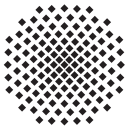


Universität Stuttgart


Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 3498

# Distributed Control Algorithms for Adapting Publish/Subscribe in Software Defined Networks

Sukanya Bhowmik

| | |
|---|---|
| **Course of Study:** | Information Technology/InfoTECH |
| **Examiner:** | Prof. Dr. Kurt Rothermel |
| **Supervisor:** | Dr. Boris Koldehofe |
| **Commenced:** | 2013-5-15 |
| **Completed:** | 2013-11-14 |
| **CR-Classification:** | C.2.1,C.2.4 |

# Abstract

Content-based routing has emerged as a popular paradigm in publish/subscribe systems for interactions between its system components (publishers and subscribers). Content-based routing of published information is extremely bandwidth efficient as a publication is forwarded only to subscribers which have expressed their interest in this published content. Rules or filters are applied on the content at one or more routers/brokers to determine its path through the network.

Most of the state of the art solutions consist of a distributed set of brokers scaling wide-area networks. However, in each of these solutions, filtering of events takes place at the application layer at dedicated brokers. This expensive filtering phase renders the advantages of content-based pub/sub with regards to bandwidth efficiency less significant as it results in higher end-to-end latency and lower throughput rates. To overcome this problem, software-defined networking may be used to build publish/subscribe systems where filtering of events can happen directly in the Ternary Content-Addressable Memory (TCAM) of network routers. Initial work has shown that it is possible to map effectively a content-routing topology to network routers controlled by a single broker, resulting in line-rate forwarding of data packets. However, a single broker limits the performance of the system with regards to scalability and is not feasible in large networks consisting of numerous network elements.

To incorporate the best of both worlds, this thesis proposes distributed control algorithms using software-defined networking that allow to build a publish/subscribe system spanning over multiple sub-networks of controllers where the controllers divide the network spatially into disjoint partitions. With respect to such an architecture, this thesis discusses the manner in which connectivity is established between sub-networks along with the costs incurred in the process. A detailed analysis of the average controller overhead and total control traffic generated in the proposed system is presented which is further supported by simulation results. It also includes an analysis on the effect of distributing control on certain performance metrics such as false positive rate of published events.

# Acknowledgements

It gives me great pleasure in acknowledging the contributions of all those without whom this thesis would not be possible. First and foremost, I wish to thank Prof. Dr. Kurt Rothermel for giving me an opportunity to do my thesis in the Department of Distributed Systems and in a topic that truly interests me.

I owe my deepest gratitude to my supervisor, Dr. Boris Koldehofe for his patience, constant involvement and immense support throughout the duration of this project. This thesis would never have been possible without his enthusiasm and invaluable guidance. I consider myself extremely fortunate to have had the opportunity to be mentored by him.

I express my sincerest gratitude to Dr. Frank Dürr, Dipl.-Ing. Ben Carabelli and Dipl.-Inf. Florian Berg for helping me with various SDN-related issues. I wish to thank Dr. Muhammad Adnan Tariq for the insightful discussions on the subject of publish/subscribe. I am also grateful to Dipl.-Inf. Martin Brodbeck for providing me with the necessary equipment for this project.

A special thanks to M.Sc. Gagan Behari Mishra for always attending to all my doubts related to his works which have been an integral part of my research. I really appreciate his help and value his advices. Many thanks to my friends Darsana Das, Sreedhar Mahadevan, Arturo Francato and Naresh Nayak for motivating and supporting me and for creating an enjoyable work environment throughout the duration of this thesis.

Finally, I wish to thank my family for always being there for me. I consider myself very fortunate to have been blessed with their immense love and support.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

The growing amount of information exchanged every day over wide area networks has made concepts such as event-notification system [1], also known as *publish/subscribe* system [2], extremely significant in today's world. Applications in distributed systems are characterized by properties such as asynchrony and referential decoupling and this is where an event-notification system plays a major role. Content delivery in a distributed system is often associated with the propagation of events and event propagation has been generally associated with the well known communication paradigm of publish/subscribe.

A publish/subscribe system, also commonly known as a *pub/sub* system, finds its application in instant news delivery, stock quote dissemination, RSS feeds, electronic auctions etc. The main idea behind all these systems is to gather information from a set of data sources and deliver them to interested users. It is easily understood that the aforementioned systems are extremely dynamic in nature and deal with multiple concurrent users. These applications have highly demanding requirements and this is where some of the well known classical abstractions of distributed applications display significant limitations. Most of the well established distributed system paradigms are based on end-to-end, synchronous communication. Considering the wide range of dynamically changing set of senders and receivers in the aforementioned applications, it is infeasible for each of the components to maintain information about every other component in the network in order to establish point-to-point communication. As a result, there was a requirement for dedicated middle-ware that would enable many-to-many communication while maintaining loose-coupling among the components comprising the network. This led to the advent of publish/subscribe system, which is the subject of study of this thesis. The strength of the publish/subscribe paradigm lies in the fact that it provides an efficient platform for many-to-many, asynchronous communication between publishers and subscribers that remain decoupled in time and space.

A publish/subscribe system comprises of mainly publishers and subscribers where information flows from publishers to subscribers in such a manner that both parties remain completely oblivious of each other's existence. This anonymity is maintained by introducing a logical intermediary, often known as, *Notification Service* [1]. Both publishers and subscribers communicate with the Notification Service which indirectly establishes communication between them. Quite evidently, one can deduce that the scalability of the system depends on the efficiency of the logical intermediary to handle multiple concurrent users and it is the logical

intermediary that defines the limits of the system. This has led to the idea of having multiple logical intermediaries, spanning a wide area network. These logical intermediaries need to communicate among themselves, exchange state information and work towards the common goal of delivering information to interested subscribers throughout the entire network.

A pub/sub system can be broadly classified into topic-based and content-based systems. The main difference between the two variants is the degree of expressiveness of the subscribers where content-based pub/sub has a significant advantage. Content-based pub/sub provides content-based routing which uses the information disseminated in the system to take routing decisions. So, effectively, the path between the publishers and the subscribers are set based on filtering techniques. These content filters determine various parameters of the system, such as, bandwidth efficiency and accuracy. For any information published, an early identification of the set of subscribers and delivery to only interested users is of primary importance. Unnecessary information flow in the network affects bandwidth efficiency and delivery to uninterested subscribers affects accuracy.

There has been a lot of research work dedicated to middle-ware implementations of content-based pub/sub systems in the past few years. [1, 3, 4, 5, 6] are a few examples, some of which are discussed further in the following chapters. However, as these are implemented in the application layer, they cannot achieve the same performance in terms of latency and bandwidth efficiency as compared to an implementation in the network layer. The application layer takes the overlay topology into consideration which almost always differs greatly from the underlying topology. The same physical link is often mapped to multiple logical links resulting in the same content being sent over a physical link multiple times, thus affecting bandwidth efficiency. Also, the matching of events with subscriptions in the content-based pub/sub implemented in the application layer is highly computation intensive and results in increased end-to-end delay and low throughput rates. As a result, the advantages of the content-based pub/sub appear less significant as compared to paradigms such as LIPSIN[7] which provides a topic-based approach for line-rate forwarding using IP multicast. Even though LIPSIN benefits from line-rate forwarding, it lacks flexibility in terms of expressiveness due to its topic-based approach and has its own limitations. Thus, an application implemented on the network layer, based on content-based routing and supporting line-rate message forwarding seems to be an ideal approach to an efficient pub/sub system. However, conducting experiments with new networking protocols in a real world setting is quite unrealistic. This is mainly due to the already installed huge base of equipment and networking protocols that are used extensively. This has created a barrier to the entry of new ideas and reluctance for experimentation on the network layer. However, this scenario has changed significantly with the advent of *Software-Defined Networking* (SDN)[8].

Software-Defined Networking is an important step taken towards programmable and active networking evolution. It provides ways to abstract the lower level functionalities and presents them as network services. Traditional switches are responsible for both route calculations (control logic) as well as forwarding of data. The SDN technology separates the control logic from

switches and hosts it on servers, also known as *controllers*. A controller has an integrated view of the whole system and can access the switches of the network through special interfaces. So, with *software* defining the network, a great deal of flexibility can be achieved. Naturally, a lot of work is currently going on related to the design and implementation of various distributed applications using software-defined networking and the content-based publish/subscribe system is no exception to this. SDN allows to execute matching of events in a content-based pub/sub directly on the Ternary Content-Addressable Memory (TCAM) of switches, thus enabling line-rate forwarding of events resulting in much better performance with regards to throughput and end-to-end latency as compared to an application layer pub/sub implementation. This thesis focuses on such an approach which takes advantage of the features of SDN to implement content-based routing.

Koldehofe et al. in [9] propose ways to build an efficient pub/sub system by utilizing the power of software-defined networking. Furthermore, Mishra in [10] presents an implementation proposed in [9] based on in-network content-based routing. However, [10] defines a system with a single controller that is responsible for handling concurrent requests from both publishers and subscribers, maintaining state information and establishing routes for efficient information diffusion from publishers to subscribers. The use of such a single controller instance may have its limitations with regards to scalability leading to enhanced processing latency. For example, the growing number of events generated in the network may saturate the CPU of a single controller. Also, the growing number of network elements may exhaust system memory. As a result, this thesis focuses on dividing the load of the pub/sub system among multiple controller instances. Increased scalability is achieved by partitioning the network among the controllers in such a way that each controller keeps only a disjoint subset of network elements up-to-date in memory. Also, since each controller is aware of only a subset of the network elements, it is directly responsible for establishing routes in its own sub-network. Considering these advantages, this thesis presents a design and implementation of a pub/sub system using *distributed controllers* in SDN for improved performance in terms of scalability. It goes on to analyze the benefits/drawbacks of distributed control as compared to centralized control with respect to average controller overhead, total generated control traffic, false positive rate etc. The analysis is further supported with simulation results.

## 1.1 Thesis Organization

The remaining part of the thesis is organized as follows :

Chapter 2 provides a background that is necessary to understand the notion of publish/subscribe systems. It introduces the components and explains the behavior of a typical pub/sub system. It further gives a brief overview of software-defined networking and discusses the role of SDN in a pub/sub system.

Chapter 3 provides a formal specification of the problem statement. It also presents some of the existing research work relevant to the subject of study of this thesis. It particularly gives

a brief overview of current content-based publish/subscribe systems. This includes a brief survey of SIENA, Gryphon, JEDI etc.

Chapter 4 gives a detailed description of a pub/sub system implemented using a single controller in SDN[10]. It describes the various algorithms used to achieve in-network content-based routing. This includes algorithms for handling advertisements, subscriptions, un-advertisements and un-subscriptions. This thesis is an extended work of [10] and utilizes all the aforementioned algorithms.

In chapter 5, the algorithms that have been realized in this thesis have been described in details. This chapter discusses the various message types introduced, state management at each of the controllers, the interaction between the distributed controllers and basically every concept introduced to implement pub/sub with distributed controllers.

Chapter 6 provides an analysis of the design and prototype implementation of the built system. It introduces the test environment used in this thesis. Also, it discusses the various experiments that were conducted and the evaluation results that were obtained from them.

Finally, Chapter 7 concludes this thesis with a brief summary of the work done. It also proposes an outline for possible future works.

# Chapter 2

# Background

The main objective of this chapter is to provide a good understanding of the key concepts that form the basis of this thesis. In the following sections we discuss the basic principles of the publish/subscribe paradigm and then focus on the technology provided by Software-Defined Networking.

## 2.1 Principles of Publish/Subscribe Paradigm

The main purpose of a publish/subscribe system is the propagation of information between participants of the system in an asynchronous and decoupled manner. A few key elements define and form the building blocks of a publish/subscribe system and are explained in details in this section. These include the participants, notification service implementation, supported client interactions, types of notifications and subscription model of a basic pub/sub system.

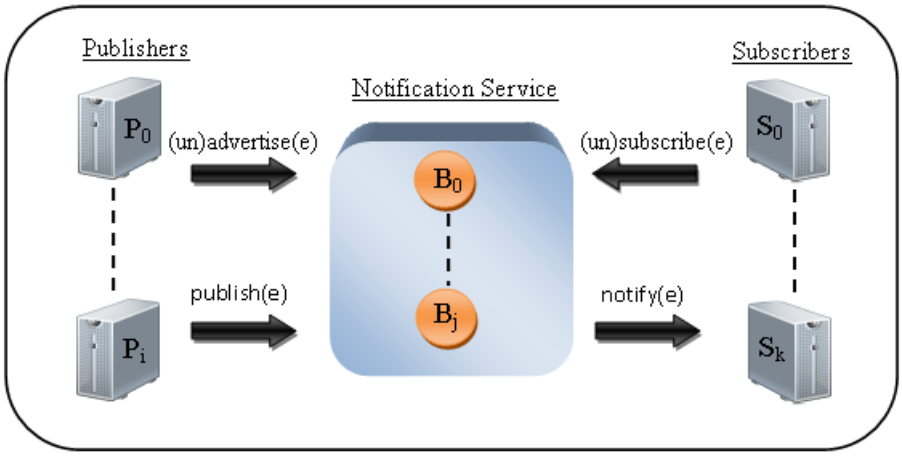



Figure 2.1: A publish/subscribe system

**Participants** A pub/sub system mainly comprises of two types of participants, namely, *publisher* and *subscriber*. A publisher sends out notifications and plays the role of a data source. A subscriber, on the other hand, behaves like a sink and consumes information produced by

a publisher. A subscriber expresses its interest in a particular event by sending a subscription request and listens for events that match this request. Each subscriber can produce multiple subscription requests and any published information that matches its subscription set is delivered to it. A publisher may send out advertisements, prior to sending notifications, which determine the type of information it intends to publish. This is referred to as the *advertisement model* in the following sections.

**Notification Service** As previously mentioned, a publisher and a subscriber do not communicate with each other directly and remain decoupled in space. This introduces an entity called the notification service, also commonly known as *broker/s*. Figure 2.1 depicts such a publish/subscribe system with $i$ publishers $(P_0, ..., P_i)$ and $k$ subscribers $(S_0, ..., S_k)$ which communicate with the notification service. The notification service itself can consist of a single broker or a set of $j$ brokers $(B_0, ..., B_j)$ as shown in the figure. In the case where a single broker exists, all publishers and all subscribers are connected to the same broker whereas in the latter case they may be connected to any broker in the complete set.

**Supported Operations and Notification Types** The notification service defines the client interactions supported by the system. The publishers and subscribers communicate with the notification service using operations provided by it. As depicted in Fig. 2.1, a publisher is mainly associated with three operations. One of these is the *publish* operation which allows information to be published by the publisher. Considering an advertisement model, a publisher can *advertise*, i.e., declare the nature of information it intends to publish and can also perform a reverse operation depicted in the figure as *un-advertise*. The notification service supports the *notify* operation which allows it to deliver published information to interested subscribers. A subscriber is associated with two more operations. These are *subscribe* which defines the interest of the subscriber and its reverse operation *un-subscribe*. The supported operations directly lead to the various types of notification generated in the system. The publish operation is associated with *publications* or more generally *events*. When these events are delivered to the subscribers, they are generally termed *notifications*. In general terms, a notification can be described as attribute-value pairs, $e = < attr_0, value_0; attr_1, value_1; attr_2, value_2; ..... >$. A subscribe operation deals with *subscription* and an un-subscribe deals with *un-subscription*. A subscription is again a pair, $s = <f, sub>$ where $f$ represents a subscription filter and *sub* identifies the subscriber. However, the filter expression cannot be expressed in general terms as it differs based on subscription models and is discussed further in the next section. In an advertisement model, advertise operation is associated with *advertisement* and un-advertise with *un-advertisement*. The structure of an advertisement is very similar to that of a subscription and also consists of a pair, $adv = <a, pub>$ where $a$ denotes an advertisement filter and *pub* identifies the publisher. So a notification $e$ matches a subscription $s$ if $e$ satisfies filter $f$ in $s$, i.e. $e \sqsubset f$. Also, a publication $e$ matches an advertisement $adv$ if $e$ satisfies filter $a$ in $adv$, i.e. $e \sqsubset a$.

Previously, we have defined events and subscriptions/advertisements in very general terms. However, this thesis is based on a specific content space representation model which should

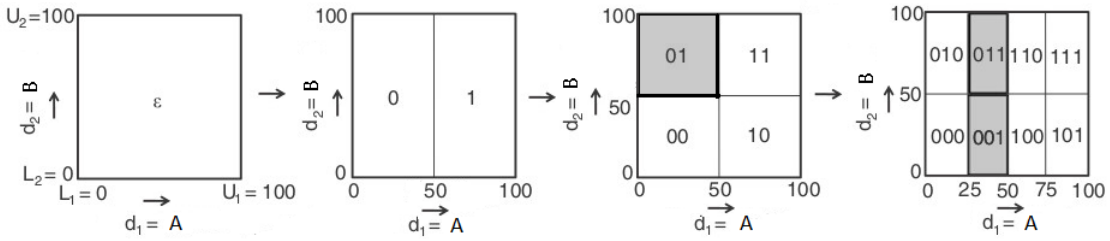



Figure 2.2: Spatial Indexing

be discussed in further details for future reference. This model has been proposed by Tariq et al. in [5] and is based on spatial indexing. In this model, the content space is represented as a d-dimensional space where each of the dimensions refers to an attribute. Any subscription/advertisement(*s/adv*) is represented by a regular sub-space in the event space and any event $e$ is represented by a point in the content space. Recursive binary decomposition of the event space generates these regular sub-spaces which are represented with binary strings called *dz-expressions*. The dz-expressions are created by the binary partitioning of the content space along each dimension. With more and more partitioning, the granularity of sub-spaces increases as does the length of the dz-expressions. So, a smaller sub-space is represented by a longer dz-expression whereas a larger subspace is represented by a smaller dz-expression. This can be further explained with an example taken from [5] which considers a 2-dimensional content space represented by $\varepsilon$. Considering these two dimensions/attributes to be $A$ and $B$, Figure 2.2 depicts $A$ along X axis and $B$ along Y axis. So, in this 2-dimensional plane, a subscription/advertisement is a rectangle. Let there be two subscriptions $s_0 = \{A = [0, 50], B = [50, 100]\}$ and $s_1 = \{A = [25, 50], B = [0, 100]\}$. So, according to Figure 2.2, $s_0$ is represented by the sub-space $\{01\}$ and $s_1$ by the sub-spaces $\{001, 011\}$. The manner in which this thesis utilizes the spatial indexing scheme will be discussed further in the following chapters.

A publish/subscribe system is defined by a number of performance factors. Bandwidth efficiency is always an important performance measure in a distributed system and publish/-subscribe is no exception to this. As a result, most of the pub/sub systems pay a great deal of attention to minimizing the bandwidth consumption making content-based pub/sub a popular paradigm. Maximum tolerable latency also contributes to defining a pub/sub system. This is basically the amount of time needed by a published event to reach the subscriber. [5] focuses on QoS in a broker-less network where bandwidth efficiency and allowed delay are defined by subscribers. Besides these, it is important to introduce two more performance metrics, namely, false positives and false negatives. False positives are events which are delivered to a subscriber which was not interested in receiving them. Whereas, false negatives are events that were not delivered to a subscriber which had expressed its interest in receiving them. Quite naturally, the aim of any pub/sub system is to keep both of these to a minimum.

### 2.1.1 Types of Pub/Sub based on Subscription Models

The expressiveness of subscriptions broadly classifies pub/sub systems into two categories. The degree of expressiveness must be chosen very carefully as various factors depend on it. On one hand, a very low degree of expressiveness can cause unnecessary information flow in the network affecting bandwidth efficiency and resulting in delivery to uninterested subscribers. In certain cases filtering of notifications may need to be done at the subscriber end. On the other hand, realization of a scalable system with a highly expressive subscription model demands very complex implementations. Thus, scalability and expressiveness are two major challenges intrinsic to the pub/sub system which must be dealt with extreme care [11]. In this section two notification selection schemes are described, namely, topic-based and content-based.

**Topic-based Publish/Subscribe** In a topic-based system, publishers publish information to named *logical channels*, more commonly known as *topics*. The subscribers subscribe to these topics and all subscribers interested in the same topic receive the same messages published under that topic. So, in the previously established notations, a filter $f$ in a subscription $s$ simply specifies a particular topic/channel. There is a lot of work done in literature based on topic-based systems which include LIPSIN [7],CORBA [12], iBus [13] and SCRIBE [4]. A topic-based system can be very efficiently implemented by simply mapping the channels to multicast groups. However, the main disadvantage of this mechanism is that it lacks flexibility with respect to expressiveness of a subscriber. A subscriber may not be interested in every information published under a topic and may be interested in only a subset. To improve this situation, various solutions have been proposed among which features the subject-based approach [14]. This approach organizes topics in a hierarchical manner such that any topic $Y$ can be declared as a sub-topic/sub-channel of another topic $X$. Even though this attempts to better the expressiveness of the subscribers, it cannot avoid the very idea of topics and the grouping of subscribers according to them.

**Content-based Publish/Subscribe** A content-based pub/sub allows subscribers much more flexibility with regards to expressiveness. Subscribers can define very fine-grained filters such that only the desired content is delivered to them. In this mechanism, filtering is done based on the content of the notification and the subscribers can specify their interests by imposing constraints on the content. So, now, we have a filter $f$ which does not simply refer to a topic but instead is a query comprising of a set of constraints over values of the attributes. The filter $f$ has the following structure , $f = <attr_0, (constraint_0, constraint_1, ...); attr_1, (constraint_0, constraint_1, ...);...>$. The set of constraints that are supported depends on the subscription language used. [15] provides a comprehensive specification of subscription models. The more powerful a subscription language, the more expressive is the subscription model. However, as discussed previously, there exists a trade-off between expressiveness and scalability. In a typical content-based system, the set of subscribers to which a notification has to be delivered is determined dynamically on arrival of the notification. Needless to say, this mechanism involves a great deal of computational overhead and as a result there has

been a lot of research on efficient and scalable filtering techniques for content-based pub/sub [16, 6, 5, 17] which include SIENA [1], Gryphon [18] and SPINE [3].

**Clustering and Filtering in Content-based Pub/Sub** Content-based pub/sub focuses on avoiding unnecessary forwarding of events through paths in the network that lead to uninterested subscribers. In this context, the method of channelization displays significant performance gains. Channelization is based on clustering of subscriptions/advertisements, mapping these clusters to a set of channels and then disseminating events within these channels. This results in reduction of unnecessary event forwarding. There exist various approaches to clustering and channel creation. Many of these deal with the absolute structural similarity between subscriptions, i.e., the intersection area of two subscriptions [19, 20, 21]. The disadvantages of this technique are that it limits the expressiveness of the content-based model to predefined numeric attributes and also does not consider the similarities between subscriptions as per event traffic. So, there are approaches which take into account the event traffic in the recent past for channelization [22, 23, 24]. Clearly, in this approach along with control messages for subscription/advertisements, the routing overlay optimization algorithm needs to continuously collect information on the recently published events and recalculate channels. Various clustering methods have been used in literature such as grid-based clustering and spectral clustering. Riabov et al. [25] propose grid-based algorithms using classical clustering techniques of computer science related to data mining. In this approach, the event-space is divided into cells and similar cells are placed in the same cluster. Even though this clustering mechanism efficiently clusters subscribers, with increasing number of subscribers or clusters the computation overhead increases significantly. As the clusters need to be periodically updated, this poses as a major drawback. An improvement to this approach has been presented in [22]. In [22], Tariq et al. propose clustering based on spectral methods in a distributed broker-less pub/sub system that utilizes concepts of graph theory to identify good quality clusters and reduces the cost of event dissemination. [9] proposes a way of implementing a pub/sub system based on channelization using software-defined networking where channels can be directly mapped to flows in the network. However, this thesis does not deal with clustering methods and rather focuses on an in-network filtering technique also presented in [9].

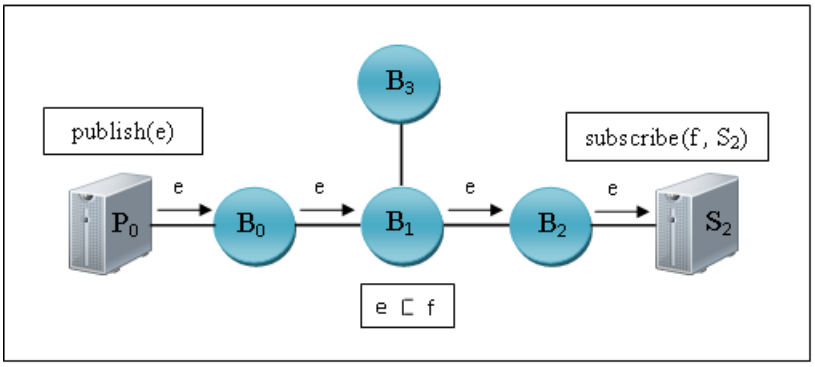



Figure 2.3: A publish/subscribe system with multiple brokers

A filter-based publish/subscribe system differs significantly from the previously discussed channelization method. In general, it consists of one or more brokers that process control messages dealing with advertisements and subscriptions and are responsible for the matching and forwarding of events. A single broker constituting the notification service would ideally be the easiest and most efficient way to realize a content-based pub/sub. However, this is not a feasible solution with respect to scalability. As a result, most implementations have a distributed set of brokers serving different clients and events are propagated along a chain of brokers. Figure 2.3 depicts a scenario where a system consists of four brokers. A publisher $P_0$ is connected to broker $B_0$ and a subscriber $S_2$ is connected to broker $B_2$. The subscriber subscribes for events that satisfy the filter *f*. $P_0$ publishes an event $e$ where $e \sqsubset f$. In order to propagate the event from $P_0$ to $S_2$, each of the brokers in the path should have information about the subscription from $S_2$. Clearly, subscription information needs to be shared among brokers which need to maintain their routing tables according to these subscriptions. In an advertisement model, advertisements of publishers also need to be shared among brokers. This brings us to the discussion on the two types of content-based routing, namely, simple routing and covering-based routing.

In simple routing, every subscription/advertisement from any subscriber/publisher in the network is flooded to every broker. So each broker has a global knowledge of subscriptions/advertisements. However, this results in unnecessary and redundant storage of information and large routing tables. This scheme is therefore not feasible in very large systems with a large, dynamic set of users. On the other hand, the covering-based scheme provides a mechanism to restrict the amount of subscription/advertisement information shared among brokers. A broker forwards a subscription/advertisement to its neighbors only if it has not formerly forwarded a subscription/advertisement which covers this one. We say that a filter $f_1$ covers a filter $f_2$ if $f_1$ defines a superset of the notifications defined by $f_2$, i.e. $f_1 \succ f_2$ [1]. So, if an event $e$ satisfies $f_2$ then it always satisfies $f_1$ but the reverse is not true. In the advertisement model, a broker can further restrict the flooding of a subscription by only sharing it with neighboring brokers from which an overlapping advertisement has previously arrived. Thus, this routing mechanism avoids unnecessary sharing of information between brokers and saves network resources. Taking advantage of this feature, this thesis presents a content-based pub/sub system which executes covering-based routing of advertisements and subscriptions between distributed logical intermediaries known as controllers in software-defined networking. As a result, software-defined networking is discussed in further details in the following section.

## 2.2 Software-Defined Networking

The advent of the network architecture, *Software-Defined Networking* has changed the existing approach to design, management and operation of the network. Changing a network is now a practical and feasible option. These changes are made possible in SDN through the separation of the control plane from the data plane, i.e., the control logic is removed from network devices such as switches and is hosted on a server called *controller*. In the classic approach,

each network device is responsible for maintaining information about its neighboring devices and forwarding traffic based on this information. SDN provides a way to have a centralized intelligence that captures a global image of the entire network and takes efficient and smart decisions when making network changes. In this way, classical distributed algorithms are now merely reduced to graph algorithms on the controller, providing easy means to experiment with the network. Something that was quite impractical even in the recent past.

