The Hidden Pub/Sub of Spotify (Industry Article)

Vinay Setty
University of Oslo, Norway
vinay@ifi.uio.no

Gunnar Kreitz
Spotify and KTH – Royal Institute of Technology, Stockholm, Sweden
gkreitz@kth.se

Roman Vitenberg
University of Oslo, Norway
romanvi@ifi.uio.no

Maarten van Steen
VU University and The Network Institute
Amsterdam, The Netherlands
steen@cs.vu.nl

Guido Urdaneta
Spotify, Stockholm, Sweden
guidou@spotify.com

Staffan Gimåker
Spotify, Stockholm, Sweden
staffan@spotify.com

ABSTRACT

Spotify is a peer-assisted music streaming service that has gained worldwide popularity. Apart from providing instant access to over 20 million music tracks, *Spotify* also enhances its users' music experience by providing various features for social interaction. These are realized by a system using the widely-adopted pub/sub paradigm. In this paper we provide an interesting case study of a hybrid pub/sub system designed for real-time as well as offline notifications for *Spotify* users. We firstly describe a multitude of use cases where pub/sub is applied. Secondly, we study the design of its pub/sub system used for matching, disseminating and persisting billions of publications every day. Finally, we study pub/sub traffic collected from the production system, derive characterizations of the pub/sub workload, and show some interesting findings and trends.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

Keywords

Pub/Sub Systems,Event Notifications,Workload Analysis

1. INTRODUCTION

Spotify is a successful peer-assisted music-streaming service that provides access to over 20 million tracks to its users residing in 20 countries. The technical architecture providing the streaming service and user behavior of *Spotify* have been described in two recent studies [1, 2]. However, little has been said about the technical details of one of *Spotify*'s most engaging features: its ability to facilitate sharing and following of various music activities among its users in real time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.
Copyright 2013 ACM 978-1-4503-1758-0/13/06 ...$15.00.

In this paper, we explain how the 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, a popular communication paradigm that provides a loosely coupled form of interaction among a large number of publishing data sources and subscribing data sinks [3]. Thus, this paper adds a new unique application to a currently known list of large-scale systems that report benefits from using pub/sub, which includes application integration [4], financial data dissemination [5], RSS feed distribution and filtering [6], and business process management [7].

The end-to-end architecture of the pub/sub engine at Spotify is the main 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.

We study the performance of the system based on recently recorded 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, subscription cardinality, and temporal subscription/unsubscription patterns. Unfortunately, there exist precious few characterizations of subscriptions and synthetic workload generators for pub/sub systems [8]. 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 the paper 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 show that the traffic due to the activity of following friends dominates the total traffic of social interactions.

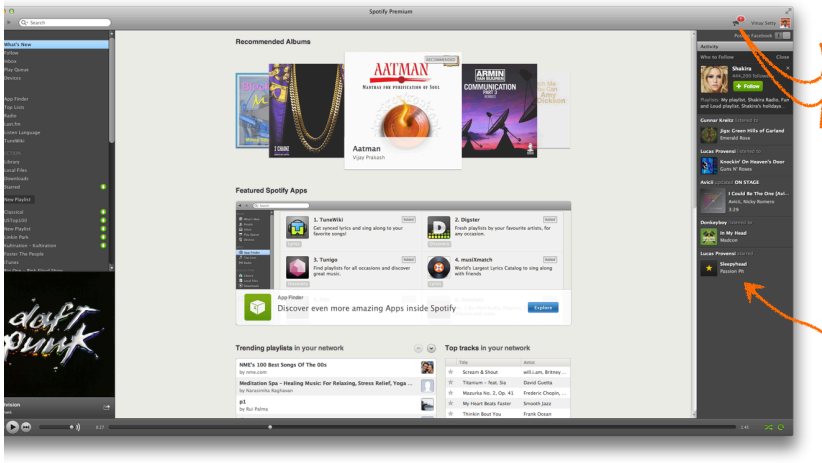

Figure 1: Spotify Desktop Client Snapshot

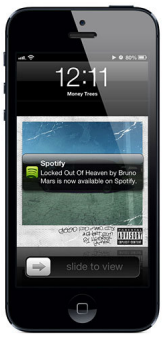

Figure 2: Push Notification

2. SPOTIFY PUB/SUB MODEL AND FEATURES

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 *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 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 of 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 or 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.

