Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Master Thesis Nr. 3450

Providing in-network content-based routing using OpenFlow

Gagan Bihari Mishra

Study Program:	M.Sc. Information Technology (INFOTECH)
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	M.Sc. Muhammad Adnan Tariq
Start Date:	10/12/2012
Submission Date:	11/06/2013
CR-Classification:	C.2

Institut für Parallele
und Verteilte Systeme
Abteilung Verteilte Systeme
Universitätsstraße 38
D-70569 Stuttgart

Abstract

Content-based routing as provided by publish/subscribe systems has evolved as a key paradigm for interactions between loosely coupled application components (content publishers and subscribers). Content-based routing aims to increase the efficiency of forwarding by utilizing the diversity of information exchanged between application components. 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.

Many middle-ware implementations for content-based publish/subscribe have been developed over the last decade. However, implemented on the application layer, their performance is still far behind the performance of communication protocols implemented on the network layer w.r.t. throughput, end-to-end latency and bandwidth efficiency. Therefore, it would be highly attractive to implement content-based routing directly on the network layer. Especially, the advent of new networking technologies namely, software-define networking and network virtualization have potential to make this reality. To this end, recently a reference architecture has been proposed allowing for the embedding of content-based routing at the network layer by utilizing OpenFlow specification.

The task of this thesis is the concrete realization of content-based routing in the OpenFlow reference architecture. In particular, the thesis focuses on the implementation/embedding of filtering-based publish/subscribe approaches in the reference architecture, as a proof of concept. The implementation is then evaluated w.r.t. message forwarding delay, false positives etc.

Acknowledgements

This work would not have been possible without the help of many people. First of all, I wish to thank Prof. Dr. Kurt Rothermel and the Department of Distributed Systems for giving me this opportunity to work on *Software Defined Networks* for my master thesis.

I am extremely grateful to my supervisor, M.Sc. Muhammad Adnan Tariq, for his invaluable guidance, support and enthusiastic encouragement throughout the duration of this thesis. I wish to express my indebtedness for his immense help and keen involvement with the project and for introducing me to the details of pub/sub systems.

I would also like to express my sincere gratitude to Dr. Frank Dürr and Dr. Boris Koldehofe for the insightful discussions on SDN and help with the test environment. I am also thankful to Dipl.-Inf. Martin Brodbeck for providing me with the necessary test-bed setup for my evaluations.

Sincere thanks are also due to the many developers with the Floodlight Controller project for helping me cope with the show-stoppers during implementation of the project.

Many thanks to my friends, Sumeet Mahajan and Sukanya Bhowmik, for making it a fun environment to work, as well as for the useful critiques of this thesis document.

Finally, I wish to thank my family for their support and encouragement throughout my studies. I express my hearty gratitude to Anupam for virtually being there with me in all ups and downs of life here in Germany.

Dedicated to my family.
Dedicated to Anupam.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

A Publisher/Subscriber (pub/sub) system [1], sometimes also referred to as an event notification system[2], is a well established paradigm for content delivery in a distributed environment. A pub/sub system consisting of publishers and subscribers, allows information distribution from publishers to subscribers in an anonymous and loosely coupled manner within time and space, where the publishers or the subscribers are unaware of each others physical existence.

The loosely coupled design, makes a pub/sub system highly useful and practical for many real world message distribution scenarios. Examples of such pub/sub systems could be a mailing list notification, RSS feeds, news updates, stock market or weather updates.[3] In today's world, with fast growing internet users space and volumes of information exchanged, the importance of pub/sub systems with regards to efficient content delivery at a large scale, is only increasing.

Publishers, the sources of information, notify the type of information they intend to publish by means of *advertisements* and broadcast events related to the advertised content. As an example, a city's traffic monitoring and notification service could send out messages when certain routes in a certain region of the city are heavily jammed and better be avoided. Such a system is of course unaware of the existence of any listener for the event released.

At the other end, the subscribers as the sinks of information, notify their interest in a particular event by means of *subscription requests* and listen to any event of their interest. For example, with regards to the city's traffic monitoring system, a subscriber could express its interest to know about any traffic updates in a particular region of the city or a specific route between one point and another. Similar to the publishers, the subscribers are also unaware of physical location of the sources. There could be a distributed set of servers for traffic monitoring and sending out notifications. However, the subscriber stays ignorant of the actual point of source from where the event is received.

The requirement of anonymous message forwarding from publishers to the interested subscribers obviously needs some kind of a mechanism, which identifies the relevant subscribers for any given event and forwards it to them. This process, also termed as *Filtering*, is of primary importance in the design of a pub/sub system for it decides the efficiency of the system with respect to run-time, bandwidth usage and accuracy. For a large number of publishers or subscribers with huge volumes of events, a poorly formulated filtering approach can easily bring down the system due to its lack of scalability. Many approaches have been studied for efficient content filtering and some of them are discussed in the next chapter. Fast and correct identification of subscribers are typically the bench-marking characteristics of any filtering approach. Particularly, *false positives*, i.e., events sent to a subscriber which was not interested in receiving it, are targeted to be minimized and *false negatives* i.e., failure of transfer of a message to an interested subscriber are targeted to be eliminated.

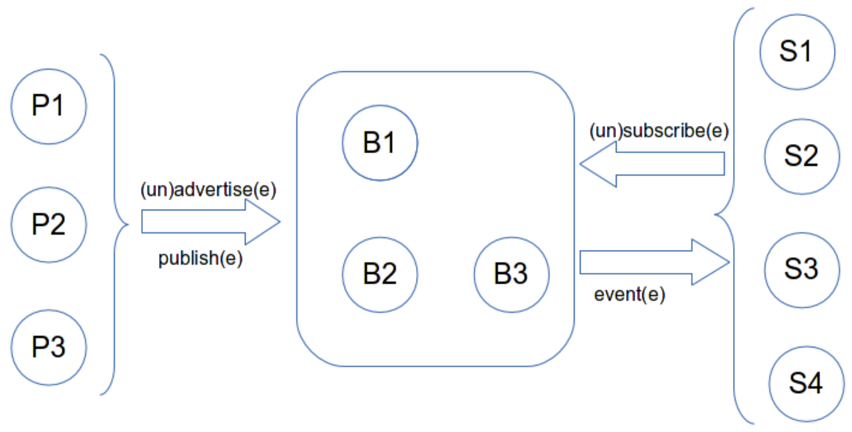


Figure 1.1: An overview of a broker-based pub/sub system

Numerous pub/sub system implementations have been proposed, which vary in the way pub/sub messages are routed and the manner in which filtering is done. At an abstract level, these can be categorized into *broker-less* and *broker-based* systems. In a broker-based pub/sub system (shown in the figure), the broker acts as a mediator between publishers and subscribers and does the filtering process for any published event. Whereas, in a broker-less environment, the filtering is actually done by the participating publishers and subscribers themselves, which is also referred to as a *peer-to-peer* paradigm.

In addition to the categorization of pub/sub systems into broker-less and broker-based types, another categorization is also possible based on the *expressiveness* of the participating subscribers as *topic-based* or *content-based.*

In a topic-based pub/sub system, the notifications or subscriptions are identified according to some predefined topics or subjects. Subscribers can subscribe to different topics of interest and will receive all the events published under that topic. The topics thus establish logical channels or groups similar to multicast groups. Any event published is sent to the group and all the subscribers of that particular topic receive the notification.

Clearly, topic-based pub/sub systems put constraints on the expressiveness of the participating subscribers. For example, taking the case of stock market quotes, the subscription for a topic-based pub/sub could simply be (*symbol = AMZN*). Which means, any stock quote of the company 'AMZN' is subscribed by the subscriber, even if it really wanted to get the notifications only when the stock prices are within a certain range. This clearly shows how the system is inflexible and forces the subscriber to get a huge amount of notifications irrespective of its interest.

This makes way for the more fine grained subscriptions in *content-based* pub/sub, where the notifications are not grouped based on any predefined subjects. In a content-based pub/sub system, the individual subscribers can specify their subscriptions to a very fine detail and receive only those events which match entirely to the criteria.

An example of such a subscription, for a stock quote could be specified as : (*symbol=AMZN, low > 50, high < 100, volume > 5000*), which shows how the subscriptions can be more expressive along each attribute.

Content-based pub/sub avails more flexibility with respect to expressiveness and hence does better bandwidth usage and creates less false positives. However, with the increase of expressiveness in the subscriptions, the filtering of notifications becomes a complex process and results in a higher run-time. Many different algorithms and approaches have been proposed to tackle this problem and some of the relevant works are discussed in the next chapter.

The presented work in this thesis targets broker-based content-based pub/sub systems and proposes a new content-filtering mechanism using the concepts of *Software Defined Networking.*

1.1 Software Defined Networking

Software Defined Networking (SDN)[4] is a technology which decouples the control plane from the data plane and gives more flexibility to control the network, based on specific requirements.

Traditionally, routers in a network are equipped with a control and a data plane, which are responsible for routing calculations and data forwarding respectively. SDN separates the control plane from the router and places it in a central server called controller. The controller thus keeps an end-to-end logical view of the whole network, which is essentially a graph, and can run custom algorithms to control the behavior of entire network; such as deciding routes for specific type of packets, analyzing network traffic etc. To achieve this, the controller communicates with the switches using standard protocols such as *Open-Flow*[5] and commands them for various activities. There can also be a distributed set of controllers for a network, but distributed controller systems have not been discussed as they are beyond the scope of this thesis. The figure below shows a simplistic overview of the SDN architecture.

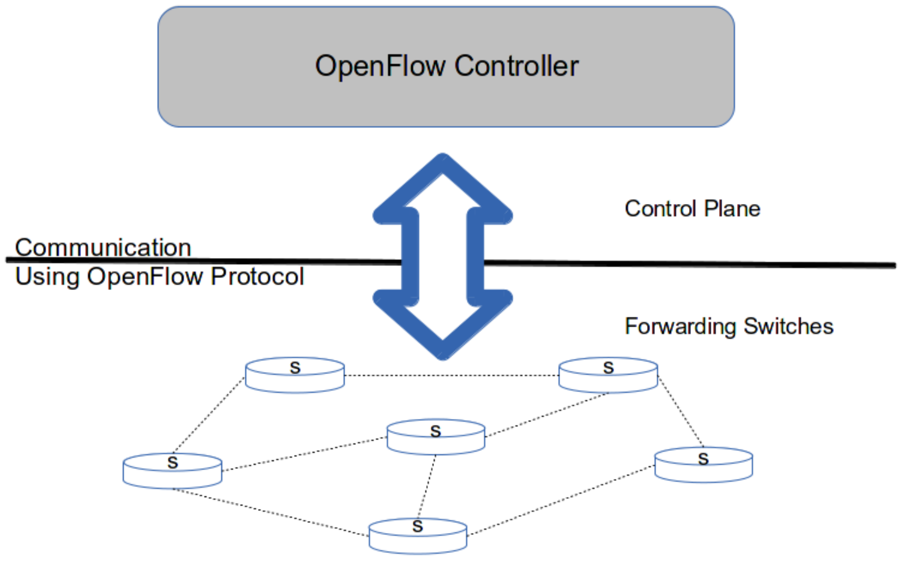


Figure 1.2: SDN Architecture

With the detachment of data and control plane, traditional distributed algorithms (e.g., Bellman-Ford algorithm) effectively reduce to a graph algorithm running on the controller. Behavior of the network can easily be experimented by adding/removing custom modules to/from the controller application, which was not so readily possible with legacy networks. Clearly SDN provides a lot of flexibility to dynamically control the network and also simplifies the network management tasks.

SDN has recently been deployed successfully for many practical scenarios such as data center load balancing, firewalls etc.[6] This thesis studies one similar possible use case of SDN for pub/sub systems, where content-based routing is being mapped to header-based routing and aims to take advantage of the ability of SDN to deploy it on the network layer.

1.2 Pub/Sub using SDN

Content-based pub/sub systems have mostly been implemented on the application layer and deployed as middle-ware applications. The application takes care of filtering, content matching, managing publishers and subscribers. Many diverse methods have been studied with a focus on reducing the filtering complexity and increase in scalability. However, being in the application layer, the performance is always limited and can never match to that of the lower level protocols such as routing protocols which directly make use of the hardware underneath.

Previously, there have been very limited work in this area of research, where a pub/sub system has been experimented on the network layer. But as a limitation, the existing methods need specialized hardware components to realize the concept. These are discussed in more details in the next chapter.

However, with the advent of *Software Defined Networking*, a whole new opportunity has been created to study the feasibility of implementing a content-based pub/sub system directly on the network layer without special hardware requirements. As discussed, SDN decouples the control plane from the routers and facilitates user defined flow programming for the data plane switches. Routes can be dynamically established, based on specific application requirements.

With a programmable network in hand, any content-based routing system can be experimented on the network layer and a pub/sub system is no exception. The controller having the complete view of the network, along with the location of publishers and subscribers, can actually program the flows from the sources to the sinks, effectively pushing the expensive filtering process on to the hardware. This surely promises better performance but also raises other questions such as how the content-based routing can be mapped to header-based routing and what could be its limiting factors.

Koldehofe et al.[7] have presented a hypothesis on how the concept of SDN can be utilized in pub/sub systems. This thesis aims to study one approach from [7] to implement a content-based pub/sub system on network layer using the OpenFlow protocol.

1.3 Thesis Organization

Chapter 1 briefly introduces *Publish/Subscribe* systems and provides some commonly used terminologies and definitions, followed by an introduction to the concept of *Software Defined Networking*. The chapter ends with an overview of the problem statement which the thesis aims to study in detail.

In Chapter 2, some relevant works from the present literature concerning pub/sub systems are studied. These include clustering approaches and various filtering methods employed in pub/sub systems. In addition, topology aware systems and quality-of-service in pub/sub systems are also studied.

Chapter 3 formulates the problem statement. It also explains the data representation formats used and gives an overview of the OpenFlow protocol.

Chapter 4 discusses the used algorithms in detail. It presents the algorithms used for controller as well as publisher and subscriber programs.

In Chapter 5, the implementation related issues have been addressed. This includes the design of message formats, the controller application and realization of algorithms discussed in chapter 4.

The test-bed setup and experiments, followed by evaluations are discussed in chapter 6. This follows with the final conclusions and discussion on possible future works in chapter 7.

Chapter 2

Background and Related Work

Many notable works have been done over the years on pub/sub systems primarily concerning run-time overhead in filtering, bandwidth usage, latency reduction etc. This chapter discusses some of the relevant works from the present state-of-art which lays the foundation for this thesis. Some of the works discussed in the subsequent sections bear great relevance to this work, as they have provided the ground work for data representation and routing issues.

Depending on implementation, content-based pub/sub systems can broadly be classified into two types as multicast-based and filter-based systems. A multicast-based system divides the event space into clusters and subscribers interested in the same group are put in the same multicast tree. The following section discusses this in detail. Later sections discuss various filter-based approaches where content matching is done for every published event and forwarded to the identified interested subscribers.

2.1 Subscription Clustering

Clustering is the method of grouping closely related subscribers and to map them to a multicast group. Messages sent to a multicast group is received by all the subscribers in the group. Sometimes these are also referred to as channelization, as they group the related contents and hence define a virtual channel for the related subscriptions. Clustering of subscriptions do indicate reduction of false positives as the subscribers are grouped into a number of clusters and thus, unrelated subscriptions are excluded from the group. However, this depends on the choice of clustering algorithm used, event distribution within the clusters and other factors. The following discusses some of the subscription clustering methods in the context of content-based pub/sub systems.

2.1.1 Grid-Based Clustering

Riabov et al.[8] present some algorithms for subscription clustering, namely *Grid-Based Clustering*, which essentially make use of algorithms normally used for data clustering in other areas of computer science such as *Data Mining.*

In this approach, the event space Ω is partitioned into smaller regular sized cells. That means, given an event space with two attributes, the grid forms a rectangle which is divided into a number of cells. The subscription of a subscriber can thus correspond to one or more cells in the event space.

The clustering algorithm aims to group similar cells into one cluster for which the definition of 'similarity' is given by the following two factors.

Feature Vectors : For each cell a in event space Ω, a subscriber membership vector $s(a)$ is defined such that:

$$s(a)_i := \begin{cases} 1 & \text{if } \exists j \text{ s.t.} b_{i,j} \cap a \neq \emptyset, \\ 0 & \text{otherwise} \end{cases}$$

A non-zero element in the feature vector of a cell indicates presence of a subscriber. For clustering of cells, the Euclidean distances between the feature vectors are minimized. Thus, for two cells a and b, if the feature vectors are identical, in other words, the Euclidean distance between them is zero, then they can be grouped into one cluster without creating any false positives.

Distance Function : The distance function $d(a,b)$ is a measure of false positives. This gives the expected number of false messages delivered to the subscribers if cells a and b are grouped together.

$$d(a, b) := p_p(a) \sum_{i \in V_S} max\{[s(a)_i - s(b)_i], 0\} + p_p(b) \sum_{i \in V_S} max\{[s(b)_i - s(a)_i], 0\}$$

where $p_p(a)$ denotes the probability density function of publications in cell a.

Given the two definitions, *hyper-cells* (group of cells with identical feature vectors) are created and standard clustering algorithms such as k-means clustering[9] are applied to the hyper-cells taking the distance function into consideration. Thus K number of clusters are created each of which are assigned a multicast group.

Apart from k-means clustering, other approaches such as Pairwise Grouping or Minimum Spanning Tree Clustering can also be used to form K number of clusters.

The basic problem that the above mentioned clustering approaches suffer is the alignment of cells. The subscribers may not necessarily subscribe to an event-space which is perfectly aligned with the cell boundaries. This obviously creates false positives as the subscriber is now considered in multiple cells even if the subscription-space intersects only a small region of the cell-space.

To cope with this alignment problem, another modification is suggested in [8] which assigns a 'weight' to each cell depending on number of subscribers and probability of publications and creates multicast groups aligned with the cell borders.

On arrival of an event, corresponding to the event's cell, associated multicast groups are searched and forwarded to the group accordingly. If no multicast group is associated, then it is forwarded using unicast.

