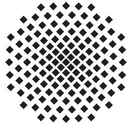

Universität Stuttgart

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 0202-003

Hybrid Application Layer and In-Network Content-Based Filtering in SDN

Muhammad Saqib Khalid

Course of Study:	Infotech
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dr. Adnan Tariq
	Sukanya Bhowmik
Commenced:	2015-04-20
Completed:	2015-10-20
CR-Classification:	C.2.1,C.2.4

Abstract

Content-routing as provided by publish/subscribe systems has evolved as a key paradigm for interactions between loosely coupled application components (content publishers and subscribers). Using content-based forwarding rules (also called content filters) installed on content-based routers (also termed brokers), bandwidth-efficiency is increased by only forwarding content to the subset of subscribers who are actually interested in the published content.

Software Defined Networking (SDN) is a method that enables installation of filters directly on the TCAM memory of network routers. Compared to traditional broker networks, SDN can significantly reduce the latency until events can be received by subscribers: i) the matching time is significantly reduced, and ii) the communication path can be adapted to directly reflect the underlying network topology and therefore reduce the number of forwarding hops for packets. Initial studies have shown that content-routing protocols based on spatial indexing are very well suited to realize a mapping of filtering operations to header-based packet matching performed in TCAM memory. Filters are represented by identifiers constituting the matching field of flows on switches. However, the limited number of available bits for content representation and the limited number of flows available on the TCAM memory for pub/sub traffic limits the expressiveness of these filters, resulting in false positives or unnecessary traffic in the system.

The objective of this thesis, is to design and implement a hybrid pub/sub middleware that allows for filtering of events both on the application layer as well as on the network layer. However, this leads to a trade-off between expressiveness and line-rate performance. In particular, this thesis investigates mechanisms to reduce false positives in the system while maintaining end-to-end latency guarantees at subscribers.

Acknowledgements

Prima facie,I want to offer this endeavor to ALLAH ALMIGHTY for wisdom he bestowed upon me, the strength, the peace of mind and good heath in order to complete this thesis. I started my journey two years back as an immigrant student here in this foreign land, clustered around me such nice, helpful and dedicated people who became the beacon house of light for me and paved my way through.

I would like to express my sincere gratitude to the University of Stuttgart for letting me fulfill my dream of being a student here, explore the true essence of knowledge and equipping me with skills to enter and face the giant technical world. Working as Masters student at the university was magnificent and challenging experience. In these years many people were instrumental directly or indirectly in shaping my academic career and it's an honor and a time of great pleasure to acknowledge their efforts, guidance and motivation without which this thesis would not have been accomplished.

In this regard, I am firstly very thankful to Prof Dr Kurt Rothemal for giving me an opportunity to do my thesis in the department of Distributed Systems and in an instinctive topic that satisfied my knowledge of thirst. I am deeply indebted to my supervisor Dr. Adnan Tariq and Sukanya Bhowmik whose stimulating motivation and valuable ideas helped me to complete this work. I want to express my special thanks and sincere gratitude to Sukanya Bhowmik for being directly involved with me and helping me through each and every step.

I am also gratified to put word of thanks to my parents and siblings for their motivation, unceasing encouragement, and affection helping me tide over my difficulties and enable to do my research and to complete this work.

Contents

Abstract i

1 Introduction 1
1.1 Thesis Organization . . . 2

2 Background 5
2.1 The Publisher/Subscriber Architecture: . . . 5
2.2 State of the Art . . . 6
2.2.1 SCRIBE: . . . 7
2.2.2 LIPSIN . . . 7
2.2.3 GRYPHON . . . 8
2.2.4 SIENA . . . 8
2.3 Conclusion . . . 10

3 Towards an ideal Publish/Subscribe System 11
3.1 Software Defined Networking . . . 11
3.2 PLEROMA . . . 13
3.2.1 The Content Model . . . 13
3.2.2 Event Matching . . . 15
3.3 Implementation . . . 15
3.3.1 Network Configuration . . . 15
3.3.2 Content Delivery Mechanism . . . 15
3.3.3 Publish/Subscribe Request Handling . . . 16
3.4 False Positives . . . 20
3.5 Problem Statement . . . 21

4 Hybrid Publish/Subscribe System 23
4.1 Design Issues . . . 24
4.2 The Control Plane . . . 25
4.2.1 False Positive Request . . . 25
4.3 False Positive Request Generation . . . 28
4.3.1 Un-Advertisement/Un-Subscription Handling . . . 28
4.4 False Positive Request Generation . . . 29
4.4.1 Listener . . . 29
4.4.2 Binary Tree Generation . . . 30
4.4.3 Matching . . . 31
4.4.4 Communicated Data . . . 32
4.5 The Application Plane . . . 32

4.5.1 Registration Request . . . 33
4.5.2 Event Request . . . 34

5 Evaluations **41**
5.1 The System . . . 41
5.2 Test Bed . . . 41
5.3 Test Setup . . . 42
5.4 False Positives In Network . . . 43
5.5 Hybrid System Delays . . . 44
5.6 Delay Variations . . . 45
5.7 Performance Enhancement . . . 46
5.8 Events at the Application plane . . . 47

6 Conclusion **49**

Bibliography **53**

List of Figures

2.1 A publish/subscribe system . . . 6
2.2 SIENA Data Model . . . 9

3.1 The SDN Archetecture . . . 12
3.2 Open Flow Switch . . . 12
3.3 PLEROMA Archetecture . . . 13
3.4 Spatial Indexing . . . 14
3.5 Spanning Tree . . . 16
3.6 IPv4 Address Structure . . . 16
3.7 Flow Handler Case . . . 18
3.8 Flow Table Switch s2 . . . 18
3.9 Flow Handler Case . . . 19
3.10 Flow Table Switch s2 . . . 19
3.11 Un-Advertisment . . . 20
3.12 Un-Subscription . . . 20
3.13 PLEROMA: False Positives . . . 21

4.1 Hybrid Publish/Subscribe Structure . . . 24
4.2 Flow with VLAN tag . . . 28
4.3 Flow table entry with VLAN tag . . . 29
4.4 Binary Tree . . . 31
4.5 Event Matching . . . 32
4.6 Direct Forwarding . . . 37
4.7 Mutual Forwarding . . . 38
4.8 Mutual Forwarding Case 2 . . . 38
4.9 Mutual Forwarding Case 3 . . . 39

5.1 Test Bed Topology . . . 42
5.2 False Positives . . . 43
5.3 Hybrid System Delays . . . 44
5.4 Delay Comparison . . . 45
5.5 Performance Comparison . . . 46
5.6 Events at the Application plane . . . 47

Chapter 1

Introduction

Distributed systems have evolved and rapidly grown in size with time and the applications have become more data hungry. This has led to research in new communication paradigms resulting in the introduction of new models like Event Notification systems [1]. These systems are designed to replace the traditional point-to-point communication between two specific entities by introducing a middleware between them. This enables the entities to communicate without having the knowledge of the existence of each other. Such a scheme presents two big advantages, namely, scalability and loose coupling. This makes possible the development of dynamic large-scale distributed applications like RSS feeds, mailing lists, stock and news updates.

Publisher/Subscriber system [2] is a brand of event notification systems, which has received a lot of attention in recent years. The publishers send out advertisements about the information they will publish and then broadcast events for the advertised information. The subscribers on the other hand are the recipients of the information. A subscriber will send out subscriptions for the desired information and subscribe to the publisher providing the required information. In a stocks update application a publisher will advertise about the stock whose updates it will be providing and different subscribers will subscribe to the publishers provding their desired information. This architecture inherently provides loose coupling and many-to-many communication. In this way the information is exchanged in a much more bandwidth efficient way then the traditional point-to-point and synchronous architecture. Thus such an architecture is ideal to meet the demands of the growing amount of information exchange for current applications.The proficiency of a pub/sub system depends on its expressiveness to direct the information from the publishers to subscribers. This divides the system into two groups topic based and content based systems [2]. In a topic-based system a subscriber subscribes to a predefined topic and then will receive all the events related to that topic, where the content-based approach provides the subscriber to apply more fine grain constraints on the data, so it only receives the events it specifically asked for. Hence the content-based approach provides more expressiveness to the user, which is a very important requirement for many applications. On the other hand the topic based approach is much simpler to implement.

Many different flavors of the pub/sub systems have been proposed in the recent years. One of the drawbacks faced by these systems as compared to the traditional point-to-point scheme is that a new entity called the broker has to be introduced between the two end points.

The brokers are responsible to manage advertisements, subscriptions and filtering of events according to the topic or content. The brokers are implemented at the application layer, this introduces extra delays due to increase in processing times at the brokers and the path length between the end points. PLEROMA [3] discusses a model which takes advantage of the concept of Software Defined Networking (SDN) [4] to produce much better performance then the application layer implementation of the pub/sub systems. SDN decouples the control logic from the data plane where as in traditional switches both are done at the switch level. The control logic is implemented by entities called the controllers who can install and modify flows between the publishers and subscribers. The matching of events is achieved at the network layer using by using the Ternary Content-Addressable Memory (TCAM) of the switches. Hence the filtering of the incoming events can be moved to the network layer, resulting in much better performance with regards to throughput and end-to-end latency.

The model proposed by PLEROMA produces promising results but suffers from limitations in expressiveness, which results in the generation of false positives in the system. False positives are the unwanted traffic in the network that is generated by the flow of events towards uninterested subscribers. This thesis presents a hybrid pub/sub system, which enhances PLEROMA. The events are normally filtered at the network layer. As the system starts to experience false positives, the events causing the false positive are redirected towards the application layer where the filters don't suffer from limitation on the expressiveness. Hence this hybrid system works towards providing line rate performance with shifting to application layer to reduce false positives.

1.1 Thesis Organization

The topics covered by this thesis are covered as follows:

The chapter two starts by providing some background information on the publish/subscribe systems and discusses their basic structure and working. Later in the chapter we discuss some design issues regarding the pub/sub systems and how some famous implementations like SCRIBE, LIPSIN, GRYPHON, and SIENA work.

The chapter three provides some background on Software Define Networking (SDN) and discuses its basic working structure. Then it presents how the PLEROMA pub/sub system incorporates the concepts of the SDN to provide a system with high expressiveness and line rate performance. We see how the content model of PLEROMA is implemented and performs event matching. Then we have a look at and PUB/SUB SYSTEM implantation based on the PLEROMA. In the end of the chapter the problem of false positives faced by the PLEROMA model is explained, which leads to the problem statement for this thesis.

The chapter four presents the concepts implementing a hybrid pub/sub system. We have a look at some design requirements for such a system and then see how the system is implemented to fulfill them.

The chapter five is dedicated towards providing some evaluations for the implemented Hybrid publish/Subscribe system. We have a look at the false positives generated in a hybrid system and its performance as compared to its parent model PLEROMA.

Chapter 2

Background

In this chapter we have a closer look at the pub/sub architecture and discuss some prominent implementations in the recent years. Then we have a look a basic concept of Software-Defined Networking and how PLEROMA adopts these concepts to present a pub/sub system with line-rate performance.

2.1 The Publisher/Subscriber Architecture:

The Pub/sub system also known as an event driven system has grown in popularity because of its asynchronous and decoupled nature. These properties make it ideal for the highly dynamic and data hungry applications of today. A simple pub/sub system is shown in the fig [2.1]. It comprises of two basic components: the communicating parties and a middle-ware to manage the communication.

The Communicating Parties The communicating parties can be classified into two groups: publishers and subscribers. The publishers are the generators of data and are unaware of the receiving parties. A publisher informs the system about the data it will publish by sending advertisements and follows it by broadcasting events that cover that advertisement. When the publisher no longer plans to send data related to an advertisement, it will send out a un-advertisement. The subscribers on the other hand are consumers of data and are unaware of the publishers. They show their interest in receiving particular information by sending out subscriptions. Then the events matching the subscriptions are forwarded towards the subscriber. If a subscriber no longer wants to receive the events related to a particular subscription, it will send out a un-subscription. Now we look at how the flow of events is managed.

The Event Notification Service The Event Notification Service (ENS) is the backbone of a pub/sub system. It directs the events from the publishers to the subscribers. The ENS comprises of many uniformly distributed entities called the brokers as shown in the fig [2.1]. The brokers can be considered as servers and their placement in the network has varied in different implementations [5] of the pub/sub systems. The publishers and subscribers register

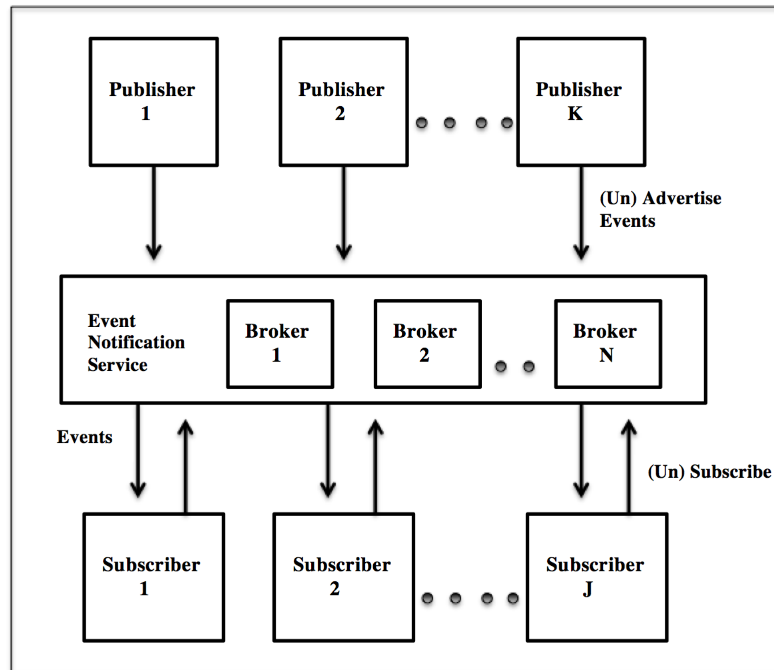

Figure 2.1: A publish/subscribe system

their requests to the broker. The broker is then responsible to perform filtering on the incoming events and direct them to the correct destination.