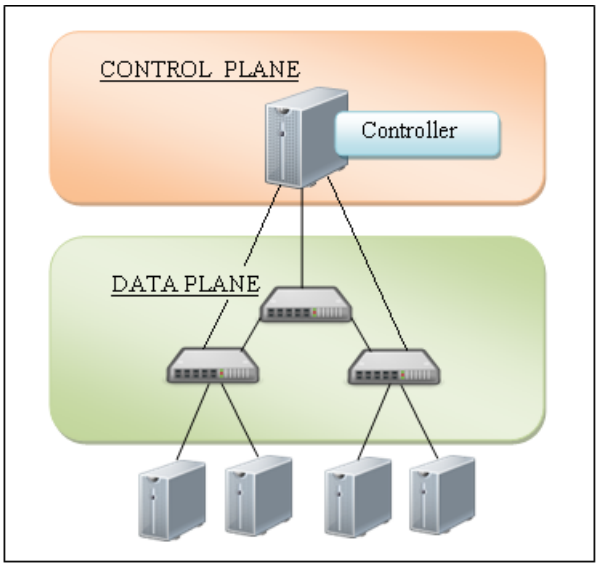



Figure 2.4: SDN Architecture

Figure 2.4 presents a brief overview of the SDN architecture with a single centralized controller which deals with the *control plane* and switches which are responsible for the *data plane*. To achieve this separation of planes, the controller provides two interfaces, namely, *northbound interface* and *southbound interface*. The southbound interface is between the controller and the switches and is commonly associated with the Openflow protocol. This interface allows various operations such as modification of flow tables, querying of traffic statistics etc. The northbound interface on the other hand is between the controller and application-specific control logic. This serves as an API to program the network for a programmer who does not need to know the details of the Openflow protocol. Through this interface, network information such as traffic statistics, topology etc. can be exposed to the application. This interface is instrumental in the translation of application events to Openflow events and vice versa.

Clearly, SDN provides limitless possibilities and immense flexibility to dynamically control the network. As a result, there are many real-world SDN use cases. These include load balancing, service chaining in data centers and dynamic enterprise WAN, traffic engineering for network operators etc. [26]. This thesis also deals with one such use case, i.e. publish/-subscribe middleware. [9] presents ways to realize content-based publish/subscribe using the capabilities of SDN and [10] provides a concrete implementation of one of these proposed

methods. So, this thesis is not the first attempt to realize pub/sub using SDN. However, [10] uses a single controller in the SDN architecture that serves every user request whereas this thesis attempts to distribute the load between multiple controllers while keeping the correctness of the system intact. The realization of pub/sub using distributed controllers in SDN is further discussed in the next section.

## 2.3 Pub/Sub using Controller/s in SDN

As discussed formerly, there has been a lot of work done previously on content-based publish/-subscribe. However, most of these have been restricted to application layer implementations where the application is responsible for serving the users, filtering and matching of events etc. Such an implementation limits the performance of the system significantly and the benefits of content-based routing become negligible as compared to simpler communication paradigms relying on line-rate processing of data packets at switches. Until the recent past, a practical implementation of content-based pub/sub on the network layer seemed like a far-fetched idea as it demanded special hardware and changes to existing network protocols. However, with SDN this has been made possible. Now, with a global view of a programmable network, the controller can locate the publishers and the subscribers and directly establish flows between them for the propagation of events to interested subscribers. In [9], Koldehofe et al. propose ways to push the expensive matching operation of events with subscriptions on to the hardware by mapping content-based routing to header-based routing. This facilitates the matching operation considerably as a hardware switch can execute matching of header information to its flow table entries very fast using Ternary Content-Addressable Memory (TCAM).

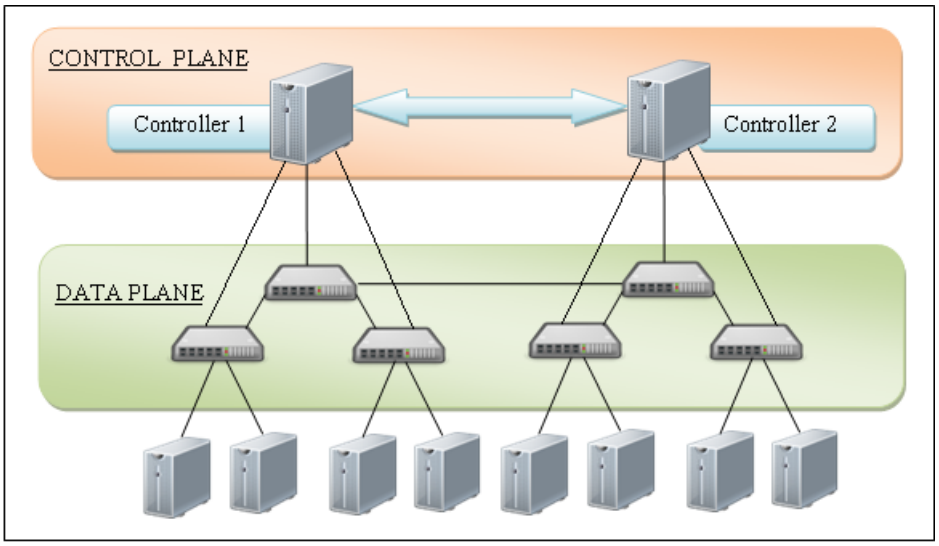



Figure 2.5: SDN Architecture with Multiple Controllers

This thesis utilizes the former and builds upon content-based pub/sub implementation presented by Mishra in [10]. We argue that a single SDN controller will be rendered less useful in a very large and dynamically changing environment. This is because, the controller alone is responsible for handling all advertisement and subscription requests, maintaining knowledge on the topology of the entire network and establishing flows over the entire network. Thus, to improve the scalability of the system, we present a solution with multiple controllers. Figure 2.5 depicts such a system with two controllers. Each of these controllers is connected to a disjoint set of switches, effectively partitioning the network into two halves. Each controller now needs to be aware of only a subset of the entire network topology, i.e., 3 switches and 4 hosts/controller in this figure. It also needs to add/delete/modify flows of only a subset of all the switches in the network, i.e., 3 switches/controller in this figure. This enhances the scalability of the system significantly but also raises questions about coordination and exchange of state information between the controllers in order to maintain correctness in the system. These questions and problems related to the distributed implementation of the control logic are discussed further in the following chapter.

# Chapter 3

# Line-rate Performance at Large Scale

This chapter is dedicated to the various challenges and problems encountered while designing a content-based pub/sub and the manner in which they are handled in existing pub/sub systems found in literature. Clearly, the main activities of a pub/sub implementation are event matching and notification delivery which emphasize the importance of the notification service and it is the design of this notification service that separates one system from the other. The simplest approach to implementing a notification service is to have a single broker serving every request in the system. So, every publisher and subscriber is connected to the same broker and every subscription/advertisement and event notification is targeted to it. The broker keeps track of all participating publishers and subscribers and active subscriptions/advertisements. So, when a notification arrives it can easily identify the subscribers to which the notification has to be delivered. Alternatively, in a distributed implementation, each broker is connected to a subset of all the participants of the system. Any publisher/subscriber sends requests to its local broker which acts as the access point to the entire network for it. So, effectively, the event matching and notification forwarding activities are distributed among multiple brokers in the system. Such a distributed setting raises questions with regards to the design of the system. One such design issue is the manner in which the brokers are interconnected in the network. One way of realizing the interconnection topology is to have a hierarchical structure. Another approach to designing the system is to organize the brokers in a peer-to-peer relationship. There can also be a third approach which is a hybrid of the first two approaches, i.e., peers of hierarchies or a hierarchy of peers. However, this approach requires a priori knowledge about the structure of the notification service's applications. Each of these approaches has their own complexities, advantages and disadvantages which have surfaced in the various works found in literature.

Another very important design issue is the selection of the routing algorithm used between the brokers. The main question that needs to be addressed here is what information needs to be communicated between brokers in order to maintain correctness in the system? Again, this problem can be approached in more than one way. A very naive solution would be to maintain subscriptions/advertisements at only local brokers and broadcast the notifications to all brokers in the network. So, each broker receives every notification, matches each of these notifications to its local active subscription set and delivers accordingly to interested subscribers. Another alternative is to broadcast all subscriptions to every broker. In this way only brokers with interested subscribers receive matching notifications. Yet, another

approach applies to the advertisement model where all advertisements are broadcast to all brokers. Then subscriptions are forwarded to local brokers having matching advertisements so that paths can be created between publishers and subscribers for the forwarding of notifications to only local brokers having interested subscribers. In the last two approaches, each broker stores information of either all subscriptions or all advertisements in the network. The last two approaches can be further improved by addressing the processing strategy design issue. The aforementioned approaches targeted broadcast and storage of every subscription/advertisement at every local broker. However, there may be similarities between the subscriptions/advertisements submitted to a local broker. So, a local broker broadcasts subscription/advertisement only if it has not previously broadcast a subscription/advertisement that covers this one. The concept of covering-based routing has already been introduced in the previous chapter and needs no further explanation here.

While designing a pub/sub system with distributed brokers, a few other factors should be considered. Any notification should be delivered exactly once to a subscriber and it should be propagated through the network only once. This can be achieved by avoiding cycles in the network by establishing acyclic paths between the publishers and subscribers. Also the coordination between the controllers should be done in an acyclic manner such that any subscription/advertisement request is processed and forwarded to relevant neighboring brokers only once by a broker. Such a design not only preserves correctness of the system but also reduces unnecessary usage of network resources. Another major design issue that emerges to be one of the most important in a content-based pub/sub system is the expensive matching operation of events against subscriptions. With a more efficient matching phase, better performance can be achieved with regards to latency and throughput.

## 3.1 State-of-the-Art

The past few decades has seen a lot of solutions addressing some of these aforementioned problems. Various approaches to the many aspects of a pub/sub have been presented in literature. The remaining part of this chapter discusses a few of the popular pub/sub systems that display distinct characteristic features. Chapter 2 gave a background of general publish/subscribe systems whereas this chapter discusses some concrete implementations of the same.

### 3.1.1 Elvin

Elvin[27, 28] is among the very first realizations of the pub/sub paradigm. The earliest version of Elvin had a centralized architecture that provided very simple and efficient event filtering mechanisms. However, as discussed earlier, a centralized event dispatcher is a major drawback with regards to scalability and as a result Elvin was extended to allow a federation of event brokers. In Elvin, events are represented as attribute/value pairs and the subscription language is based on predicates. The subscription language resembles C boolean expressions

and is supported by a wide range of operations for matching numeric as well as string values. Elvin also introduces a concept called source quenching where publishers gather information about the active subscribers interested in their events from the event brokers. If there are no active subscribers interested in a particular event that a publisher plans to publish, the event is no longer unnecessarily sent. Even though this mechanism reduces bandwidth usage and computation overhead related to events, it has proven to be very expensive due to the periodic collection of subscriber information by the publishers. The latter versions of Elvin have refined the source quenching mechanism considerably. However, a more effective approach to limiting subscription diffusion in the network is provided by SIENA which has been discussed later in this chapter in more details.

### 3.1.2 Gryphon

Another notable contribution to the field of pub/sub is Gryphon developed at IBM Watson research center[18]. Gryphon presents a content-based pub/sub framework which uses an algorithm presented by Aguilera et al. in [29] for its matching algorithm. [29] is further extended using another approach proposed by Banavar et al. in [30] to obtain an efficient multicast algorithm with partial matching at each broker in a distributed setting. The main idea of the matching algorithm in [29] is the traversal of parallel search trees where the non-leaf nodes correspond to constraints on attributes and the leaf nodes mark the end of a matched subscription. [30] extends it in a distributed setting where a tree data structure is built over the network with the brokers as the nodes of the tree and a single constraint is matched per routing step. Such an approach leads to flooding of all subscriptions throughout the network as every broker needs to maintain global knowledge of all subscriptions. This is the case of simple routing without advertisements. Gryphon produces impressive performance results through its matching algorithm but suffers from the inherent problems of simple routing as subscription flooding incurs significant network costs. Also, the matching algorithm has limited usability as it supports only a few types of attribute filters.

### 3.1.3 SIENA

One of the pioneers in the field of publish/subscribe is the Scalable Internet Event Notification Architecture (SIENA)[1] system. Maintaining a balance between scalability and expressiveness has always been a challenge in publish/subscribe and SIENA has been designed with both these factors in mind. It provides an efficient and scalable routing mechanism enabled by a network of brokers over a wide-area network. SIENA also represents events as attribute/value pairs where attributes are typed and filters are predicates over them. Subscriptions are nothing but conjunctions of these filters. A notification $e$ satisfies a subscription $s$, i.e., $e \sqsubset s$ if and only if $e$ satisfies every filter in the conjunction comprising $s$. Table 3.1 provides examples of the matching relationship as defined by SIENA.

| **Notification** | | **Subscription** |
|---|---|---|
| string sport = football | $\sqsubset$ | string sport = football<br>string team = Germany |
| string sport = football<br>string team = Spain | $\not\sqsubset$ | string sport = football<br>string team = Germany |

Table 3.1: Matching Operation

| **Subscription/Advertisement** | | **Subscription/Advertisement** |
|---|---|---|
| integer items > 50<br>integer price < 100 | $\prec$ | integer items > 0<br>integer price < 150 |
| integer items > 50<br>integer price < 100 | $\not\prec$ | integer items > 75<br>integer price < 150 |

Table 3.2: Covering Relationship

The main focus of SIENA is to avoid the flooding of notifications in the network by establishing logical paths between publishers and subscribers using two possible alternatives. In the first approach, there is no concept of advertisements. Subscriptions are flooded to every broker for event matching. So, each subscription triggers the creation of a diffusion tree spanning every broker so that each broker knows the exact direction in which it needs to forward a notification to reach an interested subscriber. The other alternative considers the advertisement model where advertisements are flooded to all brokers. Subscriptions are forwarded to brokers with publishers having matching advertisements only. Such a model further reduces the set of involved brokers. Both these mechanisms ensure the forwarding of notifications only to those parts of the network which have interested subscribers, reducing network costs significantly. SIENA uses covering-based routing that has been already discussed in Chapter 2. So, a subscription is forwarded by a local broker only if it has not already forwarded a subscription which covers this one. The same applies for advertisements. Table 3.2 depicts the covering relationship between subscriptions/advertisements as defined in SIENA. Covering-based routing further prunes spanning trees along which new subscriptions are routed, reducing the control message traffic significantly. This thesis uses a similar processing strategy where covering-based routing of dz-expressions is carried out.

As discussed in the previous chapter, the interconnection topology of brokers is an important design issue. SIENA presents mainly two topologies, namely, hierarchical and acyclic peer-to-peer interconnection topology. In peer-to-peer architecture, all brokers play the same role and are treated equally. In hierarchical architecture, each subscription is always forwarded to a parent broker. Also, each notification is forwarded by a broker to its parent broker and then selectively routed to subordinate brokers. Results show that a hierarchical setting is suitable when the subscriber density is low whereas a peer-to-peer setting is better when the notification traffic dominates the total cost of communication. A hierarchical architecture,

however, has some drawbacks. A broker higher up in the hierarchy may be overloaded as it has to process most of the requests and notifications. Also, it acts as a single point of failure in the design of the system. On the other hand, an acyclic peer-to-peer topology is much more general purpose and can be easily mapped to a realistic setting.

SIENA is a reference solution that addresses many of the problems of content-based pub-/sub but it has its own limitations. It only supports a fixed set of predefined types and predicates which limits the power of the subscription language. Additionally, it also suffers from the inherent problem of designing a pub/sub system on the application layer. The subscription/advertisement requests, coordination messages, notifications are all forwarded along paths established in the overlay network which may result in redundant forwarding of messages along the same physical link. Also, the expensive matching of notifications against subscriptions is carried out by the brokers on the application layer. Such a design not only results in wastage of network resources but also results in end-to-end delay and lower throughput rates.

### 3.1.4 JEDI

Java Event-Based Distributed Infrastructure (JEDI)[31] is a distributed content-based pub-/sub system with multiple brokers and a Java-based implementation. The publishers and subscribers in JEDI are referred to as active objects and brokers as event dispatchers. The event dispatchers are organized in a hierarchical architecture and follow a similar routing approach as the hierarchical topology in SIENA. As in the hierarchical setting of SIENA, subscriptions are forwarded upwards in the hierarchy whereas matching notifications are routed upwards and then downwards in the tree to be delivered to interested subscribers. JEDI does not present an advertisement model to further prune the propagation of subscriptions. JEDI focuses on the concept of selecting a group leader to create a dissemination tree called core-based tree. Any event dispatcher that wishes to be a part of the tree communicates directly with the group leader which incorporates the event dispatcher into the tree as a node. However, this approach requires every event dispatcher to be aware of all group leaders in the system. Also, the main drawback of JEDI is its hierarchical architecture that imposes heavy load on the root dispatchers and the failure of one such dispatcher might disconnect the entire sub-tree, resulting in loss of events.

### 3.1.5 Prefix Forwarding for Publish/Subscribe

Prefix Forwarding for Publish/Subscribe[32] is also based on the popular SIENA system. It attempts to improve the matching phase of SIENA by introducing the concept of prefix matching where instead of matching a notification at multiple brokers, it is done at a single broker. The other brokers in the path simply forward the notification towards the destination. With SIENA as the base, this system presents two additional approaches. Firstly, it introduces a normalization phase performed by the subscriber at the publish/subscribe layer. The normalized filters are further propagated to the brokers. SIENA does not restrict the constraints on

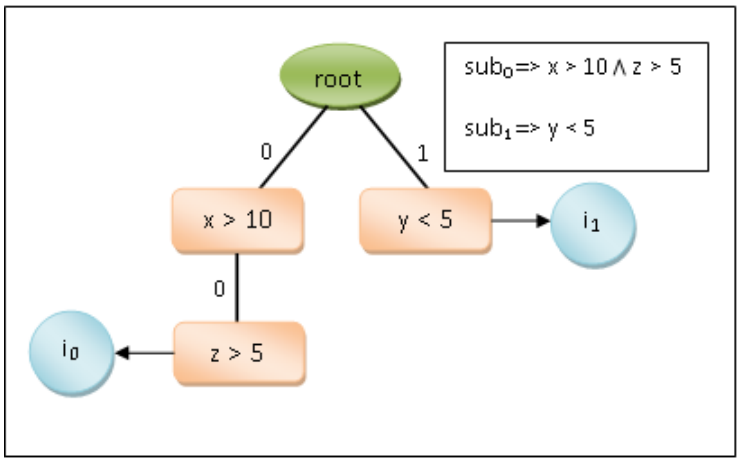



Figure 3.1: Creation of Routing Tree

attributes in a filter. This might result in redundancy in the filters. For example, *(x > 5 ∧ x > 7)* is allowed in SIENA, whereas in this approach it is normalized to *(x > 7)* which is more meaningful. These normalized filters are broadcast throughout the network of brokers. The next phase is the creation of a *Routing Tree(RT)* at an edge broker which is also maintained by all the other brokers in the system. The *RT* creation and modification is triggered by the arrival of a subscription. Each attribute constraint forms a node in the *RT*. So a subscription with multiple attribute constraints would result in multiple nodes at various levels of the hierarchy. For example a filter *(x > 5 ∧ z < 5)* would result in two nodes where one is the child node of the other. Again two separate filters *(x > 5)* and *(z < 5)* might be placed in the same level of hierarchy in the tree. In the RT, the outgoing interface is stored at each node where an entire subscription is matched which is used to route a notification if it satisfies the subscription. Fig. 3.1 illustrates the routing tree creation process where two subscription requests $sub_0$ and $sub_1$ from interfaces $i_0$ and $i_1$ are handled. Once a notification matching is done at an edge broker it is not repeated for the other brokers. Instead, a copy of the matched portion of RT known as Forwarding Prefix Tree(FPT) is attached to the notification before forwarding it to the next broker. The remaining brokers in the path simply parse the FPT to identify the outgoing interfaces.

Even though this method is an improvement over SIENA with regards to event matching, it has a few disadvantages. Firstly, the tree management operation is very complex and results in a lot of overhead due to the dynamic nature of subscription requests. Secondly, with increasing number of subscriptions, the tree size increases. This directly results in very large FPTs being attached to notifications and the forwarding of them in the network. This incurs enormous network costs and might not be feasible after a certain size of FPT. This problem may be addressed by limiting the size of FPT. However, this would result in increased false positives in the system which again might not be desirable. This indicates that an attempt to improve the expensive event matching phase at the application layer prompts the necessity of additional costs and has its own limitations.

### 3.1.6 LIPSIN

Line Speed Publish/Subscribe Inter-networking (LIPSIN)[7] is a topic-based pub/sub system which presents a novel multicast forwarding fabric on the network layer. Even though this thesis deals with content-based pub/sub, LIPSIN is worth mentioning in this context as it provides an idea about the performance of the current implementation. The structure of LIPSIN consists of the control plane and the data plane. The control plane deals with the topology system which identifies the underlying topology and the rendezvous system responsible for the matching of publishers and subscribers. LIPSIN introduces mainly two phases. The first phase is known as *recursive bootstrapping* where the control plane at the routers discover the underlying topology and communicate with each other in order to have a unified global view of the entire network. A global network graph is created at each of the routers and is used further in the forwarding process. The second phase is the forwarding phase where each link in the network is represented with link IDs which are bit-strings. LIPSIN uses the concept of *Bloom Filters*[33] also known as *zFilters* in this approach to encode these link IDs. The topology manager identifies the delivery tree for each topic, creates a zFilter for it and sends it to the concerned publishers to maintain a mapping between topics and zFilters. The zFilters are then attached to packet headers such that any node can easily identify the outgoing links along which the packets need to be routed by performing an AND operation of the zFilters and the outgoing link IDs.

Through this novel approach, LIPSIN is successful in achieving very good performance with respect to end-to-end delay and throughput. This approach uses bloom filters which completely avoid false negatives in the system. However, a bloom filter suffers from the inherent problem of false positives that lead to unnecessary delivery of information to uninterested subscribers. Also, the main down-side of this approach is that it is topic-based and bears the same problems in terms of expressiveness of subscribers as does any other topic-based pub/sub system. In spite of these disadvantages, this simple and efficient approach performs better than most of the aforementioned systems by performing line-rate processing of packets directly at the switches of the network. A design combining the best of both worlds, i.e., enhanced expressiveness of subscribers and line-rate processing of packets would prove to be an ideal solution addressing the problems of the aforementioned pub/sub systems. An attempt to build such a system has been made in [9] and is further discussed in the next section.

### 3.1.7 Event-based Systems Meet SDN

In [9, 34], Koldehofe et al. present two methods to realize content-based pub/sub using the capabilities of software-defined networking. The main idea is to have a controller provided by SDN which has a globally unified view of the entire network and is capable of establishing flows from the publishers to the subscribers directly on the switches through the Openflow protocol. The first proposed method is channelization where channels can be very conveniently mapped to flows in the setting of a flow-based approach. Each subscriber and publisher sends a list of the received events at regular intervals to an IP address that is reserved in the system for

communicating with the controller. Such an address, $IP_{fix}$ must be chosen in such a way that no flows in the system match it. When no match is detected at an Openflow switch, the packet is automatically forwarded to the controller. The controller hosts the routing optimization algorithm and performs spectral clustering of publishers and subscribers. Each cluster corresponds to a flow/channel and is designated a unique IP multicast address from the range of addresses available to the system. A minimum spanning tree is created corresponding to each channel and the flow tables of the switches are directly modified by the controller. Also, each publisher is made aware of all the unique IP addresses representing channels to which it wishes to forward its events along with their aggregated subscription information. This is to make sure that publications are not sent to the controller for flow determination every time and forwarding of events is restricted to the data plane. This results in an efficient mechanism to forward events at line-rate.

The second proposed approach is in-network filtering of events. This approach follows an advertisement model where every publisher announces the type of information it intends to publish before publishing it. It uses spatial indexing discussed in Chapter 2 for content space representation and represents advertisements, subscription and events as dz-expressions. Again, the controller is responsible for establishing flows between the publishers and interested subscribers. Each flow is represented by a dz-expression which corresponds to a sub-space in the content space. With the advent of a new advertisement/subscription, new flows are added/updated at the Openflow switches based on whether it covers an existing flow or it is covered by an existing flow or it has no relationship with any existing flow. This is further explained in the following chapter where an implementation of this approach by Mishra[10] is discussed in details. The main idea of this approach is to map dz-expressions representing advertisements and subscriptions to multicast IP addresses which serve as matching criteria in flows and install these flows on switches. In this way paths are established on the underlying network from publishers to interested subscribers. Events, also represented as dz-expressions, are mapped similarly to destination IP addresses in the header of packets to be published. In this way header-based matching of event packets is done at the TCAM memory of the Openflow switches resulting in line-rate forwarding of events. This approach overcomes some of the most concerning problems related to performance of the aforementioned content-based approaches and truly utilizes the benefits of content-based routing in publish/subscribe. However, this system considers a single controller which is responsible for the entire control logic of the system and can potentially act as a bottleneck and single point of failure. This point has been already highlighted in this thesis and is one of the main motivations towards designing a system with multiple controllers to improve scalability issues.

The survey presented above discusses pub/sub systems which are all relevant to this research work. In this thesis, on one hand, SIENA like systems provide the foundation of a content-based approach with multiple brokers and on the other hand systems like LIPSIN facilitate the idea of in-network filtering on the network layer. Also, systems such as [10] form the base on which the present implementation is designed and realized. A study of each of these systems unfolds the various advantages and drawbacks associated with them and helps identify factors

which need more attention and approaches that can be benefited from to achieve the desired goals and improve the selected performance factors for the present implementation.

## 3.2 Problem Statement

The survey above indicates that the works in literature concerning distributed network of brokers have been implemented in the application layer. Again, the content-based pub/sub implemented in the network layer providing line-rate forwarding considers a centralized controller in its design. So, to bridge the gap between both these approaches, the main objective of this thesis is the design and implementation of a content-based pub/sub system on the network layer using distributed controllers provided by software-defined networking. It attempts to maintain correctness, reduce false negatives and false positives, avoid cycles in the network and avoid duplicate delivery to subscribers while focusing on performance factors such as bandwidth efficiency, reduced latency, increased throughput and enhanced scalability. The thesis further studies and analyzes the effects on the scalability of the system when the number of controllers is varied.

# Chapter 4

# Centralized Control Algorithms

This thesis builds upon the works of Mishra[10] which implements in-network content-based routing presented by Koldehofe et al.[9] using software-defined networking. The implementation uses a single Floodlight controller which has been extended in the present implementation to a set of distributed controllers. This thesis provides a design where the network is spatially divided among multiple controllers such that each controller serves as an access point for a disjoint subset of publishers and subscribers from the complete set of participants. Internally, within a local network associated with a controller, algorithms presented in the centralized approach have been used. As a result, it is necessary to first understand the centralized control algorithms before proceeding to the distributed approach. The main goals of the centralized algorithms are :

1. exactly-once delivery of events to interested subscribers
2. avoidance of false positives
3. keeping a check on flow table size by installing only required flows at the switches

## 4.1 Content Representation and Event Matching

Mishra[10] uses spatial indexing discussed in Chapter 2 for content representation. So, subscriptions, advertisements, un-subscriptions, un-advertisements and events are all represented as binary strings called dz-expressions. Such a representation is ideal for this approach and easily supports containment relation among subscriptions and event matching. Let us consider an example from [10] to explain containment relation representation in this design. A subscription $s_1$ with dz $\{00\}$ is a sub-space in the content space which covers both subscription sub-spaces $s_2$ with dz $\{000\}$ and $s_3$ with dz $\{001\}$ . So, in previously introduced notations, $s_1 \succ s_2$ and $s_1 \succ s_3$ and it can be seen that the dz of $s_1$ is the prefix of both $s_2$ and $s_3$. Also, $s_2$ and $s_3$ are completely disjoint in space and neither $s_2$ forms the prefix of $s_3$ nor $s_3$ forms the prefix of $s_2$. So, the containment relation among subscribers can easily be mapped to prefix matching of dz-expressions. The same concept can also be applied to matching of events with subscriptions. So, an event $e$ matches a subscription $s$ if and only if the dz of $s$ is the prefix of the dz corresponding to $e$. Basically, this means that the event $e$ lies in the sub-space corresponding to $s$.