Although Grid-Based Clustering is scalable and efficiently divides the subscribers into clusters, the clustering algorithms can demand too much run-time depending on the number of subscribers or the number of clusters. And since the clusters need to be updated periodically, this overhead cannot be ignored. There are some improvements claimed in [10] by using Spectral Clustering techniques which also applies for broker-less pub/sub systems.

2.1.2 Subscription Clustering using Spectral Methods

Tariq and others[10] propose the use of Spectral Clustering[11] algorithms from *Graph Theory* in pub/sub systems which promise the formation of better and uniform clusters with reduced false positives as compared to Grid-Based Clustering.

As before, for clustering of subscribers, a similarity function is defined in this approach which makes use of the events matched for the subscriptions. Hence, the similarity between subscribers is a dynamic entity as the number of events matched with the subscriptions might alter with time.

More precisely, the similarity between two subscriptions a and b is defined to be the ratio of intersecting event sets matched by the subscription to the union of all the events matched or un-matched by the subscriptions. Lower value of the ratio indicates less similarity between the subscriptions.

Given the similarities between the subscribers, a similarity graph is maintained and standard spectral clustering algorithms such as *Ratio Association* or *Normalized Cut* are applied to the graph. Subscription clusters are thus formed by partitioning the graph into sub-graphs.

The studies by [10] show better quality of clusters in terms of randomness and accuracy and also reduced false positive rate. However, the false positive improvement also depends on the event distribution.

Another important study by [10] is the usage of clustering in a distributed manner, i.e., for a broker less environment where each of the publisher/subscriber takes part in forwarding the events.

This approach is a two step process: a) dimension reduction b) clustering. First, the subscribers are divided into a multilevel hierarchy where small groups of subscribers are formed at the lowest level. As we go up in the hierarchy, each lower level group selects a coordinator to be a part of the upper layer which is also provided with a list of randomly selected subscriptions from the group which is termed as *Landmarks* by the authors. Each group then locally calculates a reduced set of dimensions, from the list of subscriptions provided and then the reduced set of dimensions are mapped to a globally unified coordinate system. This is necessary as the reduced set of dimensions calculated within each group might differ from other groups. The global coordinate system is taken from the subscription list of the group at the top most hierarchy i.e., the root group.

Once the dimension reduction is done, distributed k-means clustering is performed either at the root level or at each lower level group followed by a merging of the clusters. Like in centralized pub/sub systems, periodic computation of clusters and global coordinates is required, which is triggered at the root level. Similarly, for new subscribers, the local coordinator of a group at the lowest level calculates its global coordinates and the subscriber joins the appropriate cluster whose center is nearest to its global coordinates.

For a broker-less environment the above study seems to be a promising way for fast and effective calculation of clusters. But the performance benefit comes at the cost of reduced accuracy which depends on the number of hierarchies and reduced set of dimensions as the dimension reduction in the subscription definitions can introduce false positives.

Nevertheless, from the point of view of implementation of a pub/sub system using SDN, clustering approaches could possibly be used with the channelization method discussed in [7] where the authors suggest an approach to cluster subscribers based on similarities in their subscription spaces and place them on a single channel. Spectral Clustering methods, having shown better accuracy can be studied in implementation of channelization using SDN. However, this thesis implements the in-network filtering approach and hence the channelization method is discussed as a possible future work or extension.

2.2 Filtering Methods and Other Pub/Sub organizations

In a filter-based pub/sub system[2], usually we have one or more brokers or in case of a peer-to-peer network each participating node can behave as a broker and the brokers carry out the subscription matching and forwarding activities. The below presents few techniques which differ essentially in the way subscriptions are matched by the brokers or the way brokers route and distribute the events.

2.2.1 Virtual Grouping

Virtual Grouping is a hybrid approach[12] for routing of events and clustering of subscribers. It is termed hybrid as it uses both the concepts of filtering by the brokers as well as clustering of subscribers.

Basically, this approach creates virtual groups or virtual trees as an overlay on the original pub/sub tree. Each of the leaf nodes in the virtual groups correspond to subscribers and the root nodes to publishers. Subscribers with the same set of subscription belong to the same virtual group. During an event propagation, the event match or filtering is done only downwards and it is forwarded only along the matching group thereby reducing unnecessary forwards to other pub/sub broker servers.

Apart from this, [12] also points a way to reduce delay in forwarding the messages within a virtual group by means of shortcuts, i.e., when alternative shorter paths are known from a root node to the leaf nodes (subscribers), then matching along all the intermediate nodes can be avoided. This shortcut is calculated in a distributed manner where each intermediate node participate in getting the shortcuts from its children nodes in the tree, similar to routing topology discovery protocols in networks.

Formation of virtual groups is almost identical to the clustering of subscriptions discussed in Grid-Based Clustering[8] where the event-space Ω is divided into cells and cells are added to the virtual group if they have the same set of subscribers. The virtual group creation is carried out in a broker system and since it depends on event space distribution and subscriptions, these groups need to be computed periodically just like clusters of subscription.

Virtual grouping is fairly a simple and efficient approach as far as reducing delivery time is concerned, but faces the similar issues as Grid-Based Clustering. The false positives can be high depending on the cell size and subscription overlapping in the event-space. However, this is inevitable for any approach which makes use of such a partitioning of event space. The performance gain derived by creating the virtual groups depends on the traffic within the group. Also, periodic computation of virtual groups can be expensive for very large number of subscribers and huge event space.

The in-network filtering implemented in this thesis using SDN, shows vague similarities with the virtual grouping where routing trees are constructed for different subscription spaces. This will be discussed in detail in chapter 4.

2.2.2 Subscription Summarization

Subscription summaries introduced in [13] targets data structures for subscriptions and efficient matching within the brokers.

The virtual grouping method, discussed above, provides a way to minimize network traffic by forwarding only along the matched groups. Usage of subscription summaries targets the same unnecessary forwarding elimination in a different manner. Also, here, the subscriptions are forwarded to the systems not in its original form, but by using compact data structures such as textitBloom Filters[14] to drastically reduce network traffic in case of large number of events.

This approach defines some data structures such as:

Subscription attribute summary (SAS), a bit vector which carries the information about relevant attributes for a specific broker.

Attribute association list (AAL), a *2*-dimensional array with fixed number of columns corresponding to the attributes and variable number of rows for each subscription, marking which attribute is present in which subscription. This is essentially similar to an adjacency matrix of a graph representation.

Arithmatic attribute constraint summary (AACS), a set of two arrays which represent the subscriptions for numeric types. One array is for the representation of ranges of attributes, i.e., max and min value of an attribute in a subscription and the other being used for equality of attributes with a fixed value.

String attribute constraint summary (SACS), a set of three bit vectors used for prefix-string-matching, suffix-string-matching and exact matching. Each of these three bit vectors are Bloom Filters and the size of them depends on the length of the string being encoded.

With the data structures in place, the event matching is a fairly simple process. When an event arrives, the SAS data structure gives a list of brokers which might be interested in this subscription, thus reducing unnecessary forwarding to all brokers. Secondly, for the event to subscription matching, AAL, AACS and SACS data structures are used, which indicate a) if all the relevant attributes in subscription are present in the event or not, b) if the ranges or fixed values for arithmetic type attributes in subscription match to that of the event or not and c) if the string attribute in the subscription match to the string attribute in the incoming event respectively.

This approach of using subscription summaries is very interesting as it reduces the network traffic to a large extent and at the same time the matching process is pretty much straight forward and simple to implement. Also, unlike clustering approaches or virtual grouping, there is no need for periodic computation of the data structures.

However, due to heavy usage of Bloom Filters, the data structures, particularly SAS and SACS are prone to have collisions and can create false positives. If the vector SAS creates a false positive, the events will be forwarded to brokers not interested in the event. This of course depends on the choice of hash functions for the Bloom Filter as well as length of the bit vector used.

From the point of view of in-network filtering, the data structures used in this approach needs to be unified to a single bit-vector, so that contents are mapped to packet headers. Usage of bloom filters is one way to achieve this. But, this is impractical due to the fact that events generated by publishers need to exactly map to the same bit-vectors, as containment relationships cannot be managed with bloom filters.

2.2.3 Prefix Forwarding

The limitations on expressiveness and repeated matching process in multiple number of broker systems is targeted in [15], where the matching process is done only once and then is forwarded till the event reaches its destination without keeping any constraints on the subscription attributes.

This approach is an improvement of the SIENA system introduced in [2]. As a first step, this method normalizes the subscriptions by removing redundant attributes, e.g., subscription $(x > 1 \wedge x > 2)$ is normalized to $(x > 2)$ by evaluation of boolean expressions within the filters. This normalization is done outside the pub/sub system i.e., by the subscriber/publisher itself before sending the subscription or advertisement to the system. Also, the subscriptions are required to be known by all the participating pub/sub router or brokers.

The meat of this approach lies in the next step which makes a *Routing Tree (RT)* at each edge system (i.e., a broker connected to the publisher) and is kept by all the participating brokers in the pub/sub system.At the edge router the routing tree is created and maintained by a Tree Optimizer. This RT is used for event filtering and forwarding when a new event arrives. The routing tree keeps the attribute constraints of the subscriptions in its nodes such that each node corresponds only one attribute constraint of the subscription. Thus for a subscription with 2 or more constraints, it would create 2 or more nodes in the routing tree. For every additional attribute constraint in that subscription (or any other subscription) new nodes are added to the tree, either at the same hierarchy or at the child level of any existing node.

For example, for two different subscriptions $(x > 1)$ and $(y > 2)$, the routing tree algorithm can add two nodes for each of them directly under the root node. But, if there is a single subscription ($x > 1 \wedge y > 2$), the routing tree would consist of one child node for one subscription attribute, under which the other attribute can be placed as a new child. Within the tree, the outgoing interface (or the address of the next broker) is stored at each node where a subscription is matched. So for the above mentioned example, at the lowest hierarchy where both the constraints are met, the outgoing interface data is stored which shows where to forward the event in case of a match.

This approach matches the events only once at the edge router and if the event is matched, a copy of the matched tree portion (Forwarding Prefix Tree (FPT)) is attached along with the event before forwarding it to the next systems. In the subsequent routers, this FPT is parsed only to get the outgoing interfaces for the next forwards.

Although this method reduces the number of matching processes and hence improves the latency requirements in message delivery, it does raise some concerns related to tree management and implementations. First of all the tree management is a complicated process and it needs repeated computation during the arrival of a new subscriber. Secondly, as the number of subscriptions increase the tree size also increases. Thus for a normal pub/sub system with hundreds to thousands of subscribers, the tree size would become too huge unless most of the subscriptions overlap. This would result in large sized FPTs being forwarded along with the events. This problem is addressed by the authors in [15], suggesting to limit the height of the FPT to a certain manageable level. However, this work-around results in higher false positives as the subscription criteria below the limiting height of the tree are ignored from the matching process.

Apart from these, the tree manageability is a tough process. This is due to the fact that all the broker systems or routers need to have a global unified view of the routing tree. Ensuring this necessity might be an expensive process in large scale settings.

2.2.4 Topology Aware Systems

The filtering methods or pub/sub implementation approaches discussed so far, are oblivious to the underlying network topology at the lower level. Each of these methods deal at the upper overlay network of brokers which generally do not represent the true characteristics of the underlay physical topology.

To make it clear, a routing algorithm deciding routes for message forwarding at an overlay level, i.e., considering brokers, publishers and subscribers, might not generate routes as efficient as the route which takes care of the underlying physical network topology. For example, the routing algorithm might decide to route a message as $a \rightarrow b \rightarrow c$ whereas, when the underlying network routers and links are considered, $a \rightarrow c$ could be a faster and direct route.

The studies done by Tariq et al.[16] exploits this idea of topology awareness at the overlay level taken along with the traffic data and subscriptions to decide routes for the events.

First of all the underlying topology is determined in a distributed manner so as to minimize Relative Delay Penalty (RDP) and link stress. The RDP is a roughly measure of closeness of a route from one hop to another with the shortest route between the same two hops. Mathematically, it is the ratio of delay caused by sending an event along the route to the delay caused when sent along the unicast path in underlay. A value of 1.0 for RDP shows that the route at the overlay level has been efficiently chosen for it has the same performance as that of physical underlay link. The other factor, link stress, which is also taken into consideration during the topology discovery, is the number of duplicate copies of an event sent over that link. The overlay topology aims to choose links from the underlay in such a way that the stress is minimized.

The complete discovery of the underlying topology using standard tools such as *traceroute* is an expensive process. Hence the topology inference in [16] is limited to a certain level so that it doesn't effect the overlay formation drastically. The authors discuss two such algorithms namely *Landmark Approach* and *Random Walk* to decide the overlay topology with minimal link stress and optimum RDP.

Once the overlay structure is available, the routing algorithm runs on this to decide the routes from publishers to subscribers. In this step, the traffic volume in the links are also taken into consideration so as to uniformly distribute the traffic among all available links. *Core-Based Trees*[17] are used for this purpose to create the spanning trees spreading over the topology graph. The choice of cores is dynamic and needs to be recomputed over time, as the event traffic in the links might change with addition/removal of publishers or subscribers or simply because of changing the volume of events generated by different publishers.

Network topology inference to decide efficient routes for event forwarding is very attractive as it directly makes use of the physical topology underneath. Thus the latency and responsiveness in such an implementation of a pub/sub system promises to be better than other approaches working at an overlay level.

However, as mentioned, the detailed topology discovery is an expensive process and limiting it to a level can degrade the quality of the overlay constructed. Secondly, the topology needs to be recomputed when the underlay fabric is changed, e.g., a physical link or router goes down and this can trigger the re-computation of the whole process.

From the point of view of implementation of pub/sub on SDN, this approach is certainly of relevance as it makes use of the router topology to get the routes. On SDN, the underlying network fabric is inherently known as part of the SDN concept itself, thereby removing the entire expensive process of topology discovery. Secondly, the core-based tree formation to form routes from publishers to subscribers can also be studied in the context of SDN. The SDN controller can use the topology information to create and manage the routing trees.

2.3 Line-Speed Content Routing

The methods discussed till now differ mainly in the algorithms and data structures used for filtering and forwarding of incoming events, with a similarity of all being in the application layer. Implementation in application layer does makes sense for it gives enough space for complicated matching algorithms, store and forwarding in case of off-line subscribers (usually done in middle-ware systems such as *message-oriented-middle-wares*), dynamic route maintenance etc. However, since pub/sub systems use the same match-and-forward concept as in network routers, feasibility of such a pub/sub forwarding fabric on the network layer is an interesting field for studies.

LIPSIN[18] is one such approach which essentially forwards the events directly by modifying forwarding tables of underlying routers. The authors specifically target topic-based pub/sub systems in a large scale environment and aim to design a multicast fabric on network layer.

The approach taken in this method is a two step process. First the underlying topology is discovered using traditional methods of a network. This process is named as *Bootstrapping* and is carried out by the participating control plane software of the routers. Thus a network graph is created. Whenever an event is seen, a forwarding tree is constructed out of the topology graph, connecting the publisher to relevant subscribers. In the next step, the data is forwarded by the data plane analogous to traditional IP packet forwarding process.

The main contribution of [18] is in the packet forwarding process in the data plane. The approach uses *Bloom Filters* to encode the links on which the packet is to be forwarded. More clearly, each link in the fabric is given a link ID, which is a bit-string of certain length. When the topology manager identifies the links on which a packet is to be forwarded, it encodes the link IDs in a bloom filter (*zFilter*) and attaches it to the packet header. Thus for each delivery tree for a topic, a zFilter is created and this is conveyed to the publishers to keep a mapping of topic to zFilter. With the zFilter attached to the packet header, any node can identify the set of outgoing links for the packet, simply by an AND operation of outgoing link ID with the packet's zFilter.

Since the forwarding of the packets is done directly on the hardware, i.e., the forwarding process is hardware accelerated, this approach definitely shows better performance as far as delay and throughput are concerned. However, as it uses bloom filters to encode the links, there might be false positives due to false link identification. This depends on the choice of length of link IDs, length of the filter, as well as hash functions used. The authors have suggested to use virtual links, which is a single link ID for a set of sequential links in a delivery tree, instead of individual link IDs. This can reduce false positives to some extent as the number of link IDs to be encoded in the filter is reduced. But if a false identification occurs on a virtual link, the packet will be forwarded all the way where the virtual link points.

Moreover, LIPSIN also suffers from scalability issues for situations when large number of events are being generated and forwarding trees must be created for each of them. Secondly, this approach only targets topic-based pub/sub systems and also needs specific hardware to realize the network. Designing a specialized router for a topic-based pub/sub system doesn't seem practical.

Another such notable work in the context of line-speed routing could be by Moscola et al. in the article [19]. Although the authors do not target a pub/sub system explicitly and concentrate rather on a hardware accelerated regular expression matcher, the applicability of their research in a pub/sub system or more generically for intelligent content-based routing is considerable.

The major difference of this approach from LIPSIN is the fact that this method can actually be used for a content-based pub/sub system. In short, the content-based router in [19] expects XML[20] messages as input (as payload), which it parses and can check individual attribute values in the received message and then depending on the *routeKey* (present in the input XML message) it identifies the destination to which the packet needs to be forwarded. Thus, in the context of a pub/sub system, the routers can be pre-programmed with forwarding table entries for specific *routeKeys* and events can be sent in the form of XML messages which the router can parse and forward. With changes in the publisher or subscribers, the forwarding tables can be updated.

However, unlike LIPSIN, this implementation has not been studied in the context of a pub/sub system and hence, the feasibility, possible false positives, maintainability and other factors are not known yet. Also, like LIPSIN, it might not be practical in a real world scenario to have a whole infrastructure of specific type routers only for the purpose of pub/sub traffic.

2.4 Quality-of-Service in Pub/Sub Systems

Quality-of-Service (QoS) in the context of a pub/sub system refers to the minimal allowable standard of offered service. More precisely, QoS for a pub/sub system can refer to one or more factors from minimal bandwidth assigned to a system, maximum tolerable latency in message delivery, maximum tolerable false positive rate at a subscriber, maximum allowable duplicate events at a system and so on.

Not many pub/sub systems define any QoS level in their implementations. However with the increasing usage of pub/sub systems for various data distribution scenarios, ensuring and defining QoS has become an important requirement.

