for a link in the network. This is the most important quantity for creating a visualization that indicates the bandwidth of all links at a particular point in time (Refer to Fig. **??**)

4.4.2 Computation

The computation of bandwidth for all links in the network from the schema provided to us is a 2 step process. We firstly compute a table *linkmaps* as described below.

1. Run the following query on MySQL via Python3

```
SELECT time_in, time_out, switch, hash
FROM packetrecords
ORDER BY hash, time_in;
```

2. This query returns for us the records of a packet as it travels the network in a chronological manner. Thus if a packet visites switches S2, S3, S5, then we will obtain the records for that packet as captured by the switches in the order of the switch visited.

3. From this data, we construct the table linkmaps containing the fields time enter (the timeout value of record at index i), time exit (the time in value of record at index i+1), from switch (the switch value of record at index i), to switch (the switch value of record at index i+1) and hash of the packet in question.

4. Once we have constructed this linkmaps table, the process of calculating throughputs is relatively straightforward. We decide an interval I that will be the width for one calculation of throughput.

5. We run the following query for a particular interval *[Tleft - Tright]* for a particular switch S which we are considering as the source for links whose throughput we are calculating:

```
SELECT to_switch, time_exit
FROM linkmaps
WHERE time_exit > Tleft AND time_exit < Tright AND from_switch = S
GROUP BY to_switch;
```

6. Now we collect all records going to a particular to switch, and calculate the throughput for that interval by dividing number of records times the packet size by the interval duration. This will be the throughput at a time (time exit) for the link represented by source as from switch and destination as to switch.

7. This query is then repeated at all I length intervals (by incrementing Tleft and Tright both by I each time) in the duration of the records of linkmaps table.

4.4.3 Visual Result

To show how a quantity like this will be helpful, refer to Fig. 4.6 the individual egress throughput on link connecting switch 9 and switch 7. It can be seen that the individual egress throughput is constant at 10 Gbps which is the capacity of the outgoing link, showing that the link is being completely utilized.

This quantity is also extremely useful for creating visualizations like that in Fig. 4.7 which give a birds eye view of what exactly is happening in the network. This view which can be changed over time, is invaluable for understanding what exactly happened in the network.

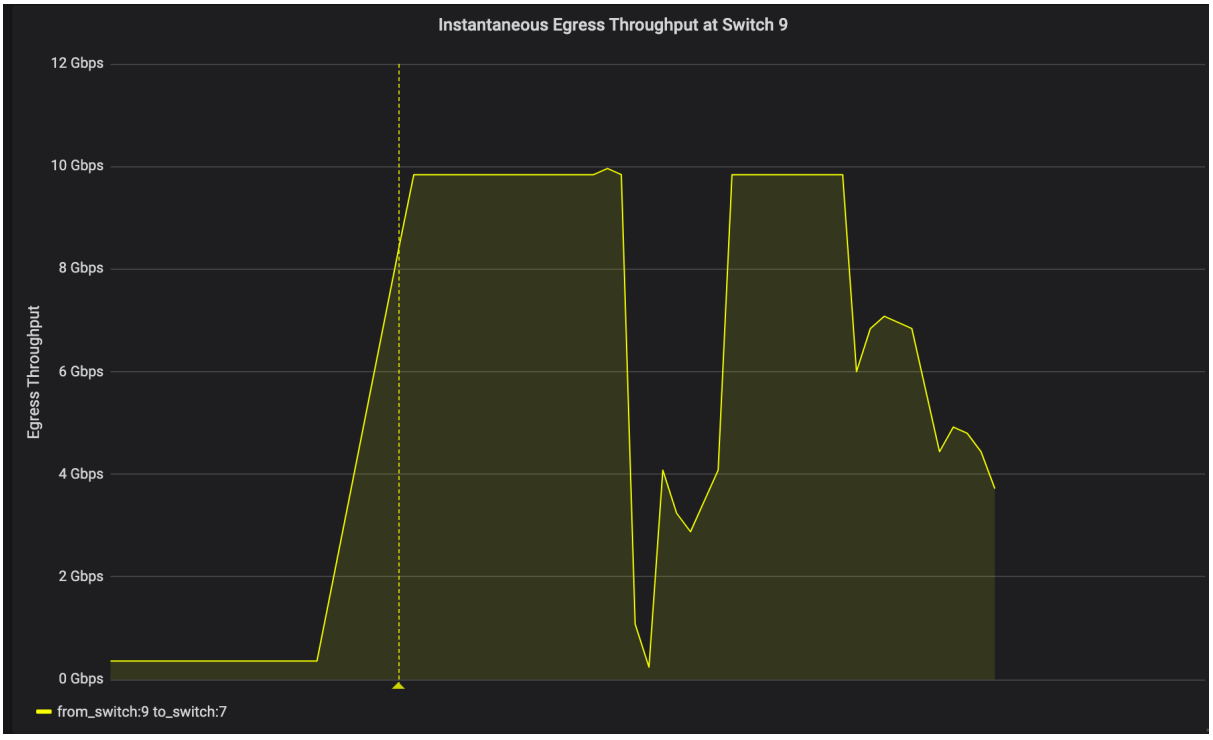

FIGURE 4.6: Individual Engress throughput at switch

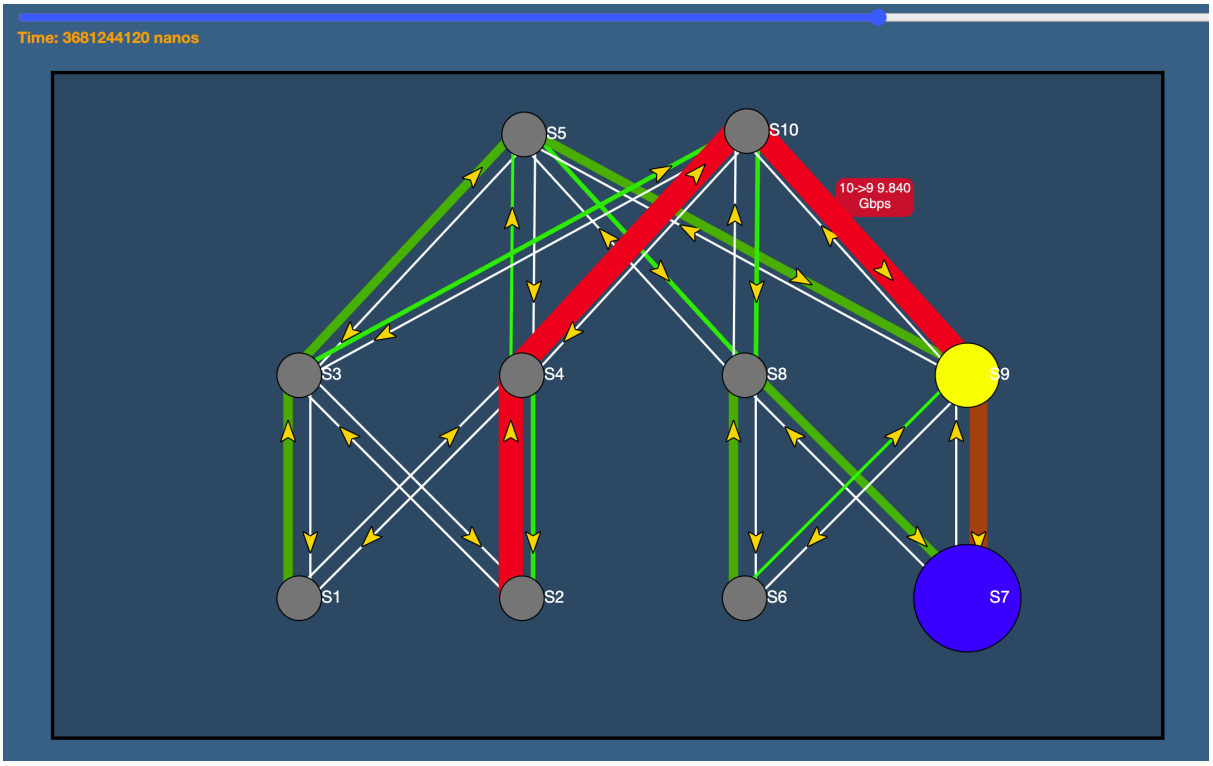

FIGURE 4.7: An example of how powerful visualizations can be created with this quantity. The thick red link indicates existence of a heavy hitter flow dude to very high throughput. Then enlarged blue switch indicates buildup of queue in the switch.

Chapter 5

Demonstration and Debugging of sample scenarios

This chapter is intended to demonstrate how a combination of relational data as well as the created visualizations can help a network administrator to the state and understand the fault in the network.

We look at the following two problems here: The synchronous incast problem and the asynchronousincast problem.

