UNIVERSITETET I OSLO
Institutt for informatikk

Cost-Effective Resource Allocation for Deploying Pub/Sub on Cloud

436

Vinay Setty, Gunnar Kreitz, Guido Urdaneta, Roman Vitenberg, Maarten van Steen

ISBN 978-82-7368-401-1
ISSN 0806-3036

15. januar 2014

Cost-Effective Resource Allocation for Deploying Pub/Sub on Cloud

Vinay Setty[1], Roman Vitenberg[1], Gunnar Kreitz[2], Guido Urdaneta[2], and Maarten van Steen[3]

[1]University of Oslo, Norway, {vinay,romanvi}@ifi.uio.no
[2]Spotify, Stockholm, Sweden, {gkreitz,guidou}@spotify.com
[3]VU University and The Network Institute, Amsterdam, The Netherlands, steen@cs.vu.nl

***Abstract*— Publish/subscribe (pub/sub) is a popular communication paradigm in the design of large-scale distributed systems. A fundamental challenge in deploying pub/sub systems on a data center or a cloud infrastructure is efficient and cost-effective resource allocation that would allow delivery of notifications to all subscribers. In this paper, we provide answers to the following three fundamental questions: Given a pub/sub workload, (1) what is the minimum amount of resources needed to satisfy 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.**

To answer these questions, we formulate a problem coined Minimum Cost Subscriber Satisfaction (*MCSS*). We prove *MCSS* to be NP-hard and provide an efficient heuristic solution based on a combination of optimizations. We evaluate the solution experimentally using real traces from Spotify and Twitter along with a pricing model from Amazon. We show the impact of each optimization using a naive solution as the baseline. Using a variety of practical scenarios for each dataset, we also show that our solution scales well for millions of subscribers and runs fast.

I. Introduction

Publish/subscribe (pub/sub) has become a popular communication paradigm that provides a loosely coupled form of interaction among many publishing data sources and many subscribing data sinks [1]. Many applications report benefits from using this form of interaction, such as application integration [2], financial data dissemination [3], RSS feed distribution and filtering [4], business process management [5], and social interaction [6]. As a result, many industry standards have adopted pub/sub as part of their interfaces. Examples of such standards included WS Notifications, WS Eventing, OMG's Real-time Data Dissemination Service, and the Active Message Queuing Protocol.

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 paper, to the best of our knowledge we provide the first formal treatment of this subject. We consider the problem for a subclass of pub/sub systems where notifications are generated due to social interaction: following the tweets of selected users in Twitter, monitoring updates to the profiles of users' friends in Facebook, or receiving instant notifications related to favorite artists and albums in Spotify. These pub/sub applications are characterized by a significant data volume, e.g., the Spotify pub/sub service described in [6] 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 [7]. 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. 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.

We adopt a standard cost model used by Infrastructure-as-a-Service (IaaS) providers such as Amazon EC2 [8]. This cost 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*), which 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. We show that in some cases 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.

We prove *MCSS* to be NP-hard and provide an efficient heuristic solution. The solution works in two stage:

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 that we derive 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 paper include: (1) a tool to estimate the amount of resources needed to deploy pub/sub for social interaction on data centers, (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.

II. Pub/Sub model and Problem Definition

A. Background and Motivation

In this paper, 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 [6]. For example, 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 notifications generated by these systems are generally in the scale of several Terabytes per day [6], [7]. 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. For human users, having a cumulative delivery rate in a given time unit beyond a certain threshold may not be beneficial. To this end, in [9] 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. A pub/sub system designed to meet the satisfaction threshold for all subscribers can save significant amount of resources (e.g. number of servers and bandwidth consumed). Given the large-scale workload to be handled by such pub/sub engines, distributing the workload on several servers becomes inevitable. This often results in replication of publications and hence demands for more resources. As a result, designing a scalable and cost-effective pub/sub engine to be deployed on a data center or a public cloud could benefit from a tool to estimate and minimize the total costs involved.

Customers of IaaS providers can usually rent virtual machines (VMs) of a certain predefined CPU, memory and bandwidth capacities either on an hourly basis or 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 (sum of the cost of VMs plus 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 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 same incoming and outgoing bandwidth capacity.

Given that we want to minimize the sum of the cost of VMs and the cost of bandwidth, it is important to note that, in our case, there is a trade-off between the number of VMs and the amount of bandwidth that is needed to satisfy all subscribers. For example, it is possible that a load allocation that uses 3 VMs requires less bandwidth consumption than a load allocation with 2 VMs. The reason is that having a larger number of VMs increases the number of subscriber-topic allocations that satisfy all subscribers, and it is very likely that some of those extra valid allocations have lower bandwidth costs.

Note also that bandwidth consumption is affected not only by the subset of topic-subscriber pairs allocated to the VMs, but also by how topics are spread across VMs. If two VMs contain topic-subscriber pairs of the same topic, it is necessary to send the corresponding publications to both VMs. Thus, concentrating topics in as few VMs as possible also helps reduce bandwidth costs. A larger number of VMs makes it easier to concentrate each individual topic in fewer VMs.