Figure 4.1: dz translation

This approach makes use of IP address matching for header-based matching of packets with the flows (representing subscriptions) installed on switches. dz-expressions representing subscriptions, advertisements and events are mapped to IP addresses. In [9], Koldehofe et al. propose the use of IPv6 addresses to accommodate fairly long dz-expressions. However, the centralized implementation uses IPv4 addresses due to the limitations of Openflow 1.0. An IPv4 address range (225.128.0.0 - 255.255.255.255) available for pub/sub traffic is used. So, a dz-expression {0101} can be converted to an IPv4 address as depicted in Fig. 4.1 by simple concatenation of it after the first 9 fixed bits in the address. After conversion, the dz {0101} becomes 225.168.0.0. Again, a dz-expression {01011} is converted to 225.172.0.0. The former dz covers the latter and there should be a way to represent the containment relation in the flows as well. This is achieved through CIDR[35] style masks in the flows installed at the switches such that 225.168.0.0/13 contains 225.172.0.0/14.

## 4.2 Content-Based Filtering and Routing

The main purpose of a pub/sub system is the dissemination of notifications from publishers to relevant subscribers. To achieve this, routing trees are created over the network of switches. This approach defines routing trees as spanning trees which span the entire network such that every switch is covered only once. This ensures removal of duplicate message delivery at the subscribers and guarantees requirement 1. This phase can be compared to protocols like OSPF[36] which are commonly executed in a distributed network topology. However, with the controller having a complete view of the network topology, this phase is reduced to a simple graph problem. Spanning tree creation takes place at the controller and is publisher-driven with a publisher as the root and each spanning tree associated with a dz family. A dz family is simply a set of dzs with a common prefix. For example, dzs {00}, {000}, {0000} and {001} belong to the same family covered by {00} in this context as each of them have the same prefix {00}. The following sub-section explains the tree creation phase in further details.

### 4.2.1 Advertisement Handling

The tree creation process starts with the arrival of an advertisement at the controller. Advertisement and subscription requests are sent to a controller from publishers and subscribers by sending them to a fixed reserved multicast address $IP_{fix}$. No flows installed on switches can have this IP address for event matching so that every control message can be sent to the controller for further processing. On arrival of an advertisement request, the controller duly notes the dz-expression associated with the advertisement along with the identity of the

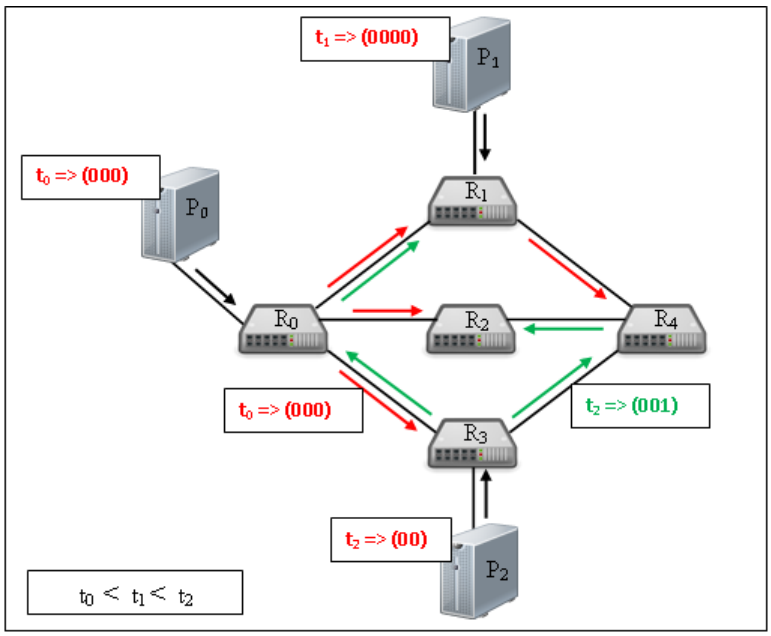



Figure 4.2: Spanning Tree Creation

publisher (switch-port tuple connected to it) that sent it. There can now be three situations depending on the value of the dz-expression.

- The dz expression does not have any containment relationship with any other existing dz family associated with one or more existing spanning trees.
- The dz-expression is equal to or covered by the dz family of one or more existing spanning trees.
- The dz-expression covers the dz family of one or more existing spanning trees.

These three cases are explained further with an example from [10] depicted in Figure 4.2. Figure 4.2 assumes the arrival of an advertisement {000} from $P_0$ at time $t_0$. Considering that there are no existing spanning trees associated to this family of dzs, the controller creates a spanning tree associated with the dz {000} and with $P_0$ as the root of this tree. The tree is created simply by using the *Breadth First Search* algorithm[37] on the network graph. The spanning tree thus created is depicted in the figure with red arrows and portrays the first case. At time $t_1$, another advertisement {0000} is sent by $P_1$ and this corresponds to the second case mentioned above where a tree with dz {000} already exists. In this scenario, since the former request covers this one, the publisher $P_1$ simply joins the existing tree. The final scenario can be explained with the arrival of the advertisement {00} at time $t_2$ from $P_2$. In such a situation, the dz is further split into {000} and {001}. So, on one hand, $P_2$ joins the existing tree for {000} and on the other hand, a new tree is created corresponding to {001} with $P_2$ as the root. The newly created tree is shown in the figure with green arrows. Now, if another advertisement {100} arrives at the controller, a completely disjoint tree would be created as

the concerned dz does not belong to the existing dz family and this again corresponds to the first case.

Another aspect that needs to be looked into while handling an advertisement is to identify existing subscriptions that match the current advertisement and establish paths between the publisher and relevant set of subscribers along the spanning trees relevant to this advertisement. This can be further understood later while discussing route calculations.

### 4.2.2 Subscription Handling

There can effectively be two cases associated with the arrival of a subscription request at the controller.

- No spanning tree associated with the subscription dz exists because no advertisement corresponding to this dz family has arrived previously from any publisher.
- One or more spanning trees relevant to the subscription dz exist. Here, a publisher of a tree is relevant to a subscriber if
  - advertised(adv) dz is equal to subscription(sub) dz, i.e., $dz_{adv} = dz_{sub}$
  - advertised(adv) dz is covered by subscription(sub) dz, i.e., $dz_{adv} \prec dz_{sub}$
  - advertised(adv) dz covers subscription(sub) dz, i.e., $dz_{adv} \succ dz_{sub}$

In the first scenario, the subscription details are simply stored at the controller for possible future actions because of the absence of a current relevant spanning tree. The second case is when a spanning tree relevant to the subscription dz exists. In such a scenario, paths are constructed from a publisher to a subscriber by establishing flows on the switches. Route calculation between a publisher and a subscriber is done using the very popular tree parsing procedure called lowest common ancestor algorithm while traversing towards the root of the spanning tree from both end nodes. This process returns a list of switch-port tuples which determine the switches on which flows need to be established along with the port through which a matching event should be routed so that connectivity is achieved between a publisher and an interested subscriber. In order to understand the process of establishing paths along the switches of the network, it is necessary to understand the flow addition/modification procedure on the switches.

#### Flow Structure

In Openflow, flows have specific structures of which the *match fields*, *priority field* and the *action rules* are utilized in this approach. Among the *match fields*, the *incoming-port* and *destination IP* fields are used. As explained earlier, advertisements, subscriptions and events are all represented by dz-expressions which can be mapped to corresponding IP addresses. So, effectively, the destination IP of a flow corresponds to a subscription space. The destination IP corresponding to a subscription dz is set with a mask such that all events which lie in the

subscription space defined by it are matched to the flow. For example, a subscription space {000} would be represented in a flow as the destination IP 225.128.0.0/12 and an event with dz {0001} would match the flow. The route calculation process from the publisher to the subscriber produces a list of switch-port tuples. Each tuple indicates the output port for that switch in the route and a packet matching the flow gets forwarded along this output port set in the *actions field*. Other than the output port, another action considered is set-destination-IP. This field is used to set the destination IP address of the packet to the IP at which a subscriber is listening at a terminal switch.

### Flow Addition

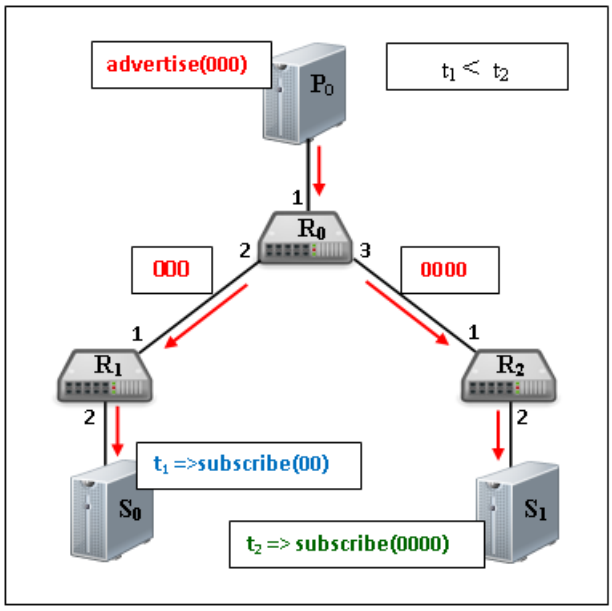



Figure 4.3: Flow establishment

The flow establishment process may be further explained with the arrival of subscriptions in different scenarios. The first case is when there is no existing flow. Figure 4.3 depicts such a scenario with an example from [10]. In the figure, there already exists a publisher $P_0$ with an advertisement request for dz {000} which triggers the creation of the depicted spanning tree for that dz. At time $t_1$, a subscriber $S_0$ subscribes for the subscription space of {00}. The algorithm uses the minimum dz between that of the publisher and the subscriber for its flow and so the IP corresponding to the dz {000} is used as a matching field in the flow. Choosing the minimum dz ensures requirement 2 which avoids false positives. In this example, flows are installed at switches $R_0$ and $R_1$ with output ports set to 2 and 2 respectively. Now, we consider another case where flows relevant to an incoming subscription already exist. So, at time $t_2$, another subscriber $S_1$ subscribes for the space associated with the dz {0000}. Now, flows corresponding to dz {0000} need to be installed on $R_0$ and $R_2$. The addition to $R_2$ is simple but the addition to $R_0$ is slightly more complicated. Firstly, a flow needs to be

| Match fields | Priority | Action Rules |
|---|---|---|
| input-port = 1<br>dest-ip = 225.128.0.0/13 | 1 | output = 2,3 |
| input-port = 1<br>dest-ip = 225.128.0.0/12 | 0 | output = 2 |

Table 4.1: Example flows at $R_0$ during subscription handling

installed at $R_0$ with match field corresponding to {0000} and output ports set to both 2 and 3. This is because any event matching {0000} must be forwarded to both $S_0$ and $S_1$. Also, this matching should be done before matching an event with the flow corresponding to {000} where the event is forwarded only to $S_0$. The matching process at the switch is stopped as soon as the first match is found and that is why matching with the flow corresponding to {0000} must occur first every time. This is ensured by setting the field called *priority*. Matching at a flow with higher priority gets preference over one with lower priority. So, in this way, all packets matching {0000} are forwarded to both subscribers but all packets matching {000} and not {0000} are forwarded to only $S_0$. Table 4.1 gives an idea about the flows at $R_0$ at this point of time.

### Flow Upgrade

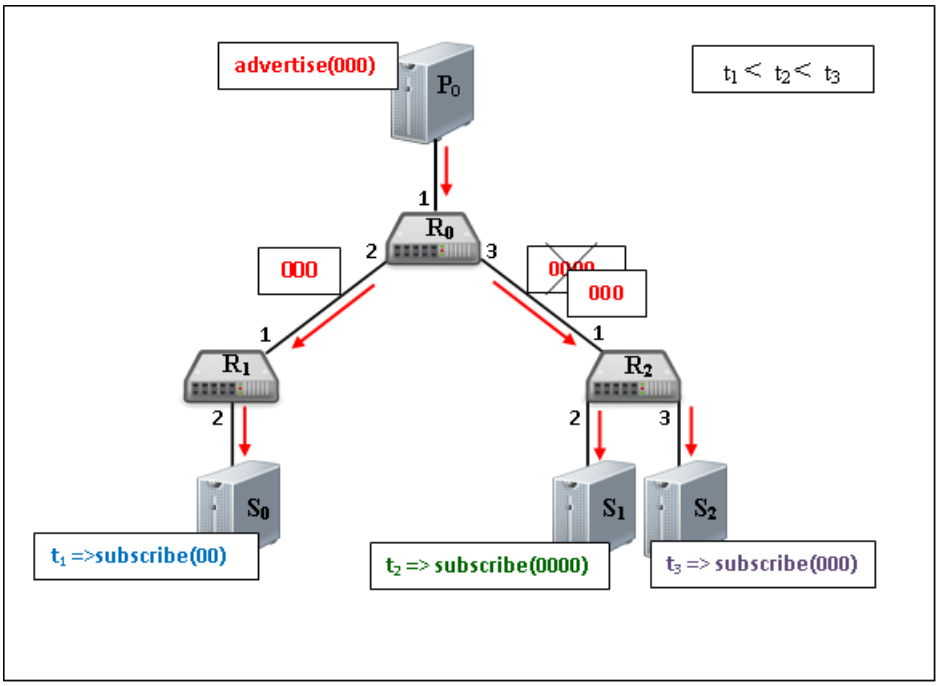



Figure 4.4: Flow upgrade

| Match fields | Priority | Action Rules |
|---|---|---|
| input-port = 1<br>dest-ip = 225.128.0.0/12 | 1 | output = 2,3 |

Table 4.2: Example flows at $R_0$ after flow upgrade during subscription handling

There may be yet another situation portrayed in Fig. 4.4 where another subscriber $S_2$ joins the same tree with a subscription {000} at time $t_3$. This results in flow addition at $R_2$ similar to the process discussed above. However, again, the process is more complex at $R_0$. It is clear from the figure that there is already an existing path from $R_0$ to $R_2$ corresponding to the previous request {0000}. Since the new dz {000} covers the existing dz {0000}, flow upgrade at $R_0$ needs to be done. As a result, the existing flow corresponding to {0000} is deleted and output port 3 is added to the flow corresponding to {000} along with previously set output port 2. Table 4.2 shows the flow table at $R_0$ at this point of time. The replacement of the existing flow which is now redundant in a flow upgrade operation contributes towards goal 3 which ensures the removal of redundant flows. Yet another scenario would have surfaced if the new subscriber $S_2$ subscribed for {0000} instead of {000}. Then, no changes would be required at $R_0$ and at $R_2$ an additional output port 3 would be added to the already existing output port 2 in the flow corresponding to {0000}.

### 4.2.3 Un-subscription Handling

An un-subscription request should be handled by a controller to make sure that false positives are kept at a minimum and no uninterested subscriber is delivered information as per requirement 2. So, on arrival of an un-subscription request, a recursive traversal of the tree is done from the terminal switch towards the publisher using the depth first search algorithm. Flows in each switch may be deleted/updated in this traversal. The traversal stops when an alternate subscriber is available or when the switch is a terminal switch. Again, with respect to flow modifications, mainly three cases are considered as depicted in Fig. 4.5.

#### Flow Deletion/Downgrade

The same example introduced above is extended considering the existing flows established in the subscription handling section. In the first scenario, at time $t_4$, $S_0$ un-subscribes, resulting in the deletion of the flow corresponding to this subscription at $R_1$. Since no other subscriber related to this subscription connected to the current switch exists, a traversal to the next switch connected to the input port of the deleted flow is carried out. In this case, as the input port of the deleted flow is 1, a traversal to switch $R_0$ is done. At $R_0$, a second case is encountered where there is an existence of another subscriber which has similar subscription space. As a result, the tree traversal is stopped and at $R_0$ the output port 2 is removed from the action field of the flow corresponding to {000}. The third scenario can be explained with an un-subscription by $S_2$. This is the case of flow downgrade complementary to flow upgrade.

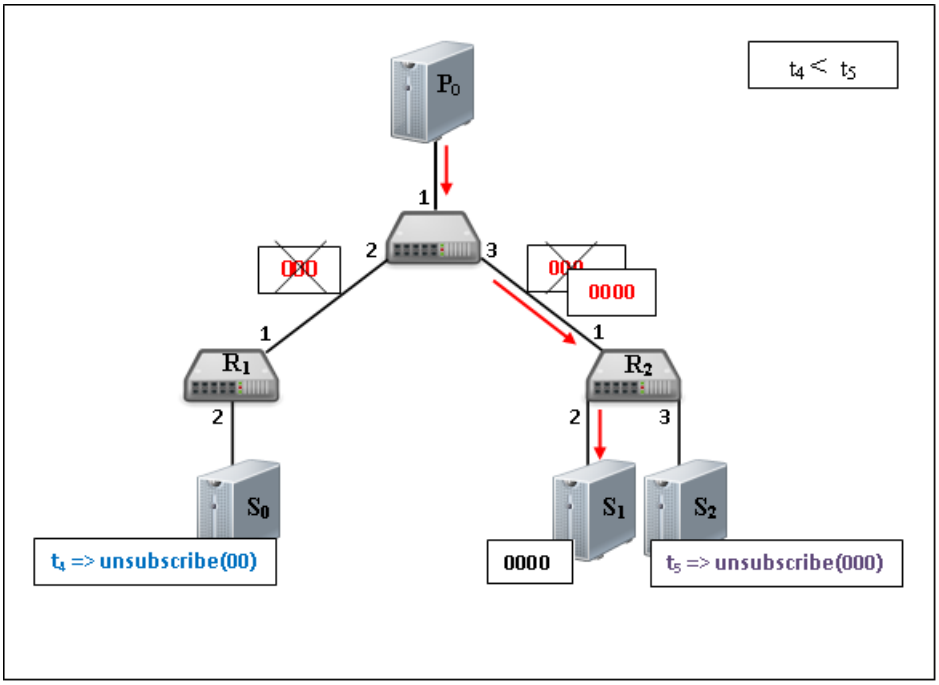



Figure 4.5: Un-subscription

| Match fields | Priority | Action Rules |
|---|---|---|
| input-port = 1<br>dest-ip = 225.128.0.0/13 | 1 | output = 3 |

Table 4.3: Example flows at $R_0$ during un-subscription handling

Here, at $R_2$, a simple flow deletion takes place. However, at $R_0$, the flow corresponding to this subscription is downgraded to a flow representing subscription space {0000} as the subscription of $S_1$ which was earlier covered by {000} is still active. Table 4.3 depicts the flow table at $R_0$ at this point of time.

### 4.2.4 Un-advertisement Handling

An un-advertisement should be processed by the controller by removing existing paths relevant to the corresponding advertisement so that no unnecessary flows remain installed on the switches as per requirement 3. An un-advertisement request from a publisher is handled in the same manner as an un-subscription request. On arrival of an un-advertisement request, for all the associated trees, a depth first search traversal is carried out with deletion or downgrade of relevant flows from the switches as they are encountered. The traversal stops when an alternate publisher is available or when the switch is a terminal switch. Figure 4.6 illustrates a case where the previous example introduced in the subscription handling section is extended

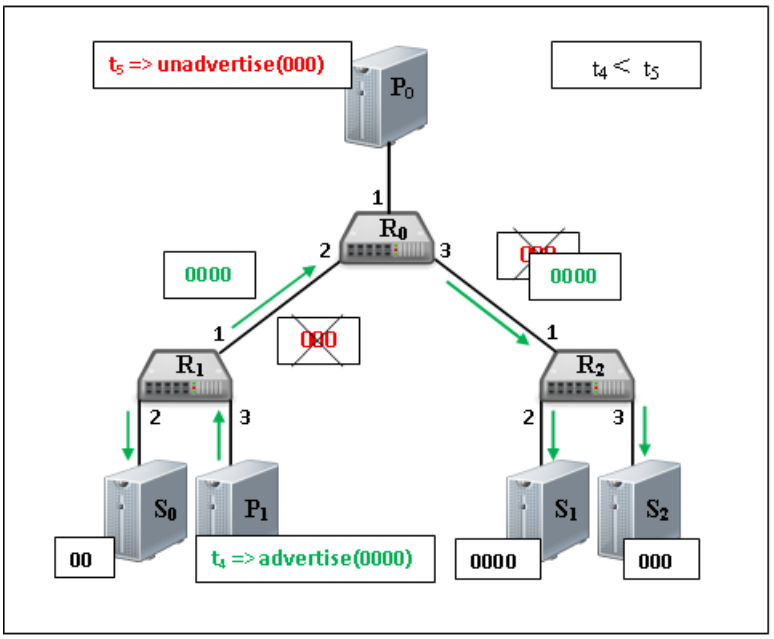



Figure 4.6: Un-advertisement

| Match fields | Priority | Action Rules |
|---|---|---|
| input-port = 2<br>dest-ip = 225.128.0.0/13 | 1 | output = 3 |

Table 4.4: Example flows at $R_0$ during un-advertisement handling

by including a second publisher $P_1$ with an advertisement space of {0000} at time $t_4$. This results in more flow adding/updating actions in the tree due to already existing interested subscribers $S_0$, $S_1$ and $S_2$ . With respect to such a setting, an un-advertisement by $P_0$ at time $t_5$ is illustrated which results again in deletion and downgrade of relevant flows in the network similar to the un-subscription process as depicted in the figure. Also, the flow table at $R_0$ at this point of time is shown in Table 4.4.

This chapter discusses the main characteristics of the prototype implementation of [10]. The algorithms for advertisement, subscription, un-advertisement and un-subscription handling are extremely important for this thesis as they are used within the local network of each controller which locally maintains spanning trees and associated flows. [10] also presents pseudo-codes for each of these algorithms which can be referred to for more details. With an understanding of the centralized pub/sub system, the next chapter presents an extension of the same to distribute the control logic between multiple controllers having disjoint views of the system.

# Chapter 5

# Distributed Control Algorithms

This chapter is dedicated to the distributed control algorithms used in the proposed design and implementation. The first step towards the realization of such an approach is to have multiple controller instances where each controller, instead of having a global view of the topology, has a local view of a dedicated part of the network. The identity of a particular network element (switch/host) is known only by a single controller from the total set of controllers. So, each network participant communicates with its own dedicated controller which acts as its access point to the system. Fig. 5.1 illustrates a pub/sub with 2 controllers, 6 switches and 4 hosts where the network is divided between the two controllers. The purple section depicts the part of the network assigned to controller $C_0$ and the blue section to controller $C_1$. Here, hosts $P_0$ and $S_0$ communicate directly with only controller $C_0$. Similarly, hosts $P_1$ and $S_1$ communicate directly with only controller $C_1$.

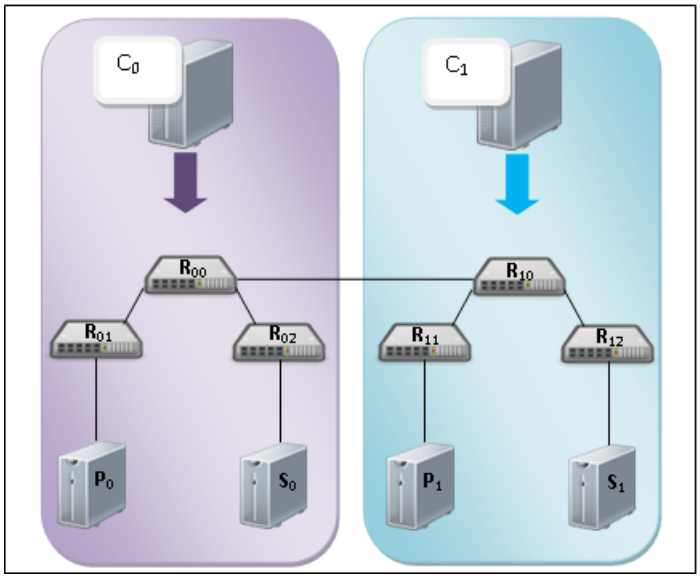



Figure 5.1: Distributed control logic

The controllers also have means to communicate among themselves in order to share information about their own local network. The system is designed in such a way that it can

support any number of controller instances depending upon the spatial partitioning of the network into disjoint sub-networks.

## 5.1 Problems

Before discussing the manner in which publisher/subscriber requests are handled, the main problems to be solved through the algorithms should be identified as follows :

1. A subscriber, irrespective of its relative position in the network, should eventually receive every event satisfying its subscription published at most after a time period $\delta$ from the time it issues the request, i.e., paths should be established between every publisher and all its relevant subscribers across sub-networks irrespective of the partitions in the network. So, connectivity should be ensured.
2. Exactly once delivery to each subscriber must be guaranteed.
3. Sharing redundant information between controllers must be avoided as this results in increased control traffic and unnecessary bandwidth usage.
4. Controller overhead should be minimized.

The design of the system and the proposed control algorithms work together to fulfil the above requirements in the system.

## 5.2 General Design Concepts

Before going into the details of the algorithms, a few important aspects related to the system design must be discussed. Firstly, it is important to classify the types of requests that need to be handled by a controller. Then, the manner in which controllers communicate among themselves is discussed followed by the way in which the centralized algorithms are incorporated in the distributed framework. This is followed by a discussion on the type of state information stored at each controller. Finally, the interconnection topology of the controllers and the reason behind its use are stated.

### 5.2.1 Pub/Sub Traffic

The publish/subscribe network traffic mainly consists of five types of messages. These are generated events and 4 types of control messages, namely, advertisement, subscription, un-advertisement and un-subscription request messages. The control messages can be further categorized into local and remote. For example, there can be a separate message type for local advertisement request sent by a local host and a separate one for remote advertisement request sent by a remote controller. If message types are used to distinguish local control traffic from remote control traffic, then, nine types of messages exist. The control messages have the same structure with three fields, namely, type of message, dz length and dz-expression.

So, eight types of control messages are directed towards controllers. A controller parses the control messages to identify the type and associated dz for further processing. The published events, however, are not directed towards the controller and simply follow routes established on the switches through event matching.

### 5.2.2 Communication between Controllers

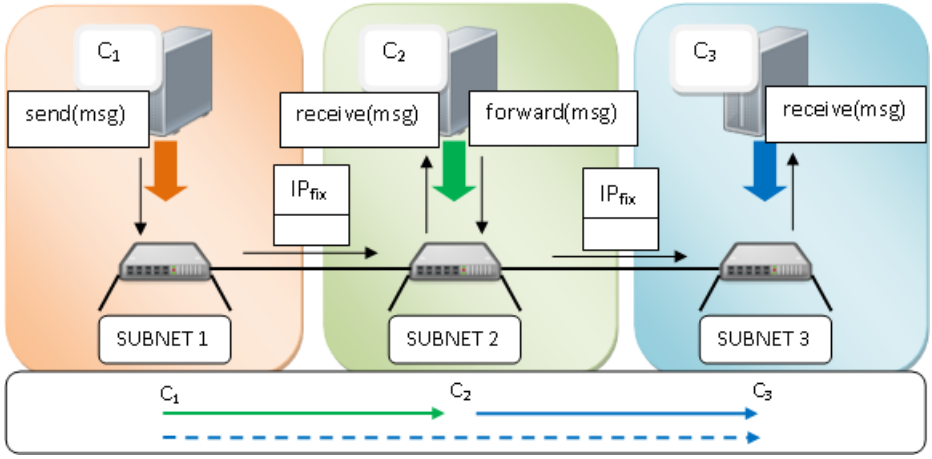



Figure 5.2: Communication between Controllers

The design of the present implementation ensures that each controller is only aware of the part of the network assigned to it and is completely unaware of the remaining topology of the system. Also, a controller does not know the identity of any of the other controllers in the network. The system has been designed in such a way so that controller overhead can be minimized as per requirement 4. However, in the process of saving overhead costs, questions are raised on connectivity between sub-networks in the system which can only be achieved if the controllers share the received local requests with one another. Since a controller does not have any idea about the total number of controllers and their identities, it cannot directly contact each of the other controllers for information sharing. However, since each subnet is connected by physical links to its neighboring subnet, the idea is to only contact the neighbors if any information has to be shared throughout the network. The neighbors in turn share the message with their neighbors and in this way the same message is shared among all controllers without each one having a global view of the others. The idea central to communication between controllers is that just like the hosts, a controller communicates with another controller by sending the packet to be shared to an address $IP_{fix}$. However, the main difference to the host requests is that the controller finds means to introduce this packet into the remote subnet of the neighboring controller with whom communication is desired. Once the packet is injected into the neighboring subnet, it is redirected to the controller assigned to that subnet by the very first switch it encounters as $IP_{fix}$ cannot have a match throughout the network. So, the packet finally reaches the desired destination and is processed accordingly

by the remote controller. In this fashion, state information is shared between controllers. Fig. 5.2 illustrates this idea. Here, $C_1$ simply introduces the message *msg* to its neighboring subnet. On receiving it, $C_2$ forwards the same to its neighboring subnet. In the process, $C_1$ manages to share information with both $C_2$ and $C_3$ in spite of not knowing their identities. Which request message should be sent/forwarded to which neighboring controller depends on various conditions and is explained later in the control algorithms.