Filtering Algorithms There are two types of filters that can be incorporated by the broker, topic-based or content-based. In a topic based scheme the subscribers will subscribe for a predefined topic, for example if a subscription for the stocks update for Google is made the subscriber will receive all the events related to changes to this stock. But some times we may need to get events related to specific changes. The content-based scheme provides the ability to apply more fine grain filters, like we can ask to get an update event for the Google stock if it falls within a particular range. Both implementations have their advantages; the topic-base strategy is simpler of the two, whereas the content-based strategy provides more expressiveness for the filters.

There are also some broker-less implementations [5] in which the publisher and subscribers are responsible for the filtration, such a system falls into the peer-to-peer (P2P) domain.

2.2 State of the Art

Now we have a look at some of the famous publish/subscribe implementations that have been presented in recent years.

2.2.1 SCRIBE:

Scribe [6] is a topic-based pub/sub system, which makes use of the P2P principals and presents a completely decentralized implementation. It is build over Pastry, a peer-to-peer object location and routing scheme. It is based on a self-organizing overlay network of nodes over the Internet. Each communicating party in Scribe is a Pastry node with a 128-bit unique-id. A topic is also associated with a topic-id, which is the hash of its name concatenated with the name of the crating node. Scribe creates a multicast tree for each topic to disseminate the events towards the interested parties. To create a topic a publisher requests the pastry service to route a create message to the node with the numeric-id closest to the topic-id. This node then adds the topic to its database and serves as the root of the multicast tree for the respective topic. If the network comprises of N nodes, the routing can be done in O(log N) steps. Similarly when a subscriber wants to subscribe for a topic with a particular topic-id it requests the Pastry service to create a path the root of the multicast tree for this topic-id. The path is created using a strategy similar to reverse path forwarding [8]. Each node in the path to the root is called a forwarder that maintains a database of topics registered. Now when the publisher wants to publish an event it can send the message directly to the root of the related tree if its IP is known. In the other case the Pastry Service routes the message to the root. Then the event is disseminated from the root of the multicast tree. Scribe presents simple self-organizing solution but suffers from the inherit expressiveness issue of topic-based filtering and just offers best-effort delivery. Thus this strategy is good for applications like whether updates but does not meet the need for applications with hard requirements on Quality of Service (QOS).

2.2.2 LIPSIN

Line Speed Publish/Subscribe Inter-networking (LIPSIN) [9] is a topic-based system. It provides much better performance compared to many other well-known pub/sub systems, as it is not an overlay but implemented on the network layer. The system is divided into two planes: the control and data plane. The control plane is responsible to establish the routes and perform matching operations. It is comprised of a rendezvous system and topology system. The data plane is responsible to disseminate events in the system.

The system is initiated by bootstrapping the topology and rendezvous systems. During the bootstrap process each node in the network learns about it local connectivity by relaying messages to it neighbors. Similarly the rendezvous nodes send messages to advertise about themselves. Each link in the network has a unique link-id that is used by the topology system along with the connectivity information to form a network graph. Now when a publisher wants to send a publication the rendezvous system finds the matching subscribers. Then the topology system creates a graph directed from the publisher to the subscribers. LIPSON then uses the concept of Bloom filters [10] to encode the link-ids in the tree. The mapping of the bloom filter to the topic is handed to the publisher, which is then used to publish events for the respective topic. When an event is received at a forwarding node each out going link-id

is ANDed with the bloom filter. If the result is a match to the link-id the event is forwarded through the link.

LIPSON provides a very efficient solution with respect to end-to-end latency and throughput. But Bloom filters have an inherit problem of creating false positives i.e. relaying events towards non-subscribers. This creates unwanted traffic in the network and greatly affects the performance of the system. Also as LIPSIN is a topic-based system the expressiveness is limited.

2.2.3 GRYPHON

GRYPHON [7] is a content-based pub/sub system that implements a broker-based strategy. It builds on a non-distributed algorithm. The Subscriptions are organized in a parallel search tree (PST) data structure where at each node level an attribute test is placed. A subscription comprises of the complete path from the root to the leaf. The matching operation is performed by following the paths from the root towards the leaf that satisfies the attribute test at each level. This is an efficient matching strategy as it takes full advantage of commonalities among different subscriptions. The system build over this matching algorithm consists of a network of publishers, brokers and subscriptions. Each broker in the network maintains a copy of the PST. When a broker receives an event it performs enough processing on the PST to find the subset of neighbors will receive the event and then forwards it to them. In this way the matching is distributed throughout the network. GRYPHON provides a very efficient solution for event matching and forwarding. However, the requirement that the subscriptions are to be replicated on all brokers causes a burden on broker management and is a stumbling block to scalability.

2.2.4 SIENA

Scalable Internet Event Notification Architecture (SIENA)[11] is a content-based event notification service that works towards providing a system with maximum expressiveness without sacrificing its scalability. The system is made up of a number of servers distributed across the network and provides access points for the clients to connect with the servers. There are five kinds of functions (advertise provided by SIENA to the clients to use the event notification service. Advertise is used by the publishers to inform about the notifications they will inject in the system. Subscribe is used by the subscribers to inform the server about the notifications it is interested in receiving. Publish is used by a publisher to send out notifications. The clients can use Un-Advertise and Un-Subscribe to cancel registered requests.

A notification fug [2.2] in SIENA is represented as a data structure that comprises of a set of typed attributes. Each attribute ä=(name, type, value) has a name, type and value. Notification selection is performed by the subscriptions and advertisements by event filters. Event filters select notifications by applying constraints on the notification attributes. A

constraint ö=(name, type, operator, value) contains an extra entry the operator as compared to the attribute. SIENA provides matching and ordering (=, >, < etc.) operators for all of its data types, also substring, prefix and suffix operators for strings. An attribute matches a constraint if they have the same name and type and the operator (valueä, valueö)=true. The fig shows some examples of notifications covered by advertisements and subscription. The only difference in the matching relation of advertisements and subscriptions with the notification arises when we have multiple constraints for the same attribute. In case of a subscription all the constraints need to be met while for an advertisement only one constraint match is enough.

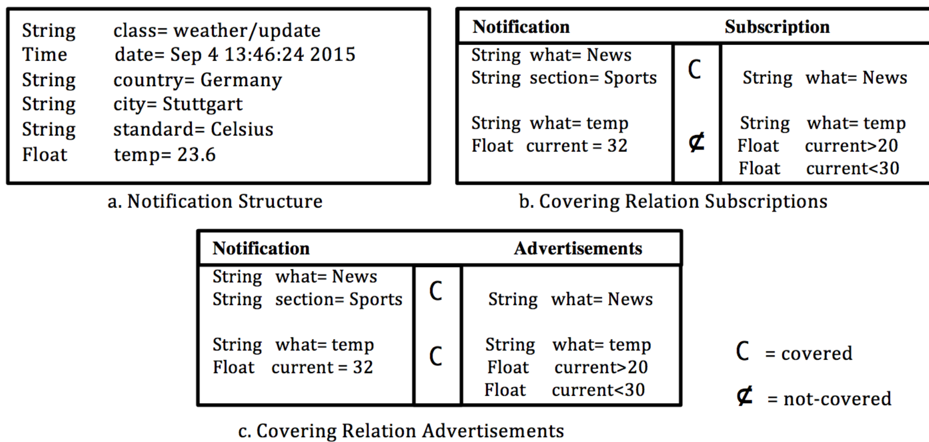

Figure 2.2: SIENA Data Model

SIENA presents two kinds of routing strategies. The first strategy creates the routing paths sending subscriptions in the network, which connects the subscribers with all the servers in a tree structure. The problem with this model is that it will cause the notifications to be delivered to all servers, which will cause unwanted traffic in the system. In the second strategy we make use of advertisements, which are sent in the network through which paths are created between the publishers and all the servers. Now when a subscription is sent in the network a path is only created between the subscribers and relative publishers. In this way the notifications are only sent to the servers in the path from the publisher to the subscriber. SIENA takes advantage of the commonalities between subscriptions by propagating only those request received at the server, which define new selectable notifications.

The SIENA architecture is built over the lower level network infrastructure. It comprises of a interconnect topology for servers and the protocol they use to communicate. There are three different kinds of architectures that can be implanted by SIENA. In the Hierarchical client/server architecture each server can have multiple incoming links from its clients but if there is only one outgoing link towards its master and a sever without an outgoing link is called the root of the architecture. Such architecture is prone to have overloads at the root and the servers close to it. In a peer-to-peer architecture on the other hand all the servers have equal rights. The third architecture presented is the hybrid of the hierarchical

and peer-to-peer architecture.

SIENA presents a very comprehensive solution to the challenges in building a content-based pub/sub system. It uses a data model that provides a very expressive solution while implanting strategies that are easily scalable. The issue is that is implemented as an overlay overt the network infrastructure and also performs expensive comparison operations on the application layer. This greatly affects the end-to-end latency and throughput of the system.

2.3 Conclusion

We have discussed some famous implementations for the pub/sub systems they present some promising features but still have some drawbacks. LIPSIN provides line-rate processing of events at the switches in the network but suffers from low expressiveness and false positives. SIENA presents a solution, which is scale-able, and has high expressiveness but suffers from latency issues as it is completely implemented at the application layer. We now look at a solution that takes a different approach incorporating the concepts of Software Defined Networking (SDN) to provide a content-based solution with line rate performance.

Chapter 3

Towards an ideal Publish/Subscribe System

This thesis builds on the Publish/Subscribe system purposed by PLEROMA [3]. PLEROMA presents a unique solution to meet the requirements of huge pub/sub systems. It incorporates the concepts of Software defined Networking (SDN) to create a Content-Based System that works on the underlying infrastructure providing line rate performance. This chapter first presents the basic architecture of a Software defined Network and then discusses the implementation of the system.

3.1 Software Defined Networking

In a traditional network system the control is distributed throughout the network performed by routers. Such a system can be seen as a closed box architecture where the routers work on predefined control logic and the user cannot program them on the go. Software Defined Networking (SDN) puts forth a new way to build and manage networks. The SDN architecture shown in the fig [3.1] decouples the control plane from the forwarding plane. This allows a central control of the network allowing the underlying infrastructure to be abstracted for applications and network services. Such a scheme is ideal to meet the requirements of the highly dynamic nature of current large applications.

The control plane in the SDN architecture comprises of centralized controllers that have a global view of the complete network. These controllers provide the users with functionality to dictate how the network should react to incoming traffic. The controller uses a northbound interface to communicate with the applications. The API used as the northbound interface depends on vendor providing the controller, for example the FloodLight [16] SDN controller uses the Java API as northbound interface. On the other hand controller uses a southbound interface to communicate with the underlying infrastructure. Open Flow [15] is the most popular standard used as a southbound interface. An additional functionality introduced into the traditional Ethernet switches allows the controllers to communicate with the switches using the open-flow protocol.

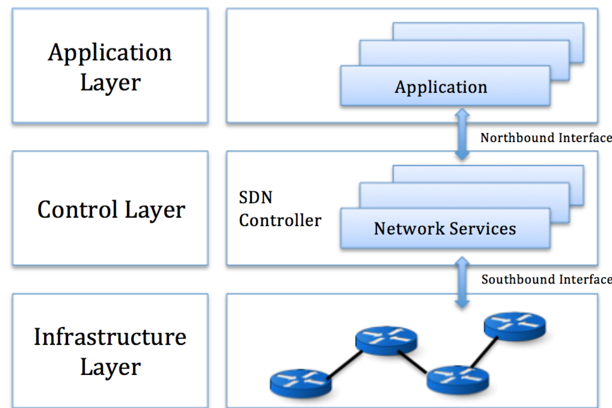

Figure 3.1: The SDN Archetecture

The basic working structure of an open-flow switch is shown in the fig [3.2]. Each switch maintains a flow table, the controllers install flows in these tables to direct the switches how to handle incoming network traffic. The fig [3.2] shows the fields stored by a flow table, the matching filed contains a set of rules that an incoming packet must fulfill to be forwarded i.e. < IP = 192.168.0.1>. The matching rules can also comprise of other attributes like the port, mac address or protocol etc. The action field defines what actions to take after a match has happened i.e. forward to < IP = 192.168.0.2, Port=2>. Finally the priority field tells which flow entry to priorities in case of multiple matches. The matching operation is performed in the switches high speed Ternary Content-Addressable Memory (TCAM).

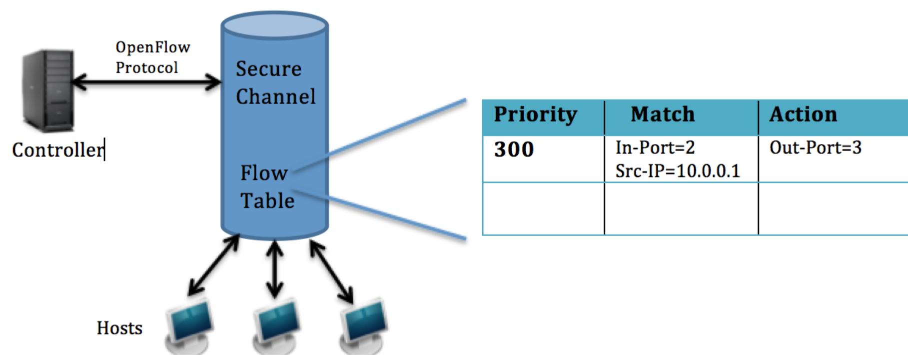

Figure 3.2: Open Flow Switch

Now we have a look at how these SDN concepts are incorporated by the PLEROMA system to develop a content-based system with high expressiveness and throughput.