B. Model and Notations

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). In the rest of the paper 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 the paper 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.

$\mathcal{C}_1$: 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.

bw_b : The total bandwidth consumption (incoming as well as outgoing) of virtual machine b.

$\mathcal{B}$: 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 $\mathcal{C}_1(|\mathcal{B}|) + \mathcal{C}_2\left(\sum_{b \in \mathcal{B}} bw_b\right)$.

C. Formal definition of the ***Minimum Cost Subscriber Satisfaction (*MCSS*)*** *problem :*

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

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

We now define the problem more formally defined below:

$$\begin{aligned} &\text{Minimize} && \mathcal{C}_1(|\mathcal{B}|) + \mathcal{C}_2\left(\sum_{b \in \mathcal{B}} bw_b\right) \\ &\text{Where,} && bw_b = \sum_{v \in V}\sum_{t \in T} x_{tvb} ev_t + \sum_{t \in T}\left(\max_{v \in V_t} x_{tvb}\right) ev_t \\ &\text{Subject to:} && bw_b \leq BC, \forall b \in \mathcal{B} \\ & && \sum_{v \in V} f_v = |V| \end{aligned} \tag{2}$$

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

$$f_v = \begin{cases} 1 & \text{if } \sum_{t \in T_v} \left(\max_{b \in \mathcal{B}} x_{tvb}\right) ev_t \geq \tau_v \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

In the above definition the total bandwidth bw_b consumed by a VM b is defined as 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 (2) is to avoid adding the event rate of a topic once for each pair and instead only once per VM. In Equation (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.

D. Hardness of DCSS *problem*

To establish the hardness of *DCSS* we prove that the well-known NP-Hard problem Partition Problem (*PP*) [10] can be reduced to a special case of *DCSS*. We now define the *PP* problem.

Definition II.1 (Partition Problem (*PP*) [10])**.** 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 II.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. ∎

III. Solution Approach

The Integer Program formulation of *MCSS* defined in Section II is NP-Hard according to Theorem II.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 (2). 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 (2). 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 (2). We also want to consider the trade-off between the number of VMs and total bandwidth consumption explained in Section II-A.

A. Stage 1: Selection of topic-subscriber pairs

The pseudocode of Stage 1 is presented in Alg. 2. In this stage, for each subscriber, we select a subset of topics 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 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 Alg. 1.)

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 cloud 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, and 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; otherwise, the benefit is the ratio ev_t / rem_v (line 4).

Under this heuristic, topics that contribute to satisfy v without exceeding the satisfaction threshold have all the same benefit-cost ratio and are preferred over those

Algorithm 1: Heuristic value of topic, subscriber pair (t, v) having selected $\mathcal{S}$

```
1  GetBenefitCostRatio($t, v, \tau, \mathcal{S}$)
   **Input**: $t, v, \tau, \mathcal{S}$
   **Data**: $h \leftarrow 0$ : Heuristic value
   $rem_v \leftarrow 0$: Remaining event rate needed to satisfy user $v$
2  $rem_v \leftarrow \tau_v - \sum_{\{(t',v') \in \mathcal{S} \wedge t' \in T_v.\}} ev_{t'}$
3  **if** $rem_v > 0$ **then**
4      $h \leftarrow \min\left(1, \frac{ev_t}{rem_v}\right)$
5  **return** $\frac{h}{2 \cdot ev_t}$
```

Algorithm 2: Stage 1 of solution for ***MCSS***: Greedy pair selection

```
1  **GreedySelectPairs**($T, V, ev, cost, Int, \tau, \mathcal{C}$)
   **Input**: $T, V, ev, cost, Int, \tau, \mathcal{C}$
   **Data**: $A$ : Array of size $|T|$
   **Result**: $\mathcal{S} \leftarrow \emptyset$ : Output set of (t,v) pairs
2  **foreach** $v \in V$ **do**
3      **foreach** $t \in T_v$ **do**
4          $A[t] \leftarrow$**GetBenefitCostRatio**$(t, v, \tau, \mathcal{S})$
5      **while** $\sum_{(t,v) \in \mathcal{S}} ev_t < \tau_v$ **do**
6          $t \leftarrow \text{argmax}_{\{t' \in T_v\}} A[t']$
7          $\mathcal{S} \leftarrow \mathcal{S} \cup \{(t, v)\}$
8          $A[t] \leftarrow 0$
9          **foreach** $t' \in T_v$ **do**
10             **if** $(t', v) \notin \mathcal{S}$ **then**
11                 $A[t'] \leftarrow$ **GetBenefitCostRatio**$(t', v', \tau, \mathcal{S})$
12     $A \leftarrow \emptyset$
13 **return** $\mathcal{S}$
```

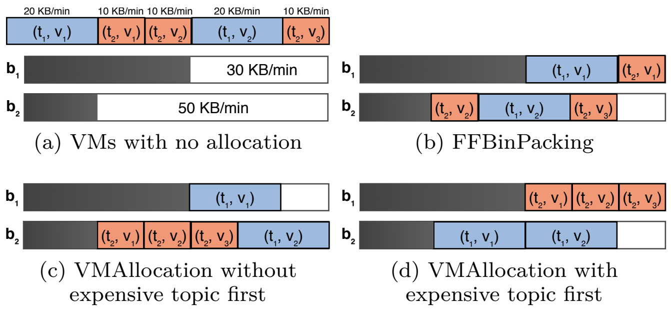
that exceed the threshold. The latter are penalized in proportion to the cost they introduce.