3. 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 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 data-centers (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

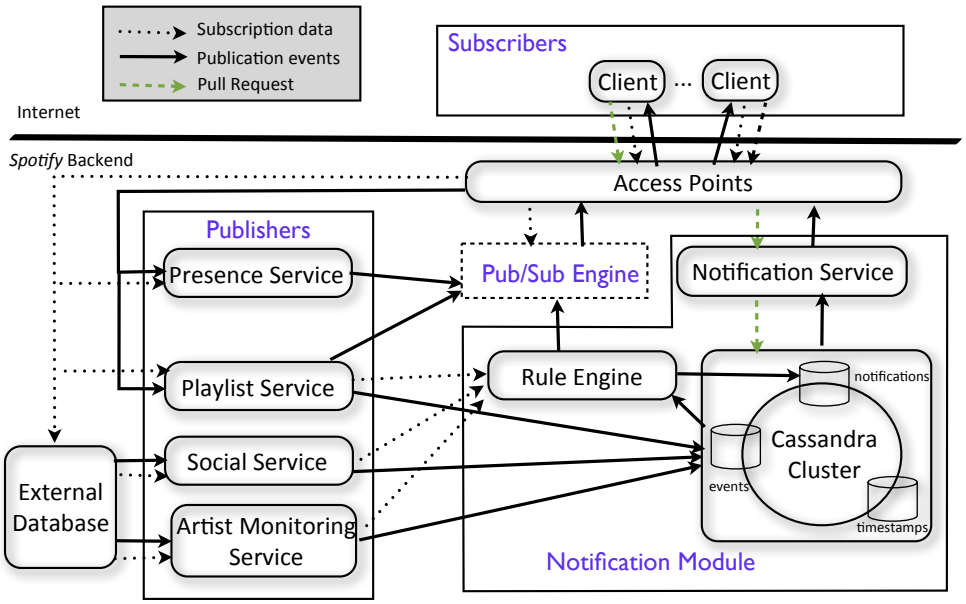

Figure 3: Architecture Supporting Social Interaction

main purpose is to host the music streaming service and all the back-end services necessary for *Spotify* to function.

3.1 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 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 Pub/Sub Engine. However, Pub/Sub Engine is light-weight 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 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 Notification Module by storing the incoming publication events in the Cassandra cluster [9] for reliable and offline delivery. This will be explained in detail later in Section 3.3.

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 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.

3.2 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 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 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

Table 1: List of topic types and corresponding services

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

includes the user name of the subscriber and the corresponding URI for the service, as listed in Table 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 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 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 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 3 and explained in detail in Section 3.4) 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 1.

The Playlist Service is mainly responsible for tracking playlist modifications made by users. As explained in Section 1, a playlist can be subscribed 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 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.

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 the artist monitoring-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 1.

3.3 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 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 2.

Email notifications: When a user is not online, events like artist, playlist and friend updates are sent via