3.2 PLEROMA

The most vital operations performed by a pub/sub system are event matching and forwarding. While implementing these functions each system faces a trad- off between expressiveness and throughput of the system. Systems with high expressiveness become more and more complex and the throughput suffers. On the other hand simpler solutions are not able to meet the demands on expressiveness by most applications. Most implementations perform these functions at the application layer, which greatly affects the performance of the system. PLEROMA is an event-based middle-ware that incorporates the concepts of Software Defined Networking (SDN) to provide a system with high expressiveness and line-rate performance. The basic structure of PLEROMA is shown in the fig [3.3]. The controller provides a global view of the underlying infrastructure, which allows locating the publishers and the subscribers and creating flows between them. When an event is received at the switch a quick matching operation id performed using the Ternary Content-Addressable Memory (TCAM) of the switches, which is capable of searching its entire contents in one clock cycle. Thus the expensive filtering operations that are usually performed at the application layer are shifted to the hardware layer, which incredibly improves performance.

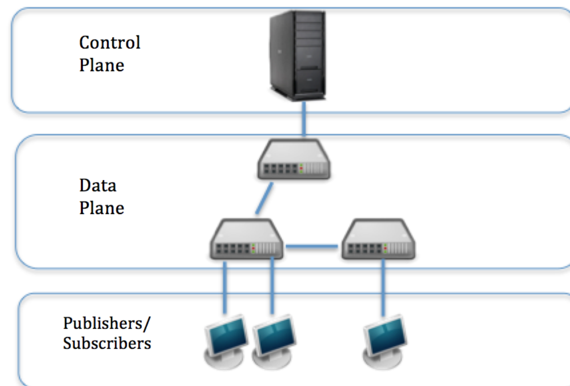

Figure 3.3: PLEROMA Archetecture

3.2.1 The Content Model

Each open-flow switch maintains a flow table maintaining a list of forwarding rules, which are based on the incoming packet header fields like the MAC address, port number, VLAN tag or IP address etc. When an event packet arrives it is forwarded according to these rules. Hence if the event filtering is to be performed at the switch level we need to embed the message content in the packet header.

PLEROMA presents a content-based subscription model based on attribute value pairs, which can be mapped to the packet header fields. The model represents events in multi-dimensional space where each dimension represents an attribute. The event space for each

dimension in turn builds on the concept of spatial indexing, where it is divided into regular subspaces using recursive binary decomposition. Each subspace acts as an enclosing approximation for the subscriptions and advertisements and is represented by a binary string called the dz-expression. The length of the dz-expression depends on the number of times a binary decomposition is performed on the particular event space. The greater the granularity of a subspace the greater will be the length of the dz-expression will grow. A subscription or an advertisement can comprise of multiple dz-expressions. The figure [3.4] shows an example of the content model, there are two dimensions representing the attributes time and temperature. The event space goes through binary decomposition three times resulting in eight subspaces each with a dz length of 3 bits. A subscription or an advertisement for the values <time=[0,25], temp=[0,100]> will be represented by the subspaces <000,010>. This binary representation can be embedded in the packet header, which can be used at the switch to perform event matching.

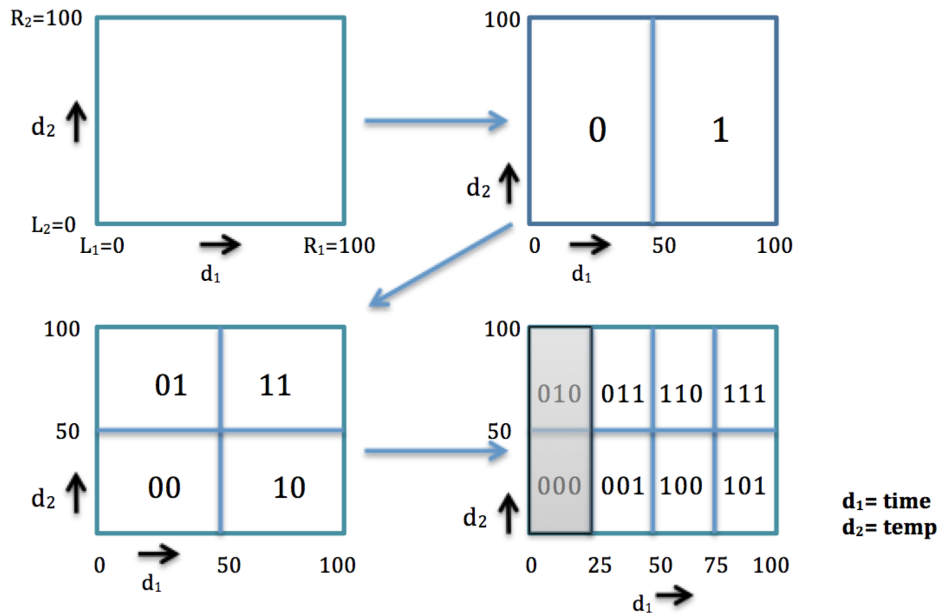

Figure 3.4: Spatial Indexing

The advantage of the content model presented by PLEROMA is the inherent ability of applying containment relation among different subscriptions. For example the subscription space 11 in the example covers the subscription spaces with dz 110 and 111. Hence if a switch already contains a subscription that covers an incoming subscription the controller does not install a new flow, only the action field of the installed flow is updated. This property greatly reduces the network traffic, as the same event is not forwarded multiple times in the network.

3.2.2 Event Matching

An event is represented as a point in the event space so the dz expressions representing the events have greater lengths then the dz expressions for the subscriptions. Taking advantage of the containment property of the dz expressions the events can be matched using pre-fix matching. Hence a subscription matches an event if its dz is the prefix of the dz of the event i.e. it covers the event. For example a subscription represented by the set of dz <000,010> is a match to an incoming event e = 010110 but a mismatch for e = 110100.

3.3 Implementation

The basic structure of the implementation is show in the figure [3.3]. The SDN controller used to implement the system is the floodlight controller, which uses JAVA API as the northbound interface. The open-flow standard is used as the southbound interface.

3.3.1 Network Configuration

The routing of events from the publishers to the subscribers is done via spanning trees. A spanning tree is created over the network of switches in such a way that each switch is covered once. This makes sure that no event is delivered twice to the same subscriber. This scheme limits scalability of the system if only one spanning tree is used. This is the reason the controller maintains multiple spanning trees. Each tree takes a dz expression, which is the prefix to all the dz expressions the tree will store. The number of trees created can be changed by varying the length of the prefix dz expression acting as the prefix for other dz that will be stored on the tree. If a dz of two bits is chosen then four trees with respective dz expressions <00,01,10,11> will be created to cover the entire event space. This partitions the event space into four equal parts, which are then executed on different threads, as these partitions are independent of each other. The root switch of each tree is chosen at random so the trees formed are different from each other .

The fig [3.5] shows an example where the switch 's2' is chosen as the root and the dz expression 01 is chosen as the prefix dz. The respective spanning tree is created and allotted to the partition, which will maintain this tree. Now each request received with a dz expression covered by 01 will be sent to this partition.

3.3.2 Content Delivery Mechanism

As discussed in the section [3.2.1] in order to perform event matching at the open-flow switches we need to map the content into a field in the packet header. This system uses IPv4 addresses to hold the content needed to be exchanged. A fixed range of multi-cast IPv4 addresses (225.0.0.0 – 225.255.255.255) has been reserved to handle all the traffic related to the pub/sub system. A multi-cast is used as there can be many different subscribers wanting the same

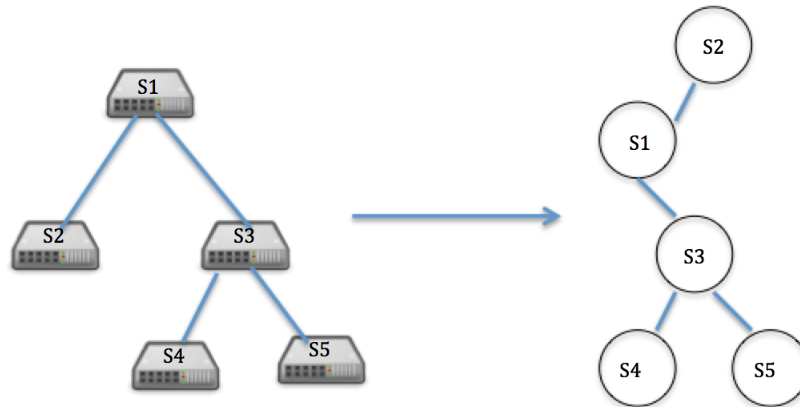

Figure 3.5: Spanning Tree

information from a publisher. This allows the information from the publisher to be sent once and then disperse through he spanning tree to the relevant subscribers. The fig [3.6] shows an example of the IPv4 address. The first 8 bits are fixed to represent that it is a multicast address and the next 24 bits can be used to place the dz expression in the address. A subnet mask is attached to the IPv4 address so that exact length of the dz to be matched at the switch is known. The example shows the representation of the IPv4 address 225.91.0.0/12, which will match the dz 0010 and all the dz expressions covered by it.

11100001 - 01011011 – 00000000 - 00000000

Figure 3.6: IPv4 Address Structure

When a match does not occur at a switch the packet is forwarded to the controller. To inform the controller about a new advertisement or subscription request a fixed IPv4 address IPfix (225.3.70.0) is used.

3.3.3 Publish/Subscribe Request Handling

The controller can receive four kinds of requests namely a subscription, advertisement, un-subscription or un-advertisement. Whenever the controller receives a new request, it processes the dz expression attached with the request and forwards the request to the responsible partition.

Advertisement/Subscription Handling

When a publisher wants to pass the information that it wants to publish some data, it sends out an advertisement request. This request is passed to its responsible partition by the controller and added to the spanning tree maintained by the partition. Then a search is performed on the tree to find all the subscriptions that are interested in the advertisement. If the search comes back empty no further action is taken. On the other hand if one or multiple interested subscriptions are found, a shortest path is created for each subscription from the publisher to the subscriber. Dijkstra's algorithm is used to compute the shortest path between the two parties. Each path is made up of open flow switches with information of the in/out ports of the switches relevant to the path. These paths along with the related dz expression are then provided to the flow handler, which creates flows between the publisher and subscriber using theses paths.

When a subscriber wants to pass the information that it wants to subscribe to some data, it sends out a subscription request. The subscription request is handled in a similar way except that in this case all the interested advertisers are searched and if found paths are established between the subscribers and advertisers and feed to the flow handler.

The Flow Handler

The flow handler receives a path of switches to be established between the publisher and the subscriber. It installs flows on each switch in the path provided. The structure of the flow as explained in the section [3.1] contains the matching field, priority and the out action to be performed. The matching field that is made up of an IPv4 address constructed using the dz expression provided with the path and the in-port value according to the path. The out action can contain the destination IP address and out-port according to the path. The flows also exhibit covering relations as the matching fields are constructed from dz expressions. A flow f1 covers another flow f2 if the dz expression related to f2 is covered by the dz expression of f1. Hence when the a flow handler is requested to install new flows to the switches different cases can occur according to the covering relation among the flows.

For example we consider a case when a path is provided to the flow handler and there are no previous flows installed on the relevant switches related to its dz expression. In such a scenario the flow handler installs the flows on each switch in the path provided. The fig [3.1] shows an example where a publisher has advertised data with dz expression 001. The subscriber 'sub1' subscribes with the dz expression 0010. The flows needed to be installed to form a path between pub and sub1 have no containment relation with the already installed flows on the switches. Hence a flow is added to each switch with the matching field comprising of IP address formed using the dz 0010.

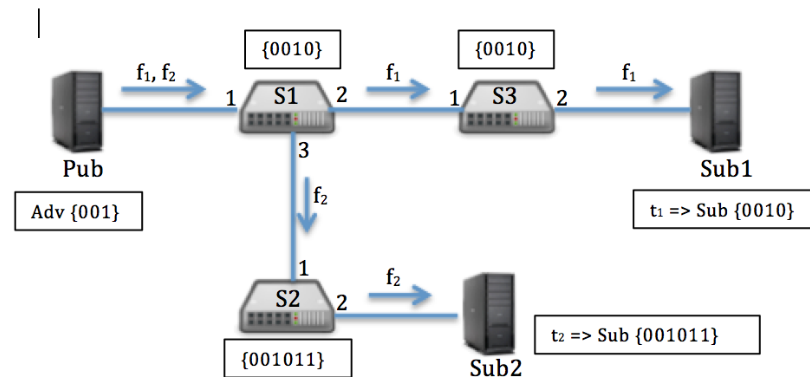

Figure 3.7: Flow Handler Case

Priority	Match	Action
300	In-Port=1 Des-IP=225.32.0.0/12	Out-Port=2,3

Figure 3.8: Flow Table Switch s2

Another case occurs when the flow to be installed at a switch is already covered by an installed flow. In such a scenario no new flow is installed, only the out action of the flows is updated as required. Now if we continue our example in fig [3.1], a new subscriber 'sub2' wants to subscribe with the dz expression 001011. No new flow is added on the switch 's1' as it already has a flow installed i.e. for sub1 that covers the flow for 'sub2', only the action field is update to add the out-port for f2. The flow f2 is added to the other switches in the path from pub to sub1.

Yet another case occurs when a flow to be installed at a switch covers one or more already installed flows. In this case the already installed flows are removed and the new flow is installed with the action field containing the actions of all the involved flows. The fig [3.9] shows that a path between the publisher 'pub' and a subscriber 'sub1' with the dz expression 11011 is installed at time t1. Now at time t2 a new request arrives to install a path between the publisher 'pub' and subscriber 'sub2' with dz 110. The new flow f2 to be installed at the switch 's1' covers the already installed flow f1. Hence f1 is removed and the flow f2 is installed with action field containing the actions of both f1 and f2.