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 Alg. 1 and stored in an array A (from Line 2 to Line 4 of Alg. 2). 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 6 to Line 11). 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.

In order to illustrate the importance of selecting the topic-subscriber pairs in a cost-efficient manner, we compare and contrast **GreedySelectPairs** (*GSP*) against a naive solution **RandomSelectPairs** (*RSP*) (Alg. 6 in Appendix A). In the naive approach, for each subscriber v in V, enough (t, v) pairs are selected in no particular order to reach the satisfaction threshold τ_v.

B. Stage 2: Allocation of topic-subscriber pairs to VMs

In the second stage, the goal is to allocate the topic-subscriber pairs $\mathcal{S}$ selected from Stage 1 to VMs. It is interesting to note that the goal of our second sub-problem



(a) VMs with no allocation (b) FFBinPacking (c) VMAllocation without expensive topic first (d) VMAllocation with expensive topic first

Figure 1: Various VM allocation optimizations

is very similar to the well known Bin Packing problem. Hence, as a first attempt we propose the generally used job scheduling technique (e.g. used in [11], [12]) First-Fit Bin Packing **FFBinPacking** (*FFBP*) strategy as a solution for Stage 2. In Alg. 3, 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 11). If a pair (t, v) can be allocated to an existing VM it is done so with the first found VM having enough free capacity to include it (Line 2 to Line 6). If none of the existing VMs have enough free capacity to include (t, v) a new VM is deployed and added to the existing VMs $\mathcal{B}$ (Line 8 to Line 11).

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 30 KB/min and 50 KB/min (see Fig. 1a) respectively and their respective occupied capacity is shown in dark grey and their respective available capacity is left unfilled. In Fig. 1b the outcome for *FFBP* from Alg. 3 is shown. Because of the First-Fit strategy, the pairs of the same topics are split on different VMs resulting in total bandwidth consumption of 80 KB/min.

Here we make an important observation that *FFBP* has high runtime complexity of $O(|T||V||\mathcal{B}|)$, 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. Since, all pairs of a topic are considered at the same time, the splitting of pairs across different VMs will be reduced, thereby reducing the bandwidth overhead. This can be observed in Fig. 1c. With this optimization the pairs related to t_2 are on same VM, however, the pairs related to t_1 are still on different VMs. We can improve this further by selecting the topic with

Algorithm 3: First-Fit Bin Packing Algorithm for Stage 2 of ***MCSS***:

```
1  FFBinPacking($\mathcal{S}, BC$)
   Input: $\mathcal{S}, BC$
   Data: $b \leftarrow$ new VM with bandwidth capacity $BC$
   Result: $\mathcal{B} \leftarrow \emptyset$ : Set of VMs with allocated (t,v) pairs
2  foreach $(t, v) \in \mathcal{S}$ do
       // try assigning to existing VMs
3      foreach $b \in \mathcal{B}$ do
4          if $ev_t \leq BC - bw_b$ then
5              $b \leftarrow b \cup (t, v)$
6              $\mathcal{S} \leftarrow \mathcal{S} \setminus (t, v)$
       // Deploy new VM if existing VMs cannot fit
7      if $(t, v) \in \mathcal{S}$ then
8          $b \leftarrow$ new VM with bandwidth capacity $BC$
9          $\mathcal{B} \leftarrow \mathcal{B} \cup b$
10         $b \leftarrow b \cup (t, v)$
11         $\mathcal{S} \leftarrow \mathcal{S} \setminus (t, v)$
12 return $\mathcal{B}$
```

Algorithm 4: Stage 2 of ***MCSS***: Customized bin packing

```
1  CustomBinPacking($\mathcal{S}, BC$)
   Input: $\mathcal{S}, BC$
   Data: $P \leftarrow \emptyset$ : Temporary set to hold topic-subscriber pairs
         to be allocated to VMs
   $b \leftarrow$ new VM with bandwidth capacity $BC$ : VM currently
   being allocated
   Result: $\mathcal{B} \leftarrow \emptyset$ : Set of VMs with allocated (t,v) pairs
2  while $\mathcal{S} \neq \emptyset$ do
3      $t \leftarrow \text{argmax}_{\{t'\}} \sum_{(t',v) \in \mathcal{S}} ev_{t'}$
4      foreach $v \in V_t$ do // Group subscribers of topic $t$
5          if $(t, v) \in \mathcal{S}$ then
6              $P \leftarrow P \cup (t, v)$
7              $S \leftarrow S \setminus (t, v)$
8      if $(|P| + 1) \cdot ev_t > BC - bw_b$ and
       CheaperToDistribute($t, \mathcal{B}, BC, P$) is true then
9          $b \leftarrow \text{argmax}_{b' \in \mathcal{B}} \{BC - bw_{b'}\}$
10         while $P \neq \emptyset$ and $ev_t \leq BC - bw_b$ do
11             while $ev_t \leq BC - bw_b$ do
12                 $b \leftarrow b \cup (t, v)$
13                 $P \leftarrow P \setminus (t, v)$
14             $b \leftarrow \text{argmax}_{b' \in \mathcal{B}} \{BC - bw_{b'}\}$
       // For the remaining pairs deploy new VMs
15     while $P \neq \emptyset$ do
           // Deploy new VM
16         $b \leftarrow$ new VM with bandwidth capacity $BC$
17         $\mathcal{B} \leftarrow \mathcal{B} \cup b$
18         while $ev_t \leq BC - bw_b$ do
19             $b \leftarrow b \cup (t, v)$
20             $P \leftarrow P \setminus (t, v)$
21 return $\mathcal{B}$
```