5.1 The Synchronous Incast Problem

5.1.1 Description

This type of problem occurs in data centers in applications like MapReduce and DFS exhibiting fan-in traffic patterns. This occurs when multiple hosts send data to a single host. In the case where the traffic from multiple hosts is synchronized(common in scatter-gather architectures, Figure 5.1), it may lead to heavy congestion for a short duration (of the order of microseconds[9]) and lead to spikes in queuing delay even when none of the flows are individually anywhere near the capacity of the link.

5.1.2 Configuration

For creating a synchronized incast scenario in our topology, we generate traffic of 1 Gbps from hosts H1 to H6. The flow are routed according to Figure 5.2 and get aggregated at switch 7.

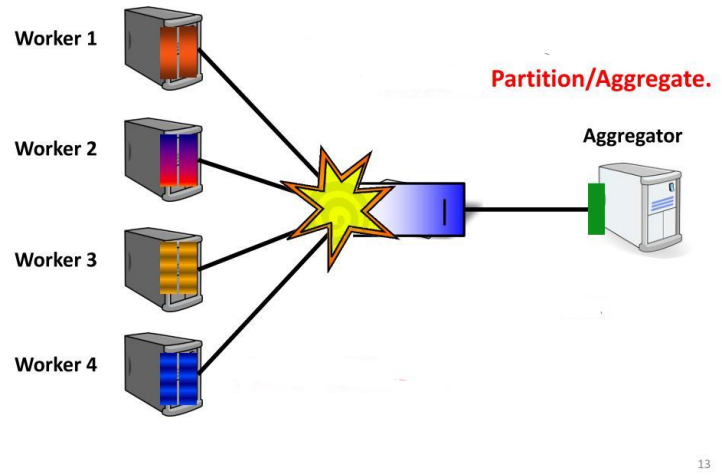

FIGURE 5.1: Synchronized Incast in Partition-Aggregate architecture

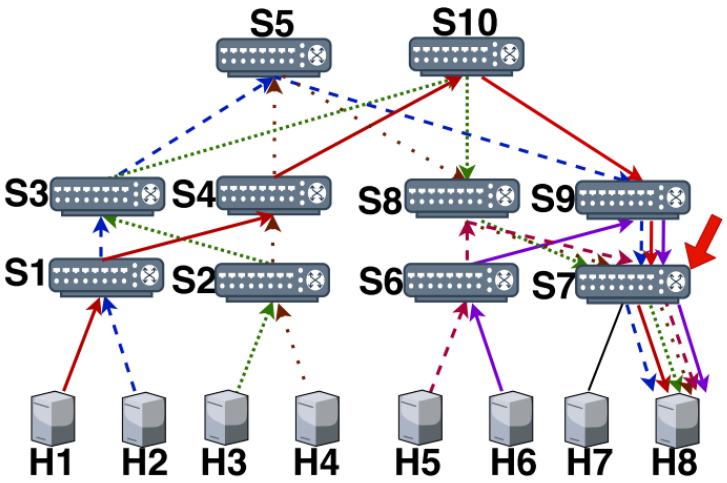

FIGURE 5.2: Synchronized Incast at switch 7 due to synchronized Fan-in traffic

5.1.3 Diagnosis

A synchronized microburst will often lead to multiple spikes occuring in a plot of the queue depth at the trigger switch. An algorithm for determining the width of the peak queueDepth is described here.

Once the peak is estimated, we look at the distribution of packets that came into the queue during the peak as well as the packets that were present in the queue at the start of the peak. We use Jain's Fairness Index (Appendix A) to estimate fairness of distribution of packets according to source IP. If the Jain's Fairness Index turns out to be greater than a predetermined threshold

Algorithm 1 Estimate Width of Peak

Require: indexOfPeak, records, peakDepth

$leftThreshold \leftarrow 0.3, rightThreshold \leftarrow 0.5$
$leftIndex = indexOfPeak - 1$
$rightIndex = indexOfPeak + 1$
while $records[leftIndex].depth \geq leftThreshold \times peakDepth$ **do**
    $leftIndex = leftIndex - 1$
end while
while $records[rightIndex].depth \geq rightThreshold \times peakDepth$ **do**
    $rightIndex = rightIndex + 1$
end while
$width = records[rightIndex].timeOut - records[leftIndex].timeIn$

of 0.7, then it is classified as a case of Synchronized Incast due to the observation that in a synchronized incast, the distribution of packets is mostly fair.

5.1.4 Results and Illustrations

The above heuristic was applied in 5 different scenarios of synchronized incast of varied transmission rates. The resultant values of Fairness Index calculated are given in table 5.1 The operator

Jain's Index	
Scenario	Value
1	0.957
2	0.980
3	0.890
4	0.780
5	0.975

TABLE 5.1: Jain's Index Values for different scenarios of Synchronous Incast

first looks at the readings of Jain's Index (Fig. 5.3 which point him towards the fact that it is a possible case of synchronized incast. Then he looks at the birds eye view of the network and seens that all flows seem to converge at the trigger switch at the same time (Fig. 5.7). Further, to help the network operator visualize the scenario, plots of queue depth at the trigger switch are plotted, along with packet distribution in the estimated peak, as shown in figures 5.5 and 5.4. The operator then looks at the plot of ingress throughput and concludes that all the flows entered the switch at roughly the same time, and there is a certain periodicity in their arrival, as shown in Fig. 5.6 These hints are enough to suggest to him that the problem is of a synchronous incast.