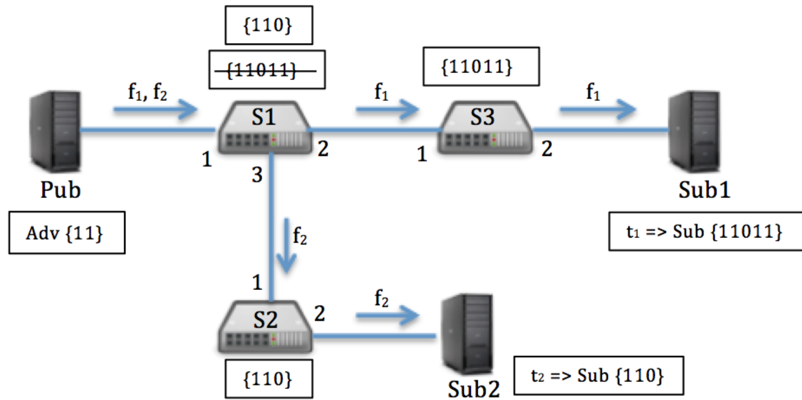

Figure 3.9: Flow Handler Case

Priority	Match	Action
300	In-Port=1 Des-IP=225.192.0.0/11	Out-Port=2,3

Figure 3.10: Flow Table Switch s2

Un-Advertisement/Subscription Handling

When a publisher needs to pass the information that it will no longer publish some data, it sends out a un-advertisement request. This request is passed to its responsible partitions by the controller. The un-advertisement process involves recursively moving from the publisher towards each subscriber that subscribed to it in a depth first search manner. All the relevant flows in the path are either deleted or downgraded. Down-gradation happens when the flow to be removed covers other flows at the switch that belong to some other publisher. The fig [3.11] shows an example where the subscriber 'sub1' has subscribed to the publishers 'pub1' and 'pub2' while the subscriber 'sub2' has subscribed for the publisher 'pub1'. When 'pub1' sends out an un-advertisement request the flows at the switches 's1', 's2' and 's3' are downgraded to the flow between 'pub2' and sub1' while the flow at switch 's4' is removed as it has no covering relation with any other flow.

When a subscriber wants to pass the information that it no longer wants to subscribe for a particular data it sends out a un-subscription request. This request is passed to its responsible partitions by the controller. The un-subscription process is very similar to the un-advertisement. It involves recursively moving from the subscriber towards each publisher that publishes data to it in a depth first search manner. All the relevant flows in the path are either deleted or downgraded.

The fig [3.12] shows an example where the subscribers 'sub1' and 'sub2' have subscribed to the publisher 'pub'. Now 'sub1' sends a un-subscription request resulting in the deletion of

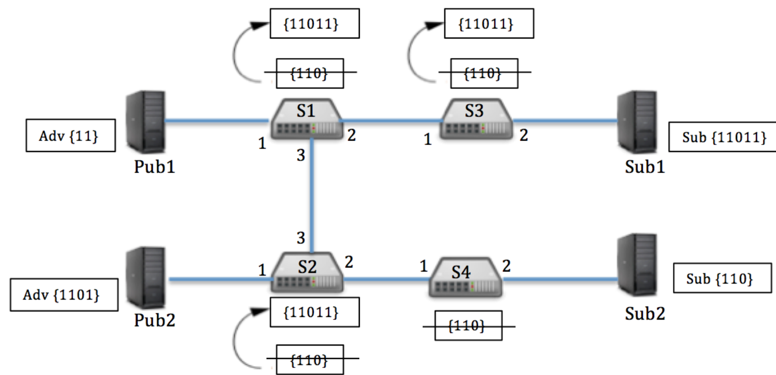

Figure 3.11: Un-Advertisment

the flow at switch 's3' and down-gradation of the flow at the switch 's2' as it covers the flow between 'sub1' and 'pub'.

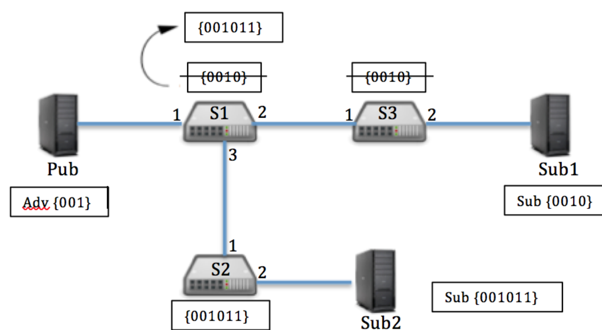

Figure 3.12: Un-Subscription

3.4 False Positives

The content model presented by PLEROMA gives a very efficient solution to represent the content of a publish/subscribe system in an expressive way at the network layer. This gives PLEROMA an edge on the other state of the art implantations as it can provide line rate performance while providing the expressiveness of a content-based filter as compared to the topic based filter i.e. in LISPIN. But this representation of the content leads to formation of false positive in the network. A false positive is an event that is sent to a subscriber who has not subscribed for the content contained in the event. The content model of PLEROMA suffers with the generation of these false positives during its normal operation. For example consider a publisher has advertised content with he the dz-expressions 000,010 having the attributes the attributes time= [0,100], pressure=[0,50]. Now a subscriber 'sub' subscribes to this publisher with the dz-expressions 0001,01011 having the attributes time= [0,65], pressure=[0,30]. The publisher generates an event with dz-expression 0000110001 having the attributes time= 55,

pressure=45. This results into a false positive in the network as this event is forwarded towards the subscriber 'sub' by the open flow switches. This behavior not only affects the performance but also is undesirable by the subscriber. The number of false positives can be decreased by decomposition of the event space into finer granularity subspaces. This results in the increase in the length of the dz-expressions as shown in the section (3.2.1).

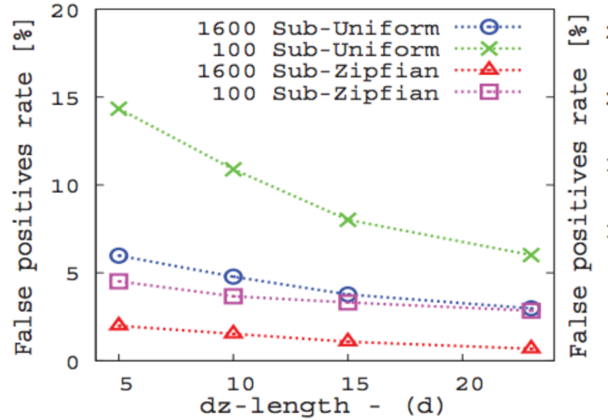

Figure 3.13: PLEROMA: False Positives

The fig [3.13] shows results published by PLEROMA showing the number of false positives in relation to the dz- expression length of the subspaces. It can be seen that the false positives fall with the increasing length of the dz-expression. But we are limited in the length of the dz that can be mapped to an IP address, like an IPv4 address can hold a dz of maximum 24 bits. The work presented in this thesis is to present an upgrade to the PLEROMA pub/sub system that does not suffer from the problem of false positives.

3.5 Problem Statement

Many different implementations of the Publish/Subscribe systems have been presented over the years trying to develop an ideal solution. SIENA presents a very powerful content-based model providing a high expressiveness but its performance suffers as it is implemented on the application layer. On the other hand LIPSIN provides line rate performance as it performs event filtering at the network layer but provides low expressiveness because of its topic-based model structure. PLEROMA puts forth a system that provides a solution with both high expressiveness and line-rate performance; still the system faces the problem of false positives. This thesis works on providing a solution that in-cooperates the desirable properties of PLEROMA and at the same time does not suffer from generation of false positives in the network. This basic concept is to develop a Hybrid system by moving the event filtering operation for the subscriptions causing false positives to the application layer. In this way the false positives can be reduced by more powerful filtering operations available at the application layer. This thesis presents the design and implementation of a working Hybrid pub/sub system and discusses its performance as compared to PLEROMA.

Chapter 4

Hybrid Publish/Subscribe System

The main purpose of the Hybrid Publish/Subscribe system is to provide a mechanism that will build on the model presented by PLEROMA to reduce the false positives generated in the network. PLEROMA suffers from the generation of false positives due to the limitation of filter operations at the network layer. If the filtration process is shifted to the application layer strong filter operations can be applied on the incoming events to remove false positives. But processing each event will lead to considerable decrease in the performance of the system. The hybrid pub/sub system provides a solution for this by providing a balance between the use of the network layer and the application layer. The idea is to process the events that are causing false positives at the application layer while all other events are processed at the network layer. Hence the hybrid system provides performance that in-between the performance provided by a complete network layer and application layer implementation respectively.

The hybrid pub/sub system is divided into of three planes namely the control plane, application plane and the data plane. The data plane consists of all the interconnected open-flow switches in the network. The publishers and the subscribers are directly connected to the data plane. The events generated by the publishers are forwarded to the subscribers according to the installed flows in the switches. The control plane is responsible for serving requests from the publishers and the subscribers. It consists of a SDN controller that has a global view of the underlying network and establishes connections between interested parties by installing flows on the open-flow switches. The flow installation process here is the same as done by PLEROMA discussed in the section [3.3.3]. The application plane on the other hand is responsible to process events that have been marked as possible false positives. The application layer also comprises of a SDN controller and can make use of all its functions.

This chapter discusses the design of the hybrid pub/sub system based on the model presented by PLEROMA. The fig [4.1] shows the structure of the hybrid pub/sub system it includes an application layer as an upgrade to the PLEROMA system. When the system first comes on line it operates as a normal PLEROMA pub/sub system. The controller discovers the underlying network topology and creates partitions each maintaining its own spanning tree according to their allotted dz-expressions. Then it waits for advertisement/subscription requests and sets up flows in the open-flow switches as explained in the section [3.3.3]. Now when a subscriber sub1 starts receiving false positives it informs the controller. As a result controller sets up the flow tables in the switches in such a way that the events related to

the subscriptions causing false positives at sub1 are directed towards the application layer. When the application layer receives events they undergo filter operations and only the events relative to the subscriber sub1 are forwarded towards the subscriber.

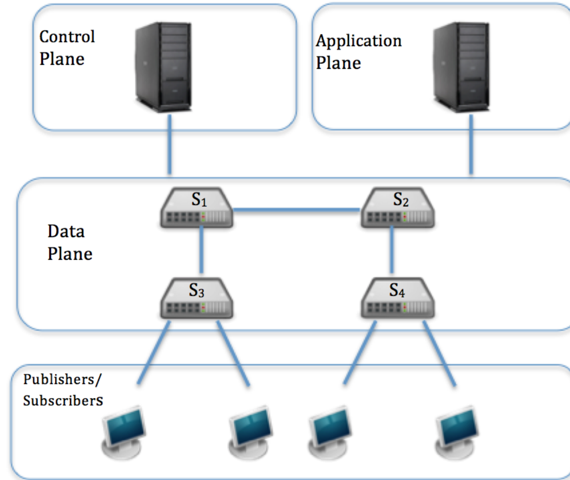

Figure 4.1: Hybrid Publish/Subscribe Structure

Thus the Hybrid system reduces the generation of false positives in the network. As the events for the subscriptions with possible false positives are processed at the application layer, it results in the increase in the end-to-end delay for these events. The events related to all the other subscriptions are filtered through the network layer and have line-rate performance. Hence it is important to achieve a good balance of events distribution among the application and network layer to get maximum performance with reduction of false positives. Now we discuss the design requirements related to the hybrid pub/sub system and how they are met.

4.1 Design Issues

The Hybrid pub/sub system provides an enhancement to the PLEROMA model to prevent the generation of false positives in the system. The goal is to design such a system that can provide the powerful filtering operations of the application layer but still provide performance that is better then an all application layer based system. Hence a nice balance is to be established on when to use the normal PLEROMA operations with line rate performance and when to shift to application layer for high expressiveness. We here have a look at some important design requirements for such a system.

* All the information regarding the false positives must be gathered and provided to the main controller in an efficient way.

* A mechanism should be created at the main controller to select the most suitable subscriptions in terms of performance form the provided list.
* The subscriptions with false positives must be treated in such a way that they don't affect the other subscriptions.
* All the necessary information regarding the selected subscription must be provided to the application layer controller.
* The application layer controller must store the subscription information in an efficient data structure that allows fast search operations.
* The application layer controller should implement mechanisms to filter out false positives and from incoming events and forward the rest to their related switches.
* Removal of the subscriptions with false positives from the system on subscriber request.

The rest of this chapter is dedicated to explain how these requirements are fulfilled to build the hybrid pub/sub system.

4.2 The Control Plane

The responsibility of the control plane is to serve the incoming requests from the publishers and subscribers. There are five types of requests that can be received by the control plane namely Advertisement, Un-Advertisement sent by the publisher and Subscription, Un-Subscription and False Positive sent by the subscriber. The two requests Advertisement and Subscription are served as done by PLEROMA explained in the section [3.3.3]. While the requests Un-Advertisement, Un-Subscription and False Positive are handled according to the extended functionality of the Hybrid pub/sub system. We now see how the control plane serves these requests unique to the hybrid pub/sub system.

4.2.1 False Positive Request

The False Positive request is sent by the subscriber and contains the subscription receiving the false positives, the total number of false positives received for this subscription and the end-to-end delay for the subscription acceptable by the subscriber. When the control plane receives the False Positive request it has to makes the decision weather the related subscription can be served via the application layer. The selection process takes place by passing the received requests through an Integer-Linear-Problem solver. Which then at the output provides a list of the subscriptions that can be served via the application layer. After selection the control plane performs the necessary flow modifications to direct the events related to the subscription towards the application layer. Then the control plane registers the subscription at the application layer by communicating the information related to the subscription. .

