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

Mid Semester Report

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
March 2020

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 PhD 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. 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. 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 and Google 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.
This thesis aims to discuss techniques for diagnosing faults in data centers at nanosecond resolution. This resolution is provided by existing data plane time synchronization protocols[1]. Data plane switches are used to record network telemetry data in a relational database, which is then queried using SQL to diagnose possible faults in a network and create relevant visualizations which can help a network administrator determine the root cause of faults in the network and address them[1].

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

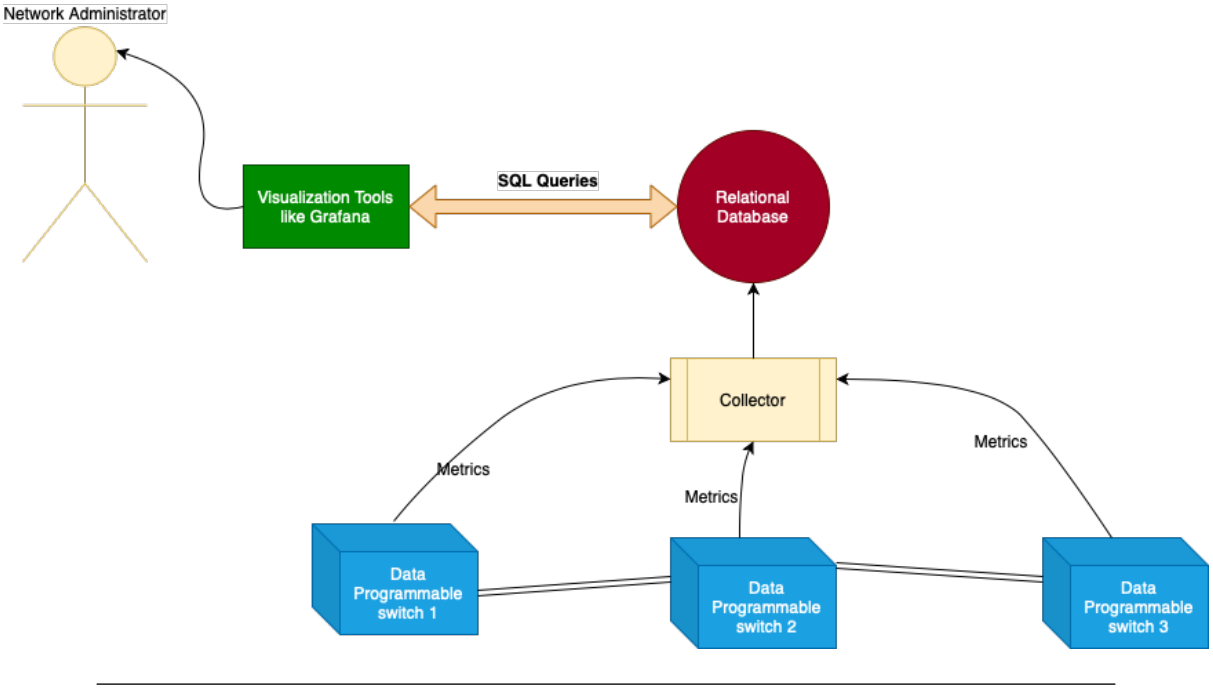

FIGURE 1.1: Architecture Diagram of proposed system

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.
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, and have a programmable control plane that can communicate with the data plane via a standardized API (like OpenFlow) 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. 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 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 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 in Figure 2.1. 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.