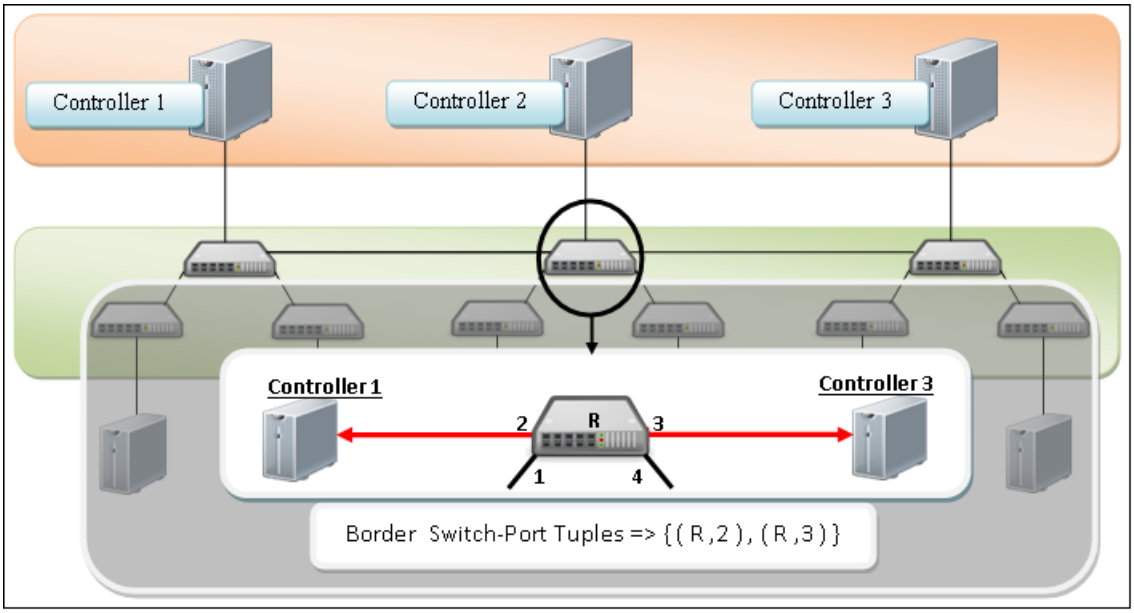



Figure 5.3: Border Switch-Port Tuples

However, an important problem still needs to be addressed. How does a controller which is only aware of its local subnet introduce the packet to be shared into a neighboring subnet? The solution to this problem is obtained by ensuring that each controller is aware of the switches and the ports of its local network which form the gateways to the outside network topology. So, a controller identifies the switch-port tuples of its local network which have links to the adjoining sub-networks and uses them to directly contact the neighboring controller/s. Any information that needs to be sent to a neighboring controller is sent through the respective switch-port tuple. Fig. 5.3 depicts a system with three controllers dividing the network into three parts where the scenario is explained with respect to Controller 2. In the figure, sub-network 2 controlled by Controller 2 has a switch R which serves as a border switch with its ports 2 and 3 linked to adjoining sub-networks. So, in the figure, if Controller 2 wishes to send a control message ((un)advertisement/ (un)subscription) to Controller 3, it does so by sending it through switch-port tuple (R, 3) with the destination IP set to $IP_{fix}$. On the other hand, Controller 3 receives the packet through its corresponding gateway to Controller 2 and information sharing is accomplished.

### 5.2.3 Incorporating Centralized Control Algorithms

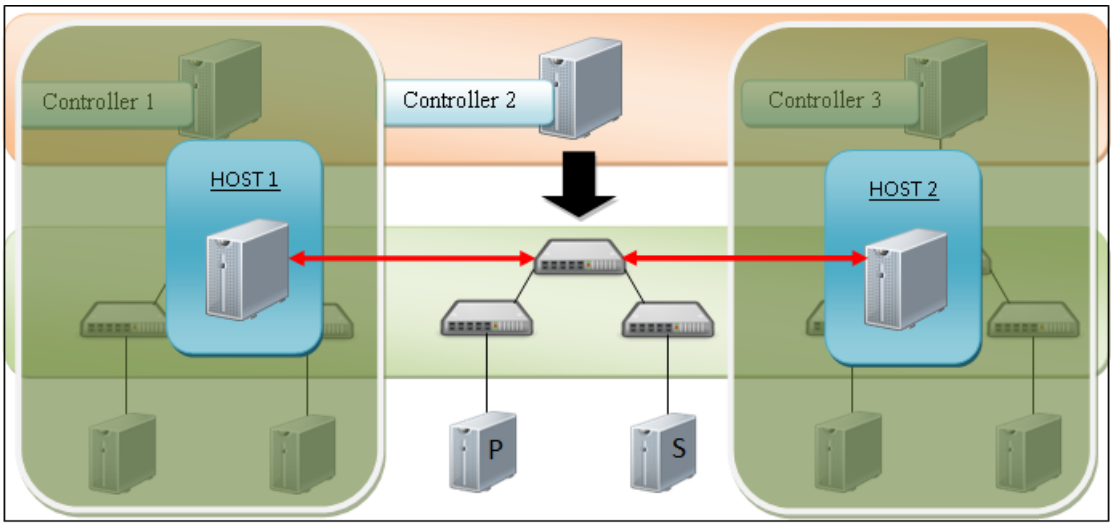



Figure 5.4: Virtual hosts

Every controller optimizes its own control network, i.e., the centralized control algorithms discussed in chapter 4 are applied independently by each controller. However, in spite of this autonomous behavior, connectivity is achieved between publishers and subscribers spread across different partitions of the network by the sharing of state information among controllers. Also, when remote controller requests arrive, a controller perceives its neighboring sub-network/s as virtual host/s (publisher/subscriber). So, when a neighboring controller sends a request, the controller considers this request to have arrived from a virtual host connected to its border gateway. This can be further explained in Fig. 5.4 where the network is again partitioned between Controller 1, Controller 2 and Controller 3. Here, Controller 2 views the parts of the network assigned to Controller 1 as a host named Host 1 and that assigned to Controller 3 as another host named Host 2 connected to its border gateways. For example, all requests sent by Controller 3 appear as requests from Host 2. Similarly, Controller 1 views the sub-networks 1 and 2 as a single host connected to its only border gateway and the same is true for Controller 3. Requests from virtual hosts are treated the same way as those from actual local hosts. The centralized control algorithms can then be applied locally considering the virtual hosts as part of the local network. This ensures connectivity as per requirement 1 and is explained in more details below.

#### Example of pub/sub tree management and route calculations across subnets

Fig. 5.5 illustrates two sub-networks controlled by $C_0$ and $C_1$ respectively. The publisher $P_0$ sends an advertisement $\{00\}$ at time $t_0$ to $C_0$ and the subscriber $S_1$ sends a subscription request $\{00\}$ at time $t_1$ to $C_1$. The requests are shared between the controllers to establish a route across the two sub-networks. This is achieved by considering the publisher/subscriber of the adjoining sub-network as a local host. So, $C_0$ views $S_1$ as a host $VS_1$ connected to its

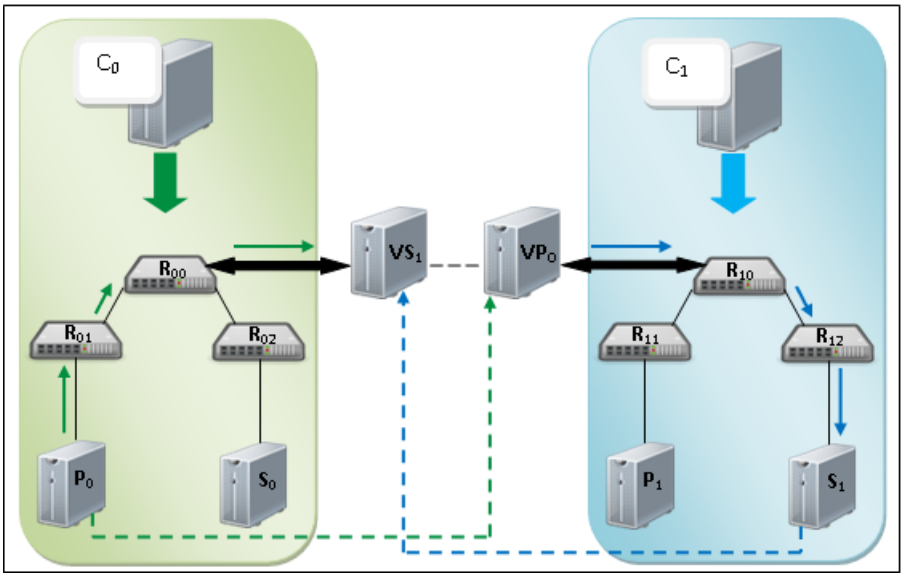



Figure 5.5: Route calculations across sub-networks

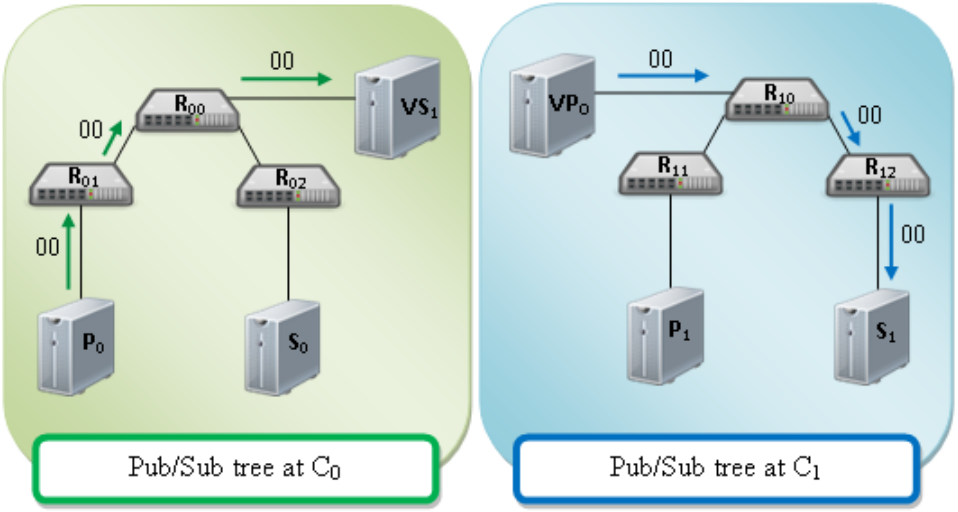



Figure 5.6: Pub/Sub trees

border port at switch $R_{00}$ and establishes flows between $P_0$ and $VS_1$ using the same methods as in the centralized approach using the local pub/sub tree where $P_0$ features as a publisher node and $VS_1$ features as a subscriber. Similarly, $C_1$ views $P_0$ as a host $VP_0$ connected to switch $R_{10}$ and establishes relevant flows in its own sub-network. The corresponding local spanning trees created at $C_0$ and $C_1$ are depicted in Fig. 5.6 and are similar to the spanning tree examples discussed in chapter 4. The spanning tree at $C_0$ is rooted at $P_0$ and that at $C_1$ has $VP_0$ as the root. The final result is a path from $P_0$ to $S_1$ as desired and displayed by the arrows in Fig. 5.5. The example can be extended to support many sub-networks. This is how the centralized advertisement and subscription algorithms fit into the distributed setting. The centralized algorithms for un-advertisements and un-subscriptions also fit into the framework in the same way.

### 5.2.4 State Information at a Controller

It is very clear from the discussion above that a controller needs to identify its border gateways to be able to share state information with other controllers. So, a list of all border switch-port tuples of the local network must be stored at the controller. This list has been referred to as *borderGateways* later in the algorithms. It is important to note that every host, i.e., a publisher or a subscriber is identified by the switch and the port to which it is attached and each request is represented by a dz-expression as in the centralized approach. So, local pub/sub trees are maintained by each controller as per the centralized control algorithms. Along with this a pair of lists for keeping counts of local publishers called *localPublisherCounts* and local subscribers called *localSubscriberCounts* corresponding to arrived dz-expressions is maintained. Each element in these lists has the structure *<dz, count>*. Another pair of lists later referred to as *remotePublishers* and *remoteSubscribers* are kept. Each element in this list consists of a pair of values which are the dz-expression representing the request and the switch-port tuple(nodePortTuple) representing the location of the virtual remote publisher/subscriber, i.e., *<dz, nodePortTuple>*. These lists maintained at each controller are instrumental in deciding the amount of state information to be shared and the adjoining controllers with which to share them.

### 5.2.5 Interconnection Topology

An important requirement associated with correctness of the system deals with exactly-once delivery of events to interested subscribers as mentioned in requirement 2. In order to ensure this property, an acyclic peer-to-peer interconnection topology for multiple controllers has been designed in this thesis. This means that each sub-network has a single path to any other sub-network in the system. So, each controller communicates with another controller through only a single path. In the centralized approach, exactly-once delivery is guaranteed as spanning trees are created covering every switch exactly once. Since, in the distributed framework, spanning trees are used in the same way to establish routes between local/virtual publishers and local/virtual subscribers, there is always exactly one path connecting two

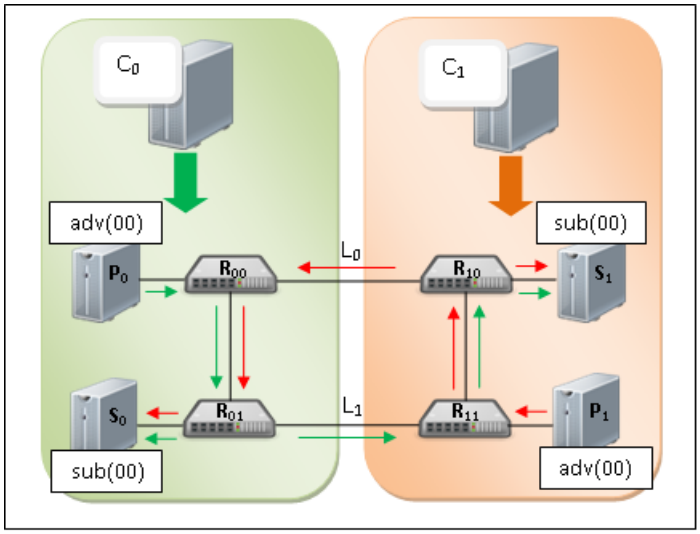



Figure 5.7: General Peer-to-Peer Architecture

hosts within a sub-network. However, there may be multiple paths between controllers (sub-networks) and a spanning tree must be created covering every controller (sub-network) exactly once. Such a configuration was necessary because of the following problems, illustrated in Fig. 5.7, encountered while designing a general peer-to-peer system. The figure depicts two sub-networks in the green and orange sections assigned to controllers $C_0$ and $C_1$ respectively. Let us assume that $P_0$ and $P_1$ advertise for the sub-space corresponding to $\{00\}$ and $S_0$ and $S_1$ subscribe for the same. So, ideally, paths should be created from $P_0$ to $S_0$, $S_1$ and from $P_1$ to $S_0$, $S_1$. The figure shows two links $L_0$ and $L_1$ connecting the two sub-networks. The path from $P_0$ is established across $L_1$ and is depicted with green arrows. Again, the path from $P_1$ is established across $L_0$ and is depicted with red arrows. It should be noted that as both the paths correspond to dz $\{00\}$, the destination IP match field in all the flows is 255.128.0.0/11. Now, let us assume that one of the publishers, say $P_0$, publishes an event satisfying the sub-space $\{00\}$, i.e., the event matches destination IP 255.128.0.0/11 of the flows. This packet is first sent to $R_{00}$ which forwards it to $R_{01}$. $R_{01}$ forwards it to $S_0$ as well as $R_{11}$ which again forwards the packet to $R_{10}$ according to the matching flow. However, at $R_{10}$ where on one hand the packet is delivered correctly to $S_1$, on the other hand it is again forwarded to $R_{00}$ as it also matches the path established for $P_1$. So, the packet now following the red arrow gets forwarded by $R_{00}$ and the cycle continues. This results in the same packet being continuously delivered to the subscribers and moving around the network in cycles leading to unnecessary bandwidth usage.

Quite naturally, cycles in a network are not desirable and must be avoided. This can be done by having special algorithms for cycle detection and avoidance. However, this thesis creates acyclic paths between the sub-networks by using a very simple mechanism based on typical distributed algorithms. The spanning tree creation process is briefly explained with a simple example depicted in Fig. 5.8. A particular controller may be chosen as the root which

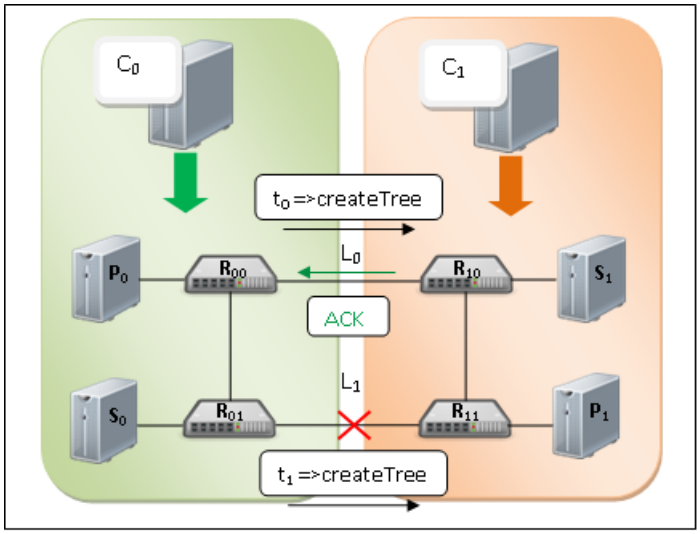



Figure 5.8: Acyclic Peer-to-Peer Architecture

initiates the process of creating the tree. The idea is that except for the root node, each node should have exactly one link connected to exactly one parent node. The root, in this case $C_0$, sends messages depicted in the figure as *createTree* through all its border switch-port tuples. Let us assume that $C_1$ receives the message sent across link $L_0$ at time $t_0$. As $C_1$ does not have a parent node yet, it immediately recognizes this link as its parent link and replies back with an ACK message along the reverse path. It also forwards the same message through its remaining border switch-port tuples which ensures that the process is repeated at every controller. On receiving the ACK message $C_0$ notes the switch-port tuple connected to $L_0$ as a valid gateway and remembers it as a child link. When the second *createTree* message arrives at $C_1$ at time $t_1$ along link $L_1$, $C_1$ simply ignores it as it has already identified its parent link. Also, as no acknowledgement for $L_1$ arrives at $C_0$, it is never added as a child link. The parent and child links added at each controller provide the final list of switch-port tuples to be used as gateways to the outside network in the acyclic setting. So, in this example, $R_{00}$ is the only gateway for $C_0$ and $R_{01}$ is the only gateway for $C_1$ which results in an acyclic architecture.

The concepts discussed above provide a general idea about the design of the system. However, the detailed algorithms for handling advertisement, subscription, un-advertisement and un-subscription requests are presented in the following sections.

## 5.3 Advertisement Handling

The advertisement handling logic aims to tackle problem 1 which deals with connectivity. In order to do so, a controller in a subnet needs to be aware of a publisher in another subnet and view it as a virtual publisher connected to its border gateway as explained before. Each controller can then establish paths from the publisher to all its interested subscribers resulting in

connectivity between the publisher and all its relevant subscribers. So, the present design and implementation is based on advertisement flooding across all controllers. This implies that a local advertisement request received by a controller has to be flooded across all sub-networks such that at any point of time active advertisements from all publishers in the system are known by every controller. So, each advertisement request should be propagated along the chain of controllers. However, this approach does not seem to be bandwidth efficient. So, in order to tackle problem 3, instead of flooding every advertisement request, this thesis follows a covering-based routing approach where an advertisement is flooded to the neighboring controllers only if it is not covered by an advertisement sent previously by the same controller. In order to implement covering-based routing, a controller maintains a local publisher count (*localPublisherCounts*) corresponding to each locally advertised dz and a list of remote publisher requests (*remotePublishers*) from neighboring controllers as mentioned before. Depending on the values of both of these, local/remote publisher requests are shared with other controllers. For every type of request sent by a host (publisher/subscriber), the controller first identifies the host as local or remote (virtual) and accordingly handles the request. So, with the arrival of an advertisement request, the controller first categorizes its sender as local or remote from the type of received packet. Depending on the type of request, the controller performs a set of actions. However, in both types of request, a decision about sharing the advertisement with remote controllers has to be taken. Four scenarios may arise in this context as follows:

1. The new advertisement is not equal to or covered by a previously sent local or remote advertisement.

2. The new advertisement is equal to or covered by one or more previously sent local advertisement/s.

3. The new advertisement is equal to or covered by a previously received advertisement from a(another) remote neighboring controller.

4. The new advertisement is equal to or covered by more than one previously received advertisements from more than one (other) neighboring remote controllers.

Both request types handle the above situations similarly. The exact operations performed for handling both types of requests are explained below.

### 5.3.1 Local Advertisements

If the request is local, the local publisher access point (node-port tuple to which it is connected) and advertised dz are duly noted and processed as per the centralized advertisement handling algorithm. This means that one or more local pub/sub trees are updated and flows are established between this publisher and already existing relevant local and virtual subscribers in the local subnet. Following this, a decision to share the advertisement request is taken based on the above four scenarios so that paths can be established between sub-networks. Each of these scenarios can be explained with examples.

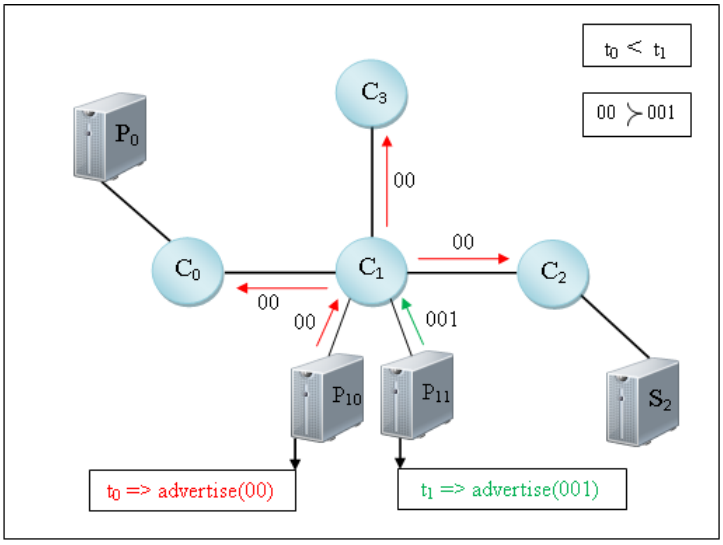



Figure 5.9: Example of local advertisements within the same sub-network

**Scenarios 1 and 2** Fig. 5.9 depicts a system with four sub-networks represented by the controllers they are assigned to. Every case is explained with respect to controller $C_1$. At time $t_0$, a publisher $P_{10}$ sends a local advertisement request for dz $\{00\}$. This corresponds to the first scenario where no other advertisement request has been previously received by the local controller $C_1$. Under such circumstances, the controller creates a controller advertisement packet with appropriate type and received dz which is sent through all border switch-port (node-port) tuples of the sub-network. The second scenario is depicted at time $t_1$ when $P_{11}$ sends an advertisement request for dz $\{001\}$. Since dz $\{00\}$ has already been advertised and $\{00\} \succ \{001\}$, therefore, the new request is not forwarded to the neighboring controllers. The idea behind this approach is that if there is a subscriber, say $S_2$ in this figure interested in the event sub-space $\{001\}$, it is adequate for controller $C_2$ to be aware of the previously advertised higher dz from the virtual publisher representing sub-network 1. Since from $C_2$'s perspective both the advertisement requests come from the same virtual publisher, only the highest dz covering all others should anyhow be considered for establishment of flows in the switches. So, the request $\{001\}$ is redundant information for $C_2$ and does not affect any flows in sub-network 2. As a result, information sharing between controllers can safely be filtered using the covering-based approach without compromising correctness in terms of increasing false negatives and false positives in the system.

**Scenario 3** The third scenario is portrayed in Fig. 5.10 where again a similar network is considered with focus on controller $C_1$. This case deals with the arrival of an advertisement $\{001\}$ from a local publisher $P_1$ at time $t_1$ after the arrival of a remote advertisement request from controller $C_0$ at $t_0$. Under these circumstances, where a single remote publisher with the same or higher dz exists, $C_1$ again creates a controller advertisement for the recent request and sends it to only the remote controller from which the previous request had arrived, i.e. $C_0$ in this example. This is done because the remote request from $C_0$ was already forwarded

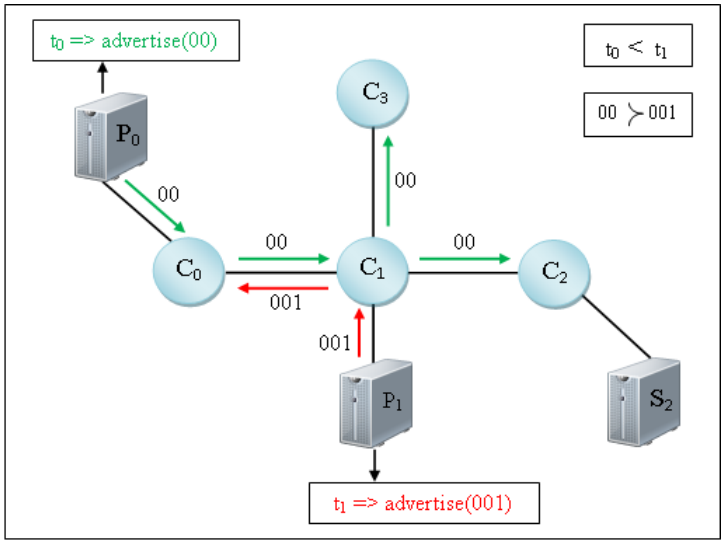



Figure 5.10: Example of local advertisement with existing remote request

by $C_1$ earlier making it a virtual publisher to all except $C_1$ which was the one to have sent the request in the first place. But on arrival of the current request, it is necessary for $C_1$ to also view this subnet as a virtual publisher. The *remotePublishers* list is used to monitor all previous remote controller requests.

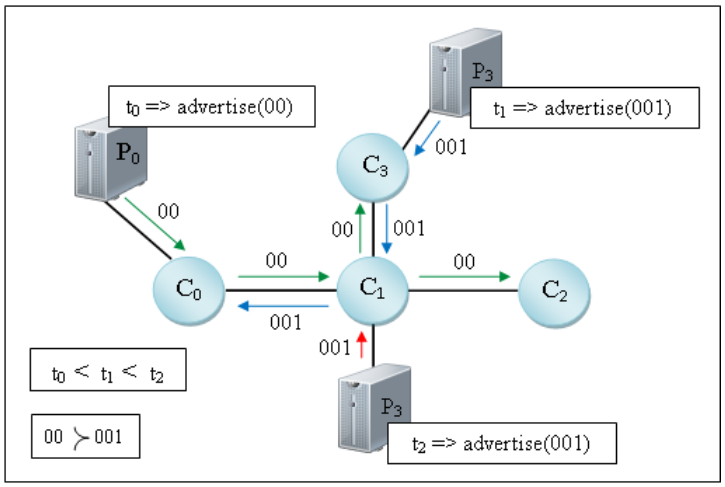



Figure 5.11: Example of local advertisement with existing remote requests

**Scenario 4** The final scenario is portrayed in Fig. 5.11 where at time $t_0$ and $t_1$ two remote controller advertisement requests arrive at $C_1$. Following these, $P_0$ sends an advertisement $\{001\}$ at $t_2$ which is equal to or covered by the previous advertisements. Under these circumstances, no further messages are sent to the neighboring controllers. So, $C_1$ simply adds the local publisher to its local spanning tree using the centralized advertisement handling

**Algorithm 1** Local Advertisement Handling

1: **procedure** *addLocalPublisher*(dzExp, pub, receivedPacket)
2: addPublisher($dzExp, pub$) {⇒ add this publisher to the corresponding local pub/sub tree for flow addition}
3: $localPubCount \leftarrow getLocalPublisherCount(localPublisherCounts, dzExp)$ {⇒ gets local publisher count for all dzs where dz $\succeq$ dzExp}
4: $rmPubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp\}$
5: **if** $localPubCount = 0 \ \wedge \ \mid rmPubs \mid \ = 0$ **then**
6: $adPacketData \leftarrow createControllerPacket(receivedPacket)$
7: **for each** $npTuple \in borderGateways$ **do**
8: createAndSendPacket($adPacketData, npTuple$) {⇒if no local/remote publisher exists, send controller ad packet through all switch port tuples in borderGateways}
9: **end for**
10: **else if** $localPubCount = 0 \ \wedge \ \mid rmPubs \mid \ = 1$ **then**
11: $rmPub \leftarrow \{rmPubs : \mid rmPubs \mid \ = 1\}$
12: $adPacketData \leftarrow createControllerPacket(receivedPacket)$
13: createAndSendPacket($adPacketData, rmPub_{nodePortTuple}$) {if only one remote publisher exists, send advertisement packet to it}
14: **end if**
15: incrementPublisherCount($localPublisherCounts, dzExp$)
16: **end procedure**

algorithm. This decision follows the same logic described above where the remote publisher requests were already forwarded by $C_1$ to concerned neighboring controllers and so there is no need to send redundant information on arrival of the local request.

