UNIVERSITY OF OSLO
Department of Informatics

Publish/Subscribe for Large-Scale Social Interaction: Design, Analysis and Resource Provisioning

PhD Thesis

Vinay Jayarama Setty

March 9, 2015

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1595

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

"It is better to have an approximate answer to the right question than an exact answer to the wrong one."

John Wilder Tukey

Abstract

Publish/subscribe (pub/sub) is a popular communication paradigm in the design of large-scale distributed systems. We are witnessing an increasingly widespread use of pub/sub for a wide array of applications both in industry and academia. For instance, the pub/sub paradigm is used for RSS feed notifications, financial data dissemination and business process management. Pub/sub has also been used in social interaction message notifications such as in Spotify. Social network interactions have grown exponentially in recent years to the order of billions of notifications generated by millions of users every day. However, there are a number of critical challenges yet to be addressed to design a pub/sub system that can scale massively.

Pub/sub systems are generally deployed in centralized datacenters or using federated organizations of cooperatively managed servers. However, an increasingly higher number of pub/sub applications are being deployed in P2P environments due to their ability to provide scalable and robust decentralized solutions. The design of a system with a goal to support notifications at massive scale from social interactions has several challenges. For one, such a large-scale system has to possess a distinctively high number of desirable characteristics all at once in order to be a viable practical solution. However, we show that the existing state-of-the-art solutions provide only a subset of these characteristics. In this thesis, we propose PolderCast, a P2P topic-based pub/sub system that is fault-tolerant, robust, scalable and fast in terms of dissemination latency while attaining a low communication overhead. We do an extensive experimental analysis of PolderCast using Twitter and Facebook traces and show that

PolderCast performs well under realistic churn compared to the widely used pub/sub system Scribe.

Understanding the challenges faced by a real pub/sub system and getting insights from the workload it drives are critical to design a pub/sub system. Yet there is a serious lack of detailed study of a large-scale pub/sub system and its workload. In this thesis, we present an overview of a pub/sub system used to drive social interaction at Spotify. We then present a detailed analysis of traces from a real deployment of Spotify pub/sub. We further analyze the Twitter traces we collected via public APIs provided by Twitter. The analysis of these traces provides several interesting observations and conclusions which can benefit pub/sub designers.

Inspired by the peer-assisted solution used by Spotify to stream music, we explore a similar solution to provide a scalable dissemination of notification events to the users. The task of distributing the workload among user peers and datacenter servers prompts a fundamental problem: How to select a subset of the pub/sub workload to be served by datacenter servers in a manner to maximize satisfaction requirements of users under resource constraints? In this thesis we provide, to the best of our knowledge, the first formal treatment of the above problem by introducing two metrics that capture subscriber satisfaction in the presence of limited resources. This allows us to formulate the problem as two new flavors of maximum coverage optimization problems. Unfortunately, both variants of the problem prove to be NP-hard. By subsequently providing formal approximation bounds and heuristics, we show however, that efficient approximations can be attained. We validate our approach using real-world traces from Spotify and show that our solutions can be executed periodically in real-time in order to adapt to workload variations.

One of the fundamental challenges which remains to be addressed in deploying pub/sub systems on a datacenter or a cloud infrastructure is efficient and cost-effective resource allocation that would allow delivery of notifications to all subscribers. Specifically, the challenge is to answer the following three fundamental questions: Given a pub/sub workload, (1) what is the minimum amount of resources needed to meet satisfaction requirements of all the

subscribers, (2) what is a cost-effective way to allocate resources for the given workload, and (3) what is the cost of hosting it on a public Infrastructure-as-a-Service (IaaS) provider like Amazon EC2. We formulate the problem to address these questions and provide an efficient solution. We do an extensive evaluation of the solution using real traces from Twitter and Spotify. With evidence from the empirical results we show that our solution can be used as a tool to allocate resources on datacenters and cloud so as to minimize infrastructure costs.

Acknowledgements

This dissertation would not have been possible without the support of many people. Therefore, I believe that credit must be given to all those people who were instrumental in finishing this dissertation. I would like to take this opportunity to thank them.

First and foremost I want to thank my advisors, Roman Vitenberg and Maarten van Steen. I thank them for believing in me and providing me an opportunity to pursue PhD under their supervision. They proved to be excellent supervisors and mentors. I especially thank Roman for having regular meetings with me in which he patiently guided me and helped me in many aspects of research. I acquired many valuable qualities a researcher must possess from him. I also thank Maarten for finding time in his extremely busy schedule for providing me his invaluable feedback. I am impressed with Maarten's ability to quickly understand my ideas and provide great insights. I appreciate both Roman and Maarten for enduring my sloppy writing and silly mistakes during my PhD. In addition to correcting my writing patiently, they constantly challenged me to write better. I believe, as a result, my writing skills are significantly improved.

I am also thankful for the external reviewers Alejandro Buchmann and Ben Y. Zhao for accepting to review my thesis amid their busy schedule. They provided insightful and valuable feedback promptly. Special thanks to Stein Gjessing for coordinating the defense process to meet our tight schedules.

I must thank Spyros Voulgaris for working closely with me during the initial months of my PhD. He shared his office with me when I visited Amsterdam. That allowed me to learn many skills faster from him. I believe that his contribution

was crucial in realizing the PolderCast system.

I am very thankful to Spotify and many of its employees for allowing me to study their pub/sub system and providing access to their production data. I especially thank Gunnar Kreitz for realizing collaboration with Spotify. In addition to that, he actively participated in our discussions and encouraged me to focus on many theoretical aspects of the problems in this thesis. His contributions as a co-author in most of our papers were invaluable and important for completing this dissertation. I also thank Guido Urdaneta who was formerly a postdoctoral researcher at University of Oslo. I believe his critical feedback and contributions significantly improved the quality of the papers we wrote. I must thank Javier Ubillos, Pablo Barrera Gonzàlez, Staffan Gimåker, Tommie Gannert and Michael Goldmann for their help in many ways during my days at Spotify.

I am also grateful to all the faculty at ND group. I especially thank Frank Eliassen, Stein Gjessing and Michael Welzl for attending my talks and providing me their feedback on several occasions. I also thank all PhD students and postdoctoral researchers at ND group who were often the main audience in my conference rehearsal talks and they provided me useful feedback on my research. They were more than just colleagues who provided me a social life in Oslo. Special thanks to Abhishek Kumar Singh for being a quiet officemate and for tolerating my clutter. He also inspired me to bike more. I also thank Lucas Provensi for not minding me barging into his office whenever I needed someone to talk to. I also thank his lovely wife Lorena Provensi for hosting parties regularly at their place. I also want to thank Navneet Kumar Pandey, Narasimha Raghavan Veeraragavan and Safiqul Islam who I could count on whenever I needed some help. Special thanks to Amirhosein Taherkordi, for patiently answering my long list of questions on the PhD defense process. I also thank Kashif Sana Dar and Naeem Khademi for being good friends and helping me in many ways (such as guidance on Norwegian language courses etc).

I thank my dear friend Avishek Anand who introduced me to the wonderful world of research in Computer Science by encouraging me to apply for IMPRS scholarship at Max-Planck Institute. I also thank my friend Megha Khosla who is now Avishek's wife for cheering me up whenever I felt down. They have been

there for me during both good and bad times.

I must thank my parents Jayarama Setty and Ramarathnamma for their constant love and support. I wouldn't be what I am today if they didn't put my education ahead of everything else in their life. I am grateful for them for supporting my decision of moving to Europe for pursuing my higher education. I hope this PhD makes them proud. I also thank my two brothers Ramamurthy Setty and Ravi Kumar Setty and my lovely sister Shobha Ramesh and their families who understood and supported many of my decisions.

Last but not the least, I want to thank Tobias Tykvart for always being there for me and comforting me with constant love and support even when we were thousands of kilometers apart. I was especially more productive whenever he visited me in Oslo. I also thank him for his help in finding typos in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

1.1.1 Publish/Subscribe Systems

Publish/Subscribe (Pub/Sub) is a communication paradigm in which subscribers express their interest as a pattern of events to be notified. On the other hand, publishers generate events that are delivered to subscribers with matching interests. Pub/sub is regarded as a technology enabler for a loosely coupled form of interaction among many publishing data sources and many subscribing data sinks. Pub/sub systems provide *space decoupling*, in which subscribers and publishers need not know each other. Pub/sub systems are *non-blocking*, in other words while subscribers are consuming the produced events, producers can continue to produce more events independently. Optionally, in pub/sub systems *time decoupling* is provided, in that it is not necessary for the publisher of an event E to be present when E is delivered.

Pub/sub has a wide array of applications both in industry and academia. Examples from industry include: Google internal pub/sub system [Reumann, 2009], Tibco financial dissemination system [TIBCO] and Spotify pub/sub for social interaction [Setty et al., 2013]. Many pub/sub systems proposed in academia are listed in [Eugster et al., 2003; Kermarrec and Triantafillou, 2013]. Many applications report benefits from using this form of interaction, such as

online delivery of notifications due to social interaction, application integration [Reumann, 2009], financial data dissemination [TIBCO], RSS feed distribution and filtering [Liu et al., 2005] and [Petrovic et al., 2005], and business process management [Li et al., 2010]. As a result, many industry standards have adopted pub/sub as part of their interfaces. Examples of such standards include WS Notifications, WS Eventing, and the Active Message Queuing Protocol.

In pub/sub systems, the subscribers are usually interested in specific information instead of the whole data produced by the publishers. Whenever a publisher generates some data, it is delivered to the relevant subscribers through an event notification. The different ways of specifying the events of interest have led to several subscription schemes [Eugster et al., 2003]. The two most widely used pub/sub schemes are described below.

Topic-Based Publish/Subscribe (TBPS): In a TBPS system, subscriptions are formed using a set of predefined "topics". In such a system, subscribers will receive all messages published for the topics to which they subscribe, and all subscribers subscribing to the same topic at the same time will receive the same messages. The publisher is responsible for generating the messages relevant for the predefined set of topics. In social interaction systems, users subscribe to the events generated by their friends and celebrities who can be represented as topics. Hence, TBPS systems are a good match for the communication patterns in social interaction systems. Hence, in this thesis, we focus on various research problems in designing scalable TBPS systems specifically in the context of social interaction systems.

Content-Based Publish/Subscribe (CBPS): In a CBPS system, the subscriptions are expressed as boolean predicates operating on attributes and values (for example: "Stock = 'AAPL' and value > 95 and value < 98 and daily-change > 2"). In a CBPS the actual content of the generated event is matched against the subscriptions to deliver the publication event to the interested subscribers. While in CBPS the subscriptions can express queries with diverse selectivity, matching the publications against the subscriptions is considered to be more expensive than TBPS. In social

interaction systems, the users are generally interested in every message generated for the topics of their interest. Hence, the rich expressiveness provided by the CBPS is expensive with very little benefit. Therefore, using CBPS for matching and delivering publications events in a social interaction system may be wasteful. There are several CBPS systems proposed in the literature [Banavar et al., 1999; Carzaniga et al., 2001; Cugola et al., 2001; Segall and Arnold, 1997]. Though there are several research problems yet to be addressed in CBPS, they are beyond the scope of this thesis.

1.1.2 Social Interaction Systems

Recently, online social networks have gained a lot of popularity. There are billions of users actively using popular social networking services regularly. One of the fundamental features of social networks is the online social interaction among its users which includes user activities such as status and multimedia sharing with friends and followers. In the recent years there is a significant increase in social interaction among social networking users. For example, Twitter users generate 400 million tweets every day which amounts to more than 6TB of tweet data [Krikorian, 2010]. Social interaction is not limited to popular social networking services such as Facebook and Twitter. It has been introduced in music streaming services such as Spotify as well [Setty et al., 2013]. The number of notifications due to social interactions among Spotify users is in the order of 2 billion per day. Such large-scale notifications require a scalable notification system.

Social Interaction Among Twitter Users: Twitter is a popular online social networking service. It is also referred to as a microblogging service, because it limits the messages to a 140 character text called "Tweet". As of 2013 Twitter has more than 500 million registered users with more than 200 million of them active daily. More than 400 million tweets per day are generated with an average tweet rate of 4000 tweets per second and a daily peak around 7000 tweets per second. Tweet rate increases to 12000 tweets per second during special large events such as a celebrity death or football world cup. The generated tweets when delivered

to the interested Twitter users across the world amounts to more than 30 billion notification deliveries per day [Krikorian, 2013]. Considering that the average tweet size is about 200 bytes [Krikorian, 2010], this amounts to more than 6TB of just plain Twitter text being delivered to the users every day. The notifications due to social interaction among Twitter users is truly large-scale. There are several instances of Twitter service outages reported. There are examples of Twitter outages caused by a surge in traffic generated by a celebrity's tweet[1].

Twitter allows users to follow any other Twitter user (with a publicly available profile) without requiring the other user to follow them back. In other words Twitter allows "unidirectional" social relationships. All the followers of a Twitter user receive the tweet from the followee. An example of a Twitter graph with user-follow relationships is shown in Figure 1.1. For example, user A follows other Twitter users B, C, D, E, G. Conversely, user A is followed by users C and E. This follow relationship can be easily turned into pub/sub subscriptions, in which user A is a subscriber subscribing to topics {B, C, D, E, G} and user A can also be a topic/publisher being subscribed by users {C, E}. There are several research works in recent years providing detailed studies of social graphs and interaction patterns of Twitter users [Kwak et al., 2010; Mislove et al., 2007]. In addition to public tweets, Twitter has recently allowed users to share private tweets which can be seen only by the intended recipients. However, social interaction in Twitter is still dominated by public Tweets. Recently, Twitter allowed tweets to include multimedia content as well, making the large-scale tweets more data intensive.

Social Interaction Among Facebook Users: Facebook is yet another popular online social networking service. As of December 2013, Facebook has 1.28 billion registered users. 1.23 billion users are active at least once a month and on average 757 million users are active every day[2]. As of May 2013, 4.75 billion pieces of content are being shared on Facebook every day. This makes Facebook the largest online social network to date.

Facebook social relations are normally bidirectional, i.e. two users are friends

[1] See e.g. http://www.bbc.co.uk/news/blogs-trending-26410106
[2] http://investor.fb.com/releasedetail.cfm?ReleaseID=821954

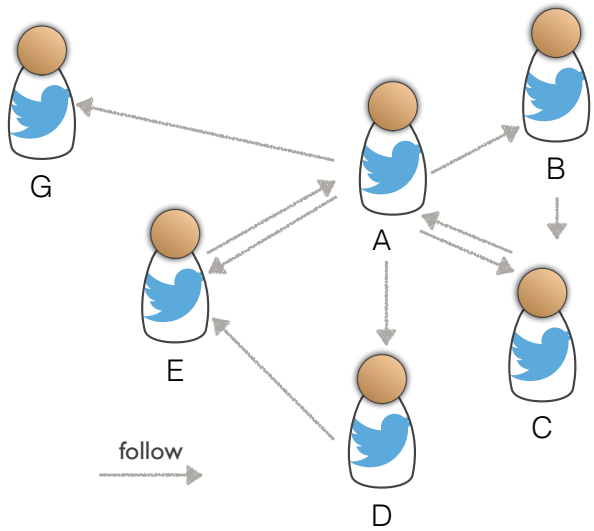

Figure 1.1: Twitter Social Graph Example

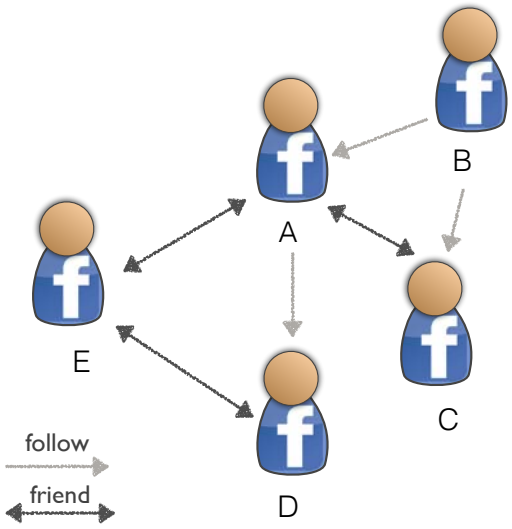

Figure 1.2: Facebook Social Graph Example

only if they mutually accept each other as friends. In addition to friend relationships Facebook also allows unidirectional following similar to Twitter. An example of a social relationship in Facebook is shown in Figure 1.2. User A has a

friend relationship with users C and E. In addition, user A follows user D, even though user D is not a friend of user A. Regardless of the relationship type, we can break this social graph into the pub/sub subscriptions. For example, in Figure 1.2, user A is a subscriber following topics {C, D, E} and user A is also a topic with subscribers {B, C, E}. Social interaction in Facebook involves status sharing, multimedia content sharing, private messaging etc. which are notified to intended friends and followers.

Social Interaction Among Spotify Users: Spotify is a successful on-demand music streaming service that provides access to over 25 million tracks to its users residing in more than 55 countries and it has 40 million registered users as of May 2014. Even though Spotify is mostly known for its music streaming service, it also provides one of the most engaging features: its ability to facilitate sharing and following of various music activities among its users in real-time. Notifications through Spotify social interaction are also large-scale. The daily notifications is in the order of over one billion, which amounts to 2TB of data delivered to the users.

Spotify users can discover and follow other Spotify users, alternatively, they can import users from their associated Facebook account and follow them. Users can follow music artists registered with Spotify. The social interaction among Spotify users involves sharing music activities with their followers. For example, a user continuously streaming music can share this activity with the followers of that user in real-time. Other social interactions include sharing, creating and updating the lists of music tracks called *playlists*, sharing and recommending individual music albums or tracks. Thanks to artist verification, followers can also receive notifications about music activities of real artists. A brief overview of Spotify social interaction is shown in Figure 1.3. A more detailed study of Spotify social interaction is conducted in Chapter 4.

1.1.3 Pub/Sub for Social interaction

Our goal in this thesis is to apply the pub/sub communication model to drive massive scale social interactions. We can map the communication model used in social interaction systems with the TBPS model in the following way: A user is a

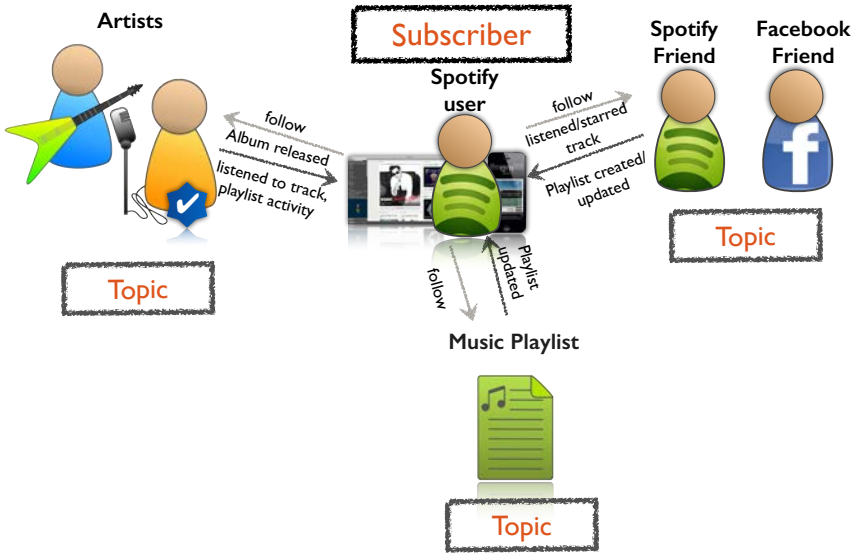

Figure 1.3: Spotify Pub/Sub for Social Interaction

"subscriber" of his friends and the other users he follows, who are represented as "topics". The notifications due to social interactions can be viewed as publications generated for user topics via "publishers".

The users can generate social interaction messages without knowing any details of how to disseminate the messages to their friends and followers. It is the job of the middleware driving the social interaction to route and deliver the notifications to the intended recipients. Such a communication pattern is widely used in pub/sub systems and they have proven to be scalable and robust. Hence, the use of pub/sub for social interaction is justified.

1.2 Motivational Scenarios and Challenges

Consider a social interaction application, such as in Spotify as shown in Figure 1.3. Millions of users in Spotify stream music at any given point in time. The users are located in different parts of the world. The social-interaction system of Spotify enables the delivery of music activity of users and artists to their friends and followers in real-time. In Spotify, some artists have millions of followers. For

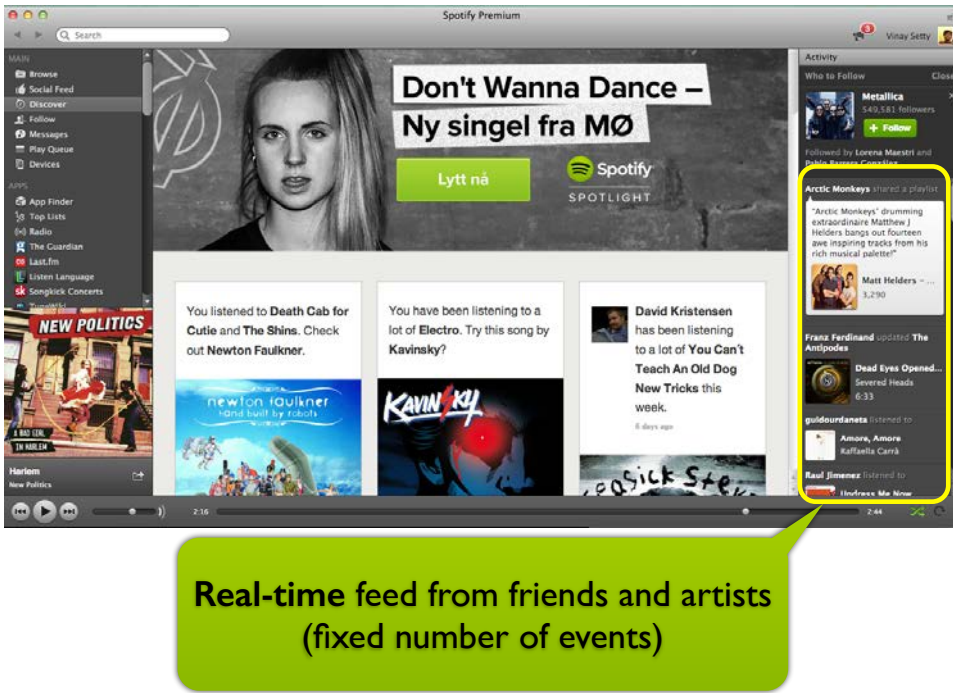

Figure 1.4: Spotify social interaction real-time notifications

example, popular artist Rihanna has around 4.3 million followers. When Rihanna listens to music tracks or creates and updates playlists on Spotify, it triggers around 4.3 million notifications. This triggers a sudden surge in pub/sub traffic. Another example from Twitter shows that when special events such as the Academy Awards (The Oscars) happen, it may result in a Twitter outage lasting up to 30 minutes. Hence, it is critical to design scalable architectures to deal with unexpected surges in pub/sub traffic.

Now consider the same scenario from a subscriber's perspective. A subscriber of Rihanna may also be subscribed to many other celebrities and receive an overwhelming number of events. Given the limited capacity of human recipients to process such events, that subscriber may simply ignore an event from Rihanna. Even if the subscriber is able to process all the events, the applications such as the Spotify client GUI (see Figure 1.4), limit the number of notifications a user can see at any given point in time. Based on these observations it may be

useful to introduce a *configurable threshold delivery rate* (τ) for each subscriber. It is wasteful to deliver the notifications generated by Rihanna to the users who will ignore them or may not be able to see them because they are not included in the τ events. The challenge here is to utilize the available limited resources of the pub/sub infrastructure for delivering a subset of events to the users who are most likely to benefit from them.

From the pub/sub service provider's perspective, the limited capacity of the dedicated pub/sub infrastructure for the notification delivery could potentially result in many issues. Service outages when there is a sudden surge in traffic is one such issue that is critical to avoid. In such scenarios, a pub/sub service with relaxed QoS guarantees is preferred over a service outage that could potentially last for hours. In this regard, the utilization of the limited capacity infrastructure can be maximized in the following ways: (1) maximize the number of subscribers receiving a minimum event delivery rate of τ, (2) maximize the cumulative event delivery rate for all the subscribers, while striving to maintain an event delivery rate up to τ for individual subscribers. In both cases, complementary mechanisms such as notification delivery through P2P (peer-to-peer) communication can be utilized to ensure that every subscriber receives a minimum of τ events. To this end, peer-assisted pub/sub systems can provide a scalable pub/sub service that continues to function even when the resources of a dedicated infrastructure are saturated.

P2P delivery can also be useful for delivering the events beyond the required τ events at any given point in time for the subscribers needing them. For example, Facebook users see a fixed number of events in real-time on their ticker window[3] by default. However, users can also optionally retrieve more events beyond the default set of events by scrolling down. Utilizing resources from the peers to deliver the events in such scenarios can be cost-effective.

An alternative way to address a sudden surge in traffic is to over-provision the resources as currently done by most existing systems. However, the cost of resources can be minimized by estimating and deploying the optimal amount of

[3]Can be seen at top-right corner of the Facebook page, for more information see: https://www.facebook.com/help/255898821192992/

resources required to handle the surge. In order to realize this, it is critical to understand the amount of resources needed in terms of number of servers and amount of bandwidth required so as to ensure a minimum event delivery rate of τ for all the subscribers. However, there are no such tools to estimate and allocate resources for such pub/sub systems. The challenge here is estimating the minimum amount of resources required given a dynamic and large-scale workload such as social interaction notifications.

From these scenarios first of all we learn that it is critical to analyze pub/sub workloads. Without them, it is difficult to prepare a system to handle unexpected surges in traffic. In addition, designing systems which can naturally scale despite sudden surges in traffic and yet remain robust is essential. We also note here that delivering only those notifications that are likely to benefit subscribers using dedicated infrastructure is a key to maximize the utilization of pub/sub infrastructure resources. Finally, designing tools to estimate and allocate the minimum amount of required resources to handle unpredictable pub/sub workload in datacenters and cloud are needed.

1.3 Research Problems Addressed

In this section we elaborate the research problems considered in this thesis inspired by the motivational scenarios identified in the Section 1.2.

1.3.1 Research problem 1: Designing a Scalable and Robust P2P Architecture for TBPS

Consider the scenario again where a celebrity like Rihanna generates a notification which is to be delivered to millions of her followers which could result in a sudden surge of pub/sub traffic. Given that there are millions of users interested in the same notification, it is feasible and useful to distribute such a notification using a P2P network. This technique saves precious resources such as CPU, memory and bandwidth used by the pub/sub infrastructure.

A number of P2P TBPS systems have been proposed over the last decade

[Baehni et al., 2004; Castro et al., 2002; Chockler et al., 2007b; Girdzijauskas et al., 2010; Matos et al., 2010; Rahimian et al., 2011; Wong and Guha, 2008; Zhuang et al., 2001]. These systems build a decentralized infrastructure in which the nodes are first dynamically organized into an application-level overlay network. Overlays are logical links connecting the nodes built on top of physical networks. The resulting network is subsequently used for event routing. In a P2P system, nodes may constantly join and leave the overlay network. If the pub/sub overlay is disrupted while a publication is being routed to the subscribers it may not reach many subscriber nodes. Hence, there should be overlay maintenance techniques at each node to rebuild the overlay when failures occur. To be able to support large-scale applications, such as social interaction, the overlay maintenance should be lightweight. The following characteristics are desirable for pub/sub overlays:

Reliable publication delivery: Correct delivery of all publications, i.e. absence of false negatives or deterministic 100% hit-ratio guarantees in a failure-free run,

Churn handling: Maximizing the number of nodes that receive the generated publication events (i.e. hit-ratio), even when the nodes are constantly leaving and joining,

Convergence Speed: Fast recovery at the end of a churn period and mending of the overlay so as to achieve 100% hit-ratio,

Low node degree: Low indegree and outdegree of overlay nodes,

Topic-Connectivity: Relay-free routing, which means that only subscribers interested in a topic are involved in routing events for that topic. Such routing avoids the need for relay nodes that forward publication messages without being interested in their content,

Scalability: Scalable with the number of nodes, topics, number of nodes interested in a topic, and number of topics a node is interested in,

Effective dissemination: Fast with as little duplicate delivery as possible, and fair distribution of load due to routing and processing, and

Low overhead: Lightweight overlay maintenance.

The design challenge is amplified due to a number of trade-offs:

1. Aiming to achieve low or fixed node degree has a trade-off with relay-free routing
2. Ensuring robustness under churn with minimal duplicate messages. For example, using techniques such as flooding of publications in the network may increase robustness but may also introduce significant amount of duplicate messages
3. Scalability and precise delivery with few false negatives and false positives are fundamentally at odds with each other

It is difficult to balance the above mentioned trade-offs. The P2P systems proposed in the literature provide only a subset of the above listed characteristics. It is also not clear if any of those systems try to balance the above mentioned trade-offs. In this regard, there is a need to analyze existing approaches both analytically and empirically with respect to the above characteristics. In addition, there is a need for a P2P architecture for TBPS that takes all the above factors into account and harmonizes them.

1.3.2 Research Problem 2: Understanding and Characterizing Large-Scale Publish/Subscribe for Social Interaction

Use of pub/sub in real-world social interaction systems In order for researchers to address the relevant problems faced by real-world pub/sub systems, understanding the working of real-world pub/sub systems and identifying their bottlenecks is crucial. Unfortunately, there is limited knowledge about pub/sub deployed in real industrial large-scale settings.

Workload characterization of pub/sub for social interaction In addition to understanding the architecture of the real-world pub/sub systems, it is also important to understand real-world pub/sub workloads. Often design

decisions are affected by workload characteristics. Moreover, there exist only a few characterizations of subscriptions and synthetic workload generators such as [Yu et al., 2009] for pub/sub systems.

There are several characteristics of a pub/sub workload which are typically used in pub/sub system evaluation. Studying these characteristics for real-world pub/sub workloads is an unexplored topic. Any pub/sub workload characterization must describe the following metrics:

Subscription size distribution: Distribution of number of topics each subscriber subscribes to

Topic popularity distribution: Distribution of number of subscribers to each topic

Publication rate distribution: Distribution of number of publication events generated for each topic in a given unit of time

Normalized notification rate distribution: Distribution of percentage of total notifications generated received by each subscriber in a given unit of time

Temporal variation of subscriptions: Temporal patterns in new subscriptions requests

Temporal variation of unsubscriptions: Temporal patterns in unsubscription requests

Temporal variation of publications: Temporal patterns in publication event generation

Gaining insights from real pub/sub systems driving large-scale social interaction and real traces from such systems is crucial for the research community to understand real problems and propose practical solutions.

1.3.3 Research problem 3: Defining and Meeting Subscriber Satisfaction Metrics in Publish/Subscribe

Subscriber satisfaction in pub/sub for social interaction: It is a known fact that notifications due to social interaction have mostly human recipients[4]. Notifications due to social interaction are also known to be overwhelming for the users. There are studies quantifying information overload in social networks [Gomez-Rodriguez et al., 2014]. Gomez-Rodriguez et al. also show that each user has a limited capacity to process events. When the user is delivered notifications at a rate beyond this threshold, the user starts to ignore them. This implies that not all notifications are critical to be delivered in order to guarantee user satisfaction. Motivated by such scenarios, it is worth introducing the concept of *subscriber satisfaction requirements* to deliver events at a configurable threshold delivery rate (τ) for each subscriber. This observation can be exploited to distinguish between a subset of the workload which is sufficient to meet subscriber satisfaction, and the rest of the workload.

The concept of subscriber satisfaction requirements helps in selecting the part of the pub/sub workload that maximizes the utilization of limited resources of pub/sub infrastructure. In addition, such techniques can also be used to offload the workload that cannot be handled by the dedicated pub/sub infrastructure to relatively cheaper resources such as P2P networks. The task of distributing the workload among user peers and datacenter servers prompts a fundamental problem: How to select a subset of a pub/sub workload to be served by datacenter servers in a manner that meets satisfaction requirements of users under resource constraints?

Selecting pub/sub workload to maximize satisfaction metrics: As shown in Figure 1.5, in a typical pub/sub system, generally there are publishers generating publications and there is a middleware hosted in a datacenter or cloud which is responsible for matching and delivering the notifications to the interested subscribers. The goal of a pub/sub infrastructure is to maintain

[4]Even though social notifications can be potentially fed into applications and services as well for example via Twitter public APIs, in this thesis our focus is on human end users.

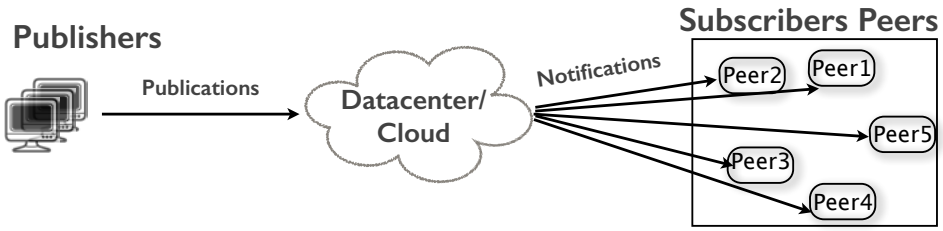

Figure 1.5: Typical pub/sub architecture for social interaction

subscriptions in-memory, match incoming publications against the subscriptions and deliver the notifications to the subscribers.

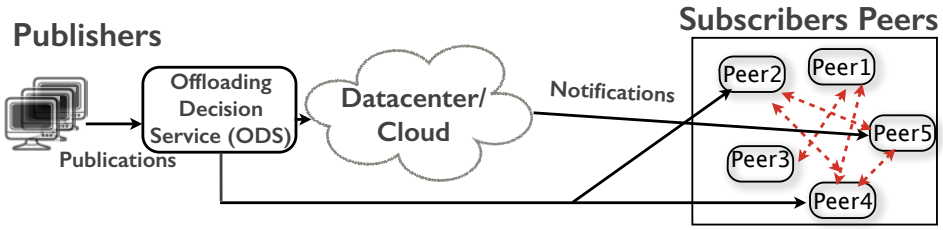

Figure 1.6: Offloading workload to an external system like P2P network in pub/sub systems

Typically pub/sub infrastructures have limited resources in terms of memory, CPU power and network bandwidth capacities etc. In such a scenario, it is useful to devise techniques to select a cost-effective subset of the pub/sub workload that meets the satisfaction requirements of users and allocate it to the dedicated pub/sub infrastructure hosted in datacenters or the cloud. If not all subscribers can be satisfied, the notifications that are required to meet satisfaction requirements of all the users can be delivered using cheaper dissemination solutions such as a P2P network. To achieve this, a component called *Offloading Decision Service* (ODS) shown in Figure 1.6, can be used to distribute the workload between the dedicated infrastructure and the P2P network. Pub/sub designers can also use ODS to estimate the number of subscribers who can be satisfied using a dedicated pub/sub infrastructure with limited capacity.

To the best of our knowledge there are no techniques available in the literature

to realize the ODS tasks described above.

1.3.4 Research problem 4: Resource Provisioning for Scalable Publish/Subscribe

Enterprises wanting to deploy pub/sub either on their in-house datacenters or public cloud infrastructures such as Amazon EC2 or Microsoft Azure face a fundamental question: what is the minimum amount of resources in terms of number of servers and total network bandwidth needed to deliver events to all the subscribers. While answering such a question is critical, it is not a trivial task.

Many cloud providers charge their customers separately for the number of servers with a certain capacity limit (such as CPU and memory) and total bandwidth consumption. Hence, a cost-effective deployment of pub/sub systems would require us to minimize both. Trying to minimize both at the same time may not be feasible, since they are at odds with each other as explained in Section 7.1.1.

The problem of minimizing cost of resources by balancing the trade-off mentioned above is computationally hard. If the solution is also required to meet satisfaction requirements of all subscribers it makes the problem even more difficult. Moreover, such a problem has never been solved before.

1.4 Contributions

This thesis advances the pub/sub research in the following ways:

- We analyze the existing P2P TBPS systems and propose PolderCast–a fast, robust and scalable P2P TBPS system. PolderCast harmonizes a number of desirable characteristics that a pub/sub overlay needs to possess by balancing trade-offs between them.

- We provide a case study of Spotify pub/sub system deployed for driving large-scale social interaction among its users and analyze a production

workload obtained from it. In addition, we also characterize a large-scale Twitter interaction trace collected by us.

- We introduce a novel concept of subscriber satisfaction requirements. Then we formulate a novel set of problems to meet satisfaction requirements of subscribers under resource constraints. By solving these problems using efficient heuristics, we provide a way to offload a subset of pub/sub workload to cheaper infrastructures such as P2P networks.

- We provide an efficient technique for cost-effective resource allocation to deploy a TBPS system on datacenter or cloud infrastructures. We do this while meeting satisfaction requirements of all subscribers.

1.5 Research Methodology

The research ideas proposed in this dissertation were validated by a combination of experimental and formal methods. In this section, we list the most important decisions taken with respect to research methodologies used in this thesis. A detailed rationale behind the choice of methodologies listed below is provided in Section 8.2.

Even though the ideas were validated through simulations, they were conducted under realistic settings using traces collected from real systems. For example, the performance of the PolderCast system was analyzed using extensive simulations driven by subscription workloads from Twitter and Facebook. The simulation settings were made more realistic by modeling churn using Skype traces [Guha and Daswani, 2005] and latency in P2P communication using the King dataset [Gummadi et al., 2002]. We also evaluated the algorithms proposed for the subscriber satisfaction and resource allocation problems using Spotify and Twitter social interaction traces.

The formal analyses performed in this thesis were instrumental in many design decisions taken. For example, NP-Hardness analysis of the problems proposed in this thesis (Sections 6.2.1, 6.2.2 and 7.1.4) proved that they are computationally hard to solve and hence developing efficient heuristics were

necessary. In another example, theoretically proving that the objective function is submodular in Section 6.4, resulted in the performance optimization of the solution. Deriving bounds on the solutions also helped us to perform relative comparison of the results we obtained.

Finally, many problems in this thesis were inspired by the interesting observations obtained by analyzing the real pub/sub systems and their workloads. For example, analysis of the pub/sub workload from Spotify and Twitter provided motivation for many problems in this thesis.

1.6 Roadmap

The contributions described in Section 1.4 are organized into chapters in the rest of this thesis as follows: In Chapter 2, we study the state-of-the-art techniques that are relevant for design, analysis and resource provisioning in pub/sub systems for social interaction. In Chapter 3, we analyze existing P2P topic-based pub/sub systems with a mini survey considering various characteristics of the P2P TBPS overlays. Further, we propose a design and experimental evaluation of a gossip-based P2P TBPS system called PolderCast which tries to balance between several conflicting overlay characteristics. A case study of a real-world pub/sub system used for driving social interaction at Spotify is given in Chapter 4. In Chapter 5, we provide a detailed analysis of pub/sub workload traces from Spotify pub/sub and Twitter. Chapter 6 is dedicated to defining novel satisfaction metrics for the subscribers and formulating and solving a number of problems of efficiently allocating workload to meet satisfaction metrics of the users. In Chapter 7, we define, analyze and solve the problems of efficiently and cost-effectively allocating resources for pub/sub. Finally, in Chapter 8, we summarize our most significant observations and conclusions. In the same chapter, we also provide an overview of promising future directions for the various research ideas presented in this thesis.

Chapter 2

Related Work

In this chapter, we explore various flavors and different architectures for pub/sub proposed in the literature and analyze their strengths and limitations in the context of this thesis. As mentioned in Section 1.3.2, analyses of real-world pub/sub systems and workloads are useful for designing scalable pub/sub systems and yet there are no such studies. In this chapter, we identify this gap in the literature. We also provide a brief survey of content-filtering and ranking techniques that are applicable for filtering events in pub/sub systems. In Section 2.6, we present an overview of state-of-the-art resource provisioning techniques. Finally, we explore the theoretical problems in the literature relevant to the subscriber satisfaction and the resource provisioning problems.

2.1 Subscription and Publication Schemes in Pub/Sub Systems

One of the main distinguishing properties of pub/sub systems is the degree of expressiveness of subscriptions they provide. The two widely used variations of subscription and publication schemes are topic-based and content-based. In this section, we list and classify the pub/sub systems following these two schemes. We also study their strengths and weaknesses in the context of pub/sub for large-scale social interaction.

2.1.1 Topic-Based Pub/Sub (TBPS) Systems

Multicast overlay per topic: In topic-based pub/sub (TBPS) systems the subscriptions are expressed as discrete topics. The subscribers are the recipients of all the events published for the topics they subscribe to. In addition, every publication generated for a topic needs to be notified to every subscriber of that topic. This property is very similar to the concept of application-level multicast, where the messages are efficiently delivered from the source to a group of nodes interested in the same messages. One way to implement TBPS systems is to build an application-level multicast network for each topic in the system. Such a technique optimizes the notification delivery to the individual topics. Two of the well-known works which build a P2P pub/sub system using the concept of application-level multicast are Scribe [Castro et al., 2002] and Bayeux [Zhuang et al., 2001]. Tibco Rendezvous [TIBCO] is known to use a similar technique but it relies on multicast protocols provided by the underlying network to deliver events for each topic independently.

Although Scribe and Bayeux optimize the delivery of events for individual topics, they are not optimized for scenarios with multiple topics, where each node subscribes to many topics. Hence, such systems do not scale for social interaction workloads typically consisting of millions of topics and subscribers. These systems are also not desirable for the pub/sub workloads with skewed popularity in topics. For example, in most social interaction systems the popularity of the topics follows a power law distribution. Hence, there are always a few topics that are extremely popular and the multicast trees for these topics would be overloaded. In addition to that, both these systems have a single point of contact known as *rendezvous node* for each topic to route the subscriptions and publications. Rendezvous nodes can become bottlenecks for popular topics and are prone to failures making these systems less robust as well. In Section 3.5, we show that Scribe is less robust under practical scenarios with heavy churn. Although using a multicast tree per topic has the limitations described above, there are some advantages as well: (1) duplication of publication events can be avoided, (2) node degree can be fixed. However, they still fail to meet all the requirements mentioned in Section 1.3.1.

Overlays with shared links: Topic-based pub/sub systems can scale well when they exploit the correlation between the subscriptions. For example, in pub/sub systems built using unstructured overlays the subscribers sharing the same topics in their subscriptions can establish a connection between them. Such connections could be shared to exchange the publication notifications that are of mutual interest between them. Building an overlay to optimally share and minimize the number of connections is known to be a computationally hard problem and a number of efficient algorithms are proposed with theoretical guarantees [Chen et al., 2010, 2011, 2012; Chockler et al., 2007a; Onus and Richa, 2010, 2011]. There are a number of other TBPS systems that exploit correlation in subscriptions to build scalable P2P overlays. SpiderCast [Chockler et al., 2007b], StAN [Matos et al., 2010], Vitis [Rahimian et al., 2011] are well-known systems in this regard. While SpiderCast and StAN strive to build and maintain scalable overlays for TBPS, it is not clear how to disseminate publication events on top of the overlay. SpiderCast and StAN are also not designed to deal with node failures and churn. While Vitis provides a dissemination protocol for the publication events and a mechanism to deal with node churn, it contains potential bottlenecks in the form of rendezvous nodes and gateways used for disseminating events.

There are more TBPS systems proposed in the literature. For example, TERA [Baldoni et al., 2007a] builds clusters of subscribers and publishers related to the same topic using a P2P clustering protocol. The notifications are routed via inter-cluster routing and disseminated in an epidemic manner within each cluster. This approach is shown to scale well for disseminating events. However, it still relies on per topic access points similar to rendezvous nodes in Scribe and Vitis to locate the clusters. We return to a more detailed analysis and comparison of these systems see Chapter 3.

TBPS in the industry The concept of topic-based subscriptions is not new to the industry. Well-known notification systems have been categorizing the events into predefined topics [ActiveMQ; JMS]. There are also several systems explicitly marketed as TBPS systems that are popular in the industry. For

example, Tibco Rendezvous [TIBCO] is used for financial data dissemination. Kafka [Kreps et al., 2011] is used in the industry to stream data in the form of topics, across datacenter servers. Kafka is designed for real-time processing of stream data typically consisting of streams of user logs and social media. As mentioned in Section 1.1.2, the Spotify pub/sub system follows the TBPS model as well for social notification delivery. A detailed architecture of Spotify pub/sub is provided in Section 4.2.1.

2.1.2 Content-Based Pub/Sub (CBPS) Systems

CBPS systems are considered to be more expressive than TBPS systems with respect to defining subscriptions and publications. Increased expressiveness comes with a higher cost in matching and requires building complex structures. For example, matching a publication with complicated subscriptions such as "Stock = 'AAPL' and value > 95 and value < 98 and daily-change > 2", would require an indexing structure to efficiently determine if the incoming publications fall within the subscription range. Without an indexing structure, checking attribute and value predicates of individual subscriptions against all incoming publications could be very expensive. The scalability of CBPS systems with millions of subscribers is unclear. Hence, CBPS systems are not desirable for social interaction systems. In this section, we explore the CBPS systems in the literature and elaborate the reasons why they are not suitable for large-scale social interaction.

In CBPS systems, since subscriptions consist of several attributes, matching them with generated publications is expensive. One of the challenges in CBPS systems is to minimize the overhead of matching subscriptions and events. A number of research works are dedicated towards speeding up the matching process. In Gryphon, the subscriptions are stored in the form of a tree. Each level in the tree represents an attribute in the subscription. Each branch in the tree represents a different value of that attribute, and the leaf nodes represent the complete subscriptions. The subscriptions sharing the same attribute and values fall under the same sub-tree. This technique speeds up the subscription matching. Other works that improve the efficiency of event matching include

[Aguilera et al., 1999; Fabret et al., 2001]. [Sadoghi and Jacobsen, 2011] builds an indexing structure called BE-Tree to speed up the event matching especially in a higher dimensional space.