Subscription Selection

The most important decision to be taken by the Hybrid pub/sub system is that which subscriptions with false positives are to be registered at the application layer controller. The control plane starts the selection process on a separate thread when it first comes online. The Selection process periodically looks for any False Positive requests from the subscribers. After reading the requests they are passed through an Integer Linear Problem (ILP) Solver, which provides a list of the subscriptions that will be processed via the application layer in the future.

The inputs provided to the solver are the list of subscribers, their related false positive count and desired end-to-end delay. The overlay delay, which is the delay experienced by the events through the application layer and the underlay delay, which is the delay, experienced by the events through the network layer are also provided to the solver. The ILP is formed as following:

$$
\begin{aligned}
&\text{For Subscriber set S} \\
&x_i = 1 \qquad \text{if filter at appliction layer} \quad S_i \in S \\
&x_i = 0 \qquad \text{if filter at network layer} \quad S_i \in S \\
&\min \qquad \textstyle\sum_{i=1}^{n} (x_i) * fp_i \\
&\text{s.t.} \\
&\qquad O_d * x_i + U_d * (1 - x_i) \leq \delta_i \quad \forall S_i \in S \\
&\qquad FP_T \leq fp_i \quad \forall S_i \in S
\end{aligned}
\tag{4.1}
$$

The number of variables created for the ILP problem is equal to the number of subscriptions in the set S. The Integer linear problem is setup in such a way that a variable xi related to the subscription Si is equal to '1' if it should be filtered at the application layer. On the other hand if the variable xi is equal to '0' it should be filtered at the network layer. The objective function for the ILP is to minimize the false positives in the network. The constraints applied on the objective function are also equal to two times the number of subscriptions in the set S. A constraint ci on the objective function states that for the variable xi to be '1' the overlay delay should be less then or equal to the desired end-to-end delay provided by the subscriber. Else the variable xi will be set to '0' and in this case the underlay delay should be less then or equal to /delta, which will always be true. This constraint makes sure to prefer the subscribers who are not affected in term of delay by moving the filter operation to the application layer. The constraint c2 applied on the objective functions states that the total false positives for a subscription must be greater then a threshold value FPT .The value of FPT can be set by at the start as per the requirement of the system.

Subscription Registration

Once the ILP solver provides the list of the subscriptions the next step is to communicate this information to the application layer controller and install the required flows on the open-flow switches. When the selection process sends back a subscription to the main controller to be moved to the application layer. The controller processes the subscription and sends it to the responsible partition depending on it dz-expression. The main difference of the partitions maintained in the hybrid pub/sub system from PLEROMA is that they maintain two trees, one Tree_{N} for the normal subscriptions and the other Tree_{FP} for the subscriptions with false positives. The trees maintain a list of exiting flows installed that are used to look up covering relations while installing flows. Each partition has two trees so that no covering operation is performed during installation of flows between the network layer subscriptions and the application layer.

Algorithm 1 **write something here**

```
procedure ADDNODE(char[] dz, final List<Integer> values)
    AdvList ← findAdv(sub) { find the interested advertisements }

    for each adv in AdvList Do

            sendMsg(sub,adv){ send to the application layer controller}

            unSubscribe(sub,Tree_n) { delele all the relative flows }

            addSubscription(sub,Tree_n){add to tree with false positives}

            addAdvertisment(sub,Tree_fp)

            path ← findpath(sub, adv, Tree_fp)

            addFlows(path,Treefp,VLAN-ID) {add flows with VLAN}
    end for
    RemoveSubscription(sub,Tree_n) {remove from the tree
                        for subscriptions without false positives}
end procedure
```

When the partition receives a subscription with false positives it performs a search operation in the Tree_{N} to find all the interested advertisements for the subscription. Then for each advertisement a message is created which includes the subscription and the related advertisement and is sent the application layer controller. After that an unsubscribe operation as explained in the section [3.3.3] is performed on the subscription. Now the subscription is added to the Tree_{FP} along with its advertisement after which a shortest path between the advertiser and the subscription is calculated using the Tree_{FP}. Then flows are installed on

the relevant switches in the path using the same process as explained in section [3.3.3]. The difference from the flows installed for subscriptions served via the network layer is that a new entry called the VLAN id is added to the match field. The VLAN tag is used to differentiate between the flows installed on the switches for the application and network layer. Hence it is ensured that no unwanted traffic is developed in the system. The same process is repeated for each interested advertisement and finally the subscription is removed from the $Tree_N$. The algorithm [1] also shows the process of subscription registration.

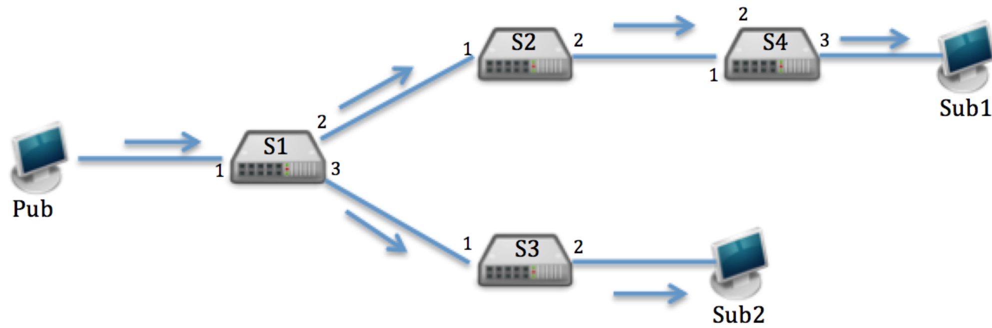

Figure 4.2: Flow with VLAN tag

The fig [4.2] shows an example where initially the subscribers sub1 and sub2 have subscribed for the same data from the publisher pub. After some time the subscriber sub2 informs the Control Plane that it is experiencing false positives and hence the subscription related to sub2 is moved to the application layer. The flows installed in the open-flow switches for sub2 are removed and new flows with a VLAN tag are installed. The fig [4.3] shows the flow table entries for the switch S1 before and after moving the subscription for sub2 to the application layer.

4.3 False Positive Request Generation

4.3.1 Un-Advertisement/Un-Subscription Handling

The process of performing the un-advertisement and un-subscription is similar to what is done in the PLEROMA model as discussed in the section [3.3.3]. But as each partition in the hybrid pub/sub system maintains two spanning trees, the subscriptions and advertisements are to be removed from both the trees.

When the control plane receives a un-advertisement request from the publisher, it is passed to the responsible partition where un-advertisement is performed by recursively moving from the publisher towards each subscriber that subscribed to it in a depth first search manner. All the relevant flows in the path are either deleted or downgraded. Down-gradation happens when

Priority	Match	Action
300	In-Port=1 Des-IP=225.192.0.0/11	Out-Port=2,3

↓

Priority	Match	Action
300	In-Port=1 Des-IP=225.192.0.0/11	Out-Port=2
300	In-Port=1 VLAN-ID = 2256 Des-IP=225.192.0.0/11	Out-port=3

Figure 4.3: Flow table entry with VLAN tag

the flow to be removed covers other flows at the switch that belong to some other publisher. The process is similar to the one explained in the fig [3.11] in the section [3.3.3]. The same process is repeated for both the spanning trees maintained by the responsible partition.

Similarly when a un-subscription request is received by the control plane the same procedure is followed with the only difference is that in this case we recursively move form the subscriber to each of its relative publishers in a depth first search manner. This process is shown in the fig [3.12] in the section [3.3.3].

4.4 False Positive Request Generation

One important design parameter is the False Positive detection in the network and then passing the information to the control plane. The subscribers are responsible to provide the control plane with information about the subscriptions that are causing false positives at their end. Each subscriber passes this information periodically to the controller; the period can be adjusted according to the need of the subscriber. When a subscriber first comes online it sends out subscription requests to the controller and then turns on its listener functionality to listen for incoming events. When events are received they are processed and if a false positive is detected the related subscription is added to a queue. This queue is then sent to the controller.

4.4.1 Listener

In order to process and verify the incoming events the listener maintains a data structure that contains all the subscriptions of the subscriber. To be able to perform quick search operations a binary tree data structure is used to store the subscriptions. The generated tree can be

balanced or skewed depending on the subscription values. Hence we get a best-case time of O(1) and worst-case time of O(n) but as mostly the search space is cut into half for each tree level we get an average search time of the O(log n). To improve the performance multiple numbers of binary trees can be created, each representing a subspace in the complete event space. For example if a root with dz-expression of two bits is selected, four trees are created with the root dz 00,01,10,11 respectively. Each tree is maintained by a separate partition executed by different threads.

Algorithm 2 Addition of Subscription Node

```
procedure ADDNODE(char[] dz, final List<Integer> values)
    focusNode ←← root {start from the root of the respective tree}
    dz.remove(root.dz) {remove the root dz bits from the dz-expression}
    currentdz ← root.dz {value of the dz-expression at the current node}
    for each bit in dz Do
      if bit == '0' then
        focusNode ← focusNode.leftChild
        currentdz ← currentdz + bit
        if focusNode == null then
            focusNode ← newnode(null, currentdz) {create an empty
            node in the path}
        end if
      end if
      else if bit == '1' then
        focusNode ← focusNode.rightChild
        currentdz ← currentdz + bit
        if focusNode == null then
            focusNode ← newnode(null, currentdz) {create an empty
            node in the path}
        end if
      end if
    end for
    if focusNode == null then
    focusNode ← newnode(values, currentdz) {add the subscription
        to this node}
end procedure
```

4.4.2 Binary Tree Generation

When the listener is started it brings the partitions online and starts them on different treads. Each partition creates a binary tree with its allotted root dz value and adds the related subscriptions to the tree. The fig [4.4] shows an example of a binary tree created by a partition with the root dz value 01. A new subscription is added to the tree by starting at the root and moving down node-by-node reading the dz-expression of the subscription, one

bit at each node. If the bit read at the current node is equal to ‘0’ we move down towards the left child of the node. On the other hand if the bit read at the current node is equal to ‘1’ we move down towards the right child of the node. This process is repeated equal to the length of the dz-expression of the subscription. The tree node finally reached represents the subscription and is used to store the subscription data. Algorithm[2] also shows how a new subscription node is added to a binary tree.

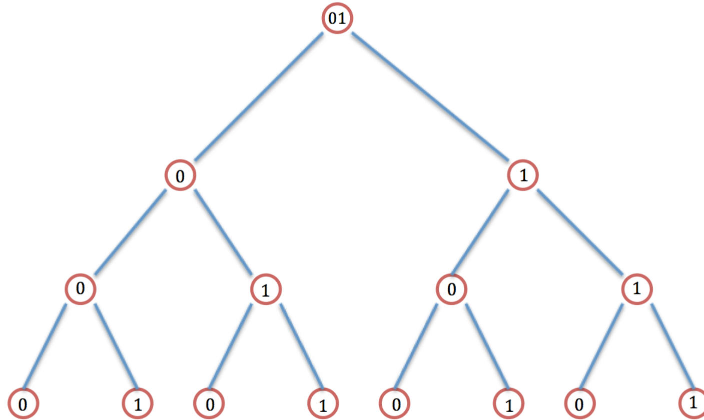

Figure 4.4: Binary Tree

4.4.3 Matching

When an event is received at the subscriber the listener passes it to the responsible partition where it is processed. The dz-expression of the event along with its attribute value pair is passed to the binary tree to perform a matching operation. Pre-fix matching is used to match the incoming event to one of the installed subscription in the tree. The tree is traversed node-by-node by reading the dz-expression of the event, one bit at each node. Similar to the subscription addition if the bit read at the current node is equal to ’0’ we move down towards the left child of the node. On the other hand if the bit read at the current node is equal to ’1’ we move down towards the right child of the node. This process is repeated until we reach a node that is not empty. For example the fig [4.5] shows an example where the event with dz-expression 010011011 is matched to a subscription 010011. After finding the node responsible for storing the subscription for the received event, the attribute-value pairs of the event are compared to that of the subscription. If the event does not fall in the desired range of the subscription it is marked as a false positive and the subscription is added to the queue to be sent to the controller.

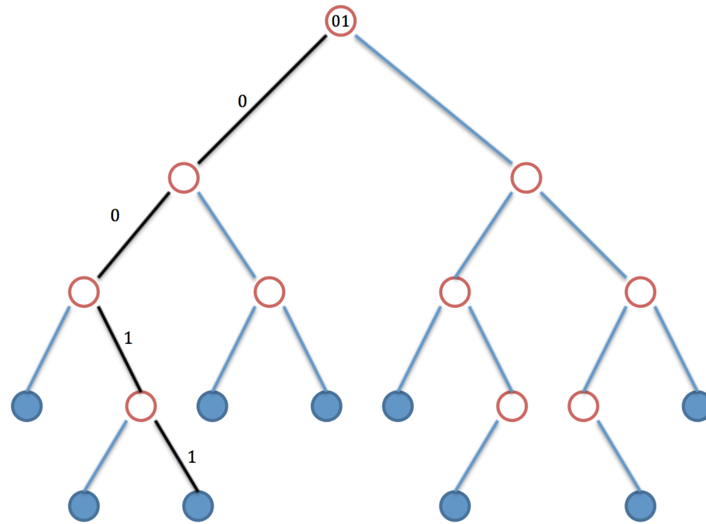

Figure 4.5: Event Matching

4.4.4 Communicated Data

The information communicated to the controller by the subscriber includes the subscription data, the total false positives received for this subscription and the desired end-to-end delay. Moving the filter operation from the network layer to the application layer for a subscription increase the end-to-end delay for the events. Thus the subscribers can additionally provide the information regarding their desirable end-to-end delay for events related to a subscription. This information is used by the controller in the process of subscription selection for diverting their events to the application layer controller.