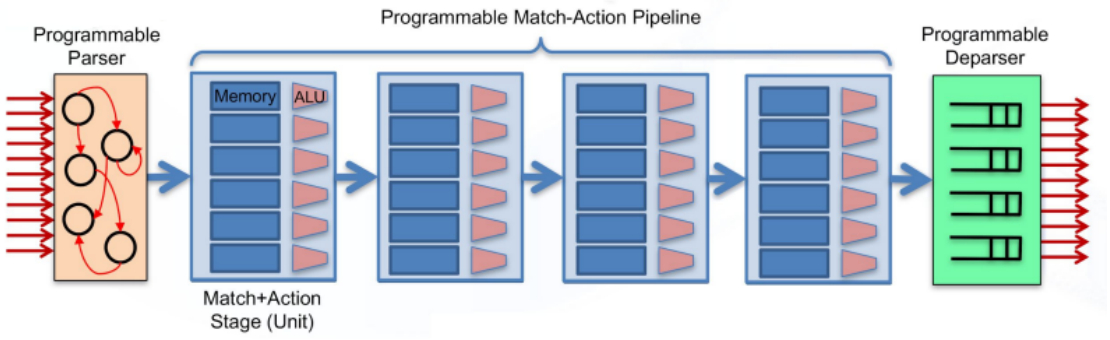

FIGURE 2.1: Generic Data Programmable Switch Architecture

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 (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, and In-Network Computing.

2.4.2 The P4 Programming Language

P4 (*Programming Protocol Independent Packet Processors*) 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

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)

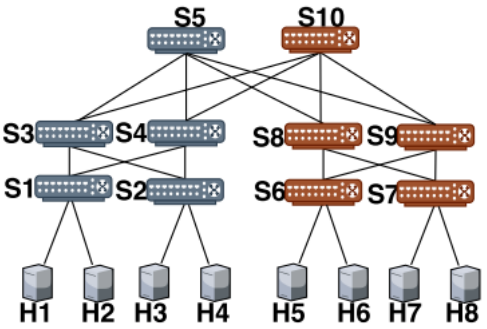

FIGURE 3.1: Evaluation Topology

The rest of this chapter is structured as follows: For a potential fault that can occur in the network, we have

- A section on the description of the fault
- A section on the configuration of switches and hosts for reproducing the fault
- A section on how the proposed fault can be diagnosed using SQL queries
- Results and relevant illustrations

The last section presents a unified scheme for diagnosing faults in networks that has been built on logic that was developed in preceding sections. It also proposes further work to be done till the end of the semester.

3.2 The Synchronous Incast Problem

3.2.1 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 3.2), it may lead to heavy congestion for a short duration (of the order of microseconds) and lead to spikes in queuing delay even when none of the flows are individually anywhere near the capacity of the link.

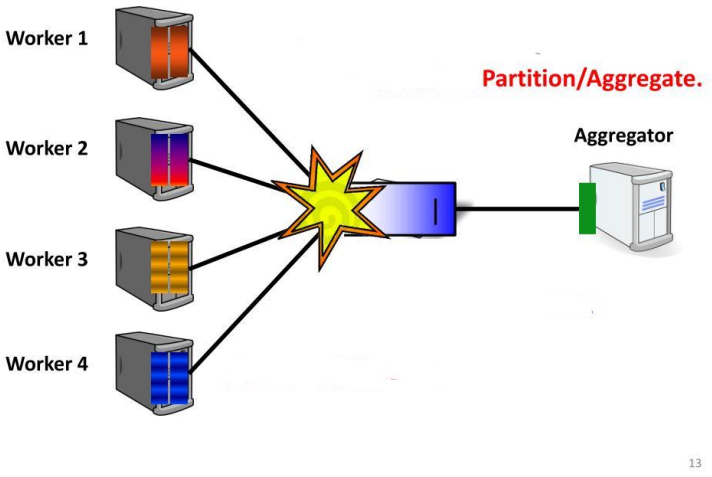

FIGURE 3.2: Synchronized Incast in Partition-Aggregate architecture

3.2.2 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 3.3 and get aggregated at switch 7.