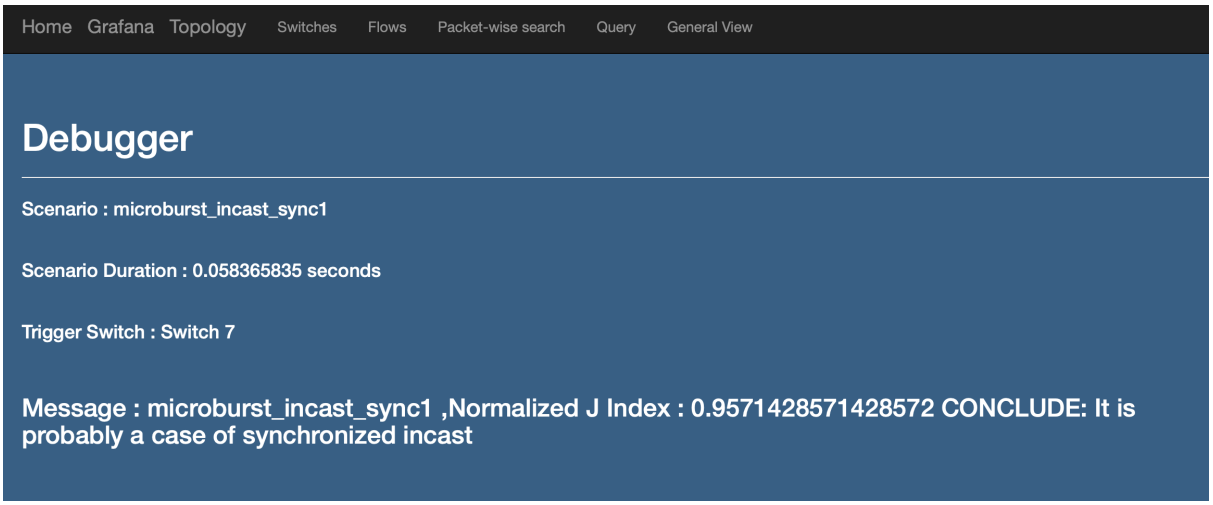

FIGURE 5.3: J Index calculated and presented to administrator

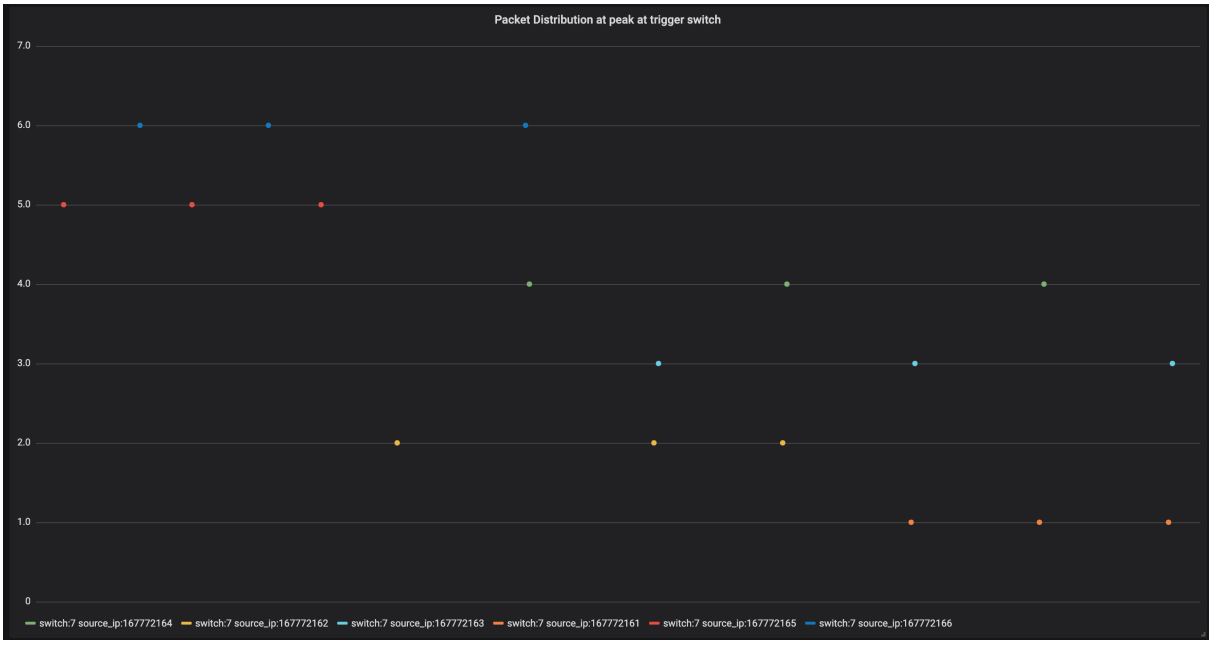

FIGURE 5.4: Packet Distribution at Trigger Switch, Sync Incast. Note the highly even distribution.

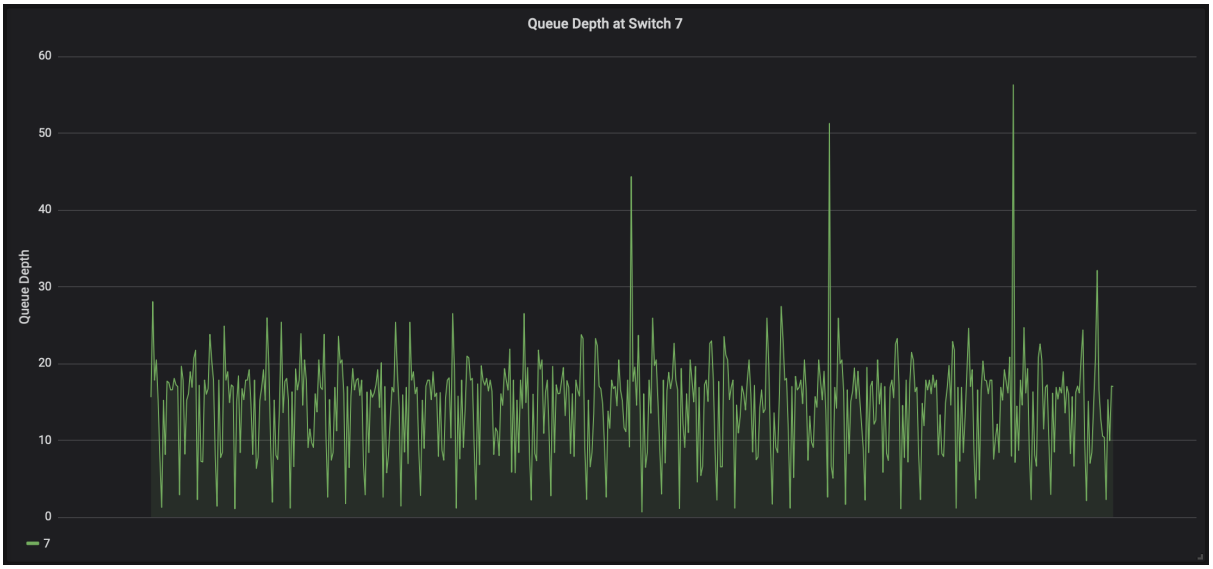

FIGURE 5.5: Queue Depth at Trigger Switch, Sync Incast

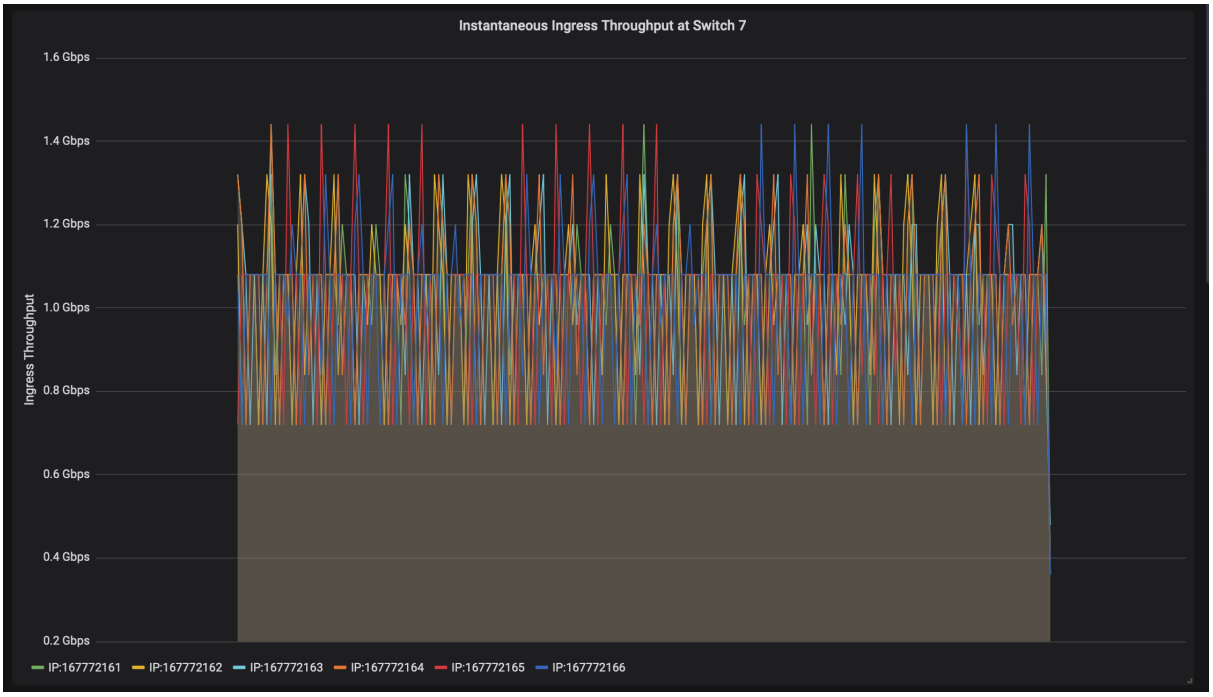

FIGURE 5.6: Ingress throughput for synchronous incast.