4.5 The Application Plane

The application plane comprises of an SDN controller and can perform all the functions provided by it. When the application plane first comes online it establishes a connection with control plane and requests the spanning trees created by the control plane for each of its partitions. When the spanning trees are received the application plane creates partitions and assign each tree to its responsible partition. After this the application starts listening for incoming requests. There are two kinds of requests that can be received at the application plane. One is the False Positive registration request that is sent by the control plane and the other is the Event request related to the registered subscriptions.

4.5.1 Registration Request

The registration request is sent by the control plane to the application plane to pass the information about the subscriptions whose events will be forwarded to the application plane. When the request is received at the application plane, it is passed to its responsible partition according to the attached dz-expression. Now the message is processed and the subscription and the relative advertisement are added to the spanning tree maintained by the partition. A shortest path is calculated using the Dijkstra's algorithm between the advertisement and the subscription. This path information is added to the subscription data and the subscription along with its dz-expression is then passed forward for storage.

Algorithm 3 Addition of a Subscription

```
procedure addSubscriber(char[] dz, final Sub)
    focusNode ← root {start from the root of the respective tree}
    dz.remove(root.dz) {remove the root dz bits from the dz-expression}
    currentdz ← root.dz {value of the dz-expression at the current node}
    for each bit in dz Do
        if bit == '0'  then
            focusNode ← focusNode.leftChild
            currentdz ← currentdz + bit
            if focusNode == null then
                focusNode ← newnode(null, currentdz) {create an empty
                node in the path}
            end if
        end if
        else if bit == '1'  then
            focusNode ← focusNode.rightChild
            currentdz ← currentdz + bit
            if focusNode == null then
                focusNode ← newnode(null, currentdz) {create an empty
                node in the path}
            end if
        end if
    end for
    if focusNode == null then
        focusNode ← newnode(Sub, currentdz) {add the subscription
            to this node}
    else
      focusNode.add(Sub, currentdz) {add subscription to
            the exiting node this node}
end procedure
```

Subscription Storage

In order to perform filter operations on incoming events the relative subscriptions are needed to be stored in an efficient data stricture with fast matching times. The data structure chosen to store the subscriptions at the application layer controller is the Binary tree data structure similar to the one use for the subscriber listener. The binary tree data structure provides an average search time of the O(log n) where n is the number of subscriptions installed. Each partition has it owns binary tree and the root of the binary tree is equal to the allotted dz-expression of the partition. When a request to store a subscription is received the binary tree is traversed node-by-node reading the dz-expression of the subscription, one bit at each node. We start at the root and if at the current node the dz bit is equal to '1' we move down to the right child of the node and if the bit is equal to '0' we move down to the left child of the node. This process is repeated equal to the length of the dz-expression. The tree node finally reached is processed and if it is empty a new node is created and the subscription is added to it and if there is already a node present the subscription is just added to the node. The storage process of the subscription here is similar to the one performed by the listener of the subscriber as shown in section [4.2.2]. The main difference is that at the application layer a node can contain multiple subscriptions installed at one node from different subscribers. The algorithm [3] shows the process of adding the subscription to the binary tree data structure.

4.5.2 Event Request

The events are generated by the publishers and forwarded to the application layer according to the flows in stalled in the open-flow switches by the control plane. The events forwarded to the application plane are related to the subscriptions that are receiving false positives. When an event request is received at the application plane it is first matched to the subscriptions stored and then filter operations are performed on the ta contained within the event and the relative subscription. Then the filtered events are forwarded towards their destination subscribers by the application layer.

Event Matching

When an event is received at the application layer it is passed forward to the responsible partition according to its attached dz-expression. At the partition the event is passed through the Binary tree maintained by it to get the list of subscriptions interested in the event after filtering out the false positives. The tree is traversed node-by-node reading the dz-expression of the event, one bit at each node. We start at the root and if at the current node the dz bit is equal to '1' we move down to the right child of the node and if the bit is equal to '0' we move down to the left child of the node. Due to the covering relation property of the dz-expressions all the subscriptions with a prefix match to the event are possible contenders to receive the event. So at each node that is traversed if there are subscriptions present at the node they are matched to the event and if the event falls in the desired range of the subscription, only then the subscribers related to the subscription is added to the list to be returned. This

process is repeated equal to the length of the dz-expression of the event. At the end the list of interested subscribers is returned to the partition. The process of event matching is shown in the algorithm [4].

Algorithm 4 Match Event

```
procedure addSubscriber(char[] dz,event)
    focusNode ← root {start from the root of the respective tree}
    dz.remove(root.dz) {remove the root dz bits from the dz-expression}
    currentdz ← root.dz {value of the dz-expression at the current node}
    for each bit in dz Do
        if bit == '0' then
            focusNode ← focusNode.leftChild
            currentdz ← currentdz + bit
            if focusNode != null then
                subList ← node.subList {get all subscriptions at the node}
                for each sub in subList Do
                    if sub.compare(event) == 'true' then
                        Subscriptions.add(sub) {add to the list to be returned}
                    end if
                end for
        end if
        else if bit == '1' then
            focusNode ← focusNode.leftChild
            currentdz ← currentdz + bit
            if focusNode != null then
                subList ← node.subList {get all subscriptions at the node}
                for each sub in subList Do
                    if sub.compare(event) == 'true' then
                        Subscriptions.add(sub) {add to the list to be returned}
                    end if
                end for
        end if
    end for
    return Subscriptions
end procedure
```

Event Forwarding

When the Control Plane forwards a subscription to be installed at the application layer it create paths from the subscription its relative advertisers and installs flows on the open-flow switches according to each path. The flows are tagged with a VLAN id in order to differentiate them from the flows added for the events processed at the network layer. The application layer uses the Packet Out functionality of the SDN controller to create packets containing the

event data and tags them with the VLAN id se they only match to the flows installed for events at the application layer.

Once the list of the interested subscribers for the event is returned to the partition the next step is to forward the event towards the subscribers. This step will greatly affect the performance of the particular subscription and will vary according to the placement of the subscribers in the network. Hence we present two different scenarios that can be used to forward the events towards the related subscribers. The first possible solution is to send the event directly to the respective subscriber. In such a technique the event will directly be passed to the switch directly connected to the subscriber and will provide fast calculation times of the switch in question. The second scenario is to push the event to a common switch in case of multiple interested subscribers in the event. In this case the calculation time for the common switch will be greater as many cases will develop for different situations. We now see how these two forwarding mechanisms can be developed.

Direct Forwarding

In the first scenario when interested subscribers for an event are found, a packet is created for each subscriber containing the data of the event and forwarded directly to the switch to which the subscriber is connected. The fig [4.6] shows an example where the subscribers sub1 and sub2 have subscribed for the same data from the publisher pub. Both the subscribers are experiencing false positives. After relying this information to the control plane the subscription related to the subscribers is moved to the application layer. Now when the publisher generates an event related to the subscription it is forwarded to the application plane from the switch S1. At the application plane the event is filtered against the subscriptions from sub1 and sub2 and it is found that only sub1 is interested in the event. Hence a packet containing the event data is created and sent out directly to the switch S4 connected to the subscriber sub1. In the case both sub1 and sub2 wanted the event then two packets will be created and forwarded to the switches S3 and S4 respectively.

The packet forwarding process is very fast in this scenario, as no processing is required to find the switch the packer must be forwarded to. Also as the packet is forwarded directly to the last switch many switches in the path can be bypassed. But for each subscriber a different packer is created and this could be undesirable in some cases so we come to the second possible implementation for event forwarding at the application layer.

Mutual Forwarding

The second Possible implementation for event forwarding is to find mutual switches for multiple interested subscriber and sending just one packet to the mutual switch. The packet will then be delivered to each subscriber following the flows installed by the Control plane in the open-flow switches with the VLAN id. Many different cases arise for this implementation, as there can be also one or multiple subscribers who are not interested in the event. In this the mutual switch should be selected in such a way that no uninterested subscriber gets the

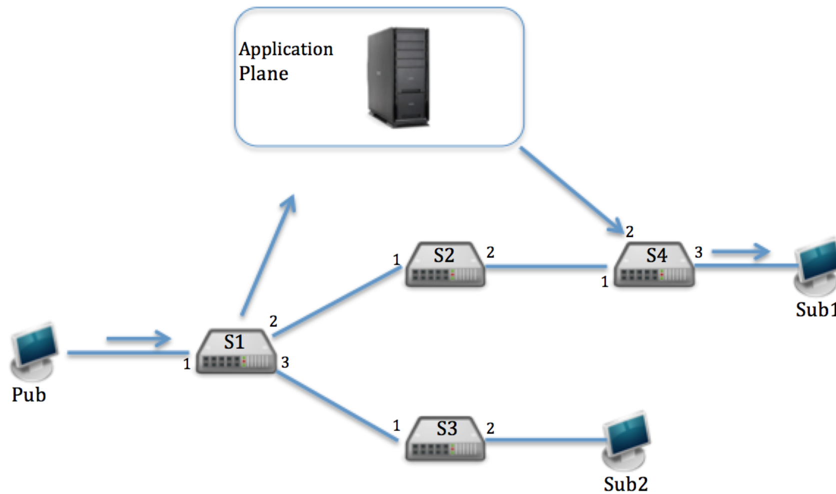

Figure 4.6: Direct Forwarding

event.In order to find the mutual switch among multiple subscribers the path information stored with each subscription at the application plane is used. The path is basically an ordered set of all the switches between the publisher and subscriber for a particular subscription. The first step in finding the mutual switch is to create two union sets, the first one is the union of all the path sets belonging to the subscribers who are interested in the event and the second, which contains all the path sets of the subscribers who are not interested in the event. If fig [4.7] is taken as an example and the subscribers Sub1 and sub2 are interested in the event and the subscriber sub3 is not interested in the event, the sets are formed as following:

* $SetInterested$ = S1, S2, S3
* $SetUninterested$ = S1, S4

Based on the content of these sets four different cases can arise.

Case 1

If the set belonging to the set of all subscribers interested in the event is empty then it means that there are no interested subscribers in the event and there is nothing to be done further.

Case 2

If the set belonging to the set of all subscribers uninterested in the event is empty then we take an intersection of the path set of each subscriber interested in the event and the packet is then sent to the last common switch in the set. As these are ordered sets so we choose the

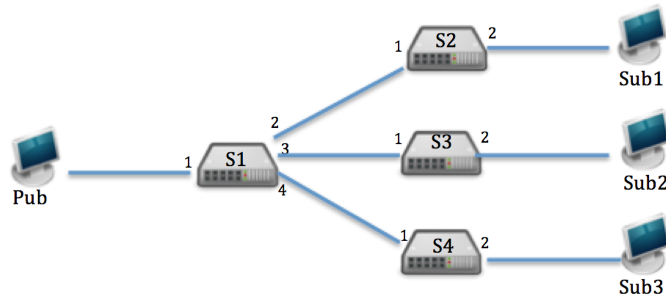

Figure 4.7: Mutual Forwarding

last common switch bypassing the switches behind this switch. The fig [4.8] shows an example for such a scenario. Where a subscription for the subscribers sub1 and sub2 is registered at the application layer. Now when an event related to this subscription comes at the switch S1 it is forwarded to the application layer where after the filter operations it is found that sub1 and sub2 are interested in the event. In this case the set of all subscribers uninterested in the event will be empty. Hence an intersection of the path sets of sub1 (S1, S2) and sub2 (S1, S3) is taken and we get a switch with one entry i.e. the switch S1. The packet is forwarded to S1 from where it is delivered to sub1 and sub2 using the flows installed by the Control Plane with the VLAN tag.

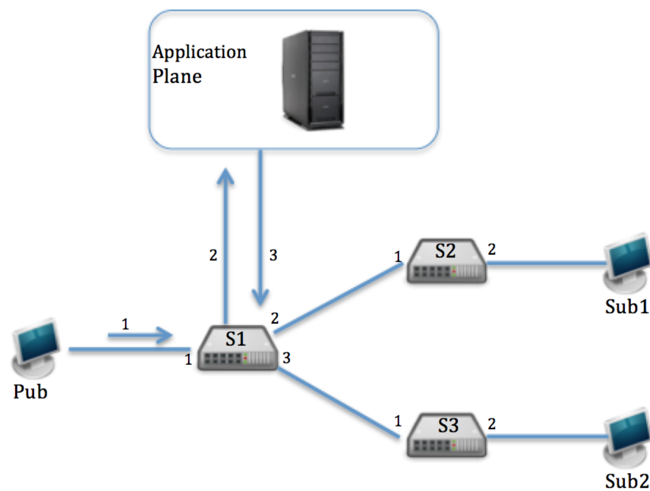

Figure 4.8: Mutual Forwarding Case 2

Case 3

If both the sets are found to be non-empty then a common switch is to be found such that the subscribers who are not interested in the event do not receive the event. The first step is to remove all the switches in the union set of the uninterested subscribers from the union set of all interested subscribers. Now this resulting set is used to perform intersection operations with the set of path switches of each of the interested subscribers. Finally the first switch in the resulting sets is used to forward the packet. As the sets are ordered, if there is any viable common switch among the subscribers it is automatically selected by choosing the first switch in the set in the previous step. If the intersection with the path set results in an empty set, this means that there is no possible common switch between the interested subscribers. In this case the packet is forwarded to the switch directly connected to the subscriber as shown in the Direct Forwarding. The fig [4.9] shows an example where the subscribers sub1 and sub2 are interested in the event and the subscriber sub3 does not want the event. Now the union set of uninterested subscribers S1, S3 is removed from the union set of wanted switches S1, S2. The resulting set S2 is used to perform intersection operation with the path sets for sub1 S1, S2 and sub2 S1, S2. In both the cases we get a set with the first entry as switch S2 which is the viable common switch in the case and the packet is forwarded to this common switch.