In all four scenarios, however, along with adding the publisher to the local pub/sub tree, *localPublisherCounts* is incremented for the received advertisement dz. The algorithm for handling advertisements from local publishers is more formally presented in Algo. 1 as a pseudo-code where the arguments to the procedure are the dz-expression, denoted by *dzExp*, the switch-port tuple of the local network to which the local publisher is attached, denoted by *pub* and finally the packet received from the local publisher, denoted by *receivedPacket*.

### 5.3.2 Remote Advertisements

A remote publisher request is handled in a way similar to the local advertisement handling process and is formally presented in Algo. 2. As mentioned earlier, an advertisement request from a remote controller is perceived as a request from a virtual publisher connected to the border switch-port tuple connecting the two concerned sub-networks. So, on receiving a request from a neighboring controller, the virtual publisher access point (node-port tuple to which it is connected) and advertised dz are noted. At first, the *remotePublishers* list is

**Algorithm 2** Remote Advertisement Handling

1: **procedure** *addRemotePublisher*(dzExp, pub, receivedPacket)
2: $coveredDzs \leftarrow \{dz_{remotePublishers} : dzExp \succ dz_{remotePublishers} \ \wedge$
$nodePortTuple_{remotePublishers} = pub\}$
3: **for each** $dz \in coveredDzs$ **do**
4: removePublisher($dz, pub$) {⇒ remove this publisher for the corresponding dz from pub-/sub tree/s}
5: removeRemotePublisher($remotePublishers, dz, pub$)
6: **end for**
7: addPublisher($dzExp, pub$) {⇒ add this publisher to the corresponding pub/sub tree for flow addition}
8: $localPubCount \leftarrow getLocalPublisherCount(localPublisherCounts, dzExp)$ {⇒ gets local publisher count for all dzs where dz $\succeq$ dzExp}
9: $rmPubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp\}$
10: **if** $localPubCount = 0 \ \wedge \ \mid rmPubs \mid \ = 0$ **then**
11: **for each** $npTuple \in borderGateways$ **do**
12: **if** $npTuple \neq pub$ **then**
13: forwardPacket($receivedPacket, npTuple$) {⇒if no local or remote publisher exists, forward received packet}
14: **end if**
15: **end for**
16: **else if** $localPubCount = 0 \ \wedge \ \mid rmPubs \mid \ = 1$ **then**
17: $rmPub \leftarrow \{rmPubs : \mid rmPubs \mid \ = 1\}$
18: **if** $rmPub_{nodePortTuple} \neq pub$ **then**
19: forwardPacket($receivedPacket, rmPub_{nodePortTuple}$) {⇒ if only one other remote publisher exists forward the packet to it}
20: **end if**
21: **end if**
22: addRemotePublisher($remotePublishers, dzExp, pub$)
23: **if** $\mid coveredDzs \mid \ = 0$ **then**
24: $subscriberList \leftarrow \{subscribers : dz_{subscribers} \succeq dzExp \ \vee \ dzExp \succ dz_{subscribers}\}$
25: $subs \leftarrow$getSubsWithMaxDzs($subscriberList$) {get the highest subscription dz(s) which cover all others}
26: **for each** $subscriber \in subs$ **do**
27: $subPacketData \leftarrow createControllerPacket(dz_{subscriber})$
28: createAndSendPacket($subPacketData, pub$) {⇒ send the created sub packet through the switch port from which ad packet came}
29: **end for**
30: **end if**
31: **end procedure**

checked for any advertisements received previously from the same virtual publisher that has a dz which is covered by the current dz. If such dzs exist, then a local un-advertisement procedure for each of them is done as per the centralized approach along with the removal of the advertisements from the *remotePublishers* list. This is done to maintain only the highest covering advertisement dzs from a particular virtual publisher (remote sub-network). Then, the newly advertised dz from the virtual publisher is processed as per the centralized advertisement handling algorithm by creating/updating relevant spanning trees. Then, as before, a decision about sharing the advertisement request with remote controllers is taken depending on the same four criteria discussed above. Again, as in Scenario 2, if a local advertised dz higher than or same as the current dz exists, then the remote controller advertisement is not forwarded any further by the current controller. Scenario 4 is handled in the same way if the newly advertised dz is equal to or covered by more than one previously received advertised dzs from two or more other neighboring remote controllers. However, if only another remote controller exists which has sent a relevant dz previously as stated in Scenario 3, then the newly received advertisement request is forwarded to it. Finally, if Scenario 1 occurs and none of the previous three cases are satisfied, then it is forwarded to all neighboring controllers except for the controller which has sent the current remote request.

Also, the *remotePublishers* list is updated with the current virtual publisher advertisement. Finally, the list of local subscriptions in the local network relevant to the arrived advertisement is identified and subscription requests with the highest dzs which cover all others are sent by the controller along the switch-port tuple through which the remote advertisement request arrived. This is portrayed in line 28 of Algo. 2. This further results in subscription handling by the adjoining controller to which the subscription request is sent which is discussed in details in the next section. It is necessary to send the subscription back to the remote controller so that a path can be established between the original publisher and this subnet. It should be noted that these new subscriptions are sent to the remote controller from which the advertisement request arrived if and only if they have not already been sent to the same controller with respect to a previously sent relevant advertisement. This check is carried out in line 23 of Algo. 2.

## 5.4 Subscription Handling

In order to achieve connectivity, subscriptions should also be shared between controllers. However, finding a solution which is bandwidth efficient is also a requirement mentioned in 3. So, subscriptions are not flooded throughout the network. Instead, a subscription is sent to a neighboring controller if and only if an advertisement relevant to the subscription has previously arrived from it. So, a subscription follows a reverse path previously taken by a relevant advertisement. Since the design follows a covering-based routing approach, a subscription is sent to the neighboring controllers with relevant publishers only if it is not covered by a previously sent subscription which further improves bandwidth efficiency. In order to implement covering-based routing, a controller maintains a local subscriber count (*localSubscriberCounts*) corresponding to each locally subscribed dz and a list of remote subscriber

requests (*remoteSubscribers*) from neighboring controllers as stated before. Depending on the values of both of these and the existence of relevant publishers in adjoining sub-networks, local subscriber requests are shared with other controllers. Again, four scenarios may arise in this context as follows :

1. The new subscription is not equal to or covered by a previously received local or remote subscription.
2. The new subscription is equal to or covered by one or more previously sent local subscription/s.
3. The new subscription is equal to or covered by a previously received subscription from a(another) remote neighboring controller.
4. The new subscription is equal to or covered by more than one previously received subscriptions from more than one (other) neighboring remote controllers.

With the arrival of a subscription request, the controller first categorizes its sender as local or remote from the type of received packet and based on the type, a set of actions are taken as explained below.

### 5.4.1 Local Subscriptions

If the request is local, the local subscriber access point (node-port tuple to which it is connected) and subscribed dz are duly noted and processed as per the centralized subscription handling algorithm to establish routes between the subscriber and relevant local publishers which include virtual publishers. Following this, a decision about sharing the subscription request with remote controllers is taken.

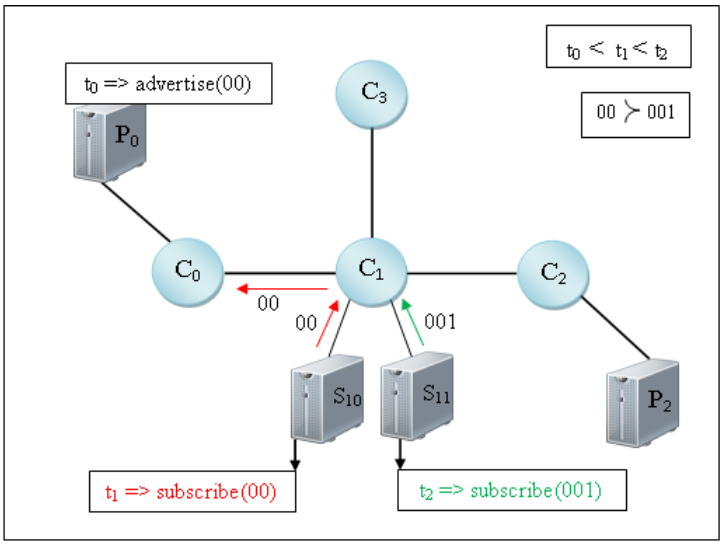



Figure 5.12: Example of local subscriptions within the same sub-network

**Scenarios 1 and 2** Fig. 5.12 depicts a system with four sub-networks represented by the controllers they are assigned to. Every case is explained with respect to controller $C_1$. At time $t_0$, a publisher $P_0$ sends an advertisement request for dz $\{00\}$ which is shared among all controllers using the algorithms discussed above. Then, at time $t_1$ a subscriber $S_{10}$ subscribes for the dz space $\{00\}$ which corresponds to the first scenario where no other subscription request has been received by the local controller $C_1$. Under such circumstances, first, all the remote publishers are identified which are relevant to this subscription as subscriptions are not flooded to every controller in the network. Next, a controller subscription packet is created with appropriate type and received dz and is sent along the border switch-port tuples connecting the identified remote controllers with relevant publishers only. So, in the figure, $C_1$ sends the subscription request to $C_0$ alone as it had previously received a relevant advertisement from it. The second scenario is depicted at time $t_2$ when $S_{11}$ sends a subscription request for dz $\{001\}$. Since dz $\{00\}$ has already been sent and $\{00\} \succ \{001\}$, therefore the new request is not forwarded to $C_0$. Again, the logic behind this is that the arrival of the former subscription at $C_0$ resulted in a path being established between $P_0$ and the border switch-port tuple connecting sub-network 0 to sub-network 1. Now, the arrival of $\{001\}$ at $C_0$ would not make any difference to the flows on the switches of sub-network 0 and is redundant information.

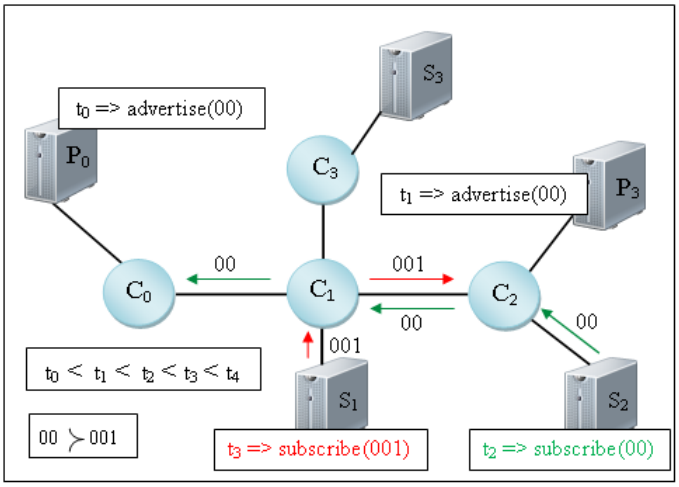



Figure 5.13: Example of local subscription with existing remote request

**Scenario 3** The third scenario is portrayed in Fig. 5.13 where again a similar network is considered with focus on controller $C_1$. It is assumed that $P_0$ and $P_3$ have advertised for dz $\{00\}$ at time $t_0$ and $t_1$ respectively. So, $C_1$ is aware of both of these advertisements. At time $t_2$, $S_2$ subscribes for sub-space $\{00\}$ which arrives as a remote subscriber request at $C_1$. The handling of this remote request is explained later. This case deals with the arrival of subscription $\{001\}$ from a local subscriber $S_1$ at time $t_3$. Under these circumstances, where a single remote subscriber with the same or a higher dz exists, the controller $C_1$ checks if the remote sub-network that is perceived as a virtual subscriber also acts as a virtual publisher relevant to the subscription. In this figure, $C_1$ views sub-network 2 as both a

virtual publisher and a virtual subscriber as both types of requests have arrived from the border switch-port tuple connecting the two sub-networks. In such a scenario, where the same virtual host poses as both a publisher and a subscriber for relevant sub-spaces, the controller creates a subscription for the recent request and sends it to that remote controller, i.e. $C_2$ in this example. The *remoteSubscribers* list is used to monitor all previous remote controller subscription requests.

**Algorithm 3** Local Subscription Handling

1: **procedure** *addLocalSubscriber*(dzExp, sub, receivedPacket)
2: addSubscriber($dzExp, sub$) {⇒ add this subscriber to the corresponding pub/sub tree for flow addition}
3: $localSubCount \leftarrow getLocalSubscriberCount(localSubscriberCounts, dzExp)$ {⇒ gets local subscriber count for all dzs where dz ⪰ dzExp}
4: $rmSubs \leftarrow \{remoteSubscribers : dz_{remoteSubscribers} \succeq dzExp\}$
5: **if** $localSubCount = 0 \ \wedge \ \mid rmSubs \mid \ = 0$ **then**
6: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \ \vee \ dzExp \succ dz_{remotePublishers}\}$
7: $subPacketData \leftarrow createControllerPacket(receivedPacket)$
8: **for each** $remotePublisher \in pubs$ **do**
9: createAndSend($subPacketData, nodePortTuple_{remotePublisher}$) {⇒ if no local/remote subscribers exist send controller packet to all controllers with relevant publishers}
10: **end for**
11: **else if** $localSubCount = 0 \ \wedge \ \mid rmSubs \mid \ = 1$ **then**
12: $rmSub \leftarrow \{rmSubs : \mid rmSubs \mid \ = 1\}$
13: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \ \vee \ dzExp \succ dz_{remotePublishers}\}$
14: **if** $rmSub \in pubs$ **then**
15: $subPacketData \leftarrow createControllerPacket(receivedPacket)$
16: createAndSend($subPacketData, nodePortTuple_{rmSub}$) {if only one remote subscriber exists and it is also a remote publisher create and send controller packet}
17: **end if**
18: **end if**
19: incrementSubscriberCount($localSubscriberCounts, dzExp$)
20: **end procedure**

**Scenario 4** The final scenario is portrayed in Fig. 5.14 where at time $t_0$ and $t_1$ two remote controller advertisement requests arrive at $C_1$. Following these, two remote controller subscription requests arrive at $t_2$ and $t_3$ with dzs {00} and {001} respectively. Finally at time $t_4$, a local subscription with dz {001} arrives at $C_1$ from $S_1$. Under these circumstances, no further subscription messages are sent to the neighboring controllers because any relevant remote controller already views this controller as a virtual subscriber because of the remote subscriptions forwarded earlier. The figure clearly explains this fact. So, $C_1$ simply adds

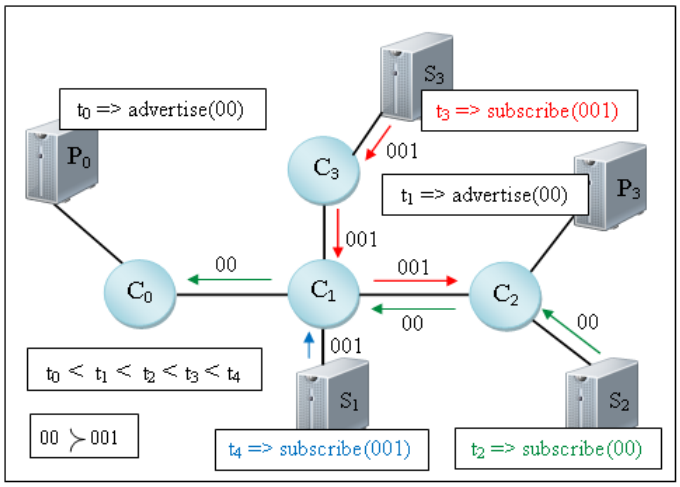



Figure 5.14: Example of local subscription with existing remote requests

the local subscriber to its local pub/sub tree/s using the centralized subscription handling algorithm.

Again, in all four scenarios *localSubscriberCounts* is incremented for the received subscription dz. The algorithm for handling subscriptions from local subscribers is more formally presented in Algo. 3 as a pseudo-code where the arguments to the procedure are the dz-expression, denoted by *dzExp*, the switch-port tuple of the local network to which the local subscriber is attached, denoted by *sub* and finally the packet received from the subscriber, denoted by *receivedPacket*.

### 5.4.2 Remote Subscriptions

A remote subscriber request is handled in a way similar to the local subscription handling procedure and is formally presented in Algo. 4. As mentioned earlier, a subscription request from a remote controller is perceived as a request from a virtual subscriber connected to the border switch-port tuple connecting the two concerned sub-networks. So, on receiving a request from a neighboring controller, the virtual subscriber access point (node-port tuple to which it is connected) and subscribed dz are noted.

At first, the *remoteSubscribers* list is checked for any subscriptions received previously from the same virtual subscriber that has a dz which is covered by the current dz. If such dzs exist, then a local un-subscription process for each of them is done as per the centralized approach along with the removal of these subscriptions from the *remoteSubscribers* list. This is done to maintain only the highest covering subscription dzs from a particular virtual subscriber (remote sub-network). Then, the newly subscribed dz from the virtual subscriber (remote sub-network) is processed as per the centralized advertisement handling algorithm by adding

it to relevant spanning trees and updating flows. Then, as before, a decision about sharing the subscription request with remote controllers is taken depending on the same four criteria discussed above.

**Algorithm 4** Remote Subscription Handling

1: **procedure** *addRemoteSubscriber*(dzExp, sub, receivedPacket)
2: $coveredDzs \leftarrow \{dz_{remoteSubscribers} : dzExp \succ dz_{remoteSubscribers} \ \wedge$
$nodePortTuple_{remoteSubscribers} = nodePortTuple\}$
3: **for each** $dz \in coveredDzs$ **do**
4: removeSubscriber($dz, sub$) {⇒ remove subscriber for corresponding dz from the pub/sub tree/s}
5: removeRemoteSubscriber($remoteSubscribers, dz, sub$)
6: **end for**
7: addSubscriber($dzExp, sub$) {⇒ add this subscriber to the corresponding pub/sub tree for flow addition}
8: $localSubCount \leftarrow getLocalSubscriberCount(localSubscriberCounts, dzExp)$ {⇒ gets local subscriber count for all dzs where dz $\succeq$ dzExp}
9: $rmSubs \leftarrow \{remoteSublishers : dz_{remoteSublishers} \succeq dzExp\}$
10: **if** $localSubCount = 0 \ \wedge \ \mid rmSubs \mid \ = 0$ **then**
11: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \ \vee \ dzExp \succ dz_{remotePublishers}\}$
12: **for each** $remotePublisher \in pubs$ **do**
13: **if** $nodePortTuple_{remotePublisher} \neq sub$ **then**
14: forward($receivedPacket, nodePortTuple_{remotePublisher}$)
15: **end if**
16: **end for**
17: **else if** $localSubCount = 0 \ \wedge \ \mid rmSubs \mid \ = 1$ **then**
18: $rmSub \leftarrow \{rmSubs : \mid rmSubs \mid \ = 1\}$
19: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \ \vee \ dzExp \succ dz_{remotePublishers}\}$
20: **if** $rmSub \in pubs \ \wedge \ nodePortTuple_{rmSubs} \neq sub$ **then**
21: forward($subPacketData, nodePortTuple_{rmSub}$)
22: **end if**
23: **end if**
24: addRemoteSubscriber($remoteSubscribers, dzExp, sub$)
25: **end procedure**

Again, if scenario 2 occurs, the new subscription is not forwarded any further by the current controller. Scenario 4 is handled in the same way. However, if only another remote controller exists which has sent a relevant subscription previously, i.e., scenario 3, then, the newly received subscription request is forwarded to it if it has a relevant publisher in its local network. Finally, if scenario 1 occurs where no relevant local or remote subscriptions have been previously received, then the request is forwarded to every neighboring controller with

relevant publishers except for the controller from which this request arrived. Also, in each case, the *remoteSubscribers* list is updated with the current virtual subscriber subscription.

## 5.5 Un-Advertisement Handling

An un-advertisement request submitted by a publisher should be communicated to every controller that has been sent the corresponding advertisement so that the effect of the advertisement can be nullified completely throughout the entire network. So, ideally, each un-advertisement request should be routed along the same paths as its corresponding advertisement message which seems quite simple. However, the procedure gets slightly complicated due to the use of covering-based routing for control messages. As all submitted advertisements are not handled in the same way, similarly, all un-advertisements cannot be handled in the same way. Similar to the previous algorithms, here too, first the type of un-advertisement request is identified as local or remote. Yet again, four scenarios are encountered during the phase deciding on the sharing of the un-advertisement request as follows:

1. The un-advertisement dz is not equal to or covered by a previously sent local or remote advertisement dz from another publisher.

2. The un-advertisement dz is equal to or covered by one or more previously sent advertisement dzs from other local publisher/s.

3. The un-advertisement dz is equal to or covered by a previously received advertisement dz from a(another) remote neighboring controller.

4. The un-advertisement dz is equal to or covered by more than one previously received advertisement dzs from more than one (other) neighboring remote controllers.

These cases are similar to the ones explored during advertisement handling and follow similar course.

### 5.5.1 Local Un-advertisements

If the request is local, the local publisher access point and dz are noted by the local controller. These are then used in the centralized un-advertisement handling algorithm in the local network to remove/downgrade flows in the local switches. After this, a decision to share the un-advertisement information with the neighboring controllers is taken as per the four scenarios.

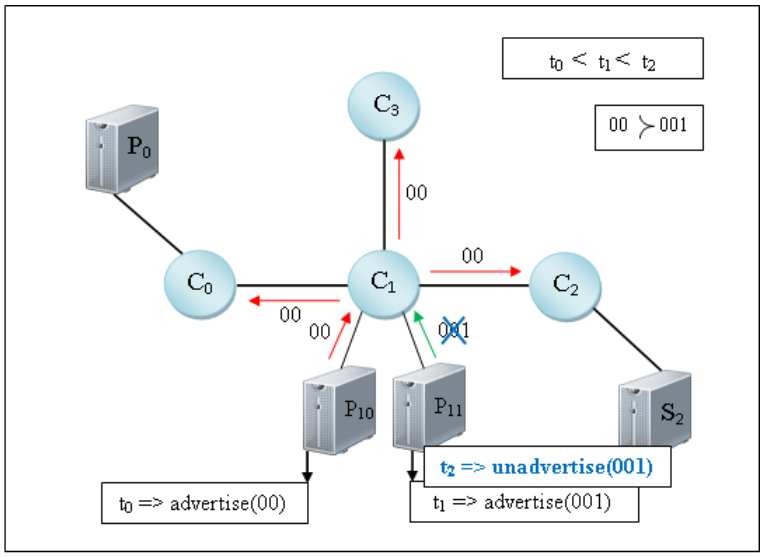



Figure 5.15: Example of local un-advertisement with other existing publishers

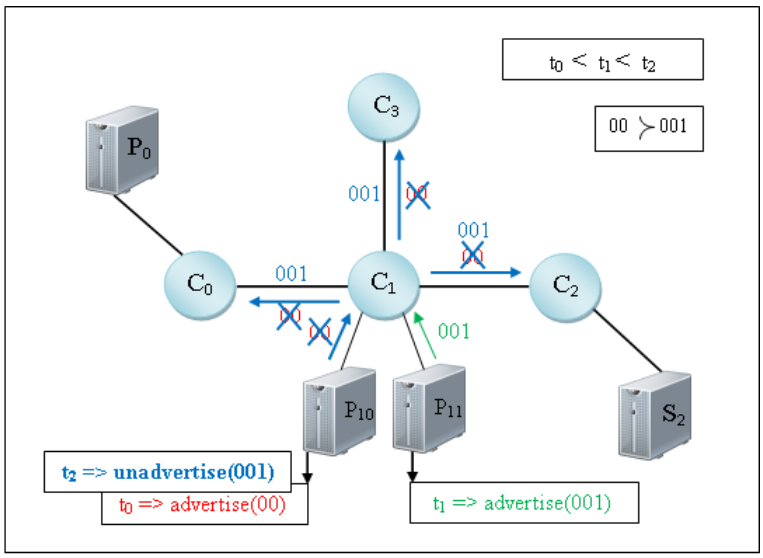



Figure 5.16: Example of controller advertisement on local publisher un-advertisement

**Scenarios 2 and 4** So, as in advertisement handling, the second and the last cases are handled in the same way where no un-advertisement requests are sent to the remote controllers. The reason behind this is that another local publisher or atleast 2 remote publishers still exist with an equal or higher advertisement sub-space for which the neighboring controllers should continue viewing the current sub-network as a publisher of the equivalent or a higher dz. Fig. 5.15 depicts a very simple example which extends an example portrayed in advertisement handling by introducing an un-advertisement request by $P_{11}$ at time $t_2$. Due to the existing active advertisement of $P_{10}$, this un-advertisement request is not forwarded to the other controllers. However, at $C_1$, the local pub/sub trees relevant to this request are modified in order to cancel the advertisement from $P_{11}$.

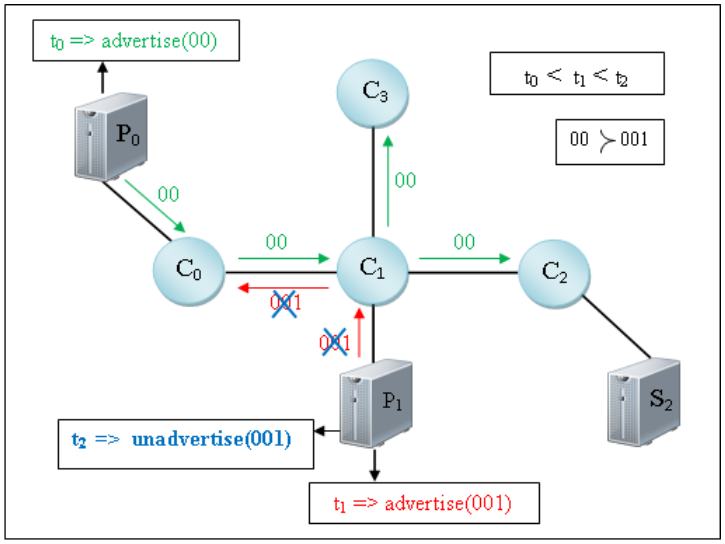



Figure 5.17: Example of local un-advertisement with an existing relevant remote publisher

**Scenario 1** The first case deals with the existence of no other publisher with the same or higher advertisement sub-space. Under such circumstances, an un-advertisement controller request is sent to all other controllers to cancel out the effect of the corresponding advertisement that was previously sent to every controller. However, just un-advertising is not enough because advertisements were shared according to the covering-based approach. So, to inform all controllers about the advertisements that were not sent earlier due to covering by the advertisement that is now cancelled, new advertisement requests are sent by the local controller. In order to identify the existing advertisement requests, the local controller uses the *localPublisherCounts* and *remotePublishers* lists. Once the highest dzs are selected, the new advertisements are distributed to remote controllers using the same logic used during advertisement handling for deciding on which advertisement to send to which remote controller. Algo. 5 presents the pseudo-code for local un-advertisement handling where the function *sendAdvertisementRequest()* in line 10 takes care of this. This situation is further explained in Fig. 5.16 where two advertisements are sent to $C_1$ at $t_0$ and $t_1$ as depicted in the figure. Then at time $t_2$, $P_{10}$ un-advertises resulting in the flooding of the un-advertisement message to

all neighboring controllers followed by an advertisement message representing {001} to inform the remote controllers of the advertisement sent by $P_{11}$ which was earlier filtered out due to covering-based routing. Even in Algo. 5, three arguments are provided to the procedure *removeLocalPublisher()*, namely, *dzExp*, *pub* (switch-port tuple representing the publisher that sent the request) and *receivedPacket*.