A relevant research in this aspect by Tariq et al.[21][22] facilitates QoS in a broker-less pub/sub system, where the QoS is defined by the subscribers. The authors have taken two factors: bandwidth constraints and delay (measured as hop-counts) to define quality of service from a subscriber's point of view. The peers in this approach, define their bandwidth and delay constraints and thereby participate in event forwarding to other peers as long as their individual constraints are not violated. The peers can also dynamically change their subscription space so as to adjust the false positives received.

For the adjustment of false positives and coarsen or refine the subscription space dynamically, the event space Ω is taken as an n-dimensional space where each subscription is a sub-space. Here n refers to the different attributes in the event space. Thus, for an event space with 2 attributes, the whole space will be a rectangle on the 2D plane and a subscription is a small rectangle within the plane. This looks similar to the event space division discussed in *Grid-Based Clustering*, however, it differs in an important aspect, that here the event space is always defined in a regular manner no matter what, so that coarsening or refining of the subscriptions and containment relations among the subscriptions is easily defined. Each of the subscription space is identified by means of a bit-string known as *dz* and the containment among the subscriptions are easily known just by the prefix matching of *dz*-expressions, e.g., *dz-0* contains *dz-00*.

The *dz*-expressions and event space partitioning is discussed in more detail in the next chapters as this thesis implements the same concept of encoding subscriptions from [22].

When a subscriber needs to reduce the number of events it is forwarding to other peers, it simply reduces the subscribed *dz* to a sub-space which allows less number of false positives, for example *dz-0* $\rightarrow$ *dz-00* and vice versa, thus keeping its bandwidth requirements and false positive rates under allowable margins.

Similarly, for delay requirements, the authors define an overlay protocol, under which the subscribers with tighter delay requirements are served before the subscribers with a relaxed delay requirement.

In another approach, Tariq et al.[23] discuss on providing probabilistic bounds on Quality-of-Service instead of guaranteed margins. This approach seems more realistic from a practical stand point, however it involves a complicated implementation and maintenance of the overlay to ensure the QoS. The subscribers, along with subscription, specify the expected quality of service along with a minimum probability for meeting these criteria. For the satisfaction of latency expectations at the peers, the authors propose dynamic algorithms where event distribution trees are created and updated in such a way that latency requirements of maximum number of subscribers are met.

Both the approaches present novel ideas for QoS specification at the subscriber end. The first approach however bounds the event space to a set of predefined attributes. Also in both the approaches, only two metrics, *viz.* bandwidth and delay are taken into consideration.

2.5 Conclusion

Many different aspects of pub/sub systems have been studied over the years and this chapter presented a few of them. From the point of view of the thesis, all the above discussed literature are relevant in some way or the other.

Particularly the content space division method discussed in QoS for pub/sub systems [22] has been used in this thesis for mapping content to the header of packets. Similarly the in-network filtering by LIPSIN gives an idea of how the performance might scale for the present implementation. The content-based router[19] has not been proven practical, otherwise, with the power of flexible flow programming by SDN, a fully content-based-router can virtually replace the application layered pub/sub system by its network layer equivalent.

Chapter 3

Problem Statement and System Model

Given the present state-of-art, discussed in the previous chapter, the following sections discuss how various issues of pub/sub systems such as filtering, content representation etc., are targeted and formulate the problem statement that the thesis studies in detail. First, an overview of the SDN infrastructure and OpenFlow protocol is presented, followed by the content space representation used in the present approach.

3.1 OpenFlow Protocol and Controller

As discussed briefly in the introduction of SDN in section 1.1, the SDN infrastructure primarily consists of a network of OpenFlow enabled switches and one or more external controllers, which program the forwarding switches as per application requirements. The controller and the switches follow the standard OpenFlow protocol for communication with each other. The protocol defines message specifications, commands and necessary syntax and semantics. Although discussion of intricate details of the protocol is irrelevant from this thesis's point of view, this subsection briefly summarizes the portions of the protocol which bear significant relevance.

The figure below (from [24]) shows where the OpenFlow protocol comes into picture with reference to an OpenFlow enabled switch.

(The protocol is still under active development and this work has used OpenFlow version 1.0 which lacks in some features such as IPv6 support.)

Flows and Flow Tables

The OpenFlow enabled switches are simple packet forwarding devices, which carry out the forwarding of incoming packets based on forwarding rules, also known as *Flows* present in

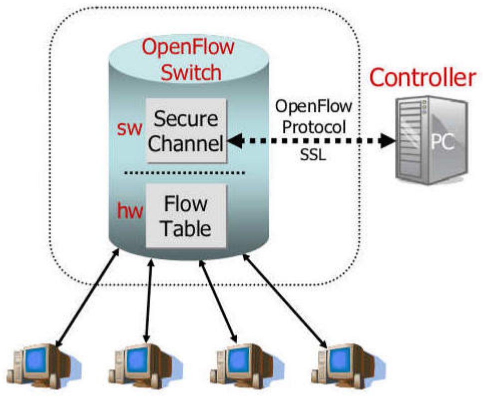


Figure 3.1: An OpenFlow switch overview

their flow tables.

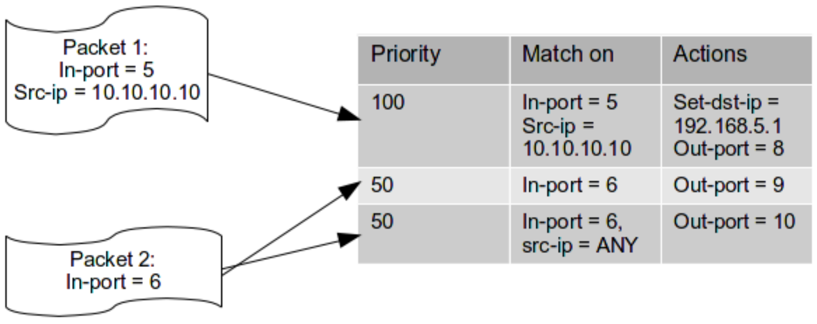
A flow consists of two predicates : a match rule and an action field. The match rule states the fields in the incoming packet header to be matched. For example, a match $< in\_port = 2 >$ will match any packet coming from the port numbered 2. Apart from the input port, other attributes from the packet header such as *source-ip, destination-ip, protocol, MAC address* etc., can also be specified as match fields.

The other predicate of the flow specifies what actions are to be taken when an incoming packet matches with the match rule. The actions could be to forward to specific port(s), change header fields, drop the packet and so on. Apart from the match headers and action list, a counter value for each flow is also stored in the flow table which counts the packets matched against that rule. OpenFlow specifies some constant values for special ports. Such as, to forward a packet to the same incoming port, output port in the action set is set to -8 or 65528 (unsigned). Similarly other constant values for FLOOD, Controller, DROP etc., are also defined.

Each flow in a switch also has priority associated with it. The flows are matched starting with the highest priority. The first flow rule that matches with an incoming packet is followed and corresponding action is taken. When multiple flows with same priority match an incoming packet, the behavior is switch dependent and hence care should be taken to avoid such scenarios. The figure 3.2 shows a simplistic example of flow matching in an OpenFlow switch.

As shown in the figure, the incoming packet 1 exactly matches with the first flow with priority 100 and therefore the corresponding actions will be applied to that packet. However for packet 2, there are two possible flows in the table at the same priority. It is an ambiguous situation and the behavior depends on the switch implementation.

The flows can also have an attribute, namely, *Timeout*, which specifies the time duration



Priority	Match on	Actions
100	In-port = 5 Src-ip = 10.10.10.10	Set-dst-ip = 192.168.5.1 Out-port = 8
50	In-port = 6	Out-port = 9

50	In-port = 6,
	src-ip = ANY
		Out-port = 10

Figure 3.2: Sample flow table in a switch

for which the flow will be active in the switch. After the timeout duration is passed, the flow is deleted from the switch. For this pub/sub implementation, the flows are installed with no timeout specifications and therefore are permanent.

Under the specification for version 1.0[25], when there is no matching flow for an incoming packet, the packet is forwarded to the controller. For example, for the above flow table, if a 3rd packet comes from input port 8, (or for example a packet from port 5 but with different source IP than 10.10.10.10) the packet will be forwarded to the controller. This thesis makes use of this fact to entertain advertisements, subscription requests and un-subscription requests.

OpenFlow Messages

OpenFlow specifies the type and structure of messages exchanged between the external controller and the switches. These messages can broadly be classified into three types as : Symmetric messages, Controller to switch messages and Asynchronous messages.

The *Symmetric Messages*, as the name suggests are those messages which are carried out in an handshake manner between the controller and the switch. These include Hello (`OFPT_HELLO`) messages, Echo request and reply messages among others. 'Hello' messages from the switches are sent at the time of start-up which lets the controller to be aware of presence of the switch.

Asynchronous Messages are unsolicited messages sent from the switch when there is no match for an incoming packet (in version 1.0), deletion of flow due to timeout or any other possible error.

Controller-to-switch Messages are usually the commands and requests sent from the controller to the OpenFlow switches, such as to add a new flow, delete a flow, retrieve specific switch information or statistics etc. Particularly, flow modification (flow-mod) messages are of relevance as they are used to set up the routes from publishers to subscribers.

It may be noted that OpenFlow uses port 6633/TCP for communication with the external controller.

Open vSwitch

Open vSwitch is a software switch which fully supports the OpenFlow protocol and hence is used for the studies in this thesis. Although the exact behavior would differ from a real hardware switch, Open vSwitch is still preferable as far as experimental analysis are concerned.

OpenFlow Controller

Along with the protocol and OpenFlow enabled switches, the controller application is an inherent component of the SDN infrastructure. The controller resides typically on a server and monitors the network and manages the programmable routes for packets. An SDN based network can have a set of controllers in a distributed manner, however in this project only a single centralized controller has been assumed. The controller holds the complete view of the switch topology and communicates with the switches using the said protocol.

Out of the many OpenFlow controllers available (e.g., Beacon[26], NOX[27], POX[28], Ryu[29] etc.) the Floodlight controller has been chosen for this thesis purposes as it has been under active development by a wide community of users and developers and also supports OpenFlow 1.0 protocol. More about the Floodlight controller is discussed in chapter 5.

3.2 Content Space Representation

The concept of a pub/sub system using SDN relies on the fact that the traditional filtering which used to happen in application layer software is now to be done by the switches in the network. This obviously raises the question as how to represent the message contents, so that the switches can actually read the content and forward them as necessary. Since the switches simply forward the incoming data based on forwarding rules, which in turn make use of header fields such as *ip-address, port number, protocol type, MAC address* etc., we are left with the problem of how to embed the message content into the header of the message, in such a way to facilitate filtering based on the header fields. The below presents the approach taken to tackle this issue.

As the packet headers can be treated as simple binary strings, the content of a message needs to be mapped to a binary string which can be represented in the form of a header field such as IP address. Although there are many possible conversion techniques available for representing data as binary string, e.g., using hash functions or bloom filters, the containment relation between the different subscription spaces in these methods is not preserved.

Discussed previously in the 2nd chapter, the thesis uses the content space representation model used in [22]. Essentially, this model represents the content space in an n-dimensional space where n is the different attributes in the content. Recursively, the entire content space is partitioned in a multidimensional binary tree and each subspace thus created is assigned a binary number.

Taking an example from [22] a content space Ω having two attributes such as *Area* and *Pressure* could be represented as a rectangle on a *2*-dimensional plane. As shown in the figure below, each axis of the rectangle corresponds to each attribute and the margins of the axes are bounded by the minimum and maximum values taken by the respective attributes.

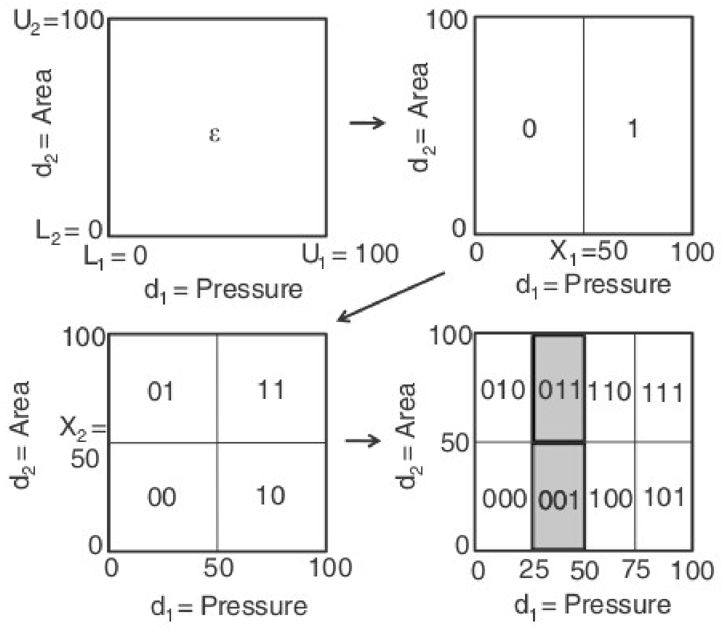


Figure 3.3: Spatial indexing

In such a representation of content space, an event ω corresponds to a well defined point and a subscription represents a rectangular subspace within the content space. Each of these sub-spaces are represented by a string of '0's and '1's known as dz-expressions. These expressions are formed by binary partitioning of the spaces along each dimension. As the space is divided more and more the length of the dz-expressions increases and so does the granularity of sub-spaces.

The subscriptions can correspond to one or more sub-spaces within the entire event space. For example, a subscription "s_1 = {Pressure = [25,50], Area = [0,100]}" maps to two sub-spaces with dz-expressions {001,011} while a subscription "s_2 = {Pressure = [25,50], Area = [0,50]}" maps to single subspace {001}.

This may be noted that as the number of attributes and hence the number of dimensions increase, the length of the dz-expressions also increase.

Containment Relation and Event Matching

The advantage of such a representation of subscription spaces is clear for the reason that it can easily identify containment relationships among subscriptions. For example a subscription space with dz-expression '00' contains the subscription space with dz-expression '000' and '001'. At the same time, the subscription spaces '001' and '000' share no common space between each other. This is implicit because, as the sub-spaces are divided, the newly created sub-spaces by default have their parent dzs as prefix. i.e., a subspace '0' divides to '00' and '01' leaving '0' as the prefix of both the newly created subspace.

Similar to containment relations among subscriptions, event matching in such a content space representation also becomes a straight forward process of prefix matching. An event ω matches a subscription s if the subscription space s contains the event point. That is, the dz of the subscription space should be a prefix in the dz of the event.

With the containment relation and event matching well defined, these dz-expressions can effectively be used for data representation and filtering purposes in the pub/sub system. As mentioned earlier, since the filtering is to be carried out directly by the forwarding switches, the dz-expressions thus created are mapped to a particular header field of the packet.

As suggested by [7], IPv6 addresses facilitate a large space for holding dz-expressions. However, since this work uses OpenFlow 1.0 which lacks in IPv6 support, in the present implementation IPv4 addresses have been used for this purpose.

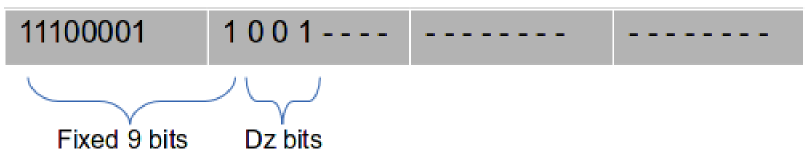


Figure 3.4: IP address structure

A fixed range of multicast IP range, 225.128.0.0 - 225.255.255.255 has been reserved specifically for pub/sub traffic. Clearly, the first 9 bits in the address are fixed, leaving the lower 23 bits for the dz-expressions. Formation of IP addresses from the dz values is thus a pretty straight forward process of string concatenation.

In this structure of IP addresses, a dz of {0010} is converted into an IP 225.144.0.0. The associated figure shows the IP address composition for a *dz* of {001}. Similarly, a *dz* of {00101} would result in an IP of 225.148.0.0. As the flows installed in the switches take care of the *dz* length to set CIDR style masks (discussed later), the containment relation is preserved in the flows. i.e., in the said example, *dz* {0010} contains the *dz* {00101} and accordingly 225.144.0.0/13 contains 225.148.0.0/14.

The filtering in the switches can take advantage of the IP address formats and their containment relationships to forward events to the necessary subscribers. As shown in the figure above, any event within the *dz* space of {001} would match 225.144.0.0/12.

3.3 Problem Statement

The objective of this thesis is to design and implement a content-based publish/subscribe system on network layer, using the capabilities of Software Defined Networking. With the resulting prototype, the thesis aims to study and analyze the effect on false positives and latency in the system for different numbers of publishers and subscribers.

Chapter 4

Content-Based Filtering and Routing

Before going into the details of implementation, this chapter presents the concepts and algorithms used in the design.

For the routing of messages from publishers to the subscriber, routing trees must be constructed using the network topology. Routing trees are nothing but spanning trees drawn over the topology graph so as to visit every switch exactly once. This is of course a fundamental requirement for removal of loops in the message forwarding route and ensure *exactly-once* delivery of messages. Traditionally, this is achieved by distributed protocols such as *OSPF*[30]. However, with the decoupled data and control plane in SDN, this reduces to a simple graph problem on the controller application.

As the SDN controller is informed about the entire network topology, construction of a routing tree is a straight forward problem of graph traversing which can be performed with any well known off-the-shelf algorithms. The present approach implemented in this design, uses publisher driven routing trees, which means routing trees are being constructed with the publisher's location taken as root, otherwise known as source-rooted trees.

The routing overlay tree construction addresses one of the problems. The other being formation of flow commands for switches along the entire path from the publisher to relevant subscribers and modifying their flow tables. These issues and some other, related to the system dynamics are discussed in the following sections. The algorithm used for the addition and removal of flows is similar to what suggested in [7].

4.1 Pub/Sub Tree Creation

Pub/sub trees are routing trees mapped to particular dz-expressions. i.e., corresponding to dz $\{00\}$ there might be one tree and corresponding to dz $\{01\}$ there might be another.

A tree with a particular *dz* takes care of the routes for all the subscriptions equal or less than its *dz*.

The formation of pub/sub trees are publisher driven, i.e., whenever a publisher comes, it creates a new tree or joins an existing tree. Also, the trees are created for each new *dz*. The below explains it in detail.