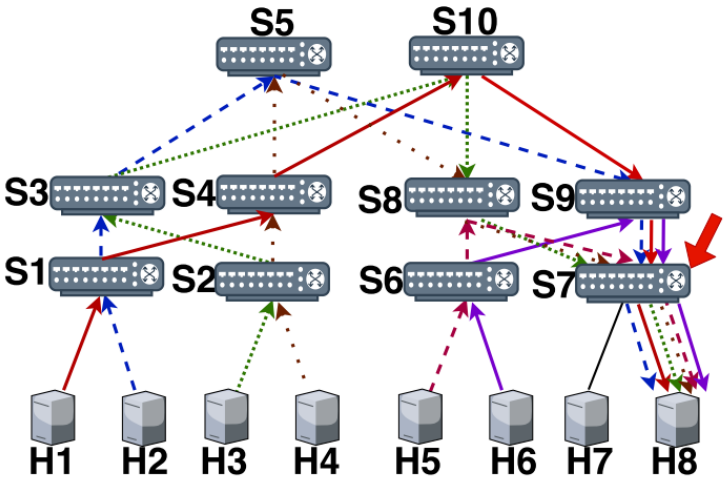

FIGURE 3.3: Synchronized Incast at switch 7 due to synchronized Fan-in traffic

3.2.3 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.

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
$width = records[rightIndex].timeOut - records[leftIndex].timeIn$

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 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.

3.2.4 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 3.1

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

TABLE 3.1: Jain's Index Values for different scenarios of Synchronous Incast

Further, to help the network operator visualize the scenario, plots of link utilization and queue depth at trigger switch are plotted, along with packet distribution in the estimated peak, as shown in figures 3.4, 3.5 and 3.6