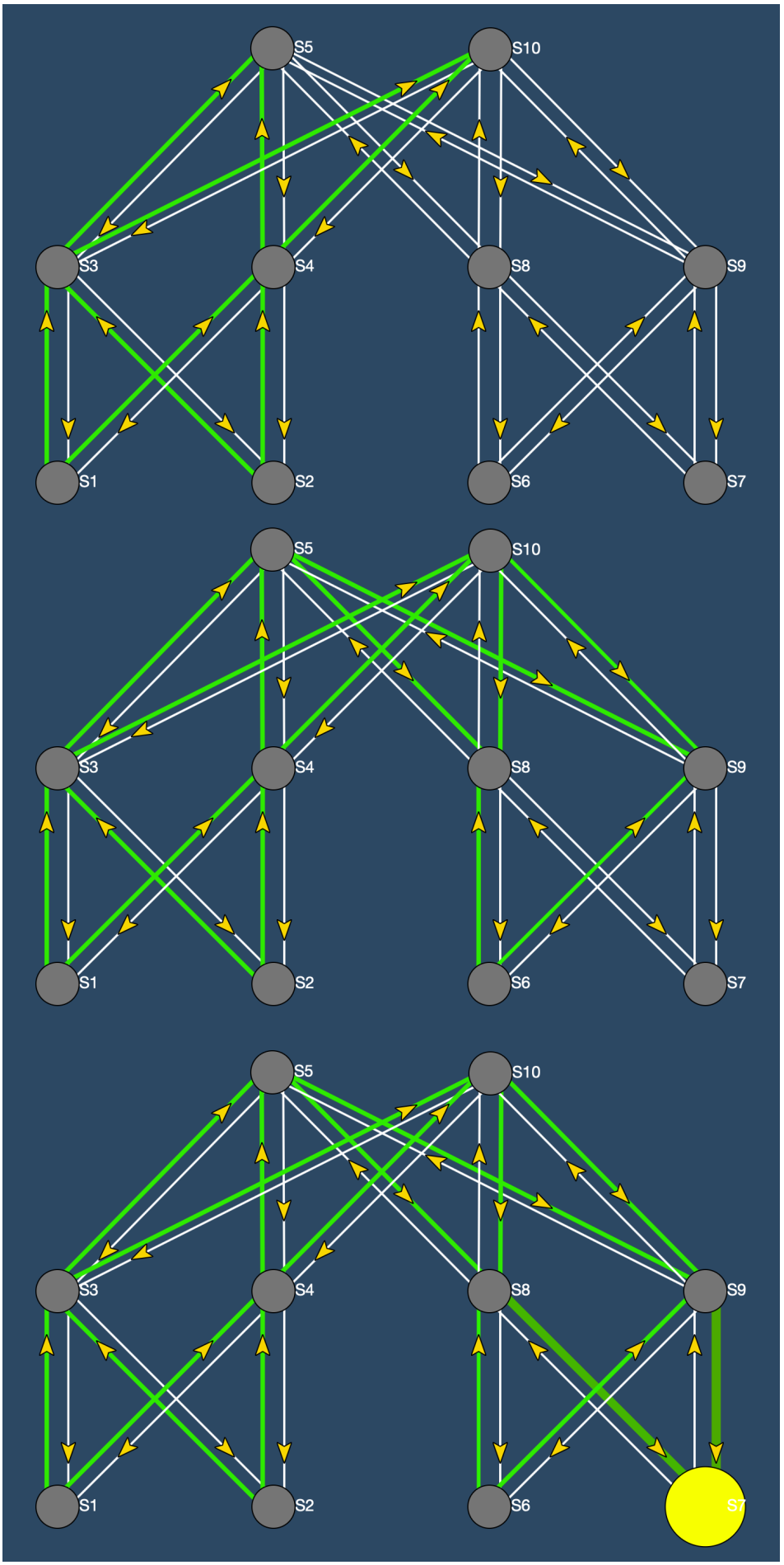

FIGURE 5.7: Birds eye view of network at chronological points in time.

5.2 Asynchronous Incast problem and Heavy Hitters

5.2.1 Description

The Asynchronous Incast problem simply means a scenario where the queue depth increases over a period of microbursts in the lack of any visible synchronization pattern. Often in these cases, we have the presence of a heavy hitter flow[10] (A flow which consumes a substantial amount of bandwidth that violates fairness) which causes congestion and overflowing of buffers in switch.

5.2.2 Configuration

For creating an asynchronous incast scenario in our topology, we generate traffic of 1 Gbps from H1, H3, H4 and 6 Gbps from H6. The flow are routed according to Figure 5.2 and get aggregated at switch 7.

5.2.3 Diagnosis

In cases of asynchronous incast with heavy hitters, we will see a single large peak in the plot of queue depth and usually a very skewed distribution in the peak. The heuristic used here is similar to the one in synchronous incast except if the Jain's Fairness Index is less that 0.45, then it is classified as an asynchronous incast problem.

5.2.4 Results and Illustrations

The above heuristic was applied in 5 different scenarios of aynchronous incast of varied transmission rates. The resultant values of Fairness Index calculated are given in table 5.2.

Jain's Index	
Scenario	Value
1	0.469
2	0.362
3	0.383
4	0.418
5	0.403

TABLE 5.2: Jain's Index Values for different scenarios of Asynchronous Incast

The operator first looks at the readings of Jain's Index (Fig. 5.8 which point him towards the fact that it is a possible case of asynchronized incast. Then he looks at the birds eye view

of the network and seens that there is an interval of time when the links along a path fill up with traffic. (Fig. 5.12). Further, to help the network operator visualize the scenario, plots of ingress throughput and compositon of queue depth at trigger switch are plotted, along with packet distribution in the estimated peak, as shown in figures 5.9, 5.10 and 5.11 Looking at the plot of ingress throughput, the operator sees that there is one heavyhitter flow with a rough throughputof 8 Gbps existing in the network. He also looks at the composition of the queue depth to see that a large number of packets at the peak are of the green flow (the heavy hitter).

These hints suggest a case of asynchronous incast.