Whenever a new publisher comes, it notifies its intention to publish by sending an advertisement. The message formats for advertisements are explained in the next chapter. Upon reception of an advertisement, the routing module in the controller can search for available trees to which the publisher can join, or in case no tree is constructed, a new tree is created with this publisher as its root node. This is done by means of a simple *Breadth First Search* on the topology graph.

A new publisher joins an existing tree if and only if the *dz* of the tree covers the *dz* of the publisher. If the *dz* of the newly found publisher covers the *dz* of the tree, it joins the tree as well as creates new sub-trees corresponding to the sub-spaces within its *dz* which are not covered.

The example below illustrates this process.

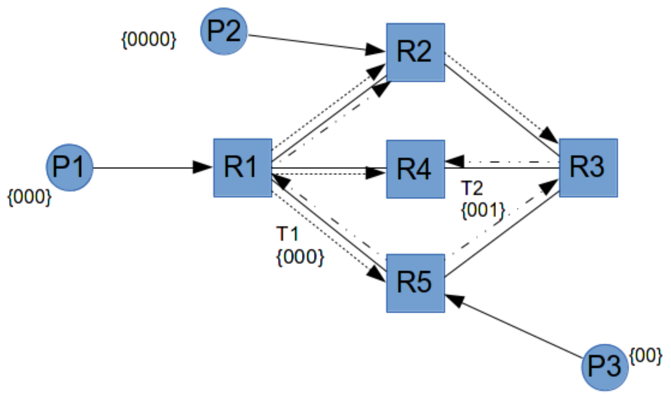


Figure 4.1: Tree creation example

In the above figure, assuming no trees have been constructed, the arrival of a publisher P1 with a *dz* of {000} creates a tree of switches mapped to the *dz* {000}. When a new publisher comes with a *dz* equal or less than that of the tree (e.g., P2 with *dz* {0000}) it joins the existing tree. The dashed lines in the figure, shows the tree 'T1' created with P1 as root.

When a new publisher comes with a *dz* which covers the *dz* of the existing tree (e.g., P3 with {00}), its *dz* is split into {000} and {001}. Therefore it joins the existing tree 'T1' as well as creates the other sub-tree 'T2' with *dz* of {001}

The algorithm for handling advertisements is as shown below.

Algorithm 4.1 Advertisement Handling

```
procedure ADVERTISEMENT($publisherStruct, dz$)
    ▷ $publisherStruct$ is the switch port tuple denoting the location of publisher in the network
    $t \leftarrow \{\text{tree: } dz_{tree} \subset dz \text{ or } dz \subset dz_{tree}\}$
    if $t = \emptyset$ then
        $tree \leftarrow$ BuildTree($publisherStruct, dz$)    ▷ Create tree with this publisher as root
        addFlowForSubs($tree, dz, publisherStruct$)
        return
    end if

    for each $tree$ in $t$ do
        if $dz_{tree} \supseteq dz$ then    ▷ Parent tree
            $tree$.JoinTree($publisherStruct, dz$)
            addFlowForSubs($tree, dz, publisherStruct$)

        else if $dz_{tree} \subset dz$ then
            $newKeys \leftarrow$ getNewKeys($dz$)
            for each $key \in newKeys$ do
                BuildTree($publisherStruct, key$)
                addFlowForSubs($tree, dz, publisherStruct$)
            end for
        end if
    end for
end procedure
```

As seen in line 16 in 4.1, in the presence of a tree with a lesser dz, it creates all the trees with covered dzs which are not present. i.e., in presence of trees with dzs {000, 00101}, the arrival of a publisher with advertisement space {0} will create trees with dzs {010, 011, 00100, 00110, 00111}. This is a simple recursive process which is given in the pseudocode below.

Algorithm 4.2 Get *dz*s for new SubTrees

```
1:  procedure GETNEWKEYS($dz$)
2:      $d \leftarrow \{dz : dz \supset dz_{tree}$ For $tree \in treeList\ \}$
3:      if $d = \emptyset$ then return $dz$
4:      end if
5:      $l \leftarrow d[0]$.length - $dz$.length
6:      for $i \in range(0, 2^l)$ do
7:          $temp\_dz \leftarrow dz +$ intToBoolean($i, l$)        ▷ String concatenation
8:          if $\exists tree \in treeList : dz_{tree} = temp\_dz$ then continue
9:          end if
10:         $dz\_list \leftarrow \{dz_{tree} :$ For $tree \in treeList, dz_{tree} \subset temp\_dz\ \}$
11:         if $dz\_list \neq \emptyset$ then
12:             $newKeys$.addAll(GetNewKeys($temp\_dz$))
13:         else
14:             $newKeys$.add($temp\_dz$)
15:         end if
16:     end for
17:     return $newKeys$
18: end procedure
```

Apart from creating new trees or joining the existing trees, the advertisement handler should also take care of addition of flows to the relevant subscribers. These are shown in the given algorithm 4.1 in line numbers 6, 13 and 19. The process of flow addition for relevant subscribers is given in later subsections.

A point to be noted here is that in line number 3 in the algorithm 4.1, we get the relevant trees for a given *dz*-string. This ideally refers to all the trees with *dz* values subset of the new *dz* or the other way round. i.e., trees with *dz*s $\{000\}$ and $\{0\}$, both are relevant for the *dz* $\{0\}$. However, since the tree creation is publisher driven and new publishers with higher *dz*s containing existing *dz*s are split, trees with *dz*s $\{000\}$ and $\{00\}$ cannot coexist. That means the existing trees at any given time, have *dz*s unrelated to each other. This simplifies flow addition to a great extent as a tree with *dz* $\{00\}$ takes care of all the flows for subscriptions lesser than or equal to $\{00\}$. The flows will be discussed in later sections.

4.2 Addition of Subscriber

The process of addition of subscribers is somewhat similar to addition of publishers, except for the fact that a new incoming subscriber does not trigger the creation of routing trees. When no tree has been constructed yet, the subscriber data is stored in the controller and is used only when a relevant publisher comes.

Algorithm 4.3 Subscription Request Handling

```
 1: procedure SUBSCRIPTION($subscriberStruct, dz$)    ▷ SubscriberStruct contains
    SwitchID and Port at which Sub is attached
 2:     $subscriberList$.add($dz, subscriberStruct$)
 3:     $t \leftarrow \{$tree: $dz_{tree} \subset dz$ or $dz \subset dz_{tree}\}$
 4:     if $t = \emptyset$ then return                  ▷ No publishers yet
 5:     end if
 6:     for each $tree \in t$ do
 7:         $pub \leftarrow tree$.getPublishers()
 8:         for each $publisher \in pub$ do
 9:             addFlowForSub($tree, dz, publisherStruct, subscriberStruct$)
10:         end for
11:     end for
12: end procedure
```

If any relevant routing trees are already available, routes are constructed from publishers to the subscriber and flows are added accordingly. The subscriber joins all the relevant trees. e.g., for a subscriber with *dz* {0} should join the trees {00,01}. The algorithm to handle new subscription requests is shown above.

4.3 Routing and Flow Modifications

As mentioned earlier, once the trees are created, routes must be calculated from publishers to the subscribers. The route calculation in a pub/sub tree from publisher to subscriber is a direct process of finding the lowest common ancestor as we move towards the root from both the end nodes. This algorithm is a standard procedure in tree parsing and hence the pseuodocode is not produced here.

As the route from the publisher to the subscriber is available, flow is added in each of the switches along the path. The formation of flow structures and addition of flows is as explained below.

4.3.1 Flow Structures

The flow structures are formed according to the OpenFlow specifications. Particularly, match fields and action rules are taken into consideration to define the flows.

- **Match Fields**

  As mentioned earlier in the previous chapter the content space which is represented in the form of *dz*-expressions are used as IPv4 addresses to define the flows. Since the first 9 bits of the IP addresses are fixed, the *dz*-expressions are confined to remaining 23 bits. Formation of such IP addresses is a simple process of string concatenation.

  Thus, the match fields for the flows are taken as *input-port* and *destination-ip* addresses.The publisher sends the events to particular IP address formed by concatenating the first fixed 9 bits with the event's *dz*. The flows are added corresponding to its advertised *dz* with a mask value, so that all the events under this *dz* space are matched to the flow. i.e., For a *dz* of {00} the flow match rule would specify the destination IP as 225.128.0.0/11 so as to match any event under {00}.

- **Action Set**

  The route from a publisher to a subscriber, calculated in a tree is simply an ArrayList of switch-port tuples starting from the publisher and ending at the subscriber. Thus, for any switch, the flow action set simply specifies the list of output ports on which the packet must be forwarded.

  The action fields also add a rule of setting the destination IP address to the IP address the subscriber is listening at, if the switch is a terminal switch. This is discussed in the next subsections.

- **Flow Priority**

  As mentioned earlier, the OpenFlow switch takes the action corresponding to the first match it finds. Therefore, for multiple subscribers with containment relationships in their subscription spaces, priority should be added to the flows so that all the subscribers can get their data of interest.

  For example, a flow for {00} to output port 1 should not interfere with another flow for {000} to output port 2. If both the flows are kept at the same priority, an incoming event will be forwarded to only one of them which is unpredictable and depends on the switch implementation. Therefore, setting appropriate priorities for such situations is mandatory. The next sections discuss this in detail.

4.3.2 Flow Addition

As seen in the figure, there are two subscribers with their subscription spaces, as well as the IP addresses at which they are listening for events. The publisher P1 has an event space of {00} to which the subscribers are subscribed. The port numbers of the switches

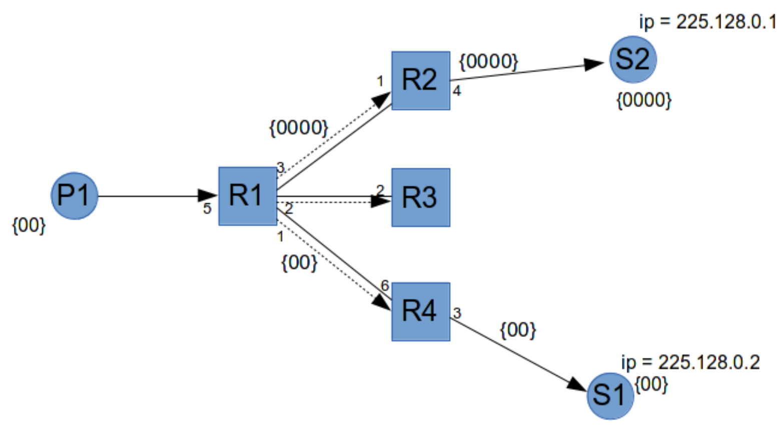


Figure 4.2: Flows in a tree

have also been specified at which they are connected with each other. The flow addition process could be explained taking different scenarios.

For no existing flow

In the beginning, assuming no flows have been added, the arrival of the subscriber S1 triggers the flow addition process. The subscription space {00} of S1 matches to that of the publisher and upon identifying the route, the flows are added as :

For switch R1:

{
    Match : input-port = 5, destination-ip = 225.128.0.0/11
    Action : output = 1
}

For switch R4 :

{
    Match : input-port = 6, destination-ip = 225.128.0.0/11
    Action : set-destination-ip = 225.128.0.2, output = 3
}

As shown, since the switch R4 is a terminal switch, that means no further switches are attached, the destination IP of the packet is changed to the IP at which the listener is listening and then it is forwarded to the given port.

Another factor which needs to be checked at the terminal switches, is if the publisher and the subscribers are running on the same host, the packet must be forwarded to the *ingress-*

port. This is done by setting output port to 65528 as defined by OpenFlow. In Json style flow-mod commands, it can also be specified as <output = ingress-port>.

Flows for multiple subscribers

Upon arrival of a new subscriber S2 with a subscription space of {0000}, the flows are added choosing the minimum *dz* between the *dz* space of publisher and that of subscriber. That means, flows for the subscriber S2 must be added for {0000}. Similarly, if a new subscriber S3 had come with a subscription space of {0}, the flows would have been added for a space of {00} as the publisher's *dz* is contained within the subscriber's *dz*.

Priorities must also be considered at the switch R1 while adding flow for the subscriber S2, as :
i) Events matching {0000} must be forwarded to both S1 and S2
ii) Events matching {00} should be forwarded only to S1

With the two requirements in place, we need 2 flow rules on switch R1 to satisfy both these requirements. These can be shown as :

Changed Flows at switch R1 after addition of S2 :

```
{
    Match : input-port = 5, destination-ip = 225.128.0.0/13
    Action : output = 3, output = 1
    Priority = 1
}
{
    Match : input-port = 5, destination-ip = 225.128.0.0/11
    Action : output = 1
    Priority = 0
}
```

As shown, the priority of the rule for lower *dz* {0000} is set higher than that of the greater *dz* {00}. And all the relevant subscribers for the higher *dz* are also added to the output list of the rule for lower *dz*.

Another point of interest while forming flow structures is the order of the action set. Considering a non-terminal switch, which forwards the packets to other switches and not to other hosts, the action set only consists of a list of output ports. However, for a terminal switch, the order of actions in the action set should be added in such a way to set the destination IPs before forwarding on to the ports. And also, this should be done after forwarding the packet to other switches.

As an example, given that a subscriber is connected to a switch at port 2 and listens at an address 225.128.0.5 and another two switches are connected at port 3 and 4, the output action set for such a flow would look like

{
    Action: Output = 3,output = 4, set-destination-ip = 225.128.0.5, output = 2
}

Flow up/down-gradation

Another point of concern in flow modification is upgrading or downgrading of the flows in case of addition of new subscribers or removal of subscribers.

Extending the above figure, assuming a new subscriber S3 is introduced in the same route to S2, this triggers the up-gradation of the existing flow in certain switches. This is shown in the figure below.

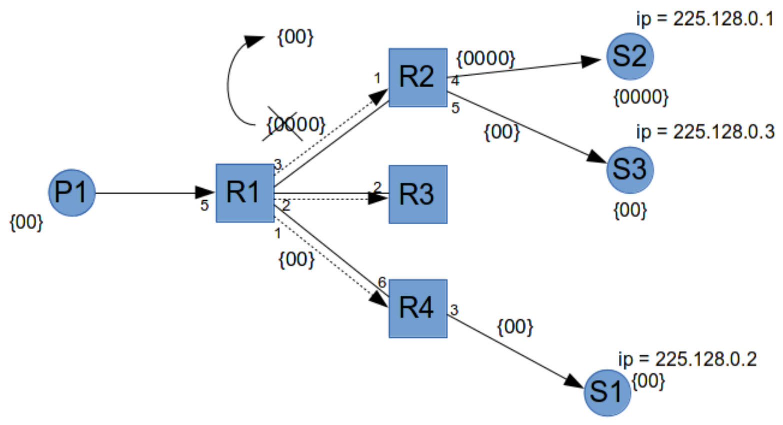


Figure 4.3: Flow upgrade

As seen, at the switch R1, the existing flow for {0000} is now covered by the new request of {00}. Therefore, the existing flow must be deleted and a new flow for the higher dz space is added. Accordingly, at the switch R2, new flows need to be inserted with appropriate priorities to take care of both the subscribers.

Another point to be noted here is, if the subscriber S3 had come before S2, no flow would need to be added in the switch R1, as the existing flow already would have covered S2's subscription space.

Flow down-gradation is the exact reverse process of up-gradation, which occurs when a subscriber is removed. For example, if the subscriber S3 sends an un-subscription request, the flow at R1 is restored to {0000} as the subspace {00} is no more necessary along that route.

Formally, the following points drive the process of flow addition at a particular switch :

- If no flow is present, the new flow is added.
- If existing flow covers the new flow, nothing needs to be done. i.e., existing flow {00} → <port 2> removes the need of adding {0000} → <port 2>.

  Formally, a flow f_1 fully covers a flow f_2 if *dz* of f_2 is contained in *dz* of f_1 and also, the output ports of f_2 are present in the output ports of f_1. i.e., :

  $$f_1 \succ f_2 \text{ if } \begin{cases} dz_{f_1} \supseteq dz_{f_2} \\ \text{outPorts}_{f_2} \in \text{outPorts}_{f_1} \end{cases}$$

- If the existing flow is covered by the new flow requested, the new flow is added and the existing one is deleted as it is no longer needed.
- If existing flow's *dz* is same as the new requested flow, but the output ports differ, i.e., existing flow {00} → <port 2> and new requested flow {00} → <port 3> , then the existing flow is updated (deleted and added again) with new set of output ports as {00} → <port 2, port 3>
- If existing flow's *dz* covers the new flow's *dz* however they have different output ports, i.e., existing flow {00} → <port 2> and new flow {000} → <port 3>, then the new flow must be added with a higher priority and should include the output port of the existing flow. i.e., the new flow should be added as {000} → <port 2, port 3> with a priority more than the existing one.

  For the above two points, a partial covering relation can be defined as, a flow f_1 partially covers a flow f_2 if *dz* of f_2 is contained in *dz* of f_1 but all the output ports of f_2 are not listed in the output ports of f_1. i.e., :

  $$f_1 \succsim f_2 \text{ if } \begin{cases} dz_{f_1} \supseteq dz_{f_2} \\ \exists port \in \text{outPorts}_{f_2} : port \notin \text{outPorts}_{f_2} \end{cases}$$

- If the existing flow's *dz* is covered by the newly added flow, however they have different set of output ports, the existing flows must be updated with new set of output ports along with appropriate priorities and then the new flow is added. This is similar to the 3rd point mentioned above.

The pseudocode to add flows along a route for a particular *dz* is shown below.