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 Fig. 1d we can see that by applying these optimizations we can allocate all the pairs of each topic to the same VM, thereby reducing the overall bandwidth consumption to 50 KB/min instead of 80 KB/min using *FFBP*.

The pseudocode for the solution for Stage 2 **CustomBinPacking** (*CBP*) with the optimizations mentioned above is presented in Alg. 4. 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). If all subscribers cannot be allocated to the same VM, 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** (presented in Alg. 7 of Appendix B). When trying to allocate to already deployed VMs, we select the VM with most available capacity (Line 9 and Line 14). Each of the above optimizations give incremental improvement to our solution in practice and we will explore their incremental impact with Spotify and Twitter traces in Section IV-D.

Algorithm 5: Lower bound for ***MCSS***

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

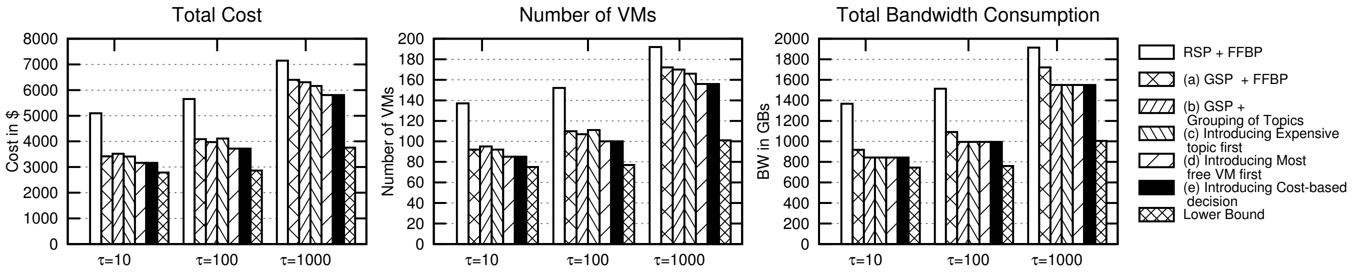
C. Lower Bound

Combining the solutions for both stages *GSP* from Alg. 2 and *CBP* from Alg. 4, gives us a complete solution for *MCSS*. While dividing the solution into two stages makes it simpler to solve, it renders our solution suboptimal. 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 IV we show that our approach works well in practice.

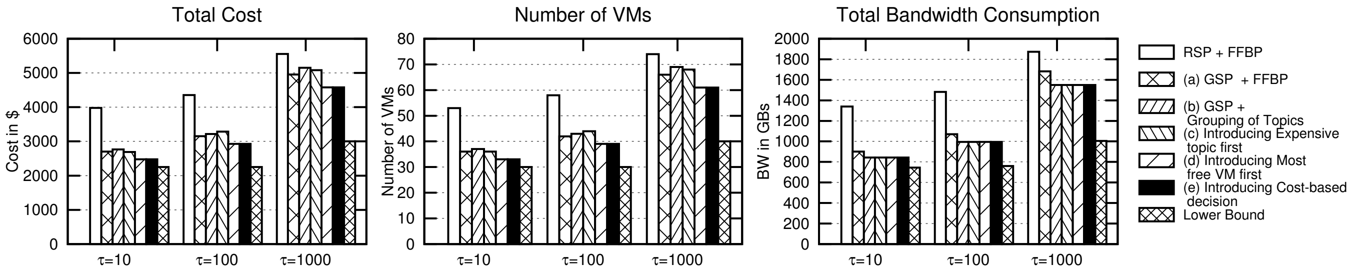
Deriving theoretical bounds on our solution is difficult because of various optimizations we introduce and we omit it from this paper. However, using Theorem A.1 in Appendix C for a given data input we can estimate a lower bound on the objective of *MCSS*. Theorem A.1 can be easily turned into an algorithm to derive the lower bound and the pseudocode is presented in Alg. 5. For each subscriber we select the bare minimum bandwidth cost required to satisfy the subscriber (Lines 2 and 3). Then we derive the lower bound on the number of VMs in Line 4 by dividing the lower bound on bandwidth consumption by bandwidth capacity BC per VM. Finally, using cost functions we derive the lower bound on total cost in Line 5. In Section Section IV-D we evaluate *GSP* with *CBP* and *RSP* with *FFBP* and compare them against the lower bound obtained using Alg. 5.