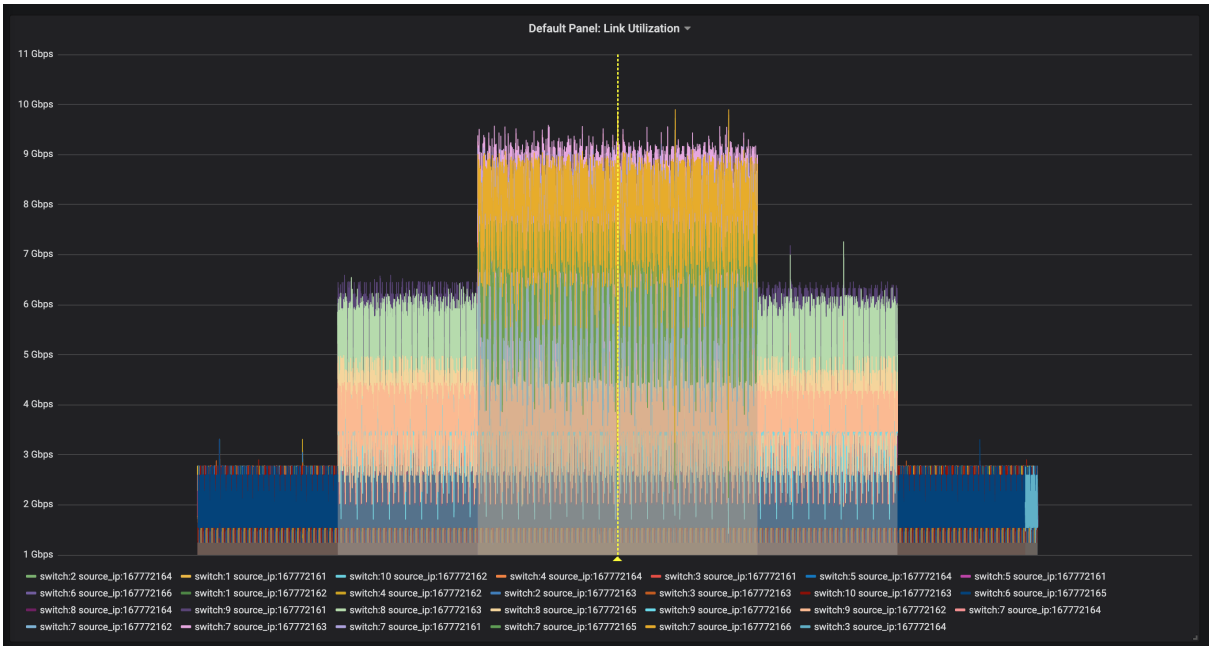

FIGURE 3.4: Link Utilization at Trigger Switch, Sync Incast

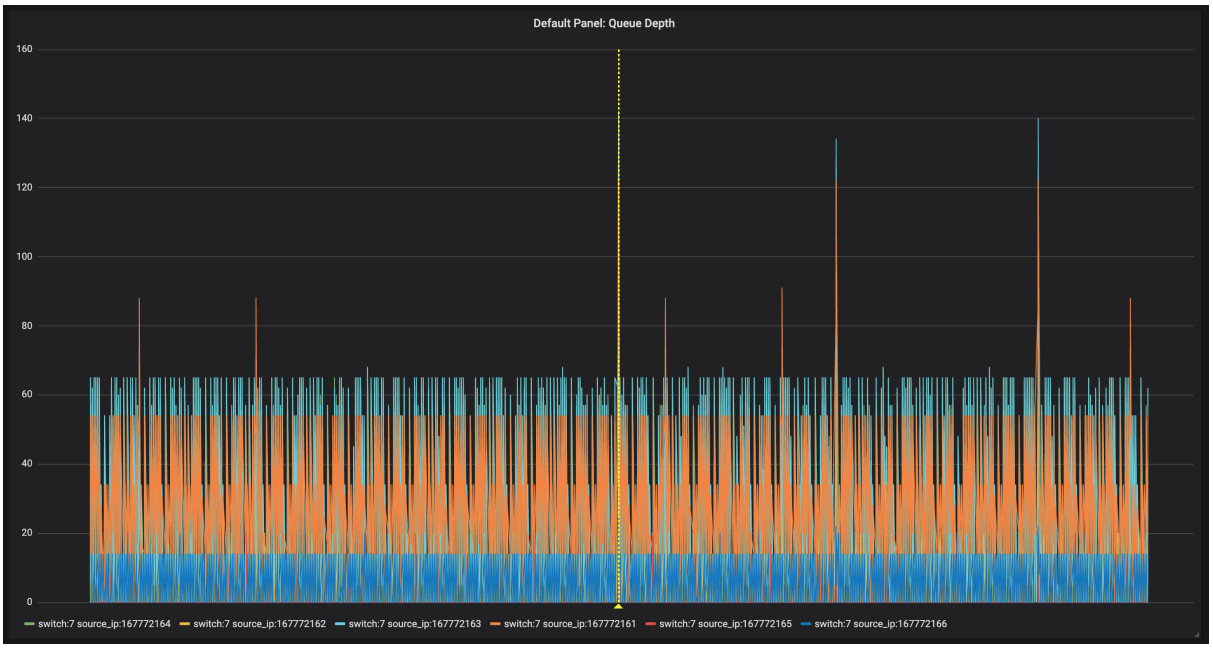

FIGURE 3.5: Queue Depth at Trigger Switch, Sync Incast

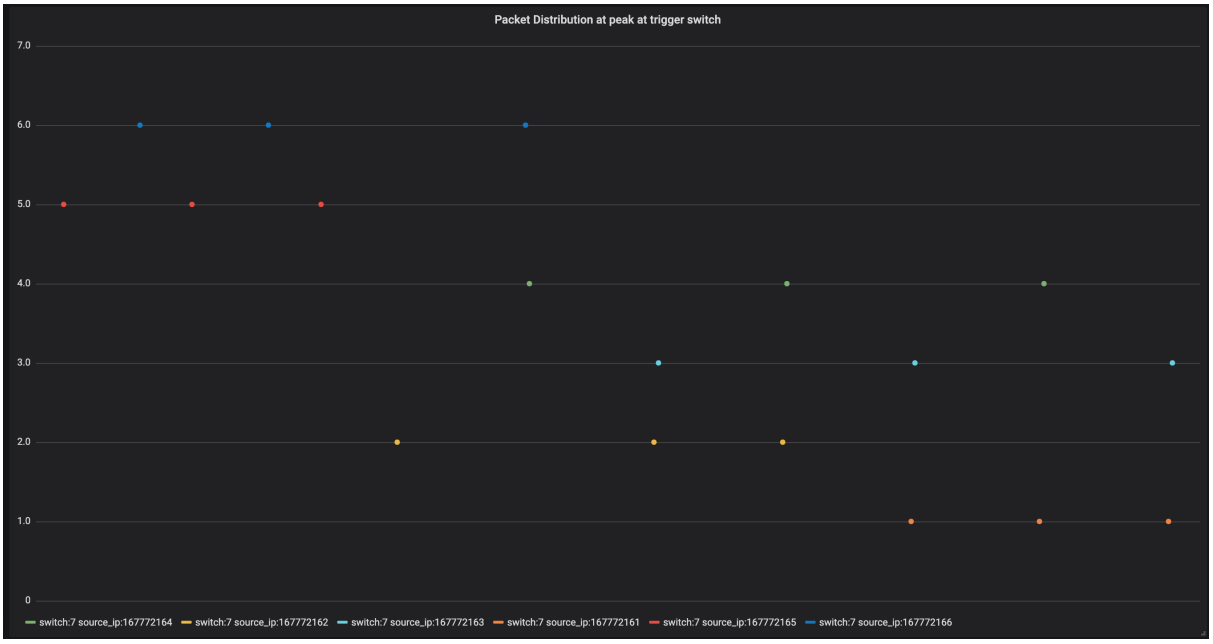

FIGURE 3.6: Packet Distribution at Trigger Switch, Sync Incast. Note the highly even distribution.

3.3 Asynchronous Incast problem and Heavy Hitters

3.3.1 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 (A flow which consumes a substantial amount of bandwidth that violates fairness) which causes congestion and overflowing of buffers in switch.

3.3.2 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 3.3 and get aggregated at switch 7.

3.3.3 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.

3.3.4 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 3.2.

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

TABLE 3.2: Jain's Index Values for different scenarios of Asynchronous Incast

Further, to help the network operator visualize the scenario, plots of link utilization and queue depth at trigger switch are plotted, along with packet distribution in the estimated peak, as shown in figures 3.7, 3.8 and 3.9