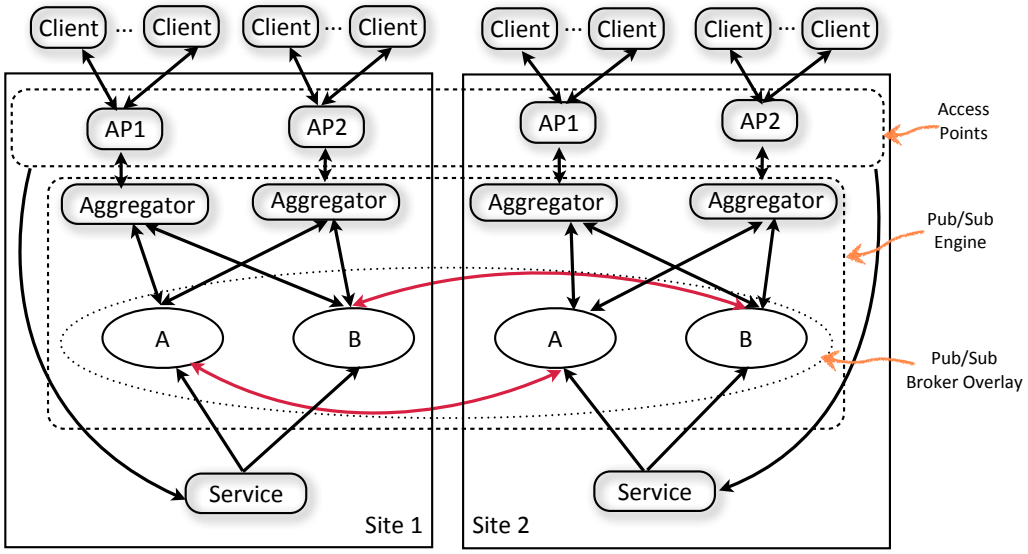

Figure 4: Real-Time Pub/Sub

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 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 Pub/Sub Engine and Cassandra for persistence.

3.3.1 *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 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 4), 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.

3.4 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.

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 subscriber-topic 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 subscrip-

tions 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. The Aggregator 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 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 in 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, broker A in Site 1 has a corresponding broker A in Site 2 (i.e., all the subscriptions obtained within Site 1 and managed at broker A, are also forwarded and replicated at 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 the broker A of Site 2 forwards the publication to the corresponding subscriber in 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.[10]

Load Balancing: Since all subscriptions are in 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. ANALYSIS OF SPOTIFY PUB/SUB WORKLOAD

In this section we study the different characteristics and patterns emerging from the pub/sub traffic 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 in this paper 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.

4.1 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, at any time 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) of the percentage of the total number of subscribers subscribing to a topic, shown in Figure 5.
- The distribution of **Subscription Size**: The CCDF of the percentage of total number of topics subscribed by a single subscriber, shown in Figure 6.
- 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 7.

It is easy to see that log-log plots CCDFs of topic popularity and subscription size Figure 6 follow a distribution close to a power law. The CCDF of Publication Event Rate, on the other hand, does not follow power-law. There is a sharp deviation around 0.0005% of the total number of publication events.

Notice that the CCDF of topic popularity and subscription size are similar to the typical degree distribution in social networks [11]. 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. Also, as mentioned in Section 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 workload for academic works on topic-based pub/sub systems as done in [10, 12].

Next we study the distribution of the number of publications attracted by subscriptions. We call it *Subscription Cardinality* per subscriber, which we define as the percentage of total publications events matching the topics subscribed by a subscriber. It is mathematically expressed as below:

$$C(S) = \frac{\sum_{t_S \in S} ev(t_S)}{\sum_{t \in \mathcal{T}} ev(t)} * 100$$

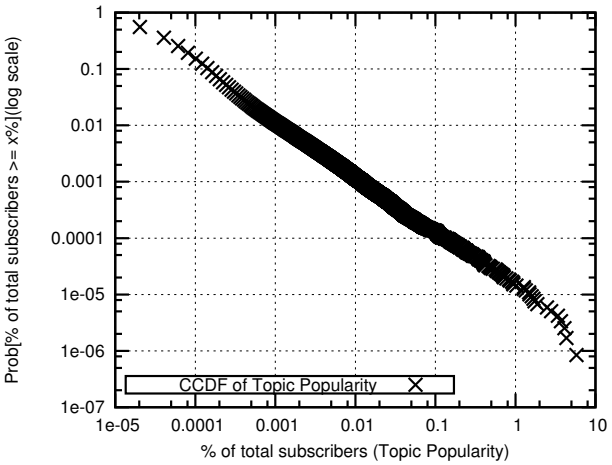

Figure 5: CCDF of Topic Popularity

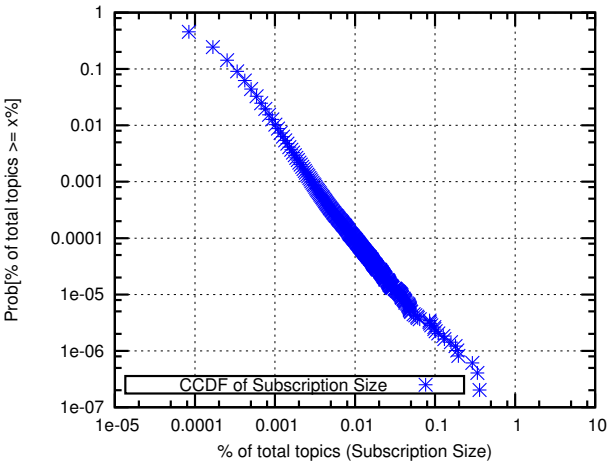

Figure 6: CCDF of Subscription Size per user

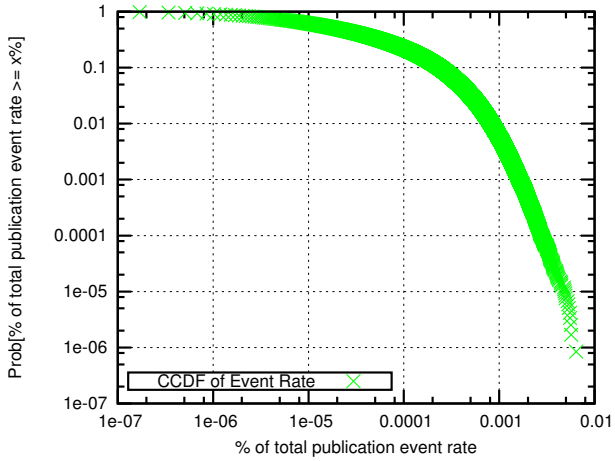

Figure 7: CCDF of Publication Event Rate per topic

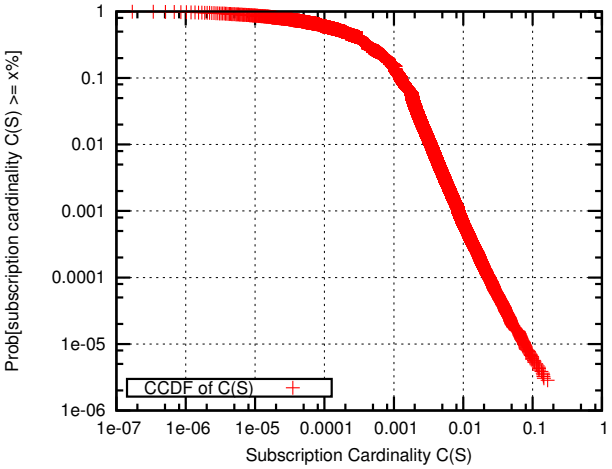

Figure 8: CCDF of Subscription Cardinality per user

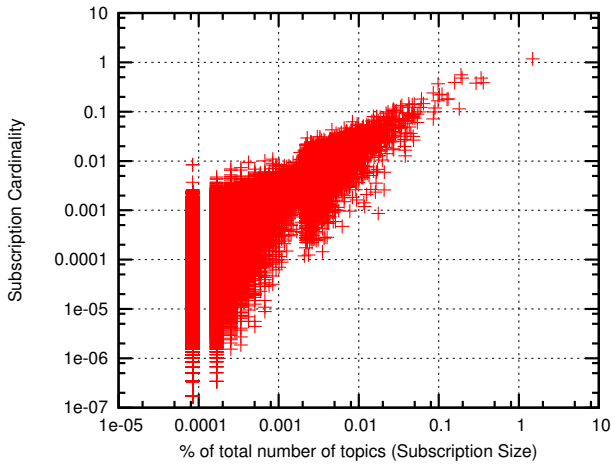

Figure 9: Relationship between subscription cardinality and subscription size (%of total number of topics)

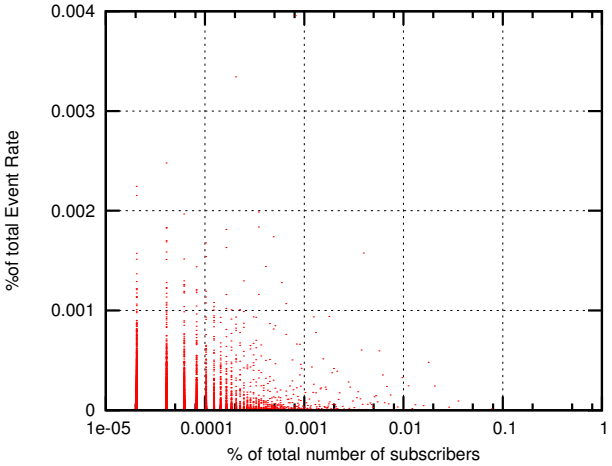

Figure 10: Relationship between topic Popularity (% of total number of subscribers) and Publication Event Rate