IV. Experimental Evaluation

The goal of the experimental evaluation is to study the effectiveness of the proposed solution in minimizing



(a) Cost metrics for Spotify data with $BC = 64$ mbps (c3.large VM type)



(b) Cost metrics for Spotify data with $BC = 128$ mbps (c3.xlarge VM type)

Figure 2: Impact of introducing optimizations (a) *to* (e) with Spotify traces

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.

A. *Experimental Setup*

We implemented all algorithms presented in this paper 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 specified in [8]. 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 [13]. We set c3.large and c3.xlarge with bandwidth capacities of 64 mbps and 128 mbps respectively (derived from [13]). Even though we repeated our experiments using other instance types, we omit their results due to lack of space and 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 [8].

Bandwidth consumption is measured in bytes; 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 [7], 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.

B. *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 data center at Stockholm (one of the 3 data centers). The events we collected were restricted to the music playback events from users with at least 1 follower. For more information about the Spotify trace, and its detailed analysis see [6].

Twitter Traces: We use the publicly available Twitter social graph well studied in [14]. 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. 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 process provided us with around 8 million active users and their corresponding 30 million subscribers, and around 683.5 million topic-subscriber pairs. This data trace can be downloaded from the link provided[1]. For a detailed analysis of Twitter traces refer to Appendix D.

C. Comparison of approaches for Stage 1

We first explore the impact of using *GreedySelectPairs* (*GSP*) presented in Alg. 2 with *RandomSelectPairs* (*RSP*) presented in Alg. 6 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 III 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. Fig. 2a 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 room for optimization, since a higher fraction of all topic-subscriber pairs are needed to satisfy the problem constraints. A similar pattern is observed in Fig. 2b for VM type c3.xlarge 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 Fig. 3a, 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 cost also decreases. We observe the same pattern in Fig. 3b as well with $BC = 128$ mbps. The improvements are 70%, 51.9% and 20.3% for $\tau = 10, 100, 1000$ respectively.

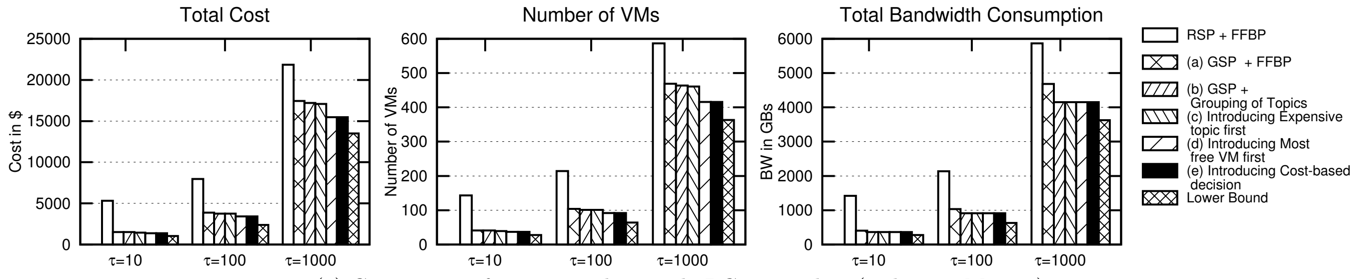
D. Comparison of approaches for Stage 2

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 III-B 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 Figs. 2 and 3 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 Alg. 5.

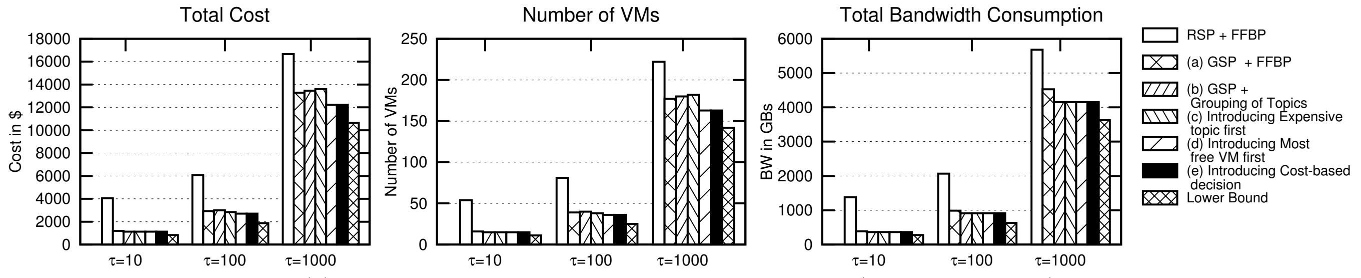
We start with *optimization (a)*, *FFBP* presented in Alg. 3. In Figs. 2a and 2b the outcome of *FFBP* when used in conjunction with *GSP* topic-subscriber pair selection technique can be seen for different values of τ and for c3.large and c3.xlarge VM types. However, as mentioned in Section III-B since *FFBP* considers the pairs to be allocated to VMs in arbitrary order and at individual pair level, there is a room for improvement. Hence, we introduced *optimization (b)*, (presented in Alg. 4 **CustomBinPacking** (*CBP*)) the *grouping of pairs* belonging to the same topic in Alg. 4 and now we 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 number of VMs and total bandwidth consumption. For example, in Fig. 2a for $\tau = 10$ and in Fig. 2b 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 II and III, 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 Figs. 3a and 3b. 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 Fig. 3b. In this case there is a decrease in bandwidth consumption of 8% which results in increase of 0.5% in VMs (1 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 III-B, the rationale behind this optimization is to give priority to expensive topics to avoid pairs belonging to 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 Fig. 2a for $\tau = 100$. However, in most cases it results in decrease in the total cost up to 2.5%. For Twitter traces, in Figs. 3a and 3b we can