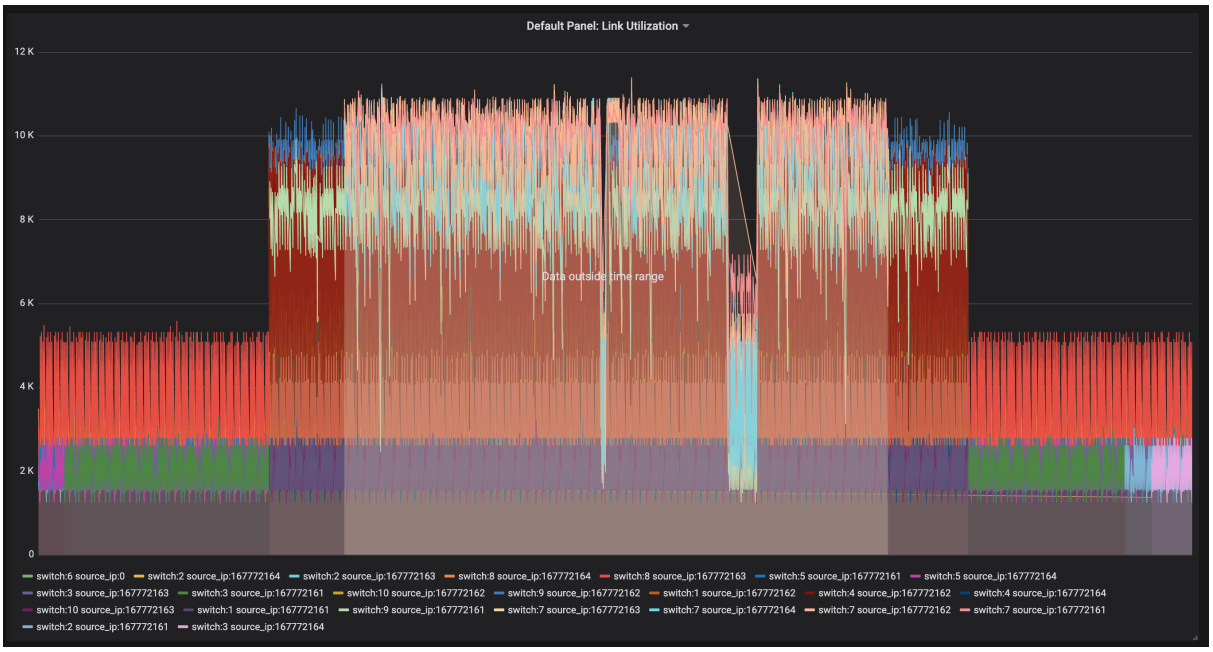

FIGURE 3.7: Link Utilization at Trigger Switch, Async Incast

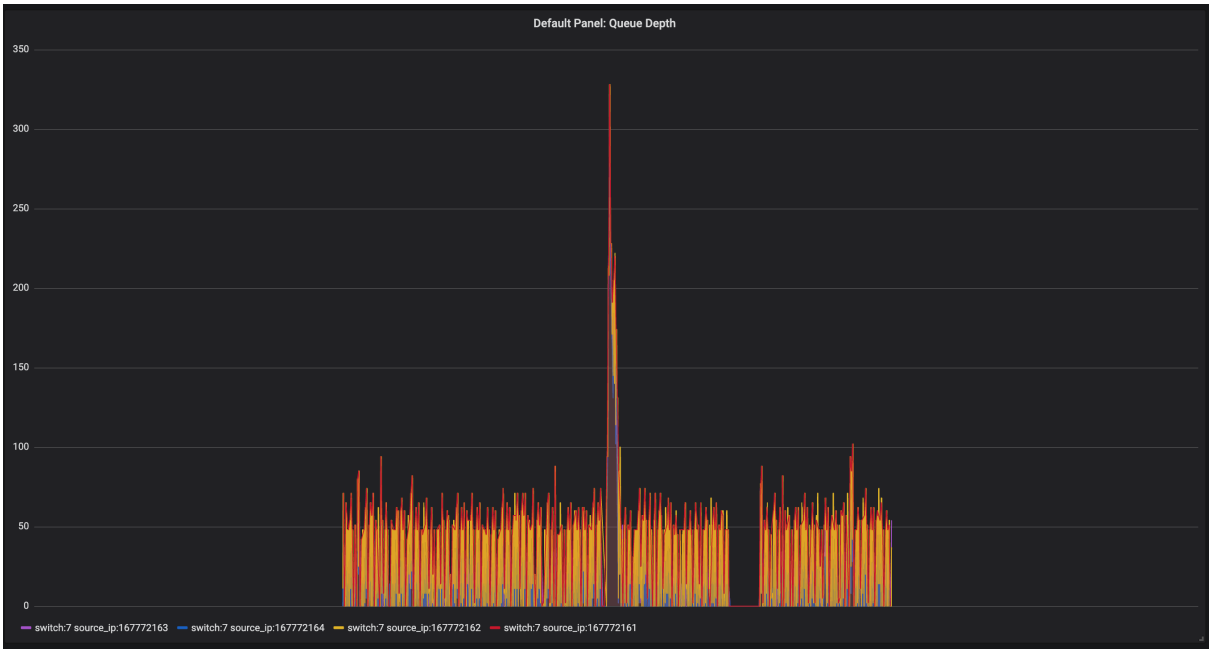

FIGURE 3.8: Queue Depth at Trigger Switch, Async Incast

FIGURE 3.9: Packet Distribution at Trigger Switch, Async Incast. Note the highly skewed distribution.

3.4 Link Underprovisioning

3.4.1 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.

3.4.2 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 3.3 and get aggregated at switch 7.

3.4.3 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.

3.4.4 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 3.3

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

TABLE 3.3: Peak Width Duration for different scenarios of Link Underprovisioning

Further, to help the network operator visualize the scenario, plots of link utilization and queue depth at trigger switch are plotted, along with packet distribution in the estimated peak, as shown in figures 3.10, 3.11 and 3.12