The matching in TBPS systems is straightforward and faster than matching in CBPS systems because of the predefined topics. Hence, social interaction systems implemented using TBPS systems do not benefit from these expensive but sophisticated matching and indexing techniques. There are also efforts to speed up the subscription matching by using dedicated hardware such as FPGA [Sadoghi et al., 2010, 2012]. However, there are no studies to demonstrate their benefits in the TBPS systems. Using dedicated hardware to improve performance is orthogonal to the problems considered in this thesis.

Some CBPS systems avoid explicit matching of subscriptions and publications. They direct subscriptions and the corresponding matching publications to a rendezvous node. The rendezvous nodes are determined using a hash function. Hermes [Pietzuch and Bacon, 2002] is an example of CBPS system proposed in the literature which uses such a technique. One of the distinctive characteristics of Hermes is that it is a type-based pub/sub system. In a type-based system, both events and subscriptions have types that define which attributes they involve. Each subscription and publication event is hashed to obtain a type and routed to a rendezvous node using an underlying DHT (Distributed Hash Table) structure responsible for that type. Rebecca [Terpstra et al., 2003], similar to Hermes builds the overlay using a Chord DHT [Stoica et al., 2001]. However, the event dissemination is scoped flooding, i.e. the events are forwarded to all the neighbors with matching subscriptions at each node. Such a dissemination algorithm in the worst-case can have complexity $O(N)$ where N is the total number of nodes in the system.

In summary, even though CBPS systems provide better expressiveness compared to TBPS systems, they are not known to scale to millions of subscribers because of the expensive event matching techniques they use. On the other hand in social interaction systems, the events need to be delivered for every subscriber of a topic eliminating the need for complex event matching techniques. Finally, to the best of our knowledge, none of the CBPS systems proposed in the

literature are used as engines to drive large-scale social notification delivery.

2.2 Publish/Subscribe System Architectures

Traditional pub/sub implementations are either centralized or based on a federated organization of cooperatively managed servers. However, a number of pub/sub applications are being deployed in P2P environments as well. Hybrid peer-assisted architectures have also been proposed. In this section we discuss the various advantages and disadvantages of these architectures.

2.2.1 Broker-Based Architectures

In broker-based architectures for pub/sub generally there are a number of dedicated servers called brokers responsible for managing subscriptions, matching, routing and delivering publications to subscribers. Broker-based architectures can be either centralized or decentralized. While decentralized architectures improve scalability, they introduce unique challenges. The brokers are generally hosted on dedicated infrastructures such as datacenters or clouds. The construction and maintenance of broker topologies is often done manually in many systems [Carzaniga et al., 2001; Cugola et al., 2001; Jacobsen et al., 2010]. Broker-based architectures can be robust, dynamic and adaptive to the pub/sub workload if the broker overlays are self-organizing. In this section, we list and classify the different broker architectures used for pub/sub systems and identify their strengths and weaknesses.

Centralized broker-based architectures A centralized architecture is driven by a single physical or a logical broker. A logical broker may be partitioned into several physical subbrokers, each subbroker is responsible for independently handling a part of the work assigned using a deterministic work partitioner. Message queuing systems such as Oracle Advanced Queuing [OracleAQ] rely on centralized architectures with a single physical broker to store and forward the messages from publishers to subscribers. Several messaging solutions supporting Java Message Service (JMS) APIs such as Apache

ActiveMQ [ActiveMQ] are designed to operate on centralized broker architectures as well.

Decentralized broker topologies In many broker-based CBPS systems such as Siena [Carzaniga et al., 2001], JEDI [Cugola et al., 2001] and PADRES [Jacobsen et al., 2010], the brokers are organized as a tree or a graph topology. In such topologies, a naive way of delivering publication notifications to the subscribers is to flood the publication events from the publishers to the subscribers as done in [TIBCO]. The expensive flooding of publication events can be avoided, if a routing path between the publishers and subscribers is established. In order to achieve this each broker has to maintain a routing table which directs events to the next-hop broker to efficiently route it to the subscribers. In this regard, there are two main routing techniques proposed in the literature: advertisement-based and subscription-based [Eugster et al., 2003].

Advertisement-Based Routing: In advertisement-based routing the publishers advertise the attribute ranges in which they will generate future publications. The advertisements are flooded in the network. When subscribers join the network the subscriptions are routed back towards the publishers following the reverse path of the matching advertisements. The path obtained this way is used for routing publication notifications from the publishers to the subscribers.

Subscription-Based Routing: In subscription-based routing every subscription from every subscriber is flooded and the publishers follow the matching subscription path to reach the subscribers. This routing path is used to disseminate all the publication events.

These two routing mechanisms have a trade-off between flooding the advertisements and flooding the subscriptions. If the publishers are known to be static or not changing often, it is preferable to use advertisement-based routing. On the other hand if the publishers are changing their publication ranges, flooding the subscriptions is preferable. There are several examples of pub/sub

systems in the literature [Banavar et al., 1999; Carzaniga et al., 2001; Jacobsen et al., 2010] that support both advertisement-based and subscription-based routing. Regardless of these routing techniques, flooding of millions of subscriptions and topics is very expensive in a pub/sub system for large-scale social interaction. Hence, these techniques are not desirable for the scenarios considered in this thesis.

Siena [Carzaniga et al., 2001] was one of the first pub/sub systems to propose a broker-based pub/sub system architecture with a tree or graph topology. In Siena, several optimizations are introduced to minimize the traffic overhead for forwarding the subscriptions and advertisements. One widely implemented optimization in CBPS systems is subscription coverage: when a subscription or advertisement is received at a broker, it stops forwarding it to its neighbors if it finds that there is already a subscription or advertisement covering the incoming subscription.

JEDI [Cugola et al., 2001] uses a tree topology similar to the hierarchical topology proposed in Siena. The subscribers are connected to the leaf brokers and the incoming subscriptions are forwarded from leaf to the root broker. Publication events are also forwarded from the leaf to the root broker. However, if there is a matching subscription at intermediate brokers the event is forwarded down that sub-tree to the leaf brokers. Unlike Siena, JEDI does not support optimizations such as subscription covering. JEDI also has the problem of load imbalance in the brokers closer to the root and the root broker itself.

PADRES [Jacobsen et al., 2010] is yet another broker-based CBPS system, which borrows several ideas and optimizations such as advertisements, subscription covering etc. from Siena and JEDI. PADRES extends the CBPS paradigm in many directions. One of the extensions proposed is *composite subscriptions*. They introduce a way to express the subscriptions as event patterns and hence, are far more expressive than the simple attribute-value predicates. In addition to subscription coverage, PADRES also introduces the concept of subscription merging. Two or more subscriptions at a broker are merged before forwarding them if they have a significant overlap. More extensions in PADRES include support for subscribing to past events; multi-path

routing exploiting acyclic overlays to improve fault-tolerance; dynamic routing algorithms to adapt the subscription and event forwarding under failures and load imbalances.

The tree and graph topologies are generally not self-organizing and adaptable to the variations in the workload. In addition, the position of the subscribers and publishers in the topology impacts the load distribution among brokers. Hence, they need special subscriber and publisher placement and migration techniques such as in [Cheung and Jacobsen, 2010]. There are also pub/sub systems inspired by the self-organization techniques used in P2P systems. In [Baldoni et al., 2007b], the authors propose a technique to build a self-organizing overlay of brokers for Siena. The self-organization is inspired by the P2P clustering techniques to group brokers matching similar events together. However, they are not designed to autonomously balance load among brokers.

2.2.2 P2P Pub/Sub Systems

P2P systems are generally built on top of overlay networks. P2P networks are known for their self-organizing and self-healing properties. In P2P systems no single node needs to be aware of the global knowledge of the system. P2P systems provide a scalable, efficient and fault-tolerant implementation of pub/sub because of self-organization techniques used by individual nodes. The self-organization and maintenance of nodes is achieved by autonomous decentralized algorithms run at each participating node. Typically, P2P overlays are classified as structured and unstructured. The same classification can be seen in pub/sub implementations as well. In this section, we discuss various features and limitations of both structured and unstructured overlays for pub/sub systems proposed in the literature.

Structured Overlays for P2P Pub/Sub P2P pub/sub systems implemented on structured overlay networks follow the concept of assigning a specific position in the network to each joining node. They usually build structures such as a ring, a tree, a torus or a multidimensional hypercube. For this purpose they leverage the structured overlay implementations such as DHT. For example, Scribe and Bayeux are built on top of a Pastry DHT [Rowstron and Druschel, 2001a] and Rebecca is

built on top of a Chord DHT [Stoica et al., 2001]. Typically these systems build a multicast tree per topic in the pub/sub system. The Hermes CBPS system is built on top of a DHT as well. Pub/sub systems built on top of DHTs generally cannot avoid having special nodes known as *rendezvous nodes* which act as a contact point for newly joining subscribers and publishers generating events. There is generally a cost of $O(log(N))$ to reach the rendezvous node each time a publication or a subscription is generated where N is the number of nodes in the system. There is an additional cost of disseminating the events to all the relevant subscribers of the publication. The overlay maintenance to handle churn depends on the underlying DHT implementation. For example, Scribe relies on churn handling provided by Pastry. Waiting for the underlying DHT to recover in order to rebuild the pub/sub overlay can cause loss of messages. In Chapter 3 we show that tree-based pub/sub overlays built using DHTs have a number of limitations in practical settings with respect to scalability and latency in delivery of publications events. In addition, we also show that a pub/sub system PolderCast built using unstructured overlays can outperform such systems.

Overlays built using multidimensional hypercube structures are yet another flavor of structured overlays. Such structures can be implemented over DHTs to facilitate CBPS in a P2P fashion. One such system is Meghdoot [Gupta et al., 2004], it builds a multidimensional hypercube structure to arrange the subscriber nodes. To achieve this Meghdoot relies on a CAN P2P network [Ratnasamy et al., 2001]. Using CAN, Meghdoot builds a d-dimensional hypercube, where d is the number of attributes. The id of a node is determined by the d-dimensional point in the hypercube. The subscriptions are defined using d attributes. Hence, Meghdoot can support CBPS. Whenever a publication event is generated, it is routed to the node responsible for the group of subscribers interested in this event. This is achieved using the proven the CAN routing technique. Meghdoot heavily relies on CAN structure to provide pub/sub functionality and hence the cost of routing messages in CAN is incurred when routing subscriptions and publications. Other similar pub/sub systems building multidimensional DHT structures include Mercury [Bharambe et al., 2004].

Unstructured Overlays for P2P Pub/Sub Unstructured P2P overlays do not have a specific structure like DHTs. Unstructured overlays are typically random graphs. However, the graph structures of unstructured overlays designed for pub/sub systems generally reflect the distribution of topic popularities and subscription sizes in the pub/sub workload. For example, the number of topics subscribed by a node is reflected by its outdegree (number of neighbors). Many unstructured overlays usually rely on gossiping protocols which typically use a random peer sampling service such as Cyclon [Voulgaris et al., 2005] to constantly discover new nodes and to keep the overlay connected and preserve random graph properties. However, in order to build pub/sub overlays without any predefined structures there is a need for a technique to connect the peers sharing the similar interests together. A widely used technique is to rely on clustering of nodes with similar interests. While this technique aids in choosing the neighbors to build a pub/sub overlay, it does not ensure the connectivity of nodes subscribing to the same topic.

SpiderCast [Chockler et al., 2007b] is an example of a pub/sub system which builds an overlay so as to connect the nodes subscribing to the same topic in a sub-overlay using node clustering techniques. SpiderCast tries to minimize the degree of each node by sharing connections with the nodes subscribing to the same topics. This is achieved by running a heuristic at each node called *Greedy Coverage* which greedily selects neighbors to cover each topic K_g times. However, such a technique may not ensure a connected graph for a per topic induced suboverlay. SpiderCast addresses this problem by introducing another heuristic called *Random Coverage* which is to select K_r random peers subscribing to the same topic. The gossiping in SpiderCast protocol is designed to apply both heuristics to build an overlay with a balance between the number of links between the nodes and connectivity. It has been shown that setting K_r to 3 is practically sufficient to achieve a connected suboverlay per topic. While SpiderCast ensures connectivity of the overlay, it does not provide any protocol to disseminate publication events in the overlay. Furthermore, it is not clear how to repair the overlay in the presence of failures and churn.

Vitis [Rahimian et al., 2011] is yet another pub/sub system which employs

gossip-based clustering of nodes with similar interests to build a pub/sub overlay. Vitis tries to maintain a constant node degree for each node. As a result, the problem of disconnected components becomes unavoidable. To address this problem, Vitis introduces special nodes called *gateway* nodes for each disconnected component. Each topic in the system has an associated *rendezvous node* and acts as the contact point for routing subscriptions and publications of that topic. Vitis connects all the disconnected gateway nodes by building and maintaining a tree of gateway nodes with the rendezvous node as the root.

StAN [Matos et al., 2010] builds the unstructured overlay using a combination of gossiping and random walk techniques. StAN avoids the problem of disconnected clusters by assuming that the overlay construction starts with a connected overlay for each topic. This approach limits the ability of the overlay to adapt to churn.

Data-Aware Multicast (daMulticast) [Baehni et al., 2004] is yet another example of an unstructured overlay for pub/sub. A unique contribution of daMulticast is that it builds a hierarchical overlay for hierarchical topics[1]. One of the limitations of daMulticast is that it relies on a separate gossiping process for each topic. While this strategy facilitates building hierarchical overlays, it becomes difficult to scale with millions of topics and hundreds and thousands of topics per subscriber as seen in social interaction systems.

Sub-2-Sub [Voulgaris et al., 2006] builds an unstructured overlay for providing a CBPS service. Unlike other P2P pub/sub systems built on unstructured overlays Sub-2-Sub allows the queries to be specified using range predicates in the subscriptions. Sub-2-Sub relies on gossiping to build sub-overlays among subscriber peers having overlapping range of values of the attributes in the subscriptions. The overlay adapts itself to the changing subscriptions, peers joining and leaving (churn and failures). One of the disadvantages of the Sub-2-Sub system is that the subscription attribute values can potentially overlap in an exponential combination of the attributes and values, resulting in an exponential number of suboverlays. This limits the

[1]For example, alternative rock music is a type of rock music, hence, the topic "music rock" becomes parent of "alternative rock music" in the hierarchy

scalability especially when the subscriptions have high number of attributes.

To summarize, unstructured overlays could eliminate the need for special rendezvous nodes and gateways. However, the existing pub/sub systems built using unstructured overlays are either incomplete by omitting publication dissemination techniques and churn handling [Chockler et al., 2007b; Matos et al., 2010] or they are not scalable because they build overlays per topic [Baehni et al., 2004]. In Section 3.2 we further study the limitations of these approaches in more detail.

2.2.3 Peer-Assisted Architectures for Pub/Sub

P2P pub/sub systems are known to be self-organizing, self-healing and also scale as the number of users grows. If the P2P networks can utilize the resources from user devices then they can scale with the number of users without any additional cost of deploying new hardware. On the other hand dedicated pub/sub infrastructures hosted in datacenters or on cloud infrastructures are known to provide a reliable pub/sub service. Peer-assisted architectures borrow the best of both P2P and dedicated infrastructures. Offloading a subset of the tasks done by the dedicated infrastructure to P2P network can save infrastructure costs such as bandwidth. In this regard, we explore the peer-assisted architectures proposed in the literature for pub/sub.

As learned in Section 1.2, given the limited resources of pub/sub infrastructure, utilizing it to maximize the subscriber satisfaction is critical. Diverting part of the traffic that cannot be handled by the dedicated pub/sub infrastructure can potentially avoid service outages. There is also evidence from peer-assisted content distribution techniques that significant bandwidth costs can be saved if resources from user devices are also used [Kreitz and Niemela, 2010]. Since social interaction systems often involve content distribution as well, using peer-assisted techniques for pub/sub to save costs is in line with the peer-assisted solutions already used by the popular companies such as Spotify.

Unlike broker-based and P2P pub/sub systems, the number of reported pub/sub systems with peer-assisted architectures is limited. [Xu et al., 2011]

propose a system coined *Cuckoo* which is designed to share the pub/sub traffic and workload between a dedicated cloud infrastructure and a P2P network. It is designed to take advantage of the reliability provided by the cloud and cost-effective and yet scalable notification delivery provided by the P2P. Cuckoo is designed towards microblogging social interaction systems such as Twitter. While this is proven to reduce the load on the cloud, it is not clear what to offload to peers while maximizing the utilization of the available dedicated cloud resources to meet given quality of service metrics. We believe more can be achieved with the same cloud resources by using a more sophisticated strategy to select what to offload. In Chapter 6, we formalize this problem and provide various algorithms that could be applied in Cuckoo to gain more benefits.

[Kazemzadeh and Jacobsen, 2012] propose a system coined Publiy+ which is designed to provide peer-assisted content delivery using the pub/sub communication model. In this work a pool of pub/sub brokers are used for coordinating and guiding the peers to the source of required content via pub/sub notifications. While pub/sub is used for improving content distribution, the pub/sub traffic itself is not offloaded to the peers. PAPaS [Ahmed et al., 2012] is another system that is focused on improving content distribution using pub/sub. However, the problem of efficient sharing of pub/sub workload between brokers and peers has not been explored.

2.3 Pub/Sub Offered as Part of Public Cloud Services

There are several cloud providers offering pub/sub as part of their services such as Amazon SNS (Simple Notification Service) [Amazon-SNS], Microsoft Azure Service Bus [MSAzureSB] and PubNub [PubNub]. Enterprises may choose to deploy pub/sub on these services as an alternative to deploying pub/sub in their own datacenters. Amazon SNS for example provides APIs to define topics and subscribers and distribute notifications to the subscribers of the topics. The subscribers can also be mobile devices receiving the notifications as push notifications. Microsoft Azure Service Bus provides a messaging service to deliver messages between cloud applications or in-house datacenter and cloud

applications. However, there is no concept of topics and subscribers in this case. PubNub provides an SDK for developing pub/sub services which can be deployed on the cloud. These systems generally offer the services for a given periodic slab-based pricing in which the customer is charged according to the usage.

The ready to deploy pub/sub services mentioned above are useful yet deficient. Resource allocation in clouds and cost estimations are critical for an enterprise wanting to move the deployment of its proprietary pub/sub engine to the cloud. However, to the best of our knowledge there are no such tools available for the designers of TBPS systems. Moreover, the problem of cost-effective resource allocation for offering pub/sub services on cloud is critical for the service providers as well and yet it has never been addressed in the literature. In Chapter 7, we propose techniques for allocating resources and estimating costs for pub/sub deployment on cloud.

2.4 Pub/Sub Workload Analysis and Characterization

Even though pub/sub is widely researched in the academia, there is a serious lack of real-world workloads for evaluating pub/sub systems. Workload generators for pub/sub are also rare [Yu et al., 2009]. Researchers in academia rely on synthetic workload generation techniques [Baldoni et al., 2007a; Castro et al., 2002; Chockler et al., 2007a,b]. A pub/sub workload generator needs to consider the distributions of topic popularity, subscription size, publication event rate of topics, normalized notification rate per subscriber etc. To the best of our knowledge, no existing work considers all these metrics together for generating a workload.

As mentioned in Section 1.2, social relations in social networks can be converted to subscriptions in pub/sub. Hence, the social relations from the social graphs characterized in [Kwak et al., 2010; Mislove et al., 2007] can serve as subscription workload for evaluating the pub/sub designed for social interaction. A number of academic works use the Twitter social graph as subscription workload for pub/sub [Chen et al., 2013; Rahimian et al., 2011; Xu et al., 2011; Zhang et al., 2013b]. However, these works still rely on synthetic workload generation for publication events. In Chapter 5, we characterize the publication event rate and notification

event rate distributions, in addition to characterizing the social graphs of Spotify and Twitter users.

Even though there have been some characterizations of pub/sub workloads from real systems in the past [Liu et al., 2005; Tock et al., 2005; Yu et al., 2009], they all mainly focus on characterizing the distribution of topic popularity in the workload.

2.5 Content Ranking Techniques

As mentioned in Section 1.3.3, the social interaction notifications generally have human recipients. Delivering every single event to the users may be overwhelming for them. Hence, selecting and delivering only a subset of all the matching events to the subscribers can be useful. One way to select a subset of events is to define metrics to express the subscriber satisfaction. While subscriber satisfaction is subjective, one way to quantify the subscriber satisfaction is to simply guarantee a fixed event delivery rate to the users (explained in detail in Chapter 6). Another possible way is to rank the events and deliver only top-ranked events. There are several techniques popular in the literature to filter and rank events. In this section, we explore and analyze the relevance of those techniques for the scenarios we consider in this thesis.

2.5.1 Ranking and Top-k in Information Retrieval (IR) Systems

The ranking functions in IR systems are typically designed for searching textual documents using keyword queries. The relevance scores of documents containing the given keywords are computed by counting the number of occurrences of a given keyword in a document (i.e. Term Frequency–TF) normalized by its occurrence in the entire document corpus (i.e. Inverse Document Frequency–IDF). Many scoring functions have been proposed in the literature based on the TF-IDF model, the most widely used one being Okapi BM25 [Manning et al., 2008]. Using such scoring models the documents can be ranked to select the top-k most relevant documents for a given keyword query. Top-k queries based on ranking elements of multidimensional datasets are a fundamental building block for IR systems. In

IR systems, given a string keyword query, the goal is to retrieve the most relevant text documents for the given query. The best known general-purpose algorithm for evaluating top-k queries is Fagin's threshold algorithm [Fagin, 1999]. There are also distributed versions of this algorithm proposed [Fagin et al., 2001]. A detailed study of different top-k algorithms used in relational databases is given in [Ilyas et al., 2008]. One of the main disadvantages of these approaches is that they are optimized for static documents and are not suitable for publication events generated in real-time.

2.5.2 Ranking Events and Top-k in Pub/Sub

There are several ranking and filtering schemes proposed for pub/sub. [Drosou et al., 2009; Pripužić et al., 2008] provide a way for the subscribers to order their subscriptions, which is then used to filter and order the matching publication events. The subscriptions can be specified with numerically quantified priorities. Further [Drosou et al., 2009] also introduce the concept of diversification if the top-k events are all similar. The concept of ordering the subscriptions by specifying priorities can be borrowed and applied to the problems addressed in this thesis to extend the satisfaction metrics defined in Section 6.2. However, we omit it from this thesis since it is orthogonal to the problems we consider.

[Machanavajjhala et al., 2008] introduce a multidimensional interval indexing structure by extending R-Tree structures [Guttman, 1984] to include the score. The subscriptions are represented as multidimensional range queries, and the publications are represented as multidimensional points. They also express the ranking from the publisher point of view. For example, given a publication, what are the top-k relevant subscriptions? Such a model is useful in targeted advertising, where advertisements can be expressed in terms of range of values and the users can be represented as multidimensional points. It is known that R-Tree structures are limited in scalability with respect to the number of dimensions.

In [Shraer et al., 2013] news articles are treated as subscriptions and tweets generated in real-time are treated as publications. Then the TF-IDF ranking

function Okapi-BM25 is used to annotate each news article with the top-k relevant tweets. Special indexing techniques are proposed to achieve this efficiently in real-time. While this approach ranks the incoming social events such as tweets against subscriptions, the effectiveness of the ranking depends on the content of the tweet (for example, if a tweet is simply a URL or some metadata such ranking technique may not work). In the application scenarios such as Spotify social interaction considered in this thesis, the content of events cannot be treated as a textual document. Hence, this approach is not suitable for the scenarios considered in this thesis.

In [Rao and Chen, 2011] the authors propose an RSS aggregation service that provides subscribers with the personalized textual content dissemination service over multiple content providers. The service allows the user to subscribe to live web content using keywords. The ranking techniques from information retrieval are used to rank the content. Using the ranking function, the content is ordered and pruned to select the top-k relevant content items and deliver those to the subscribers via a DHT of brokers. The ranking technique used in this work is suitable for subscriptions in the form of textual keyword queries. For ordering the events based on the topics such techniques are expensive and may not scale to millions of subscriptions as required by the social interaction systems.

[Zhang et al., 2013b] propose a CBPS system with a goal to deliver the top-k publication events given any ranking function. The subscriptions are expressed in terms of predicates of attributes and values. While this technique makes the subscriptions expressive, they are also expensive to evaluate.

2.6 Resource Provisioning in the Cloud and Datacenters

In this section, we present an overview of related work specific to resource provisioning techniques used in the cloud and datacenters. We are interested in works that estimate the cost of deploying distributed services such as pub/sub. Our goal in this section is to explore the literature to find the state-of-the-art

techniques for estimating and minimizing the cost of deploying pub/sub.

2.6.1 Resource Provisioning for Publish/Subscribe Systems

To the best of our knowledge there exist no works addressing the problem of cost-effective resource provisioning tailored for TBPS systems. TBPS engines widely used in industry such as Apache Kafka [Kreps et al., 2011] support deployment in cloud and datacenters. However, these systems rely on manual allocation of resources, and do not provide a way to estimate or minimize the deployment costs.

In [Tran et al., 2011] the authors propose a system coined EQS which follows a message-queue architecture similar to a TBPS system. EQS monitors the workload of individual topics and migrates them to keep the load balance among the servers in datacenters. In [Hoffert et al., 2010] the authors predict the resource requirements to adapt to the change in TBPS workload using machine learning techniques. [Barazzutti et al., 2014] propose an elastic architecture for pub/sub to scale up and down the number of servers to dynamically adapt to the incoming pub/sub workload. However, they do not consider minimizing the interserver bandwidth costs. In addition, [Barazzutti et al., 2014] also use the first-fit bin packing technique to allocate resources which is considered as a baseline against more efficient techniques proposed in this thesis.

2.6.2 Resource Provisioning for Stream Processing Systems

One relevant area of research is stream processing in the cloud [Barazzutti et al., 2013; Gulisano et al., 2012]. Stream processing and complex-event processing engines are similar to pub/sub systems. However, the operators and semantics are different. There are a number of resource provisioning techniques for stream processing system. In [Cerviño et al., 2012], the authors propose adaptive resource provisioning for processing stream queries with the goal of optimizing query latency. On the other hand, this work does not aim at minimizing monetary costs. The number of servers is adapted in the proposed scheme to accommodate the incoming event rate of streams. At the same time, the solution

does not focus on minimizing bandwidth consumption or exploring the trade-off between the number of servers and bandwidth consumption[2]. In [Castro Fernandez et al., 2013] the authors propose scaling at operator level for stream processing. While in [Heinze et al., 2013], the authors propose a demonstration of cost estimation for streaming queries, they do not aim at minimizing this cost. This idea is specific to the domain of streaming queries.

In [Ishii and Suzumura, 2011] the authors do consider the problem of minimizing cost of stream processing on cloud. However, the cost is considered only for the part of stream processing which is offloaded to the public cloud.

Resource provisioning techniques developed for a stream processing system differ from the resource provisioning considered in this thesis in many ways. Firstly, the trade-off between number of servers and bandwidth consumed cannot be exploited under stream processing semantics. In addition, there is no concept of subscriber satisfaction metrics, which is essential in our problems. Finally, to the best of our knowledge these works neither formalize nor do any theoretical analysis of resource provisioning problems.

2.6.3 Resource Provisioning for Other Applications

There are several papers addressing resource provisioning in the cloud and datacenters to minimize monetary and other costs in general [Genaud and Gossa, 2011; Vasić et al., 2012; Villegas et al., 2012]. The provisioning techniques used in these works are generic and oblivious to internal semantics of the applications they consider, which limits the optimality of allocation and its cost-effectiveness. For example, this renders exploiting the pub/sub workload characteristics to do cost-effective resource allocation infeasible.

2.7 Relevant Theoretical Problems and Concepts

In this thesis, we formalize a number of problems arising from challenges of pub/sub for large-scale social interaction. We also analyze their hardness

[2]Introduced in Section 1.3.4 and elaborated with examples later in Section 7.1.1.

theoretically and propose efficient algorithmic solutions. Hence, this section is dedicated to exploring the relevant theoretical concepts and problems in the literature.

The problems of maximizing the number of satisfied subscribers we formulate in Section 6.2.1 and Section 6.2.2 bears a strong resemblance to (set) coverage problems; the problem of Budgeted Maximum Coverage (BMC) [Khuller et al., 1999] being the closest match. A significant difference is that in our setting a subscriber may need to be "covered" more than once. This special requirement is a result of satisfaction metric constraint which is to be met for the subscribers.

We also formulate the problem of cost-effective resource provisioning coined Minimum Cost Subscriber Satisfaction (*MCSS*) problem in Section 6.2. *MCSS* resembles a combination of the knapsack [Martello and Toth, 1990] and bin packing problem [Eilon and Christofides, 1971]. There exist works that provide a formalization for the general problem of resource provisioning in the cloud, with emphasis on theoretical analysis. In [Sindelar et al., 2011] a variation of bin packing with various collocation constraints is considered for the problem of VM allocation and proved NP-Hard. However, these works do not take into account the specifics of resource provisioning for pub/sub. For example, the problem of *MCSS* has a unique set of constraints stemming from the satisfaction requirement and from the fact that topics are shared across the subscribers, resulting in the need for cost-effective selection of topic-subscriber pairs. Furthermore, the fact that the incoming bandwidth depends on the distribution of topic-subscriber pairs poses additional challenges and calls for customized allocation algorithms, which we address by introducing a customized version of bin packing with a number of optimizing heuristics.

2.7.1 Relevant NP-Hardness Reductions

In computational complexity theory there are several polynomial-time reductions and they are used for proving that one problem (with known difficulty) is no more difficult than the other. The three most common types of polynomial-time reductions are explained below:

- A **polynomial-time many-one reduction** from a problem A to a problem B is a polynomial-time algorithm for transforming inputs to problem A into inputs to problem B (for which hardness is to be established), such that the transformed problem has the same output as the original problem. An instance of problem A can be solved by applying this transformation to produce an instance of problem B, giving input to an algorithm for problem B, and returning its output.
- A **polynomial-time truth table reduction** from a problem A to a problem B is a polynomial-time algorithm for transforming inputs to problem A into a fixed number of inputs to problem B, such that the output for the original problem can be expressed as a function of the outputs for B. The function that maps outputs for B into the output for A must be the same for all inputs, so that it can be expressed by a truth table.
- A **polynomial-time Turing reduction** from a problem A to a problem B is an algorithm that solves problem A using a polynomial number of calls to a subroutine for problem B, and polynomial-time outside of those subroutine calls.

In this thesis our goal is to establish the hardness of the formulated problems by measuring the relative computational difficulty of the existing problems. Hence, we use "polynomial-time many-one reduction" techniques which is sufficient to prove the hardness.

The family of coverage problems are generally proven NP-Hard using reductions from the Max Cover problem [Hochbaum, 1997]. We instead reduce Densest-k-Subgraph (DkS) problem to our *B3M* problem, which allows us to rule out the existence of a PTAS. The technique to reduce from DkS to a flavor of the multi-coverage problem (*B3M*) is new. Seminal work on analysis of the maximization of submodular set functions was originally done in [Fisher et al., 1978]. In this thesis, we exploit the submodularity property to derive a constant approximation ratio for its greedy heuristic and to speed up the corresponding algorithm.

Typically, researchers construct a reduction to problems which involve packing items under constraints [Hochbaum and Maass, 1985] from the bin packing problem or the knapsack problem. In this thesis, we prove the NP-Hardness of the *MCSS* problem, by first formulating a decision version of the *MCSS* problem, then by reducing the well-known Partitioning Problem [Garey and Johnson, 1979] to it.

Chapter 3

PolderCast: P2P Overlay for Fast, Robust and Scalable Dissemination in Topic-Based Pub/Sub

Designing a P2P pub/sub system that is scalable, robust to failures and provides faster dissemination mechanisms all at once is a challenge. This is attributed to the high number of desirable characteristics that a large-scale P2P pub/sub system has to possess all at once in order to be a viable practical solution, as mentioned in Section 1.3.1. In particular, the list includes: (1) Correct delivery of all publications, i.e., absence of false negatives or deterministic 100% hit-ratio guarantee in a failure-free run, (2) *High hit-ratio* under realistic node churn, (3) Fast recovery at the end of a churn period and mending of the overlay so as to achieve 100% hit-ratio, (4) *Low degree* of overlay nodes, (5) *Relay-free routing* (also called topic-connectivity), which means that only subscribers interested in a topic are involved in routing events for that topic, (6) *Scalability* with the number of nodes, topics, number of nodes interested in a topic, and number of topics a node is interested in, (7) Effective dissemination: *fast*, with as *little duplicate delivery* as possible, and *fair distribution of load* due to routing and processing, and (8)

Low overhead of overlay maintenance.

The design challenge is amplified due to a number of trade-offs: low node degree and relay-free routing, robustness under churn and lack of duplicate delivery, scalability and precise delivery with few false negatives and false positives are fundamentally at odds with each other. Furthermore, each of the principal solution approaches provides a bundle of desirable and undesirable properties at the same time: dissemination over multicast trees is fast and without duplication but it is fragile, whereas gossiping is robust but lacking deterministic delivery guarantees.

In this chapter, we present PolderCast[1], a P2P architecture for topic-based pub/sub. To the best of our knowledge, PolderCast is the first solution that takes all of the above factors into account and harmonizes them. In order to substantiate this claim, we present a survey of existing approaches and analyze their performance with respect to most of the above characteristics.

This combination of desirable properties is provided by an implementation that blends deterministic propagation over maintained rings with probabilistic dissemination following a limited number of carefully selected random shortcuts. per topic rings allow for relay-free routing and 100% hit-ratio in absence of node churn, yet they are constructed in such a fashion so as to reuse the same links for multiple rings thereby minimizing the average node degree. Although at a conceptual level this overlay structure encompasses a separate Hybrid Dissemination [Voulgaris and Van Steen, 2007] overlay *per topic*, our design leverages interest locality to produce a single composite overlay with substantially fewer links and hence, lower node degrees. Our implementation is based on a new efficient epidemic-based algorithm for creating and maintaining the proposed overlay in a self-organizing way.

We evaluate and validate the properties of our system using extensive simulations in large-scale settings of up to 10K nodes, 10K topics, and 5K topics per node. We use real-world traces from Twitter and Facebook social networks to model subscriptions. Robustness with respect to node churn is evaluated

[1]The term is inspired by the Dutch *polder model*, in which diverse societal groups collaboratively negotiate to obtain broadly supported solutions.

through traces from the Skype super-peer network. We empirically show that our system (1) converges fast, (2) provides 100% hit-ratio in the absence of node churn and reasonably good hit-ratio in the presence of node churn, (3) has logarithmic dissemination speed in terms of number of hops and (4) has constant factor traffic overhead. We use widely renowned Scribe [Castro et al., 2002] as a baseline in a number of our experiments.

3.1 Preliminaries

The system consists of a set V of nodes. Each node in the system has a *unique* identifier (e.g., a hash of its IP address), assigned to it when joining the system. Node identifiers are assumed to be sortable and to occupy a circular value space. We assume that the underlying communication network is fully connected, in the sense that *any* node can send a message to *any* other node, provided it knows its IP address.

The topic-based publish/subscribe communication system is organized around a set T of topics. Each node can play the role of a subscriber or publisher or both. A subscriber v expresses its interest in a set of topics $T_v \subseteq T$. We call $|T_v|$ the *subscription size* of node v. A publisher posts an event on exactly one topic t. The published event should be delivered to *all* $|V_t|$ ($V_t \subseteq V$) subscribers interested in t (no false negatives) and *only* to them (no false positives).

Both publishers and subscribers are allowed to join and leave at any moment, without any prior notice. Node crashes are, therefore, inherently dealt with as ungraceful leaves. In fact, there is no way to distinguish between the two. We assume that a node that leaves and rejoins after a while can remember its prior state.

3.2 Survey of Related Approaches

In practice, a pub/sub system should satisfy a wide spectrum of desirable properties in the context of high robustness, low dissemination latency, low communication overhead, and high scalability. Many of those properties exhibit

an inherent trade-off with each other so that striking the right balance is a central challenge in a pub/sub system design and a guiding objective for our approach.

Table 3.1 compares the characteristics of PolderCast with principally different approaches for P2P topic-based pub/sub systems.

With respect to robustness, a pub/sub system should ideally guarantee both 100% hit-ratio without node churn and high hit-ratio in presence of node churn. Consider that existing approaches to P2P pub/sub either utilize epidemic dissemination (daMulticast [Baehni et al., 2004]), or build specialized dissemination overlays. It is well-known that while robust under churn, epidemic dissemination does not provide full reliability, even in a completely static system. On the other hand, most existing dissemination overlays for topic-based pub/sub are fragile (such as dissemination trees in Scribe [Castro et al., 2002], Magnet [Girdzijauskas et al., 2010], or Bayeux [Zhuang et al., 2001]) or at least they rely on designated nodes whose existence is critical for correct operation of distributed matching. For example, Scribe and Vitis [Rahimian et al., 2011] have a dedicated rendezvous node for each topic. Additionally, Vitis builds subclusters for each topic and the communication between subclusters is handled by gateway nodes.

While these systems provide a number of churn handling mechanisms, fragility of dissemination overlays or reliance on central nodes conceptually limit the potential for high hit-ratio under churn, as we further explore in our evaluation in Section 3.5.6. SpiderCast [Chockler et al., 2007b] builds an unstructured overlay that strives to maximize clustering of nodes according to their interest in topics. As observed in [Matos et al., 2010], this approach may yield an overlay in which highly-connected clusters are interconnected by few links, which we call *weak bridges*. Existence of such weak bridges also impacts the robustness of the system under churn.

PolderCast combines deterministic dissemination over a ring with probabilistic dissemination similar to gossiping. The former mechanism guarantees 100% hit-ratio in a static system while the latter provides a high hit-ratio under churn. This is further corroborated by the experimental evaluation in Section 3.5.6.

Table 3.1: Comparison of State-of-the-Art with PolderCast

Property\System	Scribe	Vitis	SpiderCast	StAN	daMulticast	PolderCast
Central nodes*	RV	RV&GW	WB	None	None	None
High hit-ratio under churn?	✗, see Section 3.5.6	✓	N/A	N/A	✓	✓
100% hit-ratio in absence of churn?	✓	✓	N/A	N/A	✗	✓
TCO?	✗	✗	Probabilistic	Probabilistic	Deterministic	Deterministic
Degree of node v	$O(\log \lvert V \rvert)$	$O(1)$	$O(\lvert T_v \rvert)$	$O(\lvert T_v \rvert)$	$\Theta(\lvert T_v \rvert)$	$O(\lvert T_v \rvert)$
Incl. dissemination?	✓	✓	✗	✗	✓	✓
Average Duplication Factor	None	Scoped flooding	N/A	N/A	Gossiping	$\leq$ Fanout(f)
Average Delay	$\mathrm{O}(\log \lvert V \rvert)$	$\mathrm{O}(\log^2 \lvert V \rvert)$	N/A	N/A	$O(\log \lvert V_t \rvert)$	Typically $O(\log \lvert V_t \rvert)^{\#}$

* RV: Rendezvous. GW: Gateway. WB: Weak bridge.

For more details refer to Section 3.5.4 and the discussion below in this section.

Consider the characteristics of the overlay built in various existing approaches: A low number of relay nodes is instrumental in reducing the communication and processing cost of dissemination as well as propagation latency expressed by path lengths. Furthermore, guaranteed absence of relays, i.e. *topic-connectivity* [Chockler et al., 2007a], simplifies message routing mechanisms. On the other hand, fanout is a common minimization parameter in overlay design, which strongly affects system scalability.

Unfortunately, the desirable characteristics of having a low node degree and relay-free routing exhibit a fundamental trade-off [Chockler et al., 2007a]. At one extreme is having a fixed node degree independent of the number of topics a node is interested in.

Such an approach is proposed in Vitis. This results in a relatively high number of subclusters that need to be connected by additional means, such as gateways, rendezvous nodes, and relays. Scribe builds dissemination structures on top of an underlying DHT whose node degree might be either constant or logarithmic with the total number of nodes in the system. In these systems, a pair of nodes interested in the same topic might be connected by a chain of $\Theta(\log |V|)$ relays.

At the other extreme of the trade-off are systems that build and maintain a separate overlay for each topic independently, such as Tera [Baldoni et al., 2007a] and systems that employ gossiping on a per topic basis, such as daMulticast. These approaches guarantee topic-connectivity during stable periods without churn. However, the degree of node v in these systems is in the order of the number of subscriptions: $\Theta(|T_v|)$.

SpiderCast and StAN strive to maintain a topic-connected overlay by building random links between the nodes while exploiting the correlation between node interests in order to minimize the degree. Since correlations are typically present in pub/sub workloads, this results in a lower degree compared to Tera or daMulticast. After the system becomes stable, these systems will eventually produce a topic-connected overlay with high probability. Yet, the guarantee of relay-free routing is only probabilistic, which yields low overhead and latencies, but requires additional mechanisms to route messages across potentially disconnected clusters.

The PolderCast approach we propose in this chapter provides a deterministic

guarantee of relay-free routing similar to Tera or daMulticast. At the same time, the degree is similar to that of SpiderCast or StAN due to exploiting correlations. As shown in Table 3.1, SpiderCast and StAN focus on overlay construction and maintenance and do not propose any specific routing algorithm, thereby rendering the discussion about message dissemination properties as well as hit-ratio nonapplicable to these systems.

For the rest of the approaches, we consider two salient factors that determine the efficiency of message dissemination:

(a) Average message duplication factor per node: the number of times (excluding the first) that the same published message is received by a node on average. When the routing is relay-free, average message duplication factor directly translates into the communication cost of message dissemination.

In Scribe, Magnet, and Bayeux, a routing tree is used to disseminate publications, which eliminates any duplication of messages. In the hybrid overlay approach of Vitis, the node floods a published message to those of its neighbours that are interested in the message topic. Even though Vitis has a fixed total degree per node, this fanout may be high enough so as to lead to a high number of duplicate deliveries for the same published message. In daMulticast, the configurable fanout of the epidemic dissemination used for propagating published messages governs the duplication factor. In PolderCast there is a fixed maximum dissemination fanout f (typically $f = 2$) for each topic. Each node interested in the topic forwards a message only once (the first time the node receives the message) along at most f links, which gives a bound of f on the duplication factor.

(b) Average path length: the average number of hops required for a message to reach a node interested in that message. As shown in Table 3.1, all of the structured and hybrid overlay approaches have an expected path length that is logarithmic or square logarithmic with the total number of nodes $|V|$ in the system. Yet, the inclusion of relays nodes (both at the DHT level and pub/sub implementation level) into the dissemination path causes path lengths for some nodes being significantly longer than $O(\log |V|)$, as we show in Section 3.5.4. DaMulticast performs gossiping on a per topic basis so that the expected path

length is logarithmic with the number of nodes $O(\log |V_t|)$ interested in the topic.

In our approach, we also strive to achieve expected path lengths that are logarithmic with $O(\log |V_t|)$ due to the random shortcuts links used for dissemination. From the results in [Voulgaris and Van Steen, 2007], it can be derived that if there is a sufficient number (f-1) of random shortcut links between the nodes interested in a particular topic, PolderCast guarantees average dissemination path lengths for that topic to be asymptotically logarithmic. However, our dissemination mechanism uses a fixed number of random links independently of the number of topics a node is interested in. This may potentially render the dissemination mechanism ineffective for a node that is interested in many topics, in which case the average path length may become linear with $|V_t|$ due to the use of ring links only. Fortunately, this scenario does not manifest itself for typical pub/sub workloads, as confirmed by the empirical results in Section 3.5. Note that the dissemination fanout f determines the base of the logarithm and as such, governs the trade-off between the dissemination speed and duplication factor.

Based on the analysis in this section, we conclude that the solution for topic-based pub/sub we propose is (a) free from rendezvous and relay nodes (b) robust and resistant to churn, and (c) it facilitates efficient message dissemination.

3.3 PolderCast: Disseminating Events

We present PolderCast in a top-down approach. In this section we describe the structure of the target overlay and we explain how dissemination is performed once this overlay is in place. Then, in Section 3.4, we dive into the mechanisms in charge of building and maintaining such an overlay.

3.3.1 The Dissemination Overlay

At a conceptual level we maintain a separate ring per topic augmented by random links shared across the topics. Each ring connects *all* subscribers of the corresponding topic and *only* them. Individual topic rings altogether form a single, connected, and navigable overlay. Ensuring connectivity among all

subscribers of a topic, a property known as *topic-connectivity*, allows for relay-free routing among them. It is the reason why PolderCast achieves 100% hit-ratio in the absence of node churn: When an event for a certain topic reaches *any* subscriber of that topic, it is guaranteed to reach *all* remaining subscribers by being propagated along that topic's ring. While this distribution mechanism alone might be adequate for topics with a moderate number of subscribers, its linear dissemination speed does not scale with the popularity of topics. This is the reason why we introduce random links serving as dissemination shortcuts. Propagating events across (some of the) random links to arbitrary other subscribers of the same topic, accelerates dissemination to exponential speed. It additionally provides a controlled degree of redundancy that increases robustness and hit-ratio under node churn.

In this work, we request that a publisher on topic t subscribes to t prior to publishing events, thus becoming a part of the dissemination ring. This overhead for publishers is considered acceptable by most applications and in many existing pub/sub systems.