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



(a) Cost metrics for Twitter data with $BC = 64$ mbps (c3.large VM type)



(b) Cost metrics for Twitter data with $BC = 128$ mbps (c3.xlarge VM type)

Figure 3: Impact of introducing optimizations *(a) to (e)* with Twitter traces

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 Alg. 4, we try to allocate all the pairs of a topic to the 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 the *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 Alg. 7. 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 costs 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 200 MB which 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.

E. 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 Fig. 4 we can see that the runtime of *GSP* for Stage 1 with Spotify traces is only at most 2 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 Fig. 5 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

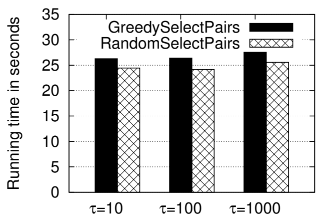


Figure 4: Stage 1 Runtime for Spotify traces

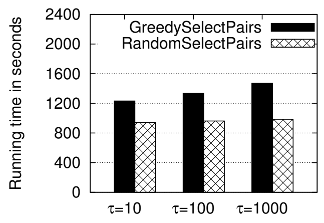


Figure 5: Stage 1 Runtime for Twitter traces

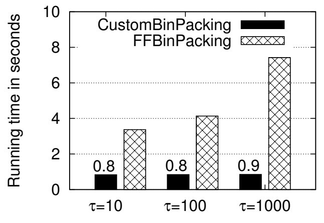


Figure 6: Stage 2 Runtime for Spotify for c3.large

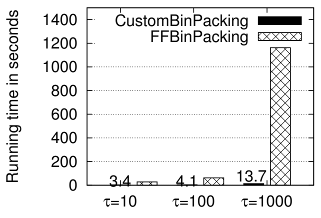


Figure 7: Stage 2 Runtime for Twitter for c3.large

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 high cost. This is a clear trade-off between quality of output and running time.

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 Alg. 4 including all other optimizations (*optimization (a)* to *(e)*) with assuming input from *GSP* readily available in main memory. From Figs. 6 and 7 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 per-topic basis to allocate them to VMs ($O(|T||\mathcal{B}|)$). 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 ($O(|T||V||\mathcal{B}|)$). It is worth noting that even though *GSP* is slower than *RSP* on its own, in combination with *CBP* the over all 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.

F. 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 moderate strength 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 [15] 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.

V. Related Work

There are many types of pub/sub systems proposed in the literature [1]. In this paper, we focus on a specific class of topic-based pub/sub which facilitates social interaction among users in systems such as Spotify pub/sub [6] and Twitter. In [9] satisfaction metrics were defined for improving the satisfaction of human subscribers in the context of pub/sub for social interaction. That work also addresses problems related to maximizing the number of satisfied subscribers under resource constraints imposed on the pub/sub engine as a single black box. However, in data center or cloud settings, pub/sub systems are scaled horizontally and hence treating the engine as a black box limits the effectiveness of resource provisioning. In this paper we address this limitation by considering a multi-server setup typical of a data center. We also focus on a different problem of estimating the monetary costs assuming realistic pricing models from public IaaS providers such as Amazon EC2.

There are several papers addressing resource provisioning in the cloud to minimize monetary costs [11], [12], [16]. 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 most optimizations introduced in this paper, such as grouping of topic-subscriber pairs by topics and selecting topics with maximum event rate first, infeasible.

To the best of our knowledge there exist no works addressing the problem of cost-effective resource provisioning tailored for topic-based pub/sub systems. One relevant area of research is stream processing in the cloud [17], [18]. However, there are only a few works in this category that specifically consider resource provisioning [15], [19]. In [15], 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 VMs 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 VMs and bandwidth consumption. While in [19], the authors propose a demonstration of cost estimation for streaming queries, they do not aim at minimizing this cost. In contrast to our work, the idea is specific to the domain of streaming queries. Finally, in both [19] and [15], there is no concept of subscriber satisfaction metric, which is essential in our problem.

There exist works that provide a formalization for the general problem of resource provisioning in the cloud, with emphasis on theoretical analysis. In [20] 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 previsioning 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.

VI. Conclusions and Future Work

In this paper, 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 data center 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.

References

[1] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM CSUR*, 2003.

[2] J. Reumann, “GooPS: Pub/Sub at Google,” Oslo, Norway, Lecture & Personal Communications at EuroSys & CANOE Summer School, 2009.