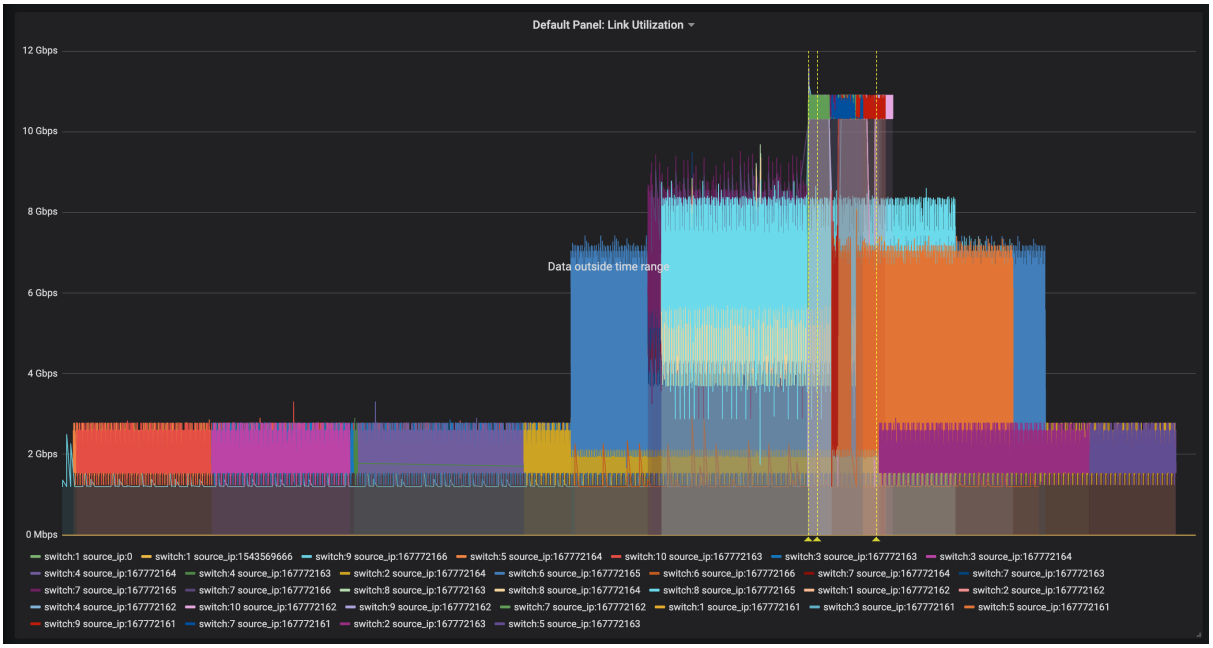

FIGURE 3.10: Link Utilization at Trigger Switch, Link Underprovisioning

3.5 Conclusions and further scope

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

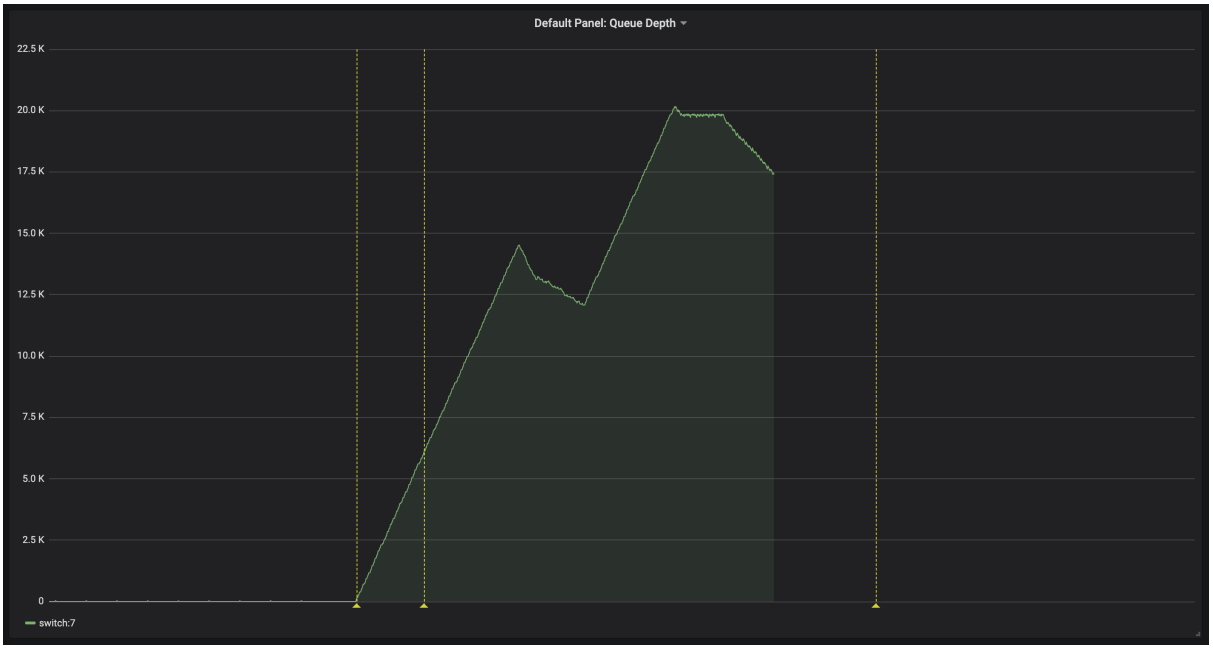

FIGURE 3.11: Queue Depth at Trigger Switch, Link Underprovisioning

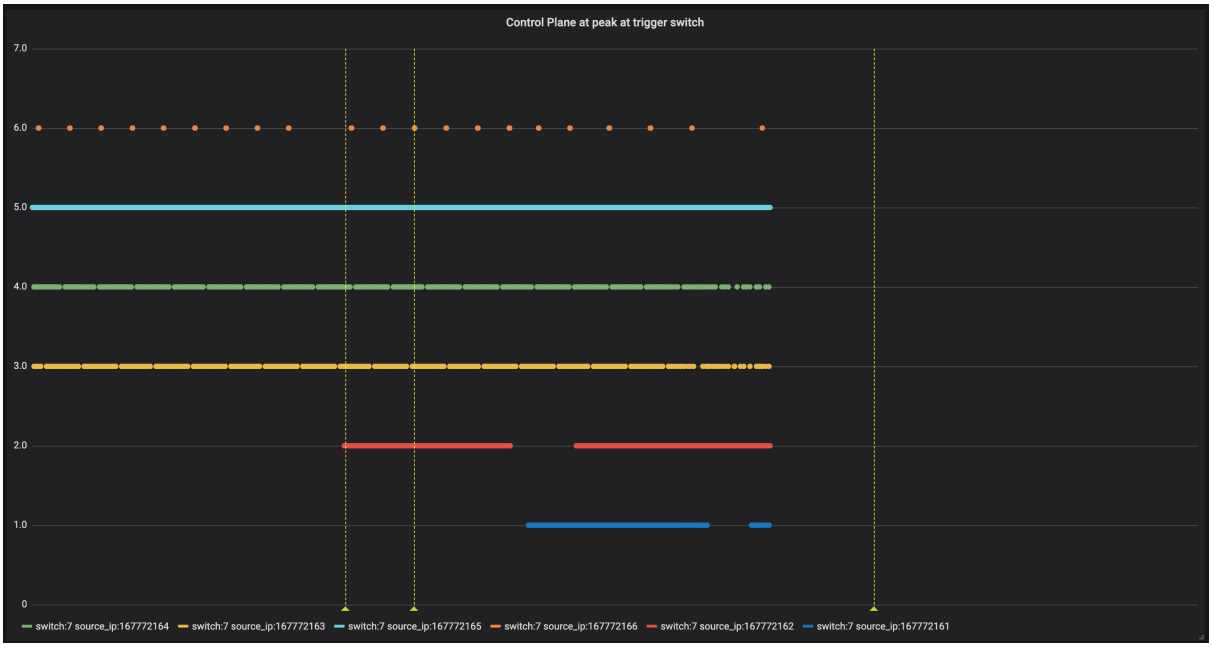

FIGURE 3.12: Packet Distribution at Trigger Switch, Link Underprovisioning. Note the overlapping dots, showing link to be overutilized.

- 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.

As I further pursue my thesis, my goal is to isolate many such problems (two problems I currently have in mind are link failure and change of control plane policy), diagnose them on an individual basis, and at the end consolidate everything together into a single unified logic for diagnosing varied faults in a network.

Appendix A

Jain's Fairness Index

Description on 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.

Bibliography

[1] Kris Kendall and Chad McMillan. “Practical malware analysis”. In: *Black Hat Conference, USA*. 2007, p. 10.