The rings for each topic are *bidirectional* and nodes are placed into rings in the order of their node ids. That is, a node p maintains, with respect to each topic t in its subscription, two links: one to its t-successor and one to its t-predecessor. The t-successor of node p is defined as the node with the *closest higher* than p's id (in modulo arithmetic), among all subscribers of topic t. The t-predecessor is defined likewise for the *closest lower* id. Figure 3.1 gives a sample topology of three topics, and the respective intermingling rings.

It should be observed that while the use of rings in hybrid dissemination structures has appeared in the past [Voulgaris and Van Steen, 2007], their application to topic-based pub/sub is new. The main challenges of using ring in pub/sub lies in combining such structures, one per topic, into a single manageable overlay. In practice, maintaining a separate ring per topic is very expensive, notably for nodes subscribed to many topics. However, it has been observed that subscriptions tend to be strongly correlated [Liu et al., 2005]. Our approach exploits this correlation in order to substantially lower the number of links maintained: A single link can serve as a ring link for multiple topics.

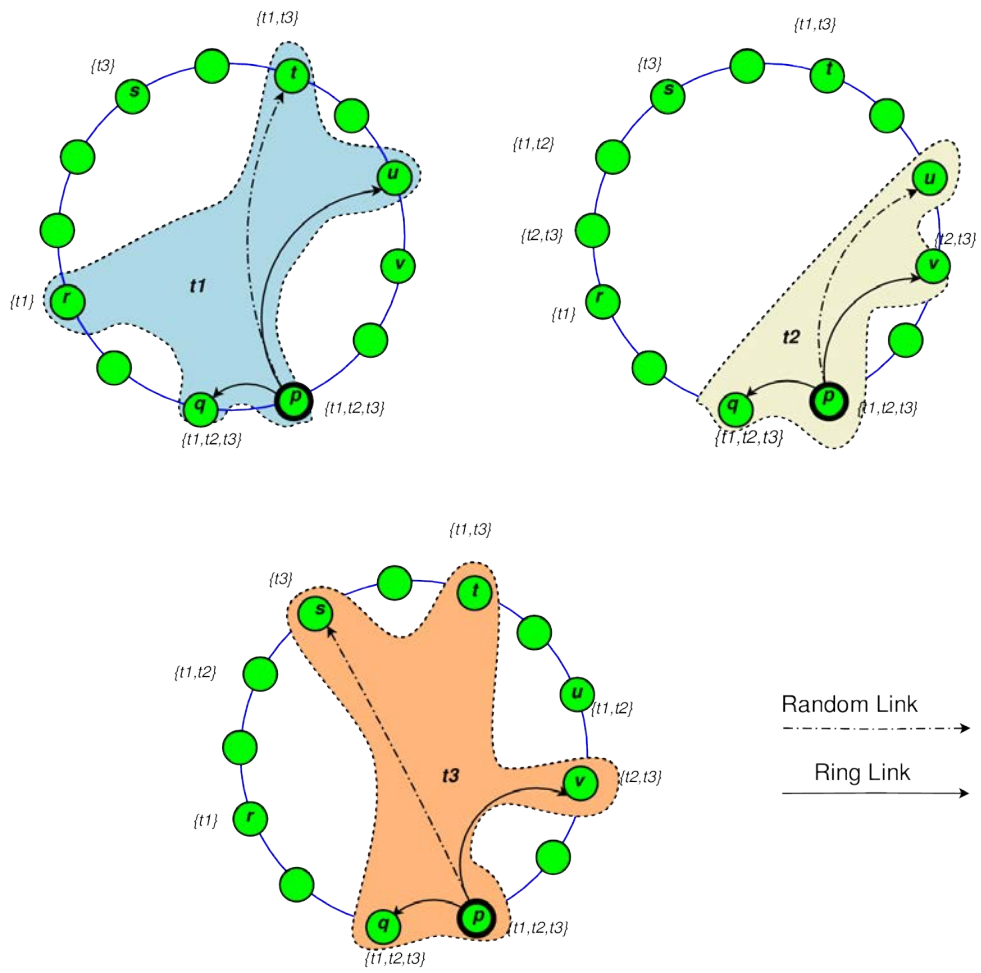

Figure 3.1: Topology for three topics $\{t1, t2, t3\}$, showing the ring neighbor links and random neighbor links originating from the node p. Note that q serves as successor of p for all three topics, and v serves as predecessor of p for topics $t1, t2$ illustrating link sharing.

It is possible to build an overlay with link consolidation across the topics as the central optimization metric in mind. This approach minimizes node degree but may result in a per topic ring being partitioned into multiple sub-rings. In order to avoid this risk, PolderCast takes a more balanced approach and builds a guaranteed ring for each topic separately but in such a way that links have a higher chance of being reused in multiple topics. Specifically, rings are constructed based on node ids instead of their subscriptions. Assume nodes p and q are both subscribed to t_1 and t_2, and they are ring neighbors for t_1. This means that they are both on the ring for t_2 and their ids are numerically close, thereby increasing the chance that they will be ring neighbors for t_2 as well. We further investigate the effect of link consolidation in our experiments in Section 3.5poldercast:sec:experimentalevaluation.

With respect to random links, their choice and quantity may have a profound impact on the performance, as discussed in Section 3.2. PolderCast combines a configurable number of random links of two types: interest-induced links formed between subscribers with similar subscriptions shorten average dissemination path lengths. At the same time, uniform random links help overcome partitions under node churn and improve load balancing by diverting incoming links from nodes that subscribe to many topics, which become a likely target for interest-induced links. We describe the algorithm for random link formation in Section 3.4 and consider the importance of the links of each type in Section 3.5.

3.3.2 Event Dissemination

Our event dissemination protocol is inspired by that of RingCast [Voulgaris and Van Steen, 2007] (the protocol is parameterized by a dissemination fanout, f): A node receiving an event for topic t for the *first time*, propagates it f times. Specifically, if the event has been received through the node's t-successor (or t-predecessor), it is propagated to its t-predecessor (or t-successor) and f-1 arbitrary subscribers of t. If the event was received through some third node, or if it originated at the node in question, it is propagated to both the t-successor and the t-predecessor, as well as to f-2 other subscribers of t. Finally, if a copy

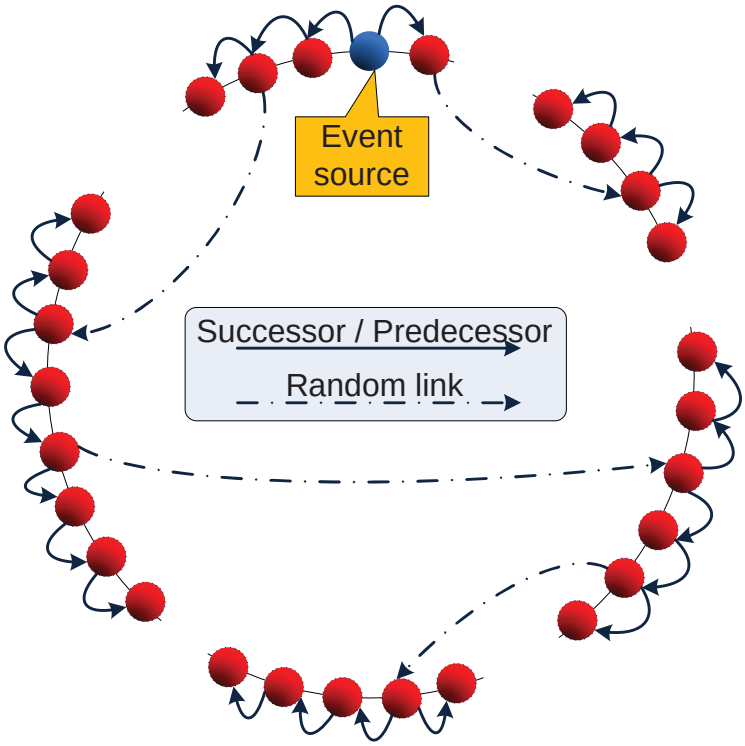

Figure 3.2: Dissemination example for a particular topic, in a partitioned ring.

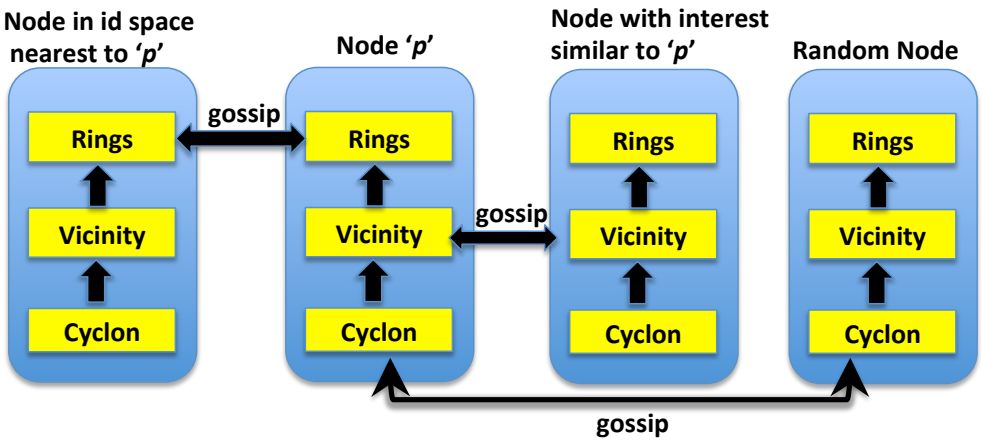

Figure 3.3: Three-layered architecture. Each layer gossips with the respective layer in other nodes.

of this event has already been received in the past, it is simply ignored.

From the results in [Voulgaris and Van Steen, 2007], it can be derived that if there is a sufficient number (f-1) of random shortcut links between the nodes interested in a particular topic, PolderCast guarantees average dissemination path lengths for that topic to be asymptotically logarithmic. Even under node churn PolderCast tries to achieve complete dissemination as shown experimentally in Section 3.5.6. Figure 3.2 gives an intuitive illustration of dissemination in a partitioned ring.

Since we apply this dissemination protocol for multi-topic pub/sub, however, analyzing its performance in PolderCast is significantly more difficult because the random links are shared across multiple topics and the number of utilizable random links varies for each and every node. Furthermore, some of the random links are skewed towards peers with multiple overlapping topics. This may interfere with the nice property of exponential dissemination speed that is inherent to many gossiping protocols. It may also cause a node whose subscription is similar to those of many other peers to become a hotspot due to a high number of incoming random links. We evaluate these aspects experimentally in Section 3.5.

3.4 PolderCast: Building the Overlay

PolderCast's overlay management mechanism is built around three modules: Rings, Vicinity, and Cyclon, as shown in Figure 3.3. Each module maintains its own view, managed by a separate gossiping protocol, which gossips periodically, asynchronously, and independently from the other two modules. In table below we list the parameters controlling the number of neighbors maintained (*view size*), and the maximum number of neighbors included in a gossip message (*gossip size*), per module.

module name	**view size**	**gossip size**
Rings	ℓ_{ring} (per subscribed topic)	g_{ring}
Vicinity	ℓ_{vic} (in total)	g_{vic}
Cyclon	ℓ_{cyc} (in total)	g_{cyc}

Considering a node p with topics T_p, the three modules operate as follows.

With respect to each topic $t \in T_p$, the Rings module on p is responsible for discovering p's t-successor and t-predecessor. It achieves this by considering a few links to arbitrary subscribers of t as a starting point, and periodically gossiping with them to trade them for other subscribers of t of gradually closer ids.

The Vicinity module is responsible for feeding the Rings module with a few neighbors for each topic $t \in T_p$, of arbitrary ids. It is based on Vicinity [Voulgaris, 2006], a topology management protocol that strives at discovering for each node the *closest* other nodes based on some *proximity function*. Per the proximity function introduced in the context of PolderCast, the more topics two nodes share the closer they are ranked. Moreover, as detailed in Section 3.4.2, our proximity function dynamically adapts to favor topics currently under-represented in the Rings module.

Finally, the Cyclon module [Voulgaris et al., 2005], is a lightweight peer sampling service [Jelasity et al., 2009], providing each node with a continuous stream of neighbors chosen uniformly at random from the whole network. As detailed in Section 3.4.3, this is essential for keeping the whole overlay connected, and enabling flexible overlay maintenance in the face of failures and node churn.

For any of the three modules, node q being a *neighbor* of node p means that p has a copy of q's *profile* in the respective module's view. A node's profile contains (i) its IP address and port number, (ii) its (unique) node id, and (iii) the ids of topics the node is subscribed to, each annotated with a *priority* that node assigns to finding neighbors of that topic. The priority of a topic is determined by the number of neighbors it has in the Rings module: topics with fewer Rings neighbors are assigned higher priority. Clearly, two or more copies of a node's profile may be different, notably when the node updates its subscriptions, or reports different priorities for its topics. When gossiping to a neighbor, a node sends a fresh copy of its profile, reflecting its current state.

Note that the three gossiping protocols comprising PolderCast are executed continuously. In a network characterized by dynamicity, due to nodes departing or joining at any time, crashing, or merely changing their subscriptions, there is no notion of *final* convergence. Instead, nodes engage in a constant convergence process.

3.4.1 The Rings Module

The Rings module manages the ring links. That is, it aims at discovering a node's successor and predecessor for each topic in its subscription, and at quickly adapting to new successors/predecessors in dynamic networks.

In that respect, each node maintains ℓ_{ring} neighbors for each topic in its subscription: $\ell_{ring}/2$ with lower and $\ell_{ring}/2$ with higher id. It periodically picks a node from its Rings view, and the two nodes exchange up to g_{ring} neighbors to help each other improve their Rings views.

Assume p selects its neighbor q for gossiping. First, p collects *all* subscribers of topics which p and q have *in common*, considering the union of views of *all three modules*. Second, it sorts them by id, and for each topic in common with q it selects the $\ell_{ring}/2$ ones with just lower and the $\ell_{ring}/2$ ones with just higher id than q's id. If more than g_{ring} nodes have been selected, it randomly picks g_{ring} of them. Finally, it sends the selected nodes (i.e., the respective node profiles) to q. Node q does the same in return.

Although the dissemination protocol requires just two ring links per topic, namely the topic successor and predecessor, Rings maintains up to ℓ_{ring} links per topic. This provides stand-by successors and predecessors to be used in case of failures or node churn. Additionally, it helps nodes navigate to their direct ring neighbors faster, once they have reached the proximity of their ids.

Finally, in order to increase the diversity of neighbors contacted for gossiping, the Rings module employs a Least Recently Used (LRU) selection policy. This prevents contacting the same neighbor twice in a short interval, when it probably has no new useful information, at the expense of not contacting some other neighbor for a much longer duration. The LRU policy also plays an important role in churn handling by PolderCast, thus its implementation details are deferred to Section 3.4.4.

3.4.2 The Vicinity Module

The Vicinity module is responsible for maintaining interest-induced random links, that is, randomly chosen links between nodes that share one or more

topics. Such links serve as input to the Rings module, as detailed in Section 3.4.1. Additionally, they are used by the dissemination protocol to propagate events to arbitrary subscribers of a topic, as explained in Section 3.3.2.

Interest-induced random links are handled by Vicinity [Voulgaris, 2006], a generic protocol for topology construction and management that lets nodes find their *closest* neighbors out of the whole network, based on some *proximity function*. In short, each node maintains a view of ℓ_{vic} neighbors and periodically gossips with them to discover nodes of even closer proximity, in which case it retains them in place of the least proximal neighbors.

Let p choose q for gossiping. Node p merges its views from all three modules. Then, it selects the g_{vic} nodes closest to q by applying the proximity function on its behalf, and ships them over to q. Upon reception, q merges the received neighbors with the union of all its views, and updates its Vicinity view to the ℓ_{vic} closest neighbors. Finally, q responds by selecting and shipping back its g_{vic} closest to p nodes.

Clearly, the proximity function plays a crucial role in Vicinity. In the context of PolderCast, the proximity function is designed to ensure that the Rings module is supplied with (arbitrary) neighbors for *all* its topics. In that respect, candidates subscribed to topics annotated with higher priority by the target node are ranked closer compared to candidates of lower priority topics. Among candidate nodes that rank equally in terms of topic priorities, proximity is determined by the number of topics shared with the target node: the more shared topics, the closer their ranking.

3.4.3 The Cyclon Module

Uniform random links are handled by the Cyclon peer sampling service [Voulgaris et al., 2005]. This module's purpose is twofold. First, it keeps the whole set of subscribers connected in a single partition, even in the presence of churn, large-scale failures, or subscription changes. Connectivity is crucial to let new subscribers find their way to their appropriate neighborhood sets, irrespectively of where they initially joined the network. Second, it constitutes a

source of links selected uniformly at random from the whole network. Such a source of random links is fundamental to the operation of the other two modules. Further details about the Cyclon protocol can be found in [Voulgaris et al., 2005].

3.4.4 Churn Handling

It is a key design goal of PolderCast to provide a high hit-ratio and reasonably low delivery latency under node churn, while keeping the number of duplicate messages controllably small. To that end, PolderCast should adapt promptly to two types of changes. First, information updates, such as newly joining nodes, new subscriptions, etc. should be propagated fast. Second, the system should quickly detect the disconnection (graceful or due to failures) of nodes, and discard related information from the network.

With respect to propagating new information fast, PolderCast relies on its fast convergence properties. When a node joins the network, for example, its Vicinity module will quickly find some neighbors for each topic. Once a neighbor has been found for some topic, the Rings module can quickly locate the appropriate successor and predecessor in an already largely connected topic ring. When a node's subscription changes, Vicinity will adjust its topic priorities to boost under-represented (new) topics. We further explore the convergence speed of PolderCast experimentally in Section 3.5.2.

With respect to ridding the system from outdated links, PolderCast employs a proactive mechanism for removing dead neighbors from node views. Whenever a node p gossips with a neighbor q, it temporarily removes q from the respective module's view, anticipating that q will respond and will be inserted anew in p's view. This way, dead neighbors are silently discarded, while alive ones are refreshed. To prevent dead neighbors from remaining indefinitely in a view, a node always selects to gossip with its least recently refreshed neighbor.

Freshness of a neighbor is approximated by an *age* field, associated with every view entry. Once per cycle, a node increments the ages of all its neighbors by one. A neighbor's age is zeroed when a gossip message (or response) is received from that neighbor. A neighbor's age is retained also when that neighbor is handed

from one node to another. This way, a dead node's links will have increasingly higher chance to be selected for gossiping (and consequently discarded), even if they are copied among third nodes.

Although the age mechanism provides only an approximation of a link's freshness, it turns out to work sufficiently well for fast removal of dead links. We investigate the impact of node churn on the performance of PolderCast in Section 3.5.6.

3.5 Experimental Evaluation

We evaluate PolderCast by simulation based on real-world traces. We focus on the overlay properties (such as the node degree), efficiency of dissemination (delays and duplicate delivery), communication overhead of overlay maintenance, and performance under node churn (hit-ratio for message delivery and speed of convergence for overlay construction). We also compare the performance of PolderCast with Scribe [Castro et al., 2002] as a baseline.

We implement both PolderCast and Scribe using the widely adopted *PeerSim simulator* [Montresor and Jelasity, 2009]. Scribe is implemented as an application atop Pastry DHT [Rowstron and Druschel, 2001a]. We use the implementation of Pastry for *PeerSim*, publicly available at [PeerSim-Pastry]. We evaluate both PolderCast and Scribe at a scale of up to 10K nodes. Experiments of similar scale are common in this area [Patel et al., 2009; Rahimian et al., 2011].

Unless otherwise mentioned, the view sizes of Cyclon and Vicinity (ℓ_{cyc} and ℓ_{vic}, respectively) were set to 20 entries each, and the gossip lengths in all three protocols (g_{cyc}, g_{vic}, and g_{ring}) were set to 10 entries. The configuration parameters for Scribe are $b = 4$ which defines the base $2^b = 16$ for the log structure of Pastry DHT and $l = 32$ for the leaves of the DHT routing table.

3.5.1 Experimental Settings

Subscription Workload: Our subscription workloads come from massively deployed social networks, namely Twitter and Facebook.

(1) Twitter dataset: We used a public Twitter dataset [Kwak et al., 2010], containing 41.7 million distinct user profiles and 1.47 billion social followee/follower relations. In Twitter, when a user posts a message (known as a *tweet*), the tweet is delivered to all followers of that user. As such, each user is modelled as a topic and all its followers are the respective subscribers. Similarly the set of users (followees) a user Alice follows, form Alice's subscription set. Note that in Twitter, relations are unidirectional, i.e., user Alice following user Bob does not require also Bob following Alice.
(2) Facebook dataset: We used a public Facebook dataset [Wilson et al., 2009], with over 3 million distinct user profiles and 28.3 million social relations as a second workload for our evaluations. Similarly to Twitter, users are modelled as topics as well as subscribers. However, in Facebook relations are bidirectional, therefore two friends in the Facebook social graph subscribe to each other in our model.

Our simulations were performed with workloads of 10K nodes (i.e., up to 10K topics and 10K subscribers), extracted from the original Twitter and Facebook social graphs in a methodology inspired from [Patel et al., 2009; Rahimian et al., 2011]. More specifically, starting with a random set of a few users as seeds, we traversed the social graph using breadth first search, until the target number of nodes was reached, and *all* edges between them were extracted to our sample.

Figure 3.4 shows the complementary cumulative distribution function (CCDF) of follower/followee counts for both the original Twitter(TW) and Facebook(FB) datasets, as well as for our respective extracted datasets in the inner plot. The plots indicate that the original dataset properties were retained in our extracted sample.

Publication Workload: Due to lack of publicly available real-world publication workload we synthetically generate publications. We post one publication event for *each* topic, initiated by a randomly picked subscriber of that topic. Although in practice, event arrival rate may vary across different topics, we use a uniform publication rate since it has no effect on the metrics we consider for evaluating the PolderCast system.

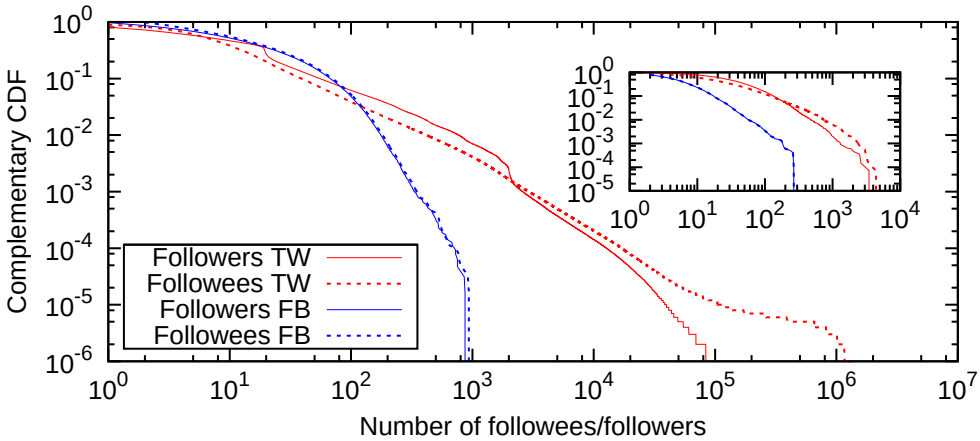

Figure 3.4: Distribution of followers and followees, for the Twitter (41.7M users) and Facebook (3M users) traces. Inner plot: trace samples used (10K users).

Latency and Churn Datasets: We use the *King dataset* [Gummadi et al., 2002] to model communication latency between nodes. Finally, we evaluate our system under node churn, using real-world churn traces: *Skype dataset*. We use Skype super-peer churn traces from [Guha and Daswani, 2005], which tracked joining and leaving timestamps of 4000 nodes for one month, starting on September 12, 2005.

3.5.2 Speed of Convergence

We first evaluate the time it takes to jump-start a PolderCast overlay from scratch. We start by 10,000 nodes that are already running Cyclon (i.e., each node has ℓ_{cyc} links to random other nodes), but whose Vicinity and Rings views are completely empty, and we let them gossip to self-organize in a PolderCast overlay. Observe that fast convergence to an optimal overlay upon the extreme case of simultaneous bootstrapping typically implies fast reconciliation after a period of milder churn.

Given the input, we start by an offline construction of correct target rings to which the systems should converge over time. Then, we deploy PolderCast.

At each cycle, we measure the percentage of target ring links that are *not* yet in place (missing links), as well as the percentage of topics for which the ring

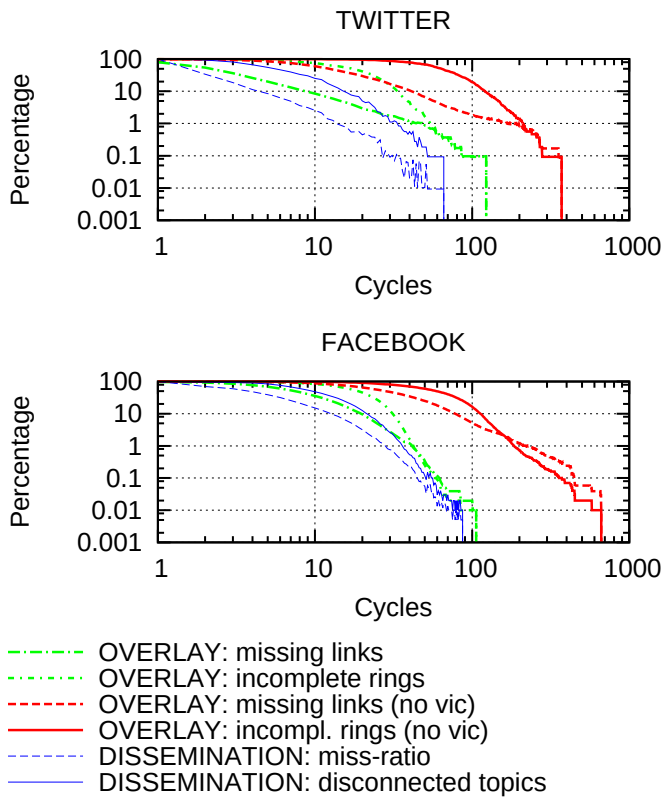

Figure 3.5: Convergence speed

has not converged yet (incomplete rings). Figure 3.5 shows these metrics for the Twitter and Facebook workloads, respectively.

In order to assess the overlay's efficiency in disseminating events, we conduct another experiment by "freezing" the overlay at the end of each cycle, and posting one event for each topic. We record the percentage of nodes that *missed* an event they should have received (miss-ratio), as well as the percentage of events that did *not* make it to all subscribers of their topic (disconnected topics). These measurements are also shown in Figure 3.5.

The results show that the overlay converges quite fast: Within 60 cycles, 99% of topic rings are complete. They also indicate that the PolderCast overlay is highly efficient even with partially complete rings because it takes fewer cycles to

achieve a connected overlay (0% miss-ratio) per topic. This is due to propagating events across random links, provided by the combination of Vicinity and Cyclon views.

We also show that our three-layered architecture explained in Sec. Section 3.4 is essential to improve the speed of convergence. In Figure 3.5 we compare the convergence speed of PolderCast, without the Vicinity layer in the middle, and we can see that it takes almost 3-6 times longer to converge. This is because Vicinity provides interest-induced random links, essential for speeding up the construction process.

Apart from the speed it is also important to make sure that the overlay construction is scalable with respect to the number of nodes that participate in a ring (topic popularity) and the number of topics a node is interested in (subscription size). As shown in Figure 3.6, even a node interested in over 400 topics converges reasonably fast. This is mainly due to having a higher number neighbours compared to a node interested in a few topics only, which offer it much higher reachability for a large number of topics.

3.5.3 Overlay Degree

In Figure 3.7 we assess the effect of a node's subscription size on its Rings view size. Due to interest locality, a single neighbor may serve multiple of its topics. This helps the node retain its Rings outdegree low, and effectively contributes to higher scalability with respect to the subscription size of nodes. We do not consider the degree due to random links here since their number is fixed and small compared to that of ring links.

For the Twitter data, PolderCast manages to exploit correlation in the subscriptions to a large extent. However, for Facebook data, the node degree grows almost linearly with subscription size suggesting less subscription correlation. In Scribe, the average degree of a node v in the system is bounded by the number of nodes in the Pastry routing table that point to node v. This number is logarithmic with the total number of nodes and independent of the number of topics that node is subscribed to. This may be an important

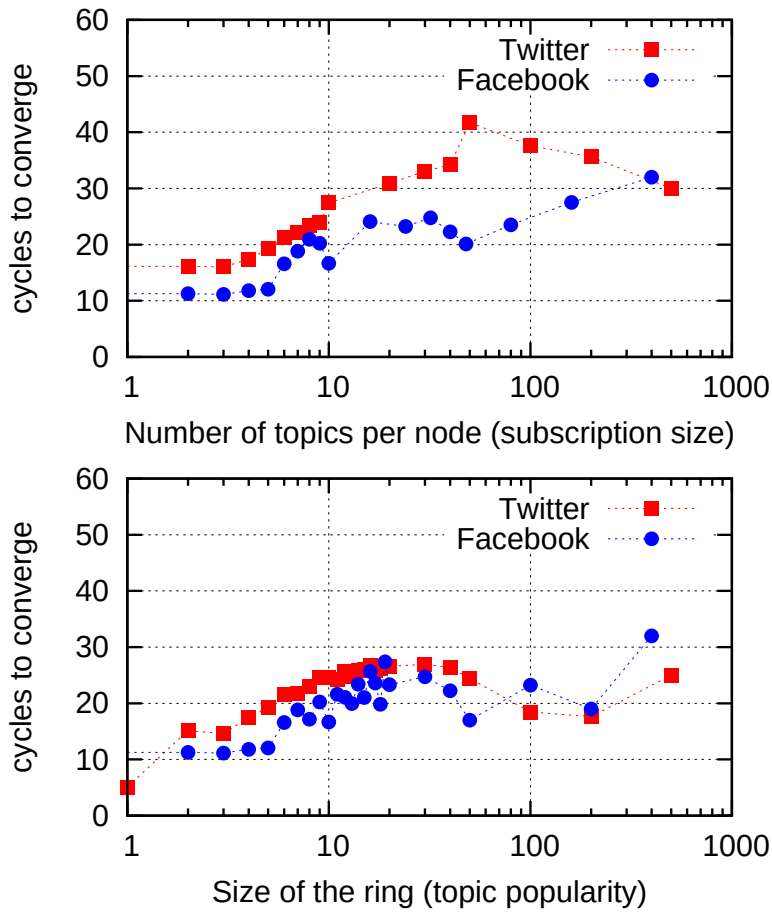

Figure 3.6: Correlation between convergence speed and size of the subscription/ring

advantage in the case of an extremely high number of topics a node is interested in.

3.5.4 Event Dissemination

We now analyze the event dissemination protocol proposed in Section 3.3.2. We measure (1) the dissemination delay, in terms of number of hops required for a publication to reach the subscribers and (2) the duplication factor, namely the ratio between the number of *all* event messages received over the number of

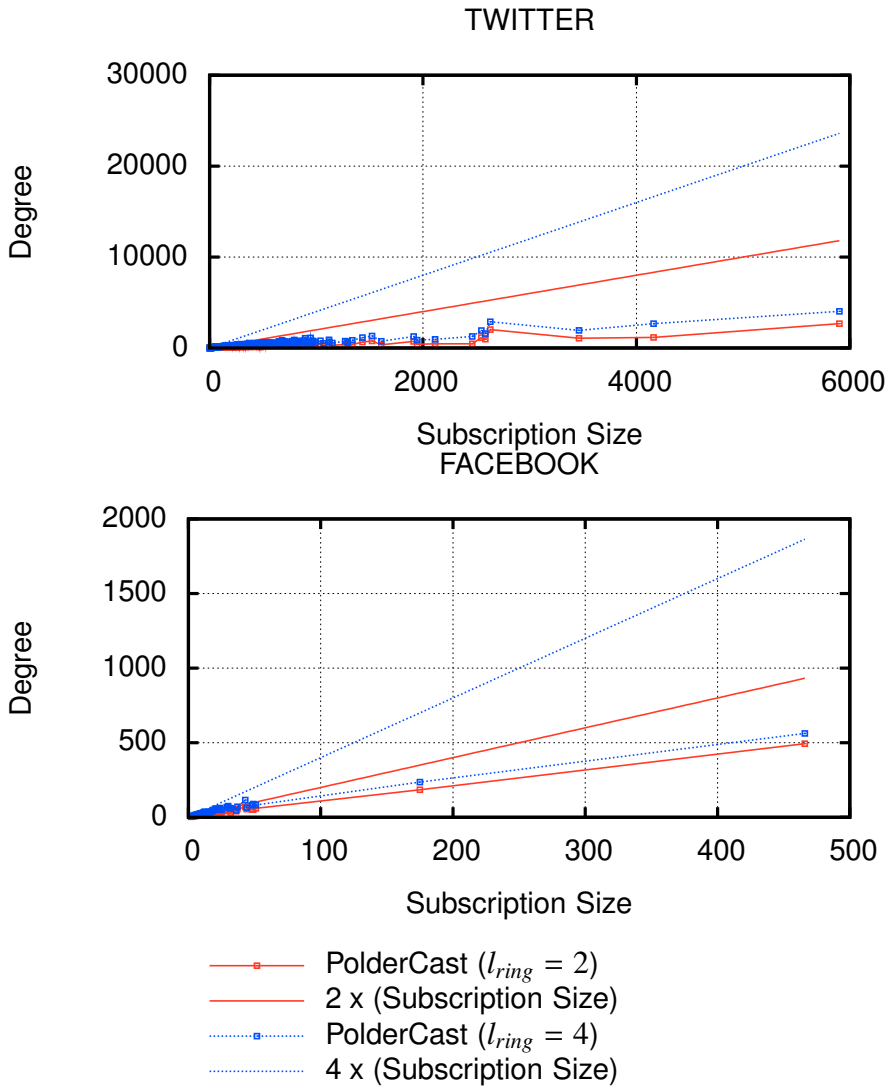

Figure 3.7: Node degree in Rings layer

distinct event messages received. The measurements were taken by injecting the publications as described earlier and averaging the two metrics for 1000 cycles. From this point on, we run PolderCast with only Facebook data with 10K nodes.

As one can see in Figure 3.8(a), with the increase in dissemination fanout the average dissemination delay significantly decreases. However, this decrease takes

place at the cost of an increase in the average number of duplicate messages seen by nodes as shown in Figure 3.8(b). To compare Scribe with PolderCast we plot the average delay in Figure 3.8(a). We can see that the average dissemination delay in Scribe is almost 1.7 times higher than the worst-case dissemination delay of PolderCast. This is due to the long chain of nodes induced by Scribe dissemination trees, even though DHT gurantees $\log |V|$ hops delay. These longer chains stem from the inclusion of relay nodes, both at the Scribe and Pastry level.

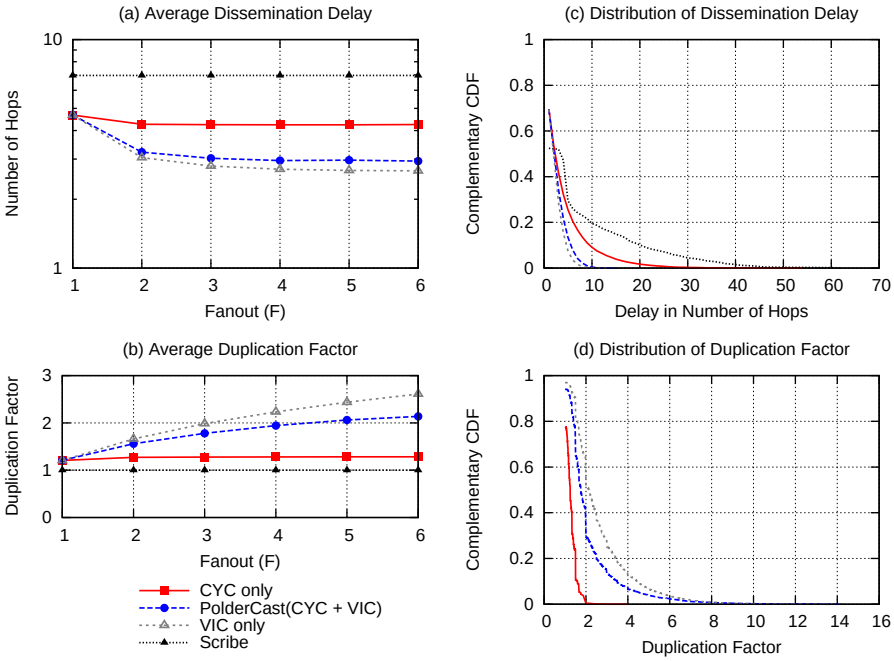

Figure 3.8: Event Dissemination Analysis

As shown in plots in Figure 3.8(a,b), the choice of random shortcut links has an interesting trade-off between dissemination delay and duplicate messages. At one extreme, if we use the Cyclon view as a source for random shortcut links, neither the dissemination delay decreases, nor the duplication factor increases with the increase in fanout f. This is attributed to the fact that since the Cyclon view is limited in size, and its view is chosen in an interest-agnostic way, the random shortcuts for a topic the node is interested in are not useful for the topics

of interest, forcing the dissemination protocol to fall back on ring links. On the other extreme, if we only use the Vicinity view as a source of random links, it leads to a significant decrease in average delay, at the cost of an increase in the average number of duplicates. In PolderCast we balance this trade-off by combining the Cyclon and Vicinity views, which results in the middle ground both for average delay and average duplication factor.

The choice of random shortcuts also has implications on the balancing of load on the nodes. In Figure 3.8(d) one can see that if only Vicinity is used for random shortcut links, around 20% of the nodes receive messages at least 4 times. This is due to the fact that nodes that are interested in many topics (> 100) have a high chance to be present in the Vicinity view of many nodes. Since we use both Vicinity and Cyclon views for random shortcuts, it reduces the number of duplicate messages for nodes interested in many topics. It should be noticed that Scribe does not have any duplicate messages since messages in Scribe are disseminated using multicast trees.

In Figure 3.8(c) we can see a similar pattern for dissemination delay and we again take the middle ground between the two extremes. Figure 3.8(c) also shows that there is a significant number of messages in Scribe with a relatively high dissemination delay, as we explained above.

3.5.5 Overlay Maintenance

The next experiment aims at evaluating the overhead in overlay maintenance. We measure the number of control messages sent and received by each node to maintain the overlay. Note that as shown in Figure 3.9 nodes interested in many topics (> 100) transmit a higher number of messages. This is due to the fact that they are more frequently selected as a target for gossiping. This factor does not play a significant role: the cycle duration can be chosen to be as high as 1 minute in real scenarios thereby rendering the bandwidth overhead negligible. On the other hand, more intensive control communication by nodes interested in many topics contributes to faster overlay convergence.

It is clear from Figure 3.9 that Scribe incurs a higher communication

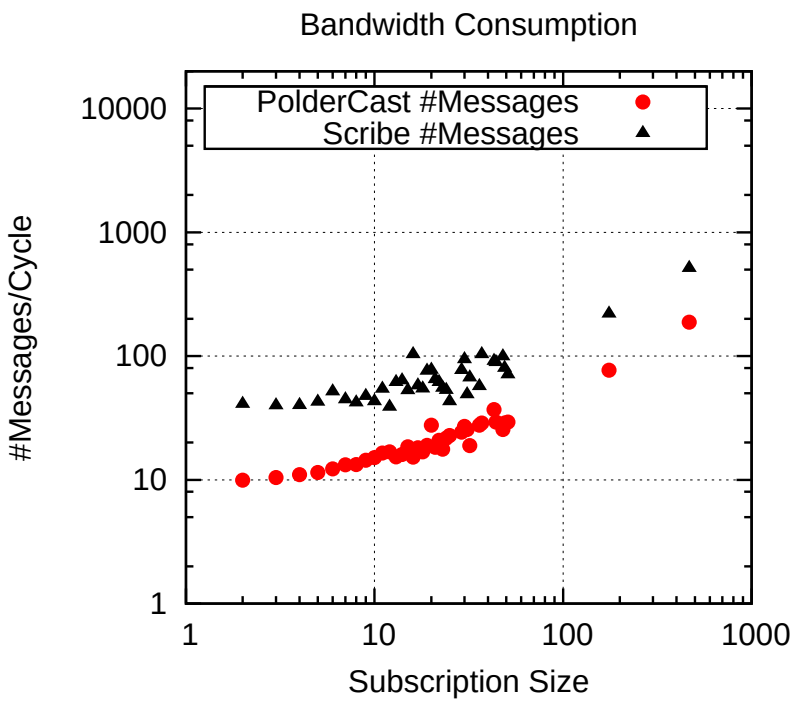

Figure 3.9: Bandwidth consumption

overhead. The number of control messages sent and received by a node v in Scribe is proportional to the number of subscriptions v is interested in. Even though each node has a limited number of children in the multicast tree to maintain, Scribe sends regular heartbeat messages for each topic (both topics of interest and topics for which v is a relay) to keep the trees connected.

The existence of relays and lack of topic-connectivity in Scribe additionally causes unwanted traffic passing through the nodes. We measure the amount of overall traffic (both control and application traffic) passing through each Scribe node and distinguish between the traffic relevant to the subscription topics of the node and unwanted traffic. In Figure 3.10 we show the amount of unwanted traffic at each node. We can see that over 90% of the nodes receive more than 80% of unwanted traffic. Such an overhead does not exist in PolderCast since topic-connectivity ensures that each node receives only the traffic relevant to the node's subscription topics.

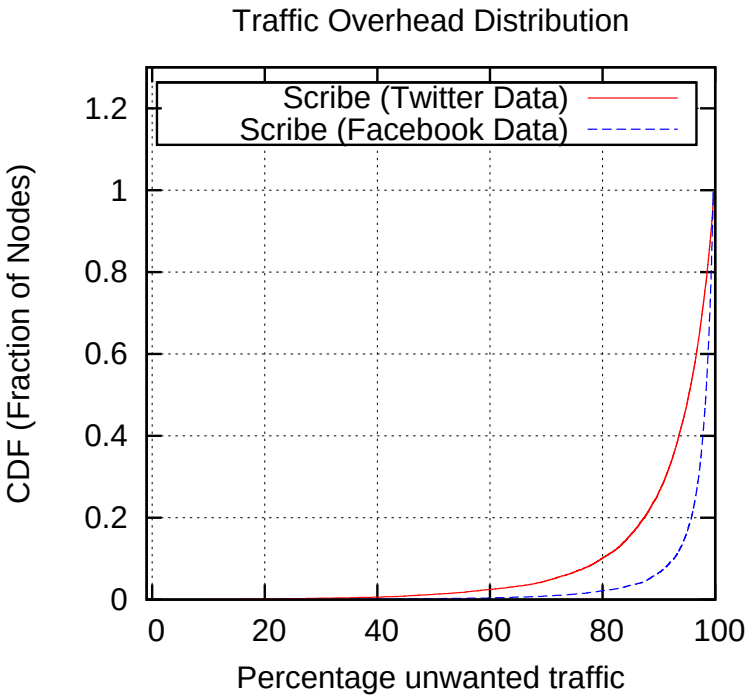

Figure 3.10: Traffic Overhead

3.5.6 Message Dissemination Under Churn

In this experiment we evaluate PolderCast and Scribe publication dissemination under the churn model described earlier. We inject publications as explained earlier with fanout f set to 2. We maintain two successors and two predecessors for each topic (ℓ_{ring}= 4). To assess the resilience of our protocol to node churn, at the end of each cycle we freeze the overlay and we measure the miss-ratio, i.e., the fraction of nodes that missed at least one publication event. It is worth noting that we set the cycle duration to be 1 minute. As a consequence, we introduce 60 times more node churn during each cycle than originally provided by the churn traces. When measuring the miss-ratio, we exclude the warm-up period of 10 seconds after the node joins the network.

As shown in Figure 3.11, for the Skype churn model the miss-ratio in

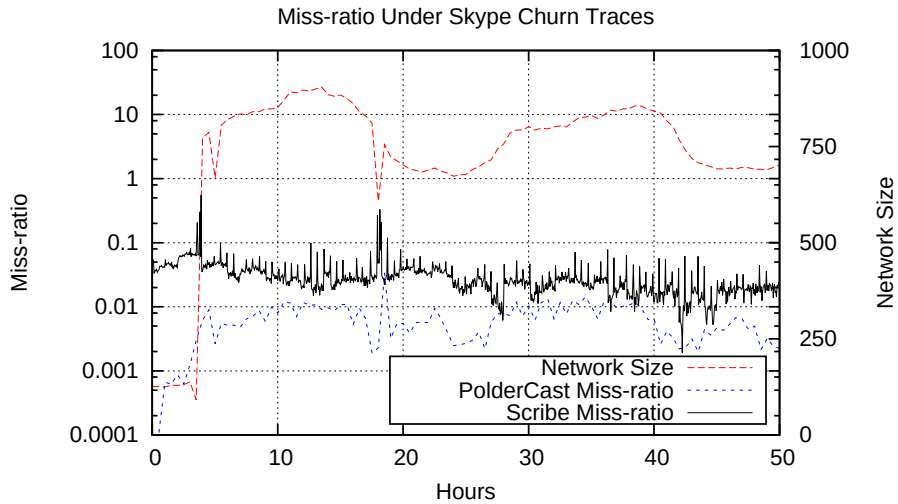

Figure 3.11: Message Dissemination Under Churn

PolderCast never grows beyond 0.01 except when there is a sharp drop in network size. In that case, the miss-ratio momentarily grows to 0.04, but stabilizes quickly. This is due to (1) the use of random shortcuts, keeping the dissemination structure connected even though the ring is partitioned, and (2) since ℓ_{ring}= 4, with the failure of one successor/predecessor the ring can still stay connected. When hundreds of nodes are joining the system (i.e., when there is a flash crowd), PolderCast continues to maintain the miss-ratio below 0.01.