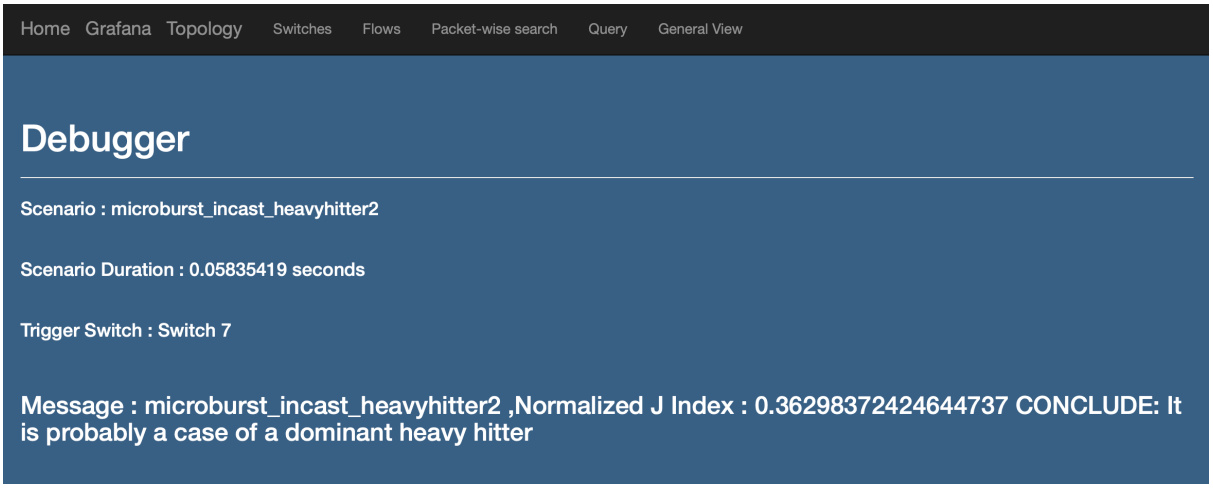

FIGURE 5.8: J Index calculated and presented to administrator

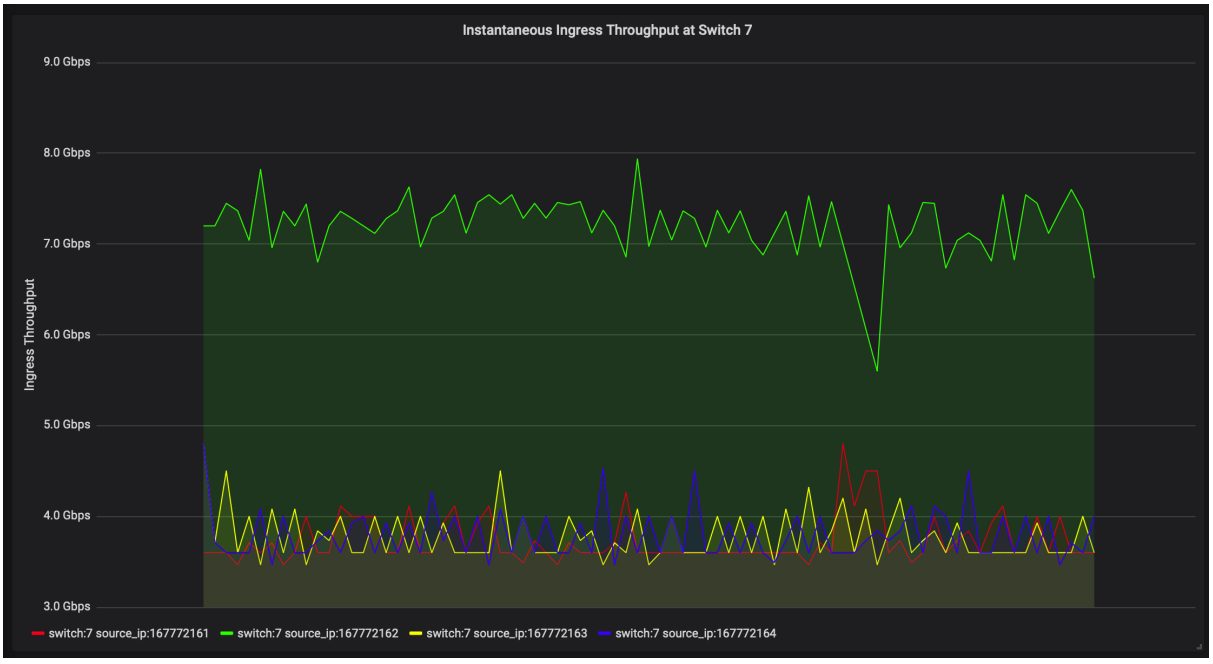

FIGURE 5.9: Ingress throughput for asynchronous incast

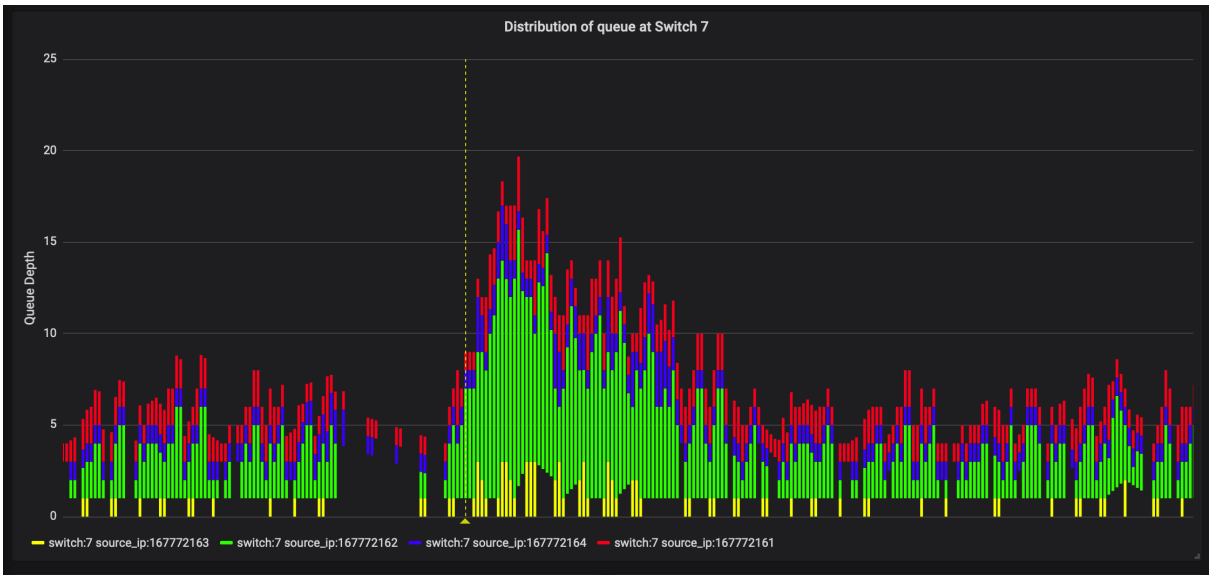

FIGURE 5.10: Queue Composition at Trigger Switch, Async Incast

FIGURE 5.11: Packet Distribution at Trigger Switch, Async Incast. Note the highly skewed distribution.

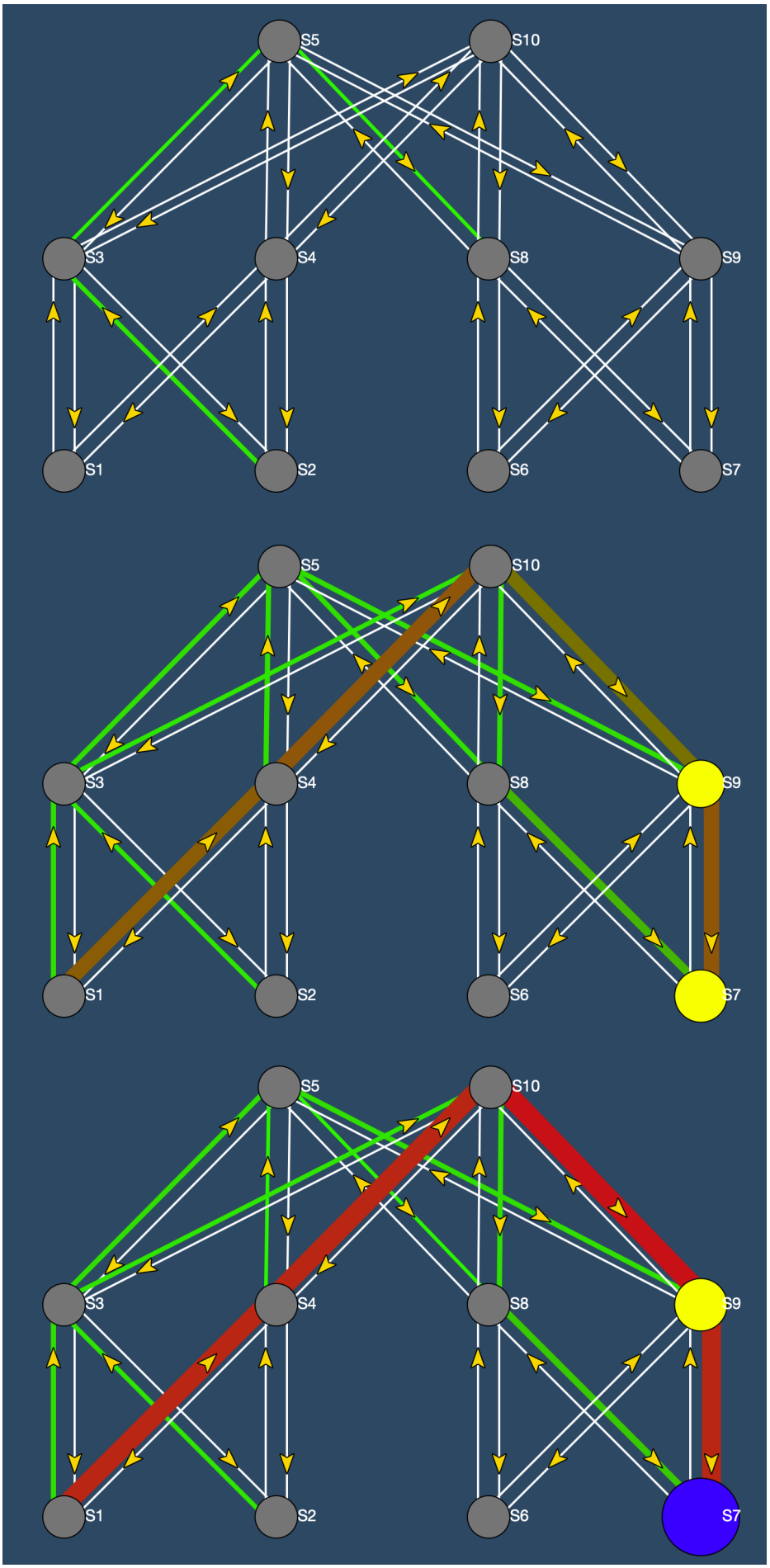

FIGURE 5.12: Birds eye view of network at chronological points in time.

5.3 Link Underprovisioning

5.3.1 Description

A link underprovision occurs when the link is operating at near, or maximum capacity for prolonged periods of time (of the order of milliseconds or seconds). This usually requires the operator to provide additional bandwidth for the link. It also leads to large buildup of queues in switches for prolonged periods of time.

5.3.2 Configuration

For creating a link underprovision scenario in our topology, we generate traffic of 6 Gbps from H3, H4, H5 and 1 Gbps from H1 and H2. The flow are routed according to Figure 5.2 and get aggregated at switch 7.

5.3.3 Diagnosis

As stated before, a link will be underprovisioned if it is overutilized for a duration of the order of milliseconds. If we find that the width of the peak as found in Algorithm 1 is of the order of milliseconds, and that the link is being overutilized, say by setting a threshold to be average utilization greater than 0.8 * max capacity, then we classify the link to be overutilized.

5.3.4 Results and Illustrations

The above heuristic was applied to 2 different scenarios of link underprovision of varied transmission rates. The resultant values of peak width calculated are given in table 5.3

Peak Width	
Scenario	Duration (in microseconds)
1	2575.626
2	4713.071

TABLE 5.3: Peak Width Duration for different scenarios of Link Underprovisioning

The operator first looks at the recommendation given by the application as to what the fault is (Fig. 5.13). The operator looks at the birds eye view of the network and seens that there is massive buildup of queues along with maximum capacity utilization of links.(Fig. 5.16). Further, to help the network operator visualize the scenario, plots of ingress throughput and compositon of queue depth at trigger switch are plotted as shown in figure 5.15. Looking at the plot of

ingress throughput, the operator sees that there is one heavyhitter flow existing in the network. He also looks at the composition of the queue depth to see that a large number of packets at the peak belong to this one flow and that the queue only starts to buildup once this flow starts entering the switch.

These hints suggest a case of underprovisioned network.