**Algorithm 5** Local Un-Advertisement Handling

```
1:  procedure removeLocalPublisher(dzExp, pub, receivedPacket)
2:  removePublisher(dzExp, pub) {⇒ remove this publisher from the corresponding pub/sub
    tree for flow modification}
3:  localPubCount ← getLocalPublisherCount(localPublisherCounts, dzExp) {⇒ gets local
    publisher count for all dzs where dz ⪰ dzExp}
4:  rmPubs ← {remotePublishers : dz_remotePublishers ⪰ dzExp}
5:  if localPubCount = 1 ∧ | rmPubs | = 0 then
6:     unadPacketData ← createControllerPacket(receivedPacket)
7:     maxDzPubs ← getNextMaxDzPubs(dzExp) {⇒ get pubs with next highest dzs}
8:     for each npTuple ∈ borderGateways do
9:        createAndSendPacket(unadPacketData, npTuple)
10:       sendAdvertisementRequest(maxDzPubs, npTuple)
11:    end for
12: else if localPubCount = 1 ∧ | rmPubs | = 1 then
13:    rmPub ← {rmPubs : | rmPubs | = 1}
14:    unadPacketData ← createControllerPacket(receivedPacket)
15:    createAndSendPacket(unadPacketData, rmPub_nodePortTuple)
16:    maxDzPubs ← getNextMaxDzPubs(dzExp) {⇒ get pubs with next highest dzs}
17:    sendAdvertisementRequest(maxDzPubs, rmPub_nodePortTuple)
18: end if
19: decrementLocalPublisherCount(localPublisherCounts, dzExp)
20: end procedure
```

**Scenario 3** The third scenario deals with the existence of just one other remote publisher with same or higher dz. Under these circumstances, the rest of the neighboring controllers need to still view the current controller as a publisher due to this other remote publisher. But this other remote controller must be informed about the current un-advertisement request so that it does not continue viewing the local controller as a publisher. A simple example of this case is illustrated in Fig. 5.17 where the un-advertisement from $P_1$ is sent only to controller $C_0$ with an existing advertisement {00}. Also, new advertisements may need to be sent to this remote controller as represented in line 17 of Algo. 5.

In all four scenarios, along with the modifications to the local pub/sub tree, the *localPublisherCounts* is also decremented for the un-advertised dz.

### 5.5.2 Remote Un-advertisements

The remote un-advertisement handling is not much different from the local one. As mentioned earlier, an advertisement request from a remote controller is perceived as a request from a virtual publisher connected to the border switch-port tuple connecting the two concerned sub-networks. So, on receiving an un-advertisement request from such a neighboring controller, the virtual publisher access point (node-port tuple to which it is connected) and advertised dz are noted. This is followed by centralized un-advertisement handling of the request from the virtual publisher resulting in pub/sub tree and flow modifications in the local network. Then the same decisions regarding spreading of the un-advertisement request are taken as in the local cases and can be understood from the pseudo-code presented in Algo. 6. Here, lines 10 and 17 take care of sending the new advertisements to relevant remote controllers in the function *sendAdvertisementRequest()*. However, care is taken to ensure that the controller from which the un-advertisement request has arrived (denoted by the switch-port tuple *pub*) is never in the recipients list for the new advertisements. Finally, the entry in *remotePublishers* corresponding to the un-advertised dz-expression and virtual publisher (remote controller) is removed.

**Algorithm 6** Remote Un-Advertisement Handling

1: **procedure** removeRemotePublisher($dzExp, pub, receivedPacket$)
2: removePublisher($dzExp, pub$) {⇒ remove this publisher from the corresponding pub/sub tree for flow modification}
3: $localPubCount \leftarrow$ getLocalPublisherCount($localPublisherCounts, dzExp$) {⇒ gets local publisher count for all dzs where dz $\succeq$ dzExp }
4: $rmPubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp\}$
5: **if** $localPubCount = 0 \ \wedge \ \mid rmPubs \mid \ = 1$ **then**
6: $maxDzPubs \leftarrow$ getNextMaxDzPubs($dzExp$) {⇒ get publishers with next highest dzs for advertisement}
7: **for each** $npTuple \in borderGateways$ **do**
8: **if** npTuple $\neq$ pub **then**
9: forward($receivedPacket, npTuple$) {⇒ forward received packet through all other switch-port tuples in borderGateways}
10: sendAdvertisementRequest($maxDzPubs, npTuple$)
11: **end if**
12: **end for**
13: **else if** $localPubCount = 0 \ \wedge \ \mid rmPubs \mid \ = 2$ **then**
14: $rmPub \leftarrow \{rmPubs : nodePortTuple_{rmPubs} \neq pub\}$
15: $forward(unadPacketData, rmPub_{nodePortTuple})$
16: $maxDzPubs \leftarrow$ getNextMaxDzPubs($dzExp$) {⇒ get publishers with next highest dzs for advertisement}
17: sendAdvertisementRequest($maxDzPubs, rmPub_{nodePortTuple}$)
18: **end if**
19: removeRemotePublisher($remotePublishers, dzExp, pub$)
20: **end procedure**

## 5.6 Un-Subscription Handling

Just like an un-advertisement, an un-subscription request submitted by a subscriber should also be communicated to every controller that has been sent the corresponding subscription previously so that the effect of the subscription can be cancelled throughout the entire network. So, ideally, each un-subscription request should be routed along the same paths as its corresponding subscription. Again, due to the use of covering-based routing for subscription messages, the process is not straightforward and all un-subscriptions cannot be handled in the same way. Again, the first step is to identify the type of un-subscription request as local or remote and take appropriate actions. Yet again, the decision to share the un-subscription with other controllers depends on four scenarios as follows:

1. The un-subscription dz is not equal to or covered by a previously sent local or remote subscription dz from another subscriber.

2. The un-subscription dz is equal to or covered by one or more previously sent subscription dzs from other local subscriber/s.

3. The un-subscription dz is equal to or covered by a previously received subscription dz from only one (other) remote neighboring controller.

4. The un-subscription dz is equal to or covered by more than one previously received subscription dzs from more than one (other) neighboring remote controllers.

### 5.6.1 Local Un-subscriptions

If the request is local, the local subscriber access point and dz are used in the centralized un-subscription handling algorithm in the local network to remove/downgrade flows in the local switches. Next, a decision to share the un-subscription information with the neighboring controllers is taken depending on the above scenarios. The local un-subscription sharing decision is very similar to the one encountered in the local un-advertisement algorithm and can be understood from the pseudo-code presented in Algo. 7. Decisions are taken based on the *localSubscriberCounts* and *remoteSubscribers* lists.

Again, cases 2 and 4 result in no information sharing. Scenario 1 is taken care of by sending the un-subscription request to all remote controllers having publishers with advertisements relevant to the previously sent subscription request. This extra filtering is done because subscription flooding was avoided. Also, just as covered highest advertisements are sent in un-advertisement handling, similarly covered highest subscriptions are sent by the controller to the remote controllers which received the un-subscription request. The function *sendSubscriptionRequest()* in line 11 of Algo. 7 takes care of this.

Finally, scenario 3 deals with an existing remote controller which had earlier sent a relevant subscription. Under these circumstances, the rest of the neighboring controllers need to still view the current controller as a subscriber due to this other remote subscriber. But this other remote controller must be informed about the current un-subscription request so that it does not continue viewing the local controller as a subscriber provided it had been sent the corresponding subscription request previously, i.e., it has a relevant publisher in its subnet. After addressing the information sharing issue, *localSubscriberCounts* is decremented for the received un-subscription dz.

**Algorithm 7** Local Un-Subscription Handling

1: **procedure** *removeLocalSubscriber*(dzExp, sub, receivedPacket)
2: removeSubscriber($dzExp, sub$) {⇒ remove this subscriber from the corresponding pub/-sub tree for flow modification}
3: $localSubCount \leftarrow$ getLocalSubscriberCount($localSubscriberCounts, dzExp$) {⇒ gets local subscriber count for all dzs where dz $\succeq$ dzExp}
4: $rmSubs \leftarrow \{remoteSubscribers : dz_{remoteSubscribers} \succeq dzExp\}$
5: **if** $localSubCount = 1 \ \wedge \ \mid rmSubs \mid \ = 0$ **then**
6: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \ \vee dzExp \succ dz_{remotePublishers}\}$
7: $maxDzSubs \leftarrow$ getNextMaxDzSubs($dzExp$) {⇒ get subscribers with next highest dzs}
8: $unsubPacketData \leftarrow createControllerPacket(receivedPacket)$
9: **for each** $remotePublisher \in pubs$ **do**
10: $createAndSend(unsubPacketData, nodePortTuple_{remotePublisher})$
11: sendSubscriptionRequest($maxDzSubs, nodePortTuple_{remotePublisher}$)
12: **end for**
13: **else if** $localSubCount = 1 \ \wedge \ \mid rmSubs \mid \ = 1$ **then**
14: $rmSub \leftarrow \{rmSubs : \mid rmSubs \mid \ = 1\}$
15: $maxDzSubs \leftarrow$ getNextMaxDzSubs($dzExp$) {⇒ get subscribers with next highest dzs}
16: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \vee dzExp \succ dz_{remotePublishers}\}$
17: $unsubPacketData \leftarrow createControllerPacket(receivedPacket)$
18: **if** $rmSub \in pubs$ **then**
19: $createAndSend(unsubPacketData, nodePortTuple_{rmSub})$
20: sendSubscriptionRequest($maxDzSubs, nodePortTuple_{rmSub}$)
21: **end if**
22: **end if**
23: decrementSubscriberCount($dzExp, sub$)
24: **end procedure**

### 5.6.2 Remote Un-subscriptions

The remote un-subscription handling is very similar to the local one. As discussed earlier, a subscription request from a remote controller is perceived as a request from a virtual subscriber

connected to the border switch-port tuple connecting the two concerned sub-networks. So, on receiving an un-subscription request from a neighboring controller, the virtual subscriber access point (node-port tuple to which it is connected) and dz-expression are identified. This is followed by centralized un-subscription handling of the request from the virtual subscriber resulting in pub/sub tree and flow modifications in the local network.

**Algorithm 8** Remote Un-Subscription Handling

1: **procedure** *removeRemoteSubscriber*(dzExp, sub, receivedPacket)
2: removeSubscriber($dzExp, sub$) {⇒ remove this subscriber from the corresponding pub/-sub tree for flow modification}
3: $localSubCount \leftarrow$ getLocalSubscriberCount($localSubscriberCounts, dzExp$) {⇒ gets local subscriber count for all dzs where dz $\succeq dzExp$}
4: $rmSubs \leftarrow \{remoteSubscribers : dz_{remoteSubscribers} \succeq dzExp\}$
5: **if** $localSubCount = 0 \ \wedge \ \mid rmSubs \mid \ = 1$ **then**
6: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \ \vee \ dzExp \succ dz_{remotePublishers}\}$
7: $maxDzSubs \leftarrow$ getNextMaxDzSubs($dzExp$) {⇒ get subscribers with next highest dzs}
8: **for each** $remotePublisher \in pubs$ **do**
9: **if** $nodePortTuple_{remotePublisher} \neq sub$ **then**
10: forward($receivedPacket, nodePortTuple_{remotePublisher}$)
11: sendSubscriptionRequest($maxDzSubs, nodePortTuple_{remotePublisher}$)
12: **end if**
13: **end for**
14: **else if** $localSubCount = 0 \ \wedge \ \mid rmSubs \mid \ = 2$ **then**
15: $pubs \leftarrow \{remotePublishers : dz_{remotePublishers} \succeq dzExp \vee dzExp \succ dz_{remotePublishers}\}$
16: $maxDzSubs \leftarrow$ getNextMaxDzSubs($dzExp$) {⇒ get subscribers with next highest dzs}
17: $rmSub \leftarrow \{rmSubs : nodePortTuple_{rmSub} \neq sub\}$
18: **if** $rmSub \in pubs$ **then**
19: forward($receivedPacket, nodePortTuple_{rmSub}$)
20: sendSubscriptionRequest($maxDzSubs, nodePortTuple_{rmSub}$)
21: **end if**
22: **end if**
23: removeRemoteSubscriber($dzExp, sub$)
24: **end procedure**

Then, the same decisions regarding spreading of the un-subscription request are taken as in the local cases and can be understood from the pseudo-code presented in Algo. 8. Here, lines 11 and 20 take care of sending the new subscriptions to relevant remote controllers in the function *sendSubscriptionRequest()*. However, care is taken to ensure that the controller from which the un-subscription request has arrived (denoted by the switch-port tuple *sub*) is never in the recipients list for the new subscriptions. Finally, the entry in *remoteSubscribers*

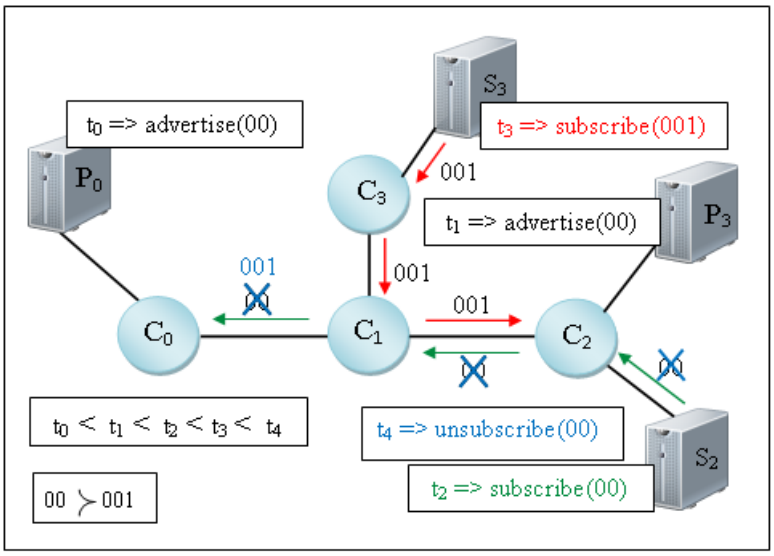



Figure 5.18: Example of un-subscription handling

corresponding to the un-subscribed dz-expression and virtual subscriber (remote controller) is removed.

In both algorithms, the arguments to the procedures are *dzExp*, denoting the un-subscribed dz-expression, *sub*, denoting the switch-port tuple connecting the local/virtual subscriber and *receivedPacket*, denoting the un-subscription message. An example of un-subscription handling is illustrated in Fig. 5.18 where the order of respective requests are depicted. Finally at time $t_4$, $S_2$ un-subscribes. As $C_2$ views sub-network 1 as a relevant virtual publisher, the request is sent to $C_1$. $C_1$ in turn forwards it to $C_0$ as it views $C_0$ as a relevant virtual publisher. However, $C_1$ identifies an existing subscription request from $C_3$ which was covered by the deleted subscription dz and was not sent to $C_0$ earlier. As a result, the subscription request is now sent to $C_0$ to make sure that there is a path between publisher $P_0$ and subscriber $S_3$ with respect to sub-space $\{001\}$.

All the algorithms discussed above contribute to establishing paths across sub-networks with publishers as the sources and interested subscribers as the end nodes. There is no particular controller with a global view of all complete paths but each controller may contribute to establishing parts of a path related to its local network. The concept of treating the rest of the network simply as a virtual publisher/subscriber makes it easier to establish/delete flows on local switches. Complete paths are established/deleted by the collective effort of the participating controllers of the network. All the algorithms presented in this chapter and those discussed in the previous chapter contribute to a working prototype implementation of a content-based pub/sub system using multiple controllers in software-defined networking.

# Chapter 6

# Analysis and Results

This chapter is dedicated to analyzing the design and implementation of the proposed system. The study is done to understand the effects of the design on parameters such as controller overhead, total control traffic, false positive rate etc. The analysis of the system with regards to controller overhead and generated control traffic for varying number of controllers is further supported by results obtained from performing experiments in a simulated network.

## 6.1 Test environment

The evaluations have been carried out in a Mininet[38, 39] environment on an i686 machine running Ubuntu 12.04 on a 2.00 GHz processor. Mininet uses the concept of OS-level lightweight virtualization for network emulation. It provides a platform that allows users to experiment with various topologies and application traffic. The same code can then be deployed in a real-world setting. By leveraging Linux features, Mininet offers an extensible CLI and API with the help of which an interactive and customizable software-defined network can be created. So, the tests have been conducted on such a Mininet environment with different number of instances of Floodlight controllers running on the same machine.

A ring-like topology created with 20 switches has been considered for the tests. Each switch has a host connected to it which may play the role of a publisher or a subscriber. Two types of data sets have been considered for advertisement and subscription requests. These data sets have been generated using uniform and zipfian distributions and are three-dimensional. Two different data sets have been used to understand the behavior of the system under different scenarios. The advertisements and subscriptions have been randomly distributed between the hosts of the network and the number of requests received per controller for different number of controller instances leading to different network configurations has been measured and the established flows have been recorded. For example, measurements were first taken in a single network controlled by a single controller with 20 switches for a particular sequence of host requests. This was followed by taking measurements for the same sequence of requests but with the network divided into 2, 4 and so on partitions. The results obtained from the experiments along with their relevance to various system performance metrics have been discussed in the following sections.

## 6.2 Control Overhead

The main motivation towards extending centralized pub/sub using SDN is to achieve better performance in terms of scalability. A centralized controller can potentially act as a bottleneck and a single point of failure. With increasing network elements, the CPU of the controller may be overloaded and/or system memory may be exhausted resulting in increased processing latency. So, the entire idea is to distribute the load of a single controller between multiple controllers.

This analysis is expected to address the following questions :

- How much knowledge about the full content-based routing topology is needed by a controller?
- How much coordination is needed between controllers?
- How many controllers/sub-networks will yield the best performance depending on a given event load and number of publishers and subscribers?

The design of the system ensures that a controller does not need to be aware of any details about the rest of the network. It only needs to know about the elements of the local sub-network assigned to it. Additionally, it needs to identify the border switch-port tuples in its local sub-network that form gateways to the remote network. Such a design reduces the overhead of a controller significantly as it does not need to take additional efforts to discover the topology of the remaining part of the network. Also, a controller creates spanning trees with incoming advertisement requests. In the case of centralized control, a single controller needs to store in memory every spanning tree covering every switch in the network. Also, it alone needs to establish all flows in every switch and store all these established flows so that it can delete/modify them when necessary. It needs to store in memory the identity of every publisher and subscriber along with their advertisement and subscription requests. With increasing network size and increasing number of requests from hosts, storing and processing such a lot of data may be overwhelming for a single controller resulting in increased processing latency. Partitioning the network implies that spanning trees created at each controller are smaller as they span across only the switches of the local network. Also, each controller is responsible for all flows installed in only the switches of its local network. This allows each controller to work autonomously within its local network and use different algorithms for flow establishment/deletion. Even though the current implementation uses the centralized algorithms of chapter 4 in every controller, nevertheless, the proposed distributed framework supports autonomous behavior within the sub-networks. Covering-based routing also provides an opportunity for each controller to be aware of only a subset of all host requests.

So, the design of the system addresses the first question regarding the amount of knowledge about the whole topology that a controller needs to be aware of. The remaining two questions have been addressed in the following discussions on average controller overhead and total

control traffic which clearly indicate the amount of coordination required between controllers along with the effect of partitioning on system performance.

### 6.2.1 Average Controller Overhead

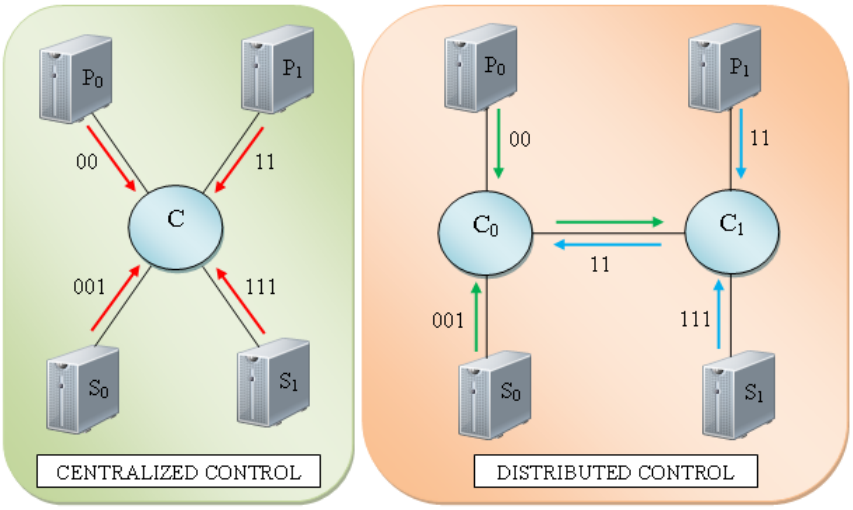



Figure 6.1: Comparison between centralized and distributed control

In the centralized approach, the control messages consist of all request messages sent by the hosts. In the distributed setting, there are mainly two categories of generated control messages. These are messages generated by hosts and those generated by controllers to share control information with other controllers. Even though this implies that the generated traffic increases in the distributed approach, analysis shows that the average overhead at each controller may reduce significantly on distributing the load. Earlier, every host request would be sent to the only controller available in the system. But now, the host requests are primarily sent only to the local controller which then takes a decision on sharing it with other controllers. This becomes apparent from Fig. 6.1 where an advertisement request from $P_0$ is handled by $C_0$ which then takes a decision to share it with $C_1$. Similarly, a request from $P_1$ is processed by $C_1$ and is also sent to $C_0$. Now, a subscription from $S_0$ is handled by $C_0$ but not forwarded to $C_1$ as its corresponding sub-network does not have a publisher relevant to this subscription. Similarly, a request from $S_1$ is handled by $C_1$ alone. So, the average number of control events handled by each controller in this simple example is 3. In contrast, the centralized controller has to handle 4 control messages. In this comparison, the average overhead at a controller is marginally less in the distributed case. This difference is more meaningful and apparent with increasing number of network elements and requests. However, determining the average controller overhead in terms of average number of requests processed per controller is not straightforward. There are a number of factors that influence this metric.

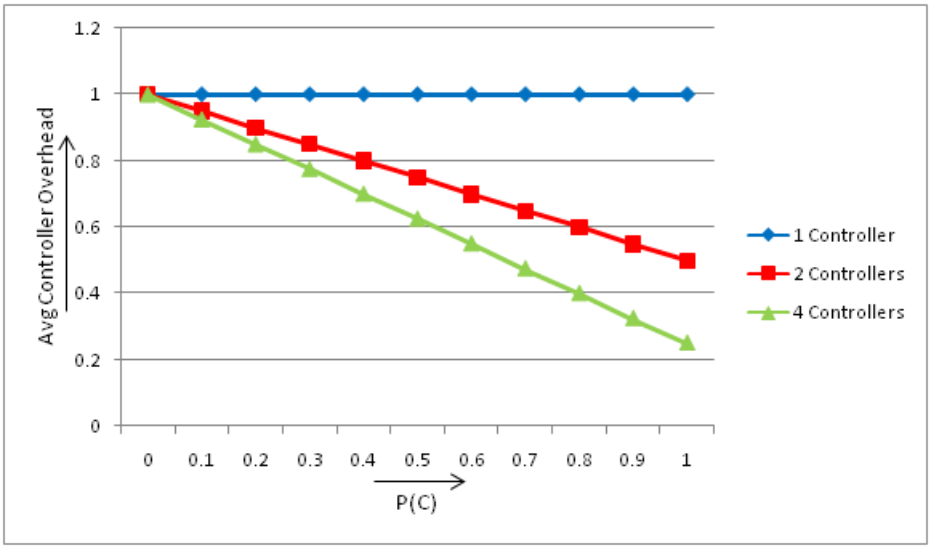



Figure 6.2: Average Controller Overhead (Advertisement)

### Advertisement Overhead

Let us first consider the overhead generated by advertisements. Considering a simple advertisement flooding approach in a network divided into $n$ partitions by $n$ controllers, the average controller overhead($ACO$) on arrival of an advertisement request is as follows :

$$\begin{aligned} ACO &= \frac{Total\ number\ of\ generated\ messages\ (n)}{number\ of\ controllers\ (n)} \\ &= 1 \end{aligned} \tag{6.1}$$

(6.1) implies that in case of general flooding of advertisements, every controller receives and processes a request and the number of controllers in the system does not affect $ACO$. However, the proposed design is based on a covering-based approach. As a result, the equation needs to be extended to allow a parameter defining the covering relationship. Let event $C$ represent the situation where the newly generated advertisement is equivalent to or covered by one or more already existing remote/local advertisements at the controller. Here, we generally consider that the advertisement is no longer shared with every controller even if a single match is obtained. Assuming the probability of occurrence of $C$ to be $P(C)$, the previous equation can be extended as follows :

$$\begin{aligned} ACO &= P(C) * \frac{1}{n} + (1 - P(C)) * \frac{n}{n} \\ &= 1 - P(C)(1 - \frac{1}{n}) \end{aligned} \tag{6.2}$$

(6.2) indicates that the average controller overhead changes with the number of controllers in the system for certain probability of occurrences of $C$. Here, the best case and the worst case scenarios for ACO are as follows :

The best case happens when there is a match at the controller for the received advertisement which can be represented as *P(A) = 1*. Under such circumstances, the advertisement is not shared with any other controller and the average controller overhead is :

$$ACO = 1 - 1 * (1 - \frac{1}{n}) = \frac{1}{n}$$

The worst case occurs when there is no match, i.e., *P(C) = 0* :

$$ACO = 1 - 0 * (1 - \frac{1}{n}) = 1$$

Fig. 6.2 plots ACO when *P(C)* is varied between 0 and 1 for networks controlled by 1, 2 and 4 controllers. The plots clearly indicate that the average controller overhead decreases with increased partitioning. The best case and worst case scenarios can be easily identified in the graphs as well.