From Figure 3.11 it can be seen that Scribe has almost 10 times higher miss-ratio than PolderCast. Especially during the flash crowd at the beginning Scribe has a significantly higher miss-ratio due to a slower construction of the multicast trees when around 600 nodes join. Similarly we can see a spike in the miss-ratio when a sharp drop in network size occurs after around hours 18. There is a spike in the miss-ratio of PolderCast as well, but the relatively higher miss-ratio of Scribe is caused by the sudden departure of several rendezvous nodes.

3.6 Summary

In this chapter we presented PolderCast, a P2P architecture for topic-based pub/sub which aims to achieve relay-free, fast and robust dissemination over a scalable overlay with a minimal maintenance cost. PolderCast achieves a delicate balance between these conflicting but desirable properties. We evaluated PolderCast with Scribe as baseline, using large-scale simulations with publicly available real-world traces from Facebook [Wilson et al., 2009] and Twitter [Kwak et al., 2010].

Chapter 4

Spotify Pub/Sub: Case Study of a Publish/Subscribe System to Drive Social Interaction

Spotify is a successful peer-assisted music streaming service that provides access to over 20 million tracks to its over 40 million active users residing in more than 56 countries[1]. The technical architecture providing the streaming service and user behavior of Spotify have been described in two recent studies [Kreitz and Niemela, 2010; Zhang et al., 2013a]. However, little has been said about the technical details of one of Spotify's most engaging features: its ability to facilitate the social interaction in the form of sharing and following various music activities among its users in real-time. As of January 2013, at any given point in time up to 10 million users were actively participating in social interaction. The social interaction traffic approximately amounts to 2TB of notification data per day.

In this chapter, we explain how the Spotify pub/sub architecture allows the users to follow playlists, artists, and the music activities of their friends. The distinctive feature of the architecture is that this entire range of social interaction is supported by pub/sub. Thus, the study in this chapter adds a new

[1]Information as of September 2014. For more recent information see: https://press.spotify.com/us/information/

unique application to a currently known list of large-scale systems that report benefits from using pub/sub, which includes application integration [Reumann, 2009], financial data dissemination [TIBCO], RSS feed distribution and filtering [Liu et al., 2005], and business process management [Li et al., 2010].

The end-to-end architecture of the pub/sub engine at Spotify is the focus of our study. The subscriptions are topic-based. The engine is hybrid: It allows relaying events to online users in real-time as well as storing and forwarding selected events to offline users who come online at a later point. The architecture includes a DHT-based overlay that currently spans three sites in Sweden, UK, and USA. The architecture is designed to scale: It stores approximately 600 million subscriptions at any given time and matches billions of publication events every day under its current deployment.

4.1 Spotify Pub/Sub Model and Features

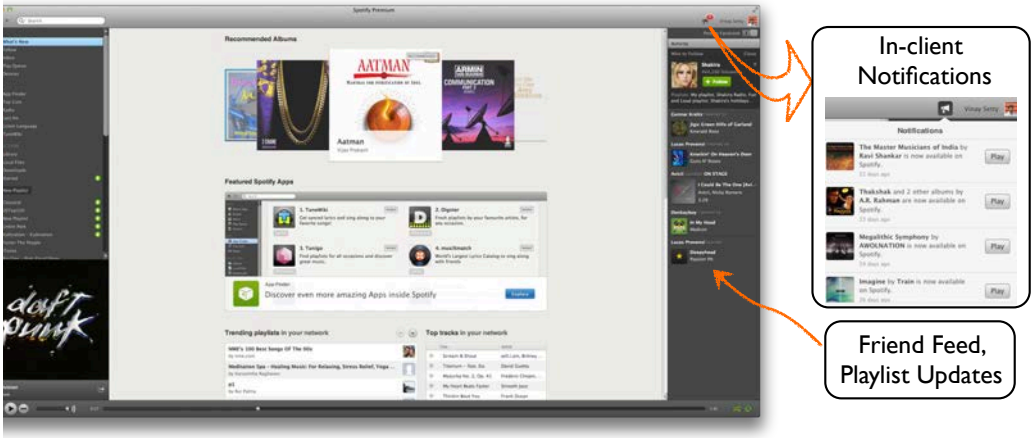

Figure 4.1: Spotify Desktop Client Snapshot

Spotify pub/sub follows the well-known topic-based pub/sub model. Users can subscribe (or follow) topics, which can be any of the following types:

Friends: Spotify allows its users to integrate with their Facebook account, and, once this integration is done, by default all Facebook friends who are also

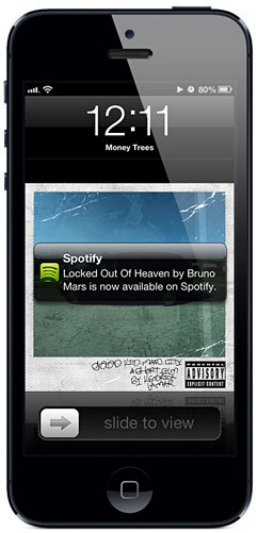

Figure 4.2: Push Notification

Spotify users become topics that can be followed. A Spotify user can also follow another Spotify user even if they have not integrated their Facebook account by finding each other by sharing music or Playlists.

Playlists: Playlists (collections of music tracks) in the Spotify system have URIs, allowing users to subscribe to playlists created by others. Additionally, a user can search for publicly available user-created Playlists within the Spotify client. Subscribing to a Playlist allows users to receive future updates to the Playlist. By default, a Playlist can only be modified by its creator, but a Playlist can also be marked as "collaborative", making it world writable.

Artist pages: Spotify has dedicated pages for each artist and allows users to follow them. This allows users to get notifications about new album releases or news related to the artist.

Any user can become a subscriber of the topics of the types mentioned above. A subscription is generally a pair of strings, the username of the subscriber and the topic name. The following are the publication events related to the above mentioned topic types:

Friend feed: When a user plays a music track, creates or modifies a Playlist, or marks an artist or a track or an album as favorite, an event notification is sent to all the friends following the user. Optionally, these events can also be published on the associated Facebook wall of the user. The friend feed can be seen at the bottom-right pane of the desktop client as shown in Figure 4.1.

Playlist updates: Whenever a Playlist is modified by adding or removing a track or renaming the Playlist, the subscribers of the Playlist are notified about the update via friend feed. The pub/sub system is also responsible for instantly synchronizing the Playlist information across all the devices of all the subscribers of the Playlist.

Artist pages: Whenever a new album related to an artist is added in Spotify and whenever a Playlist is created by an artist a notification is sent to all the followers of that artist.

It is worth mentioning that all the publication events mentioned above are delivered to subscribers in real-time (best-effort as well as guaranteed delivery) when the user is online, some of them can also be delivered as offline notification via Email, and they can be retrieved by the user in the future. For example, when a new album is added for a famous artist with millions of followers, (a) an instant notification event is sent to the Spotify client software used by all the followers of the artist who are currently online, (b) an email notification is sent to the offline followers, and (c) the event is also persisted so that current and future followers can retrieve the historical events related to the artist in the future. The persistence of the update is also essential to support multiple devices of the same user i.e. a user logged into one device may want to retrieve the notification on a different device at a later point in time.

4.2 Architecture for supporting social interaction

In this section we describe the technical architecture of the system that facilitates the social interaction between users based on the popular pub/sub

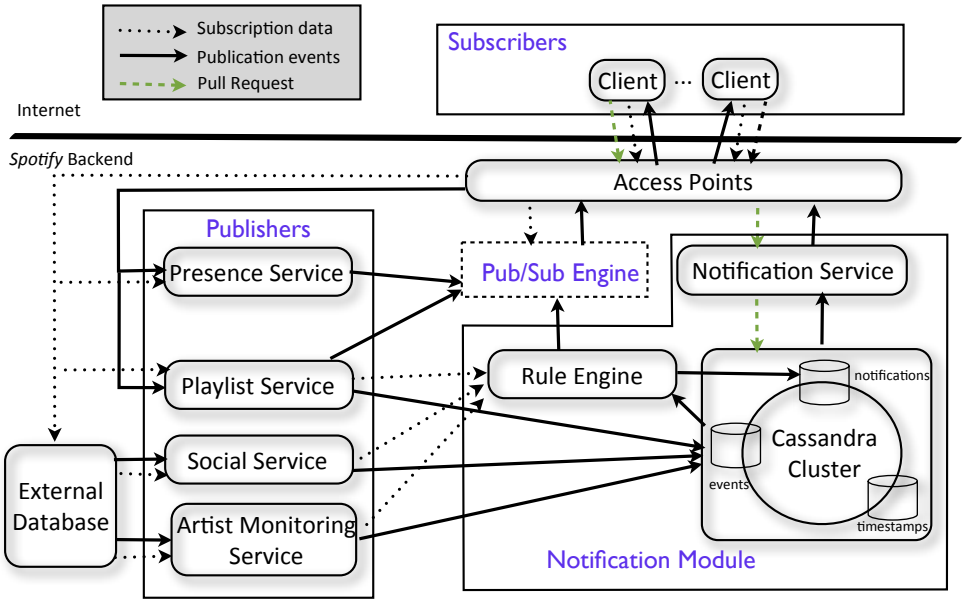

Figure 4.3: Architecture Supporting Social Interaction

communication paradigm. The pub/sub system at one end consists of publishers generating publication events and at the other end consists of subscribers, which are essentially Spotify clients. The pub/sub system is hosted across several datacenters (referred to as sites within Spotify). There are currently three sites: Stockholm - Sweden, London - UK and Ashburn - USA. These sites are not limited to hosting the pub/sub system, their main purpose is to host the music streaming service and all the backend services necessary for Spotify to function.

4.2.1 Architecture Overview

A high-level architecture consisting of subscribers, publishers, and two core components, the *Pub/Sub Engine* and the *Notification Module*, that are essential for enabling the social interaction between users is shown in Figure 4.3. The two core components are crucial for supporting high-performance real-time event delivery and reliable offline notifications in a resource-efficient manner.

Whenever designing a system for delivering publication events, the architects have to address a fundamental trade-off between latency and reliability. In order

to address this trade-off the system supports three essential event flow paths.

Real-time to online clients: The real-time delivery of events is done by the Pub/Sub Engine. However, the Pub/Sub Engine is lightweight i.e. it does not make the incoming events persistent, also there are no acknowledgments in place to detect failures, which results in a best-effort delivery of publication events without any guarantees but with low latency. Notice that in Figure 4.3 the Pub/Sub Engine directly receives input from three different sources: the Presence service, the Playlist service, and the Notification Module. Output is delivered to the subscribers via Access Points.

Persisted to online clients: The motivation for having this event-flow path for the delivery of publication events is purely based on the application requirement. The requirement is that some publication events like an album release or a Facebook friend joining Spotify are classified as critical for the users, and these critical publications must be delivered reliably, and at least once across all devices. This event flow path is realized by the Notification Module by storing the incoming publication events in the Cassandra cluster [Lakshman and Malik, 2010] for reliable and offline delivery. This will be explained in detail later in Section 4.2.3.

Persisted to offline clients: Whenever a client comes online, it can retrieve the publication events from the Notification Module by sending a pull request with the time-stamp of the last seen event. This path is shown in Figure 4.3. The client may receive the same notification twice: once when it was online last time and another time when it came back online. However, the client software can distinguish already seen publications using the time-stamp of the publications.

4.2.2 Subscribers and Publishers

The *Access Points* (APs) act as an interface to all the external clients. From a pub/sub perspective, APs are responsible for relaying client join/leave messages to

Table 4.1: List of topic types and corresponding services on Spotify pub/sub

Topic Type	URI	Service	Notification Type
User	hm://presence/user/<user-name>/	Presence	Friend feed
Playlist	hm://playlist/user/<user-name>/playlist/<playlist-id>/	Playlist	Friend feed, In-Client, Push and Email
Artist	hm://notifications/feed/ artist-id%notification-type/	Artist Monitoring	In-Client, Push and Email
Social	hm://notifications/feed/ username%notification-type/	Social	In-Client, Push and Email

various services, relaying subscription/unsubscription requests from clients to the pub/sub service, and relaying publication messages from the pub/sub service to clients. The APs are responsible for maintaining the mapping between the TCP connection to the client software and the topics and vice versa. This mapping is crucial for relaying subscriptions, unsubscriptions and publications.

All *Subscribers* in the pub/sub system are client software instances running on user devices. The client is a proprietary software application available for several desktop and mobile devices. A snapshot of the desktop client is shown in Figure 4.1. There are two ways of subscribing to a topic: Firstly, when a user explicitly follows a particular user, artist or playlist from the client interface and secondly, the social relations established from Facebook connections. In the former case subscription to the topic is done explicitly, while in the latter case subscriptions are done implicitly.

Whenever the client subscribes to a topic, the subscription information is sent to the Access Points. This information includes the user name of the subscriber and the corresponding URI for the service, as listed in Table 4.1. However, since the subscription information is needed by both the Pub/Sub Engine and the Notification Module there are two distinct subscription flow paths:

Subscriptions to the Pub/Sub Engine: The client sends a list of topics and

the URI of the relevant service to an AP, which are eventually forwarded to the Pub/Sub Engine.

Subscriptions to the Notification Module: If the subscription request is for the Social service, the Artist Monitoring Service or the Playlist service, the request is forwarded from an AP to the respective services. These services are then responsible for providing the subscription information to the Notification Module.

The *Publishers* of the pub/sub system are services running in the Spotify sites. All publications for the topics mentioned in the previous section are generated from four services, listed below and shown in Figure 4.3. The specific topics for these services are used in the form of URIs for communication and matching purposes and they are listed in Table 4.1. These URIs use a protocol internal to Spotify, denoted hm (Hermes).

The Presence Service is responsible for receiving friend feed events generated by users from client software. Whenever a user takes an action to trigger friend feed (as described in Section 4.1) the client generates a message to the user topic type via APs. The Presence service then stores the event in main memory and forwards the received event to the Pub/Sub Engine (shown in Figure 4.3 and explained in detail in Section 4.2.4) to be matched and delivered to the client software of the subscribers. All the events from the Presence service that are intended for the subscribers of a user are delivered to clients in real-time in a best-effort manner (i.e., no fault-tolerance techniques are used and hence no delivery guarantees). Also, Presence events are not persisted in secondary memory. Instead, only the last seen event is stored in main memory, due to the significantly higher volume of traffic compared to other services. All Presence events can be seen at a friend feed pane at the bottom-right corner of the Spotify desktop client software as shown in Figure 4.1.

The Playlist Service is mainly responsible for tracking playlist modifications made by users. As explained in Section 4.1, a playlist can be subscribed to

in two ways: a user can explicitly subscribe to playlists, and, in addition to that, by default all users are also subscribed to the playlists of their friends. The Playlist service treats the publications for these two types of subscriptions differently. playlist updates from friends are shown in friend feed and are delivered via the Pub/Sub Engine, and the rest are delivered via the Notification Module. The playlist service also provides subscription lists (i.e., given a playlist, all the subscribers of the playlist; and, given a user, all the subscribed playlists of the user).

The Social Service is responsible for managing the social relations of Spotify users as well as integration with Facebook. The Social service generates a publication event when a Facebook friend of an existing user who is not already using Spotify joins Spotify. It also provides an interface to obtain all the friends of a user who are subscribers to the friend feed from the given user. Finally, it is also responsible for posting user activities on the Facebook wall for those users who opted for this feature.

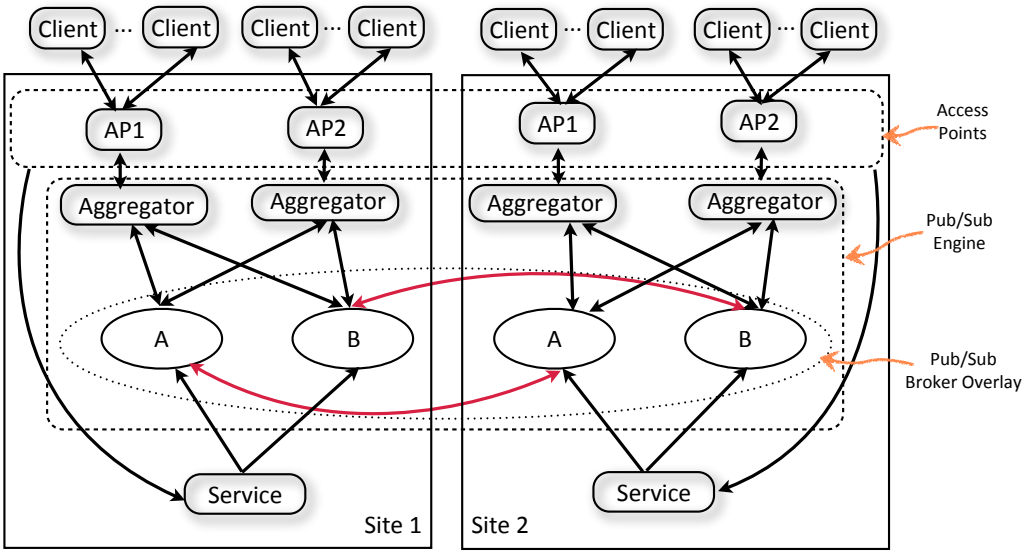

Figure 4.4: Real-Time Pub/Sub

The Artist Monitoring Service is responsible for generating publication events whenever there is a new album or track for an artist and new

playlists created by an artist. Note that this service is essentially a batch job running at regular intervals (typically once a day) that queries an external database to detect any new album releases for the artist.

A summary of all topics types that can be subscribed by the clients and the corresponding services producing publications are listed in Table 4.1.

4.2.3 The Notification Module

The publication events for all the topics are delivered to clients in several ways. The Notification Module receives the publication events from all services, except the Presence service, and then classifies them and delivers them to the subscribers in the form of the following notification types:

In-client notification: Some events like artist updates and new Facebook friends joining Spotify are shown in a notification icon at the top-right corner of the Spotify desktop client, as shown in Figure 4.1. Note that unlike friend feed, in-client notifications are persisted for guaranteed delivery.

Push notifications: Push notifications are for mobile devices. The Notification service forwards the events to the corresponding push notification services provided by the vendors of the user devices. An example of the push notification is shown in Figure 4.2.

Email notifications: When a user is not online, events like artist, playlist and friend updates are sent via email excluding the users who have opted out of this service.

A summary of the topics and the notification types with which they can be delivered to the subscribers is listed in Table 4.1.

An important component of the Notification Module is the Rule Engine. It has the logic for classifying every publication event into one of the above mentioned notification types. The rules are embedded in the Rule Engine, but

the subscription information is obtained from the respective publication services. The rules are based on the following parameters:

- Online status of the user.
- Client device type (desktop or mobile).
- User subscription preferences on email notifications.

Depending on the notification type, the Rule Engine will forward the publication event to the Pub/Sub Engine and Cassandra for persistence.

Publication Event Persistence

The motivation for persistence of publication events is driven by the following goals: reliable delivery of publications, offline delivery and future retrieval of publications, and a smooth way to deliver publication events to the same user but using clients from different devices. All publication events generated from the playlist, Social and Artist services are persisted in a Cassandra cluster in a column family called *events*, as shown in Figure 4.3. It is worth noting here that each publication event is stored as (topic, subscriber) pairs in the Cassandra cluster. This is a significant blowup of data for the topics with millions of subscribers. Since the persistence of these events requires significant storage and computing resources the following measures are taken:

- Presence events, which are of significantly higher volume (as shown with workload analysis in Section 5.1), are not persisted.
- Each publication event in the *events* column family has an expiry date of 90 days by default (i.e., no events are retained over 90 days).

Once the events are written to the *events* column family, each event is processed by the Rule Engine, which constantly polls *events* and detects the new events. Based on the generated rules, the Rule Engine decides if the events are to be sent to the Pub/Sub Engine for real-time delivery or written back to the Cassandra cluster but to a different column family called *Notifications* along with the notification

type to be used. The *Notification service*, which polls the *Notifications* column family, delivers the publication events using the notification type suggested by the Rule Engine.

Finally, to support pull requests from clients, the column family *Timestamps* is used for keeping track of the time-stamp of the last seen event for each client. Whenever a client connects to an AP, a request is sent to the Notification service with the time-stamp of the last seen event, and the Notification service responds with all publications that were generated after than the given time-stamp. Note that time synchronization is not a problem here since the clients adhere to the clock of an AP. The time-stamp check also helps avoid duplicate delivery of publication events and, once a notification is read on one device it will be shown as read in all the other devices of the same user.

4.2.4 Pub/Sub Engine

The Pub/Sub Engine consists of *Aggregators*, responsible for aggregating subscriptions and distributing publications. The core component of the Pub/Sub Engine is a DHT overlay of broker servers managing subscriptions, publication matching, and delivery. A diagram with the different components of the Pub/Sub Engine is shown in Figure 4.4.

The *Aggregators* sit between the APs and the pub/sub broker overlay. When a client connects to Spotify via an AP, it also sends a set of subscriptions by sending all the friends, playlists and artists the user is interested in. Each topic-subscriber pair is considered a separate subscription. All subscriptions are managed for matching purposes in main memory. In order to scale w.r.t. the number of subscriptions and publication events, the Aggregators are crucial. The Aggregator locally aggregates all the subscriptions for a given topic and sends a single subscription on their behalf to the pub/sub broker overlay. The Aggregator distributes the publication to the APs in the reverse direction and it is also responsible for hashing the subscription to a respective broker in the pub/sub broker overlay.

The *pub/sub brokers* are organized as a DHT (Distributed Hash Table) overlay

with the subscription as the key. The overlay of pub/sub brokers have the following responsibilities:

Managing subscriptions: pub/sub brokers are responsible for receiving subscription requests from Aggregators and storing the subscriptions in main memory. The brokers are responsible for maintaining the mapping between the topics and the corresponding Aggregator where the subscription came from. This mapping is absolutely crucial for routing the publications to the right Aggregator. Pub/sub brokers also receive unsubscription requests for a topic from the Aggregators when there are no more online subscribers for that topic.

Matching publications: pub/sub brokers match the incoming publications from the publisher services against in-memory subscriptions.

Forwarding matched publications: Once the matching entries are found the publication is forwarded to all the corresponding Aggregators.

Cross-site forwarding: The broker overlay is also responsible for forwarding publications to a different site if there are any subscribers. Note that the pub/sub broker overlay spans all the sites.

Each broker at a site has a one-to-one corresponding broker in other sites which exchange their subscriptions and publications from the corresponding sites. For example, as shown in Figure 4.4, broker A at site 1 has a corresponding broker A at site 2 (i.e., all the subscriptions obtained within site 1 and managed at broker A, are also forwarded and replicated to the corresponding broker A of site 2 and vice versa). Whenever there is a publication for a subscription at the broker A of site 1, if there is a matching subscription registered from the broker A of site 2, the publication is forwarded via a cross-site link to the broker A of site 2. Then broker A of site 2 forwards the publication to the corresponding subscriber at site 2 via an AP. This cross-site DHT overlay of pub/sub brokers facilitates interaction among Spotify users that follow each other but are connected to different sites.

Load Balancing: Since all subscriptions are main memory, it is crucial to have a scalable solution to manage them. The DHT organization of the pub/sub brokers is the key to scale in-memory storage of over 600 million subscriptions. The pub/sub broker overlay is also designed to distribute the load publication matching and forwarding load among the brokers.

4.3 Summary

In this chapter, we presented the architecture of a system that allows Spotify users to follow playlists, artists, and the music activities of their friends. The architecture is realized by pub/sub, a popular communication paradigm. We described how a hybrid system with a scalable Pub/Sub Engine driven by a DHT overlay of brokers that facilitates real-time delivery of events and also a Notification Module to persist important events for offline notification as well as future retrieval of events.

Chapter 5

Pub/Sub Workload Analysis

In this chapter, we study the pub/sub workload from traces recorded at Spotify and collected from the Twitter APIs.

For Spotify traces, the objective of the study is twofold: first, we characterize the workload of the pub/sub system in terms of event publication rates, topic popularity, subscription sizes, normalized notification rate per subscriber. Unfortunately, there exist precious few characterizations of subscriptions and synthetic workload generators for pub/sub systems [Yu et al., 2009]. In view of this shortage, the value of our characterization is that it can be used towards corroborating the validity of synthetic workloads as well as their generation. One particularly surprising finding that we explain in this chapter is that the event publication rate for a topic is not correlated with the topic popularity.

The second goal of the study is to analyze the message traffic produced by the pub/sub system and derive trends and patterns. In particular, we study the temporal patterns of subscription rate, unsubscription rate and publication event generation rate. From the observations we also conclude that the traffic due to the activity of following friends dominates the total traffic of social interactions.

For Twitter traces, we focus only on the first goal since we could obtain the data only from public APIs for analyzing publication event rates, topic popularity, subscription sizes and normalized notification rate per subscriber.

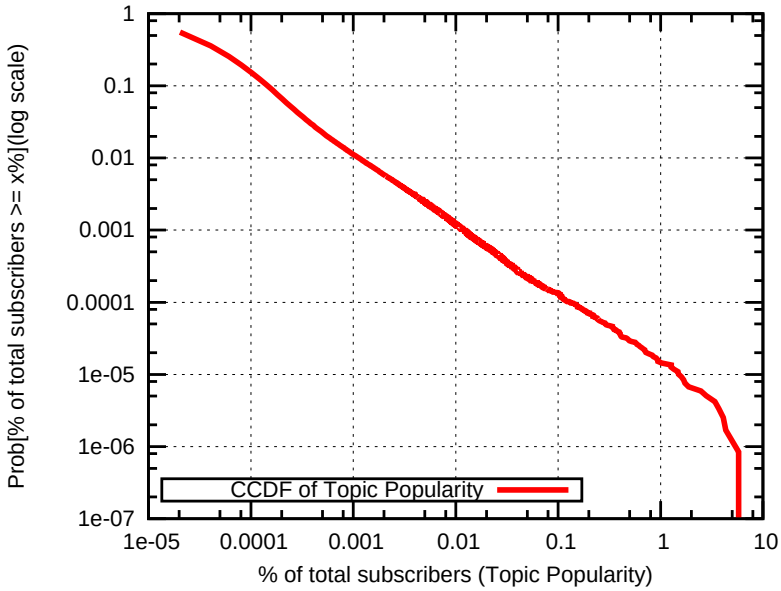

Figure 5.1: CCDF of Topic Popularity

5.1 Analysis of Spotify Pub/Sub Workload

In this section we study the different characteristics and patterns emerging from the pub/sub workload at Spotify. The main goal of the study is to characterize the workload used by a deployed pub/sub system, thereby serving as a reference for workload-modeling purposes in the pub/sub community in both industry and academia. Another goal of this study is to analyze the message traffic produced by the Spotify pub/sub system and derive trends and patterns.

All the results presented here are based on traces collected from production data. The traces were collected during 10 days from Thursday, 10 Jan 2013 to Saturday, 19 Jan 2013.

5.1.1 Analysis of Traces From The Presence Service

In this section, unless explicitly mentioned, we study the subscriptions and publications given as input to the Presence service. We restrict our analysis to the Presence service due to its dominance of the pub/sub workload in Spotify, which is illustrated later in this section. In order to simplify our analysis, in

these experiments, we consider only users with desktop clients, who have been online at the Stockholm site and have produced at least one publication in the studied time period, and their corresponding subscribers.

We study the following characteristics of the workload:

- The distribution of **Topic Popularity**: The Complimentary Cumulative Distribution Function (CCDF)[1] of the percentage of the total number of subscribers subscribing to a topic, shown in Figure 5.1.
- The distribution of **Subscription Size**: The CCDF of the percentage of total number of topics subscribed by a single subscriber, shown in Figure 5.2.
- The distribution of **Publication Event Rate (per topic)**: The CCDF of the percentage of total publication events generated for the chosen time period, shown in Figure 5.3.

In Figure 5.1, it can be seen that the log-log plot of CCDFs of topic popularity resembles a straight line until very low values of topic popularity (10^{-5}%). Similarly, the log-log plot of CCDF distribution of subscription size in Figure 5.2 resembles a straight line as well until subscription size of at least 10^{-6}%.

This indicates that topic popularity and subscription size distributions in Spotify pub/sub may follow power-law-like distributions similar to node degree distributions in typical social networking graphs [Mislove et al., 2007]. This behavior is due to the fact that subscriptions and topics in Spotify pub/sub are predominantly defined by the social relations between Spotify users. In addition, as mentioned in Section 4.1, it is known that when a Facebook friend of a Spotify user joins Spotify, by default they become subscribers of each other. This observation motivates the use of social graphs as workloads for academic works on topic-based pub/sub systems as done in [Chen et al., 2013; Rahimian et al., 2011; Zhang et al., 2013b].

[1] CCDF is the probability of a random variable X to be greater than a given value y

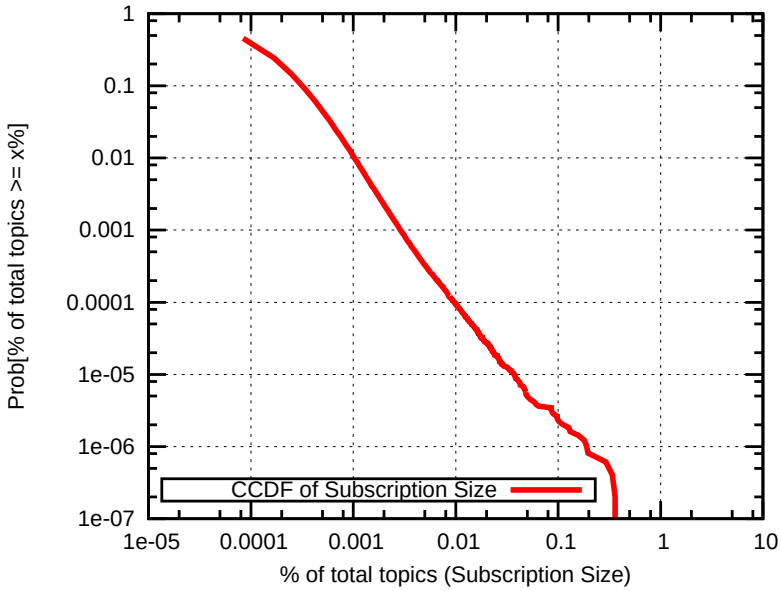

Figure 5.2: CCDF of Subscription Size per user

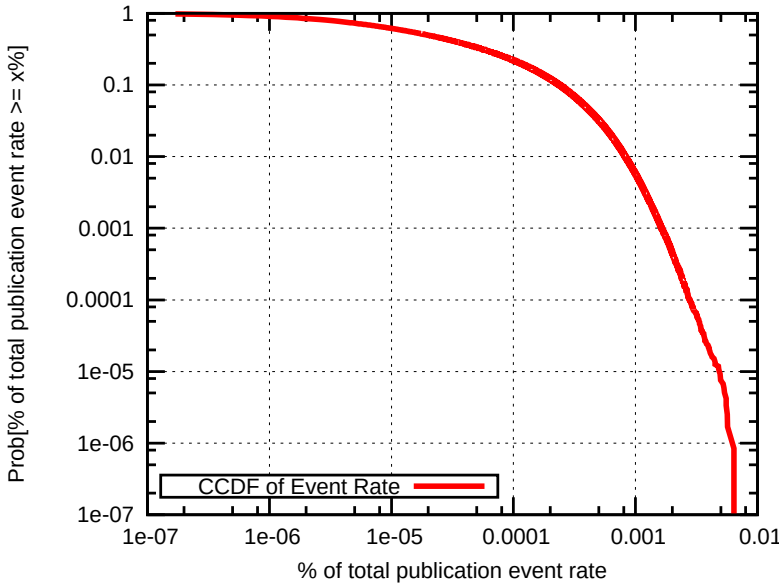

Figure 5.3: CCDF of Publication Event Rate per topic

The CCDF of Publication Event Rate in Figure 5.3, on the other hand, does not follow a power law. There is a sharp deviation around 0.0005% of the total

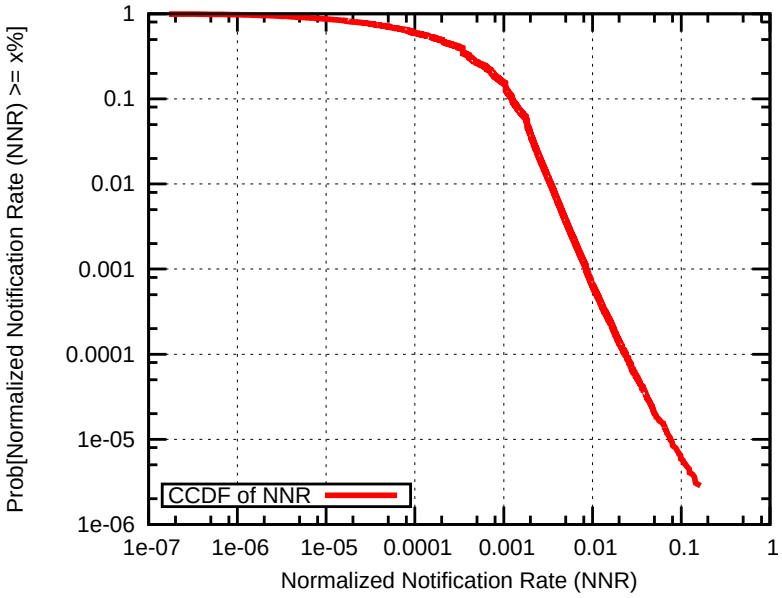

Figure 5.4: CCDF of Normalized Notification Rate per user

number of publication events.

Next we study the distribution of the number of publications attracted by each subscribers. We call it *Normalized Notification Rate* per subscriber (NNR_v), which we define as the percentage of total publications events matching the topics subscribed by a subscriber. NNR per subscriber is similar to the subscription cardinality metric proposed for content-based subscriptions by [Li, 2010]. The only difference is that NNR is defined per subscriber while subscription cardinality is defined for each subscription and each subscriber can subscribe to multiple subscriptions. It is mathematically expressed as below:

$$\text{NNR}_v = \frac{\sum_{t \in T_v} ev_t}{\sum_{t' \in T} ev_{t'}} * 100$$

Where, T is a global set of all topics, $T_v \subseteq T$ is a set of topics subscribed by a subscriber v, ev_t is the publication event rate of topic t. Thus, a subscriber with a Normalized Notification Rate of, for example, 0.1%, receives 0.1% of all publications in the system.

In Figure 5.4 the x-axis shows the Normalized Notification Rate (NNR) values

and the y-axis the probability that a subscriber has the Normalized Notification Rate value greater than or equal to that of the corresponding value shown on the x-axis (in other words CCDF). This distribution is an interesting result for the pub/sub community since the distribution of number of events received by each subscriber is an important design parameter for many pub/sub systems [Li, 2010] and they are generally estimated probabilistically. Our analysis shows a diverse Normalized Notification Rate ranging from 0.2% to as low as 10^{-7}%. In addition, more than 90% of the subscribers have NNR < 0.001% which appears to us as being very low.

Each subscriber is allowed to subscribe to an arbitrary number of topics. That results in arbitrary subscription sizes for the subscribers. A study about the correlation between subscription sizes and the corresponding matching events is crucial for understanding the resources needed to handle the publication traffic at the brokers. We do this study by considering each subscriber's Normalized Notification Rate and the corresponding subscription size. We show in Figure 5.5 that, as the number of topics followed by a subscriber (i.e., subscription size) increases, the number of publications received by the subscriber (i.e, Normalized Notification Rate) increases linearly. In Figure 5.5, we show only a 1% random sample of all the points.

As suggested earlier, the subscription workload for the Spotify pub/sub system is characterized by a social graph. However, when we study the topic popularity (number of subscribers of each topic) and the corresponding publication event rate for that topic, we see no correlation at all. i.e. a topic with very few subscribers can lead to significantly more publications than topics with many subscribers. This behavior is shown in Figure 5.6. We conjecture that the reason for this is that, unlike social networks, the activity in Spotify pub/sub is determined by the music-listening behavior of users. This implies that a frequent listener of music in Spotify does not necessarily have a high number of subscribers, similarly a user with many subscribers is not necessarily a frequent listener of music. We leave the confirmation of this conjecture for future research. Again we show a 1% random subset of the original data points.

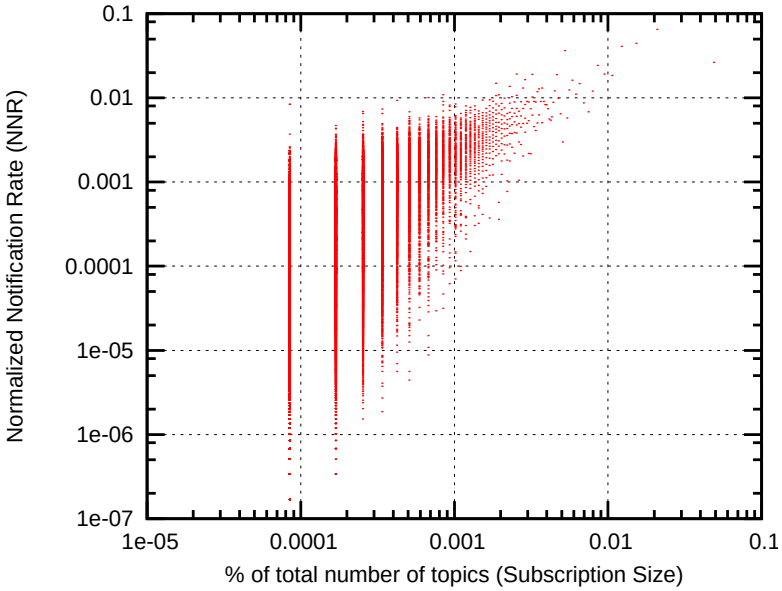

Figure 5.5: Correlation between Normalized Notification Rate and subscription size (% of total number of topics)

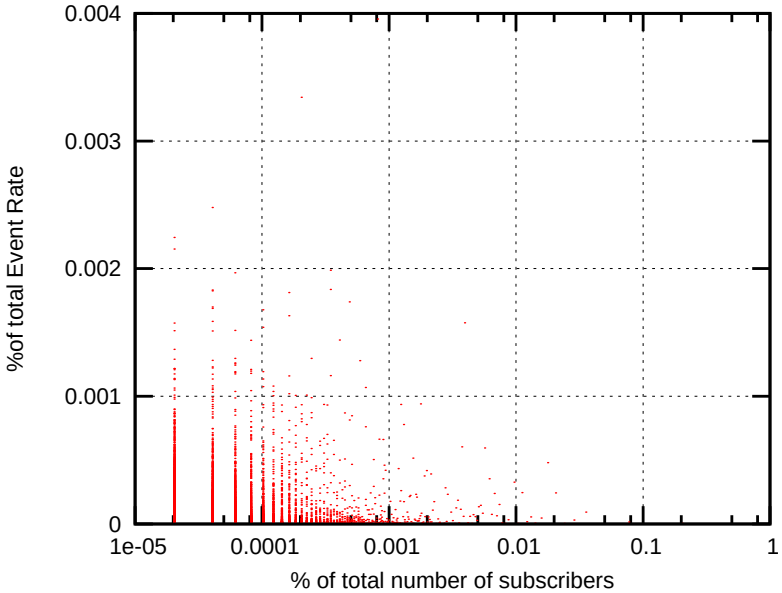

Figure 5.6: Correlation between topic Popularity (% of total number of subscribers) and Publication Event Rate

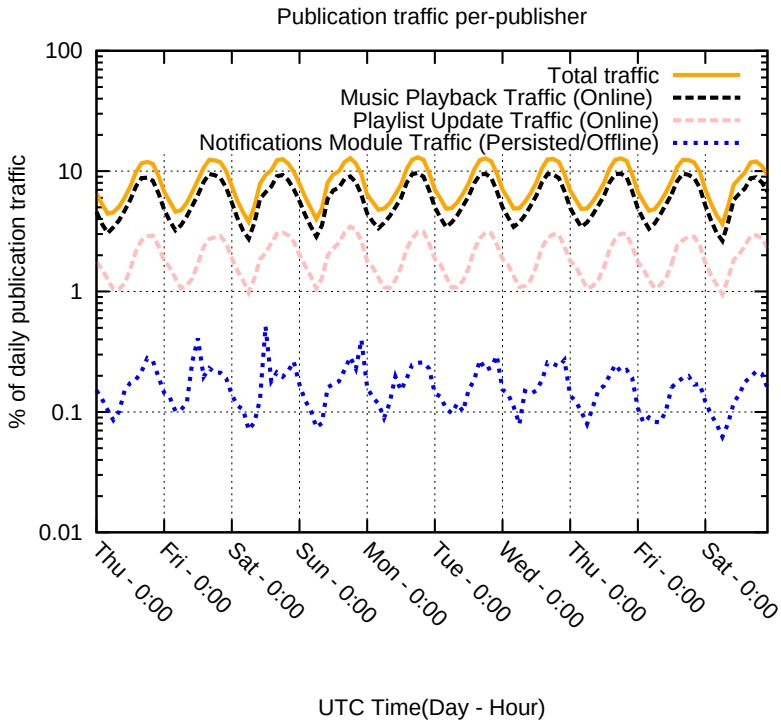

Figure 5.7: Pattern of publications generated per service-basis

5.1.2 Pub/Sub Traffic Analysis

The following measurements correspond to all the publisher services mentioned in Section 4.2 and are not limited to the Presence service. These traces also include mobile users in addition to the desktop users. We also include traces for these measurements for the same 10 days mentioned earlier.

Publication Traffic

First we study the distribution of publication traffic by separately decomposing it per service.

The Presence Traffic: From Figure 5.7 it is easy to observe that, for the Presence service, there is a periodic pattern of publication traffic on a daily basis with peak traffic towards the evening around 6 PM and the lowest traffic around 2 AM. Further, the traffic is slightly lower during weekends

compared to weekdays. Without further analysis it is easy to see that this pattern is similar to the pattern for playbacks as observed in [Zhang et al., 2013a]. The reason for this pattern is simply that the publications generated by the Presence service are generated due to the playback of music tracks. It is easy to observe from Figure 5.7 that Presence events form the majority of the publication traffic.

Playlist Traffic: There is a similar daily periodic pattern in the Playlist publication traffic, with highest traffic around 6 PM and lowest traffic around 2 AM. However, in contrast to the Presence service, the Playlist service traffic has slightly higher traffic on Sunday compared to the weekdays.

Notifications Traffic: For Notifications traffic, which includes updates to artist pages and updates from the Social service, one can observe small spikes with notifications every day stemming from batch jobs launched for artist updates. In Figure 5.7, it can be noticed the notification module generates significantly low traffic in comparison with the traffic generated from the Presence service and the Playlist service. This observation is consistent with the hybrid design principle: real-time notification for the Presence service and offline notification for artist pages and social updates.

In Spotify, as mentioned earlier the users are connected to three different sites. The need for forwarding pub/sub notifications across sites arises since users in one site sometimes subscribe to users or artists connected to other sites. To observe this, we compare the total publication traffic (from all services) generated and notified within the same site (local site) against the publication traffic generated and forwarded from the rest of the sites (called remote sites). Remote traffic is due to the music activity of users in a remote site for which there is at least 1 subscriber in the local site. As we can observe from Figure 5.8, remote traffic is nearly an order of magnitude lower than local traffic. This observation is in accordance with the design of Spotify pub/sub as described in Section 4.2.4. Specifically, the pub/sub system at each site is designed to handle high local traffic, assuming that

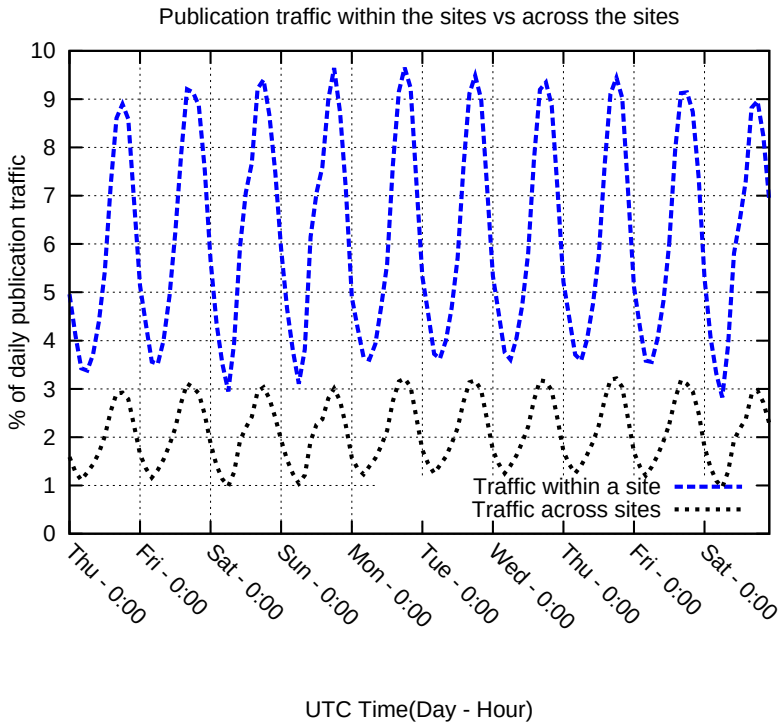

Figure 5.8: Publication traffic within the sites vs across the sites

the remote traffic forwarded across sites is significantly lower.