FIGURE 5.13: Message given to administrator

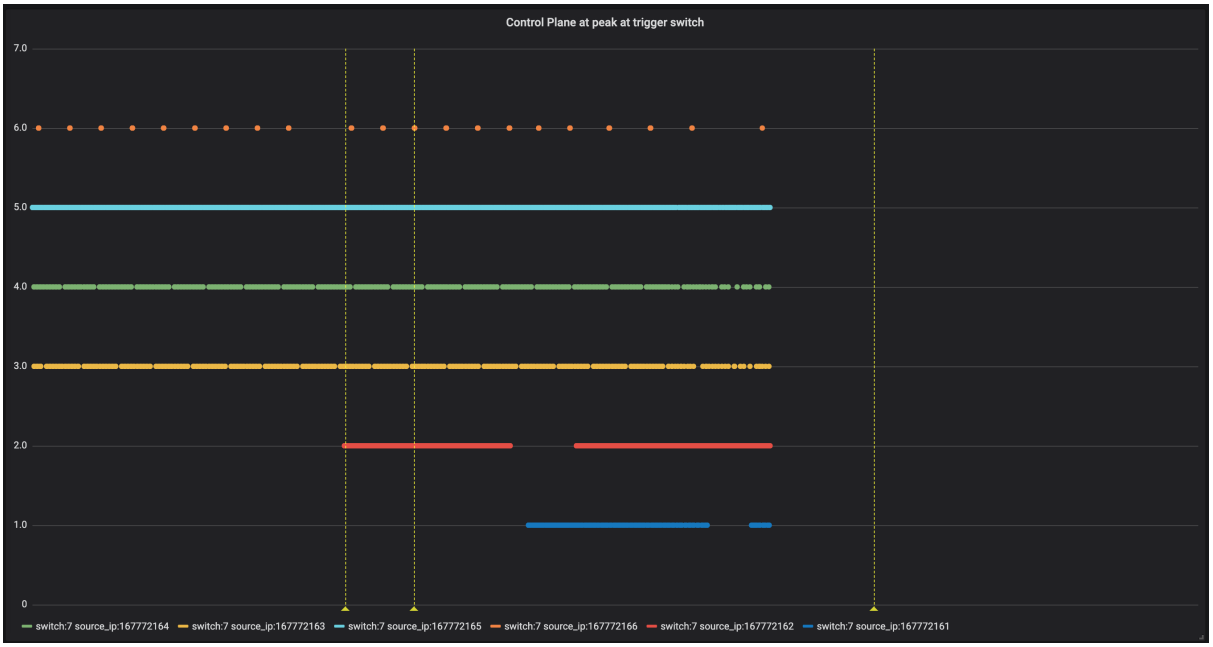

FIGURE 5.14: Packet Distribution at Trigger Switch, Link Underprovisioning. Note the overlapping dots, showing link to be overutilized.

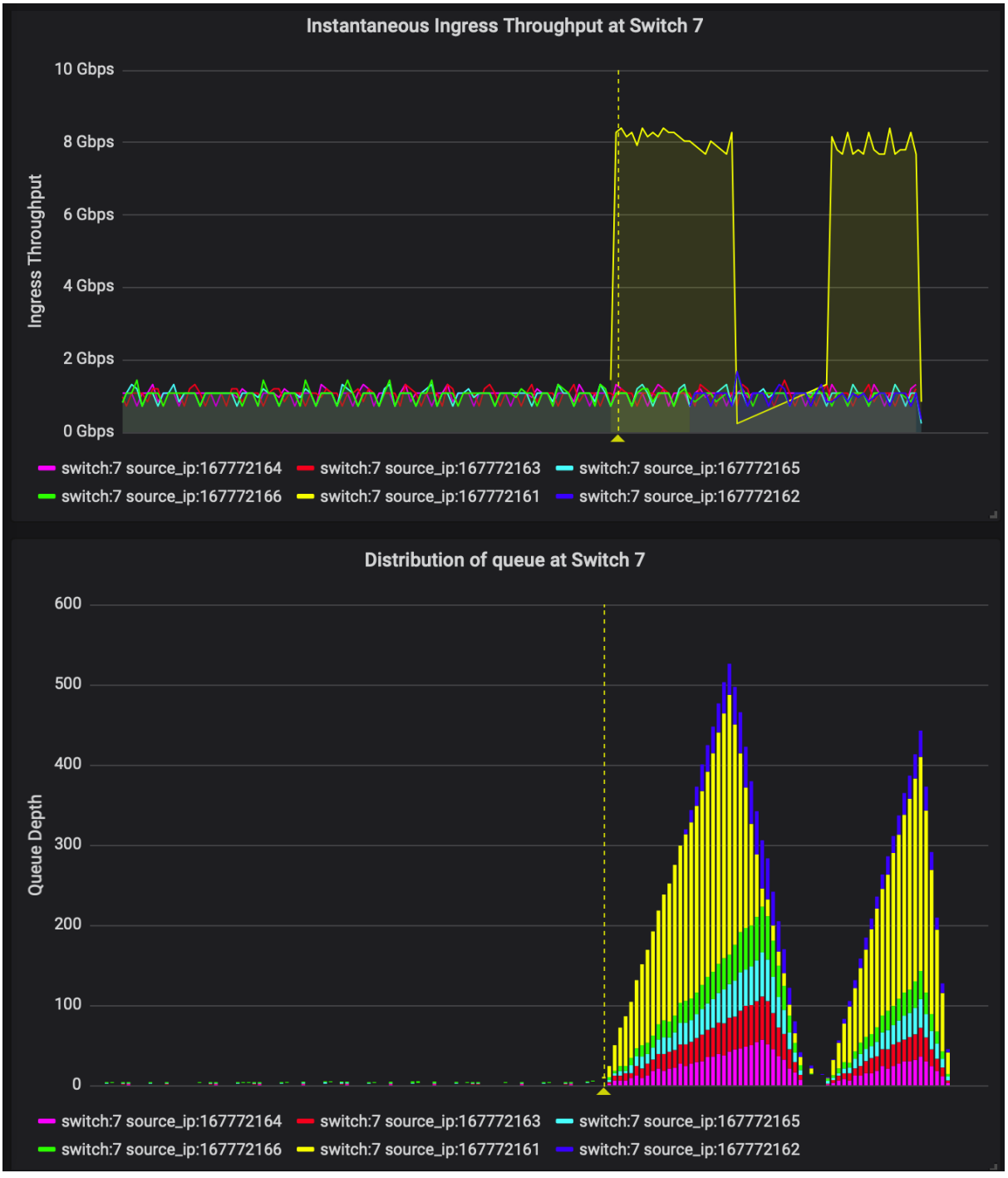

FIGURE 5.15: Ingress Throughput, Queue Composition at Trigger Switch, Link Underprovisioning

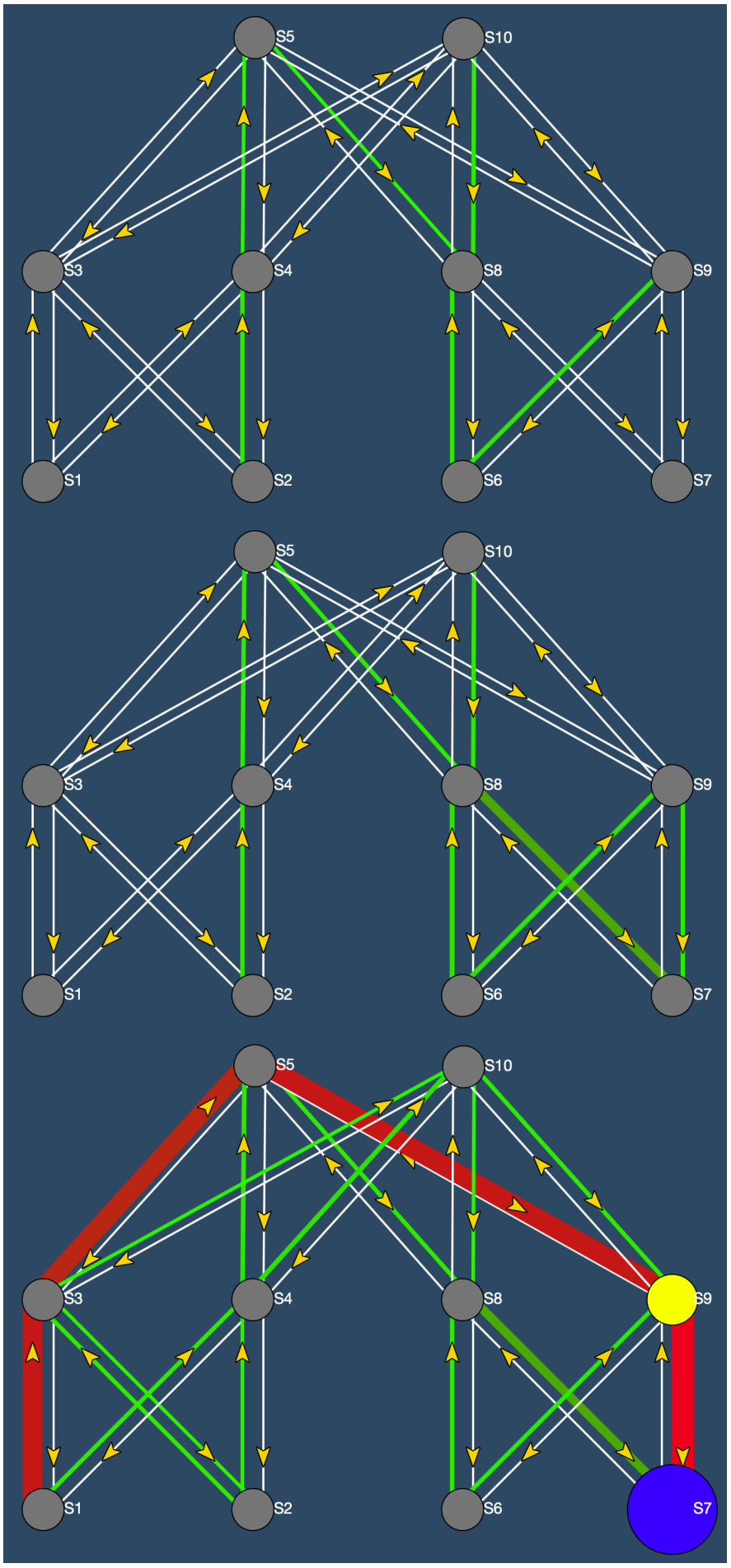

FIGURE 5.16: Birds eye view of network at chronological points in time.