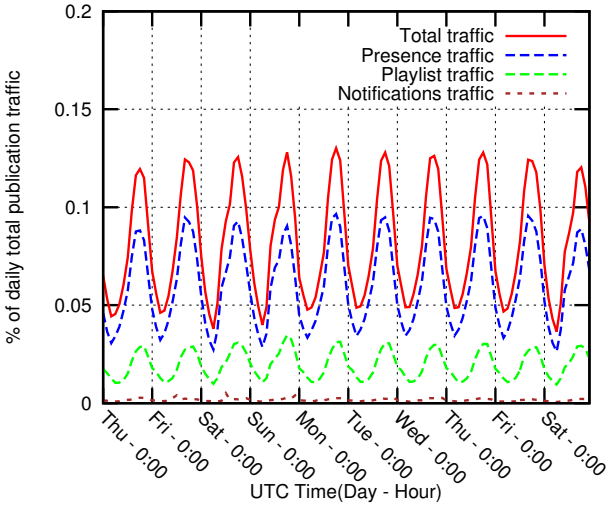

Figure 11: Pattern of publications generated per service-basis

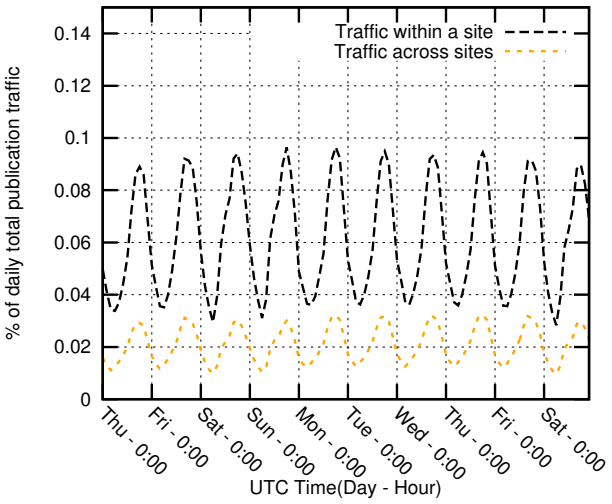

Figure 12: Publication traffic within the sites vs across the sites

Where, $\mathcal{T}$ is a global set of all topics, $S \subseteq \mathcal{T}$ is a set of topics subscribed by a subscriber, $ev(t)$ is the publication event rate of topic t. Thus, a subscriber with a subscription cardinality of, for example, 0.1%, receives 0.1% of all publications in the system.

In Figure 8 the x-axis is the cardinality of a subscription C(S) and y-axis is the probability that a subscription has cardinality greater than or equal to that of corresponding value in x-axis (in other words CCDF). This distribution is an interesting result for the pub/sub community since cardinality of subscriptions is an important design parameter for many pub/sub systems [13] and they are generally estimated probabilistically. Our analysis shows a diverse subscription cardinality ranging from 0.2% to as low as 10^{-7}%. Also more than 90% of the subscribers have $C(S) < 0.001\%$ which shows a very low cardinality.

Each subscriber is allowed to subscribe to an arbitrary number of topics and that results in arbitrary subscription sizes for the subscribers. A study about the relationship between subscription sizes and the corresponding matching events is crucial to understand the resources needed to handle the publication traffic at the brokers. We do this study by considering each subscriber's subscription cardinality and the corresponding subscription size. We show in Figure 9 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, the cardinality) also increases linearly. In Figure 9 we show a 1% random sample of all the points due to significantly high number of data points slows down the pdf rendering.

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 10. 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 to improve pdf rendering speed.

4.2 Pub/Sub Traffic Analysis

The following measurements correspond to all the publisher services mentioned in Section 3 and are not limited to the Presence service. We also include traces from all sites for these measurements for the same 10 days mentioned earlier.

4.2.1 *Publication Traffic*

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

The Presence Traffic: From Figure 11 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 in the morning. Also, 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 [2]. The reason for this pattern is simply because the publications generated by the Presence service are due to the playback of music tracks. It is easy to observe from Figure 11 that Presence events are the majority among the publication traffic, followed by Playlist events.