Algorithm 4.4 Flow Addition

```
1: procedure FLOW ADDITION($dz, route, tree$)
2:     for $i$ in $route$.size() do
3:         $swid \leftarrow route$.get($i$).getNodeId()
4:         $inPort \leftarrow route$.get($i$).getPortId()
5:         $outPort \leftarrow route$.get($i+1$).getPortId()
6:         $existingFlow \leftarrow tree$.getFlowForSwitch($swid$)
7:         $outputPorts$.add($outPort$)

        ▷ Priority is by default set to 16384, which is the mid value of the allowed range
8:         $Flow \leftarrow$ createFlowStructure($swid, inPort, outputPorts, priority$)
9:
10:                            ▷ If this is a terminal switch, set destination IP for the listener
11:        if $i = route$.size() - 2 then
12:            $Flow$.setDstIP(getIPForSub($swid$), $outPort$)
13:        end if
14:                                                  ▷ If this is the only flow, just add the flow
15:        if $existingFlow = \emptyset$ then
16:            $tree$.addFlow($Flow$)
17:            PushFlowToSwitch($Flow, swid$)
18:        end if
19:                                               ▷ If this flow is already covered, nothing to do
20:        if $\exists f \in existingFlow : f \succ Flow$ then
21:            continue
22:        end if
23:                       ▷ Delete the flows which are not needed if this new flow covers them
24:        for $flow \in existingFlow$ do
25:            if $Flow \succ flow$ then
26:                deleteFlowFromSwitch($flow, swid$)
27:            end if
28:        end for
29:                    ▷ Add output ports of all the flows which are having higher $dz$ than this
30:        for $flow \in existingFlow$ do
31:            if $flow \gtrapprox Flow$ then
32:                $port \leftarrow \{port : port \in \mathrm{OutPorts}_{flow} \land port \notin \mathrm{OutPorts}_{Flow}\}$
33:                $outputPorts$.add($port$)
34:            end if
35:        end for
36:        $Flow$.setOutputPorts($outputPorts$)
37:
38:        $prio \leftarrow$ getMaxPrio($existingFlow$)          ▷ Set appropriate priority of the flow
39:        $minDz \leftarrow$ getMinDz($existingFlow$)
40:        $prio \leftarrow prio + dz$.length - $minDz$.length
41:        $Flow$.setPriority($prio$)
```

Algorithm 4.4 Flow Addition (continued)

```
42:                                         ▷ Update the flows with lesser $dz$ than this flow
43:         for $flow \in existingFlow$ do
44:             if $Flow \gtrapprox flow$ then
45:                 $flow$.addOutPort($outPort$)
46:                 deleteFlowFromSwitch($flow$)
47:                 PushFlowToSwitch($flow, swid$)
48:             end if
49:         end for
                                            ▷ Push this new flow to the switch
50:         PushFlowToSwitch($Flow, swid$)
51:         $tree$.addFlow($Flow$)
52:         $i \leftarrow i + 2$
53:     end for
54: end procedure
```

The flow structure in line number 8 is created as per the specifications of the controller used which is discussed in the next chapter. The covering relations between the flows in line numbers 20, 25, 31 and 44 are checked based on the *dz*s and set of output ports of the ports as mentioned previously.

4.4 Un-Subscription

When a subscriber requests for un-subscription, the subscriber is removed from all the associated trees. For each associated tree, the flows are either deleted or down-graded depending on other interested subscribers attached to that particular switch. The following diagram illustrates this procedure, continuing with the same configuration as in previous sections.

The un-subscription process recursively moves from the terminal switch towards the publishers in the tree. This is similar to a classic depth first search in a tree. When the subscriber S1 sends an un-subscription request, the corresponding flow in the switch R4 is deleted. As there is no other subscriber attached to the switch, it moves to the switch connected at the port which is the input port of the deleted flow.

i.e., After deleting the flow in switch R4, we move to the switch connected at the port 6, which is R1 and repeat the same process. This is straight forward case when there are no other subscribers present.

Considering the case of multiple related subscribers, the flow downgradation process stops

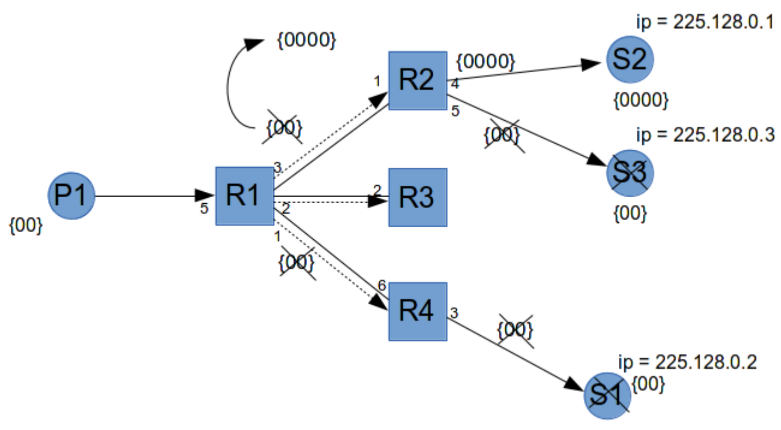


Figure 4.4: Flow down-gradation during un-subscription

at the switch when there is at least one subscriber which covers the *dz* of the un-subscribed *dz*. For example, if the subscriber S2 asks for un-subscription, the corresponding flow in the switch R2 is deleted. However, since there is another subscriber S3 present at the same switch with a *dz* {00} more than the un-subscribed one {0000}, the process stops there.

However, if the subscriber S3 requests for un-subscription, after deleting the corresponding flow in the switch R2, it must move to the switch R1 with the request of un-subscription for {00} and subscription for the next highest *dz* space required, i.e {0000} in this case. In addition to deletion of the flow, care should be taken for updating the existing flow's output ports.

For example, initially the flows at the switch R2 could be :

```
Flow 1 :
{
    Match : input-port = 1, destination-ip = 225.128.0.0/13
    Action : output = 4,5
    Priority = 1
}
Flow 2:
{
    Match : input-port = 1, destination-ip = 225.128.0.0/11
    Action : output = 5
    Priority = 0
}
```

On removal of the subscriber S3, the flow 2 should be deleted and at the same time, the output port list in flow 1 should be updated to just port number 4. This involves a deletion and addition of flow.

The recursive process along a path stops when either it reaches the publisher, or it reaches a switch with subscribers asking for the same or greater subscription space. The algorithm for handling un-subscription request for a particular tree is produced below.

Algorithm 4.5 Unsubscription Handling

```
1: procedure UNSUBSCRIBE($dz, subscriberStruct, tree, newDz$)
2:     $swid \leftarrow subscriberStruct$.getNodeID()
3:     $outPort \leftarrow subscriberStruct$.getPortID()
4:     $existingFlow \leftarrow tree$.getFlowsForSwitch($swid$)
5:     if $existingFlow = \emptyset$ then return
6:     end if
7:                     ▷ Get the flows to be deleted – covered flows with the same out-port
8:
9:     $flowsForDeletion \leftarrow \{f \in existingFlow\colon outPort \in \text{OutPorts}_f \wedge newDz \subseteq dz_f \subseteq dz\}$
            ▷ For each of these flows, remove this particular output port from the output list
10:
11:    for each $flow \in flowsForDeletion$ do
12:        $newFlow \leftarrow$ createFlowStructure($swid, inPort, outputPorts, priority$)
13:        $newFlow$.removeOutputPort($outPort$)
14:        deleteFlowFromSwitch($flow$)
                ▷ If there are other subscribers with same $dz$, no need to traverse in that direction
15:        if $dz_{flow} \neq dz$ then
16:            $inPorts$.add($flow$.getInPort())
17:        end if
18:                 ▷ If this was the only subscriber, add new flow if $newDz$ is not null
19:        if $newFlow$.getOutPorts() $\neq \emptyset$ and $newDz \neq \emptyset$ then
20:            $newFlow$.addOutPorts($outPort$)
21:            $newFlow$.setDz($newDz$)
22:            PushFlowToSwitch($newFlow, swid$)
23:        else        ▷ If the flow is covered by any existing flow, no change is needed
24:            if $\exists F : F \succ flow$ then continue
25:            end if
                                        ▷ Add new flow with updated output ports
26:            PushFlowToSwitch($newFlow, swid$)
```

Algorithm 4.5 Unsubscription Handling (continued)

```
27:         end if
28:     end for
                                        ▷ Add new flow if $newDZ$ is not null
29:     if $newDz \neq \emptyset$ then
30:         $newFlow \leftarrow$ createFlowStructure($swid, inPort, outputPorts, priority$)
31:         $newFlow$.setDZ($newDz$)
32:         PushFlowToSwitch($newFlow, swid$)
33:     end if
34:               ▷ For each input port, get max required $dz$ and recursively un-subscribe
35:     for each $port \in inPorts$ do
36:         $maxDz \leftarrow$ getMaxIncomingDz($flowsForDeletion$)
37:         $maxDz\_required \leftarrow$ getMaxIncomingDz($existingFlow - flowsForDeletion$)
38:         if $maxDz\_required \supset maxDz$ then continue
39:         end if
40:         $nextSwitch \leftarrow$ getAttachedSwitchAtPort($swid, port$)
41:         $MinDz \leftarrow \max\{newDz, maxDz\_required\}$
42:         Unsubscribe($maxDz, nextSwitch, tree, MinDz$)
43:     end for
44: end procedure
```

4.5 Un-Advertisements

The un-advertisement requests from publishers are handled in the same manner as un-subscription requests. As an un-advertisement request is received, for all the associated trees, we traverse in a depth first search manner, removing or down-grading the flows from the switches as they are encountered. As before, the traversing stops at the points where alternate publishers are available or when the switch is a terminal switch.

This is illustrated with the figure 4.5.

In the diagram, the publisher P1 with a dz space of {00} creates the tree and publisher P2 with dz space {0000} joins the existing tree. Upon the joining of P2, appropriate flows are added to the subscribers S1, S2 and S3. However, as the flows from existing publisher P1, covers the new flows from publisher P2, only one new flow needs to be added from switch R4 to R1.

When the publisher P2 sends an un-advertisement message to the controller, recursively the flows are investigated in all directions starting from switch R4. The process stops when some other alternate publisher with a dz higher or equal to the dz of P2 is found or if it is a terminal switch.

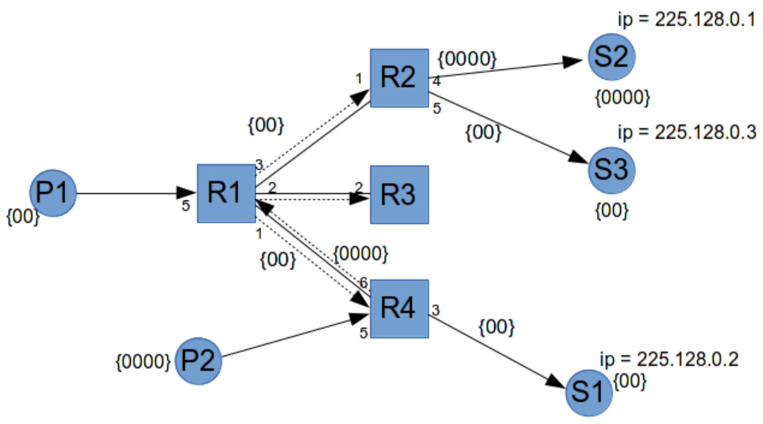


Figure 4.5: Multiple publishers

For example, in the direction from R4 towards R1, the flow from R4 to R1 needs to be deleted when the publisher P2 sends an un-advertisement message. But, at the switch R1, an alternate publisher P1 is available which sends events to the same output ports, in a *dz* space {00} more than the *dz* space {0000} of P2. Therefore the flow removal process stops at that switch.

Similar to the un-subscription process, flow down-gradation should also be done in case of un-advertisements when multiple publishers are available. This is shown with the associated figure below, with the same network topology as above.

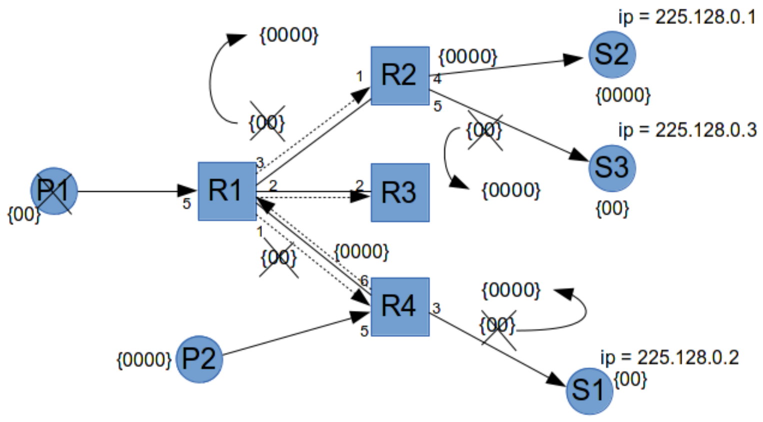


Figure 4.6: Unadvertisement

As shown, the publisher P1 sends an un-advertisement message before publisher P2. In such a case, the flows are updated to the highest incoming *dz* before moving on the next switch. i.e., in the direction from R1 to S1, first the flow from R1 to R4 is deleted and then, the flow from R4 to S1 is down-graded to the maximum incoming *dz* which is {0000} in this example.

Likewise, flow down-gradation is done along the route to the subscribers S2 and S3.

Formally, the algorithm for un-advertisement handling for a specific tree is presented as below.

Algorithm 4.6 UnAdvertisement Handling

```
procedure UNADVERTISE($dz, publisherStruct, tree, newDz$)
    $swid \leftarrow subscriberStruct$.getNodeID()
    $inPort \leftarrow subscriberStruct$.getInPort()
    $existingFlow \leftarrow tree$.getFlowsForSwitch($swid$)
    if $existingFlow = \emptyset$ then return
    end if
                              ▷ Get the flows to be deleted – covered flows with the same in-port
    $flowsForDeletion \leftarrow \{f \in existingFlow\colon inPort = inPort_f \wedge newDz \subseteq dz_f \subseteq dz\}$
    for each $flow \in flowsForDeletion$ do
        deleteFlowFromSwitch($flow, swid$)
        $outportList$.addAll($flow$.getOutPorts)
    end for
                                                         ▷ Add new flow if $newDZ$ is not null
    if $newDz \neq \emptyset$ then
        $newFlow \leftarrow$ createFlowStructure($swid, inPort, outputPorts, priority$)
        $newFlow$.setDZ($newDz$)
        PushFlowToSwitch($newFlow, swid$)
    end if
                  ▷ For each output port, get max outgoing $dz$ and recursively down-grade
    for each $port \in outPorts$ do
        $maxDz \leftarrow$ getMaxOutgoingDz($flowsForDeletion$)
        $maxDz\_required \leftarrow$ getMaxOutgoingDz($existingFlow - flowsForDeletion$)
        if $maxDz\_required \supset maxDz$ then continue
        end if
        $nextSwitch \leftarrow$ getAttachedSwitchAtPort($swid, port$)
        $MinDz \leftarrow \max\{newDz, maxDz\_required\}$
        Unadvertise($maxDz, nextSwitch, tree, MinDz$)
    end for
end procedure
```

4.6 Conclusion

This chapter discussed the algorithms, system dynamics, flow structures and presented pseudocodes for the prominent parts of the pub/sub implementation. The algorithm realizations are discussed in the next chapter with the message formats used, designs and an overview of the controller application.

Chapter 5

Implementation

With the algorithms and relevant pseudocodes formulated for various processes, this chapter moves forward to their actual realization to build up a fully functional controller application. In addition to the algorithms discussed before, other necessary components include publisher and subscriber applications running on the host systems.

Other points of concern include decision over standard message formats for communications between a publisher/subscriber application and the controller module. Similarly, data structures and design of objects for efficient implementation of the previously discussed algorithms are also of great importance.

The following presents the implementation related details of various components of the pub/sub system.

5.1 Publishers and Subscribers

The publishers and subscribers are implemented as a set of simple python scripts. Python has been chosen for this purpose for its ease of implementation and the availability of rich libraries particularly for text processing.

The publisher's advertiser script reads the advertisement data file, which could be a stock quote file for example, and generates the *dz*-expressions for the event spaces using spatial indexing technique. The algorithm to generate the *dz*-expressions has been taken from [22] and hence it is not reproduced here again. Sample stock quote data is shown below for advertisement/subscriptions and events.

advertisement/subscription = [name='AMZN'][80<low<100]
event = [name='AMZN'][low=92][high=234]

As seen, the advertisements and subscription requests are cubes in a *3*-dimensional space whereas the events are well defined points. The *dz*-expressions for each of them are generated by specifying their boundaries along each dimension (e.g., 80 and 100 are the limits for the dimension 'low'). For events, the upper and lower limit along each dimension is set to the same value.

Thus generated *dz*-expressions are then sent to the controller in a particular message format (discussed in the next section) as *advertisements*. As mentioned in section 3.1, when no flow is installed in the switch's flow table, the packet is sent to the controller for processing. The publishers and the subscribers take advantage of this fact and send their advertisements or subscription requests to a fixed IP, 225.0.0.37 for which no flow is installed in any of the switches. The packets are sent as standard UDP datagrams.

It can be noted that a publisher can get repeated *dz*-expressions for various data sets. To refrain from sending advertisements for the same *dz* again and again, the sample application sorts and removes the duplicate and unnecessary *dz*s before sending.

Similar to the advertisements, *dz*-expressions for every event is computed. But, unlike advertisements, the events are not sent to the fixed IP. For each event, corresponding IP addresses are formed and the events are sent on to that IP address. Formation of IP addresses from *dz*-expressions have been discussed in the system model chapter. No particular message structure has been defined for the events and they are simply sent over the socket as plain byte streams.

Likewise, a subscriber notifies the subscriptions in the same message format as advertisements. The program reads the input data file, generates *dz*-expressions for its subscription spaces just like advertisements and sends it over the socket as a UDP packet to the fixed IP address. Apart from notifying its subscriptions, the subscriber also sends the IP address at which it is listening for events. This address is used by the controller to deliver messages to the subscriber.

A listener application on the subscriber side, listens at the notified IP address for any incoming events. Alternatively, the listener application could have been implemented as to listen at all the addresses formed by the subscribed *dz*-expressions. However, for a large number of subscription spaces, the number of multicast addresses to which it must listen becomes too huge and is not practical. The listener application receives the data from the socket and a different thread writes the data in a text file for later processing. As the events are sent as plain byte stream, the received data is interpreted in the same manner and is stored in the text files.

Similar to the sample publisher application, the subscriber also removes the duplicate *dz*-strings which it might have generated for different data sets.