#### Un-advertisement Overhead

One may expect that an un-advertisement contributes to the overhead as much as its corresponding advertisement as an un-advertisement needs to be sent exactly along the paths followed by the corresponding advertisement in order to cancel its existence throughout the network. So, intuitively, the ACO equation for un-advertisement should be identical to (6.2). However, the existence of previously covered requests at a controller also contributes to the average controller overhead during un-advertisements. This is because an un-advertisement may trigger the generation of one or more advertisement requests. Let $D$ denote the event representing the existence of one or more advertisements that were previously covered by an advertisement which currently needs to be cancelled. Let *E(D)* be the expected value of $D$. Again, $C$ denotes the event representing a match. So, if another matching advertisement exists, then the un-advertisement is not shared with its neighbors. Otherwise, not only is the un-advertisement request shared with all other controllers, but also all covered advertisements are shared with the same. So, the previous equation can be extended as follows :

$$\begin{aligned} ACO &= P(C) * \frac{1}{n} + (1 - P(C))[\frac{n}{n} + E(D) * \frac{n-1}{n}] \\ &= 1 + (1 - \frac{1}{n})[E(D)(1 - P(C)) - P(C)] \end{aligned} \tag{6.3}$$

Again, the best case scenario is when *P(C) = 1*. Under these circumstances, the number of covered advertisements makes no difference to the average controller overhead.

$$ACO = 1 + (1 - \frac{1}{n})[E(D) * (1 - 1) - 1] = \frac{1}{n}$$

When there is no match for the received request, i.e., *P(C) = 0*, the number of covered advertisements at a controller determines ACO. If no covered dzs are present, then the ACO

value is independent of the number of partitions. On the other hand, every advertisement stored at the controller may have to be shared. So, the worst case scenario depends on *E(D)*.

$$ACO = 1 + (1 - \frac{1}{n})[E(D) * (1 - 0) - 0] = 1 + E(D)(1 - \frac{1}{n})$$

Fig. 6.3 depicts ACO for a constant value of *E(D)* ($E(D) = 1$) when *P(C)* is varied between

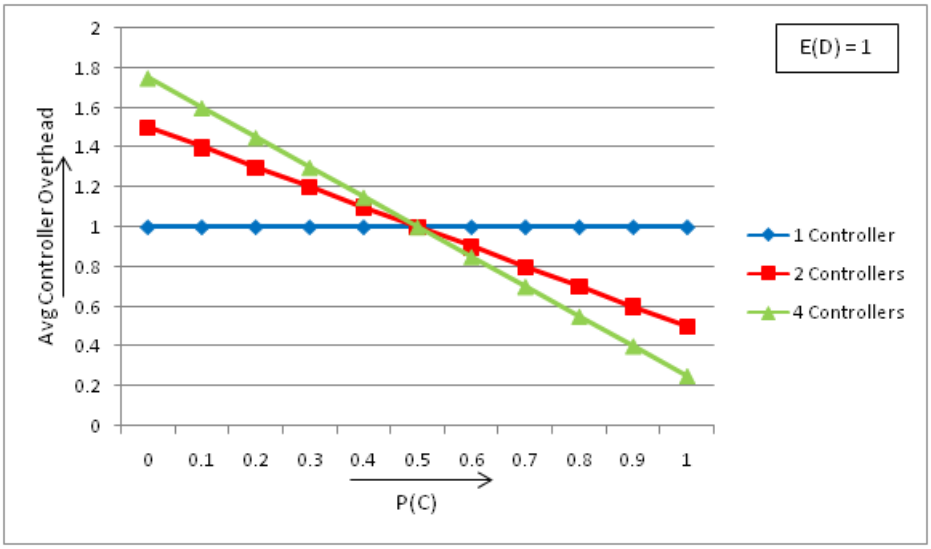



Figure 6.3: Average Controller Overhead (Un-advertisement)

0 and 1 for networks controlled by 1, 2 and 4 controllers. This portrays the various values that ACO can have between the best and worst cases for *E(D)=1*. For un-advertisements, it is clear from the plots that for $P(C) < 0.5$, increased partitioning increases average controller overhead if *E(D)=1* and for $P(C) > 0.5$, increased partitioning decreases average controller overhead. However, it must be noted that these values are strictly applicable for the considered value of *E(D)*. For, example, if $E(D) = 0$, then the worst case value for a network with any number of controllers is always 1.

### Subscription Overhead

Controller overhead caused by subscription requests are very similar to that caused by advertisements. However, another parameter needs to be considered before forwarding subscription requests. This thesis does not follow subscription flooding and subscriptions are only forwarded to remote controllers that are viewed as relevant virtual publishers by the local controller. So, let $A$ be an event representing the existence of one or more relevant publishers in the remote network. Let *P(A)* be the probability of occurrence of $A$. Again, on the event of a match, the subscription is no longer shared but if no match is found, then the request is shared depending on the existence of relevant publishers represented by *P(A)*. Then, (6.2)

can be extended for subscriptions as follows :

$$\begin{aligned} ACO &= P(C) * \frac{1}{n} + (1 - P(C))[P(A) * \frac{n}{n} + (1 - P(A)) * \frac{1}{n}] \\ &= \frac{1}{n} + P(A)(1 - P(C))(1 - \frac{1}{n}) \end{aligned} \tag{6.4}$$

Clearly, even in (6.4), the average controller overhead changes with the number of controllers for certain probability of occurrences of $C$ and $A$. The best case and the worst case scenarios depend on whether or not there is a match. If a match is not found, the worst case scenario occurs when a relevant publisher exists in the system for which the subscription has to be sent to every other controller.

Best case : $P(C) = 1$, $P(A) = \{x : x \in [0 \text{ to } 1]\}$

$$ACO = \frac{1}{n} - x * (1 - 1)(1 - \frac{1}{n}) = \frac{1}{n}$$

Worst case : $P(C) = 0$ , $P(A) = 1$

$$ACO = \frac{1}{n} - 1(1 - 0)(1 - \frac{1}{n}) = 1$$

The above equation indicates that there are three factors affecting ACO due to a subscription.

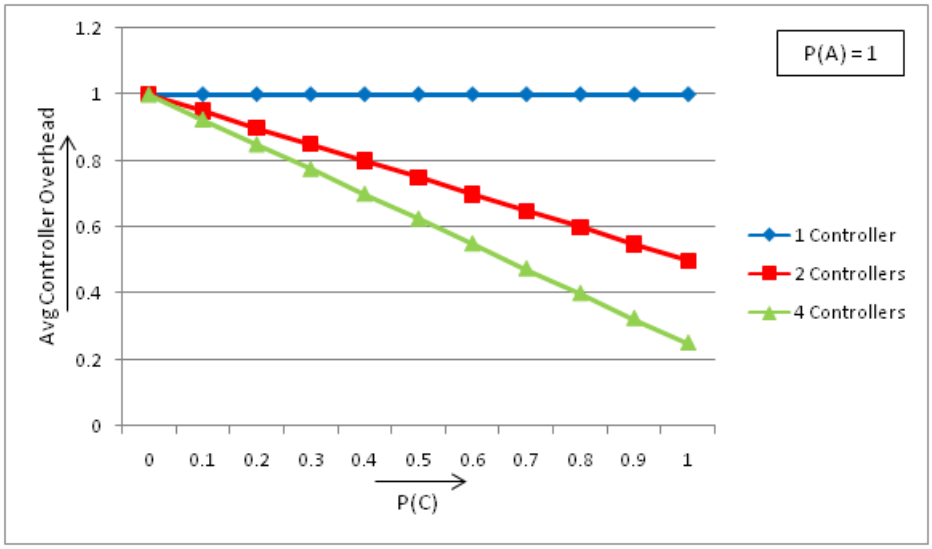



Figure 6.4: Average Controller Overhead (Subscription)

So, Fig. 6.4 keeps the value of $P(A)$ constant at 1 and plots ACO along $P(C)$ for networks controlled by 1, 2 and 4 controllers. With $P(A) = 1$, these graphs become similar to the ones representing advertisement overhead. Here too, the overhead decreases with increased partitioning. However, with decreasing value of $P(A)$, the overhead decreases further till the value $1/n$, beyond which it cannot decrease.

### Un-subscription Overhead

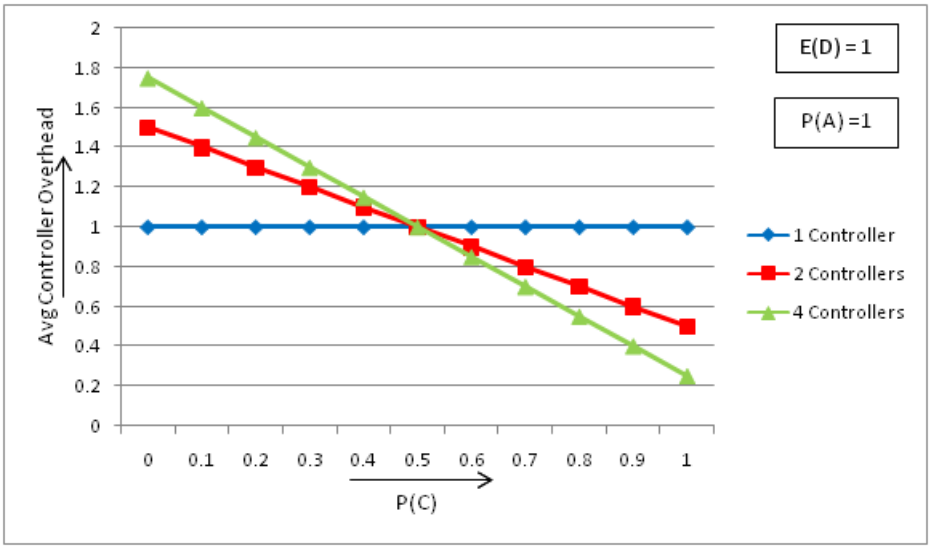



Figure 6.5: Average Controller Overhead (Un-subscription)

Again, intuitively, the *ACO* equation for un-subscription should be identical to (6.4). However, as in the case with un-advertisement, the existence of previously covered requests at a controller once again contributes to the average controller overhead during un-subscription. Let $D$ denote the event representing the existence of one or more subscriptions that were previously covered by a subscription which currently needs to be cancelled. So, let *E(D)* be the expected value of $D$. Here, on the event of a match, the request is not shared with any neighboring controller. However, if no match occurs, then the existence of a relevant publisher is considered. If such publishers exist then not only is the un-subscription message propagated to relevant controllers, but also all covered subscriptions are forwarded to the same remote controllers. So, the previous equation can be extended as follows :

$$\begin{aligned} ACO &= P(C) * \frac{1}{n} + (1 - P(C))[P(A)(\frac{n}{n} + E(D) * \frac{n-1}{n}) + (1 - P(A)) * \frac{1}{n}] \\ &= \frac{1}{n} + (1 - P(C))(1 - \frac{1}{n})[P(A)(1 + E(D))] \end{aligned} \tag{6.5}$$

Using (6.5), the best and worst case scenarios are as follows :

As before, the best case scenario is when a match occurs. In this case, the number of covered subscriptions and the existence of relevant remote publishers make no difference to the average controller overhead. So, when $P(C) = 1$, $P(A) = \{x : x \in [0 \text{ to } 1]\}$, then

$$ACO = \frac{1}{n} + (1 - 1)(1 - \frac{1}{n})[x * (1 + E(D))] = \frac{1}{n}$$

However, when there is no match for the received request, i.e., $P(C) = 0$ and there exist relevant publishers, i.e., $P(A) = 1$, the number of covered subscriptions at a controller determines

ACO. In the worst case, all subscriptions at the controller may need to be forwarded to every other controller. So, the worst case scenario depends on *E(D)*.

$$ACO = \frac{1}{n} + (1-0)(1-\frac{1}{n})[1*(1+E(D))] = 1 + E(D)*(1-\frac{1}{n})$$

In an un-subscription, ACO depends on four parameters. So, Fig. 6.5 considers the value of *P(A)* to be 1, the value of *E(D)* to be 1 and plots ACO along *P(C)* for networks controlled by 1, 2 and 4 controllers. The best case and worst case scenarios can be determined from the graphs. Again, before a particular value of *P(C)*, the overhead increases with increased partitioning and beyond this value it decreases with increasing number of controllers. It should be noted that the graphs change significantly with varying values of *P(A)* and *E(D)*.

### Influencing Factors

It is quite clear from the above equations that various factors influence the number of requests received by each controller in a distributed setting. All these factors are listed below.

- The existence of a match for the received advertisement/subscription request denoted by $C$ in the equations above is one of the most important parameters to be considered. Here, a match represents equivalence or covering relation. $C$ includes a match with one or more existing local or remote requests at a controller. However, it should be noted that a particular match has been approximated and treated in the same way as the others in the equations above for the sake of simplicity. This is the case where only one other remote controller request exists which matches the current request. Under such circumstances, other than the host request received at the controller, an additional controller generated request may also be sent to the aforementioned remote controller.
- The existence of one or more remote publisher relevant to an unmatched subscription request plays an important role during subscription/un-subscription handling. Such an occurrence has been represented by event $A$ in (6.4) and (6.5).
- Not only the existence of relevant remote publishers, but also their locations in the network are important while dealing with subscriptions/un-subscriptions. This determines the remote controllers to which a subscription/un-subscription request needs to be forwarded.
- The existence of previously covered requests at a controller also contributes to the average controller overhead during un-subscriptions and un-advertisements. This is because an un-subscription/un-advertisement may trigger the generation of one or more subscription/advertisement requests. Such an event is represented with $D$ in (6.3) and (6.5).
- Lastly, the formulated equations clearly indicate that the value of $ACO$ depends on the number of partitions/controllers($n$) in the network and the positions of publishers and subscribers in the different partitions.

It should be noted that the aforementioned equations do not portray all complex scenarios for the sake of simplicity.

The equations relevant to advertisement and subscription clearly point out that increased partitioning improves the average controller overhead except for the worst case when partitioning has no effect. However, the best case and worst case values for un-advertisements and un-subscriptions determine that they may in certain scenarios reduce average controller overhead or may increase it in certain others. So, the aforementioned equations indicate that different types of events contribute differently to the average controller overhead depending on the listed factors. Different sequences of events would lead to different scenarios resulting in different *ACO* values. So, determining the number of controllers to be used in a particular system in order to reap maximum benefits with respect to controller overhead is not definite and would vary based on request sources and event sequences. However, to get an idea of the effect of the number of controllers on the average controller overhead in a particular setting, experiments have been conducted in a simulated network.

### Experimental Results

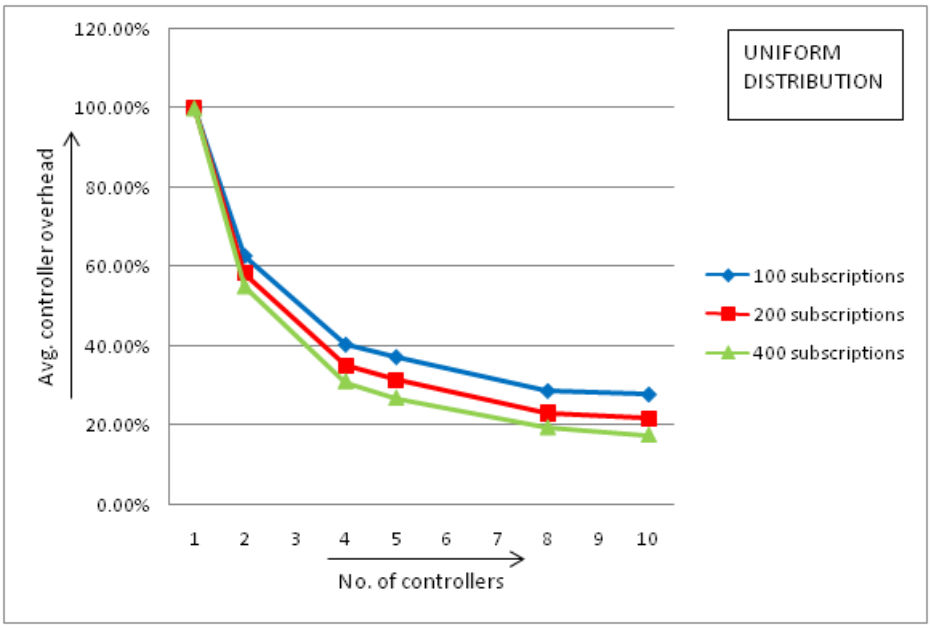



Figure 6.6: Average Controller Overhead (uniform distribution)

The environment setup, topology and data sets used have already been discussed before in section 6.1. In such a setting, the average controller overhead is measured when the selected network is divided into increased number of partitions. The same tests have been repeated on a network controlled by varying number of controllers. Fig. 6.6 presents normalized graphs

depicting average controller overhead for 100, 200 and 400 subscriptions generated using uniform distribution when the number of controller instances is gradually increased. So, the x-axis depicts the number of controllers and the y-axis stands for the normalized average controller overhead. For each subscription set depicted in blue, red and green, the average controller overhead reduces with increasing number of controllers. For example, the figure demonstrates that by distributing the control between two controllers, a benefit of around 45% as compared to the centralized control was achieved when the system was tested with 400 subscriptions. This benefit increases further by around 25% when 4 controllers are used as shown in the graph and so on. The normalization has been done to compare the benefits of partitioning when different number of subscriptions is used. It is visible from the figure that if the number of subscriptions is significantly increased, the benefit of partitioning increases as well. This further supports the above equations as with more number of subscriptions, the probability of a match at the local controller also increases. This means that gradually lesser number of subscriptions needs to be shared with remote controllers because of covering-based routing. As a result, the depicted graph for 400 subscriptions shows higher benefits with increased partitioning than the graph for 200 subscriptions which again displays higher benefits as compared to the graph for 100 subscriptions.

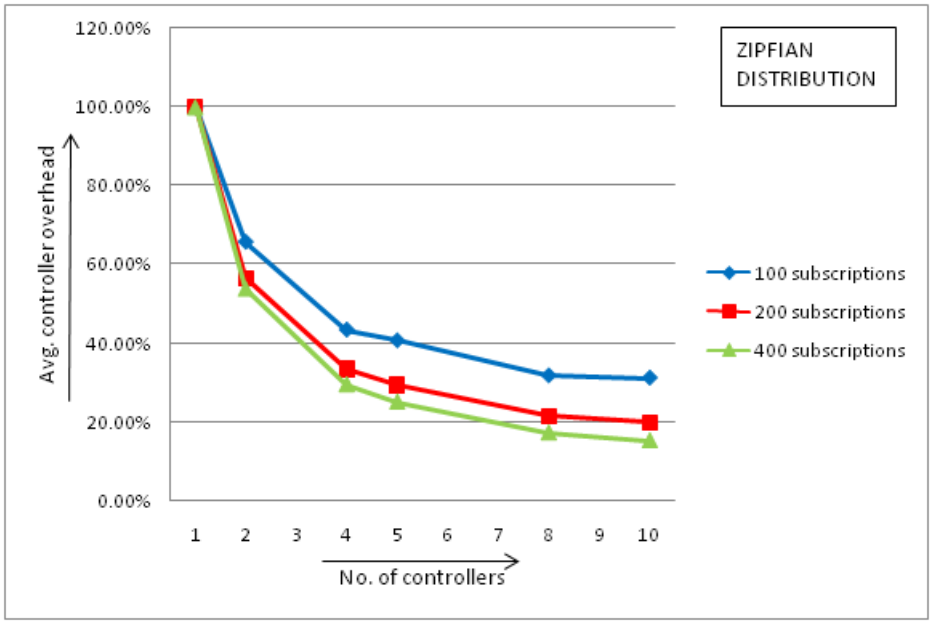



Figure 6.7: Average Controller Overhead (zipfian distribution)

Fig. 6.7 presents normalized graphs depicting average controller overhead for 100, 200 and 400 subscriptions generated using zipfian distribution when the number of controller instances is gradually increased. The x-axis and y-axis bear the same meaning as before. Even for this distribution, for each subscription set depicted in blue, red and green, the average controller overhead reduces with increasing number of controllers. In fact, this time, the figure

demonstrates that by distributing the control between two controllers, a benefit of around 47% as compared to centralized control was achieved when the system was tested with 400 subscriptions. This benefit increases further with more and more partitioning of the network. For the same reasons mentioned above, the depicted graph for 400 subscriptions reduces more rapidly with increased partitioning than the graph for 200 subscriptions which again reduces more rapidly as compared to the graph for 100 subscriptions for zipfian distribution as well.

It should be noted that the graphs presented above are simply examples of the behavior of a particular system to a particular sequence of events. The results obtained from the conducted experiments largely support the presented analysis.

### 6.2.2 Total Control Traffic

In the distributed setting, messages generated by hosts and those generated by controllers contribute to the total control traffic as compared to those generated only by hosts in the centralized approach. So, even though the distributed pub/sub may perform better in terms of scalability, it clearly generates more control traffic due to the additional controller requests used for communication between controllers. So, with increased partitioning, the total control traffic would also increase. The total generated control traffic( $TCT$) can be represented with equations simply by using the above equations for average controller overhead. The best case and worst case values for each request type have also been presented.

#### Advertisement Overhead

Using the same argument as in average controller overhead analysis (6.2), $TCT$ generated by an advertisement request can be represented as follows :

$$\begin{aligned} TCT &= P(C) * 1 + (1 - P(C)) * n \\ &= n - P(C)(n-1) \end{aligned} \tag{6.6}$$

Here, event $C$ bears the same meaning as above.

Best case : *P(C) = 1*

$$ACO = n - 1 * (n-1) = 1$$

Worst case : *P(C) = 0*

$$ACO = n - 0 * (n-1) = n$$

Fig. 6.8 depicts generated traffic with varying values of *P(C)* when 1, 2 and 4 controllers control the network. The best case and worst case values are clear from the figure. The figure also indicates that increased partitioning increases control traffic.

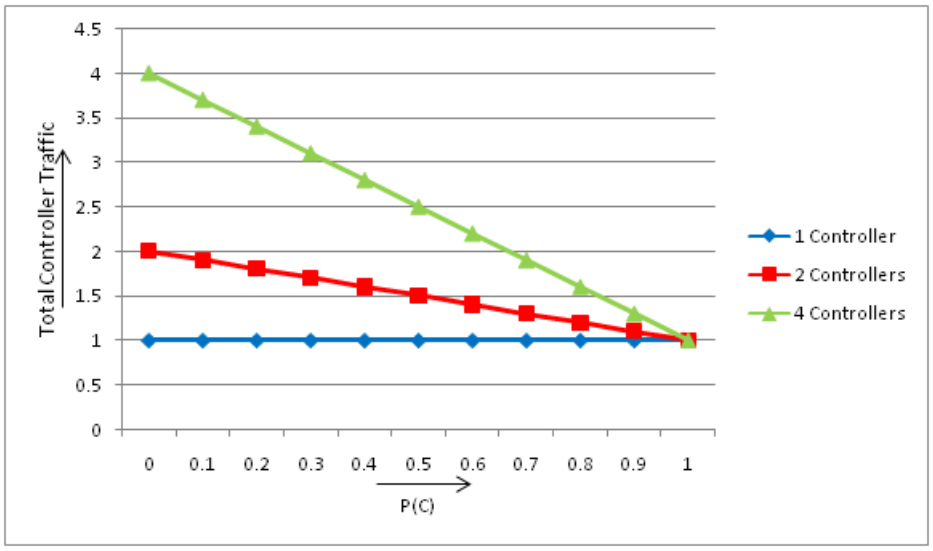



Figure 6.8: Total Control Traffic (Advertisement)

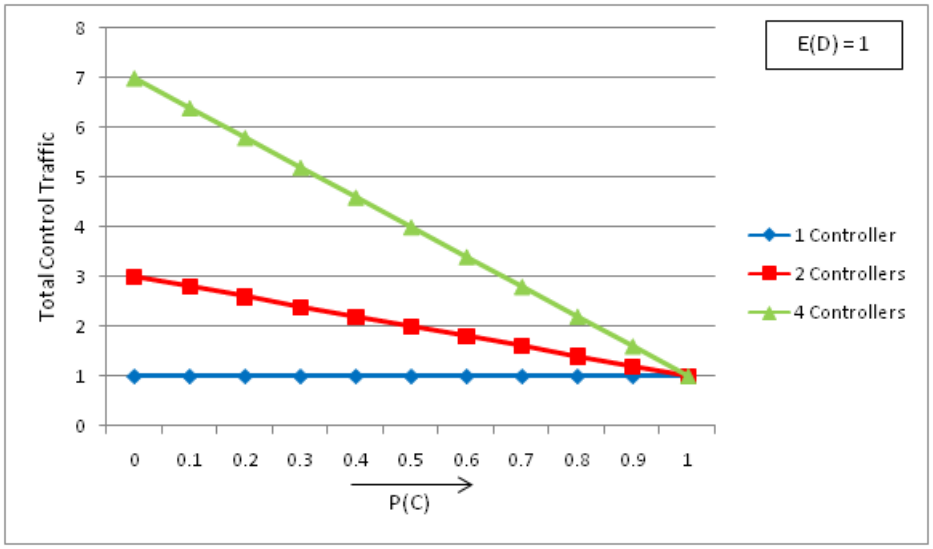



Figure 6.9: Total Control Traffic (Un-advertisement)

### Un-advertisement Overhead

The equation relevant to an un-advertisement can be derived from equation (6.3) and can be presented as follows :

$$\begin{aligned} TCT &= P(C) * 1 + (1 - P(C))[n + E(D) * (n - 1)] \\ &= n + (n - 1)[E(D)(1 - P(C)) - P(C)] \end{aligned} \tag{6.7}$$

Here, along with event $C$, event $D$ also bears the same meaning as above.

Best case : $P(C) = 1$

$$TCT = n + (n - 1)[E(D) * (1 - 1) - 1] = 1$$

Worst case : $P(C) = 0$

$$TCT = n + (n - 1)[E(D)(1 - 0) - 0] = n + E(D) * (n - 1)$$

Fig. 6.9 portrays TCT with varying values of $P(C)$ when 1, 2 and 4 controllers control the network and $E(D)$ has a constant value of 1. The best case and worst case values for the 3 scenarios and the selected constant value of $E(D)$ are clear from the graphs. The plots show that for $E(D) = 1$ un-advertisement increases control traffic further as compared to the control traffic generated for an advertisement. Also, the worst case is unbounded as it depends on the value of $E(D)$.

### Subscription Overhead

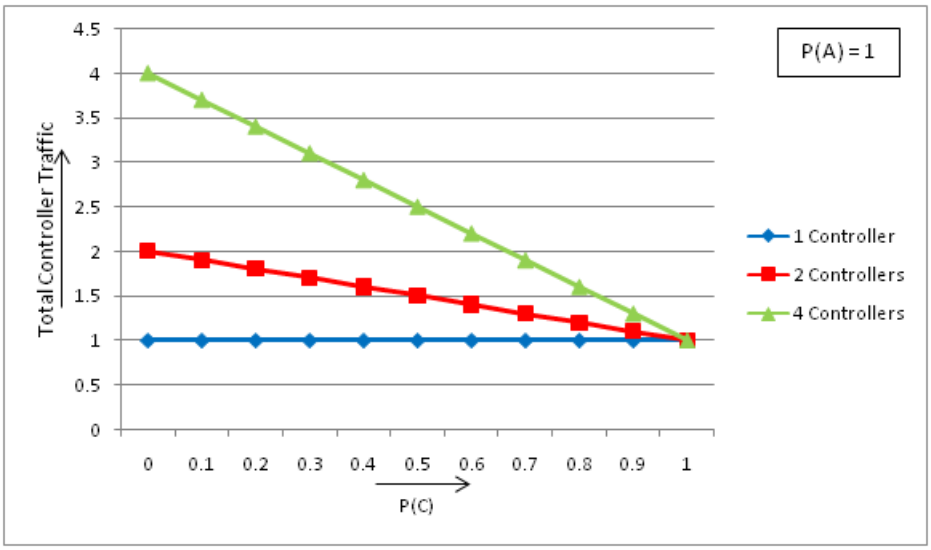



Figure 6.10: Total Control Traffic (Subscription)

The following equation formulates the total control traffic generated by a subscription request and can be derived from (6.4).

$$\begin{aligned} TCT &= P(C) * 1 + (1 - P(C))[P(A) * n + (1 - P(A)) * 1] \\ &= 1 + P(A)(1 - P(C))(n - 1) \end{aligned} \tag{6.8}$$

Again, events $C$ and $A$ mean the same as above.

Best case : $P(C) = 1$, $P(A) = \{x : x \in [0 \text{ to } 1]\}$

$$TCT = 1 + x * (1 - 1)(n - 1) = 1$$

Worst case : $P(C) = 0$ , $P(A) = 1$

$$TCT = 1 + 1 * (1 - 0)(n - 1) = n$$

Fig. 6.10 plots TCT along $P(C)$ when the network is divided into 1, 2 and 4 partitions and a constant value of $P(A) = 1$. The best and worst case scenarios can be determined from the figure as well. It is visible from the graphs that increased partitioning increases control traffic except for the best case.

### Un-Subscription Overhead

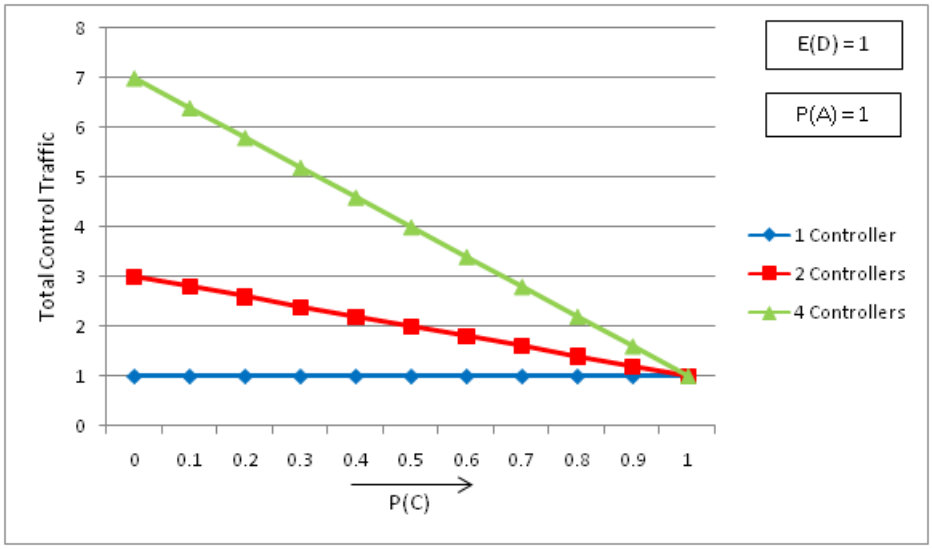



Figure 6.11: Total Control Traffic (Un-subscription)

Finally, the equation relevant to un-subscription may be derived from (6.5) as follows :

$$\begin{aligned} TCT &= P(C) * 1 + (1 - P(C))[P(A)(n + E(D) * (n - 1)) + (1 - P(A)) * 1] \\ &= 1 + (n - 1)(1 - P(C))[P(A)(1 + E(D))] \end{aligned} \tag{6.9}$$

Events $C$, $A$ and $D$ have the same meaning as before.