Subscription Traffic

Figure 5.9 shows the pattern of subscriptions and unsubscriptions. There is a periodic pattern in subscriptions and unsubscription rates as well, and this is due to users joining and leaving Spotify at regular intervals. This periodic churn behavior can help model the churn of subscribers in a pub/sub system. Many research works [Castro et al., 2002; Li, 2010] in the area of pub/sub use synthetic churn workloads or adapt churn traces from other peer-to-peer systems like file-sharing services or Skype. In this section, we characterize churn using traces from an actually deployed pub/sub system. Again, similar to publication traffic, subscription requests exhibit a daily pattern of evening peaks and early morning troughs as well. However, the weekly pattern of subscription traffic is

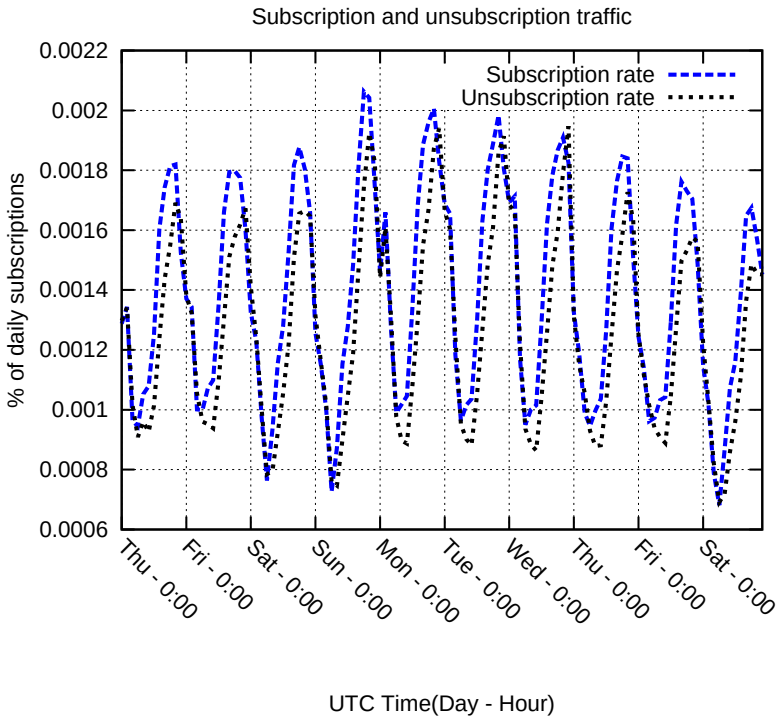

Figure 5.9: Subscription and unsubscription rate

significantly different from the weekly pattern of publication traffic. This is because the subscription traffic is a result of users logging in and out of the system while publication traffic is due to the playback of music.

From the Figure 5.9 it can also be observed that the rate of subscriptions and unsubscriptions match approximately, hence the change in number of subscriptions for the chosen 10 day period is negligible. This hypothesis is confirmed by Figure 5.10, which shows that there is little variation in the number of subscriptions for the chosen time period. In Figure 5.10 we can also see that the number of subscriptions is dominated by the Presence service. This is because when users register with Spotify for the first time, they have more subscriptions from following their friends than from following Playlists, artists and album pages. This also confirms our previous claim that the Presence traffic dominates Spotify pub/sub traffic.

In the traffic plots even though the data is aggregated from multiple sites in

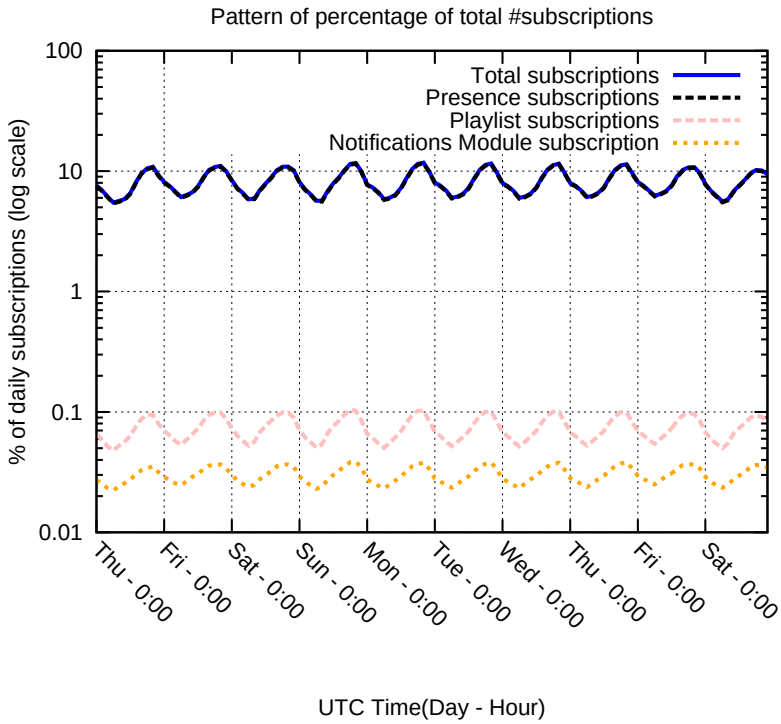

Figure 5.10: Pattern of percentage of total number of subscriptions

different timezones, we only see single peaks and troughs every day. This is because when the traces were collected (January 2013), Spotify pub/sub traffic was dominated by activity at one of the sites, while activity at other sites were negligible. Moreover, the users from a same timezone are not guaranteed to connect to the site in the same timezone due to load balancing.

5.2 Analysis of Twitter Traces

We used the Twitter social graph made publicly available by [Kwak et al., 2010], in conjunction with the information about the number of tweets that we crawled ourselves. Since the Twitter user ids in this data set are real user ids, we made use of the public Twitter APIs to obtain the number of tweets of each user in the data set from 30th Oct 2013 to 9th Nov 2013. We consider all the Twitter users who tweeted at least once during those 10 days (active users) and omit the rest. This

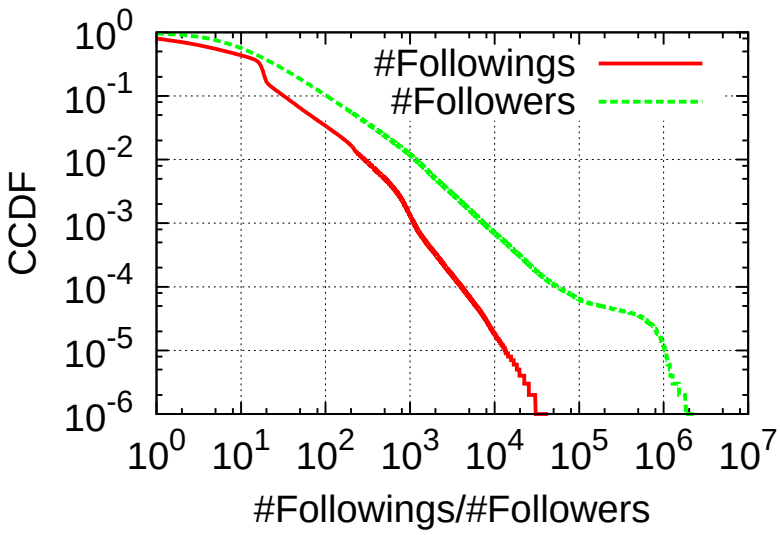

Figure 5.11: CCDF of #Followers and #Followings

process provided us with around 8 million active users and their corresponding 30 million subscribers. This data trace can be downloaded from the link provided[2].

In the first set of experiments we analyze the characteristics of the number of follower/following distributions and show that our sample is representative of the original data set. This can be verified from the Complementary Cumulative Distribution Function (CCDF) of the number of followings in Figure 5.11. The distinctive anomalies observed in [Kwak et al., 2010] at 20 and 2000 followings can be seen here too. The glitches indicate the default values in the number of followings and restrictions on number of followers imposed until 2009 respectively. There are around 550 users following more than 10000 users in our sample. The CCDF of the number of followers is also shown in Figure 5.11 and there is a visible glitch at 10^5, as seen in the original data in [Kwak et al., 2010]. In our sample, there are around 4000 users having more than 10^4 followers and 66 users beyond 1 million followers. By manual verification, they are found to be famous personalities, celebrities and news agencies.

Next we analyze the distribution of the number of tweets tweeted by users in

[2] http://tidal-news.org/data/icdcs14/tweetrates.tgz

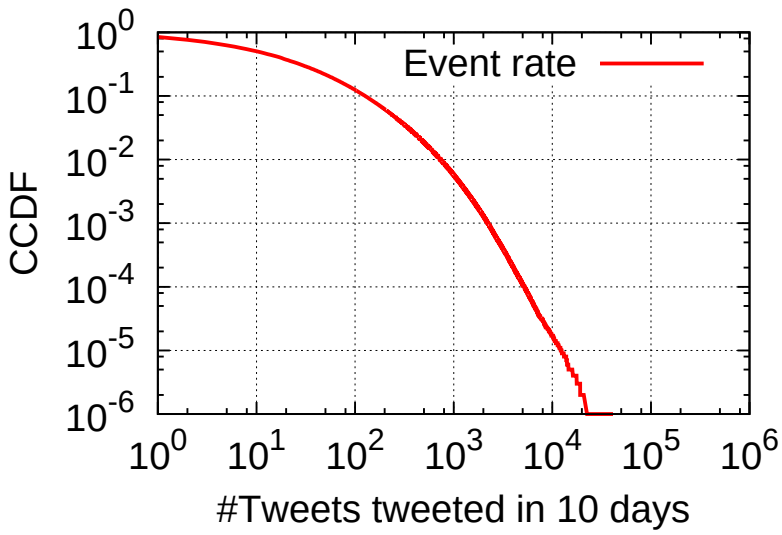

Figure 5.12: CCDF of event rate from 10 day traces

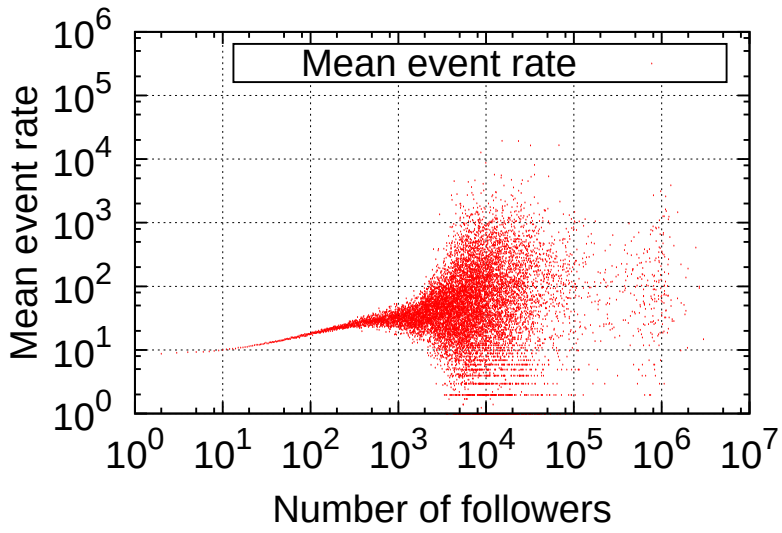

Figure 5.13: Correlation between #followers and event rate of a topic

our sample in a 10-day period Figure 5.12. Of the 8 million users who are active, around 4 million of them tweeted less than 10 tweets in 10 days. Around 46000 users tweeted more than 1000 tweets in 10 days, which is significantly high for human users. From random sample verification, these users are found to be news

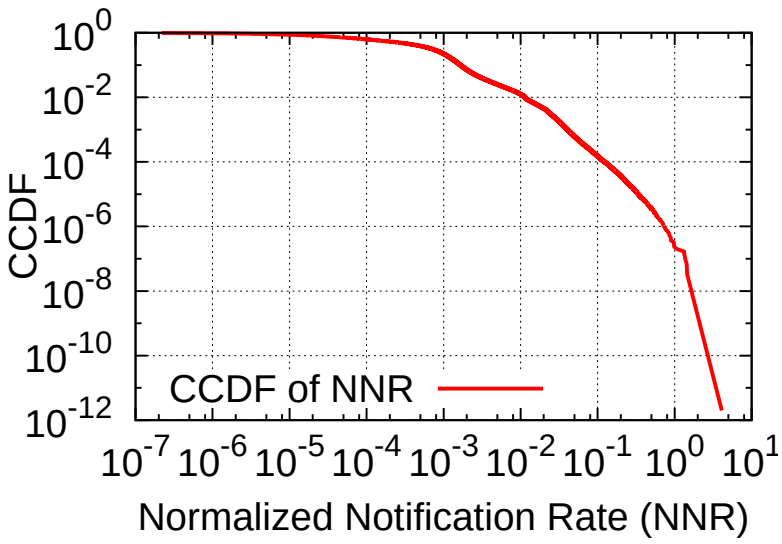

Figure 5.14: CCDF of Normalized Notification Rate (NNR)

agencies or tweet aggregation bots re-tweeting. There was one user tweeting more than 10^5 tweets and it was found to be a bot as well. Most celebrities produce relatively few tweets, despite their high number of followers. We explore this in detail in Figure 5.13. For each unique number of followers on the x-axis we show the corresponding mean tweet rate (event rate) on the y-axis. The mean event rate grows linearly with the number of followers until 10^5 followers. Finally, the smaller cloud corresponding to a number of followers between 10^5 and $2 \cdot 10^6$ has a relatively lower tweet rate than expected from the linear behavior. As mentioned earlier, this is because celebrities and popular news agencies tend to have more followers yet produce relatively few tweets.

Since our satisfaction metric τ is directly related to the number of events received by subscribers, it is worth studying the distribution of the number of tweets received by each user. For this purpose we use the Normalized Notification Rate (NNR_v) of a subscriber v defined in Section 5.1:

$$\text{NNR}_v = \frac{\sum_{t \in T_v} ev_t}{\sum_{t \in T} ev_t} \cdot 100$$

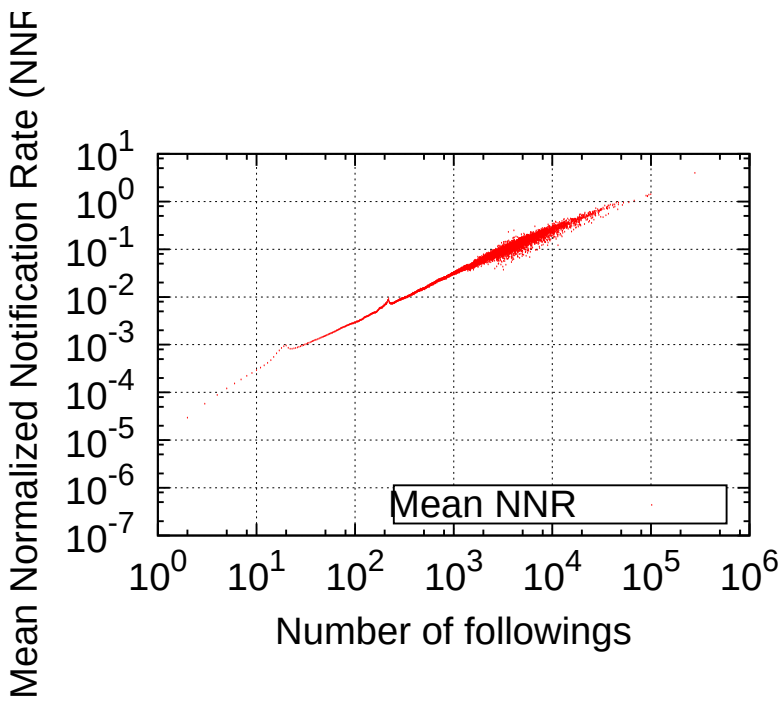

Figure 5.15: Correlation between #followings and Normalized Notification Rate (NNR)

In Figure 5.14 we show the CCDF of NNR_v. In our sample there are about 455 million tweets recorded, and around 3 million users receive more than 7000 tweets, and there is one user receiving 4% of all the tweets, i.e. 18 million tweets. Finally, we consider the correlation between the number of followings a user has and the corresponding mean NNR_v in Figure 5.15. In order to smooth the curve in this plot, we represent the data points for the subscribers having the same subscription sizes with a single point representing their mean NNR_v value. It is clear that NNR_v grows linearly with the number of subscriptions. However, there are noticeable glitches at 20 and 2000 followings, due to the same reason as the one for the glitches in the #Followings CCDF.

5.3 Summary

In this chapter, we did an extensive analysis of two real workloads from two social interaction systems: Spotify and Twitter. Both workloads were analyzed using real traces collected from the actual deployed systems.

For the Spotify workload, we had access to the production system and we characterized the system workload using the production traffic. Such characterization helps model pub/sub workloads for research. We also analyzed the pub/sub traffic at Spotify to derive trends and patterns.

We collected and analyzed Twitter traffic for 10 days crawled from the public APIs of Twitter. We observe interesting trends in publication event rate and normalized notification rate per subscriber.

Chapter 6

Publish/Subscribe to Maximize the Satisfaction Metrics of Users in Social Interaction

Traditionally, pub/sub implementations are either centralized or based on a federated organization of cooperatively managed servers, an increasingly higher number of pub/sub applications are being deployed in P2P environments [Triantafillou and Aekaterinidis, 2009]. In particular, the pub/sub service at Spotify described in Section 4.2.1 is suitable for a peer-assisted implementation, in line with the reported peer-assisted implementation of other Spotify services such as music streaming [Kreitz and Niemela, 2010]. In a peer-assisted implementation, a limited number of servers provide a guaranteed high-quality service to a subset of pub/sub subscribers while the rest of subscribers receive notifications through peers, thereby getting a best-effort service that works convincingly well in practice. The part of the workload assigned to a server is dictated by maximizing server utilization as well as the overall quality of service given to the subscribers.

In this chapter, to the best of our knowledge, we provide the first formal treatment of this subject. Specifically, we introduce a measure of subscriber satisfaction that lends itself to a large class of pub/sub notification services

where (a) publication event message delivery is best-effort: reliable delivery is desirable but it is not mandatory to deliver all notifications, and (b) every notification is intended to be read by a human user, so having a cumulative delivery rate to a particular subscriber above a certain threshold might not bring significant benefit to the user experience. For example, many applications where notifications are generated due to social interaction fall into this class of pub/sub services: following the tweets of selected users in Twitter, monitoring updates to the profiles of a user's friends in Facebook, or receiving instant notifications related to favorite artists and albums in Spotify. According to our satisfaction metric, we consider a subscriber satisfied in such applications if and only if the user receives all notifications of interest at a configurable minimum threshold delivery rate. We also provide a fractional satisfaction metric: If a subscriber receives fewer notifications than desired, the satisfaction of the subscriber is defined as a fraction of the actual and desired number of notifications.

Then, we introduce a principal optimization problem: given a server with a limited capacity, and a workload consisting of (a) a set of topics each with its own publication event rate, and (b) a set of subscribers with their interests; the goal is to *maximize* the number of subscribers with their cumulative delivery rate of publications to match a certain threshold (satisfaction metric), while respecting the *budget* constraint imposed by the limited resources of the backend servers. We define two distinct flavors of the problem: a **"Budgeted Maximum Multiset Multicover"** (*B3M*) and **"Fractional Budgeted Maximum Multiset Multicover"** (*F-B3M*) using the binary and fractional satisfaction metrics, respectively. We prove that both flavors are NP-Hard. We reduce *B3M* from the Densest-k-Subgraph (*DkS*) problem [Feige et al., 1997], a new way to reduce max cover problems. We also show that, while *B3M* does not admit a Polynomial-Time Approximation Scheme (PTAS) unless NP has randomized algorithms that run in sub-exponential time, *F-B3M* has a polynomial-time approximation algorithm with a guaranteed constant ratio of $\frac{1}{2}\left(1-\frac{1}{e}\right)$. Furthermore, we derive an upper bound for the optimal solution of each problem.

We evaluated the proposed heuristics for *B3M* and *F-B3M* using a large-scale

real data set from the pub/sub system of Spotify. We show that the heuristics provide an approximation of at least 0.7 for both problems, for the given dataset, using the derived upper bound on the optimal solution as the baseline. Finally, we propose various optimizations to make the heuristics more efficient. We show that the heuristics run in less than 30 seconds for workloads with over a million topics, and in less than one second in most realistic scenarios.

6.1 Motivating Application Scenario and Proposed Pub/Sub Architecture

As discussed in Section 1.3.3, utilizing the pub/sub infrastructure to maximize the number of subscribers receiving the notifications at a minimum threshold can be useful. In addition, the workload that cannot be handled by the dedicated infrastructure can be offloaded to a lower cost external system (such as a peer-to-peer network). In this regard, the problem of selecting a subset of the workload in such a way as to maximize subscriber satisfaction while respecting the backend capacity needs to be solved. As shown in Sections 6.2.1 and 6.2.2, this is a challenging optimization problem.

In this chapter, we propose a methodology to select a subset of the pub/sub workload such that this subset is within the capacity of a backend service with limited resources, while user satisfaction is maximized. This approach can help system managers to deal with the trade-off between deploying additional hardware and satisfying more users. It can also be used as a mechanism to drop or divert part of the pub/sub workload to an external lower cost system with lower quality of service, such as a pool of lower-reliability servers, or a set of computers belonging to end users (peers) forming a peer-to-peer network.

To facilitate the offloading of the workload we propose an idea of a service called *Offloading Decision Service* (ODS). In order to perform its work, the ODS divides the total pub/sub load on a per topic basis and then decides for each topic whether the topic can be managed by the backend service without exceeding the capacity. In this context, managing a topic means taking care of delivering the

corresponding topic events for all subscribers of that topic.

The rationale for this design decision is that we believe that organizing the pub/sub load at this granularity level greatly simplifies system design compared to an approach based on dealing with each (topic, subscriber) subscription pair individually. While offloading at (topic, subscriber) granularity may be beneficial, it poses additional overhead to the pub/sub system and the ODS, making the offloading more complicated and expensive.

We now show the benefits of the ODS in the context of practical pub/sub systems designed for social interaction.

6.1.1 A Peer-Assisted Approach to Social Interaction Among Spotify Users

Spotify uses a pub/sub system to facilitate social interaction among its users. A Spotify user can follow friends (from Facebook or native to Spotify), artists and playlists. The pub/sub system delivers the friend feed, artist updates, and playlist updates to the appropriate Spotify users. The Spotify pub/sub system is implemented as a backend service running in Spotify's datacenters. Details about Spotify's pub/sub system have been presented in Chapter 4. With the ever-growing user base of Spotify it is crucial for its pub/sub to scale accordingly. Typically, such services are scaled horizontally by deploying new hardware. In this chapter, we provide a tool that can help system managers to estimate the amount of user satisfaction that can be achieved with existing resources, and estimate how it can be improved with additional hardware. We also show how the existing Spotify pub/sub architecture can be extended to divert part of the pub/sub workload to a P2P network by solving the proposed optimization problems, in line with the existing peer-assisted streaming solution already used by Spotify.

In a peer-assisted architecture, part of the load that is normally managed by a server in a classical client/server architecture, is managed by clients themselves, which act as servers towards other clients, and are referred to as peers. This approach has the advantage that it can reduce implementation costs, and can

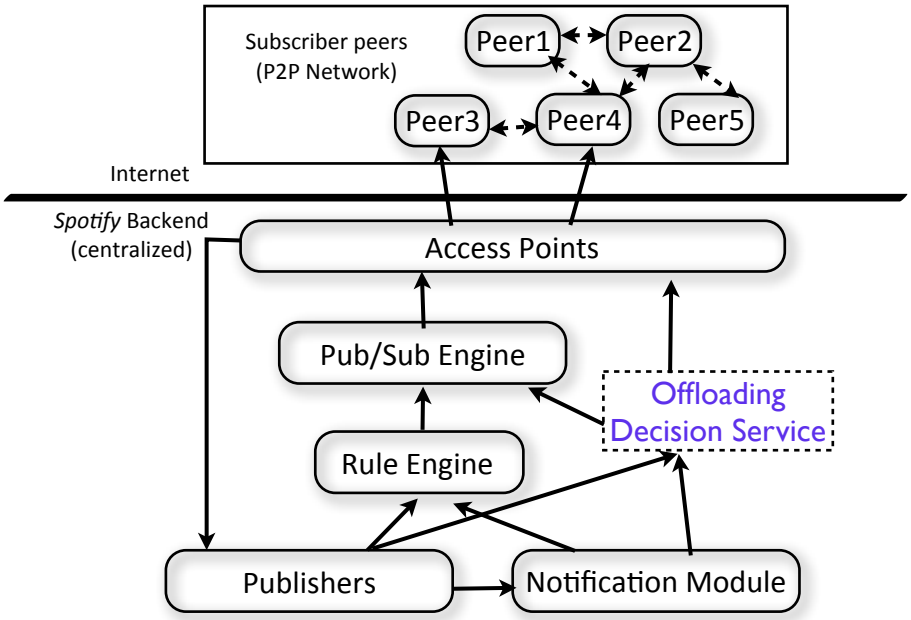

Figure 6.1: Proposed peer-assisted architecture for Spotify pub/sub

potentially scale easily with respect to the number of users, since peers bring with them an amount of resources that is proportional to the load the system has to handle.

Figure 6.1 shows our proposed peer-assisted pub/sub architecture. This architecture is designed to retain all the existing modules explained in Section 4.2. The only new module is the Offloading Decision Service (ODS) introduced earlier.

As shown in Figure 6.1, the ODS has access to Publishers and the Notification Module of the pub/sub architecture of Spotify to collect statistics about publication event rates and topic popularity. Depending on the satisfaction metric used, ODS will solve *B3M* or *F-B3M*, using collected statistics and the heuristics presented in Sections 6.3 and 6.4. The ODS then instructs the pub/sub engine to consider the list of topics it has found to maximize the subscriber satisfaction for real-time delivery of publications using its pool of brokers, while the remaining topics are offloaded to the P2P network.

The ODS constantly monitors changes to the publication event rates as well as subscriptions and unsubscriptions and uses these updated statistics to periodically recompute the solutions for *B3M* or *F-B3M* to maximize the subscriber satisfaction. Therefore, an additional requirement for the ODS is that it should employ lightweight algorithms that can be executed relatively quickly. In this regard, we propose efficient algorithms to solve *B3M* or *F-B3M* in Sections 6.3 and 6.4 and validate them in Section 6.5 to show that they can be executed in real-time for real traces from Spotify.

6.1.2 Cloud-Based Peer-Assisted Microblogging Service

In [Xu et al., 2011], Cuckoo, a new Twitter-like microblogging system that offloads the workload from the cloud to a P2P network is proposed. However, the offloading technique does not consider optimality and hence it may result in underutilization of the cloud resources. In addition, Cuckoo could benefit from our definition of satisfaction metrics to deal with overwhelming event rates of the topics related to news media. The Cuckoo design relies on offloading the topics with low publication rate and few subscribers to the P2P network. While this is proven to reduce the load on the cloud, we believe more can be achieved with the same cloud resources by using a more sophisticated strategy to select what to offload. In Sections 6.2.1 and 6.2.2 we formalize this problem and provide approximation algorithms that could be applied in Cuckoo.

It is worth noting that application of the ODS is not limited to the two scenarios described above. It is not hard to see the applicability of the ODS in many other pub/sub system with limited resources.

6.2 Problem Definitions

The two variations of satisfaction metrics mentioned earlier in this chapter prompt problems that are similar in nature but very different in hardness, as we show in Sections 6.3 and 6.4. In the first metric, we are interested in maximizing the number of subscribers receiving at least τ (satisfaction threshold) events related to them from the backend service. A subscriber is considered satisfied if and only

if at least τ relevant events are received. This definition of user satisfaction is suitable for applications with events that are relatively infrequent but important for the user. Spotify updates about favorite albums and artists fall in this category. In this regard, we define a problem coined *Budgeted Maximum Multiset Multicover* (*B3M*) in this section. In Section 6.3 we analyze the hardness of *B3M* and propose a feasible heuristic.

In the second metric we quantify the amount of benefit towards the satisfaction of a subscriber with a fraction of cumulative events delivered to a subscriber relative to the given satisfaction threshold of τ. The goal is to maximize the sum of fractional benefits of individual subscribers of the topics set to be served by the backend servers. This definition is appropriate for applications where events are frequent but of relatively low importance. An example would be Spotify's updates about the activities of the friends of each given user. In this regard we define the *Fractional Budgeted Maximum Multiset Multicover* (*F-B3M*) problem. In Section 6.4 we analyze the hardness of *F-B3M* and propose a feasible heuristic that also gives a guarantee on the quality of the output.

In both flavors of the problem, we want to ensure that the computational and communication costs to serve the events needed to maximize the number of satisfied subscribers does not exceed a given limit on the capacity of the resources at the backend service.

Before we define the problem more formally, we introduce the following notations:

T **:** A collection of l *topics* $\{t_1, t_2, ..., t_l\}$ in the system.

V **:** A collection of n *subscribers* $\{v_1, v_2, ..., v_n\}$ participating in the pub/sub system. A subscriber can subscribe to one or more topics from T. Subscribers in a typical pub/sub system are generally end-user applications (e.g. Spotify client software).

T_v **:** The *interest* of subscriber v, that is, the set of topics subscribed by v.

Int **:** The collection of interests $\{T_{v_1}, T_{v_2}, ..., T_{v_n}\}$ for all subscribers in V.

ev_t **:** *Event rate* of the publications generated for a topic t, that is, mean of events published to topic t during a given period (e.g., per minute or per hour). Without loss of generality, we assume that $ev_t > 0$. When we say 'event' in the rest of this thesis we mean a publication event message generated by the backend service for a topic intended for all subscribers of the topic.

τ **:** A system parameter that represents the *satisfaction threshold* for a subscriber. It is defined as a constant specifying the number of events to be delivered to a subscriber by the backend service in order for the subscriber to be considered satisfied. The period over which the events are to be delivered is the same as the time unit of ev_t.

τ_v **:** Subscriber-specific satisfaction threshold. In practice, the total event rate of the topics subscribed to by a subscriber is sometimes less than τ. In such cases we need to serve all the events the subscriber is interested in to meet the satisfaction threshold. It is mathematically expressed as follows: $\tau_v = min(\tau, \sum_{t \in T_v} ev_t)$.

V_t **:** $V_t \subseteq V$ is a non-empty set of subscribers to topic t. Given *Int*, V_t can be derived trivially.

$cost(t)$ **:** Represents the non-zero cost of serving a topic t by the backend service. We say that the cost of a topic is *normalized* if it costs 1 per event sent by the server to each subscriber of the topic and hence, *normalized* cost is defined as $cost(t) = ev_t \cdot |V_t|$.

$\mathcal{C}$ **:** Capacity of the backend service. A constant to quantitatively represent the amount of resources available to the backend service. $\mathcal{C}$ has same unit as *cost*.

$\mathcal{S}$ **:** Solution ($\mathcal{S} \subseteq T$). It is a set of topics that can be served by the backend service with a cost that does not exceed a given resource constraint expressed by the constant $\mathcal{C}$.

$\sigma(\mathcal{S})$ **:** Represents the sum of the satisfaction for all subscribers, given a potential solution $\mathcal{S}$. We want to maximize this function.

6.2.1 The Problem of Budgeted Maximum Multiset Multicover (*B3M*):

Given an instance of T, V and their interests *Int*, the goal of the $B3M(T, V, ev, cost, Int, \tau, \mathcal{C})$ problem is to find $\mathcal{S} \subseteq T$ so as to maximize the objective function defined below:

$$\text{Maximize } \sigma(\mathcal{S}) = \sum_{v \in V} f(v), \text{ subject to} \sum_{t \in \mathcal{S}} cost(t) \leq \mathcal{C} \tag{6.1}$$

$f(v)$ is a function that indicates if subscriber v is receiving a number of events that meets the satisfaction threshold:

$$f(v) = \begin{cases} 1 & \text{if } \sum_{\{t \in \mathcal{S} \cap T_v\}} ev_t \geq \tau_v \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

The first condition in the Equation (6.2) is the case when a subscriber v is receiving publication events at a rate not lower than τ_v. In order for v to contribute to the objective function $f(v)$, the solution $\mathcal{S}$ must include enough topics subscribed by v with a total event rate of at least τ_v.

6.2.2 The Problem of Fractional Budgeted Maximum Multiset Multicover (*F-B3M*):

We now define a relaxed version of the *B3M* problem in which we quantify the satisfaction relative to the number of events covered for a subscriber v out of τ_v events. Given an instance of T, V and their interests *Int*, the goal of the *F-B3M*$(T, V, ev, cost, Int, \tau, \mathcal{C})$ problem is to find $\mathcal{S} \subseteq T$ so as to maximize the sum of the fractions for all the subscribers.

$$\text{Maximize } \sigma(\mathcal{S}) = \sum_{v \in V} g(v), \text{ subject to} \sum_{t \in \mathcal{S}} cost(t) \leq \mathcal{C} \tag{6.3}$$

$g(v)$ is the fraction of events subscriber v receives, and it is defined as:

$$g(v) = \begin{cases} 1 & \text{if } \sum_{\{t \in \mathcal{S} \cap T_v\}} ev_t \geq \tau_v \\ \frac{\sum_{\{t \in \mathcal{S} \cap T_v\}} ev_t}{\tau_v} & \text{Otherwise} \end{cases} \tag{6.4}$$

The difference between *B3M* and *F-B3M* lies in the definition of the satisfaction metrics in Equation (6.2) and Equation (6.4) respectively. In Equation (6.2) the satisfaction is defined in a binary fashion i.e. the satisfaction is 0 when less than τ_v events are received by the subscriber and 1 otherwise. On the other hand in Equation (6.4) a fraction of events received up to τ_v is considered instead of a binary 1 or 0. This subtle difference makes the two problems fundamentally different in terms of difficulty of solving. We explore this in detail in Sections 6.3 and 6.4.

6.3 Hardness of *B3M* and its Solution Approach

In this section we prove that *B3M* is NP-Hard and we also show that *B3M* has no Polynomial-Time Approximation Scheme (PTAS). We further propose an algorithm to give an upper bound on *B3M* instances. We use this bound to evaluate a greedy heuristic we propose in Section 6.5.2.

6.3.1 Hardness of *B3M* Problem

To establish the hardness of *B3M* we prove that the well-known hard problem of Densest-k-Subgraph (*DkS*) can be reduced to a special case of *B3M*. We now define the *DkS* problem and an auxiliary unit-cost version of *B3M*.

Definition 6.1 (Densest-k-Subgraph)**.** *Given an undirected graph $G(U, E)$ the Densest-k-Subgraph ($\mathrm{DkS}(U, E, k)$) problem on G is the problem of finding a subset $U' \in U$ of vertices of size $|U'| = k$ with the maximum induced average degree. The average degree of the optimal subgraph is $2|E(U')|/k$. Here $|E(U')|$ denotes the number of edges in the subgraph induced by U'.*

The *DkS* problem can be proven to be NP-Hard by reduction from the Max-Clique problem [Feige et al., 2001]. In [Feige et al., 1997] it has been shown that

DkS is also NP-Hard even when restricted to a maximum degree of 3. The best known approximation algorithm achieves a ratio of $\mathrm{O}\big(n^{1/4+\epsilon}\big)$ and runs in $2^{n^{O(1/\epsilon)}}$ time, for any $\epsilon > 0$ [Bhaskara et al., 2010]. On the other hand, it is known that *DkS* does not admit a PTAS [Khot, 2006].

Definition 6.2 (UC-*B3M*)**.** *We define an auxiliary problem coined Unit-Cost-*B3M*(*UC-B3M*) which is a restricted version of* B3M*. We define* UC-B3M *to be an instance of* B3M *with unit-cost for all the topics* $\forall t \in T : cost(t) = 1$ *and unit event rate* $ev_t = 1$*, each subscriber subscribes to exactly two topics* $\forall v \in V : |T_v| = 2$*, no two subscribers subscribe to same set of topics* $\forall v_1 \neq v_2 : T_{v_1} \neq T_{v_2}$ *and the satisfaction threshold* $\tau_v = 2$*.*

Lemma 6.3. UC-B3M *is NP-Hard.*

Proof. Given an instance of $DkS(U, E, k)$ we construct an instance of *UC-B3M*$(T, V, ev, cost, Int, \tau, \mathcal{C})$ in the following way: we take T with topics that one-to-one correspond to the vertices in the set U. We take V to one-to-one correspond to the edges in the set E. We build *Int* from the edges incident on the vertices. For example, V_t corresponds to the edges incident on the corresponding vertex in U. We set $\mathcal{C} = k$. We now prove that there is an induced subgraph of $A(U', E')$ with average degree δ and exactly k vertices if and only if there is a solution $\mathcal{S}$ to *UC-B3M* with value at least $|E(U')|$ (i.e., the total number of edges in the induced subgraph).

To see this, we observe that a subscriber in our *UC-B3M* instance only contributes to the objective function if both of her topics are included in $\mathcal{S}$. This precisely corresponds to the condition if and only if that exact edge with the vertices corresponding to those two topics is in the induced subgraph of the *DkS* instance. We can, without loss of generality, assume that $\mathcal{S}$ contains precisely k topics as the cost of each topic is 1 and the objective function is non-decreasing in the number of selected topics.

As we know that *DkS* is NP-Hard [Feige et al., 2001], it follows that *UC-B3M* is NP-Hard too. □

Theorem 6.4. B3M *is NP-Hard.*

Algorithm 1: Heuristic value of topic t given partial solution $\mathcal{S}'$

1 **GetHeuristicB3M**$(t, ev, cost(t), Int, \mathcal{S}', \tau)$
  Input: $t, ev, cost(t), Int, \mathcal{S}', \tau$
  Data: $h \leftarrow 0$: Heuristic value
  $rem_v \leftarrow 0$: Events remaining to make user v happy
2 **foreach** $\{v \in V_t\}$ **do**
3 $rem_v \leftarrow \tau_v - \sum_{\{t' \in \mathcal{S}' \cap T_v\}} ev_{t'}$
4 **if** $rem_v > 0$ **then**
5 $h \leftarrow h + \min\left(1, \frac{ev_t}{rem_v}\right)$
6 **return** $\frac{h}{cost(t)}$

Proof. *UC-B3M* is a special case of *B3M*. From Lemma 6.3 we know that *UC-B3M* is NP-Hard and hence *B3M* is NP-Hard too. □

Corollary 6.5. *Assuming* NP $\not\subseteq \cap_{\epsilon>0}$BPTIME$(2^{n^\epsilon})$*, there is no Polynomial-Time Approximation Scheme (PTAS) for* B3M.

Proof. The statement follows directly for *UC-B3M* from the reduction given in Lemma 6.3 together with a result by Khot [Khot, 2006] saying that unless NP has randomized algorithms that run in sub-exponential time (more formally: NP $\subseteq \cap_{\epsilon>0}$BPTIME$(2^{n^\epsilon})$) there is no PTAS for *DkS*. As *UC-B3M* is a special case of *B3M*, the statement also holds for *B3M*. □

6.3.2 Greedy Heuristic for *B3M*

In the greedy algorithm to solve *B3M*, in each iteration of the algorithm, a topic t is chosen so as to maximize the ratio between its benefit and its cost. The benefit of a topic is quantified by its total contribution towards the objective function relative to the already chosen topics $\mathcal{S}'$. This is done for each subscriber of a topic in a for loop (between Lines 2 and 5 of Algorithm 1). We define the contribution of a topic t by considering the following scenarios: Adding t to the solution $\mathcal{S}'$ (a) guarantees to deliver τ_v events to its subscriber v (b) contributes partially to the target τ_v events for its subscriber v. In the first case, the contribution is of value 1. In the second case, the contribution is the ratio between ev_t and

Algorithm 2: Greedy solution for ***B3M***

```
1  GreedyB3M(T, V, ev, cost, Int, τ, C)
   Input: T, V, ev, cost, Int, τ, C
   Data: A : Array of size l
   Result: S' ← ∅ : Output set of topics
2  foreach t ∈ T do
3      A[t] ← GetHeuristicB3M(t, ev, cost(t), Int, S', τ)
4  while T ≠ ∅ do
5      t ← argmax_{t'∈T} A[t']
6      if A[t] = 0 then
7          break
8      if cost(t) + Σ_{t'∈S'} cost(t') ≤ C then
9          S' ← S' ∪ {t}
10         foreach {t' : V_t ∩ V_t' ≠ ∅ ∧ t' ∉ S'} do
11             A[t'] ← GetHeuristicB3M(t', ev, cost(t'), Int, S', τ)
12     T ← T \ {t}
13 return S'
```

the remaining events needed to reach the target τ_v (computed in Line 3). The intuition behind this choice is to give higher priority to a topic that satisfies a subscriber and hence, directly contributes to the objective function. On the other hand, a topic contributing partially to the satisfaction of its subscriber is given relatively lower priority. This step is repeated for each subscriber of the topic t and the contribution is accumulated as a sum (Line 5). Finally, in Line 6 the total contribution is divided by the topic's cost to return the benefit-cost ratio.

The pseudocode of the greedy algorithm to solve *B3M* is sketched in Algorithm 2 and the greedy strategy is to choose a topic that maximizes the objective function. In Lines 2 and 3 an array containing the benefit-cost ratio of the individual topics is initialized using Algorithm 1. In practice, this array can be a max-heap structure optimized for obtaining elements with maximum value. A topic that maximizes the benefit-cost ratio in each iteration is selected in Line 5. The topic is added to the solution if its addition keeps the cost of the solution within the budget. Otherwise the topic is ignored. If the topic is added to the solution, the benefit-cost ratio of all the topics not selected so far are

updated based on the current solution set $\mathcal{S}'$ (Lines 10 and 11). $V_t \cap V_{t'}$ is the set of subscribers common to subscribers of t and subscribers of t'. The algorithm terminates when it has considered all the available topics, or when all subscribers have been covered in which case the benefit-cost ratio of all the topics would be 0 (Line 7).

Theorem 6.6. *The run time complexity of Algorithm 2 is* $\mathrm{O}\big(|T|^2(|V| + \log|T|)\big)$.

Proof. The data structure A can be any max-heap structure supporting insertion, update, and extracting the maximum element in time $\mathrm{O}\big(\log n\big)$, e.g., a binary heap. The initialization of the array to store the heuristic values per topic done in Line 3 of the Algorithm 2 and Algorithm 7 has complexity of $\mathrm{O}\big(|T||V|\log|T|\big)$. Once a topic is selected a while loop (Lines 9 to 12) is exectued to update the topics in the top of the heap until there is no more change. This loop runs $|T|$ times in the worst-case. Within the loop, re-evaluating the heuristics has complexity of $|V|$ and updating A has takes time $\mathrm{O}\big(\log|T|\big)$. Hence the run time complexity of the Algorithm 2 is

$$\mathrm{O}\big(|V||T|\log|T| + |T|^2\,(|V| + \log|T|)\big) \approx \mathrm{O}\big(|T|^2(|V| + \log|T|)\big).$$

□

Theorem 6.6 gives the worst-case run time complexity, the cost being dominated by updating the cost for all topics in Lines 10 and 11 of Algorithm 2 when a topic is added to the solution. We remark that in practice, the code runs significantly faster than this bound would imply. One of the reasons being that the number of updates is bounded by $max_{t \neq t'}|V_t \cap V_{t'}|$, which is usually significantly lower than $|T|$.

We now turn to the subject of computing an upper bound on the optimal solution. For this analysis, we only consider the case when the cost function is normalized, i.e., $cost(t) = ev_t \cdot |V_t|$.

Theorem 6.7. *Given an instance* B3M*$(T, V, ev, cost, Int, \tau, \mathcal{C})$ where the costs are*

normalized, for any solution $\mathcal{S}$ it holds that:

$$\sigma(\mathcal{S}) \leq \max\left(\left|V'\right| : \sum_{v \in V'} \max\left(\tau_v, \min_{t \in T_v} ev_t\right) \leq \mathcal{C}\right),$$

where $V' \subseteq V$.

Proof. With normalized costs, one can see that the amortized cost to cover each subscriber v is at least τ_v. The cost is also bounded by the lowest event rate of any event in which the subscriber is interested. Detailed proof is given below.

Given a data set and a capcity constraint, we can derive an upper bound on the number of subscribers that can be maximized. Intuitively, we can spend the available capacity minimally to buy satisfied subscribers. However, since we consider the case of normalized costs, in order to obtain a tighter bound we only pay amortized cost of a topic for each of its subscribers. We elaborate this idea below:

We first show that the theorem holds for an instance where $\forall_{t \in T} |V_t| = 1$, and then show that it generalizes to the full setting with normalized costs.

When each topic has only a single subscriber, consider the capacity that must be spent to add a user to the solution set. A subscriber v can be satisfied when topics with total event rate of τ_v are selected in the solution. Hence, the minimum capacity that must be spent to satisfy a subscriber is τ_v. To tighten this bound slightly, we also observe that if $\forall_{t \in T_v} ev_t \geq \tau_v$, then the semantics of the *B3M* definition dictates that a topic must be completely paid for or not at all. Hence, the capacity that must be spent in such a scenario is $\min_{t \in T_v} ev_t$. Note that the topic's costs here are normalized, i.e., $cost(t) = ev_t * |V_t|$. Since $|V_t| = 1$, we derive the clause $\max(\tau_v, \min_{t \in T_v} ev_t)$ as a cost to satisfy a single subscriber. Clearly, the solution set must have sufficient capacity to add all users, so summing up these bounds, we get the theorem as stated.