Value	Type of Message
0	Advertisement
1	Un-Advertisement
2	Subscription request
3	Un-Subscription request
4	IP address at which the subscriber is listening

Table 5.1: Types of messages

Summarizing, the publisher hosts run separate scripts for sending advertisements and events. Similarly, the subscriber hosts run different scripts for making subscription requests, notifying its IP address and to listen for incoming events. Apart from these, un-advertisements and un-subscription requests are also done through python scripts, which differ from advertisements and subscription requests only in the message formats.

5.2 Message Formats

The messages in the pub/sub network can be categorized into five types as : i) Advertisement ii) Subscription Request iii) Un-advertisement and iv) Un-subscription v) Generated Events.

Except for the events, all other messages are directed towards the controller and it is required by the controller to identify the publishers and subscribers. As mentioned in the previously, a fixed IP 225.0.0.37 has been reserved for this purpose and hence, all these messages are sent to this address which are then entertained by the controller.

A uniform format has been chosen to represent all these types of messages. The publishers or subscribers need not notify their IP addresses or locations to the controller, as the controller automatically gets the switch and port from where the packet has been received. Therefore, the only information which is passed to the controller in these messages are: type of message, dz and length of the dz.

The messages thus have 2 fields fixed, which carry the information 'type' and length of the dz-expressions. The type could be denoted by 2 bits as there are only 4 types of messages and the length of the dz could be denoted by 5 bits assuming maximum length being 23 bits in the present implementation.

However, for extension to IPv6 support the message formats might need to be changed again. Also, the subscribers need to notify the controller, the IP address they are listening at. Therefore to keep things simple, 2 bytes have been used for the representation of type of message and the length of the dz.

Following the header, the message carries a buffer of 3 bytes and then 3 bytes for denoting the dz value. The messages have been kept of fixed length for the sake of simplicity and can be altered to varying length messages if needed.

The table 5.1 shows the mapping of the types to integer values.

As an example, for an advertisement of dz {001}, the message would simply be a byte array with the appropriate values as :

$$\text{advertisement message} = \{(\text{type})\ 0\ ,(\text{length of } dz)\ 3,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1\ \}$$

As the publishers and subscribers use python scripts for all their purposes, creation of such a message is achieved by using python's `struct` feature.

The controller parses the messages to identify publishers, subscribers and their associated dz values.

5.3 Controller Application

The core of this thesis is the controller module for pub/sub system. This controller application maintains the topology of the network, manages the publishers and subscribers, forms the flow structures and adds route from publisher to the interested subscribers. These functionality will be discussed in subsequent subsections starting with an overview on the controller used.

5.3.1 Floodlight Controller

As mentioned earlier, this work has chosen the Floodlight Controller (version 0.9) for its active user and developer group and also for the ease of implementation. The controller is released under Apache License[31] and is an open source platform. The controller is implemented in Java and can be easily integrated with any development environment such as Eclipse.

Floodlight supports addition of new user defined modules to the existing framework. These modules can use the services provided by Floodlight for their purposes. For example, Floodlight avails services such as *TopologyListener* or *StaticFlowPusher* for listening to topology change notifications or additions/removal of flows from the switches respectively. This set of well defined APIs makes it easy to implement new custom modules without disturbing the existing ones.

The figure below shows the architectural overview of the Floodlight framework.[32]

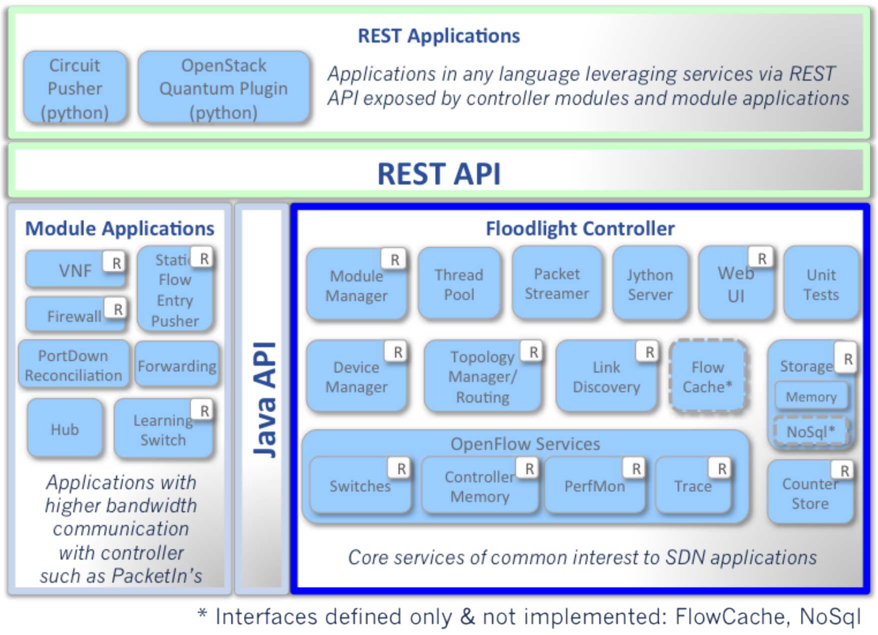

REST Applications
Circuit Pusher (python)
OpenStack Quantum Plugin (python)
Applications in any language leveraging services via REST API exposed by controller modules and module applications
REST API
Module Applications
VNF R
Static Flow Entry Pusher R
Firewall R
PortDown Reconciliation
Forwarding
Hub
Learning Switch R
Applications with higher bandwidth communication with controller such as PacketIn's
Java API
Floodlight Controller
Module Manager R
Thread Pool
Packet Streamer
Jython Server
Web UI R
Unit Tests
Device Manager R
Topology Manager/ Routing R
Link Discovery R
Flow Cache*
Storage R
Memory
NoSql*
OpenFlow Services
Switches R
Controller Memory R
PerfMon R
Trace R
Counter Store R
Core services of common interest to SDN applications
* Interfaces defined only & not implemented: FlowCache, NoSql

Figure 5.1: Floodlight controller architecture

As seen in the figure, the Floodlight controller also supports external applications through REST APIs. That means, a pub/sub routing module can as well be written as an external application which can make use of these REST APIs to get the topology information or add/remove flows. However, in this thesis, the pub/sub module has been developed as a controller module and not as an external application.

The framework also runs an HTTP server on the local machine, which provides user friendly interface to monitor the status of the network. It provides visualization of the entire topology as well as switch states, number of flows added, statistics and other relevant information.

The APIs that are used in the pub/sub module implementation include `linkDiscovery`, `StaticFlowPusher`, `FloodlightProvider` etc. Apart from these, the pub/sub module implements the interface *ITopologyListener* and *IOFMessageListener*. Without going into the details of these modules which are otherwise explained with the Floodlight manuals[33], the below mentions the relevant modules very briefly.

Floodlight Provider

Any custom module which intends to integrate with the controller framework and use any of the controller's functionality, uses this service class to listen to the incoming OpenFlow packets from the switches. This module also manages the connections to switches and converts OpenFlow messages into events that other modules can listen for. PacketIn is one such event that the custom modules can opt to listen.

The pub/sub module as no exception, uses this service to listen incoming packets and identify advertisements/subscription requests from the hosts attached to the OpenFlow switches.

LinkDiscovery

The controller maintains the link states of the network by periodically sending LLDP messages. The pub/sub module uses this link discovery service to get the latest link states and build the topology graph using it.

Forwarding

The Floodlight controller comes with its own forwarding module which basically floods the incoming packets to all the available ports. To test the custom routing module, for example the pub/sub module, this Forwarding module is disabled in the framework.

StaticFlowPusher

As the name suggests, this service is an integral part for any custom routing module as all the flow modifications (flow addition and deletion) are done via this class. Floodlight has its own objects and attributes for the representation of flow structures, i.e., match fields, action set, flow priority etc. These objects are instantiated with appropriate values and passed on to the StaticFlowPusher for addition or removal of flows.

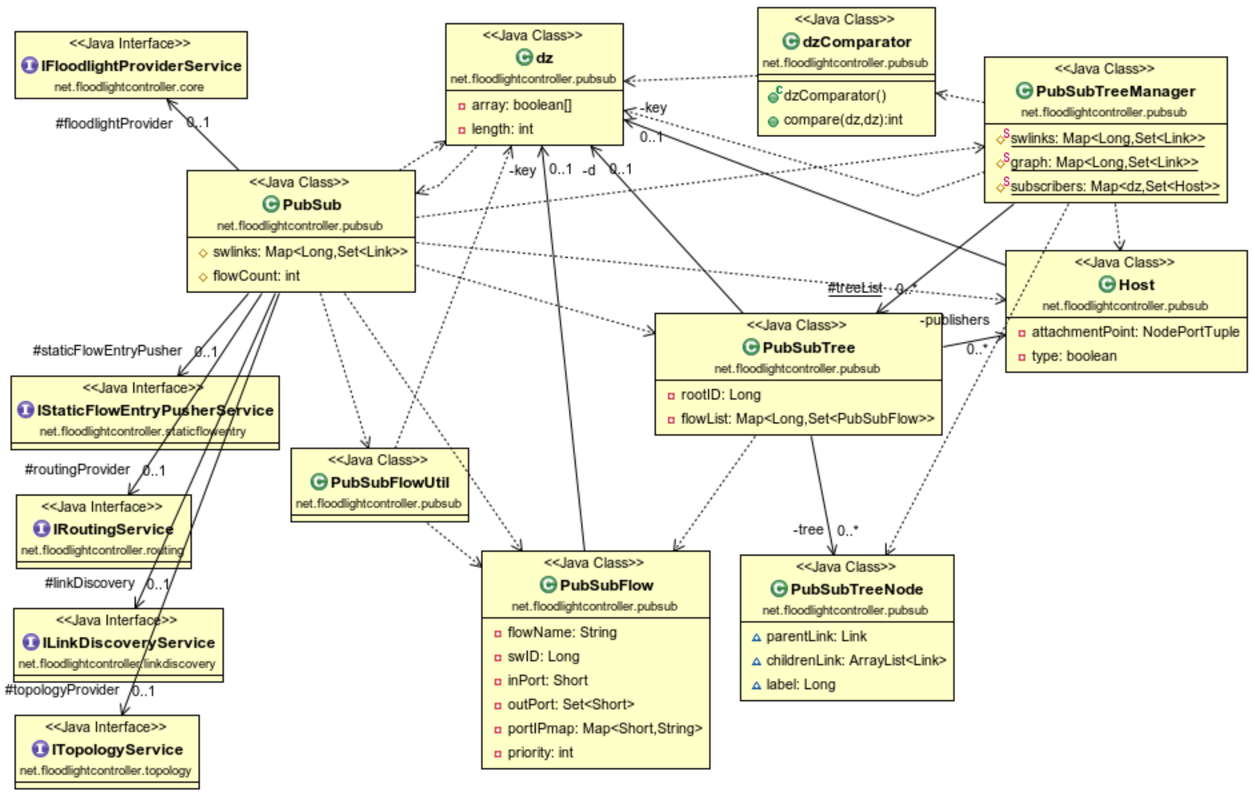
5.3.2 Pub/Sub Components

Different components of the publish/subscribe system such as publishers, subscribers, routing trees, flows etc., are stored as objects in the pub/sub module. The UML diagram below shows different classes used for the representation of these components and their relationships. The methods have not been showed here to keep it concise.

- **PubSub**
  The class named `pubsub` is the entry point of the implemented pub/sub system. It contains the implementations for listening to pub/sub messages and carry out necessary actions. As seen in the UML diagram, the module makes use of a number of services provided by the Floodlight controller.

  The switch links shown as an attribute, keeps the latest link information as received from the Floodlight APIs. The switch links data structure is implemented using a `hashmap`. It maps switch-ids with the `Link` objects attached to the switch.

  The `Link` object is a component within the Floodlight framework which simply represents a physical connection between two switches, i.e., it consists of four attributes



<<Java Interface>>
IFloodlightProviderService
net.floodlightcontroller.core
#floodlightProvider 0..1
<<Java Class>>
dz
net.floodlightcontroller.pubsub
array: boolean[]
length: int
<<Java Class>>
dzComparator
net.floodlightcontroller.pubsub
dzComparator()
compare(dz,dz):int
<<Java Class>>
PubSubTreeManager
net.floodlightcontroller.pubsub
swlinks: Map<Long,Set<Link>>
graph: Map<Long,Set<Link>>
subscribers: Map<dz,Set<Host>>
-key
0..1
-key 0..1 -d 0..1
<<Java Class>>
PubSub
net.floodlightcontroller.pubsub
swlinks: Map<Long,Set<Link>>
flowCount: int
#treeList 0..*
<<Java Class>>
Host
net.floodlightcontroller.pubsub
attachmentPoint: NodePortTuple
type: boolean
-publishers
0..*
#staticFlowEntryPusher 0..1
<<Java Class>>
PubSubTree
net.floodlightcontroller.pubsub
rootID: Long
flowList: Map<Long,Set<PubSubFlow>>
<<Java Interface>>
IStaticFlowEntryPusherService
net.floodlightcontroller.staticflowentry
#routingProvider 0..1
<<Java Class>>
PubSubFlowUtil
net.floodlightcontroller.pubsub
<<Java Interface>>
IRoutingService
net.floodlightcontroller.routing
-tree 0..*
#linkDiscovery 0..1
<<Java Class>>
PubSubFlow
net.floodlightcontroller.pubsub
flowName: String
swID: Long
inPort: Short
outPort: Set<Short>
portIPmap: Map<Short,String>
priority: int
<<Java Class>>
PubSubTreeNode
net.floodlightcontroller.pubsub
parentLink: Link
childrenLink: ArrayList<Link>
label: Long
<<Java Interface>>
ILinkDiscoveryService
net.floodlightcontroller.linkdiscovery
#topologyProvider 0..1
<<Java Interface>>
ITopologyService
net.floodlightcontroller.topology

Figure 5.2: Pub/sub UML diagram

as {*src-id, src-port, dst-id, dst-port*}. Therefore, each physical link is represented by two link objects representing both the directions of the physical link. The pub-sub module gets the switch link information, where every switch is mapped to the physical links twice, considering either direction.

- **Hosts**

  The pub/sub system does not differentiate between a publisher and subscriber and hence different type of objects have not been used for their representations. A generic object `Host` is used to represent any kind of participant in the system and a flag within that object signifies if the host is a publisher or a subscriber.

  Each host also contains an instance of an object *dz* with which it is associated as a subscriber or as a publisher. Also, they specify the attachment-point, a *SwitchID-Port* tuple, denoting the position at which they are connected in the network.

  The UML structure above shows the attributes 'type' which categorizes a Host as a publisher or subscriber and a `NodePortTuple` to identify its location in the network.

- **dz**
  Although a *dz*-expression is a simple string of ‘0’s and ‘1’s, it has been identified as a system component because of frequent operations such as comparisons and conversion to IP addresses. *dz* objects are also used as keys to identify subscribers and trees. The *dz* class implements `hashcode` and `equals` for the purpose of usage as `hashmap` keys.

  Another custom comparator class have been used for *dz*s as for ease in comparing containment relationships between the *dz*-expressions.

- **PubSubFlows**
  PubSubFlows signify the installed flows in a switch. It consists of flow attributes such as flow name, priority, associated *dz*, input port, output ports among others. A mapping between output ports to IP addresses is also maintained for the terminal switches.

  The class avails methods to convert it into JSON style flow-mod commands which is used by the Floodlight controller for addition or removal of flows. Other utility functions such as, checking covering relations between flows, getting the output ports common to different flows, etc., are provided in a different utility class namely `PubSubFlowUtil`.

- **PubSubTree and PubSubTreeNode**
  Pub/sub trees are routing trees discussed in the previous chapter. Each tree object associates with a *dz*-expression and also carries a list of publishers. For any tree, the tree’s *dz* contains the associated publisher’s *dz*. Apart from this, the tree maintains a list of flows installed within its *dz* space. The tree also provides methods such as route calculation from any host to any other host.

  The pub/sub trees are made up of nodes, each of which contains a parent link and a list of children links along with a label which signifies its level in the hierarchy of the tree. The label starts with a value of ‘0’ for the root node and increases with the hierarchy of the nodes. Labeling is necessary in order to get the routes from one host to any other using the Lowest Common Ancestor algorithm.

  The class named `PubSubTreeManager` provides regular utility methods to work on the topology graph. It also maintains the list of trees constructed each mapped to their respective *dz*s and a list of subscribers along with their *dz*s.

5.3.3 Module Loading

The Floodlight controller framework needs the custom modules to declare their dependencies on other modules as a bootstrapping process. This is required to ensure that the necessary modules are loaded before loading the custom module. Floodlight uses Java's `ServiceLoader` class to achieve this. Therefore, the new pub/sub module is listed in the available services and the necessary modules on which the pub/sub application depends are declared. `ILinkDiscoveryService`, `IStaticFlowEntryPusherService`, `IFloodlightProviderService` etc., are some of the modules that this pub/sub application depends on.

Apart from this, as the pub/sub application needs to get updates on topology changes, it adds itself as a listener of the implementation class of `ILinkDiscoveryService`. Similarly, the module also needs to get incoming messages from the publishers or subscribers. Floodlight uses the *Observer Pattern* to add listeners for such notifications.

The `init` method in each Floodlight module gets called after the module is loaded. This method initializes the necessary data structures and retrieves the instances of required implementation classes. The module gets the necessary implementation classes as :

```
floodlightProvider =
context.getServiceImpl(IFloodlightProviderService.class)
staticFlowEntryPusher =
context.getServiceImpl(IStaticFlowEntryPusherService.class)
```

where `context` is an instance of the object `FloodlighModuleContext` which is passed as an argument to the `init` method.

In the method `startup`, the module adds itself as a listener for the PacketIN events as shown below.

```
floodlightProvider.addOFMessageListener(OFType.PACKET_IN, this)
```

5.3.4 Run-time Behavior

After the module is loaded and registered for various notifications, the first notifications that the module receives are about topology updates. This is because as soon as Floodlight is switched on, it discovers the available switches and the links and thereafter forms an image of the topology. As it keeps adding the links, the topology view gets updated till the point when it has discovered all the available links.

The other run-time actions include listening to the packets and take actions such as addition of publisher or subscriber, based on the type of received message. These run-time actions are discussed as following.