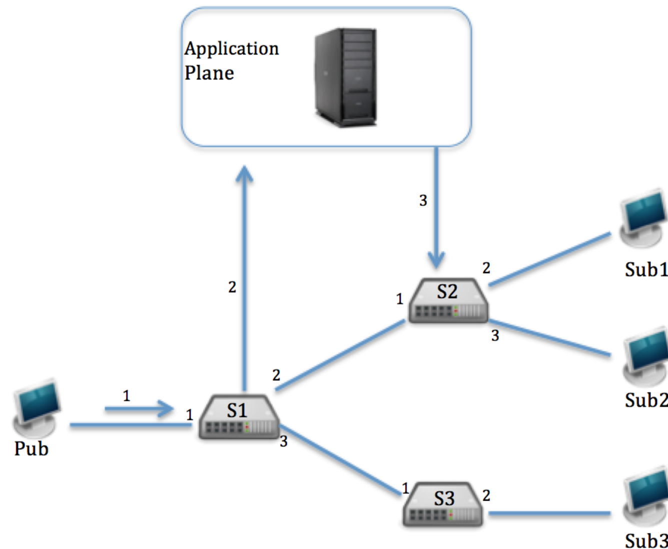

Figure 4.9: Mutual Forwarding Case 3

Chapter 5

Evaluations

This chapter presents the analysis of the purposed hybrid publish/subscribe system. The developed system has been tested with various setups on mini-net a virtual Software defined networking test environment and also in the network with real hosts. Here we discuss the tests performed in the real network environment as it provides more authentic results.

We start by having a look at the false positives generated in the hybrid pub/sub system as compared to the PLEROMA implementation. Then we compare the performance of the hybrid system to an all application layer implementation and the PLEROMA implementation.

5.1 The System

The hybrid-System comprises of three planes as discussed in the section [4]. The control plane and the application plane are each built over an SDN controller. Floodlight controller was selected to perform as the SDN controller for the hybrid pub/sub system, it is an open source java based implementation. The Floodlight controller allows for modules to be built as part of the controller. Hence the control plane and application plane logic is written in java as a Floodlight controller module. The control plane uses an ILP solver for subscriber selection process as discussed in the section [4.2.1]. The solver used for this purpose is the GLPK solver, which is an open source solver developed with ANSI C. The data plane can be crated by using any of the switches that support open-flow.

5.2 Test Bed

The initial testing of the Hybrid Publish/Subscribe system was performed using the simulation tool Mini-Net. Mini-Net provides a virtual network with programmable topologies to which external SDN controllers can be attached. It provides a very good platform for testing but as it is just a virtual setup the time delay calculations are not applicable to real world scenarios. This is the reason the system was finally tested on a real environment to provide more authentic results. The test bed basically consists of four machines; three machines serve as the publishers and subscribers each machine has 16 cores working at 3.50 GHz while the fourth machine runs the SDN Controllers for the control plane and the application plane. The

fourth machine has very powerful hardware as it supports both the application and control plane. It has 40 cores working at 3.10 GHz.

The fig [5.1] shows the topology created for the test bed. The Pica8 switch, which supports open-flow was used to create the data plane. The topology contains a total of 10 switches connected in half factory topology. Each leaf switch is connected to two hosts. Which gives us eight hosts that can be set as publishers or subscribers. We create the eight hosts by diving each of the three host machines into three virtual machines each having four cores that are working at 3.50 GHz.

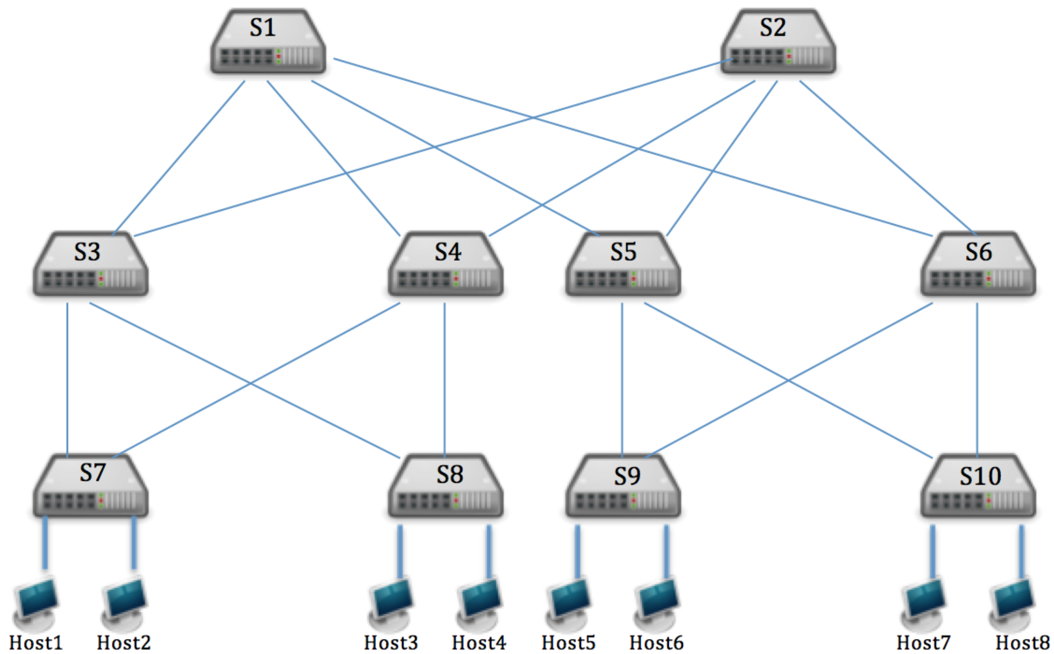

Figure 5.1: Test Bed Topology

5.3 Test Setup

To setup the tests the host 1 and host 2 as shown in the fig [5.1] were set as the publishers while hosts 3 to 5 were set as the subscribers. To perform the subscriptions uniform 2-dimensional data was generated and randomly distributed among the subscribers. The tests were performed using different number of subscriptions i.e. 500, 1000, 1500 and 2000 subscriptions respectively. The data was generated separately for each subscription set. Finally for the hybrid publish/subscribe system the ILP solver requires the time constraint from the subscribers as explained in the section [4.2.1]. For testing purposes the constrains are set randomly at run-time and two subscribers are set with critical timings constraints such that

their subscriptions cannot be moved to the application plane. Now we have a look at the results of the tests performed in different scenarios

5.4 False Positives In Network

The first test that we look at is the false positives that are generated in the network when the same data is used with the PLEROMA model, working completely at the network layer and the Hybrid Publish/Subscribe System. It should be noted that the false positives reduced using the Hybrid system directly depend on the results of the ILP solver as explained in the section [4.2.1]. Here the hybrid system is set in such a way that two out of the six subscribers have timing constraints such that they cannot be processed at the application layer while the subscriptions for all the other four subscribers can be processed at the application layer. This scenario has only been set for testing purpose. A subscriber has the ability to set the time constraint for each subscription and hence one subscriber can have some subscriptions that are processed at the application layer while the others at the network layer. For this test we show the percentage of false positives generated in the system for both the PLEROMA and Hybrid system. The percentage is calculated as follows:

$FalsePositivePercentage = \frac{Number\,of\,Undesierd\,Events}{Number\,of\,Desired\,Events}$ x 100

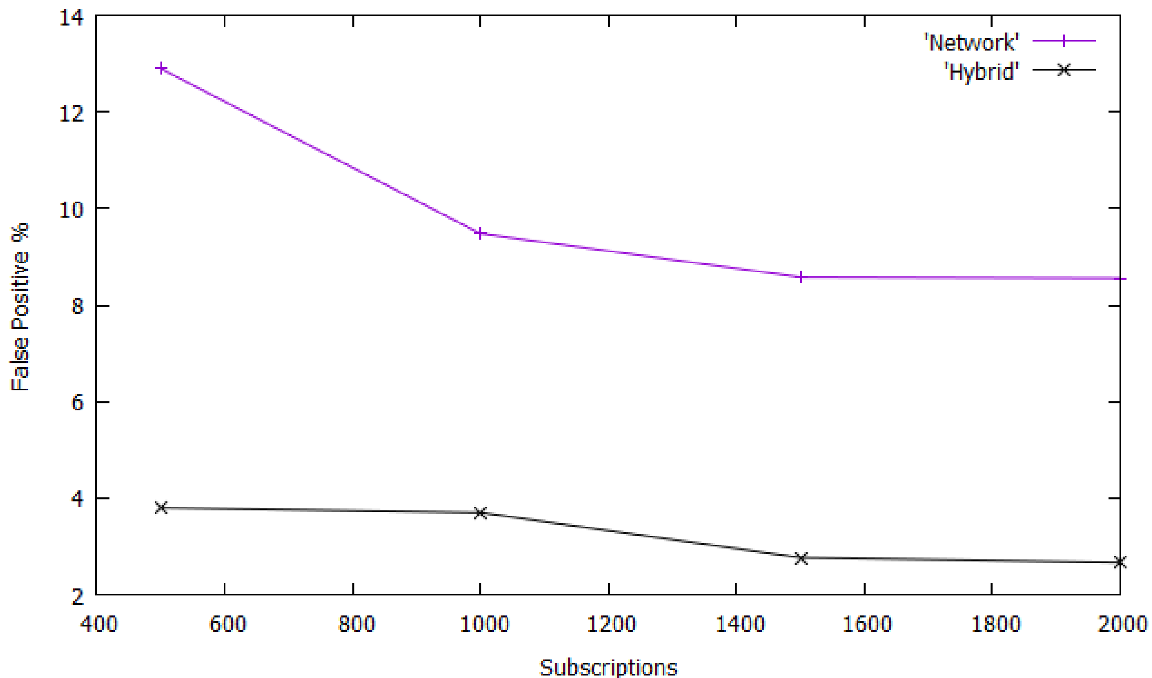

Figure 5.2: False Positives

We compute the percentage of the false positives generated in the network by taking the sum of all false events received at the subscribers and dividing them by the sum of all the desired

events at the subscribers. The fig [5.2] shows the graph with results of the experiment.

The graph compares the false positives generated in the network by the PLEROMA system against the Hybrid system. It can be seen that in the Hybrid case the false positives have dropped significantly as compared to the PLEROMA system. But still the false positives are not reduced to zero because some of the subscribers have time constraints that forbid the processing of some subscriptions at the application plane. Hence for the Hybrid system the results provided by the ILP solver to register different subscriptions at the application layer and drop others will greatly effect the false positive generation in the network.

5.5 Hybrid System Delays

The Hybrid system succeeds in reducing a large number of false positives generated in the network but due to shifting the processing of some of the subscriptions to the application layer the end-to-end delay foe events is increased. Hence this test was performed to compute the end-to-end delay for an event from the publisher to the subscriber under different number of subscriptions installed in the system for each Subscriber. The test was performed by sending 10000 events related to the installed subscriptions towards the subscribers from the publisher and then taking the average of the end-to-end delay for the events received at the subscribers. The fig [5.3] shows the graph with results of the experiment.

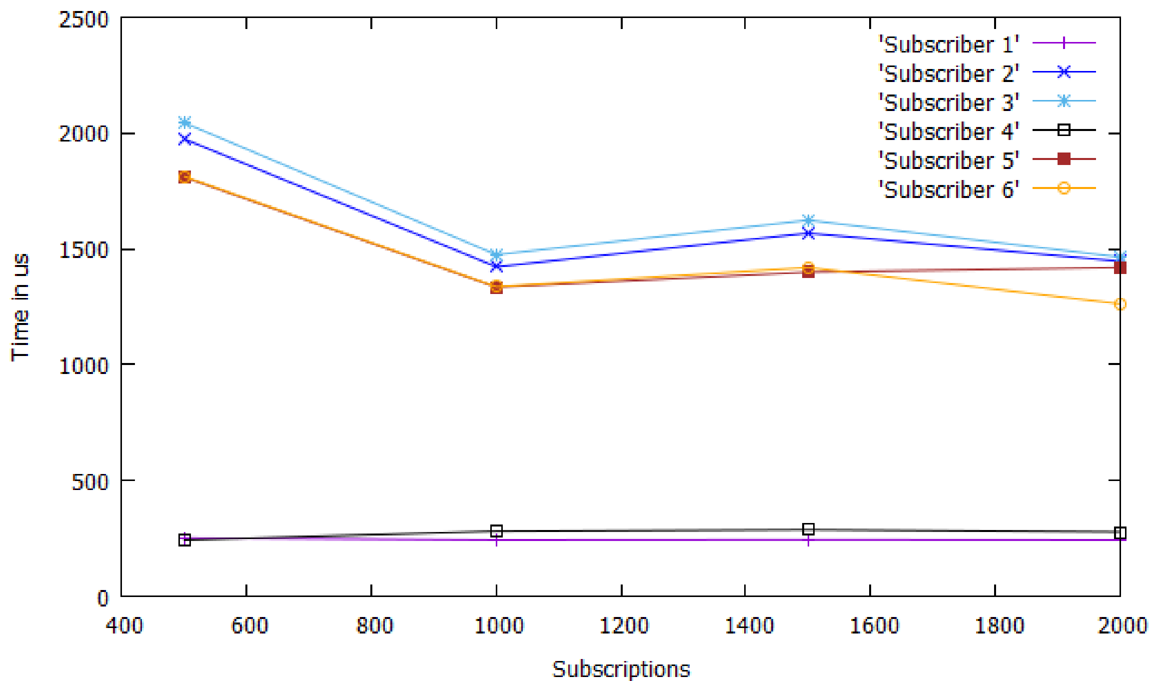

Figure 5.3: Hybrid System Delays

It can be seen in the fig [5.3] that the subscribers 1 and 2 have different delay as compared to the other subscribers. The reason is that the subscribers 1 and 2 have critical timing constraints because of which they cannot be processed at the application plane and have the delay results according to filter time at the network layer. The other four subscribers allow for the computation of their subscriptions at the application layer. Hence some of their events are processed at the network layer while others are processed at the application layer producing the delays as shown in the graph.