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

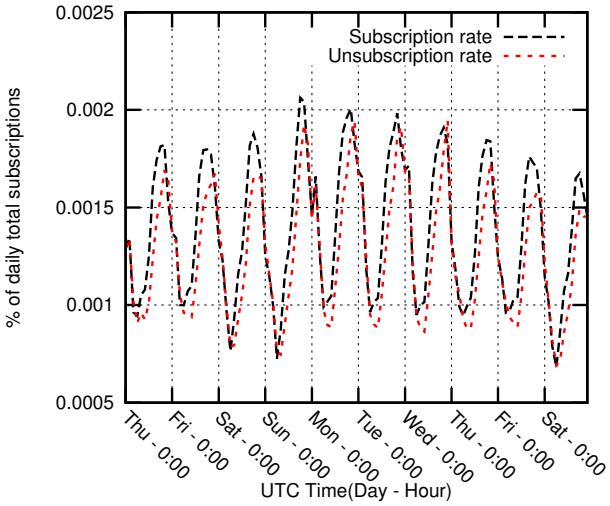

Figure 13: Subscription and unsubscription rate

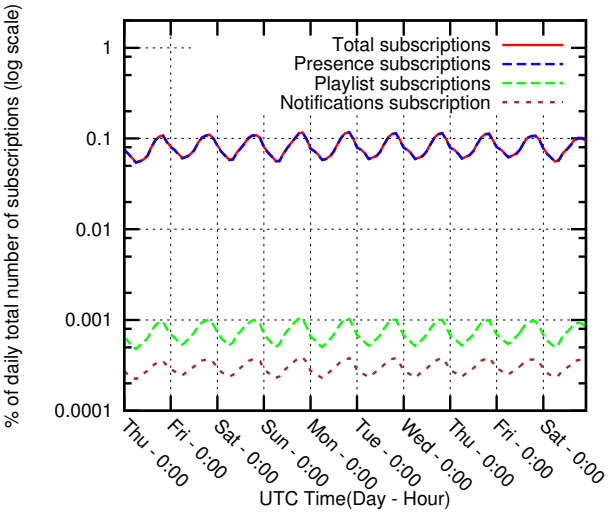

Figure 14: Pattern of percentage of total number of subscriptions

Social service, one can observe small spikes with notifications every day stemming from batch jobs launched for artist updates. Notice the significantly low traffic due to notification module compared to the Presence service and the Playlist service. This is in consistent with the hybrid design principle for real-time notification for the Presence service and offline notification for artist pages and social updates.

Next we compare the total publication traffic (from all services) generated from within the same site (local site) against the publication traffic generated 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 12, remote traffic is nearly an order of magnitude lower than local traffic. This result is in accordance with the design of *Spotify* pub/sub with each site having a pub/sub system designed to handle high local traffic, and low remote traffic with cross-site links at the pub/sub broker overlay (as described in Section 3.4).

4.2.2 Subscription Traffic

Figure 13 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 [10, 14, 13] 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 paper 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 patterns is significantly different from weekly pattern of publication traffic due to the simple fact that subscription traffic is due to the users logging in and out of the system while publication traffic is due to the playback of music. An interesting observation to make in Figure 13 is that the curve for unsubscription rate is ahead of the curve for subscriptions approximately 2 hours on average. Also the rate of subscriptions and unsubscriptions match approximately, hence the number of subscriptions for a small period remain constant. This hypothesis is confirmed by Figure 14, which shows that there is little variation in the number of subscriptions for the chosen time period. In Figure 14 we can also see that the number of subscriptions is dominated by the Presence service. This is because *Spotify* users by default have more subscriptions to follow their friends than subscriptions to follow Playlists, artists and album pages. This also confirms our previous claim that the Presence traffic dominates *Spotify* pub/sub traffic.

4.3 Summary

Here is the summary of important observations 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.
- The Publication Event Rate does not follow power law distribution.
- Subscription cardinality is significantly low (max 1%) and varies from 1% to as low as 10^{-7}%, indicating subscriptions with diverse subscription cardinality.
- Subscription cardinality of a subscriber is linearly proportional number of topics subscribed by that subscriber.
- The Publication Event Rate of a topic bears no relation to its popularity.
- 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 significant churn in subscriptions. However, the total number of subscriptions does not change much in a 10-day period.