Considering the general setting of the problem where topics can be subscribed to by multiple users, we see that our bound still holds. As costs are normalized, we can easily amortize the cost of adding a subscriber to the solution. As the cost of a topic is $cost(t) = ev_t * |V_t|$, when we select a topic we add an amortized cost

Algorithm 3: Upper bound for ***B3M*** with normalized topic costs

```
1 GetUpperBound(V, T, ev, Int, C, τ)
  Input: V, T, ev, Int, C, τ
  Data: C : Array of size n
  csubs ← ∅ : Set of subscribers covered
2 foreach {v ∈ V} do
3     C[v] ← max(τ_v, min_{t∈T_v} ev_t)
4 while V ≠ ∅ do
5     v ← argmin_{{v'∈V}} C[v']
6     if C[v] + Σ_{v'∈csubs} C[v'] ≤ C then
7         csubs ← csubs ∪ {v}
8         V ← V \ {v}
9 return |csubs|
```

of ev_t to each of the subscribers of the topic. The bound we derived on the setting where each topic has a single subscriber also applies to the amortized costs, and thus the theorem follows. □

Theorem 6.7 presents a way to compute an upper bound on the optimal solution. Since Algorithm 2 gives an unbounded approximation ratio, we make use of Theorem 6.7 to evaluate how well our proposed heuristic performs on real-world inputs (see Section 6.5.2). This theorem can be readily turned into an algorithm as shown in Algorithm 3. In Lines 2 and 3 the minimum cost to consider a subscriber satisfied is initialized in an array. Then, in each iteration the subscriber with the least cost is selected until there is no more budget left to cover more subscribers (between Lines 4 and 8). Finally, the number of selected subscribers is returned as the upper bound for the optimal solution (Line 9).

6.4 Hardness of *F-B3M* and its Solution Approach

In this section we analyze the hardness of *F-B3M*. Comparing to the results we obtained for *B3M*, the direct reduction we did from Densest-k-Subgraph no longer works as in that case it is imperative that we are not "paid" for a partially satisfied subscriber. This also means that the approximation-resistance results obtained

for *B3M* do not translate. For *F-B3M*, we are instead able to give a greedy approximation algorithm with an approximation ratio of $\frac{1}{2}\left(1-\frac{1}{e}\right)$. *F-B3M* is still NP-Hard, which we first prove by a reduction from the (unweighted) *Maximum Coverage problem* [Hochbaum, 1997].

Theorem 6.8. F-B3M *problem is NP-Hard.*

We prove that *F-B3M* is NP-Hard by polynomially reducing it from the problem of *Maximum Coverage* problem. We first define the Maximum Coverage problem.

Definition 6.9 (Maximum Coverage)**.** *In the (unweighted)* Maximum Coverage *problem, input consists of a collection of sets* $S = \{s_1, s_2, \ldots, s_n\}$ *and a parameter* k*. The goal is to find a subset* $S' \subseteq S$ *maximizing* $|\bigcup_{s \in S'} s|$ *subject to* $|S'| \leq k$*.*

Proof. Given an instance of Maximum Coverage(S, k) we construct an instance of *F-B3M*$(T, V, ev, cost, Int, \tau, \mathcal{C})$ in the following way: we take T with topics that one-to-one correspond to the sets in the collection S and let $cost(t) = 1$. We take V that one-to-one correspond to the elements of $\bigcup_{s \in S}$ and construct *Int* from set membership relationship of sets in S. We further let $ev_t = 1$, set $\tau = 1$, and let $\mathcal{C} = k$.

From this construction it is easy to see that there is a solution of size d of the Maximum Coverage instance if and only if there is a solution of value d of the *F-B3M* instance. As Maximum Coverage is NP-Hard, this concludes the proof. □

6.4.1 Greedy Heuristic

Theorem 6.10. *The objective function in the* F-B3M *problem from Expression* (6.3) *is a* ***submodular*** *function.*

Before we prove Theorem 6.10, we define the submodularity propery.

Definition 6.11 (Submodularity)**.** *A function* σ *is said to be submodular for any set* $A \subseteq B$ *if the following holds:*

$$\sigma(A \cup x) - \sigma(A) \geq \sigma(B \cup x) - \sigma(B)$$

for any element $x \notin B$.

Proof. Intuitively, the objective function for *F-B3M* is submodular because the incremental gain from adding a new topic is fractional i.e, reaching a threshold of τ to have incremental gain is not a requirement. However, a larger set is more likely to have covered more subscribers and higher number of times hence the gain is incremental. In addition to that adding a topic with subscribers already covered τ to a larger set of topics gives no incremental gain in the objective function. On the other hand adding it to a smaller set of topics would give larger incremental gain. Let us now capture the intuition mathematically. Assume that we have two solution sets $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\mathcal{S}_2 \subseteq \mathcal{S}_1$. Adding a topic $t \notin \mathcal{S}_1$ to these sets always has non-negative incremental gain in their respective objective functions. However, the amount of incremental gain depends on the following scenarios:

1. The subscribers V_t of topic t are already covered τ times in both $\mathcal{S}_1$ and $\mathcal{S}_2$. Hence, adding t does not lead to any incremental gain for both sets. Note that this case can be extended to both sets already covering equal number of times, and the incremental gain will be the same for both.

2. V_t are covered in $\mathcal{S}_1$ x times and they are covered y times in $\mathcal{S}_2$ such that $x \geq y$ (again, note that the other way round is not possible since $\mathcal{S}_2 \subseteq \mathcal{S}_1$). The following sub-cases are possible:

   (a) If $x + ev_t \geq \tau$ and $y + ev_t \geq \tau$ then, since we know that $x \geq y$, $\mathcal{S}_1$ will have lower gain because $\sum_{v \in V_t} \frac{\tau_v - x}{\tau_v} \leq \sum_{v \in V_t} \frac{\tau_v - y}{\tau_v}$

   (b) If $x + ev_t \geq \tau$ and $y + ev_t < \tau$ then, the incremental gain for $\mathcal{S}_2$ is higher because the incremental gain for $\mathcal{S}_1$ is $\sum_{v \in V_t} \frac{\tau_v - x}{\tau_v} \leq \sum_{v \in V_t} \frac{ev_t}{\tau_v}$ since we know that $x + ev_t \geq \tau$.

   (c) Finally, if $x + ev_t < \tau$ and $y + ev_t < \tau$ then, both $\mathcal{S}_1$ and $\mathcal{S}_2$ have same the incremental gain.

3. V_t are covered τ times in $\mathcal{S}_1$ but not in $\mathcal{S}_2$ (note that the other way round is not possible since $\mathcal{S}_2 \subseteq \mathcal{S}_1$). Hence, adding t to $\mathcal{S}_1$ results in no incremental gain while the objective function for $\mathcal{S}_2$ is incremented with exactly $\sum_{v \in V_t} \frac{\min(rem_v, ev_t)}{\tau_v}$, where, $rem_v = \tau_v - \sum_{\{t' \in \mathcal{S}' \cap T_v\}} ev_{t'}$.

All possible scenarios are covered using the above cases. It is easy to see that in all of the above scenarios the following always holds for any $t \notin \mathcal{S}_1$.

$$\sigma(\mathcal{S}_1 \cup t) - \sigma(\mathcal{S}_1) \leq \sigma(\mathcal{S}_2 \cup t) - \sigma(\mathcal{S}_2)$$

□

Algorithm 4: Heuristic value of topic t given partial solution $\mathcal{S}'$

```
1 GetHeuristicFB3M(t, ev, Int, S', τ)
  Input: t, ev, Int, S', τ
  Data: h ← 0 : Heuristic value
  rem_v : Events remaining to make user v happy
2 foreach {v ∈ V_t} do
3     rem_v ← τ_v − Σ_{t'∈S'∩T_v} ev_{t'}
4     if rem_v > 0 then
5         h ← h + min(rem_v, ev_t) / τ_v
6 return h
```

From Theorem 6.10 we infer that the *F-B3M* problem is essentially the budgeted maximization of a submodular function. The generalized greedy heuristic for maximization of submodular functions is known to guarantee a constant approximation factor as shown in [Fisher et al., 1978]. Unfortunately, greedily selecting topics with best benefit-cost ratio for a budgeted maximization of a submodular function no longer gives a constant approximation guarantee. Greedily choosing the topics similarly to the solution for *B3M* performs arbitrarily poorly.

To see why the simple greedy approach fails, consider an instance with two topics t_1 and t_2 with $\sigma(t_1) = 1$ and $cost(t_1) = 1$ and $\sigma(t_2) = x$ for some $x > 1$ and $cost(t_2) = x + 1$ and with $\mathcal{C} = x + 1$. The heuristic of benefit-cost ratio prefers t_1 over t_2. Having spent a budget of 1 the heuristic can no longer select t_2 and terminates with $\sigma(t_1) = 1$ while the optimal solution is choosing t_2 with the gain $\sigma(t_2) = x$ giving an approximation ratio of x.

Taking inspiration from [Khuller et al., 1999], we address this problem by running two instances of a greedy algorithm, each using a different heuristic. The

Algorithm 5: Appropriate simple greedy algorithm for *F-B3M*, given a type

```
1  GreedyFB3M(T, V, ev, cost, Int, τ, C, type)
   Input: T, V, ev, cost, Int, τ, C, type
   Data: A : Array of size l
   Result: S' ← ∅ : Output set of topics
2  foreach t ∈ T do
3      A[t] ← ComputeHeuristic(t, ev, cost(t), Int, S', τ, type)
4  while T ≠ ∅ do
5      t ← argmax_{x∈T} A[x]
6      T ← T \ {t}
7      if cost(t) + Σ_{t'∈S'} cost(t') ≤ C then
8          S' ← S' ∪ {t}
9          repeat
10             t' ← t
11             t ← argmax_{x∈T} A[x]
12             A[t] ← ComputeHeuristic(t, ev, cost(t), Int, S', τ, type)
13         until A[t'] = A[t]
14 return S'
```

Algorithm 6: Appropriate heuristic, given a *type*

```
1  ComputeHeuristic(t, ev, cost(t), Int, S', τ, type)
   Input: t, ev, cost(t), Int, S', τ, type
2  if type = G then
3      return GetHeuristicFB3M(t, ev, Int, S', τ)
4  else if type = R then
5      return GetHeuristicFB3M(t, ev, Int, S', τ)/cost(t)
```

Algorithm 7: Greedy algorithm for ***F-B3M***

1 **ModifiedGreedyFB3M**$(T, V, ev, cost, Int, \tau, \mathcal{C})$
  Input: $T, V, ev, cost, Int, \tau, \mathcal{C}$
2 $\mathcal{S}' \leftarrow$ **GreedyFB3M**$(T, V, ev, cost, Int, \tau, \mathcal{C}, \mathcal{G})$
3 $\mathcal{S}'' \leftarrow$ **GreedyFB3M**$(T, V, ev, cost, Int, \tau, \mathcal{C}, \mathcal{R})$
4 **if** $\sigma(\mathcal{S}') \geq \sigma(\mathcal{S}'')$ **then return** $\mathcal{S}'$
5 **else return** $\mathcal{S}''$

first algorithm, which we refer to as being of *type* $\mathcal{G}$, uses σ as shown in Algorithm 4. The second algorithm, of *type* $\mathcal{R}$, uses the benefit-cost ratio $(\sigma/cost(t))$. The final solution is the best of the two solutions provided by executing the algorithms of type $\mathcal{G}$ and $\mathcal{R}$, respectively. The pseudocode of the simple greedy algorithm is shown in Algorithm 5. Algorithm 7 is the pseudocode for the modified greedy algorithm to solve the *F-B3M* problem that executes the simple greedy algorithms of *type* $\mathcal{G}$ and $\mathcal{R}$ and selects the best solution.

Our simple greedy algorithm (Algorithm 5) includes an optimization that is important in practice, but does not affect the worst-case run time. After selecting a topic, the contribution of other topics needs to be updated. Here we observe that, due to submodularity, the contribution of those topics can only decrease. Thus, we loop over the sorted list of topics in descending order of value and stop updating as soon as the contribution of the topic with maximum contribution (top topic in max-heap) does not change. This is done between Lines 9 and 12.

Theorem 6.12. *Algorithm 7 has an approximation ratio of* $\frac{1}{2}\left(1 - \frac{1}{e}\right)$.

Proof. A general result for budgeted maximization of submodular functions was given by Krause and Guestrin [Krause and Guestrin, 2011][Theorem 1]. Our Algorithm 7 is a minor extension of theirs, the difference being that they only select a single element when $type = \mathcal{G}$. □

We remark that, following [Krause and Guestrin, 2011], one can also create a greedy heuristic with an approximation ratio of $1 - \frac{1}{e}$ at the cost of an additional factor of $|T|^3$ in the running time of the algorithm.

Theorem 6.13. *Given an instance* B3M*$(T, V, ev, cost, Int, \tau, \mathcal{C})$ where costs are normalized, for any solution $\mathcal{S}$ it holds that:*

$$\sigma(\mathcal{S}) \leq \max\left(\left|V'\right| : \sum_{v \in V'} \max\left(\tau_v, \min_{t \in T_v} ev_t\right) \leq \mathcal{C}\right) + 1\,,$$

where $V' \subseteq V$.

Note that Theorem 6.13 is an extension of Theorem 6.7 with a minor difference in that there may be a fractional contribution to the objective function. This fractional part is upper bounded by 1.

Theorem 6.14. *Algorithm 7 has run time complexity of* $\mathrm{O}\big(|T|^2(|V| + \log|T|)\big)$.

Proof. The same proof for Theorem 6.6 is applicable here. Hence, the proof is omitted. □

6.5 Evaluations

6.5.1 Experimental Setup

We implemented both GreedyB3M and ModifiedGreedyFB3M using C++. To evaluate these heuristics we make use of real data from Spotify's deployed pub/sub system. The data consists of about 1.1 million topics and 4.9 million subscribers. The traces were gathered for 10 days from Spotify's datacenter at Stockholm. For more information about the data traces we refer to Section 5.1. We use the normalized cost function: for each topic $cost(t) = ev_t \cdot |V_t|$. To choose $\mathcal{C}$ we analyzed the full data traces and computed the total capacity needed to handle the full traces in terms of the total cost of all the topics $\sum_{t \in T} cost(t)$. Unless mentioned explicitly, for evaluations in this chapter we set the capacity constraint $\mathcal{C}$ to be 10% of this sum. For τ we used 1%(27) to 100%(2763) of the mean event rate of all the topics. All experiments were executed single threaded on a server with 16 cores of Intel Xeon 2.13GHz processors and 32 GB of RAM.

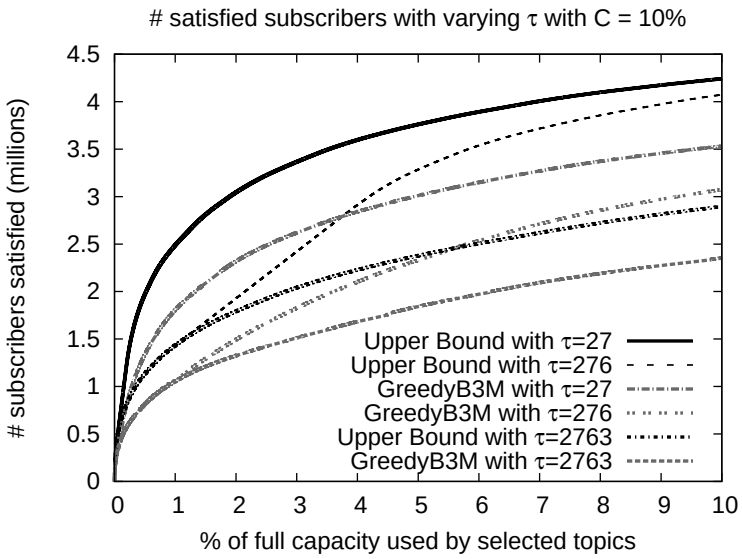

Figure 6.2: Comparison of GreedyB3M with the Estimated Upper Bound

6.5.2 Performance of GreedyB3M

First we analyze the performance of GreedyB3M (Algorithm 2) comparing it to the upper bound computed by GetUpperBound (Algorithm 3). To visualize the performance we observe that both algorithms iteratively construct solutions. Thus, in Figure 6.2 we show the progress of the GreedyB3M algorithm after selecting a topic in each iteration, by comparing the service capacity used so far (x-axis) against the number of satisfied subscribers (for a given τ) (y-axis) by the chosen topics. Note that this represents a single run of GreedyB3M until a budget $\mathcal{C}$ of 10% of the workload is reached. However, the intermediate results are equivalent to having stopped GreedyB3M at the corresponding values of $\mathcal{C}$. We can see that the gap between GreedyB3M and the upper bound increases as $\mathcal{C}$ also increases in most cases when $\mathcal{C}$ is restricted to 10%.

An interesting observation is that, with $\mathcal{C}$ equivalent to 10% of what is needed to handle the full workload, the gap between GreedyB3M and the upper bound increases as τ increases from 27 to 276 (the approximation ratio drops from 0.87 to 0.75, as shown in Figure 6.3). However, this changes when τ is increased to 2763,

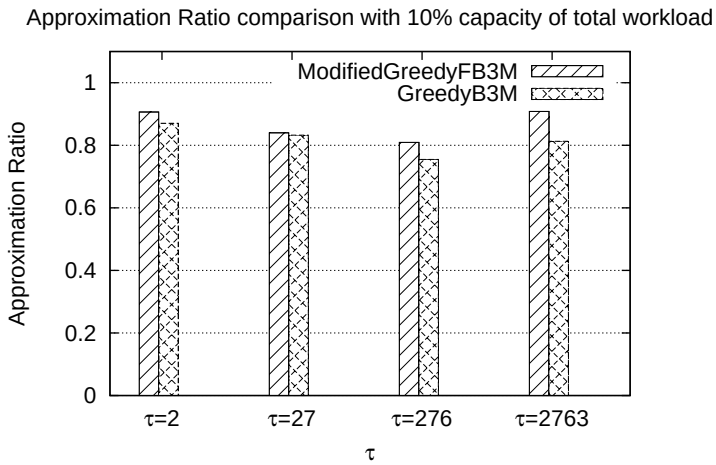

Figure 6.3: Computed approximation ratios for *B3M* and *F-B3M* with varying τ

in which case the approximation ratio of GreedyB3M increases from 0.75 to 0.82. $\tau \approx 27$ is a reasonably realistic value. With this parameter we satisfy around 72% of all subscribers (3.5 million of the total 4.9 million). The upper bound gives that at most 86% (4.2 million) of subscribers can be satisfied, with an approximation ratio of around 0.83.

6.5.3 Performance of ModifiedGreedyFB3M

We now analyze the performance of ModifiedGreedyFB3M. From the theoretical results, we know that ModifiedGreedyFB3M guarantees an approximation ratio of $\frac{1}{2}\left(1-\frac{1}{e}\right)$. In our real-world data set, we achieve a significantly better ratio (up to 0.9). Analogously to our analysis of GreedyB3M, we use the upper bound given by Theorem 6.13. This theorem can be easily turned into an algorithm identical to Algorithm 3 but with the change that in the last step (line 9) we return $|csubs|+1$ instead. Since the goal of *F-B3M* is to maximize the total satisfaction fraction among the subscribers of all the topics, the outcome is measured in terms of total fraction instead of number of subscribers. As shown in Figure 6.4, a similar pattern to GreedyB3M is observed in the approximation ratio when the τ changes from 27 to 2763. However, the gap between the ModifiedGreedyFB3M and the upper bound is much lower compared to the gap between GreedyB3M and its

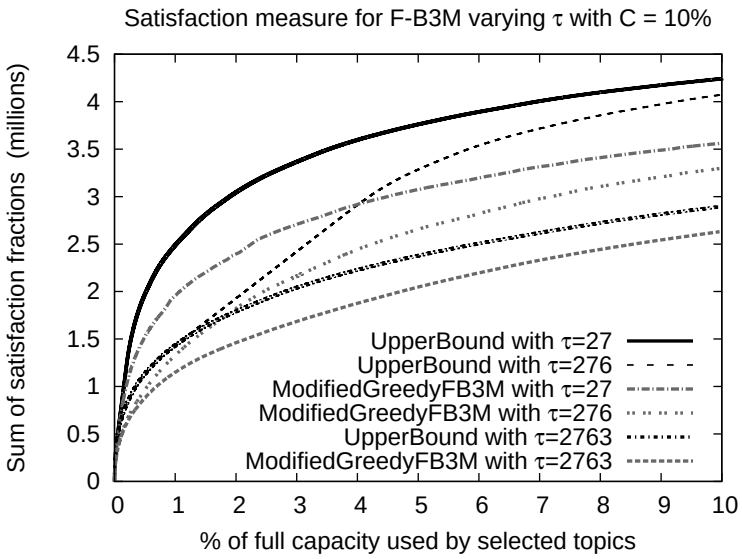

Figure 6.4: Comparison of ModifiedGreedyFB3M with the Estimated Upper Bound

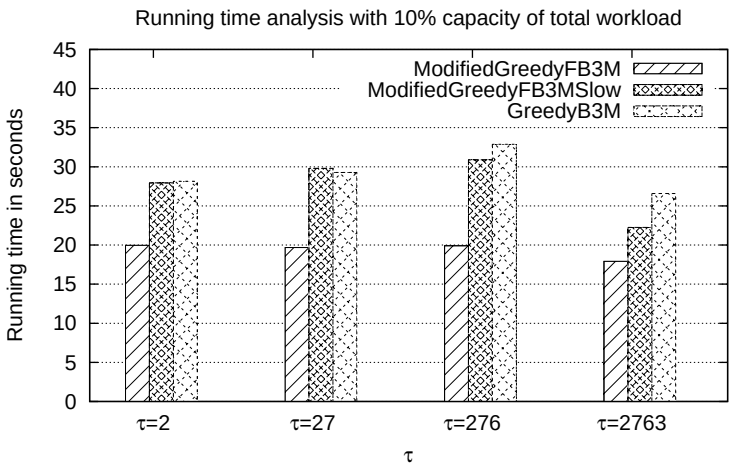

Figure 6.5: Running time comparison for Greedy Heuristics with varying τ

corresponding upper bound. For example for $\tau = 2763$ the approximation ratio between ModifiedGreedyFB3M and the upper bound is 0.9 compared to 0.82 for GreedyB3M, as shown in Figure 6.3.

GreedyB3M and ModifiedGreedyFB3M algorithms are intended to run on a

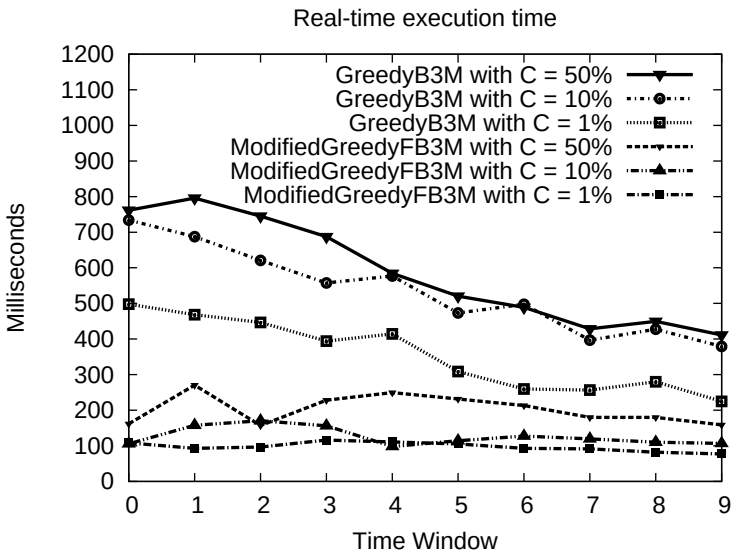

Figure 6.6: Real-time execution time of heuristics

regular basis, thus it is important that they are fast. In Figure 6.5 the running times of the greedy approaches proposed in this chapter are shown in seconds (mean of 3 runs). We also introduce a naive version coined ModifiedGreedyFB3MSlow, to evaluate the gain of exploting submodularity structure to lazily updating costs in ModifiedGreedyFB3M as explained in Section 6.4.1 . ModifiedGreedyFB3MSlow is identical to ModifiedGreedyFB3M except between lines Lines 9 and 12 of Algorithm 5. Instead of lazily updating topic costs, all the topics that have a common subscriber with the chosen topic in the current iteration are updated (same as lines 6 and 7 of Algorithm 2). From Figure 6.5 it is clear that ModifiedGreedyFB3M outperforms ModifiedGreedyFB3MSlow and runs in less than 20 seconds for all values of τ, while without optimization it takes a maximum of 33 seconds to run for $\tau = 276$. It is clear that these algorithms are in general fast to run in large-scale settings and can be run on a regular basis.

6.5.4 Real-Time Performance

The solutions for *B3M* and *F-B3M* are expected to be run periodically to recompute the solution. In these periodic computations, the input sizes are smaller as they only need to provide a solution until the next computation, meaning that topics without publications can be ignored. To evaluate their performance in this scenario, we use the stream of publications from Spotify with a fixed $\tau = 20$ and $\mathcal{C}$ varying from 1% to 50%. We divide the stream in smaller time windows, where each window is an hour long. We then execute our algorithms for the topics active in 10 consecutive time windows. In Figure 6.6 we show the execution time of the GreedyB3M and ModifiedGreedyFB3M algorithms. The algorithms execute in just a few hundred milliseconds, and ModifiedGreedyFB3M executes at least twice as fast as GreedyB3M due to the proposed optimization. The running times reflect the size of the workload and, for a typical workload in Spotify, the solutions are suitable for periodic execution in real-time. In Figure 6.7 we show that both heuristics provide similar approximation ratios. However, ModifiedGreedyFB3M performs slightly better in all cases. An interesting observation is that, as $\mathcal{C}$ increases, the approximation ratios also increase.

6.6 Summary

In this chapter, motivated by practical scenarios in a real deployed pub/sub system at Spotify, we proposed a new approach to maximize subscriber satisfaction. In the process, we introduced a new set of problems (*B3M* and *F-B3M*) to address the maximization of the number of satisfied subscribers in a pub/sub system and proposed greedy heuristics to solve both problems. We proved that *B3M* is NP-Hard by reduction from the *DkS* problem and, as a corollary, also proved that *B3M* has no PTAS under a standard assumption. *F-B3M* is a relaxed version of *B3M* that is relatively easy to solve. We proved that the objective function of *F-B3M* is submodular, derived a constant approximation bound for its greedy heuristic, and proposed a way to exploit the

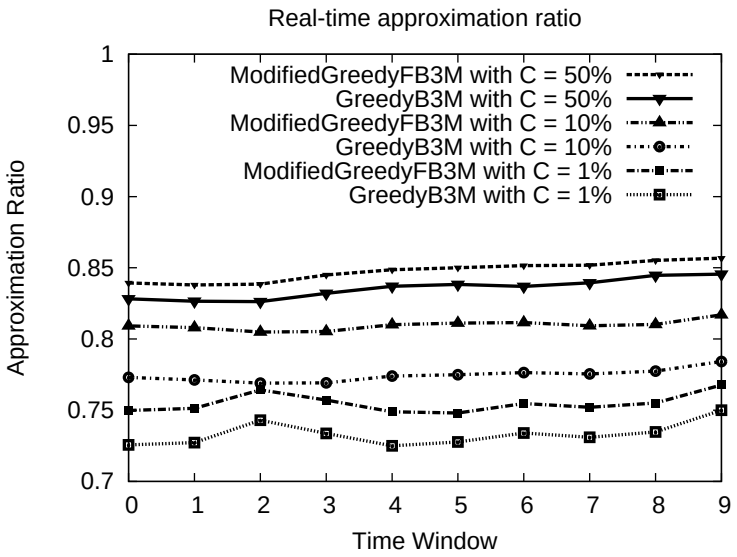

Figure 6.7: Approx. ratio measured in real-time

submodularity of the objective function to improve the running time of the heuristic for typical scenarios. We evaluated our heuristics for both problems using a large-scale real data set from Spotify's pub/sub system and compared their performance with upper bounds we derived for the optimal solutions of both problems. We illustrated that, with a realistic pub/sub workload as input, our heuristics achieve an approximation ratio of at least 0.7 and they can be run in under a second in a realistic scenario to adapt to the workload variations. We conclude that we have demonstrated that there is theoretical and practical evidence that pub/sub systems (like Spotify's pub/sub) can benefit from the algorithms presented in this chapter.

Chapter 7

Resource Provisioning for Scalable Publish/Subscribe to Drive Social Interaction

Traditionally, pub/sub engines have been deployed on in-house enterprise clusters. However, with the advent of cloud computing, a viable alternative of running pub/sub services in the cloud became available. An enterprise may choose between using a generic pub/sub engine (such as Azure Service Bus or PubNub included in Microsoft Azure and Amazon EC2, respectively) and moving the deployment of its proprietary engine optimized for the application needs to the cloud. While the questions of cloud resource allocation and cost become critical in this context, they have never been considered for pub/sub services.

In this chapter, we consider the problem of resource provisioning for a special class of pub/sub systems designed to drive notifications due to online social interaction among users. For example, as described in Section 4.2.1, in Spotify, a pub/sub engine is used to notify users about the music activity (e.g. music playback, playlist updates) of their friends and favorite artists. Another example is Twitter, where users can follow any other user, and published tweets are disseminated to all the following users. In such systems, we can model users as both topics and subscribers. A user is a topic if she has followers subscribing to

her publications and at the same time, she can be a subscriber if she follows some users.

The pub/sub applications for social interaction are characterized by a significant data volume, e.g., the Spotify pub/sub service described in Section 4.2.1 is required to send an order of 2 Terabytes of notifications every day and Twitter is known to send at least 8 Terabytes of tweets every day [Krikorian, 2010]. In addition to that, each user generally subscribes to a high number of notifications. For example, in a sample we analyzed, more than 3 million users were receiving more than 1000 tweets per day. In such applications, every notification is intended to be read by a human user so that having a cumulative delivery rate to a particular subscriber above a certain threshold will not bring any benefit. To this end, in Chapter 6 we defined *satisfaction metrics* that ensure delivery rates of at least a predefined threshold, but, past this threshold, users are not considered to be more satisfied. Therefore, in order to guarantee that every subscriber is satisfied, the system has to ensure that the rate of notifications of interest delivered to each subscriber is not below a configurable satisfaction threshold delivery rate.

Intuitively, a pub/sub system designed to meet the satisfaction threshold for all subscribers itself can save significant amount of resources (e.g. number of servers and bandwidth consumed). However, given the large-scale workload to be handled by such pub/sub engines, distributing the workload on several servers becomes inevitable. Therefore, a pub/sub engine to be deployed on a datacenter or a public cloud could benefit from a tool to estimate and minimize the total costs involved.

In order to include monetary costs of resources (VM deployment cost and bandwidth cost) in our problem, we adopt a standard pricing model used by Infrastructure-as-a-Service (IaaS) providers such as Amazon EC2. This model includes separate expense components due to the use of virtual machines and bandwidth, under resource constraints for individual virtual machines. We formulate a problem of Minimum Cost Subscriber Satisfaction (*MCSS*), that is how to allocate resources for the given pub/sub workload so as to minimize the cost while keeping every subscriber satisfied. While the main goal of solving this

problem is to help companies that move their operation to the cloud, the problem is also beneficial for minimizing resource consumption for companies that continue using in-house deployment. As we show later in Section 7.1.1, sometimes, there is an interesting trade-off between minimizing resources of different types: minimizing the number of virtual machines may lead to increased bandwidth consumption and vice versa. In other words, the problem of optimizing the cost is more complex than just separately minimizing resources of each type.

In this chapter, to the best of our knowledge we provide the first formal treatment of this subject. We prove *MCSS* to be NP-hard and provide an efficient heuristic solution. The solution works in two stages: first we select a subset of the workload that is sufficient for satisfying all subscribers. Then, we assign the chosen subset to virtual machines using an algorithm based on a customized version of bin packing, with a number of optimizations. While separating between the two stages may lead to sub-optimality in the solution, we show experimentally that this sub-optimality is insignificant for practical workloads.

We evaluate the solution empirically using large-scale real traces from Spotify and Twitter. We use two baselines in the evaluation: a (possibly non-tight) lower bound as well as a naive solution. We show that the proposed approach can cut down costs by up to 74% with Twitter traces and up to 38% with Spotify traces when compared to the naive alternative. On the other hand, our solution performs only 15% worse compared to the lower bound in many cases. Additionally, we show how we gradually improve the results by incrementally introducing a number of optimizations and evaluating the impact of each optimization. The proposed solution runs in under 30 seconds for the Spotify workload with 5 million subscribers and 1.1 million topics and under 25 minutes for the Twitter workload with 30 million subscribers and 8 million topics.

In summary, our main contributions in this chapter include: (1) a technique to estimate the amount of resources needed to deploy pub/sub for social interaction on datacenters, (2) cost-effective resource provisioning based on the Amazon EC2 pricing model, (3) formalization of the resource provisioning problem for pub/sub,

and (4) a large-scale empirical evaluation to show the practical benefits of our solution.

7.1 Resource Provisioning Model and Problem Definition

7.1.1 Intuition for the Resource Provisioning Model

Customers of IaaS providers can usually rent virtual machines (VMs) of certain predefined CPU, memory and bandwidth capacities either on an hourly basis or for a fixed duration. In addition to this, they are also charged by the total incoming and outgoing (to and from the cloud) bandwidth consumption of their application. Our goal is to find an allocation of the pub/sub workload to a set of VMs such that it minimizes the total monetary cost (combined cost of VM utilization and bandwidth) while ensuring that all subscribers are satisfied.

Intuitively, the monetary costs of deploying a pub/sub system in the cloud is directly proportional to the size of the workload it will handle (e.g. number of publications and number of recipient subscribers). Hence, choosing a subset of workload amounting to the least bandwidth consumption so as to meet the satisfaction of all subscribers can readily save costs. In our model, each topic has its own publication rate and choosing the subset of the topics to meet satisfaction metrics can reduce the workload. However, selecting a topic with all of its subscribers may not always be beneficial to all the subscribers. On the other hand, if we have a choice to include or exclude topic-subscriber pairs, depending on their contribution to the satisfaction of subscribers, we can choose a more resource-efficient workload and do a cost-effective allocation. Thus, in our model we choose a subset of the pub/sub workload at the granularity of topic-subscriber pairs.

To simplify the problem, the only capacity constraint we take into account for allocating load to a VM is the VM's bandwidth capacity. We do not explicitly consider the constraints on other resources such as CPU, memory and disks. The reason is that, in our system, resource consumption is driven by the

delivery of publications to subscribers, which is essentially a network-bounded operation. Thus, bandwidth constraints also serve as constraints on other VM resources. A pub/sub system generally has an incoming stream of publications for each topic and an outgoing stream of notifications to all the subscribers of the topic, thus requiring incoming and outgoing bandwidth resources for deployment on the cloud. In our model, we consider minimizing both incoming as well as outgoing bandwidth. Typically, every IaaS provider has different costs for incoming and outgoing bandwidth consumption. However, to simplify the problem, we assume they cost the same and that each VM has the same incoming and outgoing bandwidth capacity.

Given that we want to minimize the cost of VM utilization and the cost of bandwidth consumption, it is worth noting that there is a trade-off between the number of VMs and the amount of bandwidth that is needed to satisfy all subscribers. For example, consider a user v subscribing to topics t_1, t_2, t_3. Assume that the satisfaction threshold is specified in such a way that t_1, t_2 together satisfy v or t_3 alone satisfies v (shown in Figure 7.1(a)). In addition, assume that there are two VMs b_1 and b_2 with available capacity as shown in Figure 7.1(b). Based on the definition of our satisfaction metric, there are two possible solutions to meet the satisfaction of subscriber v, as shown in Figure 7.2. The allocation shown in **Solution 2** uses three VMs and yet consumes less bandwidth than the allocation done in **Solution 1** with two VMs.

To balance this tradeoff, we set the goal of our problem as to minimize the combined cost of VM utilization and total bandwidth consumption. We define an optimization problem with this objective in Section 7.1.3. In Section 7.2.2 as part of the solution, we employ various optimization techniques to balance the above mentioned tradeoff.

7.1.2 Model and Notations

Before we define the problem more formally, we introduce the following notations which were partly introduced in the Section 6.2:

T **:** A collection of l *topics* $\{t_1, t_2, ..., t_l\}$ in the system.

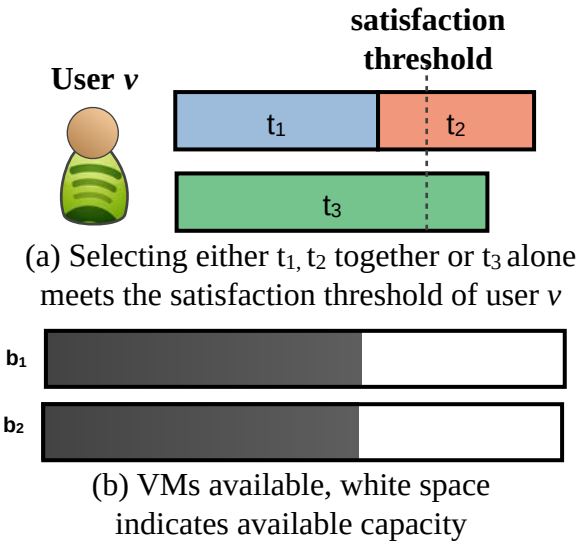

Figure 7.1: Tradeoff scenario

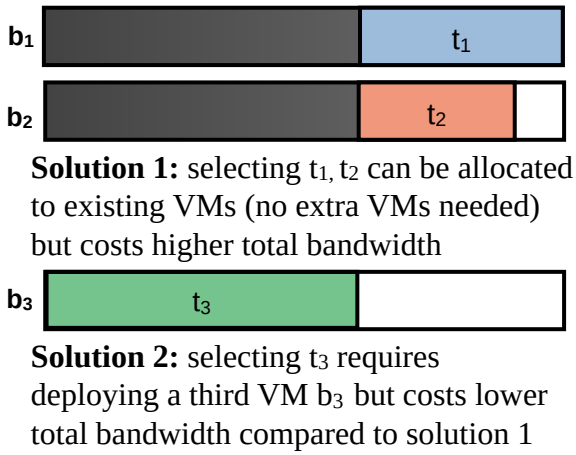

Figure 7.2: Two possible allocations to meet satisfaction threshold of the user v.

V **:** A collection of n *subscribers* $\{v_1, v_2, ..., v_n\}$ participating in the pub/sub system. A subscriber can subscribe to one or more topics from T. Subscribers in a typical pub/sub system are generally end-user applications (e.g. Spotify client software). In the rest of this chapter we use subscribers and users interchangeably.

T_v **:** The *interest* of subscriber v, that is, the collection of topics subscribed by v.

Int **:** The collection of interests $\{T_{v_1}, T_{v_2}, ..., T_{v_n}\}$ for all subscribers in V.

ev_t **:** *Event rate* of the publications generated for a topic t, that is, the average number of events published to topic t during a time unit (e.g., per minute or per hour). Without loss of generality, we assume that $ev_t > 0$. When we say 'event' in the rest of this chapter we mean a publication event message generated by the publisher of a topic intended for all subscribers of the topic.

τ **:** A system parameter that represents the *satisfaction threshold* for a subscriber. It is defined as a constant specifying the number of events to be delivered to a subscriber in order for the subscriber to be considered satisfied.

τ_v **:** Subscriber-specific satisfaction threshold. In practice, the total event rate of the topics subscribed to by a subscriber is sometimes less than τ. In such cases we need to serve all the events the subscriber is interested in to meet the satisfaction threshold. It can be expressed as follows: $\tau_v = min(\tau, \sum_{t \in T_v} ev_t)$.

$V_t \subseteq V$**:** The (non-empty) set of subscribers to topic t. Given *Int*, V_t can be derived trivially.

$cost(t, v)$**:** Represents the non-zero cost of serving a topic-subscriber pair (t, v) by any server/VM. For evaluation purpose $cost(t, v) = 2 \cdot ev_t$, to include both incoming and outgoing bandwidth requirements which are proportional to the event rate of the topic.

$\mathcal{C}_1$ **:** A function to compute the cost of renting virtual machines from the cloud service provider.

$\mathcal{C}_2$ **:** A function to compute the cost of consuming the total bandwidth (both incoming and outgoing) on the cloud by a given pub/sub workload. Note that, to simplify the problem, we assume the same cost function to compute the cost of both incoming as well as outgoing bandwidth.

BC **:** A fixed bandwidth capacity of a virtual machine which cannot be exceeded. We assume that bandwidth capacity includes both incoming and outgoing bandwidth capacity. We exclude the bandwidth consumed by any communication between the VMs in this capacity. We assume that BC is large enough to accomodate a topic-subscriber pair with maximum total bandwidth requirement i.e, $BC \geq \max_{t \in T} 2 \cdot ev_t$

bw_b **:** The total bandwidth consumption (incoming as well as outgoing) of virtual machine b. It must be ensured that bw_b never exceeds BC.

$\mathcal{B}$ **:** A set of virtual machines allocated to handle the given pub/sub workload, and an individual virtual machine is referred to as $b \in \mathcal{B}$. We want to minimize $\mathcal{C}_1 (|\mathcal{B}|) + \mathcal{C}_2 \left(\sum_{b \in \mathcal{B}} bw_b\right)$.

7.1.3 Formal Definition of the Minimum Cost Subscriber Satisfaction (*MCSS*) Problem:

Given an instance of T, V and their interests Int, the goal of $MCSS(T, V, ev, Int, \tau, BC, \mathcal{C}_1, \mathcal{C}_2)$ is to determine the minimum cost in terms of the number of required VMs and the total bandwidth consumed to satisfy all the subscribers.

To capture the allocation of topic-subscriber pairs to a VM we introduce an integer variable $x_{tvb} = 0, 1$ which is 1 if the topic-subscriber pair tv is assigned to the virtual machine b.

$$x_{tvb} = \begin{cases} 1 & \text{if } tv \text{ is assigned to } b \\ 0 & \text{otherwise} \end{cases} \tag{7.1}$$

We now define the problem more formally below:

$$\begin{aligned}
&\text{Minimize} && \mathcal{C}_1\left(|\mathcal{B}|\right) + \mathcal{C}_2\left(\sum_{b\in\mathcal{B}} bw_b\right) \\
&\text{Where,} && bw_b = \sum_{v\in V}\sum_{t\in T} x_{tvb} ev_t + \sum_{t\in T}\left(\max_{v\in V_t} x_{tvb}\right) ev_t \\
&\text{Subject to:} && bw_b \leq BC, \forall b \in \mathcal{B} \\
& && \sum_{v\in V} f_v = |V|
\end{aligned} \tag{7.2}$$

Where, f_v is a an integer variable that indicates if subscriber v is receiving a number of events that meets the satisfaction threshold:

$$f_v = \begin{cases} 1 & \text{if } \sum_{t\in T_v}\left(\max_{b\in\mathcal{B}} x_{tvb}\right) ev_t \geq \tau_v \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

In the above definition the total bandwidth bw_b consumed by a VM b is defined as the sum of two expressions. The first expression represents the outgoing traffic (number of topic-subscriber pairs assigned to b multiplied by the event rates of the topics). The second expression represents the incoming traffic, which is exactly the sum of the event rates of the unique set of topics that are assigned to a VM b. The goal of $\max_{v\in V_t} x_{tvb}$ in Equation (7.2) is to avoid adding the event rate of a topic once for each pair and instead only once per VM. In Equation (7.3) we use $\max_{b\in\mathcal{B}} x_{tvb}$ to ensure that a topic-subscriber pair (t, v) is considered towards satisfaction of v only if (t, v) is allocated to at least one VM b.

We also define $DCSS(T, V, ev, Int, \tau, BC, \mathcal{C}_1, \mathcal{C}_2, C_T)$, the corresponding decision problem of *MCSS*, which is to determine if it is possible to achieve a total cost of at most C_T, where, C_T is a given constant.

7.1.4 Hardness of *DCSS* Problem

To establish the hardness of *DCSS* we prove that the well-known NP-Hard problem Partition Problem (*PP*) [Garey and Johnson, 1979] can be reduced to a special case of *DCSS*. We now define the *PP* problem.

Definition 7.1 (Partition Problem (*PP*) [Garey and Johnson, 1979]). *The task of an instance of a partition problem* PP(S) *is deciding whether a given multiset* $S = \{x_1, x_2, ..., x_n\}$ *of positive integers* x_i *can be partitioned into two subsets* S_1 *and* S_2 *such that* $\sum_{x_j \in S_1} x_j = \sum_{x_k \in S_2} x_k$ *and* $S \setminus S_1 = S_2$.

Theorem 7.2. *DCSS is NP-Hard.*

Proof. Given an instance of $PP(S)$, we create an instance of *DCSS* in the following way: For each integer $x_i \in S$, create a topic t with $ev_t = x_i$ and a single subscriber v_i of the topic. This means that each topic t costs $2x_i$ bandwidth to be served since the incoming and outgoing bandwidth each cost x_i respectively. Set $BC = \sum_{x_i \in S} x_i$ and $\tau = max_{x_i \in S} x_i$ to ensure all topic- subscriber pairs are selected as part of the solution. We also set $\mathcal{C}_1(x) = x$, and $\mathcal{C}_2(x) = 0$, meaning that the cost of a solution will be the number of VMs used. Finally, we set the cost threshold C_T for the decision problem *DCSS* as 2.

With this reduction, a reduced instance of *PP* is in essence the same instance where all input values have been doubled. In the reduced instance, all topic-subscriber pairs must be picked and this will use up exactly as much bandwidth as 2 VMs have. Thus, if the reduced instance is a yes instance, a partition can be achieved by letting S_1 consist of all topics served by one VM. □

7.2 Solution Approach

The Integer Program formulation of *MCSS* defined in Section 7.1 is NP-Hard according to Theorem 7.2 and hence it is expensive to solve optimally in practice. Specifically, with the typical scale of pub/sub systems consisting of millions of topics and subscribers we need to deal with millions of variables to be considered in Equation (6.1). To the best of our knowledge, we are not aware of any IP solvers with the ability to scale to millions of variables. Instead, we propose a heuristic approach to solve *MCSS*. We solve the *MCSS* problem by dividing it into two relatively simpler sub-problems which are solved one after the other, thereby introducing two stages in our solution.