5.6 Delay Variations

To present a better picture about the performance of the Hybrid system a test was performed to compare the average end-to-end delay generated by the hybrid system with the delays generated if the same test is performed on the PLEROMA system and only Application plane of the hybrid system. The delay was calculated by taking the average of the end-to-end delay of an event at all the subscribers. The fig [5.4] shows the results of the test performed on all the three system with the same data set.

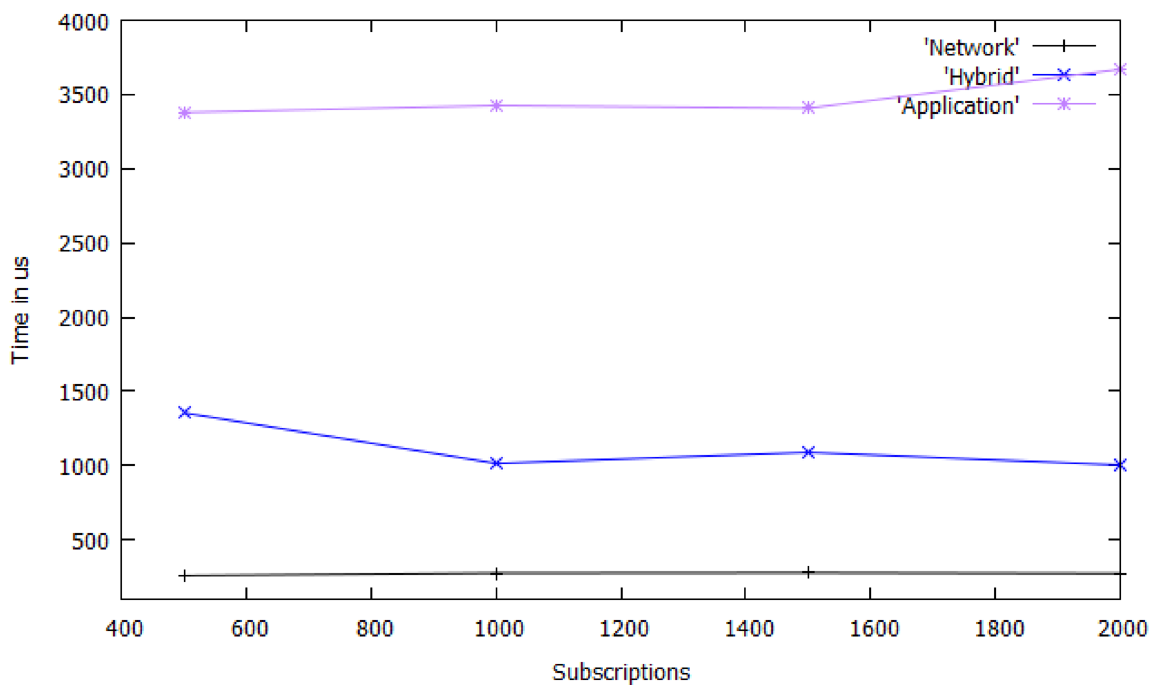

Figure 5.4: Delay Comparison

It can be seen from the graph shown in the fig [5.4] that the hybrid system provides much lower delay times then a system processing all the events at the application plane system. Still the delay for the Hybrid system is greater as compared to the PLEROMA system, as some of the events for the Hybrid system are processed at the application layer that takes much more time then processing the events only at the network layer as done in case of PLEROMA.

5.7 Performance Enhancement

The Hybrid pub/sub system can work with different number of partitions at the application plane. Each of which run on a different thread and work independently of each other, as explained in the section [4]. The number of partitions working in the system is selected at the start of the system. Increasing the number of partitions can enhance the performance of the Hybrid system. The graph in the fig [5.5] shows a test that was performed by processing all the events at the application plane of the hybrid system.

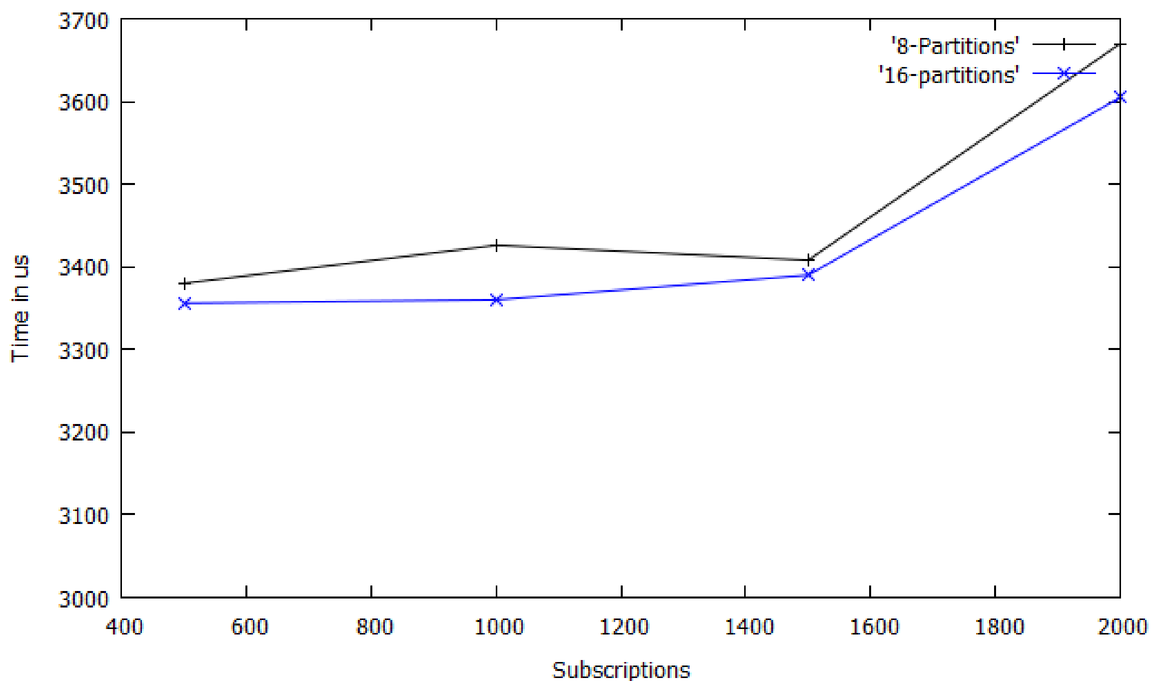

Figure 5.5: Performance Comparison

The test was performed by running the application layer of the hybrid system for the same data with 8 and 16 partitions respectively. It can be seen that the performance is improved for the test executed with 16 partitions. The improvement also depends on the number of subscriptions installed at the application plane. For these test scenarios there are a small number of subscriptions installed at the application layer. But in a real world case many more subscriptions will get installed at the application plane. In that case the performance will improve much more with increasing the number of the partitions. The performance of the Hybrid system also depends on the machine its application plane is running. The more powerful the machine the more better the performance will get.

5.8 Events at the Application plane

The basic functionality of the Hybrid system is to divide the events generated by the publishers among the network layer and application plane in such a way that the False Positives generated in the system are reduced. As the number of subscriptions registered at the application layer increases the performance of the Hybrid system decreases. Hence it is very important to to find the right balance to create an efficient system with reduced number of false positives. This test was performed to compute the percentage of events being forwarded to the applicant plane by the Hybrid system. The formula used to calculate the percentage is as follows:

$FalsePositivePercentage = \frac{Events\,processed\,at\,the\,application\,layer}{Total\,events\,received\,at\,subscribers}$ x 100

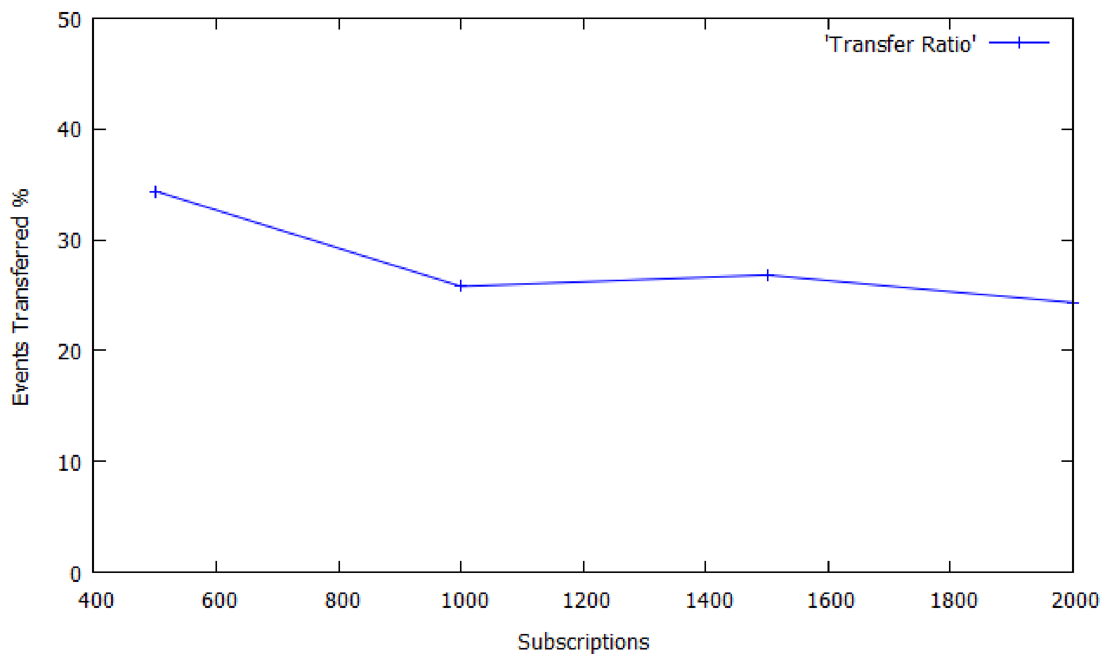

Figure 5.6: Events at the Application plane

It can be seen in the fig [5.6] that on average 25 percent of the events are being sent to the application plane which results in an acceptable performance of the overall system as shown in the previously presented graphs. This value will be changed according to the requirements of the subscribers on timing.

Chapter 6

Conclusion

This main focus of this thesis has been to enhance the system presented by PLEROMA such that it does not suffer from the inherit problem of false positive in the network. Towards this goal a Hybrid publish/subscribe system was presented and we a had a close look at the design requirements and implementation of this system. In the end the system was thoroughly tested and the results were compared to its parent model PLEROMA. The results shown by the Hybrid pub/system are promising and fall under the expected range. There is still much room for improvement, for example in terms of developing a stronger ILP for the subscriber selection process. Also a shift to IPv6 for addresses will enhance the system. In a future implementation the system can also greatly benefit from multiple instances of the application plane distributed uniformly in the the network. This will distribute the event load among multiple planes and better performance will be achieved.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature

Bibliography

[1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf *"Design and evaluation of a wide-area event notification service,"*, ACM Trans. Comput. Syst., vol. 19, pp. 332–383, Aug. 2001.

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec *"The many faces of publish/subscrib"*, ACM Computing Surveys (CSUR), vol. 35, no. 2, pp. 114–131, 2003.

[3] Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, Kurt Rothermel *"PLEROMA: A SDN-based high performance publish/subscribe middleware."*, Middleware 2014: 217-228.

[4] Wikipedia, *"Software-defined networking – wikipedia.."*, [Accessed : June, 2015].

[5] Liu, Y., Plale, *" Survey of publish/subscribe event systems."*, In: Indiana University Computer Science Technical Report TR-574. (2003)

[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, *"SCRIBE: A large-scale and decentralized application-level multicast infrastructure,"*, IEEE Journal on Selected Areas in Communications (JSAC), vol. 20, no. 8, pp. 1489–1499, 2002.

[7] IBM TJ Watson Research Center, *"Gryphon : Publish/Subscribe over Public Networks."*

[8] Wikipedia, *" Reverse path forwarding – wikipedia.."*, `https://en.wikipedia.org/wiki/Reverse_path_forwarding`[Accessed : June, 2015].

[9] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, *"“LIPSIN: line speed publish/subscribe inter-networking,"*,in Proceedings of the ACM SIGCOMM 2009 conference on Data communication, SIGCOMM ’09, (New York, NY, USA), pp. 195–206, ACM, 2009.

[10] Wikipedia, *" Bloom filter – wikipedia.."*, `http://en.wikipedia.org/wiki/Bloom_filter`[Accessed : June, 2015].

[11] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. *"Design and evaluation of a wide-area event notification service."*,ACM Transactions on Computer Systems, 19(3):332–383, August 2001.

[12] M. A. Tariq, G. G. Koch, B. Koldehofe, I. Khan, and K. Rothermel, *"Dynamic publish/subscribe to meet subscriber-defined delay and bandwidth constraints."*,in Euro- Par 2010-Parallel Processing, pp. 458–470, Springer, 2010.

[13] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. Rothermel *"Efficient content-based routing with network topology inference."*, Middleware 2013.

[14] Sukanya Bhowmik, Muhammad Adnan Tariq, Boris Koldehofe, André Kutzleb, Kurt Rotherme *"Distributed control plane for software-defined networks: a case study using event-based middleware."*,9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015.

[15] O. Consortium et al., *" Openflow switch specification v1.3"*,

[16] Floodlight, *"Floodlight SDN controller."*, `http://www.projectfloodlight.org/floodlight/`2013. [Online; accessed June-2015].

[17] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. Rothermel *"Distributed spectral cluster management: a method for building dynamic publish/subscribe systems,"*, in Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS'12, (New York, NY, USA), pp. 213–224, ACM, 2012