5.4 A unified scheme

Based on the reasoning done in the preceding sections, a unified scheme to classify arbitrary scenarios based on the above discussion would be as follows:

- Estimate the peak width in the graph
- If it is more than 1 millisecond, check if link is being overutilized. If so, it is probably a case of link underprovisioning.
- If it is less than 1 microsecond, calculate Jain's Fairness Index of packet distribution. If it is greater than 0.7, it is probably a synchronized incast.
- Else, if it is less than 0.45, it is probably an asynchronous incast with heavy hitters present.
- Else (it is between 0.45 and 0.7), plot appropriate graphs for queue depth and link utilization and let the network operator deduce the exact problem.

In this way, we can isolate many such problems (for example,link failure and change of control plane policy), diagnose them on an individual basis using the relational model, and at the end consolidate everything together into a single unified logic for diagnosing varied faults in a network.

Thus, this chapter and the previous chaper prove that the relational model is an effective means for debugging the network and identifying issues. This, coupled with effective tools and visualizations built in this thesis demonstratethat large insights can be gained into network functioning.

Chapter 6

Conclusion

6.1 Findings

6.2 Future Scope

Appendix A

Jain's Fairness Index

A description of the Jain's Fairness Index

$$\mathcal{J}\left(x_1, x_2, \ldots, x_n\right) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \cdot \sum_{i=1}^{n} x_i^2} = \frac{\overline{\mathbf{x}}^2}{\overline{\mathbf{x}^2}} = \frac{1}{1 + \widehat{c_{\mathrm{v}}^2}} \tag{A.1}$$

Raj Jain's equation rates the fairness of a set of values where there are n users, each having an associated throughput value. This metric ranges from 1/n in the worst case to 1 in the best case. In this thesis, the metric is normalized to a scale of 0 to 1 before comparing it to thresholds as mentioned in the thesis sections.

Appendix B

Grafana

Grafana (Figure B.1) is a multi-platform open source analytics and interactive visualization software available since 2014. It provides charts, graphs, and alerts for the web when connected to supported data sources. It is expandable through a plug-in system. End users can create complex monitoring dashboards using interactive query builders.

As a visualization tool, Grafana is a popular component in monitoring stacks, often used in combination with time series databases such as Prometheus and Graphite; monitoring platforms

FIGURE B.1: Grafana

such as Sensu, Icinga, Zabbix, Netdata, and PRTG; SIEMs such as Elasticsearch and Splunk; and other data sources.

In this thesis, Grafana is used extensively for graphing the visualizations made using SQL queries using its MySQL plugin. As part of my thesis, I have written a library for automating dashboard creation in Grafana through the provided HTTP API documentation so that the system can autogenerate relevant dashboards and graphs after having analysed the MySQL data using SQL queries.

A link to the GitHub repository with the library is provided:
`https://www.github.com/sankalp-sangle/MySQL-Grafana.git`
A link to the GitHub repository for Grafana is provided:
`https://www.grafana.com`

Appendix C

The Debugger Application

As part of the thesis, I have written a Debugger web application to visualize results and navigate through a loaded trace. This appendix is meant to elaborate on the structure of the web application. Further details and application code can be found on https://github.com/sankalp-sangle/FlaskDebugger

The home page of the application provides a recommendation on what could possibly be the cause of fault in the scenario.

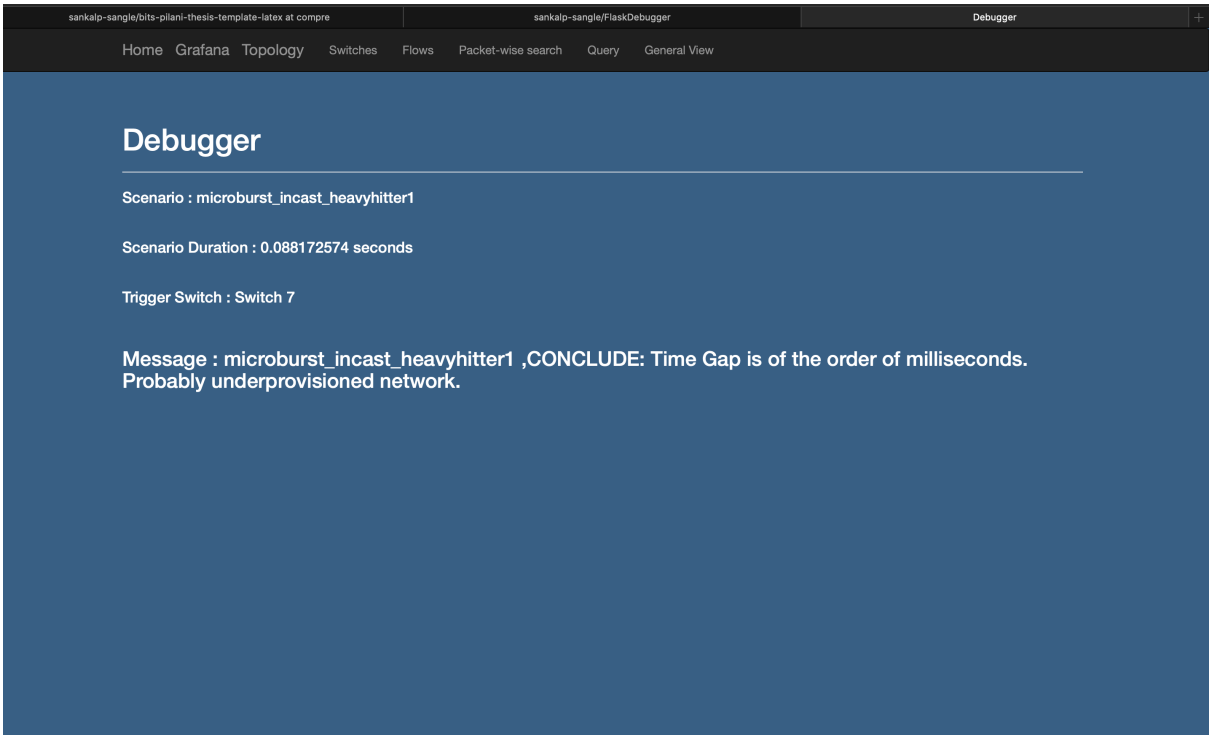

FIGURE C.1: Home Page

From the home page, one can navigate to the topology page to view the network topology. The switches are arranged in order of levels, with switches closer to the core being displayed at the top.

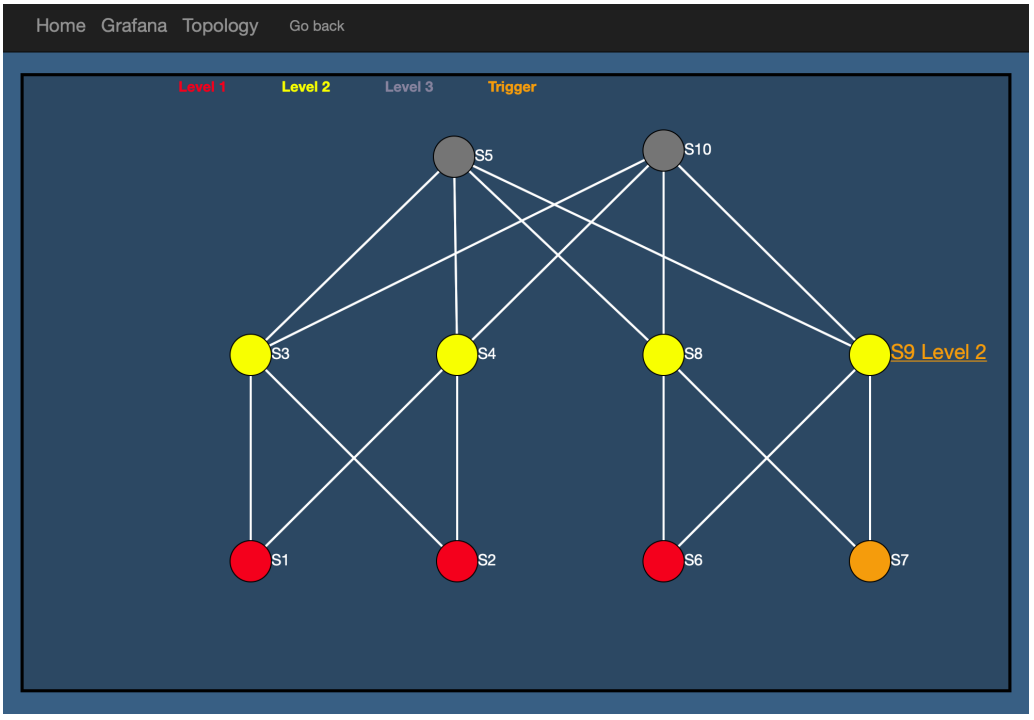

FIGURE C.2: Topology

One can click on any of the switches to see details about that switch.