In the first stage, we solve a simplified version of the *MCSS* in which we

are given a hypothetical single VM with unlimited capacity. Then the goal is to meet the satisfaction threshold of all subscribers by selecting topic-subscriber pairs and allocating them to this hypothetical VM with unlimited bandwidth capacity. This sub-problem aims at selecting those pairs that minimize the total bandwidth consumption. After having solved the first stage, we move on to the second stage, in which we know that the output of Stage 1 satisfies the constraint $\sum_{v \in V} f_v = |V|$ from Equation (6.1). The goal of the second stage is to allocate the selected pairs to VMs in a manner to satisfy the capacity constraints of the VMs from Equation (6.1). We also want to consider the trade-off between the number of VMs and total bandwidth consumption explained in Section 7.1.1.

7.2.1 Stage 1: Selection of Topic-Subscriber Pairs

The pseudocode of Stage 1 is presented in Algorithm 9. In this stage, for each subscriber, we select a subset of topic-subscriber pairs that meet the satisfaction threshold of the user while trying to minimize the bandwidth cost. Note that, for each subscriber, it is basically a variant of the knapsack problem [Martello and Toth, 1990] that can be solved optimally using dynamic programming. However, given the large number of subscribers and topics, the optimal solution is too costly in terms of execution time. Instead, we solve the problem using a greedy heuristic based on a benefit-cost ratio for each (t, v) pair (see Algorithm 8.).

The cost of a (t, v) pair is the amount of bandwidth it requires, which is $2 \cdot ev_t$ for every (t, v). This is the amount of (incoming) bandwidth required to push events for topic t into the individual VMs plus the amount of (outgoing) bandwidth required to deliver the event to user v.

We define the benefit of (t, v) in terms of the contribution of t towards the satisfaction of user v. To determine this benefit, we first calculate the remaining event delivery rate required to satisfy v, which we refer to as rem_v, which is τ_v minus the sum of the event rates of the topics already included in the solution to which v has subscribed (see Line 2). If v is already satisfied without adding (t, v), then the benefit of (t, v) is zero. If including (t, v) in the solution makes v satisfied, then the benefit of (t, v) is 1 (maximum benefit value); otherwise, the

Algorithm 8: Heuristic value of topic, subscriber pair (t, v) having selected $\mathcal{S}$

```
1  GetBenefitCostRatio(t, v, τ_v, cost(t, v), S)
   Input: t, v, τ_v, cost(t, v), S
   Data: benefit ← 0 : Benefit of t towards v
   rem_v ← 0: Remaining event rate needed to satisfy user v
2  rem_v ← τ_v − Σ_{{(t',v):(t',v)∈S∧t'∈T_v}} ev_{t'}
3  if rem_v > 0 then
4      benefit ← min(1, ev_t / rem_v)
5  return benefit / cost(t,v)
```

$$\text{benefit} \leftarrow \min\left(1, \frac{ev_t}{rem_v}\right)$$

$$\text{return } \frac{\text{benefit}}{cost(t,v)}$$

Algorithm 9: Stage 1 of solution for ***MCSS***: Greedy pair selection

```
1  GreedySelectPairs(T, V, ev, cost, Int, τ)
   Input: T, V, ev, Int, cost, τ
   Data: A : Array of size |T|
   Result: S ← ∅ : Output set of (t,v) pairs
2  foreach v ∈ V do
3      τ_v ← min(τ, Σ_{t∈T_v} ev_t)
4      foreach t ∈ T_v do
5          A[t] ←GetBenefitCostRatio(t, v, τ_v, cost(t, v), S)
6      while Σ_{(t,v)∈S} ev_t < τ_v do
7          t ← argmax_{{t'∈T_v}} A[t']
8          S ← S ∪ {(t, v)}
9          A[t] ← 0
10         foreach t' ∈ T_v do
11             if (t', v) ∉ S then
12                 A[t'] ← GetBenefitCostRatio(t', v, τ_v, cost(t, v), S)
13 return S
```

$$\text{GreedySelectPairs}(T, V, ev, cost, Int, \tau)$$

$$\text{Input: } T, V, ev, Int, cost, \tau$$

$$\text{Data: } A : \text{Array of size } |T|$$

$$\text{Result: } \mathcal{S} \leftarrow \emptyset : \text{Output set of (t,v) pairs}$$

$$\text{foreach } v \in V \text{ do}$$

$$\quad \tau_v \leftarrow min(\tau, \textstyle\sum_{t \in T_v} ev_t)$$

$$\quad \text{foreach } t \in T_v \text{ do}$$

$$\quad\quad A[t] \leftarrow \text{GetBenefitCostRatio}(t, v, \tau_v, cost(t, v), \mathcal{S})$$

$$\quad \text{while } \textstyle\sum_{(t,v) \in \mathcal{S}} ev_t < \tau_v \text{ do}$$

$$\quad\quad t \leftarrow \text{argmax}_{\{t' \in T_v\}} A[t']$$

$$\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{(t, v)\}$$

$$\quad\quad A[t] \leftarrow 0$$

$$\quad\quad \text{foreach } t' \in T_v \text{ do}$$

$$\quad\quad\quad \text{if } (t', v) \notin \mathcal{S} \text{ then}$$

$$\quad\quad\quad\quad A[t'] \leftarrow \text{GetBenefitCostRatio}(t', v, \tau_v, cost(t, v), \mathcal{S})$$

$$\text{return } \mathcal{S}$$

benefit is the ratio ev_t/rem_v (Line 4).

Under this heuristic, all topics that contribute to satisfy v without exceeding the satisfaction threshold have the same benefit-cost ratio and are preferred over those that exceed the threshold. The latter are penalized in proportion to the cost they introduce (Line 5).

For each subscriber, all pairs with topics in T_v are potential candidates for our solution. However, we want to select the pairs with the least bandwidth costs.

In this regard, for each candidate pair the benefit-cost ratio is computed using Algorithm 8 and stored in an array A (from Line 4 to Line 5 of Algorithm 9). Then, we select the (t, v) pair with maximum heuristic value in each iteration until the satisfaction threshold τ_v for subscriber v is met (from Line 7 to Line 12). In each iteration after selecting a (t, v) pair, the heuristic value of the rest of the pairs is updated since the benefit of a pair (t_2, v) decreases after having chosen (t_1, v) as the remaining number of events decreases. A set of all the chosen pairs for every subscriber V is returned in Line 13.

As an example for the selection of topic-subscriber pairs consider the scenario in Figure 7.1 of Section 7.1.1. According to our heurisitc, both t_1 and t_2 have benefit-cost ratio of $1/\tau_v$ but t_3 has benefit-cost ratio of $1/(2 \cdot ev_{t_3})$. Assuming $ev_{t_3} > \tau_v > ev_{t_1} > ev_{t_2}$, (t_1, v) and (t_2, v) pairs are selected as part of the solution and (t_3, v) is omitted.

In order to illustrate the importance of cost-effective selection topic-subscriber pairs, we compare and contrast **GreedySelectPairs** (*GSP*) against a naive solution **RandomSelectPairs** (*RSP*). In RandomSelectPairs, for each subscriber select an arbitrary subset of topics so as to satisfy the subscriber while ignoring the benefit-cost ratio. Specifically, for each subscriber v in V, enough (t, v) pairs are selected in no particular order to reach the satisfaction threshold τ_v.

7.2.2 Stage 2: Allocation of Topic-Subscriber Pairs to VMs

In the second stage, the goal is to allocate the topic-subscriber pairs in $\mathcal{S}$ selected from Stage 1 to VMs. It is interesting to note that the goal of our second sub problem is very similar to the well-known *bin packing* problem [Lewis, 2009]. Hence, as a first attempt we propose First-Fit Bin Packing **FFBinPacking** (*FFBP*) (e.g. used in [Genaud and Gossa, 2011; Villegas et al., 2012]) as a solution for Stage 2. In Algorithm 10, the pseudocode to allocate the topic-subscriber pairs to VMs in a First-Fit manner is given. Each topic-subscriber pair in $\mathcal{S}$ is considered in no particular sequence (Line 2 to Line 20). If a pair (t, v) can be allocated to an existing VM it is done so with the

Algorithm 10: First-Fit Bin Packing Algorithm for Stage 2 of ***MCSS***:

```
1  FFBinPacking(S, BC)
   Input: S, BC
   Result: B ← ∅ : Set of VMs with allocated (t,v) pairs
2  foreach (t, v) ∈ S do
       // Try assigning to the first VM that can fit (t, v)
3      foreach b ∈ B do
4          if ∀v' ∈ V_t : (t, v') ∉ b then
5              if 2 · ev_t ≤ BC − bw_b then
6                  b ← b ∪ (t, v)
7                  S ← S \ (t, v)
8                  bw_b ← bw_b + 2 · ev_t
9                  break
10         else if ev_t ≤ BC − bw_b then
11             b ← b ∪ (t, v)
12             S ← S \ (t, v)
13             bw_b ← bw_b + ev_t
14             break
       // Deploy new VM if existing VMs cannot fit (t, v)
15     if (t, v) ∈ S then
16         b ← new VM with bandwidth capacity BC
17         B ← B ∪ b
18         b ← b ∪ (t, v)
19         S ← S \ (t, v)
20         bw_b ← bw_b + 2 · ev_t
21 return B
```

first found VM having enough free capacity to include it (Line 2 to Line 12). If none of the existing VMs has enough free capacity to include (t, v), a new VM is deployed and added to the collection $\mathcal{B}$ of existing VMs (Line 16 to Line 20).

While the First-Fit strategy for bin packing is simple and strives to minimize the number of VMs used, in our setting, it is not favorable with respect to bandwidth consumption. We illustrate this with an example. Consider a case with two topics t_1 and t_2 with $ev_{t_1} = 20$ events/min and $ev_{t_2} = 10$ events/min with each message around 1KB, let $\tau = 30$ events/min and consider 3 subscribers forming 5 pairs: $(t_1, v_1), (t_2, v_1), (t_2, v_2), (t_1, v_2), (t_2, v_3)$. Assume there are two VMs b_1 and b_2 with a remaining capacity of 60 KB/min and 100 KB/min

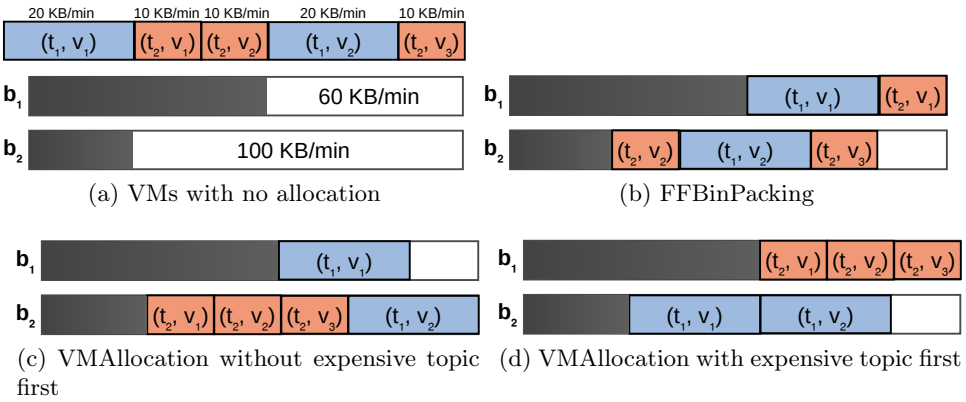

Figure 7.3: Various VM allocation optimizations

respectively (both incoming and outgoing bandwidth combined in Figure 7.3a) and their respective occupied capacity is shown in dark grey and their respective available capacity is left unfilled. In Figure 7.3b the outcome for *FFBP* from Algorithm 10 is shown. Because of the First-Fit strategy, the topic-subscriber pairs of the same topics are split across different VMs resulting in total bandwidth consumption (both incoming and outgoing) of 130 KB/min. Note that allocating t_1 and t_2 to both b_1 and b_2 results in additional overhead of replicating publications events, hence in total an extra 30KB/min (20KB/min from t_1 and 10KB/min from t_2) is contributed to the overall bandwidth consumption.

Here we make an important observation that *FFBP* has high runtime complexity of $\mathrm{O}\big(|T||V||\mathcal{B}|\big)$, because each topic-subscriber pair is considered individually. This can be improved if we group the topic-subscriber pairs of the same topic before allocating them to the VMs. This optimization, in addition to speeding up the algorithm, also has an advantage of saving bandwidth overhead. As all pairs of a topic are considered at the same time, the splitting of pairs across different VMs will be reduced, thereby reducing the incoming bandwidth overhead. This can be observed in Figure 7.3c. With this optimization the pairs

Algorithm 11: Stage 2 of ***MCSS***: Customized bin packing

```
1  CustomBinPacking(S, BC, C_1, C_2)
   Input: S, BC
   Data: P ← ∅ : Temporary set to hold topic-subscriber pairs to be
           allocated to VMs
   b ← new VM with bandwidth capacity BC : VM currently being
   allocated
   Result: B ← ∅ : Set of VMs with allocated (t,v) pairs
2  while S ≠ ∅ do
3      t ← argmax_{t'} Σ_{(t',v)∈S} ev_{t'}
4      foreach v ∈ V_t do // Group subscribers of topic t
5          if (t, v) ∈ S then
6              P ← P ∪ (t, v)
7              S ← S \ (t, v)
8      if CheaperToDistribute(t, B, BC, P, C_1, C_2) is true then
9          while P ≠ ∅ do
10             b ← argmax_{b'∈B}{BC − bw_{b'}}
11             if 2 · ev_t ≤ BC − bw_b then
12                 bw_b ← bw_b + ev_t
13                 while P ≠ ∅ and ev_t ≤ BC − bw_b do
14                     (t, v) ← any random (t', v'), such that (t', v') ∈ P
15                     b ← b ∪ (t, v)
16                     P ← P \ (t, v)
17                     bw_b ← bw_b + ev_t
       // For the remaining pairs deploy new VMs
18     while P ≠ ∅ do
           // Deploy new VM
19         b ← new VM with bandwidth capacity BC
20         B ← B ∪ b
21         bw_b ← bw_b + ev_t
22         while ev_t ≤ BC − bw_b do
23             b ← b ∪ (t, v)
24             P ← P \ (t, v)
25             bw_b ← bw_b + ev_t
26 return B
```

related to t_2 are on the same VM avoiding replication of publications related to topic t_2 (cost is amortized once per VM), hence the total bandwidth consumption is down to 120 KB/min. However, the pairs related to t_1 are still

on different VMs. We can improve this further by selecting the topic with maximum event rate first and the VM with most free capacity first. These optimizations give priority to the allocation of pairs of topics with maximum event rate, which have the most overhead when split among different VMs, to the VMs with most free capacity. In Figure 7.3d we can see that by applying these optimizations we allocate each topic and its subscribers on a minimal number of VMs, thereby reducing the incoming bandwidth consumption to 100 KB/min instead of 130 KB/min using *FFBP*.

The pseudocode for the solution for Stage 2 **CustomBinPacking** (*CBP*) with the optimizations mentioned above is presented in Algorithm 11. We consider topics and their associated subscriber pairs in the *non-increasing order* of their *event rates* for the purpose of allocation (Line 3). We then *group* the topic-subscriber pairs of the same topic together (From Line 4 to Line 7). Next we *compare the cost* of distributing among existing VMs to cost of deploying new VMs and choose the most cost-effective option (Line 8). The comparison of costs is done in **CheaperToDistribute** (Algorithm 12). The **CheaperToDistribute** algorithm is a heuristic to decide on distributing the pairs of the current topic in question to already deployed VMs or to allocate them to a new VM. This algorithm is used when the pairs of the current topic in question cannot be allocated to the current VM. In Algorithm 12, we first compute the estimated total cost when deploying new VMs and allocating to them (between Lines 2 and 4). Then we iteratively compute the cost of allocating to a VM with maximum available capacity, until there are no more pairs left in P (between Lines 5 and 14) or none of the existing VMs have enough capacity left to accommodate even a single pair. It is possible that some pairs can be left unallocated to the existing VMs in which case new VMs need to be deployed. The cost of the extra VMs needed and corresponding bandwidth consumption is computed between Lines 15 and 17. Finally Algorithm 12 returns **true** if allocating to existing VMs is cheaper and returns **false** otherwise (in Lines 18 and 19).

Each of the above optimizations gives an incremental improvement to our solution in practice. We explore the impact of each optimization with Spotify and

Algorithm 12: Computes the cost of distributing current topic to existing VMs

```
1  CheaperToDistribute(t, B, BC, P, C_1, C_2)
   Input: t, B, BC, P, C_1, C_2
   Data: curbw ← Σ_{b∈B} bw_b: Current bandwidth consumption
   curvms ← |B|: Number of VMs currently in use
   extravms ← 0: Extra VMs needed if existing VMs used
   extrabw ← 0: Extra bandwidth requirement
   newvmsbw ← 0: Bandwidth needed if new VMs are used
   newvms ← 0: Number of new VMs needed for allocation
   TV ← ∅: Temporary set of VMs
   Result: distribute ← false
   // Estimate the cost of deploying on new VMs
2  if P ≠ ∅ then
3      newvms ← ⌈(|P| · ev_t)/BC⌉
4      newvmsbw ← (|P| + newvms) · ev_t
   // Estimate the cost of distributing to existing VMs
5  while P ≠ ∅ and B \ TV ≠ ∅ do
6      b ← argmax_{b'∈B\TV} {BC − bw_b'}
7      if 2 · ev_t ≤ BC − bw_b then
8          newbw ← ev_t
9          while P ≠ ∅ and newbw ≤ BC − bw_b do
10             (t, v) ← any random (t', v'), such that (t', v') ∈ P
11             newbw ← newbw + ev_t
12             P ← P \ (t, v)
13         extrabw ← extrabw + newbw
14         TV ← TV ∪ b
15 if P ≠ ∅ then
16     extravms ← ⌈(|P| · ev_t)/BC⌉
17     extrabw ← extrabw + (|P| + extravms) · ev_t
18 if C_1(curvms + extravms) + C_2(curbw + extrabw) <
   C_1(curvms + newvms) + C_2(curbw + newvmsbw) then
19     distribute ←true
20 return distribute
```

Twitter traces in Section 7.3.4.

Algorithm 13: Lower bound for ***MCSS***

1 **GetLowerBound**$(T, V, ev, Int, \tau, BC, \mathcal{C}_1, \mathcal{C}_2)$
  Input: $T, V, ev, Int, \tau, BC, \mathcal{C}_1, \mathcal{C}_2$
  Data: $bwcostlb \leftarrow 0$: Lower bound on the cost to satisfy all subscribers
2 **foreach** $\{v \in V\}$ **do**
3 $\tau_v \leftarrow min(\tau, \sum_{t \in T_v} ev_t)$
4 $bwcostlb \leftarrow bwcostlb + \max(\tau_v, \min_{t \in T_v} ev_t)$
5 **return** $C_1(\lceil bwcostlb/BC \rceil) + C_2(bwcostlb)$

7.2.3 Lower Bound

Combining the solutions for both stages *GSP* from Algorithm 9 and *CBP* from Algorithm 11, gives us a complete solution for *MCSS*. While dividing the solution into two stages makes it simpler to solve, it renders our solution sub-optimal. By separately considering the selection of topic-subscriber pairs and their allocation to VMs, we lose an opportunity to make a better allocation of the pairs to the VMs. However, in Section 7.3 we show that our approach works well in practice.

Deriving theoretical bounds on our solution is difficult because of various optimizations we introduce and we omit it. However, using Theorem 7.3 for a given data input we can estimate a lower bound on the objective of *MCSS*.

Theorem 7.3. *Given an instance* MCSS*$(T, V, ev, Int, \tau, BC, \mathcal{C}_1, \mathcal{C}_2)$, for any solution $\mathcal{B}$ it holds that:*

$$\mathcal{C}_1(|\mathcal{B}|) + \mathcal{C}_2\left(\sum_{b \in \mathcal{B}} bw_b\right) \geq \mathcal{C}_1\left(\left\lceil \frac{\sum_{v \in V} \max\left(\tau_v, \min_{t \in T_v} ev_t\right)}{BC} \right\rceil\right)$$

$$+ \mathcal{C}_2\left(\sum_{v \in V} \max\left(\tau_v, \min_{t \in T_v} ev_t\right)\right)$$

Proof. The goal here is to derive a lower bound on the total cost of the allocation. Consider the capacity that must be spent to add a user to the solution set. A subscriber v can be satisfied when topics with total event rate of τ_v are selected

in the solution. Hence, the minimum capacity that must be spent to satisfy a subscriber is τ_v. To tighten this bound slightly, we also observe that if $\forall_{t \in T_v} ev_t \geq \tau_v$, then the semantics of the *MCSS* definition dictates that we must choose at the granularity of topic-subscriber pairs. Hence, the capacity that must be spent in such a scenario is $\min_{t \in T_v} ev_t$. Hence, we derive the clause $\max(\tau_v, \min_{t \in T_v} ev_t)$ as a cost to satisfy a single subscriber. So summing up these bounds, we get the lower bound on the outgoing bandwidth consumption to satisfy all subscribers.

Now, to derive a bound on the number of VMs, we simply divide the total bandwidth consumption by the bandwidth capacity of the individual VM *BC* and round it up.

□

Theorem 7.3 can be easily turned into an algorithm to derive the lower bound and the pseudocode is presented in Algorithm 13. For each subscriber we select the bare minimum bandwidth cost required to satisfy the subscriber (Line 2 to Line 4). Then we derive the lower bound on the number of VMs by dividing the lower bound on bandwidth consumption by bandwidth capacity *BC* per VM and using cost functions we derive the lower bound on total cost in Line 5. In Section Section 7.3.4 we evaluate *GSP* with *CBP* and *RSP* with *FFBP* and compare them against the lower bound obtained using Algorithm 13.

7.3 Experimental Evaluation

The goal of the experimental evaluation is to study the effectiveness of the proposed solution in minimizing the total cost of deploying pub/sub for social interaction in systems like Spotify and Twitter on a public cloud service. In this section, we evaluate our solution by considering each stage of the solution incrementally. We repeat all our experiments for Spotify as well as Twitter traces with various practical settings.

7.3.1 Experimental Setup

We implemented all algorithms presented in this chapter using C++. All experiments were executed on a server with Intel Xeon 1.87GHz processors and 132 GB of RAM. We executed experiments with τ varying from 10 to 1000. For the cost function we followed the Amazon EC2 cost-model[1]. We used the pricing for *On-Demand Instances* with *Compute Optimized - Current Generation*. For our experiments, we considered the pricing for 2 types of VM instances c3.large (costs $0.15 per hour) and c3.xlarge (costs $0.3 per hour), these instance types are our choice for evaluation because they have specified bandwidth limits[2]. We set c3.large and c3.xlarge with bandwidth capacities of 64 mbps and 128 mbps respectively derived from Amazon specified bandwidth limits. Even though we repeated our experiments using other instance types, we omit their results because they provide no significant new information. For the bandwidth cost we use $0.12 per GB for both incoming as well as outgoing bandwidth taken from data transfer costs of Amazon EC2 pricing model (subject to change).

Bandwidth consumption is measured in bytes per unit of time; hence, we need to convert the event rates in our model to bytes. We know that each tweet has a maximum length of 140 characters. However, from the information given in [Krikorian, 2010], the mean size of a tweet is 200 bytes; thus, in our experiments we set the message size of a twitter publication as 200 bytes as well. For the Spotify case, after measuring the mean message size of a sample of messages from Spotify traces we found it to be 111 bytes. But we set the message size as 200 bytes to make the comparison with Twitter traces easier.

7.3.2 Data Traces

Spotify Traces The trace consists of about 1.1 million topics and 4.9 million subscribers forming about 12 million topic-subscriber pairs. The traces were gathered for 10 days (9th Jan 2013 to 19th Jan 2013) from Spotify's datacenter at Stockholm (one of the three datacenters). The events we collected were

[1] http://aws.amazon.com/ec2/pricing
[2] https://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf

restricted to the music playback events from users with at least one follower. For more information about the Spotify trace, and its detailed analysis see Section 5.1.

Twitter Traces We use the publicly available Twitter social graph well studied in [Kwak et al., 2010]. We model the Twitter users as topics and their followers as subscribers. The subscriptions (subscribed topics) of a subscriber is the followings of a user (the list of Twitter users followed by the user). The number of tweets published by a particular user t corresponds to the event rate ev_t for a given period of time. We used the public Twitter APIs to obtain the number of Tweets of each user in the data set by [Kwak et al., 2010] from 30th Oct 2013 to 9th Nov 2013. We consider all the Twitter users who tweeted at least once during those 10 days (active users) and omit the rest. This process provided us with around 8 million active users and their corresponding 30 million subscribers, and around 683.5 million topic-subscriber pairs. For a detailed analysis of Twitter traces refer to Section 5.2.

7.3.3 Comparison of Approaches for Stage 1

We first explore the impact of using *GreedySelectPairs* (*GSP*) presented in Algorithm 9 with *RandomSelectPairs* (*RSP*) as a baseline on the total cost with **FFBinPacking** as Stage 2 solution for both. We run experiments with c3.large and c3.xlarge VM cost functions. From Section 7.2 we know that, unlike *RSP*, *GSP* selects topic-subscriber pairs to satisfy all the subscribers while trying to minimize the bandwidth requirement. This helps in reducing both the number of VMs and bandwidth consumption and hence the total cost. Figure 7.4(A) shows the impact of *GSP* using Spotify traces and c3.large. With $\tau = 10$ it results in a 33% reduction in the number of VMs, 22.9% bandwidth reduction and a 33% reduction in total cost. However, as τ increases to 100 and 1000, the cost reduction drops to 27.6% and 10.9% respectively. The reason for the drop in cost reduction is that higher values of τ leave little for optimization, since a higher fraction of all topic-subscriber pairs are needed to satisfy the problem constraints. A similar pattern is observed in Figure 7.4(B) for VM type c3.xlarge

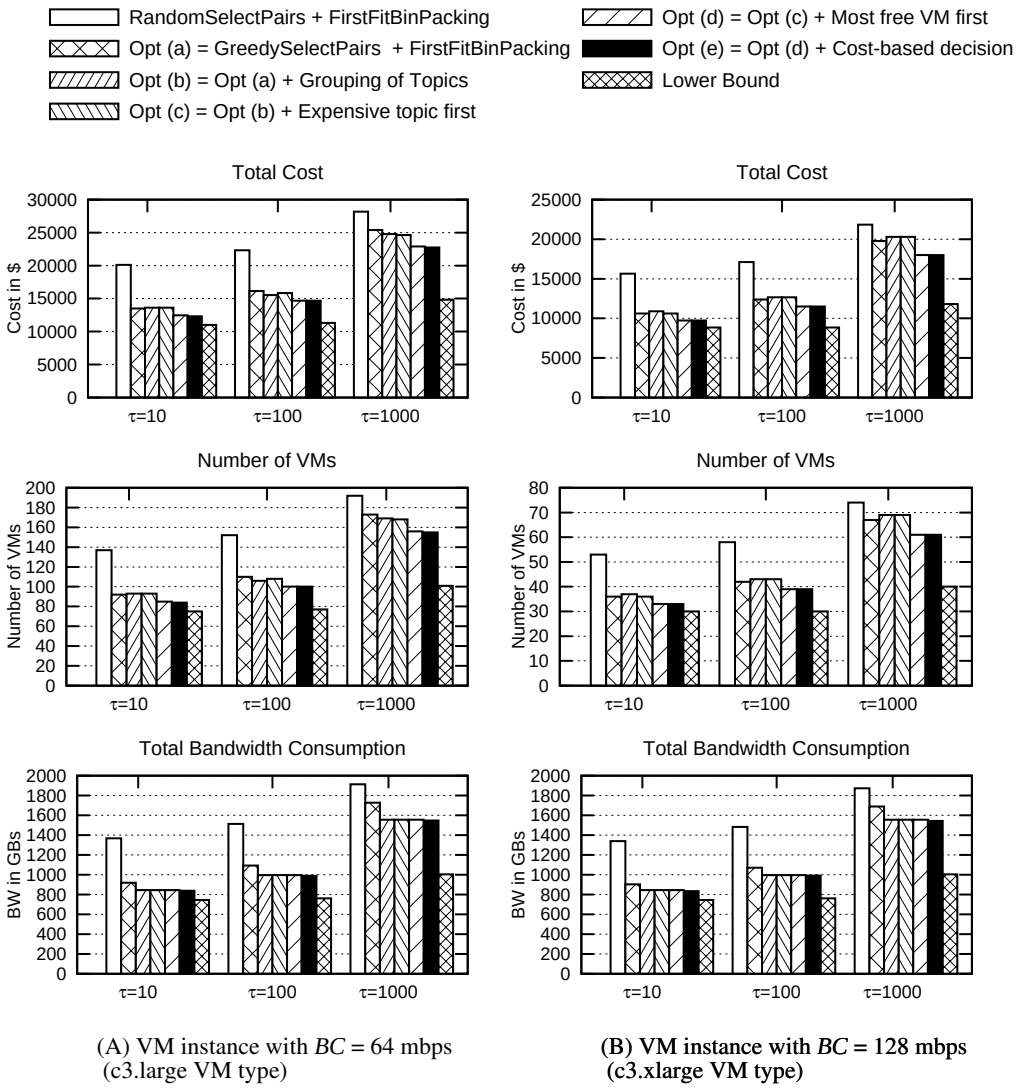

Figure 7.4: Impact of introducing optimizations (a) *to* (e) with Spotify traces

with $BC = 128$ mbps. A 32.7% reduction with $\tau = 10$ and 17.6% and 10.8% reduction with $\tau = 100$ and $\tau = 1000$ respectively.

Now we study the impact of *GSP* with Twitter traces. As seen in Figure 7.5(A), the cost reduction is significantly higher compared to Spotify traces. With $\tau = 10$ there is a reduction of 71% and 51.4% with $\tau = 100$. However, with $\tau = 1000$ the reduction is only 29.1%, suggesting that as τ increases, the room for

minimizing costs also decreases. We observe the same pattern in Figure 7.5(B) as well with $BC = 128$ mbps. The improvements are 70%, 51.9% and 20.3% for $\tau = 10, 100, 1000$ respectively.

7.3.4 Comparison of Approaches for Stage 2

In Stage 2 of our solution, the goal is to allocate the topic-subscriber pairs from Stage 1 to VMs so as to minimize the cost. In this section we explore the impact of various optimizations introduced in Section 7.2.2 for Stage 2 of our solution on the total cost. To analyze the effectiveness of these optimizations, we fix the approach for Stage 1 as *GSP* for the rest of the experiments unless mentioned explicitly. By incrementally introducing the optimizations we study their incremental impact in the following order: *(a)* with only **FFBinPacking** (*FFBP*), *(b)* introducing *grouping of pairs* by topics, *(c)* introducing *most expensive topic first*, *(d)* introducing *most free VM first*, *(e)* introducing choice of allocation *based on cost-model*. In Figures 7.4 and 7.5 the bar plots contain the corresponding bars to represent the improvement in total cost, number of VMs and bandwidth consumption respectively, in the same order of the optimizations listed above. Finally, we also compare the impact of including all these optimizations with the lower bound obtained by running the Algorithm 3.

We start with *optimization (a)*, *FFBP* presented in Algorithm 10. In Figure 7.4(A) and Figure 7.4(B) the outcome of *FFBP* when used in conjunction with *GSP* topic-subscriber pair selection technique can be seen for different values of τ and for c3.large and c3.xlarge VM types. However, as mentioned in Section 7.2.2 since *FFBP* considers the pairs to be allocated to VMs in arbitrary order and at individual pair level, there is room for improvement. Hence, we introduced *optimization (b)*, (presented in Algorithm 11 **CustomBinPacking** (*CBP*)) the *grouping of pairs* belonging to the same topic and analyze its effectiveness. The grouping of pairs optimization results in a cost reduction of about 3.5% for Spotify traces in most cases. However, in some cases we see an increase in cost up to 1.6%. This behavior is because of the trade-off between the number of VMs and the total bandwidth consumption. For example, in

Figure 7.4(A) for $\tau = 10$ and in Figure 7.4(B) for all values of τ, it can be noticed that, even though there is a decrease in bandwidth consumption of about 8 to 10%, the corresponding number of VMs increase by 2 to 4%. The increase in total cost in some cases suggests that grouping of topics alone is not always beneficial. This behavior is due to the fact that the grouping of pairs optimization is aimed at minimizing bandwidth consumption. As explained in Sections 7.1 and 7.2, because of the trade-off between the number of VMs and bandwidth consumption, we see an increase in total cost. As we show later in the experiments, this optimization has an impact in conjunction with other optimizations.

For Twitter traces, we can observe a behavior similar as that seen in Figure 7.5(A) and Figure 7.5(B). In all cases there is a slight decrease in cost due to the grouping of topics, even though in some cases there is an increase in the number of VMs. This can be clearly observed with $\tau = 1000$ and $BC = 128$ mbps, in Figure 7.5(B). In this case there is a decrease in bandwidth consumption of 8% which results in increase of 0.5% in VMs (one VM). However, the total cost still decreases because the decrease in bandwidth consumption overshadows increase in number of VMs. This behavior is again attributed to the trade-off between the two metrics.

Next we study the impact of introducing *optimization (c)*, the *ordering of topics* in decreasing order of event rates and selecting the topics and their pairs with maximum event rate for allocation first. As explained in Section 7.2.2, the rationale behind this optimization is to give priority to expensive topics to avoid pairs belonging to the same expensive topic being allocated to different VMs. This optimization can result in an increased number of VMs with a slight decrease in bandwidth consumption in some cases, as in Figure 7.4(A) for $\tau = 100$. However, in most cases it results in a decrease in the total cost up to 2.5%. For Twitter traces, in Figure 7.5(A) and Figure 7.5(B) we can notice a slight decrease in total cost up to 2.4%. It is worth noting that, even though this optimization does not show many benefits on its own, we next show that it works well together with selecting VMs with most available capacity first.

As done in Algorithm 11, we try to allocate all the pairs of a topic to the

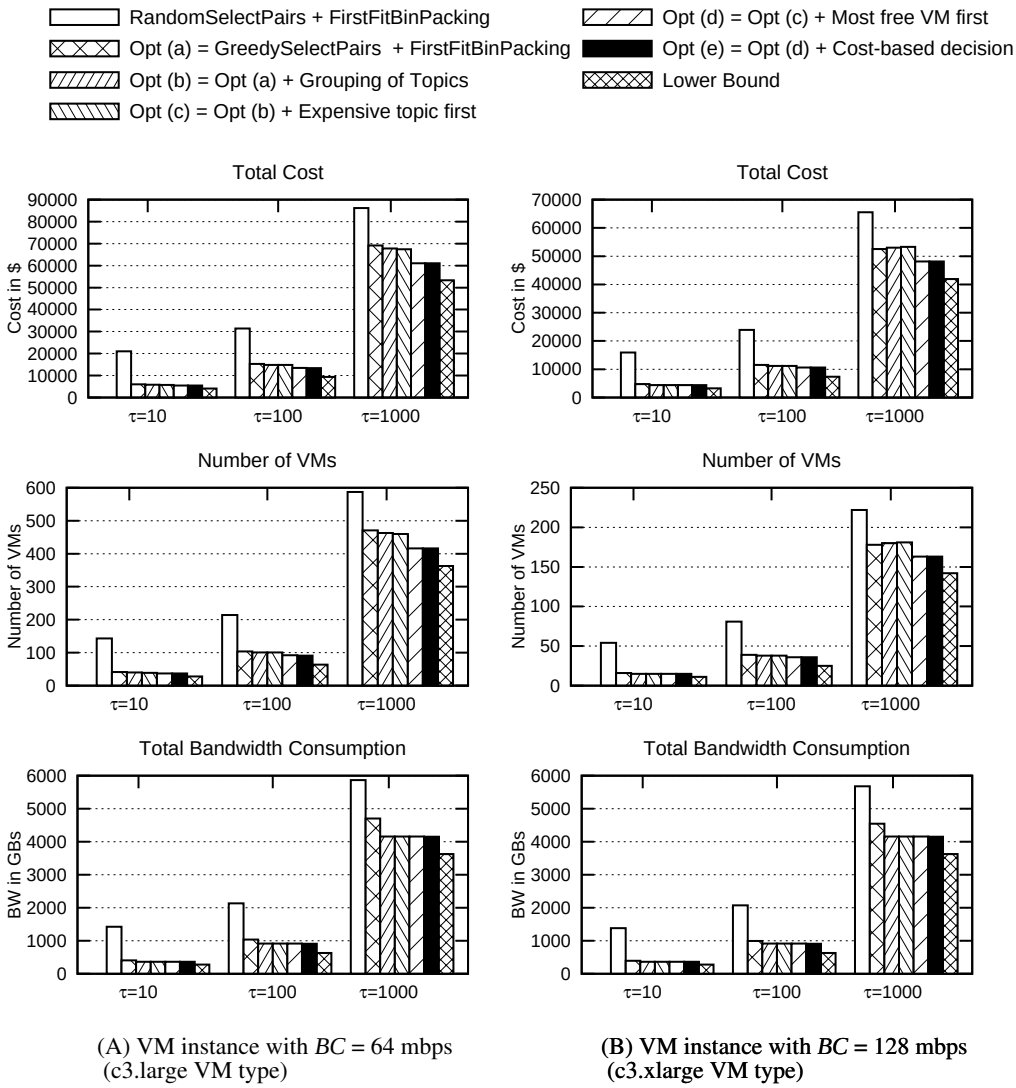

Figure 7.5: Impact of introducing optimizations (a) *to* (e) with Twitter traces

most recently deployed VM. If that is not feasible, we try to allocate them to existing VMs. Now we analyze the impact of introducing *optimization (d)* in which we choose the VMs with *most free capacity* first while allocating the pairs among already deployed VMs. For both Spotify and Twitter traces, we observe a reduction in cost with this optimization. The reduction in number of VMs is the main contributor for reduction in cost with this optimization. In most cases

bandwidth consumption remains the same or even slightly increases again due to the trade-off with the number of VMs. For Spotify traces there is a decrease in cost of up to 10.7% and for Twitter traces the decrease is up to 9.5%. An interesting observation here is that the decrease in cost is slightly higher for $\tau = 100$ and 1000 than $\tau = 10$. It is worth noting that the improvement we see from this optimization is also the result of optimizations *(b)* and *(c)*.

Finally, we introduce *optimization (e)*, the decision to allocate to existing VMs at the cost of extra bandwidth consumption against deploying new VMs *based on the cost-model* presented in Algorithm 12. The decision to deploy a new VM instead of existing VMs is done if it results in decreased total cost. This optimization is supposed to balance the trade-off between the number of VMs and bandwidth consumption. However, we observe lower cost reduction than expected. For Spotify, the maximum cost reduction is 1.2% and for Twitter it is 0.2%. The reason for this behavior is that, in our cost-model, the bandwidth per GB is only \$0.12. Thus, the bandwidth is significantly inexpensive. For example, for a topic t with ev_t 10000 events/day (2 MB/day), even if all the subscriber pairs of t are spread across 100 different VMs the bandwidth overhead is 200 MB and costs only \$0.024. With such a low overhead the cost-model hardly makes a difference. In addition to that, the cost-model is suboptimal since it takes the decision for each topic independently. Hence, the overhead of extra bandwidth due to distributing the pairs of a topic is generally significantly lower than deploying the new VMs. We leave further exploration of this optimization for future work.

7.3.5 Runtime Performance Evaluation

In this section, we show the runtime performance of our approaches. The faster runtime performance of the VM allocation approaches on cloud are crucial, since the allocation may be required to run periodically to adapt to the workload. We first analyze the running times of solutions for Stage 1. It is clear that selecting an arbitrary set of pairs (*RSP*) is faster than selecting pairs according to the greedy heuristic (*GSP*). However, in Figure 7.6 we can see that the runtime of *GSP* for

Stage 1 with Spotify traces is only at most two seconds slower than *GSP* in all cases. Increasing τ requires more topic-subscriber pairs to be selected. The near-constant time for *GSP* suggests that our approach is scalable with τ. In Figure 7.7 we can see a similar pattern for Twitter traces. However, since the Twitter trace has a much higher number of pairs (638.5 million), it results in significantly higher runtime for both *RSP* and *GSP* compared to Spotify traces. *RSP* takes up to 986 seconds, on the other hand *GSP* takes up to 1471 seconds. The slower running time of *GSP* is because it inspects all the 638.5 million pairs at least once to select the best pairs according to the heuristic. On the other hand, *RSP* selects the first subset of pairs meeting the satisfaction threshold and returns pairs which result in significantly higher cost. This is a clear trade-off between quality of output and running time.

Figure 7.6: Stage 1 Runtime for Spotify traces

Next we analyze the runtime performance of *FFBP* and **CustomBinPacking** (*CBP*) solutions for Stage 2. We restrict our comparison between running times for *optimization (a)* and the solution in Algorithm 11 including all other optimizations (*optimization (a)* to *(e)*) and assuming input from *GSP* readily available in main memory. From Figures 7.8 and 7.9 we can see that **CustomBinPacking** (*CBP*) outperforms *FFBP* up to 10 times better with Spotify traces and around 1000 times with Twitter traces. The fast runtime of *CBP* is attributed to the optimization related to grouping of pairs on a per

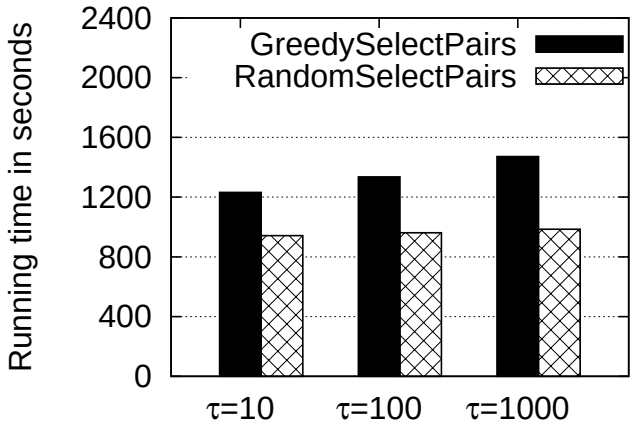

Figure 7.7: Stage 1 Runtime for Twitter traces

topic basis to allocate them to VMs ($\mathrm{O}\big(|T||\mathcal{B}|\big)$). On the other hand, *FFBP* considers the VMs in the order of first-fit, hence in the worst-case it may have to check the feasibility to allocate with all the deployed VMs ($\mathrm{O}\big(|T||V||\mathcal{B}|\big)$). It is worth noting that even though *GSP* is slower than *RSP* on its own, in combination with *CBP* the overall runtime performance is better than *RSP* in combination with *FFBP* in most cases. For example, *GSP* with *CBP* takes 1484.7 seconds in total compared to 2186 seconds taken by *RSP* with *FFBP* for Twitter traces with $\tau = 1000$ on a c3.large instance.

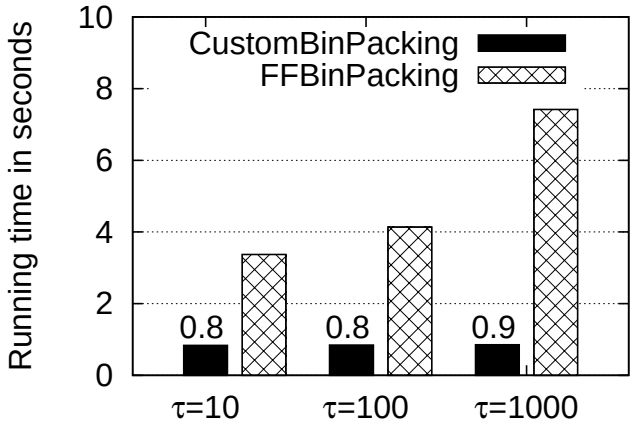

Figure 7.8: Stage 2 Runtime for Spotify for c3.large

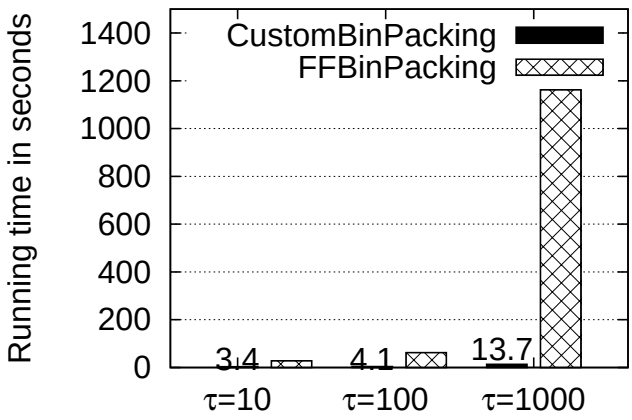

Figure 7.9: Stage 2 Runtime for Twitter for c3.large

7.3.6 Summary and Discussion

In this section, we empirically evaluate our solution by considering the isolated impact of each stage and each optimization. We compare the performance of *GSP* and *RSP* while using *FFBP* as a solution for stage 2. In summary, *GSP* provides an improvement in the total cost of up to 33% for the Spotify and 71% for the Twitter traces. Subsequently, we fix *GSP* as the solution for Stage 1 and analyze the incremental impact of individual optimizations (*(b)* to *(e)*) introduced for Stage 2. Even though each optimization is improving the cost in only a subset of cases, we observe a cumulative improvement of up to 5%. With a combination of *GSP* and *CBP* we attain a total saving of up to 74% for the Twitter traces and 38% for the Spotify traces. In absolute values, this translates into $4000 and $2000 for the Twitter and Spotify traces respectively. Note that these savings are for sampled traces (about 10% sample for Spotify and 1% sample for Twitter) for a 10 day period. We can expect higher savings for a longer period and full traces.