[3] “Tibco rendezvous,” http://www.tibco.com.

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

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

[6] V. Setty, G. Kreitz, R. Vitenberg, M. van Steen, G. Urdaneta, and S. Gimåker, “The hidden pub/sub of spotify,” in *DEBS*, 2013.

[7] R. Krikorian, “Twitter by the numers,” http://www.slideshare.net/raffikrikorian/twitter-by-the-numbers.

[8] “Amazon EC2 pricing,” https://aws.amazon.com/ec2/pricing.

[9] V. Setty, G. Kreitz, G. Urdaneta, R. Vitenberg, and M. van Steen, “Maximizing the number of satisfied subscribers in Pub/Sub systems under capacity constraints,” in *INFOCOM (To appear)*, 2014, PDF can be downloaded from http://tidal-news.org/2014/setty-2014.pdf.

[10] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.

[11] S. Genaud and J. Gossa, “Cost-wait trade-offs in client-side resource provisioning with elastic clouds,” in *CLOUD*, 2011.

[12] D. Villegas, A. Antoniou, S. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for Infrastructure-as-a-Service clouds," in *CCGRID*, 2012.

[13] "Amazon EC2 bandwidth limits," http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-ec2-config.html.

[14] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW*, 2010.

[15] J. Cerviño, E. Kalyvianaki, J. Salvachúa, and P. Pietzuch, "Adaptive provisioning of stream processing systems in the cloud," *SMDB*, 2012.

[16] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and R. Bianchini, "Dejavu: accelerating resource allocation in virtualized environments," in *ASPLOS*, 2012.

[17] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE TPDS*, 2012.

[18] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, J. Pineau, M. Pasin, E. Rivière, and S. Weigert, "Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe," in *DEBS*, 2013.

[19] T. Heinze, P. Meyer, Z. Jerzak, and C. Fetzer, "Demo: Measuring and estimating monetary cost for cloud-based data stream processing," in *DEBS*, 2013.

[20] M. Sindelar, R. Sitaraman, and P. Shenoy, "Sharing-aware algorithms for virtual machine colocation," in *SPAA*, 2011.

Appendix

A. Pseudocode to select random pairs

The pseudocode to select arbitrary topic-subscriber pairs is provided in Alg. 6. The **RandomSelectPairs** (*RSP*) algorithm, selects the first obtained pairs for each subscriber v until the satisfaction threshold τ_v is met (between Lines 2 and 4). The final set of pairs is returned in a set $\mathcal{S}$ in Line 5.

Algorithm 6: Naive alternative for Stage 1 of solution for ***MCSS***: Random pair selection

```
1 RandomSelectPairs($T, V, ev, cost, Int, \tau, \mathcal{C}$)
  Input: $T, V, ev, cost, Int, \tau, \mathcal{C}$
  Result: $\mathcal{S} \leftarrow \emptyset$ : Output set of (t,v) pairs
2 foreach $v \in V$ do
3     while $\sum_{(t,v):(t,v)\notin\mathcal{S}\wedge t\notin T_v} < \tau_v$ do
4         $\mathcal{S} \leftarrow \{(t,v) : (t,v) \notin \mathcal{S} \wedge t \in T_v\}$
5 return $\mathcal{S}$
```

B. Pseudocode to decide if it is cheaper to distribute among existing VMs

The goal of **CheaperToDistribute** in Alg. 7 is to decide if distributing the pairs of the current topic in question to already deployed VMs or allocate them to a new VM. This function is called when the pairs of the current topic in question cannot be allocated to the current VM. In Alg. 7, we first compute the estimated total cost when deploying new VMs and allocating to them (between Lines 2 and 4). Then compute the cost of allocating to a VM with maximum available capacity (Line 5) until there are no more pairs left in P (between Lines 6 and 6) 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 and needing new VMs to be deployed. The cost of the extra VMs needed and corresponding bandwidth consumption is computed between Lines 16 and 18. Finally Alg. 7 returns **true** if allocating to existing VMs is cheaper and returns **false** otherwise (in Lines 19 and 20).

C. Lower Bound Theorem

Theorem A.1. *Given an instance* MCSS$(T, V, ev, Int, \tau, BC, \mathcal{C}_1, \mathcal{C}_2)$*, for any solution* $\mathcal{B}$ *it holds that:*

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

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

Proof:

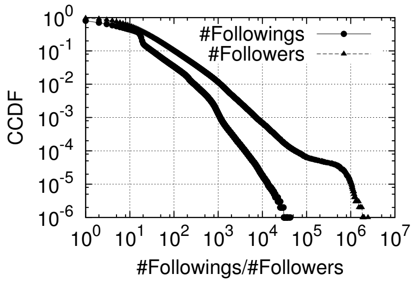


Figure 8: CCDF of #Followers and #Followings

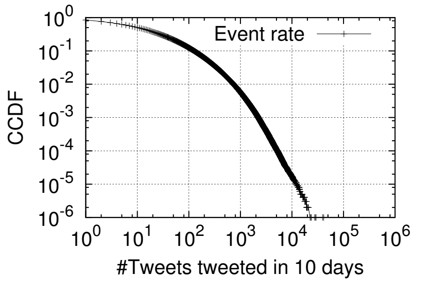


Figure 9: CCDF of event rate from 10 day traces

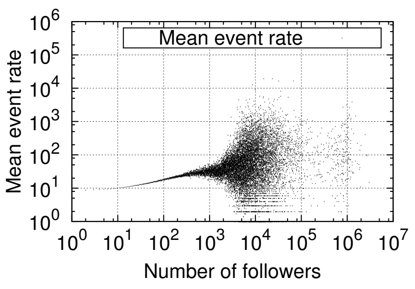


Figure 10: #followers, event rate dependency

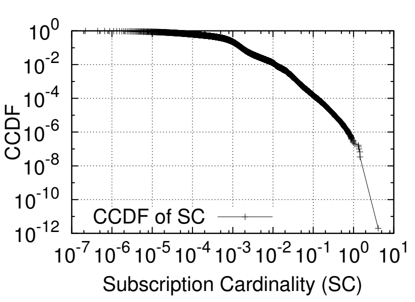


Figure 11: CCDF of *SC*

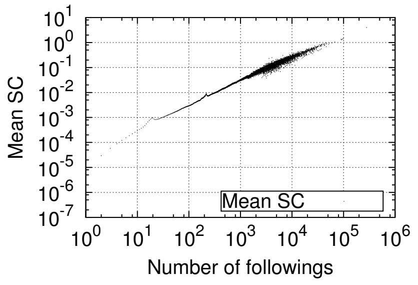


Figure 12: #followings, *SC* dependency

Algorithm 7: Computes the cost of distributing current topic to existing VMs

```
1  CheaperToDistribute($t, \mathcal{B}, BC, P$)
   Input: $t, \mathcal{B}, BC, P$
   Data: $curbw \leftarrow \sum_{b\in\mathcal{B}} bw_b$: Current bandwidth
         consumption, $curvms \leftarrow |\mathcal{B}|$: Number of VMs
         currently in use, $extravms \leftarrow 0$: Extra VMs needed,
         $TV \leftarrow \emptyset$: Temporary set of VMs,
         $newvmsbw \leftarrow 0, newvms \leftarrow 0$
   Result: $distribute \leftarrow$ false
   // Estimate the cost of deploying on new VMs
2  if $P \neq \emptyset$ then
3      $newvms \leftarrow \lceil(|P| \cdot ev_t)/BC\rceil$
4      $newvmsbw \leftarrow curbw + (|P| + newvms) \cdot ev_t$
5  $b \leftarrow \mathrm{argmax}_{b'\in\mathcal{B}}\{BC - bw_{b'}\}$
6  while $P \neq \emptyset$ and $\mathcal{B} \setminus TV \neq \emptyset$ do
7      $newbw \leftarrow 0$
8      while $newbw \leq BC - bw_b$ do
9          $newbw \leftarrow newbw + ev_t$
10         if $\forall V_t : (t, v) \notin b$ then
11             $newbw \leftarrow newbw + ev_t$
12         $P \leftarrow P \setminus (t, v)$
13     $curbw \leftarrow curbw + newbw$
14     $TV \leftarrow TV \cup b$
15     $b \leftarrow \mathrm{argmax}_{b'\in\mathcal{B}\setminus TV}\{BC - bw_{b'}\}$
16 if $P \neq \emptyset$ then
17     $extravms \leftarrow \lceil(|P| \cdot ev_t)/BC\rceil$
18     $curbw \leftarrow curbw + (|P| + extravms) \cdot ev_t$
19 if $C_1(curvms + extravms) + C_2(curbw) <$
   $C_1(newvms + curvms) + C_2(newbw)$ then
20     $distribute \leftarrow$ true
21 return $distribute$
```

Given a data set, satisfaction metrics and bandwidth capacity constraints, we can derive a lower bound on the total cost that can be minimized to. Intuitively, for each subscriber, we just choose the bare minimum cost to satisfy the subscriber. Then, use the obtained lower bound on the bandwidth consumption to derive lower bound on number of VMs. We elaborate this idea below:

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 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.

■

D. Analysis of Twitter traces

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) [2] of the number of followings in Fig. 8. The distinctive anomalies observed in [14] 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 Fig. 8 and there is a visible glitch at 10^5, as seen in original data in [14]. In our sample, there are around 4000 users having more than 10^4 followers and 66 users beyond 1 million followers. By manually verifying, 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 our sample in a 10-day period. 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 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 Fig. 10. For each unique number of followers in the X axis we show the corresponding mean tweet rate (event rate) in 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 Subscription Cardinality (SC) of a subscriber v defined in [6]:

$$SC_v = \frac{\sum_{t \in T_v} ev_t}{\sum_{t \in T} ev_t} \cdot 100$$

In Fig. 11 we show the CCDF of SC. 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 dependency between the number of followings a user has and the corresponding mean SC in Fig. 12. It is clear that SC grows linearly with the number of subscribers. However, there are noticeable glitches at 20 and 2000 followings, due to the same reason for the glitches in the #Followings CCDF.

[2]CCDF is probability of a random variable X > x