FIGURE C.3: Information about switch 7

If one wishes to view details about a particular flow, one can click on the the IP address to view more information such as switches visited.

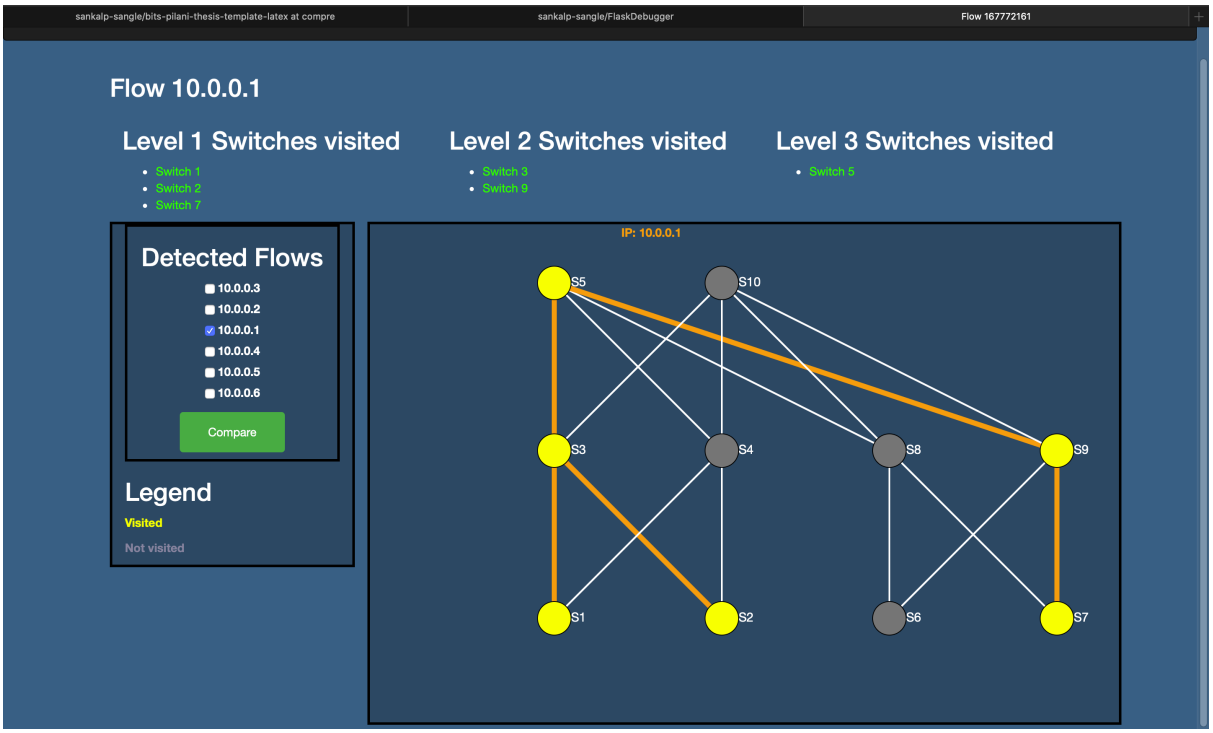

FIGURE C.4: Flow Information

Finally, if one wishes to see a birds eye view of the topology and replay it back, one can do so via the General View tab. We have options to pause, play, speed up and slow down the playback.

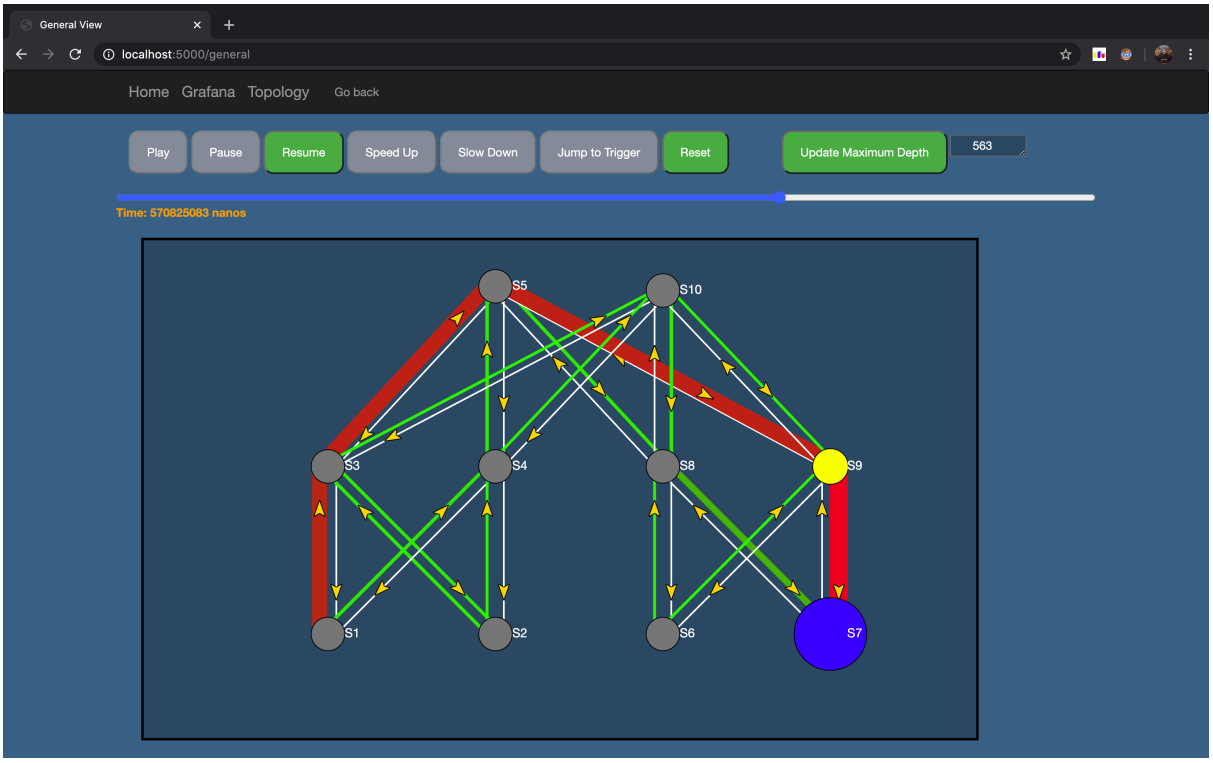

FIGURE C.5: A birds eye view of the scenario

Bibliography

[1] D. S. Alexander et al. “The switchware active network architecture”. In: (1998).

[2] Pat Bosshart et al. “Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN”. In: *SIGCOMM 2013 - Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 2013, pp. 99–110.

[3] Pat Bosshart et al. “P4: Programming protocol-independent packet processors”. In: (2014).

[4] *Grafana, The Open Visualization Platform*. URL: https://grafana.com.

[5] Sushant Jain et al. “B4: Experience with a globally-deployed software defined WAN”. In: *SIGCOMM 2013 - Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 2013.

[6] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. “Precise Time-synchronization in the Data-Plane using Programmable Switching ASICs”. In: (2019).

[7] Nick McKeown et al. “OpenFlow: enabling innovation in campus networks”. In: *ACM SIGCOMM Computer Communication Review* (2008).

[8] Justin Meza et al. “A large scale study of data center network reliability”. In: (2018).

[9] Danfeng Shan et al. “Micro-burst in Data Centers: Observations, Implications, and Applications”. In: (2016).

[10] Vibhaalakshmi Sivaraman et al. “Heavy-hitter detection entirely in the data plane”. In: *SOSR 2017 - Proceedings of the 2017 Symposium on SDN Researchs*. 2017, pp. 164–176.

Turnitin Originality Report Pending....