Best case : $P(C) = 1,\ P(A) = \{x : x \in [0\ to\ 1]\}$

$$TCT = 1 + (n-1)(1-1)[x * (1 + E(D))] = 1$$

Worst case : $P(C) = 0\ ,\ P(A) = 1$

$$TCT = 1 + (n-1)(1-0)[1 * (1 + m * 1)] = n + E(D) * (n-1)$$

In an un-subscription, even TCT depends on four parameters. So, Fig. 6.11 keeps the value of $P(A)$ constant at 1, the value of $E(D)$ constant at 1 and plots TCT along $P(C)$ when the network is divided into 1, 2 and 4 partitions. Again, the best case and worst case scenarios for such a setting are clear from the figure as is the fact that except for the best case, in all other scenarios increased partitioning increases control traffic. Also, un-subscription may contribute to the total control traffic more than a subscription does.

### Experimental Results

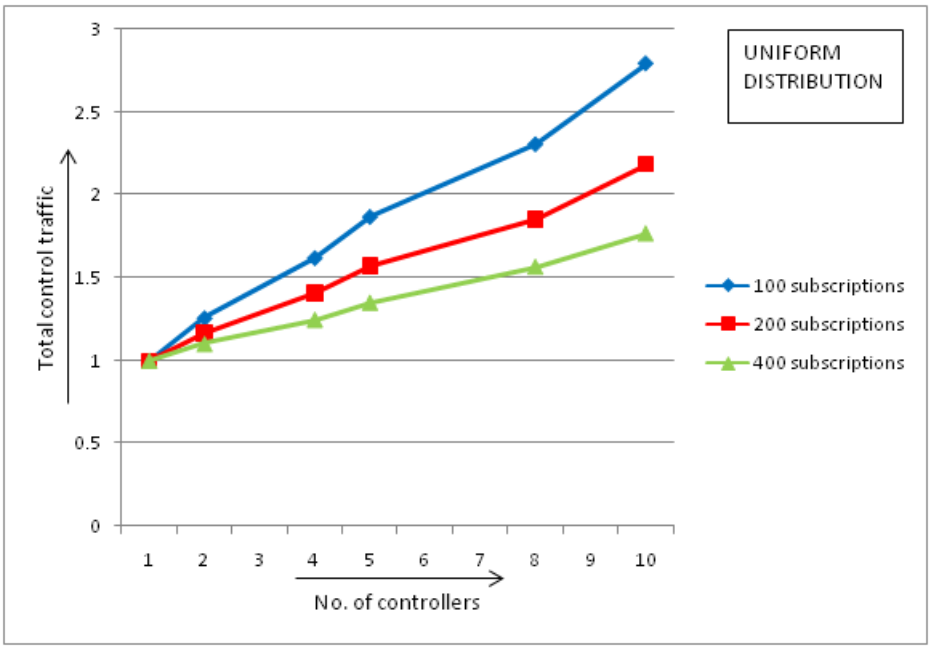



Figure 6.12: Total Control Traffic (uniform distribution)

It is quite clear from the equations above that except for the best case, in each type of request, control traffic increases with increasing number of controllers. The experiments performed confirm this fact where again the same test environment is used. In the following graphs the total control traffic is plotted on the y axis with the number of controllers along x-axis.

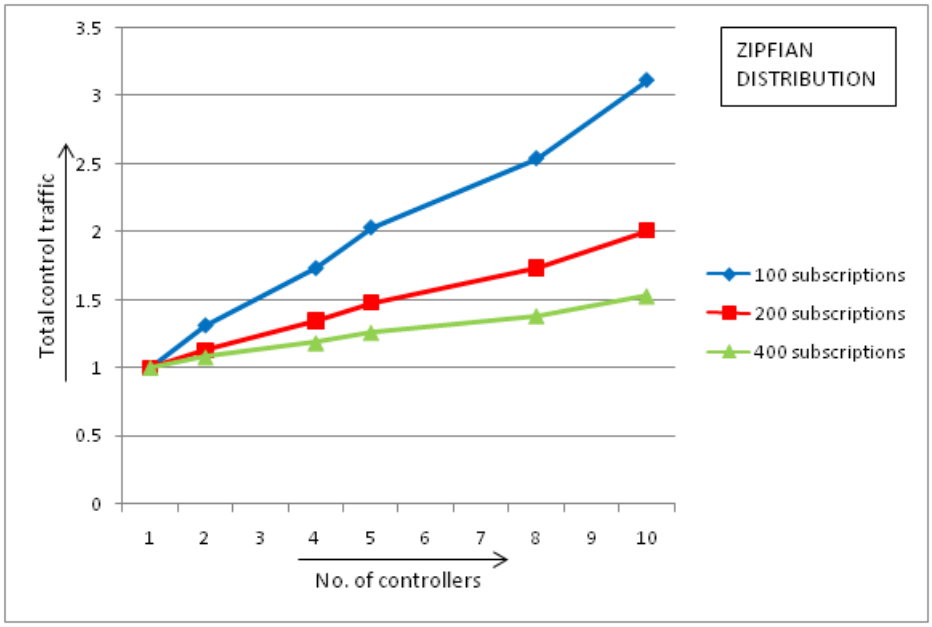



Figure 6.13: Total Control Traffic (zipfian distribution)

Fig. 6.12 plots the total control traffic for 100, 200 and 400 subscriptions generated using uniform distribution. In each case, the traffic increases with increasing number of controllers. The graphs are again normalized to compare the behavior of the system in all three cases. As expected, the comparative increase in control traffic for 400 subscriptions is lesser than 200 subscriptions which in turn is lesser than 100 subscriptions. The reason behind this is that with more number of subscriptions, the probability of a match is also more. This results in less number of controller messages being shared between remote controllers.

The graphs for zipfian distribution are depicted in Fig. 6.13. Even for this distribution, for each subscription set depicted in blue, red and green, the total control traffic increases with increasing number of controllers. For the same reasons discussed in uniform distribution, here too the graph for 100 subscriptions grows more rapidly, followed by the graph for 200 subscriptions and finally the one for 400 subscriptions.

So, the results indicate that partitioning of the network should be done according to bandwidth constraints and performance requirements of the system in question. Increased number of controllers may result in a lower average controller overhead, but it may result in very high control traffic. So, both these parameters should be considered before taking a decision on the total number of controllers to be used in the system. Finally, it can be concluded that addressing the question on the number of controllers to be used in the system is not straightforward. It cannot be generalized and depends on all the factors influencing it as pointed out in this chapter.

## 6.3 Other Performance Metrics

All discussions carried out in this chapter up till now have been related to the control messages. However, this section deals with system performance with respect to events/notifications/publications.

False positives are notifications which are delivered to a subscriber which is not interested in receiving them. Moreover, false positive rate(FPR) may be defined as follows :

$$FPR = \frac{Number\ of\ false\ positives}{Total\ number\ of\ notifications\ received} * 100$$

The performance of the present system with respect to false positive rate is similar to that of the system with a single controller implemented by Mishra. The reason behind this is that both approaches represent the content-space in the same way as presented by Tariq et al.[5]. Also, both carry out header-based mapping using remaining 23 bits of an IPv4 address for representing the dz expression. Clearly, distributing the control logic between multiple controllers has no effect on the false positive rate due to data representation. So, the behavior of the system with respect to false positives is similar to the centralized implementation and can be referred to in [10]. Experimental results in [10] indicate that false positive rate decreases with increasing dz-length. Mishra argues that with increasing dz-expression length, the granularity of event sub-spaces also increase which results in decreased false positives. However, in this scheme, only 23 bits are available for dz-expression representation. So, dz-expressions that differ after 23 bits cannot be differentiated. For example, if a subscription is represented as {<23 bits>0}, then its subscriber may receive events corresponding to the sub-space {<23 bits>1} where the first 23 bits of the event are identical to that of the subscription. The restriction imposed on the dz size is the root cause of this problem. However, it is important to note that as was the case in the centralized algorithms, here too no false positives are additionally introduced by the system implementation with respect to dz-expressions, i.e., if a subscription is assigned dz say *dzExp*, it will never receive an event which does not belong to the sub-space *dzExp*. Finally, it can be concluded that partitioning the network has no effect on false positive rate.

False negatives are events that are not delivered to a subscriber which has expressed its interest in receiving them. Needless to say, the aim of any pub/sub system is to keep this performance metric to a minimum. So, ideally, a subscriber should eventually receive any event $e$ published after at most time period $\delta$ from the time the subscriber generates a subscription $s$ where $e \sqsubset s$. The design of the present system ensures that paths are established between every publisher and all its relevant subscribers across sub-networks irrespective of the partitions in the network. Flow establishment operations within each local network and exchange of control information between controllers maintains connectivity between publishers and relevant subscribers. So, one can argue that partitioning has no effect on connectivity in the network.

The current system enables line-rate forwarding of published events. However, an analysis of the effect of partitioning on end-to-end delay and bandwidth usage of events is not straightforward as this depends on many factors such as complexity of the network topology, the manner in which network has been partitioned, the optimization algorithms used locally etc. However, one can argue that in a strictly acyclic topology, partitioning does not have any effect on the bandwidth usage of events as due to the acyclic nature of the network, there always exists a single path between two hosts. As a result, the established paths are identical irrespective of the number of partitions in the system.

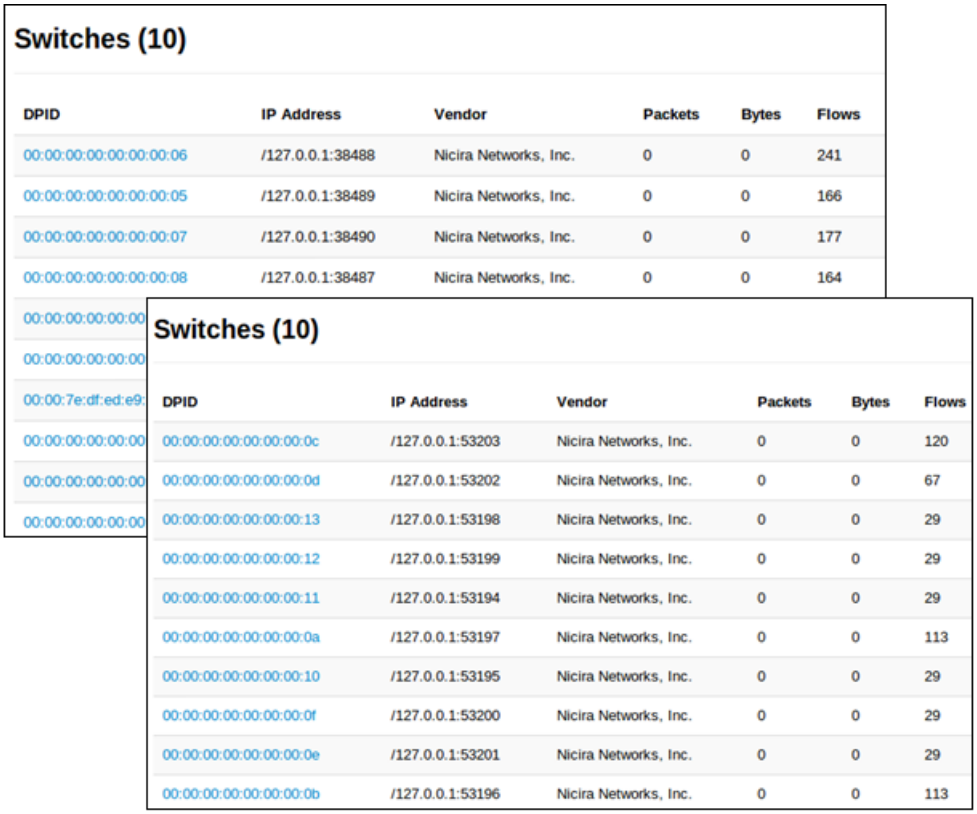
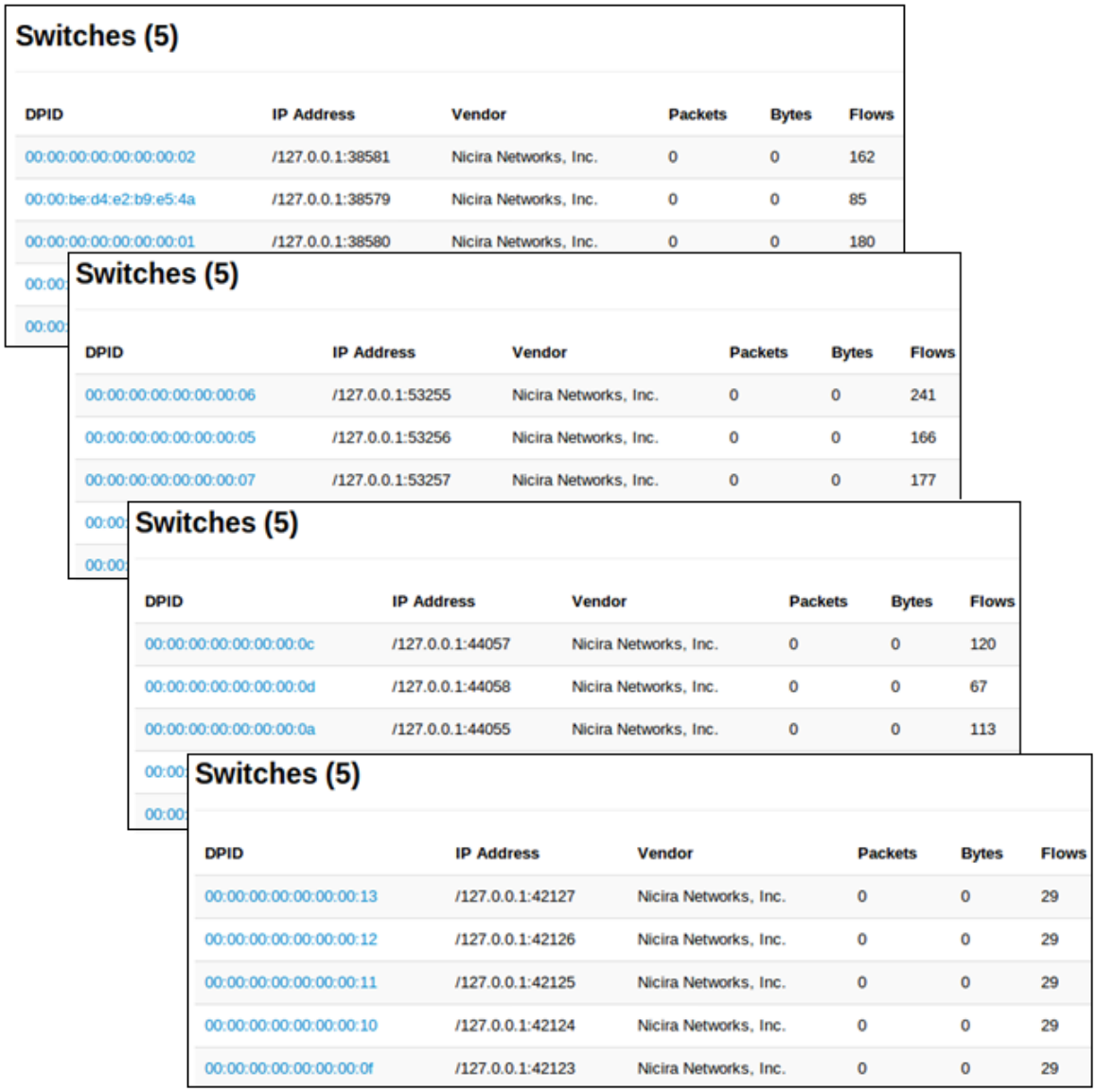



Figure 6.14: Example flows on switches when network has 2 controllers

The aforementioned arguments can be further supported by experiments conducted in the same environment as stated above. Here, a strictly acyclic topology of 20 switches each connected to a host was considered and 200 subscriptions generated from zipfian distribution were divided randomly among hosts. Then, the network was gradually partitioned among different number of controllers and the same experiment was repeated with the same requests with each network configuration. Results showed that for such an acyclic topology, partitioning has no effect on the flows and the same flows were installed in every scenario. For example, Fig. 6.14 illustrates an overview of the flows installed at the switches of each sub-network when the network was divided into 2 partitions of 10 switches each. Again, Fig. 6.15 illustrates



Switches (5)

| DPID | IP Address | Vendor | Packets | Bytes | Flows |
|---|---|---|---|---|---|
| 00:00:00:00:00:00:00:02 | /127.0.0.1:38581 | Nicira Networks, Inc. | 0 | 0 | 162 |
| 00:00:be:d4:e2:b9:e5:4a | /127.0.0.1:38579 | Nicira Networks, Inc. | 0 | 0 | 85 |
| 00:00:00:00:00:00:00:01 | /127.0.0.1:38580 | Nicira Networks, Inc. | 0 | 0 | 180 |

Switches (5)

| DPID | IP Address | Vendor | Packets | Bytes | Flows |
|---|---|---|---|---|---|
| 00:00:00:00:00:00:00:06 | /127.0.0.1:53255 | Nicira Networks, Inc. | 0 | 0 | 241 |
| 00:00:00:00:00:00:00:05 | /127.0.0.1:53256 | Nicira Networks, Inc. | 0 | 0 | 166 |
| 00:00:00:00:00:00:00:07 | /127.0.0.1:53257 | Nicira Networks, Inc. | 0 | 0 | 177 |

Switches (5)

| DPID | IP Address | Vendor | Packets | Bytes | Flows |
|---|---|---|---|---|---|
| 00:00:00:00:00:00:00:0c | /127.0.0.1:44057 | Nicira Networks, Inc. | 0 | 0 | 120 |
| 00:00:00:00:00:00:00:0d | /127.0.0.1:44058 | Nicira Networks, Inc. | 0 | 0 | 67 |
| 00:00:00:00:00:00:00:0a | /127.0.0.1:44055 | Nicira Networks, Inc. | 0 | 0 | 113 |

Switches (5)

| DPID | IP Address | Vendor | Packets | Bytes | Flows |
|---|---|---|---|---|---|
| 00:00:00:00:00:00:00:13 | /127.0.0.1:42127 | Nicira Networks, Inc. | 0 | 0 | 29 |
| 00:00:00:00:00:00:00:12 | /127.0.0.1:42126 | Nicira Networks, Inc. | 0 | 0 | 29 |
| 00:00:00:00:00:00:00:11 | /127.0.0.1:42125 | Nicira Networks, Inc. | 0 | 0 | 29 |
| 00:00:00:00:00:00:00:10 | /127.0.0.1:42124 | Nicira Networks, Inc. | 0 | 0 | 29 |
| 00:00:00:00:00:00:00:0f | /127.0.0.1:42123 | Nicira Networks, Inc. | 0 | 0 | 29 |

Figure 6.15: Example flows on switches when network has 4 controllers

the same when the network was divided into 4 partitions of 5 switches each. In both cases, each switch has the exact same flows irrespective of the partition it belongs to. This indicates that the published events would follow the same paths in both scenarios. The identical flows in this simple set of experiments further confirm the earlier arguments that partitioning does not have any effect on connectivity and false positive rates.

# Chapter 7

# Conclusion and Future Work

This thesis presented a content-based pub/sub system implemented at the network layer that uses the power of software-defined networking for scalable event dissemination. The design enables spatial partitioning of the network between multiple controllers where each controller does not need to maintain information regarding the remaining part of the topology, reducing overhead significantly. The proposed distributed framework incorporates existing centralized algorithms providing in-network filtering by using the capabilities of SDN. The framework provides each controller with the flexibility of using independent optimization algorithms within their local sub-networks. However, establishing routes between hosts of each local subnet is not enough as state information must be shared among controllers to ensure connectivity between publishers and subscribers spread across sub-networks. So, the system provides simple means for coordination between controllers and tries to reduce unnecessary bandwidth usage required for the same. This is achieved by employing a covering-based routing strategy and avoiding subscription flooding. The presented solution takes advantage of all the performance benefits of an implementation on the network layer through line-rate forwarding of events as well as the benefits of distributed control in terms of scalability.

The distributed control algorithms were discussed in details along with an analysis of the same. Experiments were conducted on a simulated network which confirmed the intuitions regarding the average controller overhead and total traffic generated on distributing control. The evaluations were mainly done to identify the behavior of the system with increased partitioning. The experiments showed that average controller overhead may be greatly reduced with increased partitioning, but it comes with a price. With increasing number of controllers, the generated control traffic also increases which affects bandwidth efficiency. So, it can be concluded that deciding on the number of partitions which yields maximum performance benefits cannot be generalized as it depends on various factors and also system requirements. Besides these, the performance of a system with distributed controllers with respect to false positive rate and connectivity were analyzed and it was concluded that partitioning has no significant effect on them. It was also inferred that for a strictly acyclic network topology, the bandwidth usage of a set of published events does not differ with increased network partitioning.

As is true in most cases, there is always room for improvements and enhancements in any design and the proposed design is no exception to this. One such enhancement would be to get

rid of the acyclic peer-to-peer topology constraint and extend the system to support general peer-to-peer topology. Also, along with general peer-to-peer, other interconnection topologies like hierarchical and hybrid may be implemented using SDN and their performances can be compared with the proposed system. Also, load balancing features can be introduced in the system by monitoring the traffic across each link. This may result in reconfiguration of paths. However, the feasibility of such an extension should be determined along with the incurred costs associated with traffic monitoring and reconfiguration.

The presented design spatially partitions the network in order to distribute control. However, control distribution can be done in various other ways. One such method would be to identify the tasks and divide them between a distributed set of controllers. Also, the proposed design assumes a fault-free system. However, all network elements run the risk of failure. So, fault tolerance may be incorporated to add robustness to the system. Additionally, identifying quality of service attributes relevant to the system and evaluating them in realistic settings may prove to be useful.

Moreover, the current system uses Openflow 1.0 which only supports IPv4 addresses. This imposes a constraint on the system as only 23 bits are available for the dz-expressions which in turn impose a restriction on the maximum allowable dz-expression size. This results in increase in false positives. So, in future, the same system can be extended to use IPv6 addresses with versions of Openflow supporting IPv6. The proposed distributed algorithms also provide means to use more optimized algorithms for route calculations within local sub-networks which can further improve the performance of the proposed design. So, the distributed framework can incorporate other centralized algorithms providing in-network filtering.

Finally, it should be noted that all tests have been conducted in a simulated network using Mininet. So, in future, experiments can be conducted using real infrastructure with real network elements. The evaluations done in this thesis were limited by the simulated network. In a realistic setting, many other performance metrics can be successfully analyzed. Also, synthetic data was used for the experiments which can be replaced with more meaningful real world data.

# Bibliography


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

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[3] J. A. Briones, B. Koldehofe, and K. Rothermel, "Adaptive Publish/Subscribe for Wireless Mesh Networks," *Studia Informatika Universalis*, vol. 7, no. 3, pp. 320–353, 2009.

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

[5] 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, pp. 2140–2153, Dec. 2011.

[6] M. A. Tariq, B. Koldehofe, G. G. Koch, and K. Rothermel, "Providing Probabilistic Latency Bounds for Dynamic Publish/Subscribe systems," in *KiVS*, pp. 155–166, 2009.

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

[8] Wikipedia, "Software-defined networking – wikipedia.." `http://en.wikipedia.org/wiki/Software-defined_networking`, [Accessed : October, 2013].

[9] 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*, MW4NG '12, (New York, NY, USA), pp. 3:1–3:6, ACM, 2012.

[10] G. B. Mishra, "Providing in-network content-based routing using OpenFlow," Master's thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, June 2013.

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Challenges for distributed event services: Scalability vs. expressiveness," in *Engineering Distributed Objects '99*, May 1999.

[12] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time CORBA event service," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, (New York, NY, USA), pp. 184–200, ACM, 1997.

[13] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The Information Bus: an architecture for extensible distributed systems," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, (New York, NY, USA), pp. 58–68, ACM, 1993.

[14] TIBCO Inc., "TIB/rendezvous. White Paper." `http://www.rv.tibco.com/`, 1999.

[15] G. Mühl, "Generic Constraints for Content-Based Publish/Subscribe," in *Proceedings of the 9th International Conference on Cooperative Information Systems*, CooplS '01, (London, UK), pp. 211–225, Springer-Verlag, 2001.

[16] G. Mühl, *Large-Scale Content-Based Publish-Subscribe Systems*. PhD thesis, TU Darmstadt, November 2002.

[17] P. R. Pietzuch, *A scalable event-based middleware*. PhD thesis, University of Cambridge, June 2004.

[18] IBM TJ Watson Research Center, "Gryphon : Publish/Subscribe over Public Networks." `http://researchweb.watson.ibm.com/gryphon/Gryphon`, 2001.

[19] A. K. Y. Cheung and H.-A. Jacobsen, "Load Balancing Content-Based Publish/Subscribe Systems," *ACM Trans. Comput. Syst.*, vol. 28, pp. 9:1–9:55, Dec. 2010.

[20] 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 *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, (Berlin, Heidelberg), pp. 458–470, Springer-Verlag, 2010.

[21] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. V. Steen, "Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks," in *IPTPS'06: the fifth International Workshop on Peer-to-Peer Systems*, 2006.

[22] 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.

[23] A. Majumder, N. Shrivastava, R. Rastogi, and A. Srinivasan, "Scalable Content-Based Routing in Pub/Sub Systems," in *INFOCOM*, 2009.

[24] O. Papaemmanouil, "SemCast: Semantic multicast for content-based data dissemination," in *ICDE*, pp. 242–253, 2005.

[25] 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 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, (Washington, DC, USA), pp. 133–142, IEEE Computer Society, 2002.

[26] SDNCentral, "SDN Use Cases." `http://www.sdncentral.com/sdn-use-cases/`, [Accessed : October, 2013].

[27] W. Segall and D. Arnold, "Elvin Has Left the Building: A Publish/Subscribe Notification Service with Quenching," in *Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia*, 1997.

[28] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, "Content Based Routing with Elvin4," in *Proceedings of AUUG2K*, 2000.

[29] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, (New York, NY, USA), pp. 53–61, ACM, 1999.

[30] G. Banavar, T. Ch, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," pp. 262–272, 1999.

[31] G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 827–850, Sept. 2001.

[32] Z. Jerzak and C. Fetzer, "Prefix forwarding for publish/subscribe," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, DEBS '07, (New York, NY, USA), pp. 238–249, ACM, 2007.

[33] Wikipedia, "Bloom filter - wikipedia." `http://en.wikipedia.org/wiki/Bloom_filter`, [Accessed : October, 2013].

[34] B. Koldehofe, F. Dürr, and M. A. Tariq, "Event-based Systems Meet Software-defined Networking," in *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pp. 649–671, John Wiley & Sons, Ltd., June 2013.

[35] Wikipedia, "Classless Inter-Domain Routing – wikipedia.." `http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing`, [Accessed : October, 2013].

[36] Wikipedia, "Open Shortest Path First – wikipedia.." `http://en.wikipedia.org/wiki/Open_Shortest_Path_First`, [Accessed : October, 2013].

[37] Wikipedia, "Breadth-first search - wikipedia.." `http://en.wikipedia.org/wiki/Breadth-first_search`, [Accessed : October, 2013].

[38] Mininet, "Mininet Network Simulator." `http://mininet.org/`, [Accessed : October, 2013].

[39] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, (New York, NY, USA), pp. 19:1–19:6, ACM, 2010.