The runtime for the Spotify traces on a moderately powerful server is under 30 seconds for our complete solution, suggesting that it is fast and it can be run periodically to re-allocate the workload. For example, it can be run every hour to adapt to the changes in the event rates, new subscriptions, unsubscriptions, etc. However, for the Twitter traces it runs relatively slower (about 25 minutes)

because of the larger scale. Even though our solution can be run at longer periods (e.g., once per day), it is desirable to adapt in a dynamic and online fashion. In some works such as [Cerviño et al., 2012] dynamic approaches are suggested for adaptive provisioning. However, in order to solve our problem there is a need to take into account additional factors such as the effects of dynamic workload on the user satisfaction metric. We plan to tackle the challenge of devising an online algorithm as part of future work.

7.4 Summary

In this chapter, we have proposed a new approach for resource provisioning for pub/sub in the cloud using a cost-effective resource allocation. The approach is directed towards a particular class of pub/sub that is used to drive social interaction, e.g., among Spotify and Twitter users. To formalize the challenge of cost-effective resource allocation, we have introduced the *MCSS* problem and established its hardness by a reduction from the well-known partitioning problem. We have provided an efficient heuristic for *MCSS* consisting of a number of optimizations. Our approach can be used as a tool by pub/sub architects to estimate and provision resources to satisfy all subscribers in a datacenter or in a cloud. We have evaluated the proposed heuristic solution empirically using large-scale real traces from Spotify and Twitter. Using an Amazon EC2 pricing model, we have showed that our solution can save up to 74% and up to 38% of the total cost for Twitter and Spotify respectively when compared to a naive alternative. We have also provided a comparison against a derived lower bound and showed that in many cases our approach results in a cost that is only 15% higher.

Finally, our approach has a reasonably low computation time, as corroborated by the experiments. Hence, it can also be used for dynamic allocation if run at periodic intervals to re-provision the resources and re-allocate to the workload. In the future, we plan to extend this work to fully support dynamic on-demand provisioning and allocation for pub/sub.

Chapter 8

Conclusions and Future Work

Designing pub/sub for large-scale social interaction is inherently a difficult task. There are several challenges involved in scaling a pub/sub system to millions of subscribers and billions of publications. In this thesis, we identified a number of critical challenges not yet addressed in the literature. We transformed those challenges into several research problems and provided efficient solutions. We also validated our ideas under realistic scenarios using large-scale traces from real-world systems. In this chapter we list the most significant observations and conclusions of this thesis. Finally, we present an overview of possible future directions for the research ideas introduced in this thesis.

8.1 Summary of The Results

8.1.1 PolderCast

PolderCast was designed with the aim of harmonizing several conflicting and yet desirable characteristics of a P2P TBPS system. In Chapter 3, we proposed a layered architecture that relies on gossiping techniques to build and maintain a pub/sub overlay. We also proposed a hybrid dissemination algorithm using a combination of neighbor and random links in the ring for robust and faster dissemination of publication events.

We evaluated PolderCast using the Twitter and Facebook traces. The

experiments corroborate that PolderCast swiftly constructs a topic-connected overlay, which is scalable, robust and promotes fast dissemination of events with minimal overlay maintenance overhead when compared to Scribe. This is achieved by a unique amalgamation of per topic dissemination over a hybrid dissemination structure consisting of a maintained ring per topic and carefully chosen random shortcut links provided by the underlying Peer Sampling Service [Jelasity et al., 2009]. The hybrid dissemination structure is maintained over a gossiping architecture with three layers:

Rings: protocol at the top layer responsible for maintaining rings

Vicinity: protocol at the middle layer collecting nodes with similar interests for two important purposes: (a) feeding the Rings layer to swiftly build and maintain the induced ring per topic (b) acting as a pool for random shortcuts between the nodes with similar interests

Cyclon: at the bottom layer keeping the overlay connected and providing purely random shortcuts.

Along with the robust overlay structure, the regular gossiping at each layer is responsible for repairing any damages or changes to the overlay ensuring a reliable dissemination service.

With traces from Twitter and Facebook we showed that PolderCast quickly constructs a topic-connected overlay for every topic in the system. We also demonstrated that overlay construction and maintenance is scalable with the number of nodes per topic and topics per node. We then illustrated that PolderCast facilitates event dissemination with a balance between delay and the number of duplicate messages. An extensive analysis of different choices for random shortcut links for achieving the desired balance between dissemination delay and amount of duplicate messages was conducted. Finally, we tested PolderCast performance under churn using Skype super-peer churn traces [Guha and Daswani, 2005]. With Scribe as a baseline, we showed that PolderCast provides a more robust performance in these experiments.

8.1.2 Study of Real-World Pub/Sub Systems and Workloads

Case Study of Spotify Pub/Sub Spotify is a popular music streaming service. Apart from its large user base and large music catalog, the traffic it generates due to social interaction is also large-scale. Use of pub/sub to drive such a large-scale notification system is of special interest to the pub/sub research community.

In Section 4.2.1, we presented the architecture of Spotify pub/sub that allows users to follow their friends, public playlists, their favorite artists etc. The architecture facilitates both real-time and offline delivery of notifications due to music activities of their friends and artists and updates to the playlists they follow. At the core of the pub/sub system is a DHT-like ring overlay of pub/sub brokers. The overlay is responsible for matching and delivering publication events. The events are also persisted in a Cassandra [Lakshman and Malik, 2010] cluster for offline delivery to the users who are not online during the real-time notification delivery.

Pub/Sub Workload Analysis It is crucial to gain insights into pub/sub workloads for design of effective pub/sub systems. In this regard, we did a detailed analysis of large-scale pub/sub workloads from the Spotify and Twitter social interaction systems in Chapter 5. The workload from Spotify consisted of 1.1 million topics and 4.9 million subscribers. On the other hand, the Twitter traces were larger than the Spotify traces with 8 million topics and 30 million subscribers.

Here we list a summary of important observations we gathered from the analysis of the Spotify pub/sub workload:

- Topic popularity and subscription sizes follow a distribution close to a power law, similar to degree distributions in social graphs.
- Publication event rate does not follow a power law distribution.
- The normalized notification rate per subscriber is very low (max 1%) and varies from 1% to as low as 10^{-7}%, indicating subscriptions with diverse values.

- The number of events received by a subscriber in a given unit of time (normalized notification rate per subscriber) is linearly proportional to the number of topics subscribed to by that subscriber.
- The publication event rate of a topic bears no relation to its popularity. We conjecture that this is due to music activity not being the determinant factor in social relations.
- Publication traffic shows a daily pattern. It is lowest at 2 AM and highest around 6 PM. It also shows a weekly pattern with slightly lower traffic during weekends.
- Publication traffic from local sites is much higher compared to publication traffic from remote sites.
- Subscription and unsubscription rates imply significant churn in subscriptions. However, the total number of subscriptions does not change significantly in a 10-day period.
- Both subscription and publication traffics are dominated by the traffic generated from the real-time notifications due to music listening activity.

For Twitter traces, we were able to obtain only the data from public Twitter APIs. As a result of this the data we obtained limited us to do the analysis of publication event rates, topic popularity, subscription sizes and normalized notification rate per subscriber. Unfortunately, we could not conduct the analysis of temporal patterns in the Twitter traffic.

Here we list the important observations from the analysis of Twitter traces:

- Topic popularity and subscription sizes follow a power law distribution except for a couple of anomalies in subscription sizes. In particular, subscription sizes at 20 and 2000 respectively showed anomalies. This is due to the restrictions Twitter imposed until 2009 in the maximum number of followings a user can have (2000) and the default number of recommendations to follow (20).

- An interesting observation stemming from the distribution of subscription sizes is that, around 550 users follow more than 10000 users which is a large subscription size for a TBPS system.

- We observed an anomaly in the distribution of the number of followers at 10^5. We verified that this was due to celebrities having significantly higher number of followers than normal users. Around 66 users in our traces had more than 1 million followers.

- Looking at the dependency of the number of followers and publication event rate, we observe that it is a linear dependency up to a certain point. The linearity breaks due to celebrities since even though they have a significantly higher number of followers, they tend to tweet less often.

- Studying the correlation between subscription sizes and the normalized notification rate per subscriber, we observe that it is a linear dependency except that we see the same glitches observed in the distribution of subscription sizes.

The results obtained above characterize a real pub/sub workload from two large-scale social interaction systems. These results come handy for pub/sub designers in building pub/sub systems. In addition, these results are useful for generating synthetic workload that can be used to evaluate pub/sub systems. The Twitter traces are available for the public[1].

8.1.3 Subscriber Satisfaction Problems

Inspired by the social interaction at Spotify and observing in Chapter 6 that social notifications mostly have human recipients, we proposed subscriber satisfaction metrics. To the best of our knowledge we are the first to propose subscriber satisfaction metrics with threshold limits on the number of events a subscriber can receive. We introduced two flavors of the satisfaction metrics: (1) a binary satisfaction metric in which the subscriber is considered satisfied only if

[1] Can be downloaded here: http://tidal-news.org/data/icdcs14/tweetrates.tgz

100% of the satisfaction threshold is met (2) a fractional satisfaction metric in which a fraction of satisfaction threshold (up to 100%) is met. We proposed new approaches to maximize the satisfaction metrics. These two metrics resulted in two different problems. We coined those two problems as Budgeted Maximum Multiset Multicover (*B3M*) and Fractional Budgeted Maximum Multiset Multicover (*F-B3M*).

Here is the list of most significant results and conclusions we obtained by analyzing and solving *B3M* and *F-B3M* problems:

- We proved that B3M is NP-Hard by reducing the Densest-k-Subgraph (*DkS*) problem to it. Unlike existing reductions that transform the Max Cover problem to a variety of covering problems, we introduce a novel reduction from the *DkS* problem to the *B3M* problem.
- Reducing from the *DkS* problem also helped us to prove that there is no PTAS for B3M assuming NP $\not\subseteq \cap_{\epsilon>0}\text{BPTIME}(2^{n^{\epsilon}})$.
- We provided a greedy heuristic solution for the B3M problem that works in practice.
- We proved that F-B3M is also NP-Hard by reducing the Budgeted Max Cover problem to it.
- We provided a greedy algorithm with constant approximation guarantee for the F-B3M problem.
- By exploiting the submodularity property of the objective function of the F-B3M problem we optimized the performance of the greedy algorithm for the F-B3M problem.
- We derived a loose upper bound for both B3M and F-B3M problems to compare the result of our solutions to the result obtained by the upper bound.
- We validated the solutions by evaluating them with the large-scale traces from Spotify.

- We showed that our solutions for B3M can meet practical values of satisfaction metrics for up to 72% of all the users, with infrastructure capacity limited to 10% of what is required for delivering all publication events. Under the same settings, a cumulative satisfaction value of over 3.5 million was achieved for F-B3M, compared to 4.2 million achieved by the derived upper bound (ratio of 0.86 to the upper bound).

- We also showed that our algorithms run in under 30 seconds for 1.1 million topics and 4.9 subscribers and under 1 second for smaller workloads including traces from 1 hour time windows.

The results above indicate that there is both theoretical and practical potential for pub/sub systems to benefit from our solutions. In particular, our solutions provide a way to efficiently select part of the workload so as to maximize subscriber satisfaction under resource constraints and offload the rest to P2P networks.

8.1.4 Resource Provisioning for Pub/Sub

In Chapter 7, we proposed a cost-effective resource allocation technique for large-scale TBPS systems. Our solution is useful for resource allocation both in public clouds and in-house datacenter infrastructures. For datacenter settings, our solution minimizes the number of servers and amount of bandwidth required to meet the satisfaction metrics of all the subscribers. For cloud settings, our solution minimizes the monetary cost of deploying pub/sub for a given pricing model. We introduced a novel problem coined Minimum Cost Subscriber Satisfaction (*MCSS*) problem to formalize the challenges of cost-effective resource allocation for pub/sub.

Important observations and results from the analysis of the *MCSS* problem and evaluation of its solution are listed below:

- By reducing the well-known Partitioning Problem [Garey and Johnson, 1979] to our *MCSS* problem we proved that it is NP-Hard .

- We provided a two stage greedy heuristic for the *MCSS* problem.

- We proposed a customized bin packing solution with several optimizations to improve the cost-effectiveness of resource allocation.

- By using two large-scale real-world traces from Spotify and Twitter, we validated and extensively evaluated our solution.

- In order to show the effectiveness of our allocation technique, we compared our solution consisting of customized bin packing with a naive baseline consisting of the first-fit bin packing algorithm. We selected this baseline because many existing resource allocation techniques rely on the first-fit bin packing algorithm.

- We also derived a lower bound that the objective function of *MCSS* can achieve. The lower bound we derived is not necessarily tight and provides a minimum cost that any allocation strategy could achieve. Then we compared our solution to the lower bound to measure the gap between the theoretical minimum and our result.

- We showed that our solution saves server and bandwidth costs up to 72% for the Twitter traces and 37% for the Spotify traces compared to the baseline solution. On the other hand, our solution incurs only up to 15% higher cost compared to the derived loose lower bound.

- We demonstrated that our resource allocation algorithm is fast. In particular, it runs under 1 second for the Spotify traces consisting of 1.1 million topics and 4.9 million subscribers. In addition, for the Twitter traces consisting of 8 million topics and 30 million subscribers it runs under 10 seconds.

With the results from the large-scale evaluation of our solution, we conclude that our solution provides a method to minimize the costs incurred in deploying a TBPS system that scales to millions of subscribers.

8.2 Lessons Learned

In the process of designing scalable pub/sub solutions for social interaction we learned several lessons during this thesis. In this section we highlight the most significant of them.

8.2.1 Workload Analysis

It is critical to analyze real pub/sub workloads for designing effective pub/sub systems.

The workload analysis we did in Chapter 5 provided us many insights with respect to potential issues that a pub/sub system may face. These insights also inspired us to propose novel problems aimed for designing effective pub/sub systems.

Massive Scale: Social interactions are massive in scale consisting of millions of users and billions of notifications. Therefore, it was clear that we needed techniques that can scale massively.

Event Overload: Many users receive an overwhelming number of notifications. This observation inspired us to introduce novel satisfaction metrics for limiting the number of events received by the users. Further, this prompted us to formulate the subscriber satisfaction problems under resource constraints.

Skewed Popularity: Some topics are extremely popular and some topics generate publications at a significantly higher rate than others. When there is increased pub/sub traffic, part of the workload that cannot be handled by the pub/sub infrastructure could be offloaded to a cheaper infrastructure. The peer-assisted design proposed in this thesis is inspired by such scenarios.

Traffic Variation: Pub/sub traffic can vary significantly at different points in time. The unpredictability of the pub/sub traffic inspired us to design a

tool that can estimate and allocate the required resources in datacenters and clouds.

While workload analysis potentially has many benefits, obtaining real workloads is not an easy task. The evidence for that is apparent from the lack of studies of real workloads in the literature. In this regard, collaborating with enterprises using similar systems and applications can be useful. Inspiration for many research problems in this thesis is a result of a collaboration with Spotify that allowed us to study their pub/sub design and workload.

Once the traces have been collected, there are several issues that need to be addressed before they can be analyzed or used for evaluation. First of all the data must be sanitized. For example, in the Spotify traces we collected there were large numbers of diagnostic messages sent along with the pub/sub traffic and eliminating them was essential for obtaining the real pub/sub traffic. Furthermore, processing the collected traces when they are of massive scale often requires scalable data processing techniques. For instance, after collecting the Spotify traces, we had to run a few Hadoop map/reduce jobs to compute a number of aggregate values such as the event rate of all topics. Data sanitization and preprocessing before using the traces are essential and sometimes time consuming.

8.2.2 Theoretical vs. Experimental Validation

In Computer Science, the validation of proposed ideas is critical. There are typically two ways of validating ideas: (1) formal methods consisting of theoretical proofs and guarantees (2) empirical validation through experiments. Systems built with sound theoretical foundations provide robust guarantees on performance and quality of results. On the other hand, experimental validation provides a testament that the idea works in practice. One of the lessons learned in this thesis is that, while designing a large-scale distributed system, a combination of both approaches yields better and faster results than using either of them individually. In this section we explain the rationale behind this choice with examples.

With the help of formalization and theoretical proofs we showed that *B3M*,

F-B3M and *MCSS* problems are computationally hard and exact solutions for them would be computationally expensive and not scalable. As a result, in this thesis we had to resort to heuristic solutions.

Furthermore, with the help of more theoretical proofs we were able to establish certain feasibility results. For example, using Corollary 6.5, we proved that there is no PTAS for the *B3M* problem. This gave us a proof that designing an approximate algorithm for the *B3M* problem is difficult.

In another instance, theoretically proving that the objective function of *F-B3M* exhibits the submodularity property assured us a greedy approximate solution with constant approximation ratio (based on [Fisher et al., 1978]). Further, we were able to exploit the submodularity property of the objective function to improve the performance of our solution (depicted in Algorithm 5).

From these examples the benefits of theoretical analysis of the proposed problems and solutions is clear. While theoretical analysis provides sound guarantees on the problems and solutions, it may not provide insights on how the solutions may behave in practice.

For assessing the performance of the proposed algorithms, typically, the worst-case complexity is derived. However, the worst-case complexity may not represent their performance in practice. In this regard, extensive experimental analysis using real traces obtained from deployed systems help in validation of our ideas for practical settings. For example, in Chapters 6 and 7, validating our ideas with real large-scale traces from Spotify and Twitter assured us that our solutions can potentially scale to millions of users efficiently.

While experimental validation provides insights on performance in practical settings, it does not quantify the quality of results. While deriving approximation ratios for the proposed algorithms is a reliable way to quantify their relative performance, it may be difficult and time consuming in many cases. For example, the two stage solution proposed for the *MCSS* problem in Section 7.2 consists of a number of optimizations, making it difficult to derive an approximation ratio. In many cases, deriving upper or lower bounds provides a faster and yet a reliable way of comparing how far a given solution is from the theoretical optimal solution. In Sections 6.3, 6.4 and 7.2, we derived such bounds

that helped us in determining the relative performance of our approaches compared to the best performance that any feasible solution can achieve.

8.2.3 Simulation vs. Real Deployment

Implementing the proposed algorithms and system designs is essential for experimental validation. Computer Science researchers are often faced with the choice of simulating the system or implementing and deploying the real system. Simulation simplifies the implementation of a system design, speeds up obtaining results and can potentially scale massively. However, the trade-off is that it does not consider all the issues faced in practice. Alternatively, building and deploying prototypes in real system settings provides more accurate results at the cost of increased engineering efforts and slower result collection.

In this thesis the choice of validating the system using simulations was made. However, the simulation settings are made as realistic as possible with many careful considerations:

Trace-Driven Simulations: For evaluating all the algorithms and system designs presented in this thesis, we used real traces collected from production systems. Before using the traces, they were analyzed in detail to understand their characteristics and confirm that they are representative. For example, we used the Twitter and Facebook social graphs as the subscription workload for evaluating the PolderCast system. In addition, we used traces from the Skype P2P network [Guha and Daswani, 2005] for modeling the churn and we introduced latency from the measurements taken by [Gummadi et al., 2002]. Finally, we used the Spotify and Twitter social interaction traces to evaluate the algorithms proposed for the subscriber satisfaction and resource allocation problems.

Sensitivity Analysis: In order to consider the practical settings and parameters for the simulations, we did an extensive sensitivity analysis of our simulations. For example, in Section 3.5, we evaluated the performance of the PolderCast dissemination algorithm with different values of the

dissemination fanout parameter. More examples are from Sections 6.5 and 7.3, where we repeated our experiments with up to four exponential variations of a value derived from the traces for the satisfaction threshold parameter (τ).

Scale: We scaled our simulations for evaluating the subscriber satisfaction and resource allocation problems to several millions of subscribers. Hence, these simulations provide evidence that it is possible for our solutions to scale massively. Obtaining such a result from a real deployment would require us to implement a full pub/sub system and deploy it in a production environment such as Spotify with participation of millions of users. It is difficult to achieve that in practice. On the other hand, we were able to run PolderCast simulations with up to only 10000 nodes due to scalability issues of the PeerSim simulator. However, we believe that PolderCast can potentially scale further.

8.3 Research Limitations

In this thesis, we thoroughly analyzed the research problems arising from the design of pub/sub for large-scale social interaction and we also proposed solutions for those problems with both theoretical as well as empirical results. However, as in most research works, there are still limitations to the research conducted in this dissertation due to several technical and non-technical hurdles. In this section, we discuss important limitations of this dissertation and reflect on reasons behind them and suggest possible solutions to overcome these limitations in the future.

8.3.1 Pub/Sub Trace Analysis

One of the main challenges faced by pub/sub researchers is the lack of real-world workloads. Even though we analyzed real traces from production social-interaction systems in Chapter 5, they still have following shortcomings:

1. The Spotify trace analysis presented in this thesis does not provide absolute numbers. For example, the topic popularity, publication event rate and

subscription cardinality are expressed in percentages instead of their total values. This is a result of the restriction imposed by the Spotify organization not to disclose absolute numbers due to business reasons.

2. Both Spotify and Twitter traces were samples collected for a period of 10 days. While this provides a good sample to analyze their characteristics, traces collected over longer periods could exhibit long-term patterns in the pub/sub traffic. Moreover, traces collected over longer periods may avoid any coincidental characteristics that traces collected over shorter periods exhibit. However, 10 days are sufficient to capture the short-term weekly patterns that are generally observed in most traces driven by user behaviors such as in [Zhang et al., 2013a]. For Spotify traces, collecting them on a production system was allowed for only limited time. Similarly, for Twitter, public APIs are limited in access. Given sufficient resources, time and unlimited access to the traces, it is possible to study long-term patterns and obtain perhaps more interesting results.

3. Even though the Twitter traces we collected were made public, the Spotify traces are not available for use by other researchers to evaluate their systems. In addition, lack of access to Spotify traces also limits the reproducibility of the results obtained from the experiments that utilize the Spotify traces. Restriction by the Spotify organization not to make their traces public is the reason for this limitation. Hence, there is a need for formalizing and implementing a model to generate synthetic traces that mimic the actual Spotify trace distributions. This is important in order for researchers to generate sample traces that are representative of the original traces. We could not design a generator due to limited duration of the thesis.

8.3.2 Trace-Driven Simulations

Utilizing traces from real applications to validate the pub/sub systems through simulations or emulations is widely practiced by academic researchers. However, that practice still has limitations.

1. Using specific traces for validation could limit the generality of systems. To avoid this, we used two different traces in both Chapter 3 and Chapter 7. However, there is still a need for evaluating the solutions presented in this thesis with more (general) traces. Designing a pub/sub system to handle generic traces could make it more robust and when considering alternative scenarios.

2. Trace-driven simulations adopted in this thesis are a good representative of the real deployment and they provide insightful results. However, it may not be feasible to simulate all scenarios that could occur in practice using trace-driven simulations. Given sufficient time and resources, it should be possible to deploy them in real settings with real users to validate the systems in the future.

8.3.3 Formal Analysis

Formal analysis to provide theoretical guarantees was used on several occasions in this thesis. However, formal analysis for some algorithms were omitted due to their difficulty.

1. In Chapter 6, while we provide an approximation ratio for the proposed algorithm to solve the *F-B3M* problem, we were unable to provide a similar analysis for the proposed heuristic for solving the *B3M* problem (Algorithm 2). As shown in Section 6.3, the *B3M* problem is NP-Hard (reduction from the *DkS* problem). The difficulty of providing a solution with an approximation ratio to the *B3M* problem is similar to that of solving the *DkS* problem with an approximation guarantee. Therefore, any efforts to derive an approximation ratio for Algorithm 2, will presumably take a lot of effort and such it is beyond the scope of this thesis.

2. Similarly, in Chapter 7, while we prove that the *MCSS* problem is NP-Hard, we do not provide any approximation guarantees for the proposed heuristics. This is because, decomposing the proposed solution for the *MCSS* problem into two different steps (as done in Section 7.2), makes it harder to derive an

approximation bound. On the other hand, we derive a lower-bound on the objective function of the *MCSS* problem that provides a baseline to compare our solution against.

8.4 Future Research Directions

We plan to extend our work in several directions as explained in this section.

8.4.1 Extensions to PolderCast

While PolderCast is focused on building pub/sub overlays, one of the interesting applications is to deploy social networking features on it. Extending PolderCast to deploy a real social network poses many more challenges.

PolderCast could be extended to include techniques to persist the publication messages, so that nodes joining after being offline for a period could retrieve the messages they lost. One of the main reasons why the hit-ratio of PolderCast under churn does not reach 100% (in Section 3.5) is that when nodes come back online they have no way of retrieving the lost events. We believe it would be a useful extension to improve the reliability of PolderCast. Moreover, notification persistence for future retrieval is one of the essential features of social networks. The challenge here is designing a strategy to partition and replicate events so as to maximize their availability. We could use the replication and data partitioning strategies utilized in the existing P2P systems [Blake and Rodrigues, 2003; Pace et al., 2011; Rowstron and Druschel, 2001b]. However, these techniques violate topic-connectivity by storing events on nodes not subscribing to events' topics. Hence, there is a need to consider this problem in the context of pub/sub overlays.

To realize PolderCast as a fully functioning P2P pub/sub system for social interaction, we must be able to deploy it on a real P2P network. However, deploying it on a P2P network consisting of user computers and devices such as smartphones requires several extensions to PolderCast. The challenge here is to preserve the desirable characteristics of a P2P pub/sub system described in Section 1.3.1 while deploying it on a real network.

- Overlay construction must consider the techniques to establish end-to-end connectivity between the peers. NAT traversal techniques could help in this direction. Some nodes may not be reachable at all due to firewall restrictions. In such situations the neighbor selection and the dissemination algorithm of PolderCast may have to take these restrictions into account.

- Since P2P systems rely on resources provided by the participating devices, it is crucial to design meaningful incentives for the users to share resources. Without an incentive mechanism it is hard to convince users to share their resources and ensure fairness among participating peers. While we could borrow incentive mechanisms used in existing P2P networks such as in bit torrent networks [Levin et al., 2008], it is not clear if those incentives are appropriate for pub/sub systems.

- Even though privacy issues are not addressed in this thesis, they are very important to consider before deploying PolderCast on a real P2P network. An example of a privacy issue is: while gossiping is a lightweight technique to build pub/sub overlays, it reveals node interests and other information to the neighbors. Developing gossiping techniques that effectively build pub/sub overlays without revealing private information of the nodes is an open problem.

8.4.2 Peer-Assisted Pub/Sub System

In Section 6.1, we presented a peer-assisted architecture that includes a way of offloading pub/sub workloads to P2P networks. In order to build a working peer-assisted system, there is a need for designing a peer-assisted architecture that can seamlessly combine a P2P network and a dedicated infrastructure.

In this regard, integrating a P2P TBPS system such as PolderCast with the peer-assisted architecture proposed in this thesis could lead to a new problem: how to share the workload between the dedicated infrastructure and currently available peers with a certain capacity so as to meet the satisfaction requirements of all users? By considering the available resources in a P2P network in real-time,

we could design a more reactive algorithm to disseminate messages. For example, a sudden surge in events generated by popular topics could be disseminated using a combination of the dedicated infrastructure and a P2P network by considering currently available resources in both systems.

8.4.3 Different Satisfaction Metrics

The satisfaction metrics we proposed in this thesis (in Chapter 6 and Chapter 7) are inspired by the Spotify application scenario, where the event delivery rate beyond a certain threshold is considered not useful for the users. However, the satisfaction metrics can be extended to include more metrics such as ranking and top-k semantics explained in Section 2.5. If a scoring function to rank the events received by a user is given, extending the satisfaction metric to include top-k semantics leads to a new problem: how to maximize the total score of the 'k' events that each user receives at any point in time such that the resource constraints are not violated. Alternatively, the problem is to select a minimum cost subset of the workload so as to maximize the total score of the 'k' events delivered to each user. One possible solution for such a problem is to modify the heuristics used to solve *B3M*, *F-B3M* and *MCSS* problems (presented in Sections 6.3, 6.4 and 7.2) to include weighted benefit values for each topic-subscriber pair based on the given scoring function. Similarly, the benefit-cost heuristic could be adapted to match other satisfaction metrics.

8.4.4 Online Algorithms

The theoretical problems considered in this thesis (*B3M*, *F-B3M* and *MCSS*) are solved for a static setting. While the fast performance of our solutions is suitable to run them periodically to adapt to the changes in the pub/sub workload, it is also useful to provide online solutions to our problems, since they guarantee interruption-free service with a certain quality of results. An online solution should be able to process individual inputs such as users joining, users leaving, new subscriptions, unsubscriptions and publication event rate changes and take appropriate actions. The challenge here is that the actions taken must ensure

that the quality of the resulting solution must be within certain bounds compared to the optimal solution or an upper bound.

8.4.5 Building a Complete Pub/Sub System

It would be both a research and engineering challenge to integrate the solutions presented in this thesis to build a fully functioning pub/sub system.

In particular, the solution for cost-effective resource allocation can be used as a building block in conjunction with other building blocks to design a complete pub/sub system. For example, a directory service providing the information about which server a given topic-subscriber pair is allocated to is essential for routing incoming publications to the appropriate servers.

Another essential building block is to be able to allocate resources via the APIs given by the Infrastructure-as-a-Service (IaaS) providers. In addition to that, this building block is also required to migrate topic-subscriber pairs from one server to another depending on the outcome of the allocation algorithm. The challenge here is allocating and migrating swiftly without disrupting the pub/sub service.

Bibliography

Apache ActiveMQ. http://activemq.apache.org/.

Amazon Simple Notification Service. https://aws.amazon.com/sns/.

Microsoft Azure Service Bus. https://azure.microsoft.com/en-us/services/service-bus/.

Java Messaging Service. https://jcp.org/en/jsr/detail?id=343.

Oracle streams advanced queuing user's guide. http://docs.oracle.com/cd/E11882_01/server.112/e11013/toc.htm.

PASTRY-PEERSIM: An implementation of the Pastry protocol for PeerSim. http://peersim.sourceforge.net/code/pastry.tar.gz.

PubNub: Enabling Realtime Connectivity for the Internet of Things. http://www.pubnub.com/.

TIBCO Rendezvous Concepts. https://docs.tibco.com/pub/rendezvous/8.4.2/doc/pdf/TIB_rv_concepts.pdf, 2012.

Marcos K Aguilera, Robert E Strom, Daniel C Sturman, Mark Astley, and Tushar D Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61. ACM, 1999.

Norman Ahmed, Mark Linderman, and Jason Bryant. Papas: Peer assisted publish and subscribe. In *Proceedings of the 7th Workshop on Middleware for Next*

Generation Internet Computing, MW4NG '12, pages 1–6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1607-1. doi: 10.1145/2405178.2405185. URL http://doi.acm.org/10.1145/2405178.2405185.

Sebastien Baehni, Patrick Th Eugster, and Rachid Guerraoui. Data-aware multicast. In *Dependable Systems and Networks, 2004 International Conference on*, pages 233–242. IEEE, 2004.

Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni, and Sara Tucci-Piergiovanni. Tera: topic-based event routing for peer-to-peer architectures. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 2–13. ACM, 2007a.

Roberto Baldoni, Roberto Beraldi, Leonardo Querzoni, and Antonino Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to siena. *The Computer Journal*, 50(4):444–459, 2007b.

Guruduth Banavar, Tushar Chandra, Robert Strom, and Daniel Sturman. A case for message oriented middleware. In *Distributed Computing*, pages 1–17. Springer, 1999.

Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 63–74. ACM, 2013.

Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 567–576, July 2014.

Ashwin R Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *ACM SIGCOMM Computer Communication Review*, 34(4):353–366, 2004.

Aditya Bhaskara, Moses Charikar, Eden Chlamtac, Uriel Feige, and Aravindan Vijayaraghavan. Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k-subgraph. In *Proceedings of the 42nd ACM symposium on Theory of computing*, pages 201–210. ACM, 2010.

Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *HotOS*, volume 3, page 1, 2003.

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

Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8): 1489–1499, 2002.

Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 international conference on Management of data*, pages 725–736. ACM, 2013.

Javier Cerviño, Evangelia Kalyvianaki, Joaquin Salvachúa, and Peter Pietzuch. Adaptive provisioning of stream processing systems in the cloud. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 295–301. IEEE, 2012.

Chen Chen, H.-A. Jacobsen, and R. Vitenberg. Divide and conquer algorithms for publish/subscribe overlay design. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 622–633, June 2010. doi: 10.1109/ICDCS.2010.87.

Chen Chen, R. Vitenberg, and H.-A. Jacobsen. Scaling construction of low fan-out overlays for topic-based publish/subscribe systems. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 225–236, June 2011. doi: 10.1109/ICDCS.2011.68.

Chen Chen, Roman Vitenberg, and Hans-Arno Jacobsen. A generalized algorithm for publish/subscribe overlay design and its fast implementation. In Marcos K. Aguilera, editor, *Distributed Computing*, volume 7611 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2012.

Chen Chen, Roman Vitenberg, and Hans-Arno Jacobsen. Elasto: Dynamic, efficient, and robust maintenance of low fan-out overlays for topic-based publish/subscribe under churn. Technical report, University of Toronto, 2013.

Alex King Yeung Cheung and Hans-Arno Jacobsen. Publisher placement algorithms in content-based publish/subscribe. In *Proc. IEEE ICDCS*, pages 653–664, 2010.

Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 109–118. ACM, 2007a.

Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. SpiderCast: a scalable interest-aware overlay for topic-based pub/sub communication. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 14–25. ACM, 2007b.

Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *Software Engineering, IEEE Transactions on*, 27(9):827–850, 2001.

Marina Drosou, Kostas Stefanidis, and Evaggelia Pitoura. Preference-aware publish/subscribe delivery with diversity. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-665-6.

Samuel Eilon and Nicos Christofides. The loading problem. *Management Science*, 17(5):259–268, 1971.

Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv. (CSUR)*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078.

Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João Pereira, Kenneth A Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD Record*, volume 30, pages 115–126. ACM, 2001.

Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, February 1999. ISSN 0022-0000. doi: 10.1006/jcss.1998.1600. URL http://dx.doi.org/10.1006/jcss.1998.1600.

Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. ACM. ISBN 1-58113-361-8. doi: 10.1145/375551.375567. URL http://doi.acm.org/10.1145/375551.375567.

Uriel Feige, Michael Seltser, et al. On the densest k-subgraph problem. *The Weizmann Institute, Rehovot, Tech. Rep*, 1997.

Uriel Feige, David Peleg, and Guy Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

Marshall L Fisher, George L Nemhauser, and Laurence A Wolsey. An analysis of approximations for maximizing submodular set functions - ii. In *Polyhedral combinatorics*, pages 73–87. Springer, 1978.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Stéphane Genaud and Julien Gossa. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 1–8. IEEE, 2011.

Sarunas Girdzijauskas, Gregory Chockler, Ymir Vigfusson, Yoav Tock, and Roie Melamed. Magnet: practical subscription clustering for internet-scale publish/subscribe. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 172–183. ACM, 2010.

Manuel Gomez-Rodriguez, Krishna P. Gummadi, and Bernhard Schoelkopf. Quantifying information overload in social media and its impact on social contagions. In *Proceedings of the 8th International AAAI Conference on Weblogs and Social Media (ICWSM)*, 2014.

Saikat Guha and Neil Daswani. An experimental study of the skype peer-to-peer voip system. Technical report, Cornell University, 2005.

Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *Parallel and Distributed Systems, IEEE Transactions on*, 23 (12):2351–2365, 2012.

Krishna P Gummadi, Stefan Saroiu, and Steven D Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18. ACM, 2002.

Abhishek Gupta, Ozgur D Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273. Springer-Verlag New York, Inc., 2004.

Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

Thomas Heinze, Patrick Meyer, Zbigniew Jerzak, and Christof Fetzer. Demo: Measuring and estimating monetary cost for cloud-based data stream processing. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 333–334. ACM, 2013.

Dorit S. Hochbaum. Approximation algorithms for NP-hard problems. PWS Publishing Co., 1997.

Dorit S Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and vlsi. *Journal of the ACM (JACM)*, 32(1):130–136, 1985.

Joe Hoffert, Douglas C Schmidt, and Aniruddha Gokhale. Adapting distributed real-time and embedded pub/sub middleware for cloud computing environments. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, pages 21–41. Springer-Verlag, 2010.

Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

Atsushi Ishii and Toyotaro Suzumura. Elastic stream computing with clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 195–202. IEEE, 2011.

Hans-Arno Jacobsen, Alex King Yeung Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The PADRES publish/subscribe system. *Principles and Applications of Distributed Event-Based Systems*, 2010.

Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.

Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Publiy+: A peer-assisted publish/subscribe service for timely dissemination of bulk content. *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 345–354, 2012. ISSN 1063-6927. doi: http://doi.ieeecomputersociety.org/10.1109/ICDCS.2012.62.

Anne-Marie Kermarrec and Peter Triantafillou. XL peer-to-peer pub/sub systems. *ACM Comput. Surv. (CSUR)*, 46(2):1–45, November 2013. ISSN 0360-0300.

doi: 10.1145/2543581.2543583. URL http://doi.acm.org/10.1145/2543581.2543583.

Subhash Khot. Ruling out PTAS for graph min-bisection, dense k-subgraph, and bipartite clique. *SIAM Journal on Computing*, 36(4):1025–1071, 2006.

Samir Khuller, Anna Moss, and Joseph Seffi Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999.

Andreas Krause and Carlos Guestrin. A note on the budgeted maximization of submodular functions. Technical Report CMU-CALD-05-103, Carnegie Mellon University, 2011.

Gunnar Kreitz and Fredrik Niemela. Spotify–large scale, low latency, P2P music-on-demand streaming. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1–10. IEEE, 2010.

Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

Raffi Krikorian. Twitter by the numers. http://www.slideshare.net/raffikrikorian/twitter-by-the-numbers, 2010.

Raffi Krikorian. Timelines at scale. http://www.infoq.com/presentations/Twitter-Timeline-Scalability, 2013.

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8.

Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

Dave Levin, Katrina LaCurts, Neil Spring, and Bobby Bhattacharjee. Bittorrent is an auction: Analyzing and improving bittorrent's incentives. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM

'08, pages 243–254, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-175-0. doi: 10.1145/1402958.1402987. URL http://doi.acm.org/10.1145/1402958.1402987.

Rhyd Lewis. A general-purpose hill-climbing method for order independent minimum grouping problems: A case study in graph colouring and bin packing. *Computers & Operations Research*, 2009.

Guoli Li. *Optimal and Robust Routing of Subscriptions for Unifying Access to the Past and the Future in Publish/Subscribe*. PhD thesis, Graduate Department of Computer Science, University of Toronto, 2010.

Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web (TWEB)*, 4(1):2:1–2:33, January 2010. ISSN 1559-1131. doi: 10.1145/1658373.1658375.

Hongzhou Liu, Venugopalan Ramasubramanian, and Emin Gün Sirer. Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 29–34, 2005.

Ashwin Machanavajjhala, Erik Vee, Minos Garofalakis, and Jayavel Shanmugasundaram. Scalable ranked publish/subscribe. *Proceedings of the VLDB Endowment*, 1(1):451–462, 2008.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

Silvano Martello and Paolo Toth. *Knapsack problems*. Wiley New York, 1990.

Miguel Matos, Ana Nunes, Rui Oliveira, and José Pereira. Stan: exploiting shared interests without disclosing them in gossip-based publish/subscribe. In *IPTPS*, 2010.

Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In

Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, pages 29–42. ACM, 2007.

Alberto Montresor and Márk Jelasity. Peersim: A scalable P2P simulator. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 99–100. IEEE, 2009.

M. Onus and AW. Richa. Parameterized maximum and average degree approximation in topic-based publish-subscribe overlay network design. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 644–652, June 2010. doi: 10.1109/ICDCS.2010.54.

Melih Onus and Andréa W. Richa. Minimum maximum-degree publish-subscribe overlay network design. *IEEE/ACM Trans. Netw.*, 19(5):1331–1343, October 2011. ISSN 1063-6692. doi: 10.1109/TNET.2011.2144999. URL http://dx.doi.org/10.1109/TNET.2011.2144999.

Alessio Pace, Vivien Quéma, and Valerio Schiavoni. Exploiting node connection regularity for dht replication. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 111–120. IEEE, 2011.

Jay A Patel, Étienne Rivière, Indranil Gupta, and Anne-Marie Kermarrec. Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks*, 53(13):2304–2320, 2009.

Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. G-ToPSS: Fast filtering of graph-based metadata. In *Proc. ACM WWW*, pages 539–547, 2005.

P.R. Pietzuch and J.M. Bacon. Hermes: a distributed event-based middleware architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 611–618, 2002. doi: 10.1109/ICDCSW.2002.1030837.

Krešimir Pripužić, Ivana Podnar Žarko, and Karl Aberer. Top-k/w publish/subscribe: finding k most relevant publications in sliding time window

w. In *Proceedings of the second international conference on Distributed event-based systems*, pages 127–138. ACM, 2008.

Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H Payberah, and Seif Haridi. Vitis: A gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous routing in unstructured overlay networks. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 746–757. IEEE, 2011.

Weixiong Rao and Lei Chen. A distributed full-text top-K document dissemination system in distributed hash tables. *World Wide Web*, 14(5-6):545–572, 2011.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

John Reumann. GooPS: Pub/Sub at Google, 2009. Lecture & Personal Communications at EuroSys & CANOE Summer School.

Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001a.

Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 188–201. ACM, 2001b.

Mohammad Sadoghi and Hans-Arno Jacobsen. BE-Tree: An index structure to efficiently match boolean expressions over high-dimensional discrete space. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 637–648. ACM, 2011.

Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proceedings of the VLDB Endowment*, 3(1-2):1525–1528, 2010.

Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. Multi-query stream processing on fpgas. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1229–1232. IEEE, 2012.

Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, pages 3–5. Brisbane, Australia, 1997.

Vinay Setty, Gunnar Kreitz, Roman Vitenberg, Maarten van Steen, Guido Urdaneta, and Staffan Gimåker. The hidden pub/sub of spotify:(industry article). In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 231–240. ACM, 2013.

Alexander Shraer, Maxim Gurevich, Marcus Fontoura, and Vanja Josifovski. Top-k publish-subscribe for social annotation of news. *Proc. VLDB Endow.*, 6(6): 385–396, April 2013. ISSN 2150-8097. doi: 10.14778/2536336.2536340. URL http://dx.doi.org/10.14778/2536336.2536340.

M. Sindelar, R. Sitaraman, and P. Shenoy. Sharing-aware algorithms for virtual machine colocation. In *SPAA*, 2011.

Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

Wesley W Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8. ACM, 2003.

Yoav Tock, Nir Naaman, Avi Harpaz, and Gidon Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED Parallel and Distributed Computing and Systems (PDCS)*, pages 320–326, 2005.

Nam-Luc Tran, Sabri Skhiri, and Esteban Zimányi. Eqs: An elastic and scalable message queue for the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 391–398. IEEE, 2011.

Peter Triantafillou and Ioannis Aekaterinidis. Peer-to-peer publish-subscribe systems. In Ling Liu and M.Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2069–2075. Springer US, 2009. doi: 10.1007/978-0-387-39940-9_1221.

Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. volume 40, pages 423–436. ACM, 2012.

David Villegas, Athanasios Antoniou, Seyed Masoud Sadjadi, and Alexandru Iosup. An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 612–619. IEEE, 2012.

Spyros Voulgaris. *Epidemic-Based Self-Organization in Peer-to-Peer Systems*. Phd thesis, VU Universiteit Amsterdam, 2006.

Spyros Voulgaris and Maarten Van Steen. Hybrid dissemination: adding determinism to probabilistic multicasting in large-scale P2P systems. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 389–409. Springer-Verlag New York, Inc., 2007.

Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

Spyros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec, Maarten Van Steen, et al. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *IPTPS*, 2006.

Christo Wilson, Bryce Boe, Alessandra Sala, Krishna PN Puttaswamy, and Ben Y Zhao. User interactions in social networks and their implications. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 205–218, 2009.

Bernard Wong and Saikat Guha. Quasar: a probabilistic publish-subscribe system for social networks. In *IPTPS*, 2008.

Tianyin Xu, Yang Chen, Lei Jiao, Ben Y. Zhao, Pan Hui, and Xiaoming Fu. Scaling microblogging services with divergent traffic demands. In *Proc. ACM/IFIP/USENIX Middleware*, pages 20–40, 2011.

Albert Yu, Pankaj K Agarwal, and Jun Yang. Generating wide-area content-based publish/subscribe workloads. *Network Meets Database (NetDB)*, 2009.

Boxun Zhang, Gunnar Kreitz, Marcus Isaksson, Javier Ubillos, Guido Urdaneta, Johan A Pouwelse, and Dick Epema. Understanding user behavior in Spotify. In *INFOCOM, 2013 Proceedings IEEE*, pages 220–224. IEEE, 2013a.

Kaiwen Zhang, Mohammad Sadoghi, Vinod Muthusamy, and Hans-Arno Jacobsen. Distributed ranked data dissemination in social networks. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 369–379. IEEE, 2013b.

Shelley Q Zhuang, Ben Y Zhao, Anthony D Joseph, Randy H Katz, and John D Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20. ACM, 2001.