Topology Discovery

As part of the OpenFlow protocol, the switches are discovered by the Floodlight controller when the controller is run. These are carried out by Floodlight's `LinkDiscovery` module which sends out LLDP messages and maintains the state of the links. Whenever there is a change in the link state, the topology is changed and hence, the topology needs to be recomputed. Each time the topology is changed, the method `topologyChanged` is invoked.

The pub/sub module lists itself as a listener of the topology change notifications and this way, the module always keeps an updated image of the entire network topology. The topology is formed from the switch link data as it gets from the `linkDiscovery` provider class. These switch link information are stored using a `hashmap` where switch IDs are used as keys and the associated links are put in as values :

```
Map<Long, Set<Link>> swlinks = linkDiscovery.getSwitchLinks()
```

The topology is simply a graph of the network, represented using an adjacency list, in which the node is represented by the *switch-id*. Each *switch-id* is mapped to a number of `Links` in which this *switch-id* is the source node. This topology information is then used to build routing trees with any given switch as its root node. Construction of routing trees from a graph uses a simple *Breadth First Search* approach.

As mentioned before, the `Link` objects represent each physical link twice, in either direction. Therefore the data structure `swlinks`, contains two links for every switch considering the switch as both source and destination.

The graph of the topology removes this redundancy in the switch link information. In the adjacency list representation every switch is associated only with the links in which this switch is the source switch. That means, the graph data structure is exactly of the same format as the switch-links information, but with redundancies removed. This is declared as :
`Map<Long, Set<Link>> graph` where, $\forall \text{link} \in \text{Set}<\text{Link}>, \text{link.src} = \text{Long}$

Listening to Packets

To identify (un)advertisement or (un)subscription messages, each incoming packet destined to the fixed IP address 225.0.0.37 needs to be examined, which is done in the method `receive`, as this method is triggered every time the controller gets a new packet from the switches.

For each such incoming message, the destination IP is checked. If the destination IP matches with the designated fixed IP address, it learns that the packet belongs to the pub/sub traffic. Apart from the destination address, other header fields such as layer-3 protocol is also checked. The pub/sub system uses UDP packets for data transmission, therefore the incoming packets are checked if they are UDP packets.

The controller provides well defined set of APIs for extracting the ip-address, protocol-type, ethernet address and other header information from an incoming packet. The APIs provided by Floodlight, extract the packet information layer after layer starting from the outermost ethernet header information. This is done as :

```
Ethernet eth = IFloodlightProviderService.bcStore.get(cntx,
IFloodlightProviderService.CONTEXT_PI_PAYLOAD)
IPv4 ipPkt = (IPv4)eth.getPayload()
UDP udpPkt = (UDP)ipPkt.getPayload()
Data dataPkt = (Data)udpPkt.getPayload()
byte[] arr = dataPkt.getData()
```

As shown, the innermost payload is extracted after stripping off all the outer header information. The validity of the packet can now be checked by examining the contents of the byte array. For a valid pub/sub message, the byte array should have 8 bytes with proper individual byte values. If the incoming packet is identified as a valid pub/sub packet, appropriate actions are taken, otherwise it lets the packet to be processed by other listening modules.

For a valid (un)subscription/(un)advertisement, this triggers the corresponding methods for further actions such as addition/removal of publishers or subscribers and so on. These method algorithms have been discussed in the previous chapter.

Tree Management

The tree management i.e., all the tree related operations are carried out by a utility class called `PubSubTreeManager`. This includes building of routing tree for any given node as a root, maintaining the updated graph, keeping a mapping between *dz*s and *tree*s etc.

As a new publisher is identified by the packet listener, the method to add publisher is invoked. This method implements the algorithm 4.1. If a new tree is constructed for an incoming advertisement, a mapping is kept between the associated *dz* and the newly constructed *tree* object. This mapping (shown in the UML structure above) is necessary to identify relevant trees for a given *dz*.

Similarly, when a new subscriber is discovered by the packet listener or a subscriber is removed, the tree manager class updates the mapping between *dz* to the list of subscribers.

The addition of a publisher, either a tree is created or an existing tree is updated with a list of publishers containing this publisher. In either case, this identifies the interested subscribers (stored in the `hashmap`) and builds routes for each of them from the publisher. The route calculation makes use of the parent links and the labels stored in each tree node.

Flow Management

Most of the flow management operations, such as addition/removal of flows are implemented within the PubSub class of the module. The algorithms have been discussed in the previous chapter.

Whenever a publisher/subscriber is added or removed, the flow addition or removal procedures are invoked. The common functions such as checking covering relations among the flows or identifying redundant flows for example, are provided in a utility class namely `PubSubFlowUtil`. Each of these covering relations of the flows simply take care of comparing the associated *dz*-expressions and the output ports set.

The addition of flows takes the given route information and the *dz* to add the flows. As each flow object provides a method to create JSON style flow structures, the string is obtained and an API from the Floodlight's `StaticFlowPusherService` is called with the given flow string and switch-id to add or remove flows. The formation of JSON string takes care of particular ordering of the actions in the action set, for terminal switches, as discussed in the previous chapter.

A sample flow structure in the form of a JSON string looks like :

```
{
    "switch":"00:00:00:00:00:00:00:01",
    "name":"flow-mod-1",
    "cookie":"0",
    "priority":"32768",
    "ingress-port":"1",
    "active":"true",
    "actions":"output=2"
}
```

This can be noted that the flow name is internal to Floodlight and is not conveyed to the switches. These names come handy to identify flows and delete the flows when removal of a publisher/subscriber is done, as the Floodlight's API for deletion makes use of the flow name for this purpose. The pub/sub module uses the pattern "PS_" for the flow names.

The pub/sub module uses a custom flow pusher API written within the Floodlight framework, namely `addFlowFromJSON` which takes a flow structure in the above mentioned JSON format, and adds the flow to a switch. This custom API was written as the existing

Floodlight's API for flow modification failed to set CIDR style network masks.

During removal of the flows the flow manager utility functions provide the fully covered or partially covered flows for processing, as suggested in the algorithms in previous chapter.

5.4 Conclusion

This chapter presented the overview of how the pub/sub logic discussed previously have been implemented on the Floodlight controller framework. The implementation has been done using simple data structures wherever possible and keeping the design to be simple and extensible for future requirements. There are many points of improvements as far as the code is concerned. These are pointed out in the final note in chapter 7.

The implementation is tested and evaluated in a simulation framework as well as on a real test-bed. The results are discussed in the next chapter.

Chapter 6

Evaluations

The design and implementation of the pub/sub system has been tested with various types of data sets and topologies. Primarily, false positive rates for different configurations of publishers and subscribers and delay characteristics in message transmission have been studied. Variation in flow table sizes is also noted for different dz-lengths.

The evaluations have been carried out both in a Mininet[34] environment as well as on a network with real hosts. The results presented herewith refer to the tests conducted on the real network. However, in either case, Open vSwitch software switches have been used. And hence, particularly the delay characteristics may not reflect the characteristics of real physical switches.

6.1 Testbed Setup

For the preliminary simulation and analysis with sample data sets, Mininet test environment was used. The mininet environment comes with a pre-configured virtual machine in which custom topologies of any number of hosts and any number of switches can be designed and specified to mininet by means of simple python scripts.

In addition, mininet also provides an inbuilt controller when any external controller has not been specified. Otherwise, it initializes the custom topology designed and attaches with the external controller running.

Mininet environment proved to be quite useful for the first set of tests and debugging of the controller application. However, with large number of events, subscribers and publishers, it is not practical to hold the simulations on the mininet environment. The initial tests in the mininet environment were conducted on an i686 machine running Arch Linux on a 2.00 GHz processor.

The topology used for further studies is shown as below. The topology uses a hierarchical arrangement of the switches connected with eight host systems. All the switches shown are installations of Open vSwitch software switches.

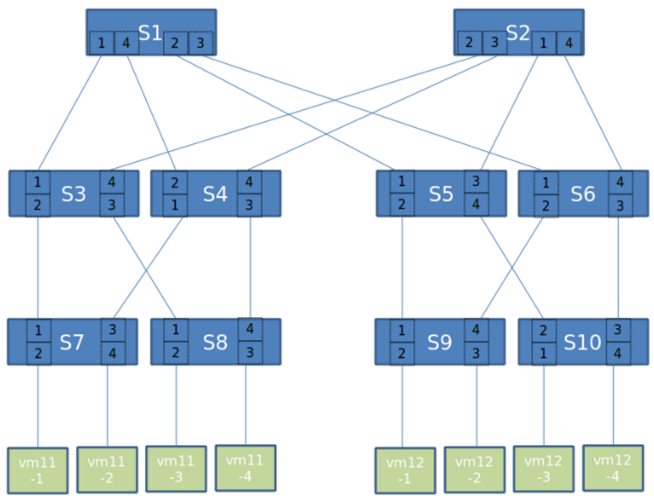


Figure 6.1: Test topology

The publisher and subscriber applications are run on the end hosts, (vm11-1 to vm12-4 in the above topology), and the Floodlight controller is run externally on a Linux machine with 2.40 GHz processor.

6.2 Data Sets and Experiments

Random data sets were generated using both uniform and Zipfian distribution. For either case, 3 dimensional data was generated and randomly distributed among the various hosts for advertisements/subscriptions. For Zipfian distribution, 5 hot-spots have been taken with an exponent of 0.8.

dz-expressions were generated keeping 250 as the maximum number of *dz*s for any event space. This factor is tunable in the algorithm presented in [22]. The factor 250 was chosen so as to keep the generated *dz*-expressions long enough for the experiments.

6.2.1 False Positives with *dz* Length

False positives are the messages received by a subscriber in absence of a subscription request for the message. Such messages are of course undesirable as they add up to the load on network.

False positive rate is measured in terms of %-age of total events received as:

$$\text{FPR} = \frac{\textit{Number of undesired events}}{\textit{Total number of events received}} \text{ X } 100$$

The longer the dz-expressions are, the lesser the false positives. This is obvious, as the length of the dz-expressions increase, the granularity of the event spaces increase and hence the false positives delivered to a subscriber decreases.

The following shows the variation of false positive rate with the length of dz-expressions for different set of subscribers for uniform distribution of data samples.

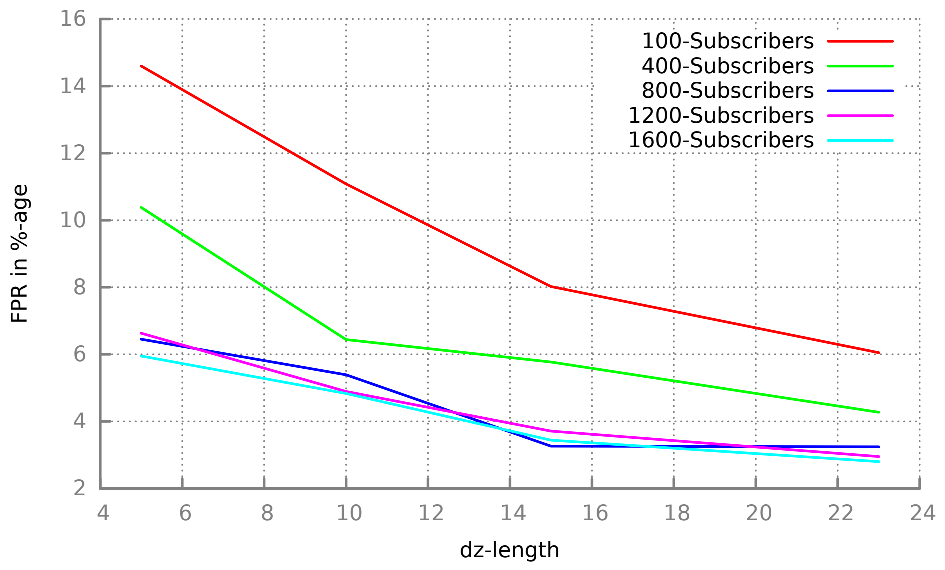


Figure 6.2: False Positive Rate with dz-length (uniform)

As seen in the figure, with increase in the length of the dz-expressions the false positives decrease to a range of 2.5-3%. The variation of false positives is also there with number of subscribers. This is justifiable as the subscriptions are distributed randomly among the different subscribers, large number of subscriptions actually represent the near-ideal case.

Moreover, this is highly dependent on the data sets used for the experiments. In the present case, 3 dimensional data have been used and along each dimension and the value ranges are fixed at <0-10000>. The publishers send out events in all possible event space. As we have only 23 bits for the representation of dz-expressions, events which differ in dz-strings only after the 23rd bit cannot be differentiated.

Thus, for a less number of subscriptions, an event space which was not subscribed might fit into the filtering criteria because of less dz-length and is counted as false positive. But for large number of subscriptions, the same event might have been included by some other subscription space and hence is no more a false positive.

i.e., for a subscription s_1 with a subscription space {<23-bits-prefix>001} an event with a dz of {<same-23-bits-prefix>000} is a false positive. But when a new subscription s_2 is added with a subscription space of {<same-23-bits-prefix>000} the previous event is no more a false positive even though no flow modifications or subscription addition is done on the controller. As for both the subscription spaces, the controller can only get the first 23 bits and cannot differentiate.

It may be noted that there are absolutely no false positives as far as dzs are concerned. i.e., a subscriber when subscribes to a dz of {00001} for example, it is guaranteed to receive no event which does not belong to this subscription space. But, when individual attributes along the dimensions are considered, false positives occur. And this clearly proves that the more is the length of the dz-expressions, better is the granularity and lesser will be the undesired messages.

The below shows the variation of false positives rate with dz-length when zipfian distribution is used.

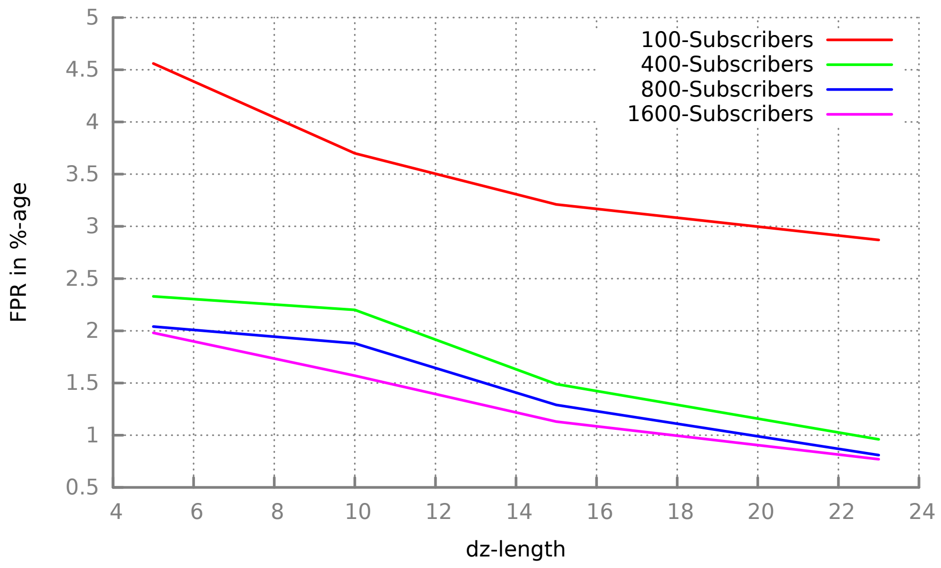


Figure 6.3: False Positive Rate with dz-length (zipf)

Similar to the uniform distribution the false positive rate for zipfian distribution decreases with increasing dz-length. The average false positive rate was found to hover around 1-2% considering large number of subscriptions as near-ideal case.

6.2.2 Delay Variations

Latency or delay in message transmission is measured by marking time-stamps within the messages, both at the publisher and the subscriber. The publisher marks the time stamp before sending the message over the socket and similarly, the subscriber puts the time of arrival upon reception of the messages.

The messages are then parsed later for analysis. The below shows the variation of delay with the number of subscriptions.

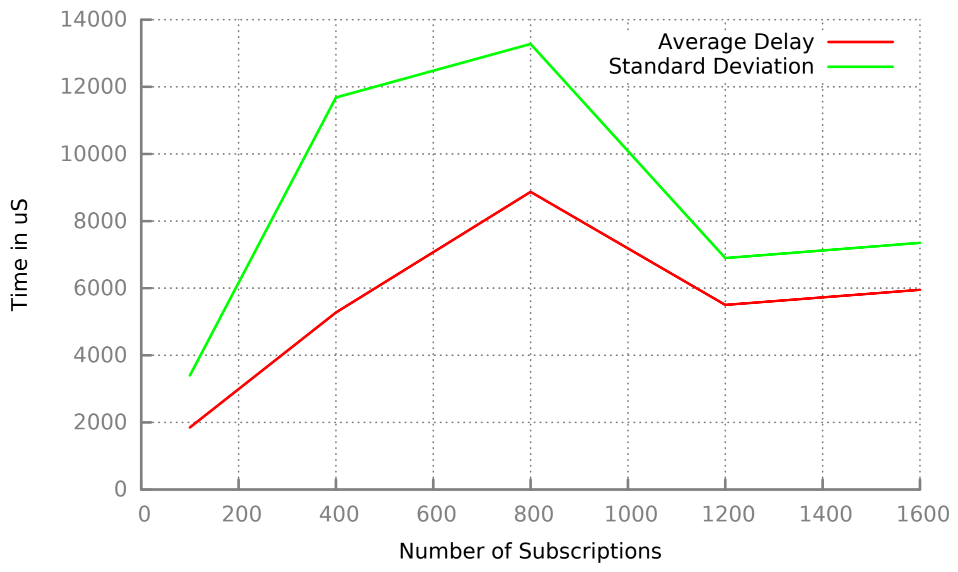


Figure 6.4: Delay with Number of Subscriptions

As the number of subscriptions increase, the delay in the message transmission is expected to grow due to increase in traffic volume. The overall pattern shows an increasing trend in the delay as we move from 100 subscriptions to 1600. However, there are anomalies in between and also the standard deviation is too high. This suggests that the delay in messages is actually spread over a large spectrum and possibly more tests are needed to converge the data points to a pattern. Overall, the delay was found to be bounded within a range of 5-6 milliseconds.

Similarly, the delay variations with dz-lengths have been measured for certain host systems which is presented in Fig 6.5.