- Both subscription and publication traffic is dominated by traffic related to the Presence service.

5. RELATED WORK

Pub/sub systems have been a subject of research for several decades, and a large number of topic-based pub/sub systems have been proposed in that period. Many academic research systems like Scribe [14], Bayeux [15], Tera [16], SpiderCast[17], Vitis [12], PolderCast[10] have focused on topic-based pub/sub. Several topic-based pub/sub systems have been proposed in the industry as well [4, 5].

However, supporting social interaction between *Spotify* users poses a specific set of requirements. There is a need to handle publications with different levels of criticality. As a result, there is a need for resource-efficient real-time delivery of events as well as reliable and offline delivery. *Spotify* also demands at-least-once delivery of events across all devices of the same user. In addition to that, there are multiple services generating publication events which need to be efficiently channeled via different paths to the subscribers. These requirements result in the need for a hybrid pub/sub system with different event flow paths.

Given the lack of real data, researchers in academia rely on synthetic workload generation techniques [14, 16, 18, 17]. Even though there have been some characterizations of pub/sub workloads from real systems in the past [6, 19, 8], they all mainly focus on characterizing the distribution of topic popularity in the workload. To the best of our knowledge, this work is the first to characterize the distributions of topic popularity, subscription sizes, distribution of per-topic publication event rate and its relationship with topic popularity, as well as the distribution of subscription cardinality and its relationship with subscription sizes. All of the analyses have been performed based on an actual deployed system.

6. CONCLUSIONS

In this paper, 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. We did an extensive study of the system by analyzing real traces collected from an actual deployed system. We characterize the system workload, which helps model pub/sub workloads for research. We also analyze the pub/sub traffic at *Spotify* to derive trends and patterns.

Acknowledgments

We would like to thank the following engineers of *Spotify* specifically, for their invaluable inputs and support for this paper: Tommie Gannert, Mikael Goldmann and Javier Ubillos.

7. REFERENCES

[1] G. Kreitz and F. Niemela, "Spotify – large scale, low latency, P2P music-on-demand streaming," in *P2P*, 2010.

[2] B. Zhang, G. Kreitz, M. Isaksson, J. Ubillos, and G. Urdaneta, "Understanding user behavior in spotify," in *IEEE INFOCOM*, 2013.

[3] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, 2003.

[4] J. Reumann, "GooPS: Pub/Sub at Google." Lecture & Personal Communications at EuroSys & CANOE Summer School, 2009.

[5] "Tibco rendezvous." `http://www.tibco.com`.

[6] H. Liu, V. Ramasubramanian, and E. G. Sirer, "Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews," in *IMC*, 2005.

[7] G. Li, V. Muthusamy, and H. Jacobsen, "A distributed service-oriented architecture for business process execution," *ACM Transactions on the web*, 2010.

[8] A. Yu, P. Agarwal, and J. Yang, "Generating wide-area content-based publish/subscribe workloads," in *Network Meets Database (NetDB)*, 2009.

[9] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS*, 2010.

[10] V. Setty, M. van Steen, R. Vitenberg, and S. Voulgaris, "Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub," in *Middleware*, Springer-Verlag New York, Inc., 2012.

[11] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *SIGCOMM*, IMC, 2007.

[12] F. Rahimian, S. Girdzijauskas, A. Payberah, and S. Haridi, "Vitis: A gossip-based hybrid overlay for internet-scale publish/subscribe enabling rendezvous routing in unstructured overlay networks," in *IPDPS*, 2011.

[13] G. 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.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, 2002.

[15] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination," in *NOSSDAV*, 2001.

[16] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, and S. Tucci-Piergiovanni, "Tera: topic-based event routing for peer-to-peer architectures," in *DEBS*, 2007.

[17] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication," in *DEBS*, 2007.

[18] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Constructing scalable overlays for pub-sub with many topics," in *PODC*, 2007.

[19] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky, "Hierarchical Clustering of Message Flows in a Multicast Data Dissemination System," *Parallel and Distributed Computing and Systems (PDCS 05)*, 2005.