As seen, with shorter dz-lengths the number of events received at a host is expected to be large. Which can cause delay both at the Open vSwitch software switches (due to frequent matching operations) as well as at the receiving operating system on the hosts (because of possible queuing in the kernel's network driver). Where the delay is almost

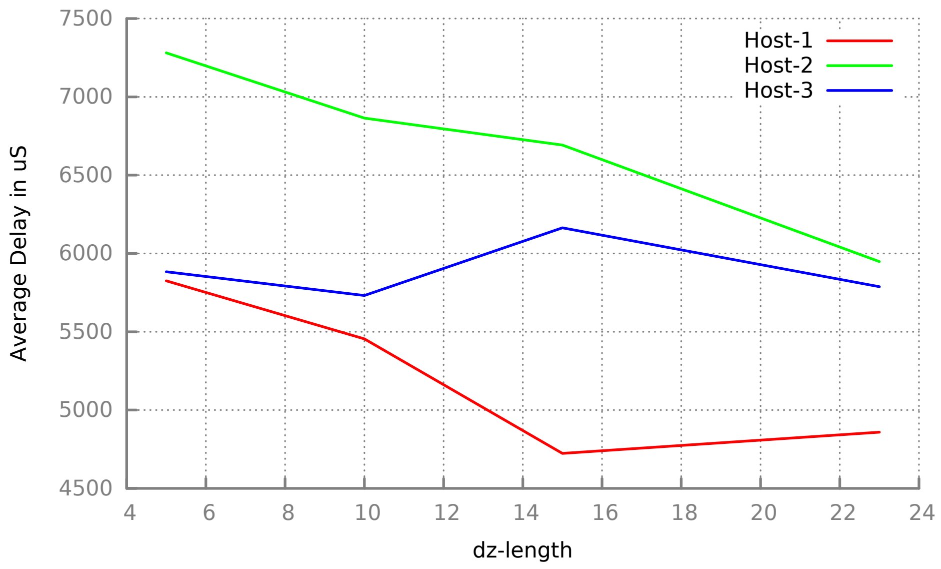


Figure 6.5: Delay with dz-length

unchanged with the change in dz-length, it suggests that the filtering of events might not have been affected by change in subscription's dz-length. For example an event with dz $\{0001\}$ matches both the subscriptions $\{00\}$ and $\{000\}$ and therefore, change in the length of subscription dzs from 2 to 3 would not affect the number of events received and causing the delay to be almost the same.

However, again the delay suffers from little anomalies and although a generic statement can be made that with shorter dz-strings the delay is more due to more traffic, a well defined relationship of the latency in transmission with the dz-length cannot be established.

Moreover, at the Open vSwitches, lesser dz-lengths create less number of flows and smaller flow tables. This is analyzed in the next section. Therefore, the delay produced by the Open vSwitches depends on both: the number of flows to be matched and the number of events to be matched. The exact relationships of either of these factors has not been tested here.

Some related studies[35][36] have claimed that the forwarding duration in Open vSwitches lie around 200 μS or more depending on the packet sizes. This has also been verified in this thesis by sending individual events and measuring the latency. It may be noted that in the evaluations, the massage sizes are of 60 bytes or more depending on the length of the dz-expression.

Considering this fact of Open vSwitch delay, the latency in the present study in the ranges of milliseconds seems more dependent on the host machines and other traffic present in the network.

In another experiment, the delay is studied with the number of received events. The associated figure 6.6 depicts the findings.

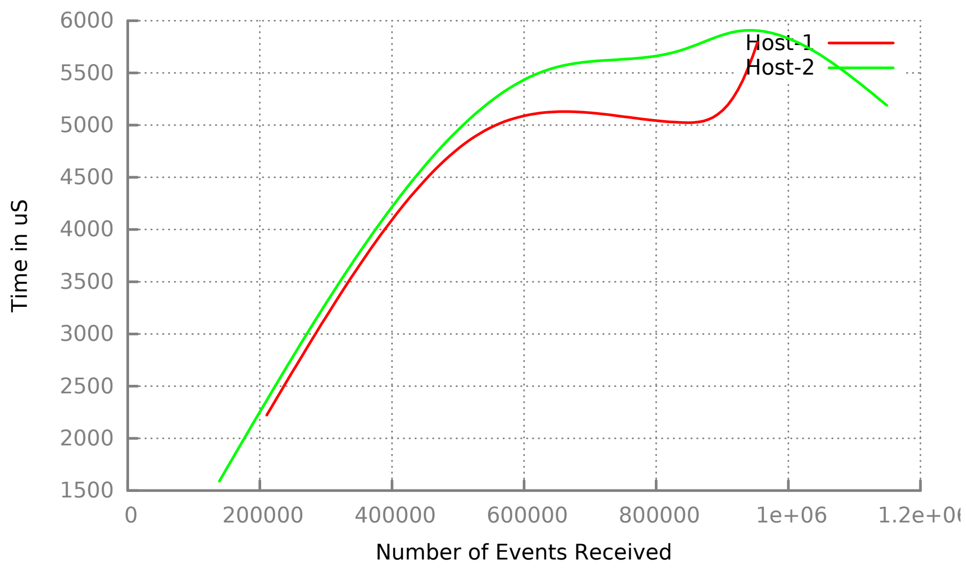


Figure 6.6: Delay with Volume of Events Received

As shown for 2 hosts, the delay quite expectedly increases with the number of messages received and more or less settles in the range of 5-6 milliseconds.

Summarizing, the studies related to the delay variations obviously showed that with large number of events the delay increases to a scale of 5-6 milliseconds in the present test-bed. The delay with number of events can be accounted for, considering the number of match operations and traffic volume received at the host machines. But, the flow matching overhead is not a prominent driving factor for the high latency as found in [35] and [36].

Interestingly, it can be noted that the end host machines running the publisher and subscriber applications are actually virtual machines running over another operating system. Such a system can add extra latency, as high as 100ms, claimed by [37]. This could also be a possible reason for the high delay seen in the packets.

It would be interesting to see how the delay comes up with real hosts instead of virtual machines. Apart from this, the delay characteristics can also be studied in a more realistic test environment with hardware switches and large number of publisher and subscriber systems as end hosts and each of them sending events in parallel.

6.2.3 Effect on Flow Table Size

As the number of subscribers increase, the average number of flows per switch naturally shows a tendency to grow. However, this is highly dependent on the subscription distributions among the subscribers and their physical location within the network. When many of the subscribers subscribe to a very few set of dz spaces, this would not affect the flow table sizes. On the other hand, if subscribers show interest in many unrelated dz spaces and are evenly distributed throughout the network, it would certainly increase the number of flows per switch.

Similarly, the flow table sizes also vary with the lengths of the dz-spaces. As mentioned earlier, the flows are added considering the finest of the dzs among publisher and subscribers. As the granularity of subscription spaces are increased, the dz lengths increase and hence, the subscribers subscribe to more number of finer event spaces. Naturally this would result in an increase in the number of flows.

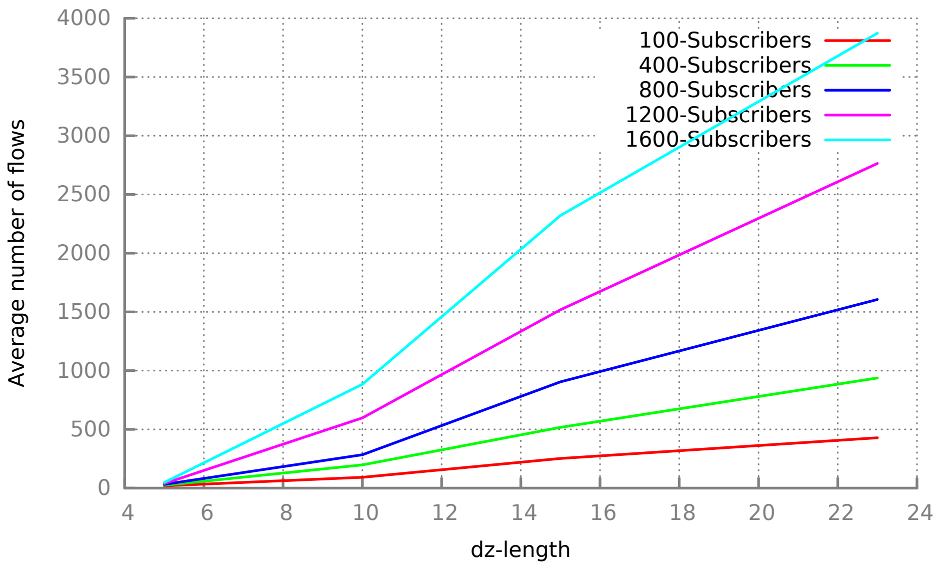


Figure 6.7: Flow Table size with dz-length

The above shows the variation of average number flows with dz-lengths and subscriptions. As mentioned, this represents only the characteristics of this particular test configuration and cannot be generalized. However, the increasing trend of flow table sizes is understood and is applicable to any subscription distribution pattern.

Chapter 7

Conclusion and Future Work

This thesis presented a basic realization of a pub/sub system using the power of *Software Defined Networking* and evaluated the approach. While the results are satisfactory and promises that with longer *dz*-expressions better filtering is possible, there are still many other aspects yet to be analyzed. Much space is still there for improvements as far as algorithmic design or the efficiency of the program is concerned. Certain points of possible enhancements and factors that can be studied in future are discussed below.

First of all, this thesis has implemented the in-network filtering approach discussed in [7]. The other approach namely *Channelization*, could be implemented as a new pub/sub module and compared with the present implementation. Also, the present approach uses IPv4 for filtering the data which constrains the *dz* length to a maximum of 23 bits. With the advent of OpenFlow controllers supporting IPv6, the implementation can be extended to work with IPv6.

Secondly, as far as algorithmic designs are concerned, there are possible improvements particularly related to the tree management. In stead of a static root in a given tree, possibly depending on the volume of events generated from the associated publishers, roots might be altered which can efficiently distribute the load on different links.

Also, with increase in lengths of the *dz*s the number of trees in the module increases, which can make this a memory expensive module in the controller framework. For each new publisher or subscriber, all the relevant trees need to be traversed to get efficient routes for content delivery. As the number of trees grow, this can clearly become a cumbersome process resulting in high run-time overhead. Further ways to reduce the number of routing trees needs to be analyzed. One possible way to minimize the number of routing trees could be to merge sub-trees and create a tree with a parent *dz* containing all the *dz*s of the removed trees. For example, trees with *dz*s {000, 001} can be merged to a single tree of *dz* {00}, without any effect on false positives (as the flows are always installed choosing the minimum dz).

In large topologies with thousands of switches, route calculations for huge number of publishers and subscribers can also exhibit performance related issues. Possibility of using a number of SDN controllers in a de-centralized manner may actually improve this issue. Such a system of distributed SDN controllers can be studied either to distribute the network into sections where each section is handled by its local controller or to just distribute the computationally expensive processes among themselves.

Apart from this, as found in evaluations, the flow table sizes grow rapidly with increase in the dz-length. Adoption of IPv6 addresses as dzs can therefore result in large number of flow entries in each of the switch's flow table. Methods to confine the flow table sizes to a certain allowable range should therefore be investigated.

With regards to the implemented program, as always, a piece of software evolves with time and the implementation is of course not final. There can be many improvements concerning the class designs and efficient usage of algorithms. Improvements such as making this module a multi-threaded application is a certain possibility, where different threads could be created whenever a new publisher or subscriber is discovered and expensive procedures can be run asynchronously reducing the run-time of controller.

Apart from this, the evaluations have only been done keeping the false positives, delay etc., in focus. However, in an SDN system with a centralized controller, the performance of the controller module is also of great importance. Certain tools such as CBench[38] are available for this purpose. It would be very useful to analyze the controller program and find possible improvements with regards to data structures and implementations which can affect the program's memory foot print and run-time overhead.

Also, since later releases of the OpenFlow protocol might remove the clause by which unmatched packets are being sent to the controller, the implementation needs to be modified to handle that situation. Because in such a case, a fixed IP cannot be used for discovering publishers or subscriber systems. One possible solution for such a scenario could be to enable REST APIs in the pub/sub module and the publishers or subscribers can notify their advertisements or subscription requests through `http` calls. Else, static flows could be added in each of the switches at the time of power on, to forward the packets with the fixed IP address as destination IP, to the controller.

Moreover, providing REST API services might sooner or later become a requirement, as with large number of publishers and subscribers, it would be very useful to monitor the available trees, associated publishers, flows and their statistics.

Another recently announced controller framework namely OpenDaylight[39], backed by a number of industries and The Linux Foundation[40] promises more active development in the codebase and early support for the later versions of OpenFlow protocol. This pub/sub application can certainly be ported to the OpenDaylight framework, without much effort, if needed as both these controller frameworks have similar set of interfaces for integration of new custom modules.

Lastly, the evaluations done in this thesis used synthetic data for the experiments. Behavior with some real world data such as stock market quotes could be studied. Also, this project has used Open vSwitches for the evaluations. Although they fully support all the functionality of OpenFlow, the true delay characteristics are not reflected. Study of latency and bandwidth usage on real switches could throw some more light on the scalability of the presented solution.

Bibliography

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

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 332–383, 2001.

[3] "Wikipedia – pub/sub pattern." `http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern`. Accessed: 13-May-2013.

[4] Wikipedia, "Software-defined networking – wikipedia." `http://en.wikipedia.org/w/index.php?title=Software-defined_networking&oldid=554014025`, 2013. [Online; accessed 10-May-2013].

[5] Openflow, "Openflow." `http://www.openflow.org/`, 2013. [Online; accessed 10-May-2013].

[6] SDNCentral, "SDN use cases." `http://www.sdncentral.com/sdn-use-cases/`, 2013. [Online; accessed 10-May-2013].

[7] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel, "The power of software-defined networking: line-rate content-based routing using openflow," in *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, p. 3, ACM, 2012.

[8] A. Riabov, Z. Liu, J. L. Wolf, P. S. Yu, and L. Zhang, "Clustering algorithms for content-based publication-subscription systems," in *Proceedings of the 22nd International Conference on Distributed Computing Systems.*, pp. 133–142, 2002.

[9] Wikipedia, "K-means clustering – wikipedia." `http://en.wikipedia.org/w/index.php?title=K-means_clustering&oldid=554472558`, 2013. [Online; accessed 12-May-2013].

[10] 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.

[11] Wikipedia, "Spectral clustering – wikipedia." `http://en.wikipedia.org/w/index.php?title=Spectral_clustering&oldid=554427629`, 2013. [Online; accessed 10-May-2013].

[12] R. Zhang and Y. C. Hu, "Hyper: A hybrid approach to efficient content-based publish/-subscribe," in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pp. 427–436, IEEE, 2005.

[13] P. Triantafillou and A. Economides, "Subscription summaries for scalability and efficiency in publish/subscribe systems," in *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, pp. 619–624, IEEE, 2002.

[14] Wikipedia, "Bloom filter – wikipedia." `http://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=554063487`, 2013. [Online; accessed 10-May-2013].

[15] Z. Jerzak and C. Fetzer, "Prefix forwarding for publish/subscribe," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pp. 238–249, ACM, 2007.

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

[17] Wikipedia, "Core-based trees – wikipedia." `http://en.wikipedia.org/w/index.php?title=Core-based_trees&oldid=441834800`, 2011. [Online; accessed 12-May-2013].

[18] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "Lipsin: line speed publish/subscribe inter-networking," in *ACM SIGCOMM Computer Communication Review*, vol. 39, pp. 195–206, ACM, 2009.

[19] J. Moscola, J. W. Lockwood, and Y. H. Cho, "Reconfigurable content-based router using hardware-accelerated language parser," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 2, p. 28, 2008.

[20] Wikipedia, "XML – wikipedia." `http://en.wikipedia.org/w/index.php?title=XML&oldid=554490422`, 2013. [Online; accessed 12-May-2013].

[21] 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.

[22] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, and K. Rothermel, "Meeting subscriber-defined qos constraints in publish/subscribe systems," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 17, pp. 2140–2153, 2011.

[23] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. Rothermel, "Providing probabilistic latency bounds for dynamic publish/subscribe systems," in *Kommunikation in Verteilten Systemen (KiVS)*, pp. 155–166, Springer, 2009.

[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[25] O. Consortium *et al.*, "Openflow switch specification v1. 0."

[26] "Beacon controller." `https://openflow.stanford.edu/display/Beacon/Home`, 2011. [Online; accessed 12-May-2013].

[27] "NOX controller." `http://www.noxrepo.org/nox/about-nox/`. Accessed: 10-May-2013.

[28] "POX controller." `http://www.noxrepo.org/pox/about-pox/`. Accessed: 10-May-2013.

[29] "Ryu controller." `http://osrg.github.io/ryu/`. Accessed: 10-May-2013.

[30] "OSPF." `http://en.wikipedia.org/wiki/Open_Shortest_Path_First`. Accessed: 13-May-2013.

[31] Floodlight, "Floodlight SDN controller." `http://www.projectfloodlight.org/floodlight/`, 2013. [Online; accessed 12-May-2013].

[32] "Floodlight controller architecture." `http://docs.projectfloodlight.org/display/floodlightcontroller/Architecture`. Accessed: 10-May-2013.

[33] "Floodlight manual." `http://docs.projectfloodlight.org/display/floodlightcontroller/Floodlight+Documentation`. Accessed: 10-May-2013.

[34] Mininet, "Mininet simulation environment." `http://mininet.org/`, 2013. [Online; accessed 12-May-2013].

[35] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an openflow architecture," in *Proceedings of the 23rd International Teletraffic Congress*, pp. 1–7, ITCP, 2011.

[36] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation," in *Passive and Active Measurement*, pp. 85–95, Springer, 2012.

[37] J. Whiteaker, F. Schneider, and R. Teixeira, "Explaining packet delays under virtualization," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 38–44, 2011.

[38] "CBench." `http://www.openflow.org/wk/index.php/Oflops`. Accessed: 13-May-2013.

[39] "Opendaylight project." `http://www.opendaylight.org/`. Accessed: 13-May-2013.

[40] "The Linux Foundation." `http://www.linuxfoundation.org/`. Accessed: 13-May-2013.